Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University, St. Louis 63130
(TEL) 314-935-7538, (FAX) 314-935-7302

This article will appear in the Communications of the ACM Special Issue on Object-Oriented Experiences, Vol. 38, No. 10, October, 1995. Section 2 illustrates an example pattern description that is a synopsis of material from [1]. Section 3 is new material that summarizes the lessons learned while applying a design pattern-based reuse strategy on production large-scale distributed systems being developed on several commercial systems.

Abstract

Design patterns help to enhance software quality by addressing fundamental challenges in large-scale system development. These challenges include communication of architectural knowledge among developers, accommodating new design paradigms or architectural styles, and avoiding development traps and pitfalls that are usually learned only by experience. This article describes lessons learned from applying a design pattern-based reuse strategy to develop object-oriented communication software frameworks for several production distributed systems. Design patterns are in danger of becoming yet another buzzword. This article attempts to replace the hype with insights and recommendations gained from several years of experience.

1 Introduction

Despite dramatic increases in network and host performance it remains difficult to design, implement, and reuse communication software for complex distributed systems. Examples of these systems include global personal communication systems, network management platforms, enterprise medical imaging systems, and real-time market data monitoring and analysis systems. In addition, it is often hard to directly reuse existing algorithms, detailed designs, interfaces, or implementations in these systems due to the growing heterogeneity of hardware/software architectures and diversity of operating system platforms.

Design patterns [2] are a promising technique for achieving widespread reuse of software architectures. Design patterns capture the static and dynamic structures and collaborations of components in successful solutions to problems that arise when building software in domains like business data processing, telecommunications, graphical user interfaces, databases, and distributed communication software. Patterns aid the development of reusable components and frameworks by expressing the structure and collaboration of participants in a software architecture at a level higher than (1) source code or (2) object-oriented design models that focus on individual objects and classes. Thus, patterns facilitate reuse of software architecture, even when other forms of reuse are infeasible (*e.g.,* due to fundamental differences in operating system features [3]).

This article describes how design patterns are being applied on a number of large-scale commercial distributed systems. Patterns have been used on these projects to enable widespread reuse of communication software architectures, developer expertise, and object-oriented framework components. These systems include the system control segment for the Motorola Iridium global personal communications system [4]; a family of network monitoring applications for Ericsson telecommunication switches [3]; and a system for transporting multi-megabyte medical images over high-speed ATM networks [5] being developed at BJC health system and Washington University School of Medicine (BJC/WUSM). This article also presents ways to avoid common traps and pitfalls of applying design patterns in large-scale software development processes.

The remainder of this article is organized as follows: Section 2 illustrates a design pattern that appears frequently in event-driven communication software frameworks; Section 3 summarizes our experiences (both positive and negative) applying a design pattern-based reuse strategy in several large-scale commercial distributed systems; and Section 4 outlines pattern-related topics that will be the focus of considerable research activities in the next few years.

2 Example Design Pattern: the Reactor

This section describes the *Reactor pattern*. This pattern was identified while developing reusable event-driven communication software at Ericsson, Motorola, and BJC/WUSM. Portions of the material below was culled from documentation used on these projects.

Design patterns have been described using several formats [2, 6, 7]. The format used below is based on the work of Gamma et al. [2]; it contains the following parts:

- The intent of the pattern
- The design forces that motivate the pattern
- The solution to these forces
- The structure and roles of classes in the solution
- The responsibilities and collaborations among classes
- The positive and negative consequences of using the pattern
- Implementation guidance
- Example source code[1]
- References to related patterns

2.1 Intent

The Reactor pattern dispatches handlers automatically when events occur from multiple sources. This pattern simplifies event-driven applications by decoupling event demultiplexing and event handler dispatching from application services performed in response to events.

2.2 Motivation

Communication software must respond to events generated from multiple sources. For example, network management applications for monitoring and controlling space vehicles in the Motorola Iridium satellite constellation receive traps sent by HP OpenView agents, telemetry data sent via CORBA requests, and user interface events generated by Motif. These events arrive on multiple I/O handles that identify resources (such as network connections) managed by an operating system. Input events from peers may arrive simultaneously on multiple handles. Therefore, single-threaded software must not block indefinitely reading from any individual I/O handle. Blocking can significantly delay the response time for handling events from peers associated with other handles.

One way to develop this type of event-driven software is to use multi-threading. In this approach, a separate thread is spawned for every connected peer. Each thread blocks on a `read` system call. A thread unblocks when it receives an event from its associated peer. At this point, the event is processed within the thread. The thread then re-blocks awaiting subsequent input from `read`.

There are several drawbacks to using multi-threading for handling events in communication software:

- threading may require complex concurrency control schemes;
- threading may lead to poor performance due to context switching, synchronization, and data movement;
- threading may not be available on an OS platform.

