COMPUTING SCIENCE

Using Metaobject Protocols to Satisfy Non-Functional Requirements

R.J. Stroud and Z. Wu

TECHNICAL REPORT SERIES

No. 533 October, 1995
Using Metaobject Protocols to Satisfy Non-Functional Requirements

R.J. Stroud and Z. Wu

Abstract

Traditional approaches to the realisation of non-functional requirements such as dependability and distribution transparency are usually not transparent to application programmers and thus increase the complexity of the system. Using a different approach to implement a particular non-functional requirement involves application programmers in making changes to the system. Achieving a clean separation between the implementation of functional and non-functional requirements would reduce the complexity of the final system and thus enhance its maintainability and flexibility. In this paper, we present a metaobject protocol approach to satisfying non-functional requirements that uses meta level programming techniques to make a clean separation between functional and non-functional components, and thus makes it easier to revise the implementation of a particular non-functional requirement in order to meet new demands.
STROUD, Robert James

Using Metaobject Protocols to Satisfy Non-Functional Requirements


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 533)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
WU, Zhixue

Abstract

Traditional approaches to the realisation of non-functional requirements such as dependability and distribution transparency are usually not transparent to application programmers and thus increase the complexity of the system. Using a different approach to implement a particular non-functional requirement involves application programmers in making changes to the system. Achieving a clean separation between the implementation of functional and non-functional requirements would reduce the complexity of the final system and thus enhance its maintainability and flexibility. In this paper, we present a metaobject protocol approach to satisfying non-functional requirements that uses meta level programming techniques to make a clean separation between functional and non-functional components, and thus makes it easier to revise the implementation of a particular non-functional requirement in order to meet new demands.

About the author

Dr. R.J. Stroud joined the Department of Computing Science as a Research Associate in 1983 and has been a lecturer since 1991. His research interests include the design of object-oriented programming languages and the concept of reflection and meta-object protocols.

Dr. Z. Wu is a Research Fellow in the Department of Computing Science supported by the University Research Committee of the University of Newcastle upon Tyne. His research interests are in distributed systems focusing on supporting non-functional requirements by using the object-oriented technology and the concept of reflection and meta-object protocols.

Suggested keywords

FLEXIBILITY METAOBJECT PROTOCOLS OBJECT-ORIENTED PROGRAMMING
REFLECTION

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 U.D.C. 681.322.06
Using Metaobject Protocols to Satisfy Non-Functional Requirements

R. J. Stroud and Z. Wu
Department of Computing Science,
University of Newcastle upon Tyne
{R.J.Stroud, Zhixue.Wu}@newcastle.ac.uk

ABSTRACT

Traditional approaches to the realisation of non-functional requirements such as dependability and distribution transparency are usually not transparent to application programmers and thus increase the complexity of the system. Using a different approach to implement a particular non-functional requirement involves application programmers in making changes to the system. Achieving a clean separation between the implementation of functional and non-functional requirements would reduce the complexity of the final system and thus enhance its maintainability and flexibility. In this paper, we present a metaobject protocol approach to satisfying non-functional requirements that uses meta level programming techniques to make a clean separation between functional and non-functional components, and thus makes it easier to revise the implementation of a particular non-functional requirement in order to meet new demands.

1. Introduction

In order to be useful, a software system must meet the requirements against which it was constructed. Furthermore, it is not sufficient for a system to simply perform some function, it must perform that function “well” in some sense. Thus, it is convenient to group the requirements on a system into two categories, namely functional requirements which are primarily concerned with the purpose of a system (i.e. what it does) and non-functional requirements which are more concerned with its fitness for purpose (i.e. how well it does it). Typically non-functional requirements include concerns such as dependability (e.g. reliability, security and safety), concurrency transparency and distribution transparency. Such requirements are general purpose and therefore any software technique for addressing them in an application-independent fashion should be reusable across a wide range of possible problem domains.
To be effective, the realisation of non-functional requirements in a reusable fashion should be transparent as far as the application programmer is concerned. Ideally, it should be possible to design a system that concentrates on meeting a set of functional requirements, and then add components that address additional non-functional requirements without disturbing the structure of the original system. Achieving such a clean separation would reduce the complexity of the final system and thus enhance its maintainability. In practice, however, if the underlying system does not support a particular non-functional requirement, traditional approaches usually cannot address this issue in a way that is transparent to application programmers. The code that must be added to the application program increases its complexity and makes it harder to reason about its correctness. Furthermore, using a different approach to implement a particular non-functional requirement requires making changes to application programs.

