Patterns in Flexible Server Application Frameworks

James C. Hu
jxh@entera.com
Entera, Inc.
Fremont, CA, USA

Christopher D. Gill
cdgill@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO, USA

Abstract

This article describes a collection of recurring patterns in the context of flexible server application frameworks. These patterns are organized into three categories, corresponding to distinct architectural levels of scale: foundation, scaffolding, and framework. In addition to identifying well-known patterns at each of these levels, this article describes three new patterns: the Concrete Bridge pattern at the foundation level, and the Library and Strategized Concurrency patterns at the framework level.

1 Introduction

The context of this paper comes from experience designing, developing, refining, refactoring, and reimplementing an object-oriented framework in C++ called the JAWS Adaptive Web System (JAWS). JAWS is an application framework for Web servers, designed as a research tool to study sytematically how different design decisions impact the performance of a server. Achieving this required that the framework efficiently factor out differences in threading strategies (*e.g.*, thread pool versus thread-per-request), event completion strategies (*e.g.*, synchronous versus asynchronous), protocol specifications (*e.g.*, HTTP, RTSP, FTP, *etc.*), and algorithms (*e.g.*, cache replacement algorithms, hash algorithms, lookup algorithms, *etc.*). JAWS factors out these points of variation by employing classic design patterns [1], discovering new design patterns, and leveraging components available in the ACE [2] framework, upon which JAWS is built [3]. A full discussion of the JAWS framework is beyond the scope of this paper, but a detailed description can be found in [4].

The evolution of the JAWS framework has led to some interesting insights into what makes a framework usable, accessible, scalable, and maintainable. A usable framework is *generative*, *i.e.*, one that guides the developer to create a complete application. An accessible framework allows the developer to use the framework by only studying the documented interfaces. A scalable framework results in applications that can cope with growing demands on the application, both internal (*e.g.*, new features),

and external (*e.g.*, higher than expected server usage). A maintainable framework results in applications that are easy to extend, modify, and debug by software engineers other than the ones who originally wrote the framework.

A framework will only be considered usable when application developers are convinced that the framework provides, or can support through extension, all the tools needed to complete their tasks. To meet potentially unexpected demands, the framework must provide good abstractions that are open to extension, so that new tools can be added easily and seamlessly, either by the framework developer or the application developer.

Making the framework accessible requires dedication on the part of the framework developer to document critical framework components. The importance of this cannot be over-emphasized. If application developers feel they are being short-changed on documentation, the framework will never be adopted for use.

Scalability is an important feature that directly impacts the longevity of a framework. If the framework is insufficiently extensible, it will likely be replaced by a rewrite, or even worse, the framework principle itself might be entirely rejected by the software team. Similarly, if the application that results from the framework is not able to utilize available hardware resources (such as additional memory or CPUs) to improve performance, then the framework will likely be labeled the cause of the problem and rejected.

Maintainability is perhaps more a goal than a design criterion. However, its importance should not be underestimated simply because less ideal software code bases exist. This is especially true in the arena of open source software projects, which rely upon the input and feedback from a community of software developers. Successful development requires that the software itself be resilient enough to withstand that kind of scrutiny.

This paper explores those patterns that have proven most useful in the development and evolution of JAWS into a useful, accessible, scalable, and maintainable framework. Section 2 previews the patterns that are most important in the JAWS framework, and describes categories of architectural scale at which these patterns are applied. Sections 3-5 provide detailed explanations of those patterns not previously described elsewhere in the patterns literature. Section 3 describes the Concrete Bridge pattern, a variation of the Bridge pattern that enables applications to bind an implementation at compile-time, in cases where the application does not benefit from dynamic binding. Section 4 explores the Library pattern, a useful design pattern for maintaining a collection of objects. Section 5 describes the Strategized Concurrency pattern. This pattern enables applications to choose between different concurrency models at run-time.