Often, a more convenient and portable way to develop event-driven servers is to use the *Reactor pattern*. The Reactor pattern manages a single-threaded event loop that performs event demultiplexing and event handler dispatching in response to events from multiple sources. The Reactor pattern combines the simplicity and efficiency of single-threaded event loops with the extensibility offered by object-oriented programming.

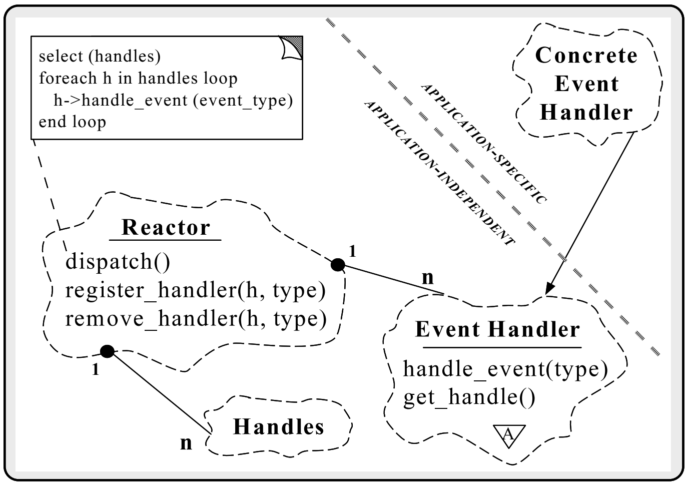
2.3 Applicability

Use the Reactor pattern when:

- one or more events may arrive concurrently from multiple sources, and blocking or continuously polling for events on any individual source of events is inefficient;
- each individual event handler possesses the following characteristics:
  - it exchanges fixed-sized or bounded-sized messages with its peers without requiring blocking I/O;
  - it processes each message it receives within a relatively short period of time;
- using multi-threading to implement event demultiplexing is either:
  - *infeasible* – due to lack of multi-threading support on an OS platform;
  - *undesirable* – due to poor performance on uniprocessors or due to the need for overly complex concurrency control schemes;
  - *redundant* – due to the use of multi-threading at higher levels of an application's architecture.

2.4 Structure

The structure of the Reactor pattern is illustrated in the following Booch [8] class diagram:



[1] Due to space limitations the sample code has been omitted from this article. See [1] for a complete example of the Reactor pattern.

In Booch notation dashed clouds indicate classes, an inscribed "A" indicates an abstract class, directed edges indicate inheritance relationships between classes, and an undirected edge with a small bullet at one end indicates a composition relation between two classes.

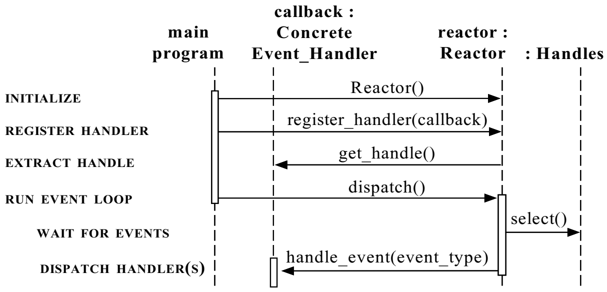
2.5 Participants

The participants in the Reactor pattern include the following:

- **Handles**
  - Handles identify resources (such as network connections, open files, and synchronization objects) that are managed by an operating system.
- **Reactor**
  - Defines an interface for registering, removing, and dispatching `Event Handler` objects. An implementation of the `Reactor` interface provides a set of application-independent event demultiplexing and dispatching mechanisms. These mechanisms dispatch application-specific `Event Handlers` in response to events occuring on one or more `Handles`.
- **Event Handler**
  - Specifies an interface used by the `Reactor` to dispatch callback methods defined by objects that are pre-registered to handle certain types of events (such as input events, output events, and signals).
- **Concrete Event Handler**
  - Implements the customized callback method(s) that process events in an application-specific manner.

2.6 Collaborations among Participants

- Sources of events (such as network adaptors, file systems, and transaction managers) communicate with the `Reactor` via `Handles`.
- Developers subclass `Event Handlers` to implement application-specific event processing. When an `Event Handler` subclass object is registered with the `Reactor` the application indicates the type of event(s) (*e.g.,* input event, output event, signal event, etc.) this `Event Handler` wants the `Reactor` to notify it about.
- To bind the `Reactor` with `Handles`, a subclass of `Event Handler` must override the `get_handle` method. Hence, when an `Event Handler` subclass object is registered with the `Reactor` the object's `Handle` is obtained by invoking the `Event Handler::get_handle` method. The `Reactor` then combines this `Handle` with other registered `Event Handlers` and waits for events to occur on the `Handles`.
- The `Reactor` triggers `Event Handler` methods in response to events on the `Handlers` it monitors. When events occur, the `Reactor` uses the `Handles` activated by the events as keys to locate and dispatch the appropriate `Event Handler` methods. This collaboration is structured using the method callbacks depicted in the following object interaction diagram:



The `handle_event` method is called by the `Reactor` to perform application-specific functionality in response to an event. The type of the event that occurred is passed as a parameter to the method.

