Design Patterns in Enterprise

Steve MacDonald
Department of Computing Science
University of Alberta
Edmonton, Alberta
CANADA T6G 2H1
stevem@cs.ualberta.ca

Abstract

The Enterprise parallel programming system allows programmers to create, compile, execute, and debug parallel applications that execute over a network of workstations. The run-time system, which is responsible for the correct execution of user programs, was redesigned and re-implemented using object-oriented technology. This paper details the object-oriented components of the design where the *Composite*, *Adapter*, and *Strategy* design patterns were applied.

1. Introduction

Design patterns are a popular tool among object-oriented designers and programmers to promote re-use of common elements in the design and implementation of programs [4]. Design patterns provide a general, application-independent solution to a particular class of problems. These patterns can be used as components in a much larger program. They allow designs to be described in high-level terms that can be understood and communicated easily, provided a common terminology can be found.

Frameworks are one possible result of design patterns. A framework is a pattern (or composition of patterns) with an existing infrastructure that implements a partial solution to a particular problem. This partial solution is augmented with application-specific code by the user. An example of a framework is the Model-View-Controller (MVC) model for graphical user interfaces in Smalltalk. The existing infrastructure provides the graphical components and a set of classes that can be subclassed to tailor the system for a particular application. While the patterns in MVC can be used in other contexts, this framework can be used only for user interfaces.

This paper presents some of the design patterns that were used in the design and implementation of the new, object-oriented version of the Enterprise run-time system [8]. Enterprise is a parallel programming system that allows users to create programs from existing parallel design patterns (such as the pipeline and master/slave patterns) and execute them on a network of workstations. The run-time system is responsible for the correct execution of user programs, including communication, synchronization, and process management. This system was re-written using an object-oriented design and implementation to create a solid basis for further research. For details on the actual design of the new system, interested readers are directed to [7].

The principal points of this paper detail our experiences with the use of design patterns. The principal lessons and contributions are:

- that design patterns can be used to create a flexible system,
- that specific design patterns can be generalized by incorporating them into more complicated class structures. This ability allows more complicated programs to take advantage of design patterns,
- that the Enterprise programming system can also be viewed as a method for building frameworks out of a set of existing design patterns, and
- that design patterns have associated costs that should be examined by groups wanting to use this technology.

This paper is organized as follows. Section 2 briefly introduces the Enterprise parallel programming system; the information here provides the context for the different patterns. Section 3 details the interesting design patterns

The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., Dept. 2G/894, 1150 Eglinton Avenue East, North York, Ontario, Canada. M3C 1H7.

found in Enterprise, and Section 4 contains some discussion about our experiences with patterns. Section 5 presents conclusions.

2. The Enterprise Parallel Programming System

The Enterprise parallel programming system provides facilities for users to develop, compile, execute, debug, and animate parallel programs. These programs are executed over a network of workstations using some form of process communication.

The greatest strength of Enterprise is its two programming models that were developed to simplify the task of writing parallel programs. They allow users to concentrate on their application by handling all of the details of parallel execution. First, there is the *programming model* to specify the semantics of the application code. Second, there is the *meta-programming model* to specify the parallel aspects of the program using the analogy of a business organization, or enterprise.

The programming model is based on the *futures* model of parallel programming [5]. Certain procedure calls are marked to be executed remotely with the results of the call tagged as futures. When a remote procedure call is made, the arguments are packed into a message and sent to the processor responsible for executing that procedure. The caller continues to execute concurrently with the remote procedure until a future is accessed. At this time, the caller must *resolve* the future, blocking if the results are not available. The following code fragment illustrates the programming model.

```
result = parallelCall(x);
/* Continue executing while
parallelCall is executed */
/* Block if result is
unavailable. */
y = result + 1 ;
```

The strength of this model is that it preserves the semantics of sequential C. The user can either write normal code, or incorporate existing code into a parallel application with little difficulty. The only changes the user may wish to make is to separate parallel calls from accesses to the results, to increase the available concurrency. These changes are purely for performance reasons; the program will work correctly without them, but not as fast.