2 Pattern Language

Here we describe those patterns that appear in the JAWS framework. The first group of patterns, labeled *foundation* patterns, are small-scale software microarchitectures, many of which have already been captured and fully described in the patterns literature. The second group, labeled *scaffolding* patterns, consists of higher level patterns that are fundamental to systems and application programming. The third group, la-

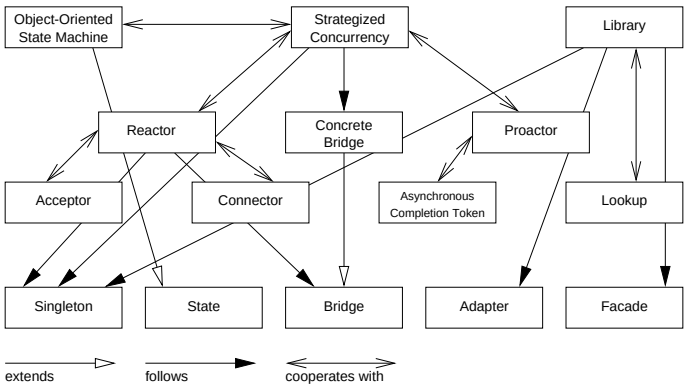

Figure 1: Patterns in the JAWS framework

beled *framework* patterns, contains the highest level patterns in the JAWS framework. Figure 1 illustrates how these patterns are related to each other.

2.1 Foundation Patterns

These patterns are used extensively in the JAWS framework. These patterns are singled out as those that were explicitly incorporated into the framework by reading the literature for solutions to specific problems. Other patterns may exist that are yet to be recognized.

- Singleton[1]
- Adapter[1]
- Bridge[1]
- State[1]
- Facade[1]
- Concrete Bridge (Section 3)

2.2 Scaffolding Patterns

These patterns describe fundamental components of the JAWS framework. They build upon the Foundation patterns, and used in conjunction with them to form the framework.

- Acceptor[5]
- Connector[5]
- Reactor[6]

- Proactor[7]
- Asynchronous Completion Token[8]
- Lookup[9]

2.3 Framework Patterns

These patterns appear at the highest level of abstraction in the JAWS framework. The interfaces to these components are available to the application developer to utilize framework resources and to specialize application behavior. JAWS framework resources include a file caching subsystem, network I/O, and concurrency mechanisms. The JAWS framework runtime system calls into application specific code using a state-machine model.

- Library (Section 4)
- Strategized Concurrency (Section 5)
- Object-Oriented State Machine [10]

3 Concrete Bridge Pattern

Concrete Bridge allows applications to bind a known implementation to an interface at compile time, while preserving a separation of concerns and minimizing coupling of the interface and implementation.

3.1 Context

The Bridge [1] pattern elegantly decouples an abstraction from its implementation. It uses inheritance polymorphism to achieve this decoupling. However, a consequence of inheritance polymorphism is that it also introduces overhead due to dynamic method binding. Furthermore, it may be desirable to know at compile time the concrete type that provides the abstration's implementation. The Generic Bridge [11] pattern addresses the latter design force, but still relies on inheritance polymorphism between the generic class template and the interface class referenced by the client.

3.2 Forces

This pattern resolves two opposing forces. The first is the need for loose coupling between interfaces and implementations to reduce compile-time dependencies, improve extensibility, and enhance data hiding. The second is the need for an efficient means to associate implementations with interfaces at compile time, and also to reduce unnecessary overhead.

3.3 Problem

Decoupling an interface from its implementation is desirable, but inheritance polymorphism is not always an acceptable solution.

Some applications, particularly in the real-time domain, cannot tolerate unnecessary overhead. In such cases, compile-time binding of the implementation to an interface removes virtual function lookup overhead. Furthermore, removing inheritance polymorphism may also improve testability of such applications, as the implementation of a particular interface is then fixed before run-time.

3.4 Solution

Eliminate inheritance polymorphism from the relationships connecting the client class, through the interface class, to the concrete implementation class. Parameterize the interface class with the type of its implementation class, and instantiate the interface with a concrete implementation type at compile time.

The parameterized interface class in Concrete Bridge takes on the same role as the Abstraction participant in the Bridge pattern. However, rather than delegating to an abstract base class, the parameterized interface class delegates to a generic type parameter.

In both cases, a concrete implementation type is bound before any implementation methods are invoked, but that binding likely occurs later in the case of inheritance polymorphism. Also in both cases, the interface class requires certain methods of the implementation class, which must be available to ensure type safety. For inheritance polymorphism, an implementation base class is used to ensure that only types that implement these methods can be bound to the interface class. The implementation base class may provide definitions for these methods, or leave them abstract, thus ensuring that any concrete derived type will provide or inherit a definition for each of the methods.

In the case of type parameterization, the interface class requirements must be met directly by the concrete implementation type. Taken together, the syntax and semantics that must be supported by the concrete implementation type constitute a *concept* in the terminology of generic programming [12]. Any concrete type that is suitable as an implementation class is then said to *model* the concept specified by the interface class.