2.7 Consequences

The Reactor pattern has the following benefits:

- It improves the modularity, reusability, and configurability of event-driven application software by decoupling application-independent mechanisms from application-specific processing policies.
- It improves application portability by allowing its interface to be reused independently of the underlying OS system calls that perform event demultiplexing.
- It provides applications with coarse-grained concurrency control that serializes the invocation of `Event Handlers` and minimizes the need for more complicated synchronization or locking within an application process.

The Reactor pattern has the following drawbacks:

- *Event Handlers* are not preempted while they are executing. Therefore, a handler should not perform blocking I/O on a `Handle` since this will significantly decrease the responsiveness to clients connected to other I/O handles. Therefore, for long-duration operations (such as transferring a multi-megabyte medical image) the Active Object pattern [9] (which uses multi-threading or multi-processing) may be more effective.
- Applications written using the Reactor pattern can be hard to debug because their flow of control oscillates between the lower-level demultiplexing code (supplied by the framework) and the higher-level method callbacks (supplied by application developers).

2.8 Implementation

The Reactor pattern can be implemented in many ways. This section discusses several topics related to implementing the Reactor pattern.

- *Event demultiplexing* – A `Reactor` maintains a table of objects that are derived from the `Event Handler` base class. Public methods in the `Reactor`'s interface register and remove these objects from this table at run-time. The `Reactor` also provides a means to dispatch the `handle_event` method on an `Event Handler` object in response to events the application has registered to receive.

  The `Reactor`'s `dispatch` method blocks on an OS event demultiplexing system call (such as Windows NT `WaitForMultipleObjects` or UNIX `select`) until one or more events occur. When events occur, the `Reactor` returns from the event demultiplexing system call. It then dispatches the `handle_event` method on any `Event Handler` object(s) that are registered to handle these events. This callback method executes user-defined code and returns control to the `Reactor` when it completes.

- *Registering objects vs. functions* – The Reactor pattern shown in Section 2.4 registers `Event Handler` subclass objects with a `Reactor`. The use of objects makes it convenient to subclass `Event Handlers` in order to flexibly reuse and extend existing components, as well as to integrate data and methods together. Another approach is to register a function rather than an object. The use of functions makes it convenient to register callbacks without having to define a new class that inherits from `Event Handler`. A hybrid approach can be used to support both objects and functions simultaneously.

- *Event handling interface* – The diagram in Section 2.4 illustrates an implementation of the `Event Handler` base class interface that contains a single method (`handle_event`) used by the `Reactor` to dispatch events. In this case, the type of the event (*e.g.,* input event, output event, signal event, etc.) is passed as a parameter to the method. This approach makes it possible to add new types of events without changing the interface. However, this approach encourages the use of switch statements in the subclass's `handle_event` method, which limits extensibility.

  Another way to implement the `Event Handler` interface is to define separate virtual methods for each type of event (*e.g.,* `handle_input`, `handle_output`, `handle_signal`, etc.). This approach is easier to extend since subclassing does not involve switch statements. However, it requires the framework developer to anticipate the set of `Event Handler` methods in advance.

- *Synchronization* – The `Reactor` can serve as a central event dispatcher in multi-threaded applications. In this case, critical sections within the `Reactor` must be serialized to prevent race conditions when modifying or activating shared variables (such as the table holding the `Event Handler` subclass objects). A common technique for preventing race conditions uses mutual exclusion mechanisms like semaphores or mutex variables.

  To prevent deadlock, mutual exclusion mechanisms should use *recursive locks*. Recursive locks are an efficient means to prevent deadlock when locks are held by the same thread across `Event Handler` method callbacks within the `Reactor`. A recursive lock may be re-acquired by the thread that owns the lock *without* blocking the thread. This property is important since the Reactor's `dispatch` method performs callbacks on application-specific `Event Handler` objects. Application callback code may subsequently re-enter the `Reactor` object using its `register_handler` and `remove_handler` methods.

- *I/O semantics* – The I/O semantics of the underlying OS significantly affect the implementation of the Reactor pattern. The standard I/O mechanisms on UNIX systems provide "reactive" semantics. For example, the UNIX `select` system call indicates the subset of I/O handles that may be read from or written to synchronously without blocking.

