The Application of Patterns to Concurrent Control Flow

by

Caroline Kierstead

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2001

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Concurrency is an important programming paradigm to take advantage of the multiple processors available in current computers. This essay examines design patterns as a means of organizing the current body of literature on concurrency. A simple pattern catalog is presented, which divides concurrent design patterns into three main groupings: synchronization, mutual exclusion, and client-server. Within the category of client-server patterns, design patterns can be further subdivided into client-side patterns, server-side patterns, and client-server interactions. This last category neatly covers the areas missing from the previous two, subsuming both delegation and publication patterns. These divisions account for both the common and exotic patterns found in concurrent programming, without descending to the level of idioms such as semaphores or monitors.

Acknowledgements

I would like to thank my supervisor, Dr. Peter Buhr, for his help and patience throughout this process. Michael Van Biesbrouck and Jack Rehder have provided support, ideas, and the occasional text book loan. And, of course, my family and friends for their encouragement. Without them, nothing could have been done.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There is no new thing under the sun. – Old Testament 8:304

The essence of pattern creation is the codification of knowledge for future reuse. Patterns are found in subject areas ranging from architecture, where noted architect Christopher Alexander's work provided motivation for pattern work in software engineering, to business hierarchy management, to handbooks such as "The Civil Engineering Handbook" or the "Handbook of Chemistry and Physics", to the biological classification of life. However, in order to discuss a *pattern*, it must first be defined.

1.1 Definition

Brad Appleton summarizes multiple pattern definitions well, stating that

"a pattern involves a general description of a recurring solution to a recurring problem replete with various goals and constraints. But a pattern does more than identify a solution, it also explains *why the solution is needed*!"[13]

The most succinct definition belongs to James O. Coplien: "A pattern is a solution to a problem in a context" [100]. However, John Vlissides is quick to point out that this definition is incomplete, since, at a minimum, it leaves out the concepts of *recurrence*, *teaching*, and a *name* [129], which are defined as:

recurrence of the situation makes the solution relevant outside the immediate problem.

teaching allows a user to understand the pattern sufficiently so that a solution is tailored to problem variants. This concept is accomplished mostly through the description and resolution of acting forces, and the application consequences.

name allows easy reference to the pattern. This concept is required for communication of a shared vocabulary among participants.

Knutson, Budd, and Cook take the idea of patterns, which are often invented solely for a specific paradigm such as object-oriented programming, further. They discuss the notion of a "true pattern", which is reasonable in any paradigm [72].

The culmination of current pattern development is an increasingly common, if not canonical, template used by the majority of pattern writers to describe patterns. (In some sense, a meta-pattern, describing the general form of an actual pattern. This idea of a meta-pattern is not to be confused with Pree's *metapattern*, discussed later in this essay.) At a minimum, the pattern must consist of a name, a statement of the problem the pattern is intended to solve, the problem's context or applicability, a description of the forces and constraints and their interactions, the solution, illustrative examples (which often includes known instances of the pattern), and any related patterns. Most authors also include a rationale explaining how and why the pattern works. An example of this format for a specific pattern called the "Observer" pattern is found in Appendix A.

However, having a pattern is not enough. Given the increasing number of patterns, a means of classification is required in order to reduce the search time for an appropriate pattern. Unfortunately, just as there are many definitions of patterns, there are many classification schemes.

1.2 Pattern Taxonomies

The most well-known pattern taxonomy is the one proposed by the *GoF*[1]. There are three main classifications of patterns in the *GoF* literature. From low-level to high-level [69, 25], these consist of: idioms, design patterns, and architectural patterns.

An *idiom* is a programming language specific pattern. It explains a component's implementation or the relationship among components, using a language's given features. Nat Pryce refines this as "Idioms are language-specific in that the problem

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides wrote the seminal work, "Design Patterns". The patterns community frequently refers to them by the nickname, *Gang of Four*, abbreviated to *GoF*.

they solve, or the context in which those problems are encountered, are caused by the language" [94]. Some examples of programming idioms are: nested classes in C++, interfaces in Java, or counted pointers [2]. An example of a C coding idiom is `while(*dest++ = *src++);`, which copies the contents of one array to another [77].

A *design pattern* is not implementation-specific like an idiom. It allows the refinement of a software system's components or their relationship. By describing a frequently occurring structure of communicating components, it solves a general design problem within a particular context. As used by the *GoF*, the term design pattern (as opposed to pattern), has come to apply specifically to object and class relationships [77]. An example of a design pattern is the previously mentioned Observer pattern, which defines a one-to-many dependency among objects. It notifies and updates all registered dependents automatically when the dependent object changes state [47].

An *architectural pattern* specifies an application's fundamental structural properties. It also provides a set of predefined subsystems with known responsibilities and interrelationships. It is a high-level strategic pattern relevant to large-scale components and the system's overall mechanisms and properties. The Broker pattern is an example of an architectural pattern used in a distributed system with decoupled components that interact via remote service invocations. The broker component is responsible for communication coordination. Another example of a

[2]This idiom is used to simplify memory reclamation in the absence of garbage collection. A reference counter is introduced into an object body that counts the number of references to the object. The object is deleted when the counter reaches zero.

common architectural pattern is that of the Model-View-Controller (MVC) pattern. In the MVC pattern, an interactive system is divided into a model (core data and functionality), views (information displays for the user), and controllers (process user input). The user interface is formed by controllers and views. The MVC propagates changes to ensure consistency between the model and the user interface.

A *framework* conceptually fits in at the architectural pattern level; however, it is broader based than what is implied by an architectural pattern. Jézéquel et al. describe it best as:

> A framework is a reusable software architecture that provides the generic structure and behavior for a family of software applications, along with a context that specifies their collaboration and use within a given domain [69].

In other words, it is a reusable software architecture that acts as a template for a working application. The framework is customized by implementing or overriding the missing pieces, resulting in the final application. Unlike a class library, control flow is bidirectional between the framework and the application. It differs from a design pattern in that it focuses on reuse at the level of algorithms, implementation, and detailed design. In contrast, design patterns concentrate on reuse of recurring architectural design themes [33]. While frameworks consist of software, design patterns represent knowledge about software.

Wolfgang Pree adds the term *metapattern* to the classification terminology. A metapattern is "a set of design patterns that describes how to construct frameworks independent of a specific domain" [93].

1.3 Design Patterns

Of the three major classification levels, this essay focuses on design patterns. Hereafter, the term "pattern" means "design pattern". In addition, this focus is narrowed further to concurrent design patterns starting in Chapter 2. The remainder of this chapter examines design pattern categorizations to lay the groundwork for categorization of concurrent design patterns.

There are many design-pattern categorization philosophies. The most prominent is the *GoF* categorization. The *GoF* patterns are solely object-oriented patterns. These patterns are subdivided along two axes [47]. The first axis divides patterns into creational, structural, and behavioural patterns. The second axis consists of the the scope or granularity of the pattern in that it is applied at either the class or the object level. If a pattern is applied at the class level, it describes the relationship between the class and its subclasses. While this classification scheme is broad enough to fit the majority of design patterns without unnecessarily restricting them, it is limited to object-oriented patterns. It also suffers from the fact that it is difficult, when searching for patterns, to distinguish between behavioural and structural patterns. That is, when the pattern itself is unknown, it is unclear whether the pattern applies to classes or to objects [125].

Buschmann et al., in [25], extend the *GoF* pattern taxonomy by introducing a problem-oriented view of the pattern system. As before, the patterns are first organized into architectural patterns, design patterns, and idioms. Then, within each of these categories, patterns are loosely organized according to purpose. The purposes consist of: [3]

From Mud to Structure Includes patterns that assist in decomposing an overall system task into cooperating subtasks.

Distributed Systems Includes patterns that provide a foundation for systems whose components are located in different processes or in several components and subsystems.

Interactive Systems Includes patterns that provide a foundation for human-computer interaction systems.

Adaptable Systems Includes patterns that allow applications to adapt or extend themselves dynamically.

Structural Decomposition Includes patterns that assist in decomposing systems or complex components into cooperating parts.

Organization of Work Includes patterns that express how components interact to provide a complex service.

Access Control Includes patterns that protect and restrict access to services or components.

[3] The final four categories were added by Buschmann et al. to accommodate the remaining *GoF* patterns not handled by the previous categories.

Management Includes patterns that manage homogeneous collections (objects, services, or components) as a group.

Communication Includes patterns that assist in organizing communication among components.

Resource Handling Includes patterns that assist in managing shared components and objects.

Creation Includes patterns that assist in object instantiation and recursive object structures.

Service Variation Includes patterns that allow an object or components behaviour to change.

Service Extension Includes patterns that assist in dynamically adding new services to an object or object structure.

Adaptation Includes patterns that assist in interface and data conversion.

Unfortunately, this categorization scheme is difficult to work with [125]. Some patterns, such as the Broker or Observer, can be used as either an architectural or a design pattern. Additional problems result from the fact that some patterns fit under several different categories. For example, the Pipes and Filters pattern is placed under the Distributed Systems category when it could just as easily be used in a centralized system. This over-specification may lead a novice to not consider using a pattern outside of the listed category even when it is appropriate.

Douglas C. Schmidt classifies patterns into tactical and strategic[4] patterns [108]. Tactical patterns have a relatively localized impact on a software design and are domain-independent. For example, Singleton, Strategy, State, and Adapter are classed as tactical patterns. Strategic patterns significantly influence software architecture. Some strategic patterns are Acceptor, Active Object, Reactor, and Proactor.

Walter Zimmer [135] categorizes the *GoF* patterns by their relationships. The relationships are defined as "X uses Y in its solution", "Variant of X uses Y in its solution", and "X is similar to Y". He notes that even with this scheme, it is sometimes difficult to place a particular relationship in exactly one category. Another problem arises from the fact that the categorization depends partly upon subjective criteria. It also obscures the purpose of the design patterns, making it useful only for someone who is already familiar with the *GoF* design patterns.

James Noble provides a classification scheme similar to Zimmer's [87]. He chooses to divide patterns into primary and secondary relationships. The primary relationships consist of patterns which use other patterns, patterns which refine other patterns, and patterns which address the same problem as another pattern. The secondary relationships may be inverses of the primary relationships (patterns used or refined by other patterns), or new, complex relationships. These other secondary relationships include pattern variants, a pattern variant used by another pattern, similarity of patterns, combination of patterns to solve a problem,

[4] "Strategy: the art of projecting and directing the larger military movements and operations of a campaign. Usually distinguished from tactics, which is the art of handling forces in battle or in the immediate presence of the enemy." Oxford English Dictionary [116]

one pattern requiring a solution to another pattern, a pattern using itself, and an elaboration of a sequence of patterns from the simple to the complex. These relationships are an extremely useful taxonomy, but do not help a designer from the standpoint of determining which patterns are initially required to solve a problem.