The parallel aspects of the program are specified using the meta-programming model. This model relates the parallel aspects of the program to the different assets of a business enterprise. The assets are divided into two types: *singular assets* and *composite assets*. A singular asset is akin to a process and is responsible for executing the procedure that shares its name. A composite asset is a container that describes how the different assets inside of it are related, specifying the parallelization technique the user wishes to use for the program. Composite assets provide the most common parallelization techniques such as the master/slave and the pipeline, which are design patterns used to solve common problems in parallel programs. The user combines the different asset types (7 in all) to create an *asset graph* that shows the overall parallel structure of the application. Interested readers are directed to [6, 8] for more details.

The Enterprise parallel programming system itself consists of three components: the graphical user interface, the precompiler, and the run-time system.

The graphical user interface provides an integrated environment for program development, execution, and debugging. For the purposes of this paper, its main feature is that it allows the user to construct the asset graph.

The precompiler takes the user's source code and the asset graph as inputs and produces a parallel version of the source code as output. The precompiler parallelizes the code by inserting calls to the run-time library into the application code. In addition, the precompiler creates *stubs code* to pack and unpack the arguments to an asset call. This is the responsibility of the precompiler because it has access to all the necessary type information, which cannot be derived by the run-time system. To demonstrate, the simplified post-processed code fragment from above becomes:

```
_ENT_Send_parallelCall(x,
   &result);
/* Check if results are
available. */
_ENT_Wait(&result);
x = result + 1 ;
```

where _ENT_Send_parallelCall() is a stubs routine generated by the precompiler and _ENT_Wait() resolves the supplied future.

The run-time system is responsible for the correct execution of an Enterprise application. This responsibility includes:

- implementing the precompiler-inserted function calls,
- launching the application (launching is done in a distributed fashion, with each Enterprise process using the asset graph to launch the assets it may call),
- processing messages, including all the necessary synchronization and run-time consistency checks,
- managing futures,
- gathering run-time information for support tools such as the animation and controlled replay components, and
- shutting down the program.

3. Design Patterns in Enterprise

Design patterns were used in several places in Enterprise, both in the programming system and its implementation. The meta-programming model provides several parallel design patterns to the user to create the asset graph. The implementation of the run-time system uses design patterns in several places to make the system more flexible and maintainable. Most of the patterns in this section can be found in Gamma *et al*. [4], although there are many other references for design patterns.

3.1 Design Patterns in the Meta-Programming Model

The first place that design patterns appeared was in the meta-programming model of Enterprise itself. Specifically, the composite assets can also be stated as design patterns. Enterprise currently has two composite assets; the *department asset* and the *line asset*.

The line asset is a standard pipeline. In Enterprise, it consists of a *receptionist asset* and a set of heterogeneous assets with a specific order. The receptionist asset is defined to be the entry point of all composite assets. Each asset in the line accepts requests from the asset above it, works on its input data, and gives its results to assets below it. This asset has an obvious correspondence to a pipeline pattern.

The department asset, our version of the master/slave pattern, is similar to a *Reactor* pattern [9] with implicit handle registration. The department consists of a receptionist and a set of heterogeneous assets. The receptionist accepts requests and forwards them to the appropriate component asset. This behaviour corresponds to the multiplexing of events by different entities in the reactor, where a centralized dispatcher invokes different handle functions for incoming events.

We can also look at the meta-programming model as a method of creating frameworks by composing the existing design patterns. If we define the problem being solved as the execution of a program over a network of workstations, then the run-time system implements the infrastructure, invoking application-specific asset code in a way that is analogous to the subclassing in the MVC framework. The design patterns making up the framework are given by the asset graph, and the precompiler links everything together.

We can also argue that Enterprise is also a pattern builder or a language to a limited extent. If we view an asset graph as a pattern that is composed from the existing parallel patterns, then users can build new compositions or patterns from the Enterprise user interface. One can adapt these new patterns to different applications by inserting the proper code for each asset. This argument falls apart, though, because these new patterns are not supported using the same mechanisms provided for the original Enterprise assets. Users should be able to create new design patterns that can be incorporated into the meta-programming model. If we restrict the new patterns to be compositions of existing ones, then defining them and their behaviour is relatively easy. Defining arbitrary patterns is much more difficult, requiring that one specify the flow of control between the different component assets and the flow of control between the exit points of the new asset and other assets. This behavioural information must be given in some way that can be used by the run-time system and the precompiler to properly manage the new pattern and to verify the structure of applications using it. This is clearly a difficult problem, and represents work being done by others in design pattern research [10].