Concrete Bridge offers flexibility during the software development process, permitting development to proceed unhindered by changes to underlying implementations, and allowing developers to experiment with alternative implementations. Inheritance polymorphism can even be reintroduced below the level of the bound implementation class, if desired, at least during the experimental phase of development. This latter technique is similar to the consequences of the Generic Template Method pattern described in [13].

Once a particular implementation is chosen for the finished application, the parameterized class can then be instantiated with that concrete implementation. This binds an implementation to the interface at compile time, avoiding unnecessary overhead for run-time binding.

3.5 Structure

Figure 2 shows the basic structure of the Concrete Bridge design pattern. The type-parameterized `Abstraction` class template provides the interface to the client. It is parameterized with an `Abstract Implementor` type.

The parameterized type must satisfy the syntactic and semantic requirements of the `Abstraction` class template. For example, the `Abstraction` class template's `Operation ()` method calls the `OperationImp ()` on its concrete implementation class instance. The `OperationImp ()` method must 1) be present, 2) provide the necessary type signature to allow correct invoaction, and 3) have the necessary semantics to correctly implement the interface class template's `Operation ()` method.

Taken together, the syntactic and semantic requirements of the interface class template on its parameterized implementation type *specify* a generic programming *concept*, as shown in Figure 2. Because UML lacks symbols to convey these ideas from generic programming, we show the concept as a note with a constraint titled `Implementor Concept`.

We also provide appropriate labels conveying the corresponding generic programming terminology for each relationship between the parameterized type, the concept, and the concrete implementation classes. Specifically, any concrete class that can be *bound* to the class template must be a *model* of the concept *specified* by the requirements on the parameterized type.

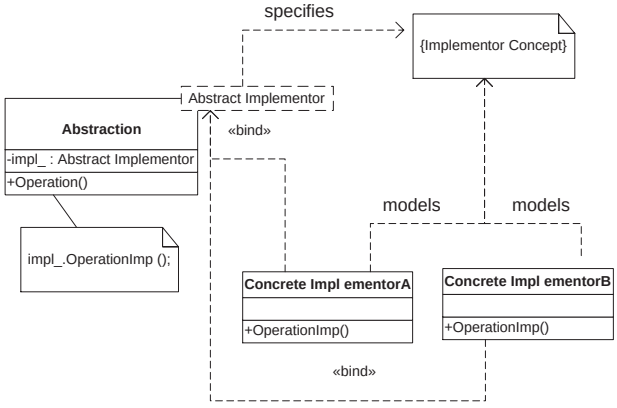

Figure 2: Structure of the Concrete Bridge design pattern

3.6 Consequences

Just as the Bridge pattern, the Concrete Bridge pattern decouples interface and implementation, improves extensibility, and enhances hiding of implementation details from clients. In addition, the Concrete Bridge allows a single implementation to be bound to an interface at compile time. If the application requires the ability to configure the

implementation type of an abstraction at run-time, or has need for the abstraction to change its implementation polymorphically at run-time, then there may be no advantage to using the Concrete Bridge pattern over the Bridge pattern.

Some languages, such as Java, do not support parameterized types. However, support for parameterized types is a highly useful feature of modern languages, and significant research efforts are underway to extend languages such as Java to support parameterized types [14].

3.7 Known Uses

The ACE ORB (TAO [15]) provides a reconfigurable Scheduling Service [16] implementation, that uses Concrete Bridge to bind a particular scheduling strategy at compile time. Concrete classes implement well-known real-time scheduling strategies such as RMS and EDF [17]. The application binds a particular strategy to the scheduling service implementation at compile time, avoiding virtual method overhead while preserving flexibility for the application developer to experiment with different scheduling strategies.

The JAWS framework also employs Concrete Bridge, in its implementation of the Strategized Concurrency pattern. The relationship between these two patterns in JAWS is described in greater detail in Section 5.

3.8 Related Patterns

Bridge [1], Generic Bridge [11], External Polymorphsim [18].

4 Library

The Library pattern decouples the creation of a new object from caching an existing instance of the object. Thus, whether or not the object is being retrieved from cache is transparent to the requester of the object. The Library pattern provides access to a collection of named objects of the same type, each created with different initialization parameters. Clients retrieve objects from the Library, and insert objects into the Library, by name. If an object is inserted with the same name as an existing object, the Library ensures that future requests for that name receive the newer object, that existing references to the old object are not affected by the insertion of a newer version, and that when there are no more references to the old version, any system resources associated with it are properly recycled.

4.1 Context

An application requires a searchable repository of versioned objects of the same type.