Walter F. Tichy provides an alternative classification [125]. He divides patterns into the following problem-solving categories:

Decoupling These patterns partition a software system into independent components that are built, changed, replaced, and reused independently of each other. Abstract Data Types, Client-Server, and Encapsulation patterns are examples in this category.

Variant Management These patterns treat different objects uniformly by factoring out the common elements. The Superclass, Template, and Visitor patterns are examples in this category.

State Handling These patterns manipulate object state generically. The Singleton, Flyweight and Memento patterns are examples in this category.

Control These patterns control execution and method selection. The Blackboard, Strategy, and Master-Slave patterns are examples in this category.

Virtual Machines These patterns simulate processors. The Interpreter, and Rule-based Interpreter patterns are examples in this category.

Convenience Patterns These patterns simplify coding. The Convenience Class, Default Class, and Null Object patterns are examples in this category.

Compound Patterns These patterns are composed of other, visible patterns. The Model-View-Controller, Bureaucracy, and Active-Bridge patterns are examples in this category.

Concurrency These patterns control parallel and concurrent execution. The Semaphore, Critical Region, and Reactor patterns are examples in this category.

Distribution These patterns solve problems relevant to distributed systems. The Remote Procedure Call, Acceptor and Connector, and Broker patterns are examples in this category.

While these categorizations are not as limiting as those proposed by Buschmann et. al., it is unclear how patterns developed in subject areas such as artificial intelligence, or databases would necessarily fit these categories. As well, these categories are supposed to be mutually exclusive, but where would a software agent pattern be placed? It could be considered part of a concurrent or distributed system, or a form of decoupling. What about the pattern of an agent's interaction with both clients and servers? None of these categorizations truly address the qualitative aspects of patterns either. Though the relative advantages and disadvantages are listed, nobody distinguishes between a pattern and a *good* pattern. That is left up to the user.

Fundamentally, if the taxonomy is too broad, it becomes difficult to find the relevant pattern in the mass of other related patterns. Patterns are potentially misplaced or duplicated if the taxonomy is too narrow. On top of this, room for growth must be possible, as patterns in new areas are discovered.

Reasonable broad pattern categories are that of Creational, Abstraction, Decoupling, and Interaction. These patterns are classified objectively since they are the ones best defined in terms of purpose.

Creational patterns [47] abstract the instantiation process by encapsulating and hiding creation details. A system using these patterns only needs awareness of the specified abstract interface. The Abstract Factory, Builder, Factory Method, Prototype, and Singleton design patterns are all creational patterns.

Abstractional and Decoupling patterns often have some overlap since one means of decoupling components is to abstract them. Abstractional patterns encapsulate information, often reducing system complexity. The Facade, Strategy, Command, and Memento design patterns are examples of abstraction patterns.

Decoupling patterns partition a software system into independent components, which are built, changed, replaced, and reused independently. The Mediator, Iterator, and Bridge design patterns are examples of decoupling patterns.

Interaction patterns specify how system components interact and communicate. Note that there is some overlap with decoupling patterns since a decoupling pattern invariably specifies how the decoupled components interact. The Observer, Chain of Responsibility, and Proxy design patterns are examples of interaction patterns.

After this point, it is too limiting to try and group patterns by the problems solved. A more helpful approach is to group related patterns where possible and provide an overall road map.

Chapter 2

Concurrency Pattern Catalog

A **pattern catalog** *is a collection of related patterns (perhaps only loosely or informally related). It typically subdivides the patterns into at least a small number of broad categories and may include some amount of cross-referencing among patterns.*

A **pattern system** *is a cohesive set of related patterns which work together to support the construction and evolution of whole architectures. Not only is it organized into related groups and subgroups at multiple levels of granularity, it describes the many interrelationships among the patterns and their groupings, and how they may be combined and composed to solve more complex problems. The patterns in a pattern system should all be described in a consistent and uniform style and need to cover a sufficiently broad base of problems and solutions to enable significant portions of complete architectures to be built.* [13]

The remainder of this essay is best described as a pattern catalog, rather than a pattern system. Though some authors consider these patterns architectural in nature, I believe the appropriate context turns them into design patterns. Within the broad topic of concurrency, I divide design patterns into the following metapattern categories: synchronization, mutual exclusion, and client-server (see Figure 2.1). The client and server cooperate to perform a job using some form of synchronization or communication pattern. When there are multiple clients, access to the server requires mutual exclusion. This form of interaction is called *direct communication*. Synchronization and mutual exclusion may also be required for passive objects, through which clients and servers interact. This form of interaction is called *indirect communication*.

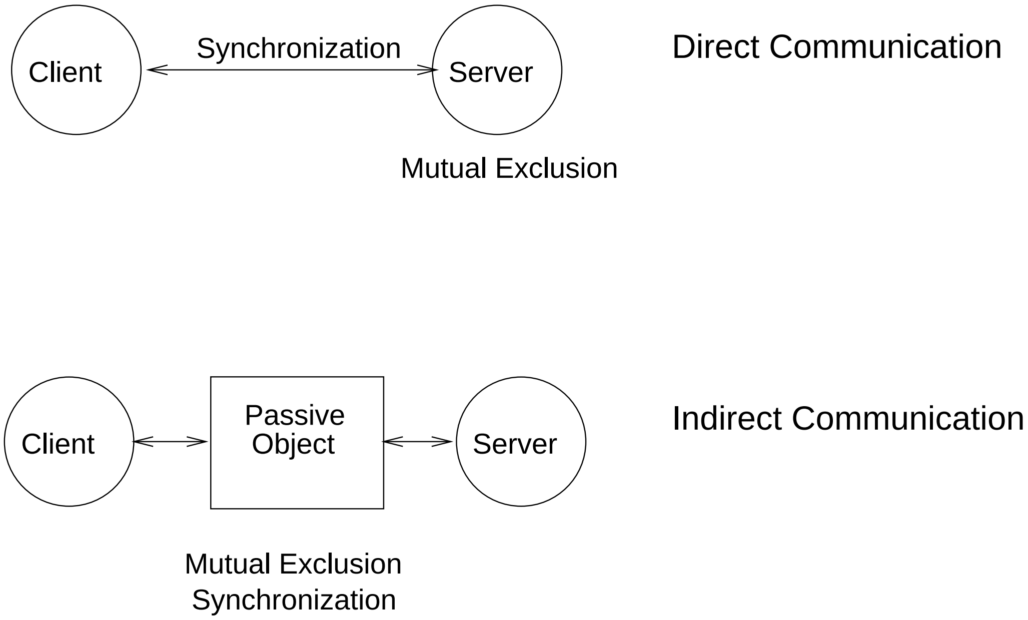


Figure 2.1: Fundamental Interaction Patterns

2.1 Synchronization

Synchronization occurs when a task[1] waits until another task reaches a particular point in its execution. The provision of synchronization allows communication. Communication may require mutual exclusion of a shared resource managed by a server or passive object.

Design patterns used to perform synchronization:

Completion Token In [56], the authors introduce a token-based synchronization design pattern called *Asynchronous Completion Token* or *Magic Cookie*. This pattern allows a client to determine that the server has completed an asynchronous action. The token is generally an opaque object passed to the server by the client and returned upon completion of the service. Another variant allows the token to act as a synchronous callback mechanism.

Services "Waiting For" blocks the tasks performing services until a condition such as data transferral occurs [3].

Rendezvous is a meeting or synchronization between two or more tasks at a prespecified piece of code [8, 113]. First proposed as a *binary rendezvous* between two tasks, this idea expanded later to a *multiway rendezvous*. A multiway rendezvous allows an arbitrary number of asynchronous tasks to rendezvous [28, 16]. In [49], a distinction is made between a *simple rendezvous* and an *extended rendezvous*. A simple rendezvous is a unidirectional exchange of

[1]Note that the terms "task", "process", and "thread" in the definitions are subsumed by the term "task".

information while an extended rendezvous (also referred to as a *transaction*) is a bidirectional transfer of information.

Remote Procedure Call (often referred to by the acronym RPC) allows a client to invoke the execution of an operation on a remote object as if it was a local object. Ideally, the action is transparent to the client, which is suspended until the action completes [8, 15, 113]. Tessier and Keller refer to this as a *Remote Operation* [123]. Doug Lea's *Request Object*, a message containing an encoding of a method name and marshalled arguments, is an RPC mechanism [76]. Burns and Davies talk about a *remote invocation* model [22].

There are many possible mechanisms for implementing RPC. In [62], a Remote Proxy (Proxies are discussed in Section 2.3.1.1) is used on the client side to perform RPC transparently.

Delta Prolog uses an *event goal* to ensure both communicating tasks are "locked together" before the message is transferred [35]. Thus, an event goal provides synchronization. Both the sender and receiver must use the same event goal name, and if using conditional expressions, they must both evaluate to true. Since a sender can only rendezvous with one receiver at a time, an event goal provides mutual exclusion on the communication channel between the sender and the receiver by keeping a third task from interfering with the event goal. (The initial paper on Delta Prolog notes that communication was implemented via mailboxes on a VAX/VMS system, while sockets were used under a UNIX 4.2 system [91].)

Termination Synchronization is one of the simplest forms of synchronization. A thread is spawned in order to perform work. The results are guaranteed to be complete upon the thread's termination, so the parent thread need only wait until the child thread has ended. Doug Lea refers to this as *thread join* [76]. Massingill, Mattson, and Sanders describe this as the *ForkJoin* pattern [81].

Object Synchronization Pattern This design pattern decouples object synchronization from object concurrency and functionality, allowing different synchronization policies to be implemented as required. Synchronization mechanisms are encapsulated with the object, rather than distributed among the client tasks. Since they are abstracted, they are easily replaceable. This approach allows new policies to be tested and their performance observed before a final solution is chosen. The synchronization policies are also separated from the concurrency policies, allowing the concurrency policies to be modified without affecting the synchronization policies. Based upon the chosen policies, the Synchronizer schedules calling tasks as appropriate.

A similar idea is the *Object Synchronization Pattern*, also known as *Object Concurrency Control*, or *Object Serialization* [101].

Completion Callback A client sends a one-way message to a server. When the server has completed the operation, it sends a one-way *callback* message to the caller [76]. It may have the same structural design as an Observer. This pattern is also known as *Self-Addressed Stamped Envelope*, *SASE*, and *Callback* [12]. In [36], a Callback variant is mentioned known as *Named Reply*.