  Implementing the Reactor pattern using reactive I/O is straightforward. In UNIX, `select` indicates which handle(s) are ready to perform I/O. The `Reactor` object then "reacts" by invoking the `Event Handler` `handle_event` callback method for each ready handle. This method performs the I/O operation and the associated application-specific processing.

  In contrast, Windows NT provides "proactive" I/O semantics. Proactive I/O operations proceed asynchronously and do not cause the caller to block. An application may subsequently use the WIN32 `WaitForMultipleObjects` system call to determine when its outstanding asynchronous I/O operations have completed.

  Variations in the I/O semantics of different operating systems may cause the class interfaces and class implementations of the Reactor pattern to vary across platforms. Schmidt and Stephenson [3] provide a detailed evaluation of how differences between proactive and reactive event demultiplexing affect implementations of the Reactor pattern on UNIX and Windows NT.

2.9 Known Uses

The Reactor pattern has been used in many object-oriented frameworks and event-driven applications:

- The X windows toolkit uses a version of the Reactor pattern to structure its main event loop. This implementation registers and dispatches function calls, rather than objects.
- The InterViews window system distribution [10] implements the Reactor pattern in its `Dispatcher` class category. The `Dispatcher` is used to define an application's main event loop and to manage connections to one or more physical GUI displays.
- The ADAPTIVE Service eXecutive (`ASX`) framework [1] uses the Reactor pattern as the central event demultiplexer/dispatcher in an object-oriented toolkit for experimenting with high-performance parallel communication protocol stacks.
- The Reactor pattern has been used in a number of commercial products. These products include the Bellcore and Siemens Q.port ATM signaling software product, the Ericsson EOS family of telecommunication switch management applications [3], the network management portion of the Motorola Iridium global personal communications system [4], and in an enterprise medical image delivery system for BJC/WUSM [5].

2.10 Related Patterns

The Reactor pattern may be viewed as a variation on the Observer pattern [2]. In the Observer pattern, subscribers are updated automatically when a single subject changes. In the Reactor pattern, handlers are informed automatically when events from multiple sources occur.

A `Reactor` provides a Facade [2] for event demultiplexing. A Facade is an interface that shields applications from complex object relationships within a subsystem.

The virtual methods provided by the `Event Handler` base class are Template Methods [2]. These template methods are used by the `Reactor` to trigger callbacks to the appropriate application-specific processing functions in response to events.

The Active Object pattern [9] decouples method execution from method invocation in order to simplify synchronized access to a shared resource by methods invoked in different threads of control. This pattern is often used in place of (or in conjunction with) the Reactor pattern when `Event Handlers` perform long-duration activities. Likewise, the Reactor pattern can be used in place of (or in conjunction with) the Active Object pattern when threads are not available or when the overhead and complexity of managing large numbers of threads is undesirable.

3 Lessons Learned

This section describes lessons learned from developing object-oriented communication frameworks based on design patterns at Motorola Iridium [4, 11], Ericsson [3], and BJC/WUSM [5]. These large-scale distributed system projects have identified, documented, and applied dozens of new or existing design patterns. Patterns were used to leverage prior development expertise, as well as to reduce risk by reusing software architectures across diverse OS platforms and subsystems.

The Motorola Iridium and Ericsson projects were among the first large-scale distributed system projects to adopt a software reuse strategy based on the concepts, notations, and techniques of design patterns. Patterns identified and applied in these projects have been described in [3, 4, 1, 9, 11, 12]. These projects have clarified many of the benefits and drawbacks of using design patterns to systematically capture and articulate communication software architectures.

This section discusses the lessons learned and outlines workarounds for problems we encountered using design pattern-based reuse strategies in production software environments. Our experiences using patterns on the Ericsson, Motorola, and BJC/WUSM projects were quite similar. Recognizing these common themes across different projects increased our confidence that these experiences generalize to using patterns on other large-scale software systems.

• **Patterns enable widespread reuse of of software architecture** even if reuse of algorithms, implementations, interfaces, or detailed designs is not feasible. The constraints of the underlying OS and hardware platform significantly affect design and implementation decisions. This is particularly problematic for communication software, where non-portable OS features are often used to enhance performance and functionality.

The Ericsson telecommunication switch management project [3] illustrated the importance of patterns-based architectural reuse. This project underwent several major changes over three years. The original prototype was developed on UNIX using sockets, `select`, and TCP/IP as the communication mechanisms. After six months, however, the OS platform changed to Windows NT with WIN32 named pipes, NETBEUI, `WaitForMultipleObjects`, and TCP/IP as the communication mechanisms. A year later, the scope of the project changed again to integrate with a much larger switch management subsystem. All these changes involved extensive porting and modification of existing communication software.