4.2 Forces

An application may face the following pairs of opposing forces:

- The need to access a large number of objects, *vs.* the need to minimize resource utilization through sharing.
- The need for a sharing mechanism, *vs.* the need to hide the details of the sharing mechanism from the users of the objects.
- The need to populate the sharing mechanism, *vs.* the need to alter the bindings between names and their associated objects dynamically.

4.3 Problem

An application must be able to access objects of similar type but with different initialization parameters. Fundamentally, this problem encompases the creation of these objects, the population of a searchable container, and the management of requests for the object.

A naive way to build a Library is to create a number of Singletons and add them to a Lookup.

```
class Moby_Dick
{
public:

  static Book * instance (void)
  {
     if (! book_)
       {
         book_ = new Book ("Moby Dick");
         Lookup::instance ()->register ("Moby Dick", book_);
       }
     return book_;
  }

};
```

Now, when clients want to access ”Moby Dick”, they will go to the Lookup and find it. This is unsatisfactory for several reasons. First, the Lookup interface is explicitly exposed to the client. The client must understand what items are being stored in the Lookup, and how to use the Lookup interface to get them. Second, the process of placing books into the Lookup is entirely static in this approach. New objects cannot be added seamlessly. Third, if a new version of ”Moby Dick” appears, clients must explicitly manipulate the Lookup to make the new version visible. Similar problems arise when initializing a Library using an iteration loop over a list of titles.

4.4 Solution

Use the Library pattern to hide the details of using the underlying Lookup and to manipulate the underlying object. This pattern borrows ideas from Singleton (create on first use), Proxy (surrogate object) and Facade (unified interface).

The Library pattern hides the Lookup interface from the user, and also permits dynamic search and insertion.

```
template <class Library>
class Book_Proxy
{
public:

  Book_Proxy (const char *name)
  {
    this->book_ = Library::instance ()->find (name);
    if (this->book_)
      this->book_->acquire ();
  }

  ~Book_Proxy (void)
  {
    this->book_->release ();
  }

  // ...
  // Delegate methods into underlying book_,
  // mimicing the public interface of Book.

private:

  Book_Type *book_;

};


template <class Title, class Book, class Lookup>
class Library
{
public:

  Book * find (const Title & name)
  {
    Book *book = this->lookup_.find (name);

    if (book && ! book->current ())
      {
        book = 0;
      }

    if (! book)
      {
        Book *old_book;

        book = new Book (name);
        book->acquire ();

        this->lookup_.rebind (name, book, &old_book);

        if (old_book)
          old_book->release ();
      }

    return book;
  }
```

```
private:

  Lookup lookup_;

};
```

4.5 Structure

Figure 3 Shows the basic structure of the Library design pattern.

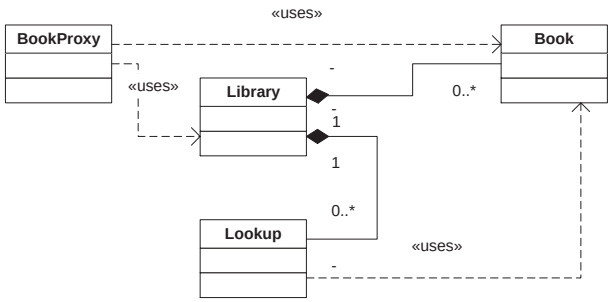

Figure 3: Structure of the Library design pattern

4.6 Consequences

Cache coherency problems may appear when using the pattern in a distributed context. That is, if multiple distributed processes are applying Library on a common collection and one process detects an update, then it would be desirable to communicate this information to the other processes. The implementation described here leaves it to each process to perform its own discovery, which may be inefficient in a distributed context.

4.7 Known Uses

A file server may wish to avoid re-opening files that are commonly requested. It may wish to cache the contents of these files so that future requests will be able to be served from memory rather than being re-read from the disk. This is how JAWS uses the Library pattern.

The most direct uses of this pattern are in networking applications. Domain Name Service (DNS) servers use this pattern to hide the details of the DNS protocol and mappings to IP addresses from users of Fully Qualified Domain Names (FQDNs). The failed Singleton approach of this pattern is analagous to the use of `/etc/hosts` to manage IP address lookups. Caching DNS servers will cache the results so that future lookups for the same FQDN do not occur until after the entry expires.

Web caches, such as Squid [19], employ this pattern to determine whether a requested Web page was cached on a previous retrieval. The Web browser is typically

shielded from whether the object originated from the Web server or was delivered from a cache.