Here, labeling results returned from a call allows expression of different kinds of results. For example, based upon a calculation's result, the server invokes either a `success` or a `failure` method.

Future According to Gregory V. Wilson,

> *A future is simply a commitment by a process to use the result of a calculation at some later date. When a future is evaluated, a new process is created; when the parent of that process tries to read the future's result, it is automatically suspended until the child has returned a value. It is the runtime system's responsibility to decide whether to execute a future in parallel with its creator, or to use a lazy evaluation strategy, which only calculates values when needed* [132].

Other authors view a Future simply as an object (rather than a task) acting as a virtual representation of the real object. If the data is accessed before it is filled in, the user blocks until the data is filled in. These two definitions represent two different implementation patterns of a Future, one by Lazy Initialization and one by Virtual Proxy [54]. Futures are also known as *promises* in RPC [14], *early reply* [49], an *Active Object* [124], *IOU*s [126], and *wait-by-necessity* constructions in Eiffel// (a parallel extension to Eiffel) [76]. James Noble provides two variants of his Result Object, both of which can act as a future. One is the *Future Object*, the other is the *Lazy Object* [86].

While not intended to act as a Future, Hoffert's *Triggered Placeholder* (or

Stub) is a creational pattern allowing the delay of an object's creation until a particular method on the object is invoked. In the meantime, a temporary placeholder is returned to the invoker. The trigger method returns the real object and deletes the placeholder.

Synchronization is also a requirement for communication since two tasks must coordinate the information transfer.

2.1.1 Communication

While communication by itself is not a design pattern, there are commonalities in form and use that appear frequently enough to be termed a pattern. A stronger argument for the inclusion of this section is made by noting that many of the standard design patterns rely not only upon a task hierarchy, but also upon the communication flow for the design pattern definition.

Communication is first categorized by the number of participants (see Table 2.1[15, 131, 115]) and direction of information flow (see Table 2.2 [51, 129]). It can also be divided into asynchronous versus synchronous communication.

2.1.1.1 Number of participants

Wilson, in [132], refers to a broadcast as a *one-to-all replicated communication*, a useful distinction since he also describes a *scatter*, also known as a *one-to-all personalized communication*. Here each message type is the same, but the data sent is different.

Receive From	**:**	**Send To**	**Description**
1	:	1	Also known as "one to one" communication. The Pipes and Filter design pattern is an example of this form of communication.
1	:	N	Also known as "one to many" communication. If the set of recipients is restricted, this form of communication is termed a *multicast*. It is termed a *broadcast* if "everybody" is sent the message. When the Observer pattern communication changes to more than one registered recipient, it is an example of this form of communication.
N	:	1	Also known as "any to one" communication. Most Client-Server interactions are an example of this form of communication since there are usually multiple clients requesting a service from a server.
N	:	M	Also known as "many to many" communication. The Client-Server pattern also fits this form of communication when each client broadcasts for the first available server.

Table 2.1: Patterns of Participant Numbers in Communication

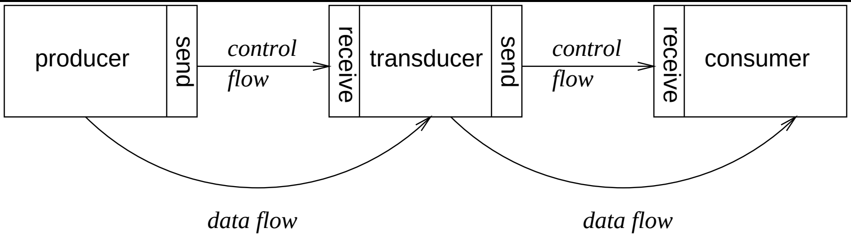
Bobby Woolf provides a pattern for broadcasting or multicasting messages, though it has other purposes. Object Recursion, also known as Recursive Delegation distributes the processing of a request over a structure by polymorphic delegation [134]. In this fashion, it can broadcast a message to all nodes in a linked structure.

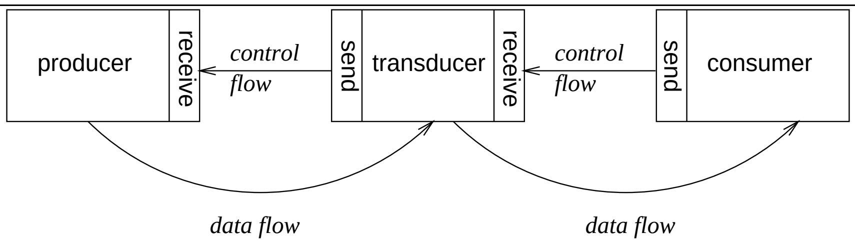
2.1.1.2 Direction of Information Flow

In [51], all four basic flows are presented, but it is carefully pointed out that inward- and outward-directed communication are not commonly used due to the inefficiencies introduced by the polling or blocking characteristics of most operating systems. In general, push and pull models of communication data flow are the ones most frequently discussed since they cover the standard means of transferring information. However, [43] presents several other interaction patterns, including the push and model models:

Round Robin Polling uses the pull interaction pattern to poll multiple devices in sequence. It requires that the data providers are trustworthy and fault-tolerant.

Opaque Interaction Patterns are patterns in which the caller is unable to monitor the callee's progress. These patterns are subdivided into synchronous and asynchronous versions. In Synchronous Opaque Communication, the caller waits until callee returns control. In Asynchronous Opaque Communication, the caller is allowed to continue with other work while the callee may or may

<table>
<tr><th>Direction</th><th>Description</th></tr>
<tr><td>Forward</td><td>

Also known as process driven communication or the push model.</td></tr>
<tr><td>Backward</td><td>

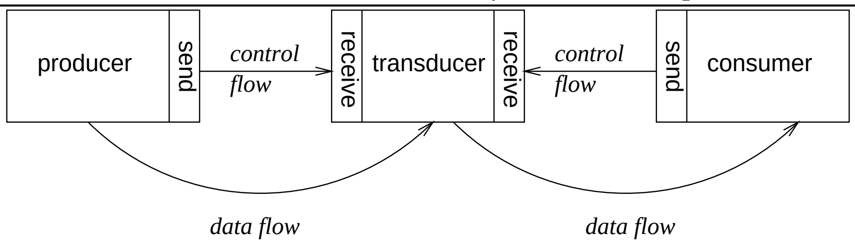
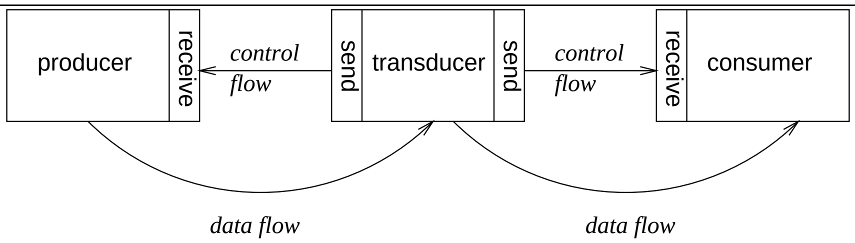
Also known as data driven or call by need or the pull model.</td></tr>
<tr><td>Inward</td><td>
</td></tr>
<tr><td>Outward</td><td>
</td></tr>
</table>

Table 2.2: Patterns of Data Flow in Communication

not handle the message asynchronously.

Monitorable Interaction The caller is informed of the callee's progress if the execution time varies widely. The caller can either request the current status (Pull-Monitorable Interaction pattern) or the callee can send periodic status reports (Push-Monitorable Interaction pattern).

Abortable Interaction Either party involved in the communication (or both) has the capability of deciding to abort the message processing. Variant interaction patterns such as Abortable Asynchronous Opaque, Abortable Monitorable, Abortable Pull-Monitorable, and Abortable Push-Monitorable also exist.

Handshaking interaction patterns break a large amount of data into a series of messages. The caller controls the exchange (Caller-Controlled Handshaking), or the callee (Callee-Controlled Handshaking) controls the exchange, or both control the exchange (Dual-Controlled Handshake). The controlling party decides when the sequence of messages is terminated.

Andrews provides three communication patterns for interacting peers: centralized, symmetric, and ring [8]. In the centralized approach, a task acts as a central coordinator for the other tasks. All tasks in the symmetric communication pattern perform the same algorithm, and thus communicate in the same symmetric pattern. Finally, in a ring, each task receives a message from its predecessor and sends a message to its successor. The last task acts as the predecessor to the first task. Thus, the centralized approach is a form of inward-directed communication. The other two forms rely upon the push and pull models of interaction. Interacting peers are used to solve iterative parallel problems such as exchanging values to determine the maximum and minimum values, and matrix multiplication. The authors of [31] list a *Peer-Peer* pattern used in flight control systems to determine flight path intersections.

Andrews also presents the heartbeat, pipe, and probe/echo interaction patterns [7, 8]. In the heartbeat pattern, processes occasionally exchange data using a send and then receive interaction. This pattern is particularly useful when data is divided among workers who are responsible for updating specific sections of data where new values depend upon values held by either the workers or their immediate neighbours. These problems arise in areas as diverse as parallel sorting, matrix multiplication grid computations or region labeling in image processing, cellular automata in simulation of biological growth, or solving partial differential equations.

The pipe interaction pattern incorporates the notion of streams [15, 18, 127, 115]. In the pipe interaction pattern, data flows between processes using a receive and then send interaction. The pipe is open-ended, closed, or circular in nature (see Figure 2.2). In an open pipeline, the input source and output destination are not specified. A closed pipeline is an open pipeline connected to a coordinator providing the input for the first task and receiving the output of the last task. A circular pipeline has the last task acting as the source of input to the first task. This form of interaction occurs in areas such as UNIX pipes, prime number generation (the Sieve of Eratosthenes), and distributed vector or matrix multiplication.