3.2 Design Patterns in the Run-time System

The second place that design patterns appeared in Enterprise is in the implementation of the run-time system. The patterns used here are generalizations of existing patterns in the literature. This section details their use and other issues that arose during implementation. Since C++ was the language used, both the design and implementation take advantage of multiple inheritance.

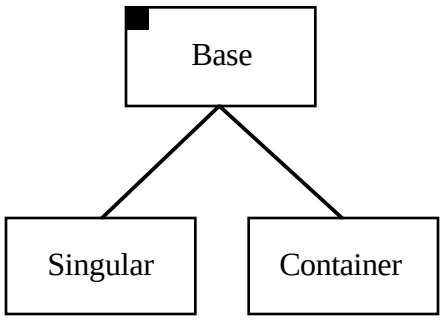

Figure 1: The basic composite pattern

3.2.1 The Internal Asset Graph

Part of the run-time system involves creating an internal representation of the asset graph, which is used to derive relationships between the different assets in the program for consistency checks. Since Enterprise has two kinds of assets, singular and composite, the internal asset graph fits into the *Composite* design pattern.

The basic Composite pattern is shown in Figure 1. A black square in the upper left corner of a class indicates it is an abstract superclass. The pattern consists of three classes: *Base*, *Singular*, and *Container*. The Base class is usually purely virtual, providing the interface for its subclasses. The Singular class represents a single instance of the class, implementing the methods of its superclass to perform those operations on its instance data. The Container class holds a collection of Base instances as components in its instance data. This class implements the methods from its superclass by invoking the method on each member in the collection. This pattern can be used to compose an arbitrary tree of composites by having instances of the Container class as components, which would be traversed in a depth-first manner if the above implementation of methods is used.

The set of classes responsible for representing the asset graph is given in Figure 2. This class diagram can be related to the basic composite pattern using the Asset class as the Base class, the CodableAsset class as the Singular class, and the CompositeAsset as the Container class. The abstract class ReplicableAsset contains all instance data for assets that can be replicated. In an application, there can be many of these classes that provide additional characteristics not included in the basic Composite pattern. The subclasses of the classes that make up the pattern are used to precisely specify the characteristics of each asset type beyond simply being singular or composite. These classes are necessary because a department behaves differently than a line even though both are composite assets. We believe that this represents a generalization of the Composite pattern, where the concrete classes (both Singular and Container) can be refined through subclassing. Furthermore, the introduction of other abstract classes can give the concrete classes even more characteristics beyond the simple composite and singular designations. This extension allows for a richer set of concrete classes, and provides an elegant method of specifying the characteristics of each class. The resulting application is more maintainable and extendible as the characteristics are encapsulated, and new characteristics can be added easily.

3.2.2 Parsing the Graph File

After the user constructs the asset graph, the user interface generates a textual representation called the *graph file*. When an Enterprise application starts, the first process of the application reads the graph file from disk, creates its internal graph, and creates the processes for the application in a distributed manner, with each process creating the processes it can invoke. During the startup procedure, each process is sent a series of messages containing the contents of the graph file and its identity within the application. Because the same executable is used for each process, the run-time system must be able to parse the graph file from either disk or a memory stream. Ideally, we should be able to re-use the methods for parsing the graph by just specifying a different stream. This is not possible, though, because memory streams and

file streams do not share the same interface or have identical behaviour. For instance, memory streams do not have open or close methods, since these operations do not apply. To re-use the parsing methods, the run-time system composes an abstract superclass with the *Adapter* pattern. The abstract superclass allows a concrete instance to be accessed via its superclass, with the actual instance of the class left unspecified until run-time. The Adapter pattern allows a class to respond to a different interface. The composition is shown in the class diagram in Figure 3.