UNIX file systems can also be viewed as using this pattern. The `open ()` system call can first check for the existence of the file, and create a new one if it does not exist, or open it. The directory lookup is hidden.

4.8 Related Patterns

Lookup [9], Object Lifetime Manager [20].

A similar pattern is mentioned as part of the pattern language for *New Clients with Old Servers: A Pattern Language for Client/Server Frameworks* [21], but the pattern was not fully described.

5 Strategized Concurrency

The Strategized Concurrency pattern allows clients to alter dynamically the concurrency behavior of the server. It does so by decoupling the concurrency implementation from the application design.

5.1 Context

A server may have to deal with varying load conditions. For example, at times a server may face "bursty" loads, in which client request rates peak at varying intervals. At other times, the server may face steady loads, in which client request rates remain fairly consistent.

Typical concurrency strategies employed by multi-threaded servers include Thread Pool and Thread-Per-Request. A server may not want to be tied to a single concurrency strategy to adjust dynamically to varying load conditions.

5.2 Forces

An application may need to resolve the following forces. First is the need to experiment with different concurrency strategies to determine what provides optimal performance. Second is the need to alter the concurrency strategy of the application at run-time to deal with varying load conditions. Third is the need to minimize overhead when accessing the concurrency mechanism. Note that the third force directly opposes the first two.

5.3 Problem

The Thread Pool concurrency strategy avoids thread creation overhead, and fixes the concurrency resources available to the application. However, it makes it inflexible to cope with varying loads. If the application expects that at times many requests are going to be made, the pool must be made large enough to deal with the maximum expected load. On some implementations, this may lead to unncessary overhead even

if the threads are idle, because the synchronization mechanism may wake up all threads instead of just one.

To deal with ”bursty” conditions, it would be optimal to offer more concurrency when the server needs it, and less when it does not. The Thread-Per-Request concurrency strategy can be used to achieve this. However, this benefit comes at a cost of thread creation overhead. In addition, the application may possibly saturate available concurrency resources. The benefit of using this method depends on how bursty the traffic is and how long lived the sessions are to the server.

During stable periods of known predictable load conditions, thread pool would provide optimal performance. During bursty periods, a more dynamic concurrency strategy like thread per request could be more optimal.

5.4 Solution

Use the Strategized Concurrency pattern to provide distinct concurrency solutions for distinct operating regions.

In the dynamic case of the Strategized Concurrency pattern, the application can create a policy that decides which concurrency strategy to apply at discrete points during server execution. For example, a time based policy may cause the server to use Thread Pool concurrency at certain times of the day, and Thread-Per-Request at other times of the day.

In the static case of the Strategized Concurrency pattern, the application can apply a single fixed concurrency strategy for those cases where the load is well known and for which dynamic overhead must be avoided.

The ability to promote both dynamic and static solutions is achieved by applying the Concrete Bridge pattern, using either a concrete or abstract implementation class.

Discrete transitions should be used when changes in load conditions can only be managed by a transition to a different strategy. For example, consider a transition from 1) a very stable load of short requests that is best managed by a fixed thread pool, to 2) a very bursty load profile with long-running operations that is better managed by a thread-per-request strategy.

Continuous transitions should be used when changes in load conditions can still be managed by the same strategy, but with different characteristics. For example, consider a dynamic thread pool that can raise and lower the number of its threads between a high and low water mark.

The implementation closely follows Active Object, but specializes the methods that enqueue and dequeue requests from the object’s job queue. When a job producer enqueues a request, this triggers the object to decide whether or not to create a new thread. When a consumer thread dequeues a request, this triggers the object to decide whether or not the thread should be reaped.

In a Thread-Per-Request strategy, the decision is always to create a new thread on enqueue, and always to let the thread die when the queue is empty on dequeue. In a Thread-Pool strategy, the enqueue decision is never to create a new thread, and the dequeue decision is only to reap a thread if the concurrency object is being destroyed.

In a Hybrid strategy, the decisions involve a simple algorithm. A new thread is created upon insertion of jobs into the queue if there are not enough threads already

waiting, and the total number of running threads has not exceeded a pre-determined limit. Upon removing jobs from the queue, a thread will allow itself to expire if it detects that it is an extra thread (thread count higher than low water mark) and there are not any jobs in the queue.

5.5 Structure

Figure 4 shows the basic structure of the Strategized Concurrency design pattern.