Finally, in the probe/echo pattern, the probe is equivalent to a send, while the echo is equivalent to a receive. This disseminates and collects information in graphs

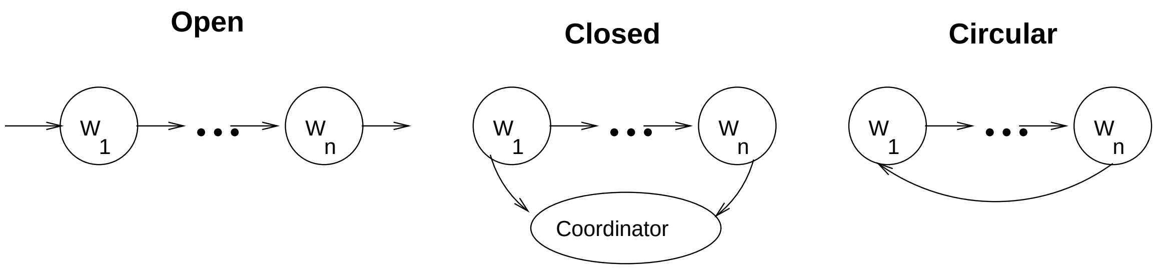


Figure 2.2: Pipeline Forms

and trees, acting as the concurrent analog of a depth-first search. This pattern is used in broadcasting to all nodes in a network, and in constructing the topology of a network. Grady Booch provides two additional forms of cooperative communication techniques, remote procedure calls (discussed as a synchronization technique) and client-server interaction (see Section 2.3) [18]. In [115], Shaw encapsulates these patterns in her architectural pattern, Communicating Processes, since it does not specify the particular communication topology, delivery requirements, number of participants, or synchronization.

2.1.1.3 Asynchronous versus Synchronous Communication

There are two main forms of communication, asynchronous and synchronous. In asynchronous communication, the sender does not block upon sending a message, unlike synchronous communication. The message's recipient blocks until the message is received.

An asynchronous message passing pattern is the Batch Communication Style since it allows both the sender and receiver to be asynchronous. Here the message from the sender is stored and forwarded to the receiver [131]. Doug Lea lists polling (repeatedly querying some condition) and balking (refusal to proceed if some precondition is not met) as two useful communication techniques [76]. Strangely enough, Mark Grand has promoted the balking technique from a communication technique to a concurrency design pattern [53].

Events, as well as messages, can act as a form of communication [76]. Aarsten, Elia, and Menga refer to this as *Actions Triggered by Events* [3]. Tichy documents the *Event-Based Integration* design pattern allowing participants to register interest in common data or event channels [124]. Shaw discusses a similar architectural pattern, *Implicit Invocation*. The Implicit Invocation pattern allows a collection of acting tasks to potentially trigger the execution of other tasks based upon the notification of system events [115].

2.1.1.4 Communication Simplification

Finally, here are some patterns used to simplify communication.

Distributed Symmetric IPC encapsulates the usual means of connecting two peers, via one peer making an accept call while the other peer makes a connect call, into a single connect call. The calling task can ask for either a connection to a specified task, or to all tasks in a list. The pattern uses the Client-Server Rule to decide which task of the communicating pair takes the role of Client, and which takes the role of Server. No assumption is made about the order of task start-up. The Client-Server Rule assigns the roles in such a fashion as to avoid role conflicts. Iterative back-offs are used when making the connections, until either all connections are made or a timeout occurs.

This pattern allows a network of peer tasks to connect with each other, without using a communication server. The authors, however, are quick to point out that more realistically, one task acts as a master, directing and initiating the connections. The master is then responsible for providing fault tolerance for the connections.

Composite Message is a decoupling pattern. It defines an abstract form of a message and a protocol for communication between tasks and the base system. In this fashion, it decouples tasks that should be loosely coupled and localizes task interactions for tightly coupled tasks [105].

Command wraps a request into an object passed to the server [47]. The Command can be stored in a history list or manipulated in other ways. A Command is an object-oriented replacement for a Callback. Since its lifetime may be independent of the original request, it can be transferred to a different task for processing.

Eskelin provides a variant known as *Interruptible Command* or *Override Current Processing* [42]. Based on Command, it provides "a mechanism for efficiently interrupting a currently executing command in favor of the execution of a new command." In particular, only the most recently requested command executes when multiple commands are invoked. The command itself occurs frequently, and takes a great deal of time to process.

Composite Call is similar to Command. The pattern collects multiple operations to be performed into a single action performed by another task (which may be located in another process, or another network node). This pattern contains the server interaction in the sense that it knows the actions the server must perform. Though it may contain several operations, the client need only send it once [90], which improves the system efficiency by reducing the amount of communication required. It can also allow the server to be dynamically extended in terms of the service provided. For example, consider a file server performing both reads and writes. It is more efficient for the client to send the entire loop that executes the reads and writes over to the server for processing than make the requests individually.

2.2 Mutual Exclusion

Mutual Exclusion is the prevention of multiple tasks from accessing a critical section at the same time. A critical section consists of a pairing of code and shared data.

Patterns with the intent of providing mutual exclusion are:

Synchronizer provides mutual exclusion on shared resources for multiple tasks [60].[2] Mutual exclusion is provided via a construct called a critical section[3], guarded methods, or mutual exclusion techniques such as transactions.

[2] Within the paper, the authors use the term synchronization when the context clearly indicates that the intended meaning is the provision of mutual exclusion. Hence, the design pattern misnomer.

[3] Historically, this construct is called a "critical region" to differentiate it from the logical notion of a critical section [20].

Mark Grand lists a variant, the *Single Threaded Execution* pattern, that is defined as providing mutual exclusion on method calls when concurrent invocations are made [53].

Michel Raynal mentions the idea of a *central coordinator* [97]. The central coordinator is the boundary between the two basic forms of distributed mutual-exclusion algorithms in his taxonomy: permission and token-based algorithms. Processes ask the coordinator for permission before entering the critical section, and permission can be integrated into a token managed by the coordinator. The *Lock Server*, also known as a *Lock Manager*, works in the same fashion [63].

While the previous patterns constitute forms of critical region, the next variants of the Synchronizer are closer in form to a conditional critical region[4]. In [55], the Synchronizer selects all requests satisfying its synchronization constraints and assigns each of them to a thread, using either a thread-per-request or a thread-per-object policy.[5]

The Actor model also uses a Synchronizer, a special kind of actor that observes and limits invocations accepted by a group of actors [5]. It synchronizes a group of actors by delaying the invocations of shared actors until the specified restrictions (temporal ordering, as well as atomicity) are met.

Double-Checked Locking pattern provides mutual exclusion in a thread-safe

[4] A *conditional critical region* provides mutual exclusion like a critical region, but adds the capability of specifying a condition that must be true before entry to the critical region is allowed.

[5] *Thread-per-request* allocates a thread to every request. *Thread-per-object* allocates one thread to every object receiving requests, thus all the requests to one object are serialized.

manner while attempting to reduce contention and overhead [111]. For example, it can avoid duplicating the initialization of an object by multiple threads. This occurs when one thread recognizes that the object requires initialization but another thread has already started initialization. Both the Lazy Initialization and Virtual Proxy patterns can use it [54].

Local Serialization Pattern serializes access to shared resources having coarse-grained operations. Object concurrency control policies are decoupled from object-specific algorithm semantics and concurrency generation policies. It is also known as: *Critical Section*, *Local Atomicity*, and *Object Concurrency Control* [102].

Patterns used to implement mutual exclusion:

Transaction is an operation, usually a composite of several actions, that must be performed *atomically*, i.e., uninterrupted by the actions of other tasks. Grasson presents a variant of his Synchronizer pattern that uses Transactions by adding a Coordinator object to enforce two-phase locking and transaction identifiers associated with object requests [55]. In [60], three forms of transactions are listed:

Optimistic Transaction aborts the operation if serializability is compromised. No blocking is performed. This approach should only be used when other techniques (sharing policies, mutual exclusion, etc.) avoid conflict, or contention is low.

Two-Phase Locking Transaction locks a resource when it is read or written. If more than one resource is to be modified, multiple locks may need to be acquired in the lock collection phase. If the lock is only released when the transaction is completed, deadlock may result. This problem can be avoided by releasing the lock and restarting the lock collection phase when it is discovered that a required lock has already been obtained by another task.

Multiversion Two-Phase Locking Transaction copies the resource before it is updated. All updates are consolidated such that serializability is preserved when the transaction successfully terminates.

In [49], a Transaction is also described as an *extended rendezvous*.

Lock Patterns provides some simple locking patterns, each designed to balance the forces of memory latency, memory size, memory bandwidth, granularity, and fairness in different fashions [83].

Test-and-Set Lock uses a test-and-set based locking primitive when contention is low, fairness and performance are unimportant, or memory size is a limiting factor.

Queued Lock uses a queued-lock primitive to solve the problem of high contention and meet a strict fairness restriction.

Reader/Writer Lock deals with a classic problem in concurrency. Multiple tasks can read a shared resource, or a single task can write to it. The read and write operations must be mutually exclusive. This problem is a restricted form of *Parallel Fastpath*, where an aggressive locking pattern is used for the majority of the work (the Fastpath), and a more conservative locking scheme is used for the remainder.

Three possible types of locks are presented:

Queued Reader/Writer Lock uses a queued-reader/writer-lock primitive. It solves the problem of moderate to high read-to-write ratio, a high degree of contention, and where fairness is important.

Counted Reader/Writer Lock uses a counter-reader/writer-lock primitive. The lock maintains the cumulative number of requests and completions for the readers and writers. Each requester must remember the current number of requests, increment the appropriate request counter, and then wait for all prior conflicting requests to complete. Readers wait for all prior write requests to complete, while writers wait for all prior requests to complete. It solves the problem of moderate to high read-to-write ratio, a high degree of contention, and coarse-grained parallelism.

Distributed Reader/Writer Lock uses a per-CPU lock for readers and an additional lock to serialize writers. A reader acquires only its own CPU lock, while a writer must acquire the writer serialization lock as well as each of the reader-side CPU locks. It solves the problem of a high read-to-write ratio, and high read-contention.

Douglas C. Schmidt offers a *Strategized Locking* pattern to provide mutual exclusion; but, it allows an application or service to configure the implementation choice.[109]

Some authors classify the mechanisms used to provide mutual exclusion and synchronization as design patterns. For example, Douglas C. Schmidt lists a design pattern for a *Monitor Object*, also known as a *Thread-safe Passive Object* [110]. While it is true that some of these can be implemented in multiple fashions, and thus they could be considered abstractions, they do not qualify as full-fledged design patterns.

2.3 Client-Server Patterns

Client-Server is the most basic design pattern in this category [51, 131, 133, 9]. Since Clients and Servers are simply roles, tasks may sometimes act as a Client, and sometimes act as a Server. A Client makes a request of a Server.

Under the division of client-server design patterns can be found client- and server-side patterns. Note that patterns on the client-side do not affect the server, and vice versa.

2.3.1 Client-Side Patterns

On the client-side, the design patterns interpose an intermediary between the client and the server (see Figure 2.3). The presence of the intermediary may be transparent to the client (Proxy and related design patterns), or not (Mediator and related design patterns). From the view-point of the server, the intermediary is just another client.