The abstract class GraphStream provides the interface for manipulating the graph file. This interface consists of a limited number of normal file operations along with some higher-level methods, such as bypassing white space and reading tokens. The MemoryStream class contains an instance of *istrstream*, a C++ class that manipulates memory streams. The FileStream class privately inherits from the *fstream* class to manipulate the graph file on disk. Private inheritance allows a subclass to access the data and methods of a superclass without exporting them. Thus, the FileStream class can use methods in the fstream class in its implementation, but fstream methods cannot be invoked on an instance of FileStream

The abstract class GraphStream gives us the ability to use a concrete class without specifying the exact one until run-time, as opposed to hardcoding the actual stream type in the code. Hardcoding the class would require two sets of operations, one for each concrete class. Instead, the proper type can be instantiated at run-time and the abstract class can be used to reference the real stream. Now, one set of methods can be written for the GraphStream class so that they will work with either files or memory streams.

The Adapter pattern was required because memory and file streams do not behave identically. The principal difference between them was in creating a new stream, because memory streams do not need to be opened or closed. Since the purpose of the GraphStream class was to use both kinds of streams invisibly, it was necessary to create semantics for these methods. In this case, the open method was defined so that FileStreams open the supplied file name and MemoryStreams create a memory stream using the string argument. The close method closes the file for FileStreams and deleted the instance of istrstream in MemoryStreams.

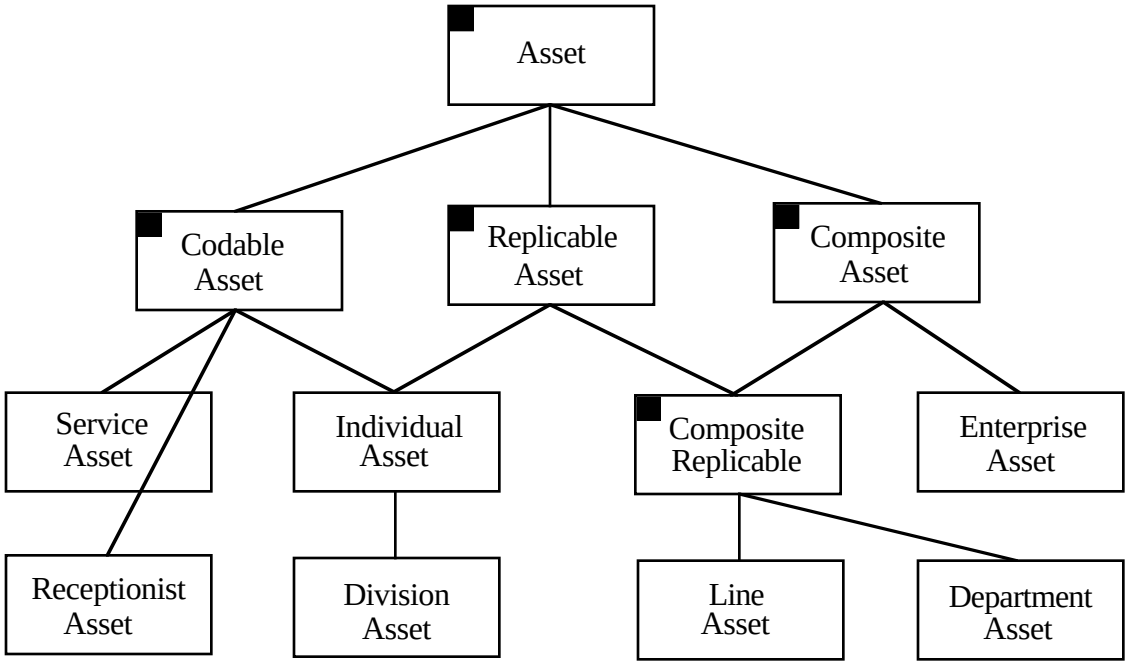

Figure 2: The asset graph class hierarchy

.