In such a volatile environment, reusing design patterns was often the only viable means of leveraging previous development expertise. For example, fundamental differences in the I/O mechanisms available on Windows NT and UNIX precluded the direct reuse of Reactor pattern implementations across those OS platforms. We were, however, able to reuse the Reactor pattern itself, customizing portions of it to conform with the characteristics of the OS platforms. This reduced project risk significantly and simplified our redevelopment effort.

• **Pattern descriptions should contain concrete examples.** Because patterns abstract the properties of successful designs they are not limited to a single implementation. As described above, this makes it possible to evolve and adapt patterns to changes in environments and requirements. Although patterns are inherently abstractions, however, patterns should not be described too abstractly. We found that many developers had a hard time understanding precisely how to implement patterns when they were described using only object diagrams and structured prose.

To overcome this problem, we provided source code examples with our pattern descriptions that gave concrete guidance for implementing the patterns. Whenever possible, we presented multiple implementations of each pattern to help developers overcome the "tunnel vision" that resulted from a limited pattern vocabulary. Many examples used in our pattern descriptions came directly from communication frameworks we built for the projects. This helped developers grasp the key points of each pattern because they already understood the forces and requirements that motivated the pattern. In addition, this approach helped convince management to support the use of patterns because it reinforced our claim that design patterns had direct relevance to their projects.

• **Patterns improve communication within and across software development teams** because they provide developers with shared vocabulary and concepts. Patterns capture essential properties of software architecture, while suppressing details that are not relevant at a given level of abstraction. Thus, they provide a comprehensible way of documenting complex software architectures by expressing the structure and collaboration of participants at a level higher than (1) source code or (2) object-oriented design models that focus on individual objects and classes.

We found that patterns helped to elevate the level of discourse among team members. For example, once developers understood patterns like Factory Method [2] and the Reactor they could convey and justify their design and implementation decisions swiftly and clearly to other team members. In addition, patterns helped to bridge the communication gap that existed between software developers, managers, and non-technical team members in marketing and sales. Managers and non-technical team members often could not understand the system at the level of detailed object models or source code. However, they frequently could understand and evaluate the consequences and tradeoffs among software architecture concepts that were expressed as patterns. Their feedback was valuable to ensure that our technical solutions did not drift away from the overall system requirements.

• **Pattern names should be chosen carefully and used consistently.** For patterns to achieve widespread use on a project, developers must share an unambiguous vocabulary of common patterns. Selecting appropriate pattern names is hard due to the tension between parsimony and descriptiveness. Concise names like Reactor or Iterator are appealing because they convey the essence of a pattern with minimal verbal effort. This brevity is conducive to rapid-fire design brainstorming sessions. However, these verbal shorthands can be confusing unless developers have internalized the underlying concepts and can associate them immediately with the appropriate patterns.

We addressed this problem by publishing more descriptive aliases along with our patterns. For example, the Reactor pattern's alias was "dispatch handlers automatically when events occur from multiple sources." Ensuring names and aliases are used consistently throughout a project reduces the likelyhood that developers will waste time debating the consequences, structure, etc. of a pattern, only to realize they were actually talking about different patterns, or different variations of the same pattern.

• **Patterns explicitly capture knowledge that experienced developers already understand implicitly.** Therefore, after being introduced to design patterns, most developers adopted pattern nomenclature and concepts eagerly. This enthusiasm stemmed in part from the fact that pattern descriptions explicitly codified knowledge they understood intuitively. For example, many developers intuitively understand the forces that cause certain solutions to be preferred over alternatives. However, these non-functional forces are often not captured adequately with existing design methods and notations. We found that the use of patterns helped experts document, discuss, and reason systematically about sophisticated architectural concepts.

In addition, explicitly capturing expertise through patterns helps to impart this knowledge to less experienced developers. For example, the Reactor pattern reused across OS platforms in the Ericsson project represented knowledge gained over years of experience with event-driven communication software on many projects at different companies. By carefully documenting key patterns in our domain, we were able to preserve and share this expertise. This saved developers a great deal of time that would have otherwise been spent rediscovering these patterns in new contexts.

• **Patterns may lead developers to think they know more about the solution to a problem than they actually do.** One downside to the intuitive nature of patterns is that developers may not fully appreciate the challenges associated with implementing a pattern. Simply knowing the structure and participants in a pattern (such as the Reactor pattern) is only the first step. As described in [3], a significant development effort and commitment of time and resources may be necessary to implement a pattern correctly, efficiently, and portably in a particular context. We addressed this problem by continually emphasizing to developers that learning patterns complements, but does not substitute for, solid design and implementation skills.