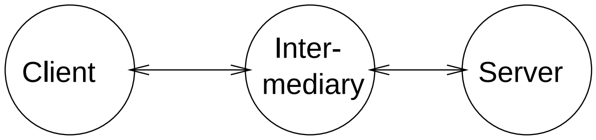


Figure 2.3: Client-Side Design Patterns

2.3.1.1 Proxy

The most basic form of transparent intermediary is that of the *Proxy* [103]. As the name suggests, it represents an object or task to the user. Thus, it must present the same interface as the original. James O. Coplien's *Handle-Body* idiom underlies the Proxy pattern since it adds a level of indirection to hide the underlying details, separating the interface of a class from its body. The "handle" is the proxy, while the "body" is the underlying object [32].

The Proxy design pattern exists under many different names, and with many variations. The *GoF* calls this a *Surrogate* [47]. Wolf and Liu refer to it as a "ghost pattern" [133]. Doug Lea notes that a Proxy is a variant of the *GoF* Adapter[6] pattern, where the Adapter has the same interface as its delegate [76]. There are several notable Proxy variants. The *GoF* list virtual, remote, and protection proxies [47]. Hans Rohnert adds the cache, synchronization, counting, and firewall proxies [103].

Virtual Proxy allows lazy construction of an object. When processing or loading a component is costly, it is only performed upon demand. The Virtual Proxy

[6] An Adapter converts the interface of a class into an interface clients need [47].

hides whether or not the component is fully loaded from the client, and loads as necessary. It can thus implement *Futures*.

Cache Proxy allows multiple local clients to share results from the outside by extending the proxy with a data area to temporarily hold results. The cache must be maintained and refreshed.

Remote Proxy provides a local representation for an object in a different address space. The request's arguments are packaged into a message and transmitted transparently to the "real" body class in the foreign name space. Coplien refers to this type of proxy as an *Ambassador* [32]. Heuser and Fernandez describe the *RPC Client* design pattern as an elaboration of the Remote Proxy. Marquardt presents the *Transparent Remote Access* design pattern [80]. Buschmann and Meunier list the *Proxy-Original* design pattern [24].

Protection Proxy protects the original from unauthorized access by checking the access rights of every client. A similar pattern is the *Authenticator*, which can provide a negotiation protocol as part of the access protocol [21].

Synchronization Proxy (which should more properly be called a Mutual Exclusion Proxy) controls multiple simultaneous client accesses using an appropriate mutual exclusion scheme, depending upon the allowed operations.

Counting Proxy maintains the number of references to the original object and deletes the original when the count reaches zero. Thus, it automatically deletes obsolete objects.

Firewall Proxy encapsulates the protection and networking code needed to communicate with a potentially hostile environment.

A Proxy can also be used as a *Gateway* [75]. A Gateway serves as a midpoint between client-controlled and server-controlled activation policies, repackaging a set of methods split into different services.

2.3.1.2 Mediator

Another form of intermediary design pattern is the *Mediator*. A Mediator is a task that encapsulates, controls, and coordinates the interactions of a group of tasks [47]. Since all tasks in the group now only need to know about the mediator, instead of about each other, the number of interconnections is reduced. By placing a mediator between peers, loose coupling is increased [100]. Name servers and relays are examples of Mediators [36]. By this definition, an Administrator is also a Mediator (Administrators are discussed in Section 2.3.2.2). The following list describes a number of variants on the Mediator design pattern.

Courier assists communication between two administrators so that neither blocks waiting for a message receipt or reply [50]. In [36], the authors discuss a *Waiter* allowing the invoker to continue without blocking while the waiter blocks for it. A similar notion is that of the *Forwarder* [22], also known as the *Forwarder-Receiver* [25]. A Forwarder is an intermediary between two tasks. It acts as an agent for the client, blocking on the client's behalf if the contacted task is not ready to receive the call. In [59], the authors use what they call a Courier design pattern to solve the problem of passing arbitrary requests and information through a fixed interface. Information is now packaged as an object itself and passed as an argument among tasks, through the Courier intermediary. By expanding the types of messages sent, interfaces need not be changed as often. As well, other message-sending strategies such as broadcasting can be implemented.

Gateway a Mediator that decouples cooperating peer tasks throughout a network. In this fashion, they interact without having direct dependencies on each other.[7]

Emissary represents the client task to the server task. It is chosen by the client who configures it to react to events that occur while the server processes the client's request. The Emissary can interact with the client to obtain further information as necessary upon receiving messages from the server [52].

Mediator-Worker uses the Mediator to decouple cooperating Worker tasks, thus removing direct dependencies. It can also be used to present a front for more complex functionality by allowing individual services to be combined [24].

Event Channel mediates among event producers (referred to as suppliers) and event consumers. This approach allows a supplier to deliver events to one or more consumers without requiring any of the participants to be aware of each other [96].

[7]Schmidt's ACE project papers.

Shopper design pattern allows a consumer to obtain an arbitrary number of items from a producer without additionally coupling them together. The consumer passes a list of objects to the Shopper who obtains the objects from the providers using some sort of selection strategy. The information on how to locate or rendezvous with the producers may be provided by the consumer, the producer, or the Shopper [38].

Data Filter filters client requests in a distributed system, according to predefined policies maintained in some sort of policy or client database. The filtering can be performed locally or remotely [46]. Used in combination with the Bodyguard, Authenticator, and RPC Client design patterns, an *Object Filter and Access Control* pattern can be constructed. This pattern provides registered clients, through a variety of network protocols, with a filtered data stream whose content may be sensitive and require access control [58].

2.3.1.3 Broker

The Broker design pattern decouples clients and servers. It is classified by [25, 120] as an architectural pattern for use in a structured distributed system where remote invocation is the main means of interaction. Servers register with the broker. Clients access the servers by sending requests to the broker who locates the appropriate server, forwards the request to it, and transmits the results as well as any exceptions back to the client. An example of use is the CORBA (Common Object Request Broker Architecture) Object Request Broker.

Stal lists five Broker implementation variants:

Direct Communication Broker System allows the client to communicate directly with the remote broker instead of passing the request to a local broker who is then responsible for forwarding it. Another possibility is that clients communicate directly with the server once the broker notifies the client of the available server communication channel.

Message Passing Broker System passes on messages from the client to the server. The server uses the message type to determine the service to perform. This approach is used in systems where message passing is used instead of Remote Procedure Calls.

Trader System allows the broker to determine which server or servers can provide the request, instead of forwarding the client request to exactly one uniquely identified server.

Adapter Broker System uses an adapter layer to hide the broker's interface. This layer is responsible for registering and interacting with the servers. Allowing multiple adapter layers enables different server implementation strategies.

Callback Broker System causes the broker to be the driving force in the system. When an event arrives, the broker invokes the callback method of the appropriate client or server. Thus, there is no need to distinguish between the clients and servers.

Olson describes four Broker variants [100].

Transceiver-Parcel is designed for a peer-to-peer interaction rather than client-server. The broker is deliberately kept as simple as possible, only aware of "parcels" (a parcel contains whatever method calls are needed to cause the receiver to do the bidding of the sender). As well, all tasks in the system (including the broker) should use the same communication method, which is deliberately kept simple. Upon receipt of a parcel, the broker notifies the receiving task by invoking its execute method, passing the address of the parcel as an argument. The receiver then invokes the visit method on the parcel.

Going Postal is similar to the Transceiver-Parcel in that the broker is kept as simple as possible. Decoupling, flexibility, and extensibility are essential, though efficiency is not considered crucial. The broker is now responsible for registering tasks, receiving parcels, and routing them as appropriate. It uses a registrar object and a routing object to accomplish its duties. This pattern is also known as *Broker as Intermediary*.

Going to Court is used when the broker application is distributed across processes and/or processors, but how it might be distributed may vary. In effect, every task uses at least one broker for communication. Each broker should have a proxy for each type of task it has in its own address space. The proxy is responsible for marshalling parcels routed to it and forwarding them to a gateway that reconstructs the original parcels and routes them to its local broker. This pattern is also known as *Broker as Divorce Attorney (In a Really Ugly Divorce)*.

Going to the Chapel is used when the basic broker architecture is too inefficient. Now the broker serves to "introduce" the two tasks who then communicate directly. The broker is notified when the communication is complete. This pattern is also known as *Broker as Matchmaker*.

Schneider discusses a similar idea, the *Matchmaker* [113]. The Matchmaker acts as a clearing house to pair up asynchronous request and reply messages to implement synchronous communication. Andrews refers to the Matchmaker as a *Centralized Clearing House* [7].

In [78], the authors refer to another form of Matchmaker, the *Patch Panel*.

Some other variants of the broker are:

Dispatcher provides location transparency in a distributed environment through use of a name service. The Dispatcher hides communication connection details between client and server [119, 25].

A similar idea is presented in [31] as *Sender-Pass Through-Receiver*. This pattern passes the Sender's request through the intermediary, which forwards it to a Receiver.

Manager-Agent pattern is similar to the Broker pattern, in that managed resources are grouped into agents that are accessed by managers who perform management operations. Agents are responsible for monitoring their resources and notifying managers of exceptional behaviour. Unlike the Broker, the Manager-Agent does not have the concepts of service location and transparency provided by the Broker. Additionally, either party can initiate communication asynchronously [123].

Entity Broker [128]. The authors use it to mediate between the user interface, business object, and persistence manager layers in an application.

Switchboard manages connections among clients and device couriers. The client requests a connection to a particular device from the Switchboard. The Switchboard associates the appropriate device courier with the client. It then transfers information between the client and the device courier. The device courier obtains input from the device server [122].

2.3.1.4 Other

Other intermediary patterns include:

Curried Object to store the constant or slowly varying arguments from the original communication protocol. This approach provides a simpler protocol since these arguments are eliminated. The Curried Object stores the original server object, and forwards messages to this object. In the forwarding process, it passes along the stored arguments and updates the slowly varying arguments [86].

Facade provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use since an object represents many others. It differs from a Mediator in that it abstracts a subsystem of objects to provide a more convenient interface and its protocol is unidirectional [47]. A Facade makes requests of the subsystem classes but not vice versa, unlike a Mediator.

Mailbox is a task that acts as a temporary buffer between two tasks. It allows the active process to pass data asynchronously (via the mailbox). If the buffer is full or empty, attempts to add or remove messages cause the invoker to block [22].