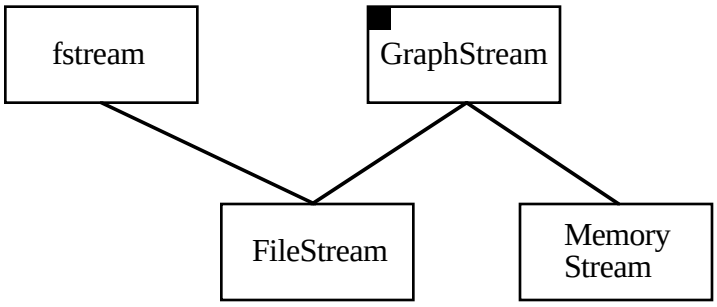

Figure 3: The stream inheritance hierarchy

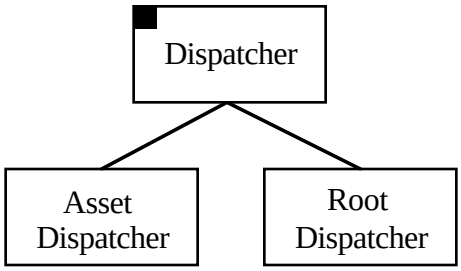

Figure 4: The dispatcher class hierarchy

3.2.3 Run-time Behaviours

The next pattern discussed in this paper relates to the modeling of the behaviour of a process in the run-time system. This function is accomplished through a *dispatcher/behaviour* pair, which is an implementation of the *Strategy* design pattern. Like the asset graph classes, this implementation is not straight from the design pattern, but also includes other classes and refinements specific to this application.

The dispatcher hierarchy is given in Figure 4. The dispatcher is responsible for receiving messages and forwarding them to an instance of a behaviour class for processing. The behaviour instance is an instance variable in the dispatcher class. In Enterprise, there are two different kinds of dispatchers: Asset and Root. The difference between them is that the Asset dispatcher contains a buffer for holding messages that cannot be processed immediately. An example of such a message is an incoming request message that arrives while the process is busy executing another. Since Enterprise processes are single-threaded, this message must be stored for later processing. The Root dispatcher is always able to process messages as they arrive.

The behaviour class hierarchy is shown in Figure 5. These classes are used to model the response of a process to a message. The DispatcherBehaviour class provides a common, abstract interface to the dispatcher. This interface is broken down into further method calls which are implemented by the subclasses until the desired behaviour is achieved. Each of the concrete classes represents a different behaviour that can be exhibited by a process in an Enterprise application. These are:

- SeqRootBehaviour: This behaviour executes an Enterprise program sequentially. It sets a flag to make all asset calls local. The first asset is called sequentially.
- ParRootBehaviour: This behaviour is the root of an Enterprise program that is being executed in parallel. It sends the command line arguments to the first asset in the program and awaits the reply. It is also responsible for processing logging messages, which

contain the run-time information used by the animation and replay components.

- ManagerBehaviour: This behaviour represents an external manager process, which is responsible for managing the workers of a replicated asset. In the new version of Enterprise, it is also possible for this manager to be collapsed into another process. To prevent multiple implementations of this functionality, the actual management of worker processes for a particular asset is the duty of the *Process* class. This class also provides information on the processes responsible for executing an asset. An instance of this class is part of each asset.
- SingleAssetBehaviour: This behaviour represents an instance of an unreplicated asset.
- WorkerBehaviour: This behaviour represents a worker in a replicated asset. The difference between this behaviour and SingleAssetBehaviour is the need for reply messages: this behaviour must always generate a reply message to indicate the availability of the process to its manager, whereas the other only needs to reply when there is data to be returned.
- DivisionWorker: This behaviour represents a worker in a division, a recursive divide-and-conquer asset type. Division workers must do additional work to properly determine if the process is a leaf of the asset, which determines if further recursive calls are made in parallel or sequentially.

Looking at the dispatcher and behaviour pair, an alternative solution would be to merge the two classes into one, letting the behaviour both read and process incoming messages. However, the Strategy pattern has the benefit that a more dynamic system can be created by allowing the behaviour of a process to change during execution. The dispatcher can delete its current behaviour instance and instantiate another, which will change the way subsequent messages are processed. From the perspective of the dispatcher, this change is invisible. If the dispatcher and behaviour classes were merged, this operation would be much more difficult.