• **Resist the temptation to recast everything as a pattern.** Another downside to the intuitive nature of patterns is that it may lead to *pattern overload*. We noticed that the benefits of patterns became diluted if too many aspects of a project were expressed as patterns. This problem arose when development practices were relabled as patterns without significantly improving them. For example, some developers spent a great deal of time recasting relatively mundane concepts (such as binary search, building a linked list, or recursion) using the pattern form. Although this was intellectually satisfying, pattern overload became counter productive when it did not markedly improve software quality.

• **Focus on developing patterns that are strategic to the domain and reuse existing tactical patterns.** Existing pattern catalogs [2, 7] do an excellent job of documenting many domain-independent, tactical patterns (such as Factory Method, Abstract Factory, and Singleton). Tactical patterns have a relatively localized impact on a software architecture. For instance, the Iterator pattern [2] allows elements in a collection to be accessed sequentially without violating data encapsulation. Although this pattern is widely applicable, the problem it addresses does not have sweeping architectural implications.

In contrast, strategic design patterns have an extensive impact on the software architecture for solutions in a particular domain. For example, the Half-Sync/Half-Async pattern [12] decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency. This pattern greatly simplifies synchronization strategies in complex concurrent systems (such as BSD UNIX). We focused most of our energy on documenting patterns related to our domain (communication software) and we reused existing tactical patterns rather than reinventing them. Focusing on strategic domain patterns also helped to minimize the likelyhood of pattern overload.

• **Institutionalize rewards for developing patterns.** Unfamiliar design paradigms and methods may be perceived as threats to the traditional power structure and base of expertise in an organization. We observed a manifestation of this problem where some developers were reluctant to share their domain patterns. They viewed this knowledge as a competitive advantage over other developers. These types of problems are indicative of deeper issues related to the reward structure in a corporate culture, which is often hard to change [13]. We addressed this problem to the extent possible by instituting incentives for developing useful pattern descriptions. These descriptions were counted as "deliverables" used to measure individual performance. We measured the utility of design patterns by how widely they were adopted and used successfully (particularly by developers other than the original authors).

• **Useful patterns arise from practical experience.** Therefore, we found that it was important to work closely with domain experts in order to identify and document key patterns in the communication domain. One consequence of the experiential basis of patterns is that they are discovered "bottom-up" rather than invented "top down." One sign that pattern overload is taking place is when developers start planning to "invent patterns."

• **Patterns help ease the transition to object-oriented technology** for developers who were trained in traditional design techniques. Many patterns in our communication frameworks (such as the "pipes and filters architecture" [7]) originated in non-object-oriented contexts such as operating systems and databases. By explicitly recognizing and rewarding the experiential basis of useful patterns, we were able to leverage valuable developer expertise from earlier design paradigms [14].

• **Patterns are validated by experience rather than by testing,** in the traditional sense of "unit testing" or "integration testing" of software. This can be problematic because it is hard to determine when a pattern description is complete and correct. The most effective way we found to validate our patterns was through periodic pattern reviews. These reviews helped to enrich the pattern vocabulary within and across development teams. We modeled these pattern reviews as "writers workshops." At these reviews, developers from different teams presented useful patterns they observed in their software. Group members discussed the strengths and weaknesses of each pattern, accentuating positive aspects of the patterns, sharing their own experience, and suggesting improvements in content and style.

• **Directly involve pattern authors with application developers and domain experts.** We found that isolating pattern authors from development teams resulted in overly abstract patterns that did not capture the essence of successful designs. This is similar to the problem that arises in large-scale reuse initiatives that become disconnected from application development. In both cases the resulting software artifacts can be too general to solve the actual domain requirements. We addressed this problem by using the pattern review techniques described above to glean system-level patterns from domain experts on the projects. These patterns were incorporated into the project following careful scrutiny in pattern reviews.

• **Integrating patterns into a software development process is a human-intensive activity.** Like other software reuse technologies, reuse of patterns does not come for free [14]. Identifying and documenting useful patterns requires both concrete experience in a domain *and* the ability to abstract away from concrete details to capture the general properties of patterns. We found that relatively few individuals possess both these skills with equal proficiency. Therefore, engaging groups of developers with diverse backgrounds and skills in pattern review sessions was essential to leverage patterns effectively.

However, pattern reviews required a significant investment by organizations. The review process is fundamentally interactive and creative, rather than automated and rote. Organizations that do not actively encourage these reviews (*e.g.,* due to tight schedules or due to a view that software development should be a mechanical process) may devote inadequate time and developer resources to this review process.

While we generally found the educational benefits of the pattern reviews justified the costs, we also recognized that this style of review process does not scale up easily. For example, experienced developers with deep knowledge of the communication domain were fertile sources of strategic and tactical patterns, as well as invaluable mentors in pattern reviews. However, these experts were often very busy with other tasks, and could not always spare much time to write or review pattern descriptions thoroughly. This is another reason why documenting patterns should be institutionalized in an organization's reward structure.