Warden mediates between proxies and transporters in a distributed environment. It "simplifies the management of object sharing over a network, and provides message dispatching conformance and assignment of access rights in non-local environments, to prevent the incorrect access to an object in collaborative applications" [34]. Hays, Loutrel and Fernandez also refer to this pattern as a *Bodyguard* [58].

Router decouples input mechanisms from output mechanisms. This decoupling enables it to route data correctly without blocking a Gateway and allows it to customize its concurrency strategies [107].

Proactor simplifies asynchronous application development. It integrates the demultiplexing[8] of completion events and the dispatching of the corresponding event handlers. These are decoupled from the services performed in response to events. Its pro-active event dispatching model allows multiple concurrent

[8]In electrical engineering, a demultiplexer is a circuit that receives information on a single input line, and transmits this information along one selected output line. The corresponding multiplexer selects input from one of many input lines and directs it to a single output line. As used in this pattern, the demultiplexer receives (and probably serializes) multiple simultaneous events or messages and dispatches them as appropriate.

events to be started so that the thread performing the operation is outside of the application; hence, the application is not required to have multiple threads. It invokes event handlers defining completion hooks. The Proactor pattern is used in such places as the ACE project, and I/O Completion Ports in Windows NT. It is related to both the Reactor and Observer design patterns, though the Reactor is an alternative concurrency approach [95, 108, 107].

Reactor serializes event handling from multiple sources within an application at the level of event demultiplexing. This approach allows single-threaded applications to wait on event handles, demultiplex events, and dispatch event handlers efficiently. It invokes event handlers defining initiation hooks. The event handlers must exchange messages fixed or bounded in size without requiring blocking I/O and the messages must be processed in a relatively short period of time. The Reactor design pattern can eliminate the need for more complicated threading, synchronization, or locking within an application. It is related to the Observer design pattern and similar to Factory Callback though it is behavioural in nature instead of creational [108, 107, 123].

In some sense, a *Demon* may be considered a form of Reactor. A Demon is "A portion of a program that is not invoked explicitly, but that lies dormant waiting form some condition(s) to occur." Unlike a *Daemon*, a Demon is usually a process within a program rather than a program in an operating system. Demons are frequently used in artificial intelligence program. For example, demons might implement inference rules in a knowledge-manipulation program. As information is added, the demon appropriate to the type of information activates and creates additional information by applying its inference rules to the information [10].

An *Active Object* can implement the Reactor. An Active Object enables a method to execute in a thread of control separate from the one that originally invoked it. This fact requires the implementation of a rendezvous policy. This design pattern is also known as a *Concurrent Object* or a *Serializer* [74]. Rumbaugh et al. state that "an actor is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph. In a sense, the actors lie on the boundary of the data flow graph but terminate the flow of data as sources and sinks of data, and so are sometimes called *terminators*" [104]. In [75], Doug Lea states that "Listener-based objects are also sometimes called Reactors, Object Adapters, Guardians, Skeletons, Executives and Demultiplexers." He notes that they may also server as *Parsers*, and *Builders*. He adds in [76] that an Active Object is also known as an *Actor*. The authors of [73], however, point out that the concept of Actors is more general, as originally envisioned by Carl Hewitt and later expanded on by Gul Agha. Here, Actors consist of a society of cooperating agents who communicate by asynchronous message passing. Thus, an Active Object is only one possible implementation of an Actor [5].

Process Control regulates a physical, continuous process. Input comes from process variables, input variables, manipulated variables, and sensors. Because the controller is now decoupled from the process, it can be easily replaced [124].

2.3.2 Server-Side Patterns

Historically, Gentleman, Shepard and Thoreson list two basic forms of servers, a *Proprietor* and an *Administrator* [51] (see Figure 2.4).

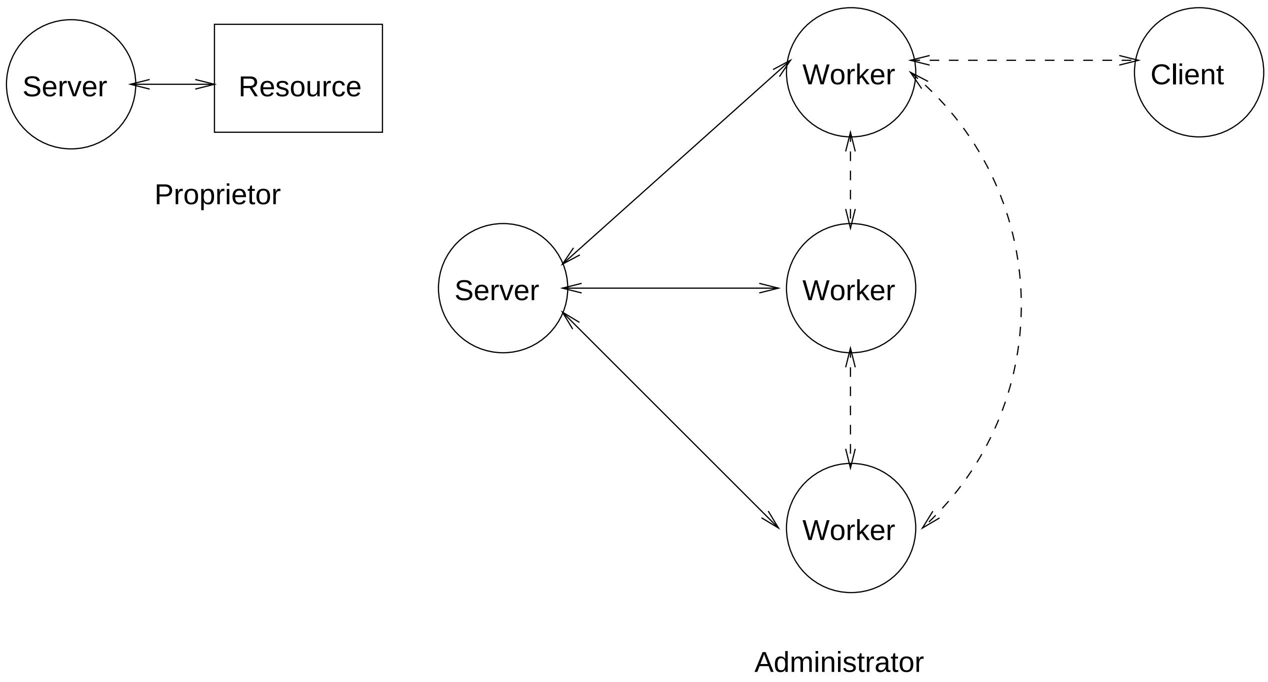


Figure 2.4: Server-Side Design Patterns

2.3.2.1 Proprietor

A Proprietor owns and manages some resource (Andrews and Schneider refer to a similar idea as a *Caretaker* in [9]). The only way other tasks can perform operations on the resource is to request that the Proprietor perform the operation on their behalf. The Proprietor thus provides mutual exclusion on the resource, and possibly some form on synchronization. There are many variants on the theme of a Proprietor design pattern:

Leasing manages resources in a fault-tolerant distributed system. Clients request access to the resource for a finite period of time. Once the granted lease expires, the corresponding resource is freed. The holder of the lease is able to request a lease extension if the lease has not yet expired. It can also cancel a lease once it has finished with the resource [67].

Lock Server provides mutual exclusion in a distributed system. It allows each client to work with a consistent view of the shared resource since each client must obtain a lock for the resource before proceeding, and must return the lock upon completion. It is also known as a *Lock Manager* [63].

Lookup provides a lookup service in a distributed system. Services register with their references and associated properties. The lookup server determines the most appropriate service or services based upon client requests. Multiple Lookup servers can be combined into a *Federation* of lookup services [66].

Manager handles a collection of objects. It takes care of creation, destruction, and manipulation of the objects. A client requests the object from the Manager and then interacts directly with the object. Once the operation is complete, the client returns the object to the Manager [124, 118]. Gehani and McGettrick present a similar concept known as a *Guardian* [49]. Operations on the resources are performed by executing the provided handlers.

In [114], the authors describe a *Task Manager* to handle thread creation and termination. By encapsulating these services, the domain code is rendered portable, and concurrency strategies are easily changed since only the Task Manager code needs to be re-written.

Tessier and Keller introduce the *Manager-Agent* pattern. It decentralizes the management of resources, simplifying control in a distributed system. An Agent is in charge of a group of resources related by some criteria. The Agent represents the resources to the rest of the management system and may take on some managerial aspects for the resource as well. The Manager handles some management function over the entire system. There may exist multiple Managers, and Agents may report to more than one Manager [123].

Repository provides a central data structure for a complex body of information that must be established, augmented, and maintained. Multiple clients need to access and manipulate the data, often concurrently. A large, centralized, transaction-oriented database is an example of a repository. The *Blackboard*, discussed in Section 2.3.2.2, is a related pattern [124, 115]. Hu and Gill, in [65], discuss the notion of a *Library*, which is similar. Here, the Library decouples the creation of a new object from the retrieval of an existing object from the repository cache.

Resource Exchanger manages resources shared among multiple server tasks. At some point, when a server requests a resource, such as a buffer, it must hand over another instance of the resource in exchange. This requirement allows the Resource Exchanger to maintain a constant pool of resources and minimize delay times. In addition, servers build up a credit (or lack of it) with the Resource Exchanger. Servers with a high load eventually use up their credit, which allows low load servers to be processed. This scheme thus reduces the overall server load, and allocates resources fairly [106].

Service Configurator enables the configuration and reconfiguration of communication services at any point in time without affecting other services. All services must have a uniform interface for configuration and control. It can initiate, suspend, resume, and terminate services dynamically. It is also known as a *Super-server* [68].

Acceptor decouples passive connection from the service after the connection is established. It creates, accepts, and activates a new handler whenever an event dispatcher notifies it that a connection has arrived from a client [107].

Connector decouples active service from the task's service after the service is initialized. It allows the services to evolve independently and transparently from the mechanisms used to establish the connections. It acts as a factory that assembles the resources necessary for a synchronous or asynchronous connection [107].

Command Processor separates the request for a service from its execution. The requests are managed as separate objects, which are scheduled for execution. Additional services such as requesting storage of commands for later rollback may also be done [117, 124]. It is also referred to as *Controller-Command* in [24] though the authors later refer to it as a Command Processor in [25].

View Handler assists in managing the views of application-specific data or multiple windows provided by a software system. It allows clients to open, manipulate, and dispose of views. It also coordinates dependencies among views and organizes their updates (frequently, views are updated in a priority ordering). Since the updates are performed by the data supplier at the view's request, it is the supplier who is responsible for notifying all dependent components (which could include the View Handler as well as views) about a change to its internal data. A variant of the View Handler uses Command objects to keep the handler independent of specific view interfaces [25].