In this paper, we suggest using a metaobject protocol approach to satisfying non-functional requirements. Meta level programming allows a clean separation to be made between the functional and non-functional components of a system, and thus makes it easier to revise the implementation of a particular non-functional requirement in order to meet new demands. We first discuss the problems with the traditional approaches to this problem in the next section. The metaobject protocol approach to satisfying non-functional requirements is presented in section 3 together with some examples to show its feasibility. The paper is concluded with a brief discussion of some problems we need to solve in our future research.

2. Traditional approaches for dealing with non-functional requirements

In this section, we outline some of the traditional approaches to dealing with non-functional requirements and discuss their problems.

2.1 System-based approach

Many common non-functional requirements such as persistent data storage, data sharing and distributed programming are well recognised and supported by most operating systems. More specialised non-functional requirements such as persistent programming, concurrent programming and fault tolerance are also supported in some modern operating systems. For example, Delta-4 [Powell 91] provides several replication protocols for supporting transparent error processing. If the underlying system can support non-functional requirements transparently, application programmers need only be concerned about the implementation of functional requirements. Another advantage of this approach is that it usually provides better performance than approaches that build support for non-functional requirements on the top of the underlying system. However, because the implementation is encapsulated inside the underlying system, it is very difficult for application programmers to reason about the behaviour of the system and to make any changes to the system behaviour. Thus, this approach makes it hard to customise the system to meet particular
requirements from applications.

2.2 Language extension approach

Object-oriented programming techniques are based on the premise that software should be modelled as a set of objects that interact through well-defined interfaces. This approach to design reduces complexity by allowing a software system to be decomposed into a set of cleanly separated components. Furthermore, if these components are chosen carefully, they may be used in the construction of several related applications. Reusable components may be used to extend the semantics of an object-oriented programming language by providing a wider range of building blocks with which the programmer can construct applications. With a pure object-oriented programming language, there should be an almost seamless join between user-defined objects and system-defined objects. It should be possible to develop reusable components that address particular non-functional requirements independently and then combine them recursively.

There are essentially three ways in which an object-oriented programming system can be extended: by adding semantics to the programming language, by extending the run-time support mechanisms, or by adding further object definitions to the object library. Only the last of these extensions can be achieved entirely within the language, without requiring modifications to the compiler or interpreter. Thus, extensibility is a broader concept than simple reuse, and mechanisms for achieving extensibility include tools like pre-processors and interpreters as well as the more conventional notion of reusable objects.

Each of these techniques for extending a language has been used successfully in the past to adapt a programming system for a particular purpose. For example, Arjuna [Shrivastava et al 91] provides persistence and atomicity by inheritance, but uses a pre-processor to implement distribution transparency. PC++ [Wu et al 95] supports an implicit approach to implementing user-defined atomic data types via inheritance and pre-processing. Avalon [Detlefs et al 88] builds an object-oriented system on top of the distributed transaction facilities provided by Camelot [Spector et al 88]. Argus provides linguistic support for atomic data types [Weihl 88]. SOS adds support for persistency and migration to C++ objects with the aid of the compiler and a run-time object management system [Shapiro et al 89].

However, although previous work in this area has demonstrated that it is indeed possible to build reusable components that address these kinds of non-functional requirements, such work has also highlighted a number of inadequacies in the support provided by many conventional object-oriented programming languages for this kind of reuse. In particular, the realisation of non-functional requirements often requires the application programmer to program in a stylised way that obscures the functionality of the application, or requires a specialised implementation of a programming language that is hard to be customise in order to meet new demands from applications.
For example, implementing an atomic data type in Arjuna requires the programmer to invoke particular methods to set and release locks, thus mixing the implementation of functional requirements with the implementation of non-functional requirements. Switching the concurrency control mechanism used by an atomic object from a pessimistic method to an optimistic method requires re-implementing the object, even though the implementation of the concurrency control method can be provided by the system. The problem is more serious in Argus since programmers are required to provide their own concurrency control method for their atomic objects. The implementation of functional and non-functional requirements is totally intertwined in this case and it is thus very hard to reason about the correctness of an object or to change the concurrency control method it uses. PC++ allows programmers to implement atomic data types as if for a sequential and reliable environment and then adds the code necessary to support atomicity in a concurrent, unreliable environment, thus making a clear separation between the functional and non-functional aspects of a system from the programmer’s point of view. However, PC++ uses a pre-processing method to collect the information needed for making synchronisation decisions and this makes it hard for the system to support multiple concurrency control methods [Stroud and Wu 95].