• **Pattern descriptions explicitly record engineering tradeoffs and design alternatives that resolve non-functional forces.** Because pattern descriptions explicitly enumerate consequences and implementation tradeoffs they can be used to record why certain design choices were selected and others rejected. For example, the description of the Reactor pattern in Section 2 explains precisely when to apply the pattern (*e.g.,* when each event can be processed quickly) and when to avoid it (*e.g.,* when transferring large amounts of bulk data). If this rationale is not captured explicitly it may be lost over time. This loss deprives maintenance teams of critical design information, and makes it difficult to motivate strategic design choices to other groups within a project or organization.

• **Carefully document the contexts where patterns apply and do not apply.** When developers first write pattern descriptions they tend to emphasize the beneficial consequences of the patterns without thoroughly covering the drawbacks of using the pattern in certain contexts. For example, as described in the previous paragraph, the Reactor pattern can be an inefficient event demultiplexing mechanism on multi-processor platforms because it serializes application concurrency at a coarse-grained level [9]. If this caveat is not explicitly captured in a pattern description, developers may apply the pattern inappropriately. Therefore, pattern descriptions should enumerate both the benefits and the drawbacks of a pattern, as well as motivate the context in which the pattern applies or does not apply.

• **Successful pattern descriptions capture both structure and behavior.** Expressing the behavioral aspects of a pattern is hard because behaviors involve dynamic collaboration between participants. However, patterns that do not clearly describe dynamic behavior are difficult to understand and apply. We found object interaction diagrams and object interaction graphs were particularly useful for depicting key collaborations in a design without requiring the attention to detail necessary to understand source code.

• **Patterns facilitate training of new developers** by allowing developers joining the projects to absorb the key strategies and tactics in the software design quickly. We exposed developers to our pattern documentation before having them delve into the software. We found that the ability to express the intent, structure, and behavior of our frameworks in terms of patterns lowered the learning curve for new developers by giving them a broad understanding of the architecture in our communication frameworks.

We expect that this aspect of patterns will prove useful for maintenance programmers. However, the projects at Ericsson, Motorola, and BJC/WUSM are all new systems that have not yet entered the long-term maintenance phase. We are collecting additional information on how patterns affect maintenance over the software lifecycle.

• **Implementing patterns efficiently requires careful selection of language features.** Existing patterns literature [2, 6, 7, 15] has focused primarily on software quality factors other than performance. This may be acceptable in domains where non-functional requirements are more important than efficiency. For example, graphical user interfaces are often judged more in terms of their usability, extensibility, and portability than their raw performance.

In contrast, communication software has traditionally emphasized high performance more than other software quality factors. Thus, we found that many developers were initially concerned about the performance costs of using design patterns in the communication domain. To allay these concerns, many of our pattern implementations used C++ parameterized types extensively, rather than inheritance and dynamic binding. Parameterized types do not degrade the run-time efficiency of performance-critical applications because template instantiation occurs at compile-time. In contrast, alternative techniques for extensibility using inheritance and dynamic binding incur a run-time performance penalty in C++, due to virtual function table dispatching overhead.

• **Patterns help to transcend "programming language-centric" viewpoints.** Focusing on design patterns helped us to move away from "programming language-centric" views of the object paradigm. This was beneficial because it enabled experienced developers from different language communities (such as Lisp, Smalltalk, Ada, Eiffel, C++, C, and Erlang) to share design insights of mutual interest without being distracted by "language wars."

Once we moved beyond language syntax and semantic differences, it was remarkable how much commonality was shared by successful software solutions for a given design problem. However, we also found that many developers wanted to see pattern examples illustrated with the programming language they were most familiar with (C++ in our projects).

• **Managing expectations is crucial to using patterns effectively.** One recurring problem we encountered using patterns centered around managing the expectations of development team members. Some team members had misconceptions about precisely how and what design patterns contributed to project success. For example, the use of patterns does not guarantee flexible and efficient software. Moreover, in their abstract form, patterns cannot be used directly by programmers in their implementations. Tools do not yet exist that transform design patterns into code automatically. Custom implementation is often required, unless the patterns have been integrated into a reusable framework or library.

We worked hard at Ericsson, Motorola, and BJC/WUSM to prevent design patterns from becoming yet another buzzword. We did this by candidly reporting the benefits and limitations of patterns and stressing that patterns are just one of many important tools in a development organization's toolkit. Patterns are no silver bullet that will absolve developers from having to wrestle with complex analysis, design, and implementation issues. In our experience, there is simply no substitute for creativity, experience, and diligence on the part of developers.