Sponsor-Selector allows a client to request the appropriate resource from the Selector. In turn, the Selector broadcasts requests to Sponsors who rate their resources and return the ratings. The Selector uses the ratings to select the resource that is returned to the client for use [130].

Gatekeeper-Request-Resource passes a request for a resource to a Gatekeeper who manages the resources and passes on the request to the appropriate Resource. The Resource then processes the request [31].

2.3.2.2 Administrator

An Administrator hides worker tasks in the same way that a Proprietor hides a resource(s). The Administrator can delegate work to these worker tasks, and concurrency is improved further by ensuring that it is the workers who block when requesting work instead of the Administrator [50, 48]. Ideally, an Administrator only blocks when it has no work or management to perform.

The Administrator has two means of controlling workers. It can create Workers as needed and terminate them when the work is done, or it can create an initial pool of Worker tasks that are used as necessary [51]. There are several design patterns related to the issue of Worker creation. These consist of [92, 108]:

Thread per Request creates a thread for each client request, allowing all client requests to run concurrently.

Thread Pool handles requests for an unlimited number of clients, using limited stateless server resources. If the request cannot be currently filled, it is blocked until a thread is returned to the pool. This pattern is also known as a *Resource Pool* [60]. The variant known as *Client-Server-Service* allows clients to monitor the request's progress since the server publicizes its state [3, 1]. Douglas C. Schmidt's "Thread Pool" is an example of a Resource Pool.

Thread per Session creates a thread for each client session, handling all of that client's requests. This pattern is also known as *Thread per Connection*.

Some examples of worker tasks are:

Notifier which notifies the Administrator that an event has occurred [50].

Timer which notifies the Administrator of an amount of elapsed time [50].

Courier which allows the Administrator to communicate without blocking for a reply [50].

Assassin which deletes other tasks for the Administrator [19]. The *Evictor* is an Assassin variant. It removes idle servants based on a Least Recently Used algorithm. It can be extended to support distributed garbage collection, in order to reclaim the space occupied by unused servants [61]. Henning describes using an Evictor variant for a Trader. That variant uses a separate reaper thread to get rid of the unused servants.

A similar pattern is the *Undertaker* [40], which handles dangling reference pointers not recognized by the system as garbage, or the *Vulture* [11], which is responsible for terminating unauthorized software services and logons.

Cheriton presents a *Death Proprietor* for processing requests to destroy tasks. It also sends messages to the System Proprietor to reclaim the resources of destroyed tasks [29].

Overseer which manages other workers tasks [19].

Secretary and Director where the Secretary contains the set of all common state variables accessed by the Directors, and the Director makes requests of the Secretary, which then is responsible for coordinating the Director based upon the stored state information. Dijkstra uses "the metaphor of directors and a common secretary because in the director–secretary relation in real-life organization it's also unclear who is the master and who is the slave!" [37].

Shell and Tenant is an abstraction pattern for workers [51]. The *Shell* task takes on the role of any type of worker (known as a *Tenant*) based upon the information it receives from the Administrator.

2.3.2.2.1 Independent Workers

Most patterns that rely upon solving a problem using independent workers collect events or job requests into some sort of queue. These patterns include:

Distributed Bag of Tasks uses a "bag" that contains independent work requests and is shared by multiple worker tasks. Each worker repeatedly removes work from the bag and completes it. The processing of the request may generate more work requests to place in the bag. The manager implements the bag, hands out work, collects the results, and detects termination [8].

Magee and Kramer, in [79], refer to this as a *Supervisor-Worker* pattern. They note that is also known under the names *Replicated Worker* [7], *Process Farm* [22], and *Agenda Parallelism* [26].

Work Crew consists of a fixed set of worker tasks. The workers remove jobs from a queue, where jobs consist of computational work. If the worker can subdivide the job, it will do so, placing all pieces but the one it is working on back in the job queue. When the worker has completed its piece, it checks to see if all of the help requests have been answered. If they have not, it works on the next piece. The cycle continues until the entire job has been completed [98].

Manager-Agent is used to regroup a number of heterogeneous resources, whose interfaces cannot be modified to one homogeneous interface, under the supervision and control of an Agent. The Agent represents the resources to the rest of the management system. If so desired, the Agent can be responsible for managing certain aspects of the resources. Each Agent reports to one or more Managers, who handle some management function over the entire system. The Agent performs operations on the resources on behalf of the requesting Manager. If the Agent notices changes in its resources requiring the Manager to be notified, it reports the changes [123].

Master-Slave introduces redundancy, fault tolerance, safety and correctness. The Master task delegates work to independent Slave tasks and computes a final result from the results the Slaves return. The Master, when calculating the final result, may use different strategies for selecting among the Slave-returned results. These strategies include: taking the first result returned, taking the result the majority returned, taking the average of all the returned results, taking a result returned from a Slave that did not fail, or sometimes declining to select any result (for example, if they all returned different results). The Slaves may also use different strategies for providing the service for which they are responsible [24, 123, 48]. In [36], this design pattern is also referred to as *host-helper*. Variants listed in [25] include:

Object Group by Maffeis uses the Master-Slave variant that provides group communication and fault tolerance in a distributed environment.

Master-Slave Pattern for Parallel Compute Services by Brooks concentrates on describing how the Slaves can be implemented as processes.

Slaves as Threads by Kleiman, Shah, and Smaalders investigates using threads to implement the Slave task.

Workpool Model by Knopp and Reich uses a Workpool of Workers, corresponding to the idea of Slaves, to handle client requests. The request function sent by the client corresponds to the Master.

Gaggles by Black and Immel builds upon the Master-Slave pattern to use replicated service objects. The service objects are represented by the Gaggle, which forwards client requests to one of the replicated service objects.

Buschmann, in [23], states that the Master-Slave design pattern is based on the *Actor-Agent-Supplier* variant of the *Actor-Supplier* design pattern. This connection can be seen if the client requesting the service takes on the role of the Actor, the Master takes on the role of the Agent, and the Slaves act as the Suppliers.

Pacherie and Jézéquel see the Master-Slave design pattern as one possible refinement of their *Operator* design pattern, also known as *Ubiquitous Agent* [89].

Wilson discusses another variant, the *Crystalline Model*, also known as *Single*

Program, Multiple Data or *SPMD* [132]. This pattern consists of a finite set of worker tasks, more cannot be created dynamically, and a single controller. The worker tasks are organized in a regular topology since each communicates only with a direct neighbour. They each have their own data space and work independently until a communication event arrives, at which point they must all participate. The controller is also independent, and can communicate directly with any of the workers.

A variant of the Master-Slave for mobile computing, *Supervisor-Worker*, is discussed in [45]. The pattern is designed to protect mobile agents from having unauthorized tasks acquire or alter information. This protection is accomplished by building a central knowledge-base and management unit on top of the Master-Slave pattern, which ensures all information is process correctly, and at the right times. The participants in the pattern consist of the Agent, the Supervisor, and the Worker. The Agent is mobile, and has its own constraint manager keeping track of the constraints for the work the Agent is supposed to accomplish. The Supervisor divides up the work, controls the workers, and merges reports. It develops work completion strategies, creates subdivisions of the work, and keeps track of required information such as merge constraints, and Worker assignments. The Worker completes the job assigned by the Supervisor and sends reports to the Supervisor.

Cela and Alfonso examine the standard centralized version and two distributed versions of a Master-Slave pattern to solve preconditions of Sparse Approximate Inverses [27]. In the distributed versions, the Slaves communicate among each other to assist in solving the problem since the data has already been distributed. In the first version, a Slave answers a request for data only when it sends a request. In the other version, a Slaves answers a request for data only when a user defined signal arrives.

EmbarrassinglyParallel pattern uses a collection of concurrent, independent tasks to solve a problem [82]. In particular, it attempts to organize the computation so that each task completes at about the same time. Tasks that are faster take on a larger share of the computation. These tasks may be homogeneous, or heterogeneous. During execution, more tasks may be created, depending upon the problem being solved. The authors describe three variants of this pattern.

1. Sub-solutions are accumulated in a shared data structure. The tasks are no longer completely independent since they share and must synchronize access to the data structure.
2. A termination condition other than all tasks completing exists. This condition is particularly useful when an overall solution can be obtained without having to solve all subproblems.
3. Not all subproblems are initially known. This situation occurs when subproblems are generated while solving other subproblems.

This pattern is also known as *Master-Worker*, or *Task Queue*.

Two additional patterns, the *SeparableDependencies* and the *GeometricDecomposition* patterns are also presented. The first is related to EmbarassinglyParallel in that the tasks have dependencies, which when removed, allow the problem to be solved using the EmbarassinglyParallel pattern. The other is used when the concurrency is based on parallel updates of sections of a decomposed data structure, and the update of each section requires data from the other sections.

Leader/Followers is used to efficiently process events arriving from multiple event sources shared by multiple tasks. One task, the Leader, waits for an event. The Followers queue themselves, awaiting their turn as Leader. When the Leader detects an event, it promotes a Follower as the new Leader, and then processes the event. This demultiplexes and dispatches the event to the designated event handler. Upon completion, the task takes on the role of a Follower [112].

2.3.2.2.2 Cooperative Workers

The following design patterns rely upon cooperative Workers:

Collection-Worker uses a Collection to control a finite number of Workers. The Collection only performs operations that apply across its entire set of Workers. The Worker does all it can with what it knows, or what others may tell it in combination with what it knows [31].

Blackboard is a solution to the pattern recognition problem of transforming raw data (such as that collected by sensors) into a higher-level data structure when no deterministic algorithm for the transformation exists [25]. Instead, algorithms applying partial transformations are used, but the order in which the transformations are applied is unknown. The pattern involves three components: the Blackboard, a Moderator, and at least one Contributor (also referred to as a Knowledge Source). The Blackboard is a shared data structure maintaining the different versions of the data, ranging from the original raw form to the final product.

The Moderator is responsible for choosing among the Contributors' proposals. It selects the proposal calculated to advance the problem the furthest, allows the Contributor who submitted the proposal to update the Blackboard, and repeats the cycle. If necessary, the Moderator may reverse its decisions if it determines that the present trend will not lead to the desired solution.

The Contributors are independent tasks that communicate only with the Moderator. They view the Blackboard, and make a proposal to modify some level of the data structure. The proposal is accompanied by a metric indicating the degree of certainty of success, based upon the current state of the Blackboard.