2.3 Summary

In summary, traditional approaches to addressing non-functional requirements cannot make a clean separation between functional and non-functional programming, either at the system level or at the application level, thus making it hard to customise the implementations of non-functional requirements to meet the demands of particular applications. In the next section we will show how using metaobject protocols to satisfy non-functional requirements can help to solve this problem.

3. The Metaobject Protocol Approach

In this section, we will show how metaobject protocols can be used to satisfy non-functional requirements. At first the basic idea will be described and then two examples will be given to demonstrate the feasibility of this approach.

3.1 The basic idea

An important requirement for the implementation of non-functional requirements is flexibility. With the dramatic emergence of new application areas, system software needs to meet ever increasing user demands and expectations. Because applications have quite different and even contrary requirements, it is impossible for a particular solution to a non-functional requirement to match all the demands from different application areas. Thus the implementation of the non-functional components must be made flexible and customisable dynamically. To achieve this goal, a clean separation between the implementations of functional and non-functional requirements must
be made so that application programmers can concentrate on meeting functional requirements. Components that address particular non-functional requirements should be implemented independently from functional requirements, and it should be possible to add such non-functional components to a system without disturbing the structure of the original system.

The combination of reflection and object-oriented programming in the form of a metaobject protocol [Kiczales et al 91] provides precisely the mechanism we need to produce flexible and reusable non-functional components. This is because the metaobject protocol technique enables the internal behaviour of an object to be manipulated at a meta level. Thus, the behaviour of an object can be changed by defining new kinds of metaobject. This feature makes it possible to implement non-functional requirements in a way that is transparent to the application program whilst still remaining flexible enough to meet the demands of a wide range of applications.

Using a metaobject protocol approach, functional requirements are satisfied by application objects implemented by application programmers. Non-functional requirements are satisfied by metaobjects implemented by system programmers. Different solutions to a non-functional requirement are realised in different metaobjects. The actual behaviour of an application object is decided not only by its implementation but also by the metaobject it is associated with. This association can be thought of in terms of a binding between an object and a metaobject. The behaviour of an application object can be changed without re-implementing that object but by simply changing its binding to a metaobject. This is illustrated in Figure 1.

![Figure 1. Binding between an object and a metaobject](image)

The metaobject protocol approach makes it much easier for a system to meet new requirements from applications because the behaviour of the application can now be observed and manipulated at an abstract level by a metaobject and thus changed more flexibly.

### 3.2 Atomic data types

A typical non-functional requirement in a database system is to maintain data consistency in a
concurrent and unreliable environment. An object-oriented approach to achieving this goal is to use *atomic data types* [Weihl 85], abstract data types that also have synchronisation and recovery properties. Instances of atomic data types, called atomic objects, are responsible for ensuring their own data consistency. Thus, an implementation of an atomic data type must contain both functional and non-functional parts – in addition to the data structures and operations that implement the functionality of the object, extra data structures and operations need to be defined to satisfy atomicity requirements, namely concurrency control and failure recovery.

Since the strategy that an atomic object uses to maintain data consistency may vary between different environments, a flexible system should allow an atomic object to change its concurrency control and recovery method dynamically without changing the code which implements its basic functionality. We have used a metaobject protocol to achieve this goal. Thus, atomicity requirements are satisfied by a metaobject that controls the invocation of operations on the basic application object. We have implemented two metaobjects which use different concurrency control methods to satisfy atomicity requirements: a pessimistic protocol and an optimistic protocol [Stroud and Wu 95]. The pessimistic metaobject uses the well-known 2-phase locking (2PL) protocol and the optimistic metaobject uses the dual-level validation (DLV) method [Wu et al 95]. In order to use a particular concurrency control method to maintain data consistency, an atomic object need only associate itself with the appropriate metaobject. No changes to the implementation of the application object are needed in order to change its concurrency control policy.