Over time, the contribution of patterns will become evident as software developers gain experience incorporating patterns into their development practices. Our experience applying design patterns in large-scale distributed systems was that they contributed to developing quality software by addressing fundamental challenges in large-scale system development. These challenges include communication of architectural knowledge among developers, accommodating new design paradigms or architectural styles, and avoiding development traps and pitfalls that are usually learned only by experience.

4 Concluding Remarks

Patterns capture the static and dynamic aspects of successful solutions to problems that commonly arise when building software systems. If software is to become an engineering discipline, these successful practices and design expertise must be documented systematically and disseminated widely. Patterns are important tools for documenting these practices and expertise, which traditionally existed primarily in the minds of expert software architects.

Over the next few years a wealth of software design knowledge will be captured in the form of strategic and tactical patterns that span disciplines such as client/server programming, distributed processing, organizational design, software reuse, real-time systems, business and financial systems, and human interface design. In addition, the following aspects of patterns will receive increased attention in the next few years:

- *Integration of design patterns together with frameworks* – Patterns can be viewed as abstract descriptions of frameworks that facilitate widespread reuse of software architecture. Frameworks can be viewed as concrete realizations of patterns that facilitate direct reuse of design and code. One difference between patterns and frameworks is that patterns are described in language-independent manner, whereas frameworks are generally implemented in a particular language. However, patterns and frameworks are highly synergistic concepts, with neither subordinate to the other.

  The next generation of object-oriented frameworks will explicitly embody dozens or hundreds of patterns – and patterns will be widely used to document the form and contents of frameworks [16]. Ideally, systems of patterns and frameworks will be integrated with tools like on-line pattern browsers that contain hyper-text links to navigate quickly through multiple levels of abstraction.

- *Integration of design patterns to form systems of patterns* – most literature on patterns is currently organized as design pattern catalogs [2, 6, 7]. These catalogs present a collection of individual solutions to common design problems. As more experience is gained using these patterns, developers will integrate groups of related patterns to form *pattern systems* (also called *pattern languages*). These pattern systems will encompass a family of related patterns that cover a particular domain (such as communication software).

  In the same sense that comprehensive application frameworks support larger-scale reuse of design and code than do class libraries, pattern systems will support larger-scale reuse of software architecture than individual patterns. Developing comprehensive systems of patterns is challenging and time consuming, but will likely provide the greatest payoff for pattern-based software development during the next few years.

- *Integration with popular object-oriented methods and software process models* – Patterns help to alleviate software complexity at several phases in the software lifecycle. Although patterns are not a software development method or process, they complement existing methods and processes. For instance, patterns help to bridge the abstractions in the analysis and architectural design phases with the concrete realizations of these abstractions in the implementation and maintenance phases. In the analysis and design phases, patterns help to guide developers in selecting from software architectures that have proven to be successful. In the implementation and maintenance phases they help document the strategic properties of software systems at a level higher than source code and individual object models.

This paper just scratches the surface of activities the patterns community is currently engaged in. A number of books [2, 6, 7, 15] have been published (or will soon be published) on these topics. The Pattern Languages of Programming conference [6] is an annual forum dedicated to improving the expression of patterns. There are also pattern workshops at on object-oriented conferences (such as OOPSLA, ECOOP, and USENIX COOTS). The World Wide Web URL `http://st-www.cs.uiuc.edu/users/patterns` contains a comprehensive on-line reference to pattern-related material.

Acknowledgements

I would like to thank Mohamed Fayad, Jim Coplien, Adam Porter, Tim Harrison, Ehab Al-Shaer, and the anonymous reviewers for improving the quality of this paper. I would also like to thank Paul Stephenson of Ericsson for many hours of discussion on design patterns and techniques for developing object-oriented communication software frameworks.

References

[1] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

[3] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[4] D. C. Schmidt, "A System of Reusable Design Patterns for Communication Software," in *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)* (S. P. Berczuk, ed.), Wiley and Sons, 1995.

[5] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

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

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wileys and Sons, to appear 1996.

[8] G. Booch, *Object Oriented Analysis and Design with Applications* (2^{nd} *Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[9] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the* 2^{nd} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

[10] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.

[11] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Active and Passive Establishment of Network Connections," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

[12] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the* 2^{nd} *Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, September 1995.

[13] J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Programs* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, June 1995.

[14] M. Fayad, W. Tsai, and M. Fulghum, "Transition to Object-Oriented Software Development," *Communications of the ACM*, Jan. 1996.

[15] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.

[16] R. Johnson, "Documenting Frameworks Using Patterns," in *OOPLSA '92*, (Vancouver, British Columbia), pp. 63–76, ACM, October 1992.