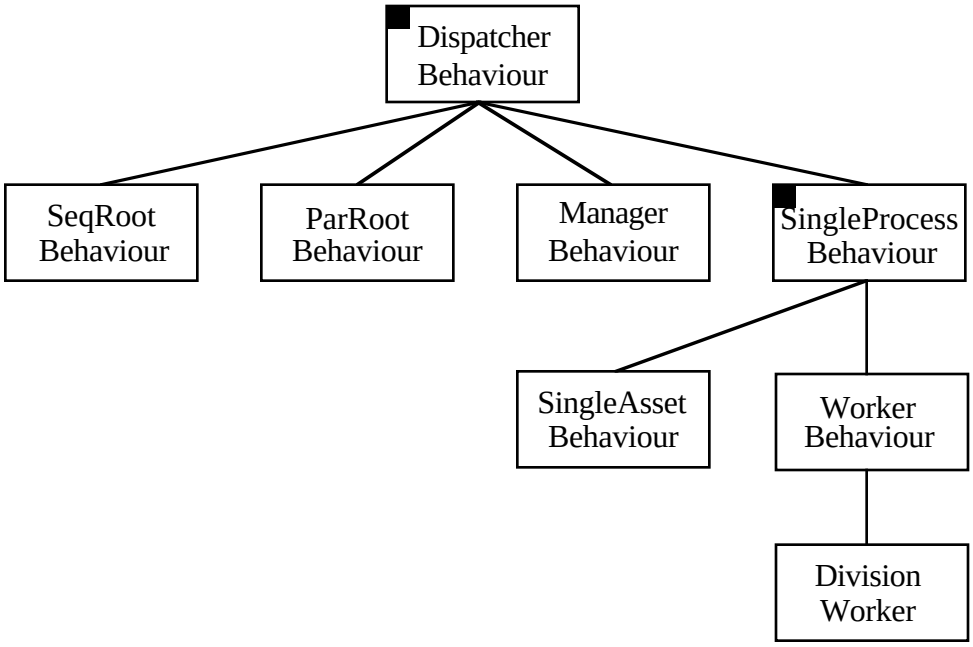

Figure 5: The behavior class hierarchy

The ability to change the behaviour of a process is very useful in a system with dynamically replicated processes. In Enterprise, this leads to the idea of *promotion* and *demotion*. When an application starts execution, we can create a single process for each replicated asset, which uses the SingleAsset behaviour. At some point, the process can determine that more replicas are necessary and promote itself to a manager, which then spawns additional workers and manages them. Later, if the workers become unnecessary and are removed, the process can demote itself to a SingleAsset and handle requests itself.

3.2.4 Communications Manager

The final pattern we will examine is another application of the Adapter class, which is used to provide the ability to incorporate different libraries for a specific portion of the implementation. In this case, we are dealing with the *communications manager*, the class in the Enterprise run-time system that encapsulates the communications system. We wish to provide a minimal set of communication primitives to the run-time system in such a way that different libraries can be used to implement them without having to significantly change any other part of the code. The minimal set of primitives also prevents special or unusual features of any given library from becoming an integral part of the system.

Currently, Enterprise supports a PVM [11] message-passing communication manager and a Treadmarks [1] distributed shared memory communication manager. It is important to note that the Treadmarks implementation was restricted to providing communication between processes; the run-time system does not assume shared memory. Eventually, this implementation will multiplex between distributed shared memory and local shared memory on a multi-processor machine where available.

This organization of the communication manager is such that only one implementation of the class is available at any given time. The exception, of course, is the multiplexing of distributed and local shared memory mentioned previously. Although there are systems that allow for multiple communication systems to be active [3], we do not believe this will be a limitation. The main difficulty in allowing this is that, in our experience, each system uses a conflicting set of signals and timers, which prevents interoperability. A similar problem occurs for the initialization of the different systems. Finally, this solution would require that all libraries be linked into the application, which makes the executables large.

4. Discussion

One of the more interesting items about this work was the author did not get involved with design patterns until after this project had been completed. Only then were the above patterns identified. This fact may be further evidence of the validity of the work being done in the pattern community. It certainly suggests that these solutions are not unique to this application. In fact, these ideas have been communicated to several other designers for use in their applications.