Figure 2 shows the implementation of a *credit* operation for an *Account* atomic object. The functional part of the operation is implemented as usual. Two methods for satisfying the non-functional requirements are implemented in two metaobjects: the *Meta_2pl* metaobject for the 2PL method, and the *Meta_dlv* metaobject for the DLV method. The actual behaviour of the *credit* operation depends on which metaobject is bound to the *Account* object. If the application object is bound to the *Meta_2pl* metaobject, then the *credit* operation will apply for a lock and take a snapshot of the object state before performing any access to the object. If the application object is bound to the *Meta_dlv* metaobject instead, the *credit* operation will record the operation name and arguments before performing any action, and record the result of the operation before returning it to the caller of the operation.

If other concurrency control methods are required by particular applications, they can also be supported by simply implementing them as new metaobjects. They can then be used by any atomic object without needing to make changes to the original implementation of the system.
3.3 Replication

Fault tolerance is another typical non-functional requirement. Replication is a fundamental method for achieving fault tolerance in a distributed system. By replicating information in several nodes of a system, the failure of some of the nodes that hold that information will not result in a loss of data. It may also be necessary to maintain backup processes that will take over from a failed process and complete time-critical computations or computations that have acquired mutual exclusion on shared resources.

There are different techniques for implementing replication. We will use the passive replication method as our example to show how to achieve replication transparency by using metaobject protocols [Fabre et al 95]. With passive replication, only one of the replicas, called the primary replica, processes the input messages and produces output messages. Other replicas, called standby replicas, do not process input messages, neither do they produce output messages. Their internal states are however regularly updated by means of checkpoints from the primary replica. When the primary server crashes, one of the backup replicas takes over the responsibility of providing continued service to the client and a new backup replica is started.
Figure 3 shows the process of executing an operation on a replicated object when the passive replication method is used. Whenever an operation of a replicated object is called, the metaobject associated with it intercepts the call. Then the metaobject executes the appropriate method at the base level object, and sends the resultant state of the base level object to all the standby replicas before returning the result to the caller. When a standby replica receives a checkpoint message from the primary replica, it updates the state of its local base level object and sends back an acknowledgement.

![Diagram of replication process](image)

**Figure 3.** The execution of an operation on a replicated object

Other replication methods such as active and semi-active replications can be implemented in a similar way. Any of these methods can then be used to replicate an object according to the particular requirements of the application.

4. Summary

Using metaobject protocols, a clean separation between functional and non-functional programming can be made. Functional requirements are satisfied by objects implemented by application programmers, whilst non-functional requirements are satisfied by metaobjects implemented by system programmers. The actual behaviour of an object can be changed without re-implementing the object but rather by changing the binding of the object to a metaobject or by interacting with its metaobject directly. This approach makes it much easier for a system to meet new requirements from applications because its behaviour is now more flexible.

Reflection has been used in many application areas: flexible programming [Kiczales et al 91], concurrent programming [Matsuoka et al 91], distributed systems [Chiba and Masuda 93], and soft real-time applications [Honda and Tokoro 92]. The Apertos operating system [Yokote 92] uses the framework of object/metaobject separation for supporting very large scale, open,
distributed systems featuring mobile computing.

We are interested in using reflection as a general approach to addressing non-functional requirements such as fault tolerance and distribution transparency [Stroud 93], and have implemented some non-functional requirements, such as atomic data types [Stroud and Wu 95], persistent systems [Stroud and Wu 94] and fault tolerant applications [Fabre et al 95] by using the metaobject approach. Although there are still some problems such as dynamic binding between objects and metaobjects that need to be resolved, our experiments make us believe that using metaobject protocols is a feasible approach for satisfying non-functional requirements. We believe that as reflective implementation technologies become better understood, reflection will just come to be viewed as another form of indirection mechanism that is no more expensive to use than inheritance or delegation but can be used to solve a different class of problems.

Acknowledgements

Zhixue Wu is a Research Fellow supported by the Research Committee of the University of Newcastle upon Tyne. This work has also been partially funded by the CEC as part of the PDCS 2 Basic Research Project no. 6232. We are grateful to the anonymous referee for his or her comments on an earlier version of this paper.

References