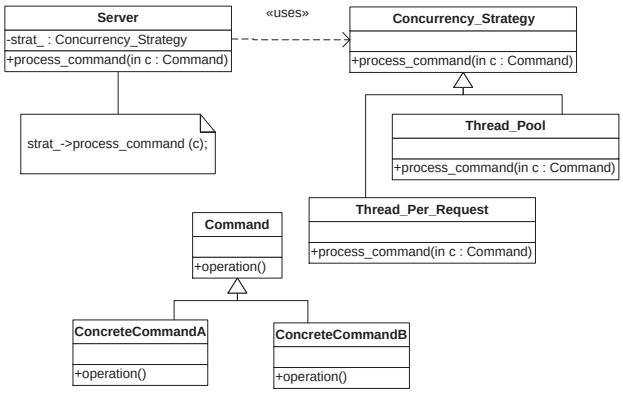

Figure 4: Structure of the Strategized Concurrency design pattern

5.6 Consequences

By employing the Concrete Bridge pattern, the Strategized Concurrency pattern gives application developers the ability to configure experimentally the concurrency implementation to be used at run-time. It also allows the application to change its concurrency implementation during execution. Furthermore, it allows an application to bind an implementation to an interface at compile time.

If the overhead costs of run-time flexibility are too great, then binding a "continuous" strategy may provide better performance. If the self-monitoring costs employed by the "continuous" strategy are too great, then loose coupling can be used along with "discrete" strategy transformations. Careful performance measurements are required to determine which approach will provide the best performance for a given application.

5.7 Known Uses

The JAWS application framework employs this pattern to achieve flexible control of the thread creation policy of the server application.

In ACE, the Reactor implementations include the TP_Reactor and Select_Reactor. The former is designed to be used in a Thread Pool dispatching strategy, while the Select_Reactor uses single threaded dispatching. The Reactor employs the Bridge pattern, so dynamic selection of the concurrency used for dispatching is possible.

5.8 Related Patterns

Bridge, State, Strategy [1], Concrete Bridge, Active Object [22].

6 Summary

Applying patterns at several levels of architectural scale can improve overall integrity and generativity of an application framework. In presenting these patterns, and an architectural context within which they recur, we seek to illustrate their applicability for server application frameworks, and to extend the state of the practice in developing such frameworks.

Acknowledgements

We would like to thank David Levine and Michael Kircher for their comments on the initial ideas for this article, and the patterns it describes. We are also grateful to our PLoP 2000 shepard, Ali Arsanjani, for expertly helping improve the quality of this paper.

References

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

[2] D. C. Schmidt, "The ADAPTIVE Communication Environment (ACE)." www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[3] D. C. Schmidt, "An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse," *;login:*, Nov. 1998.

[4] J. Hu and D. C. Schmidt, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, ch. JAWS: A Framework for High Performance Web Servers. Wiley & Sons, 1999.

[5] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[6] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *Proceedings of the* 1^{st} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, August 1994.

[7] I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," in *The* 4^{th} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[8] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *The* 3^{rd} *Annual Conference on the Pattern Languages of Programs (Washington University technical report #WUCS-97-07)*, (Monticello, Illinois), pp. 1–7, February 1997.

[9] M. Kircher and P. Jain, "Lookup Pattern," in *Submitted to EuroPLoP 2000 conference*, (Irsee, Germany), July 2000.

[10] A. S. Ran, "Patterns of Events," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[11] U. Kothe, "Requested Interface," in *The 1^{st} European Pattern Languages of Programming Conference*, July 1997.

[12] M. H. Austern, *Generic Programming and the STL*. Reading, MA: Addison-Wesley, 1999.

[13] T. Geraud and A. Duret-Lutz, "Generic Programming Redesign of Patterns," in *Submitted to EuroPLoP 2000 conference*, (Irsee, Germany), July 2000.

[14] A. C. M. Joseph A. Bank and B. Liskov, "Parameterized types for Java," in *24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, (Paris, France), pp. 132–145, ACM, December 1997.

[15] Center for Distributed Object Computing, "TAO: A High-performance, Real-time Object Request Broker (ORB)." www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[16] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, to appear 2000.

[17] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[18] C. Cleeland, D. C. Schmidt, and T. Harrison, "External Polymorphism – An Object Structural Pattern for Transparently Extending Concrete Data Types," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[19] Duane Wessels, "Squid Web Proxy Cache." http://www.squid-cache.org/, 1999.

[20] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction," *C++ Report*, vol. 12, Jan. 2000.

[21] J. O. Coplien and D. C. Schmidt, eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

[22] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.