The patterns identified seem to be useful tools, particularly for the maintenance of this system. Once the patterns are explained, all of the involved researchers can quickly explain their ideas for expanding on the system. This communication is critical as the run-time system was intended as a basis for further research, and has been supporting an active group of six researchers for about eight months. During this time, there have been many changes to the original code to include support for parallel I/O, shared memory, and a better recursive asset type, all using the Strategy pattern to expand on or create behaviours for these tasks.

The last topic is the effect of patterns on the implementation of object-oriented programs. Typically, patterns provide additional abstraction in order to elegantly handle design problems. In return, they may provide common terminology and serve as a starting point for new designs. However, as with all abstraction, these benefits come at a price. First, for those not familiar with design patterns, there is a learning curve while the associated literature is reviewed. Such users would also suffer if simply given code that makes heavy use of design patterns, as the abstractions will not be clear to them. This problem can lead to the *yo-yo effect* [2], where designers must continually traverse the class structures to find the methods and information they need to follow the program. Finally, most of this abstraction is implemented in object-oriented languages using abstract classes, virtual methods, and multiple inheritance. Although hybrid languages such as C++ are designed to

minimize the impact of virtual methods, the overhead is still present.

It seems that design patterns, like all software and design tools, must be evaluated before they are used. Within the proper environment, they can be a useful tool for design and communication, but there are some costs that should be considered. The patterns provide more flexibility than a simple implementation, but typically require more abstract classes that can increase the cost of method invocations. Communication of a design can be done at a higher level, but only if all parties involved use common and consistent terminology.

5. Conclusions

This paper presented some of the design patterns used in the new implementation of the Enterprise run-time system. Some of these patterns were already present in the meta-programming model of Enterprise, and were also being used to solve some of the design and implementation problems of the run-time system. These problems include the internal representation of the asset graph, the parsing of the graph file, the behaviour of a process during execution, and the communication manager. Each of these examples required a generalization of the original pattern, which even further demonstrates their adaptability.

This paper also demonstrated how Enterprise can be viewed as a way of building frameworks out of existing design patterns, with the run-time system providing the infrastructure for designing different applications.

Finally, we presented some discussion about the use of design patterns. Although they are a valuable tool for designers and programmers, their use incurs some costs that must be addressed.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and by Ph.D. fellowships from the Centre for Advanced Studies, IBM Canada. The author would also like to thank the referees and the editor of this paper for their helpful suggestions.

About the Author

Steve MacDonald is in the Ph.D. program at the University of Alberta. His main interest is in parallel object-oriented programming systems. He received a B.Math. from the University of Waterloo and a M.Sc. from the University of Alberta. He also wishes to acknowledge his Ph.D. fellowship from the Centre for Advanced Studies, IBM Canada.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, pp. 18-28, 1996.

[2] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1991.

[3] I. Foster, C. Kesselman, and S. Tuecke, "Nexus: Runtime Support for Task-parallel Programming Languages," Argonne National Laboratory Technical Report ANL/MCS-TM-205, February 1995.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Object-Oriented Software Architecture*: Addison-Wesley, 1995.

[5] R. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 501-538, 1985.

[6] P. Iglinski, S. MacDonald, C. Morrow, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron, and D. Woloschuk, "Enterprise User's Manual Version 2.4," University of Alberta Technical Report TR 95-02, January 1995.

[7] S. MacDonald, "An Object-Oriented Run-time System for Parallel Programming," M.Sc. thesis, Department of Computing Science,. University of Alberta, 1996.

[8] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons, "The Enterprise Model for Developing Distributed Applications," *IEEE Parallel and Distributed Technology*, vol. 1, pp. 85-96, 1993.

[9] D. Schmidt. "Reactor: An Object Behavioural Pattern for Event Demultiplexing and Event Handler Dispatching." Appeared in *Pattern Languages of Program Design.* Edited by J. Coplien and D. Schmidt, Addison Wesley, 1995.

[10] S. Sui, M. De Simone, D. Goswami, and A. Singh, "Design Patterns for Parallel Programming," To appear in Parallel and Distributed Processing Techniques and Applications (PDPTA'96), August 1996.
[11] V. Sunderam, “PVM: A Framework for Parallel Distributed Computing,” *Concurrency: Practice and Experience*, vol. 2, pp. 315-339, 1990.