In [25], the Repository is considered a generalization of the Blackboard. The authors also consider the Blackboard to be an extreme variant of the *Production System* by Forgy and McDermott. Within the Production System, subroutines are represented as condition-action rules, and data is globally available in working memory. The action specified by a condition-action rule is only performed when the associated condition is true, and the rule has been selected by the conflict resolution module.

The authors of [36] list a variant of the Blackboard, which they consider to be a variant of the Master-Slave pattern. Here the Coordinator serves as the Blackboard, which is now a work queue from which the Contributors take work to be computed. The results can then be fed back into the Blackboard.

Leader/Collaborator/Collaboration design pattern by [2] is based on the Client-Server-Service pattern, except new roles are assigned to the participants. It is intended for use in the context of cooperative, autonomous software agents. Here, the agent (Leader), upon determining that collaboration is required to fulfill its task, asks one or more Collaborators for assistance. When asking for help, the Leader indicates what type of collaboration is required. If a Collaborator agrees to help, a Collaboration object is created and its reference is returned to the Leader. (The Collaboration object encapsulates a thread of execution and manages the agent's collaboration process.) The Leader creates its own Collaboration object, which is configured to work directly with the other Collaboration object. These objects have direct visibility of each other while the collaboration is underway. From this point on, all communication is among peers since there is no longer any role distinction.

Model-View-Controller decouples the user interface of a system from its core functionality. The interactive application is divided into three components: the Model, the View, and the Controller. The *Model* contains the core functionality and data of the application. Its information is presented by one or more *Views* to the user (each View may present the information in a different format). Every View has a corresponding *Controller*. The Controller translates user input into a service request to the Model. Changes in the Model's state cause the Views to update the information presented to the user. This pattern implies a reliance upon the Publisher/Subscriber design pattern [25].

As the pattern is described, all communication is performed via update method calls, unlike the following Presentation-Abstraction-Control (PAC) pattern. No mention is made of issues such as mutual exclusion of service requests or updates, or of how consistency in the Model is maintained given that a Controller may need to obtain more information from the Model after the initial service request is made.

Presentation-Abstraction-Control is an alternative approach to the problem solved by Model-View-Controller (MVC). A tree-like hierarchy of cooperating agents is responsible for an interactive system. The hierarchy is tree-like in the sense that there is only one agent at the top-most level, but there are several intermediate-level agents and many low-level agents. Each agent depends upon all of the agents higher up in the hierarchy. The top-level agent is responsible for the core functionality of the system, and any parts of the user interface that cannot be assigned to lower-level agents. An intermediate agent represents either multiple low-level agents, or the relationships between them. A low-level agent represents the basic concepts of the system, upon which the users can act [25].

Each agent is divided into *Presentation*, *Abstraction*, and *Control* sections, though the Presentation section may be non-existent at the top levels of the hierarchy. The Presentation section provides the "visible behaviour" of the agent. It is equivalent to the MVC View and Controller.

The Abstraction section maintains and manipulates the data model that underlies the agent, which is the same as the Model in MVC. A particular point to note is that if a low-level agent requests information, all agents in the path to the top-level agent must participate in the communication. As well, if an agent depends upon data stored in another agent, then a pattern such as Publisher/Subscriber must be used to ensure notification of updates.

The Control section allows the agent to communicate with other agents, and connects the Presentation and Abstraction sections. It acts as a Mediator, passing on changes from the top-level agent, and requests from the lower-level agents.

While it is mentioned that multiple threading can be used to implement this pattern, no mention is made of how mutual exclusion or synchronization (a necessity of communication) is to be performed.

2.3.2.2.3 Workers Talking to Clients

The key to these patterns is that the delegated task ends up communicating with the client instead of the sever.

Sender-Lookup-Receiver has the Sender look up the Receiver in a Lookup service. The Sender then contacts the Receiver directly [31].

Caller-Dispatcher-Caller Back has the Sender call the Dispatcher who tells Caller Back to return the Sender's initial call [31].

Matchmaker see the Broker as Matchmaker in section 2.3.1.3.

A similar notion can be found in the V Distributed System where a client can use a group identifier to multicast to all servers in a group in an effort to find the server responsible for managing a particular object. Once the responsible server answers, the client can communicate with it directly and thus avoid multicasting to the group as a whole [30].

2.4 Client-Server Interaction Patterns

There are two main forms of interaction available beyond the basic communication of a request from a client to a server. The request, or subsequent work, can be delegated to other tasks in the system. It may also be necessary to provide a means of keeping tasks aware of changes in the system state by using update patterns.

2.4.1 Delegation Patterns

Figure 2.5 illustrates the basic form of a client-server interaction pattern. Note that the roles of client and server are fluid since the server can in turn become the client of another server. Other forms of interaction involve the Administrator delegating an operation to multiple Workers. The Workers may be independent, or cooperative. They can also communicate directly with the Client.

The *Pipeline* is the most basic form of interaction pattern. It is a collection of tasks in which the output of one becomes the input of another [9, 15, 18, 127, 115, 79]. The information can be either pushed or pulled through the Pipeline. In this

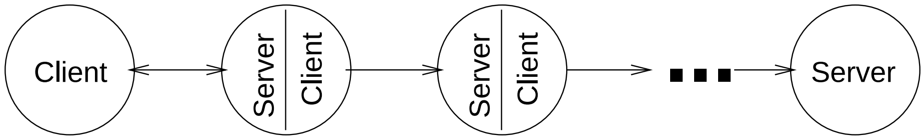


Figure 2.5: Delegation Pattern

fashion, it can effectively delegate work to other tasks in the Pipeline. It is also known as *Pipes and Filters* since a Pipeline can be used to filter information as it passes through the pipe. The filters may be sources (produce data), sinks (consume data), transformers, or removers. A pipe may simply be a data connection such as a *Stream* [4, 39], or a data connection plus some other structure such as a repository (see Figures 2.6 and 2.7). [124, 85, 25].

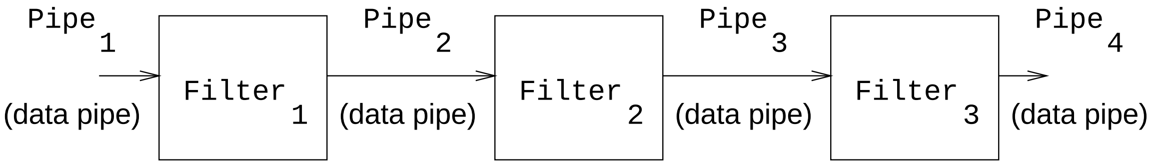


Figure 2.6: Simple Pipeline with Pipes as Data Connections

Pipeline variants include:

Translator pattern can be viewed as a limited Pipeline since it servers to marshal and unmarshal messages [123].

Producer-Consumer is a Pipeline where each task in the pipeline acts as a *filter* [7, 8, 57, 61]. Buschmann and Meunier refine the idea further by listing several variants of the Producer-Consumer pattern such as the Producer-Repository-

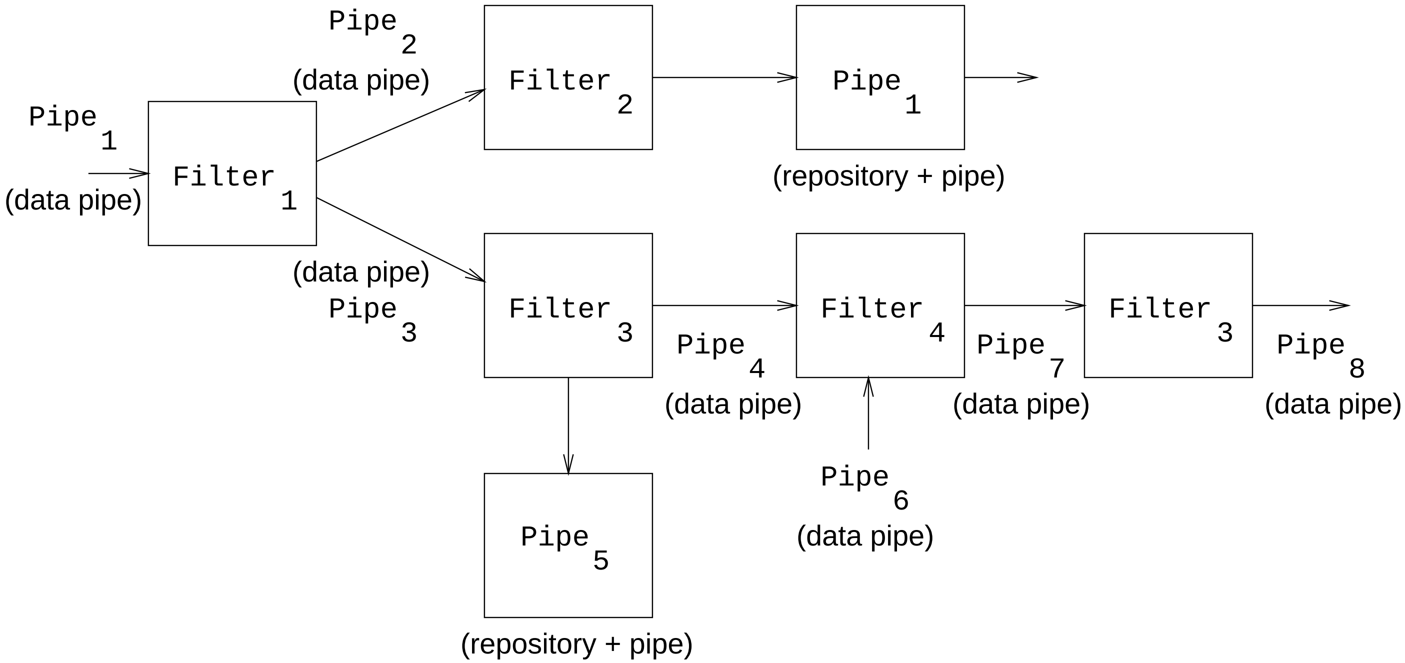


Figure 2.7: Network Pipelines with Pipes as Data Connections and Repositories

Consumer pattern, and the Producer-Sensor-Consumer pattern in [24] but do not describe them further.

The *Forwarder-Receiver* decoupling design pattern is another form of Producer-Consumer. It provides transparent communication among peers. Every task is provided with both a Forwarder and a Receiver (see Figure 2.8). The Forwarder marshals and delivers the message to the other Receiver. The Receiver unmarshals and delivers the message to its associated peer [24, 25]. Since the Readers and Writer pattern is a restricted form of the Producer-Consumer design pattern, it is also considered a form of Pipeline [124, 76].

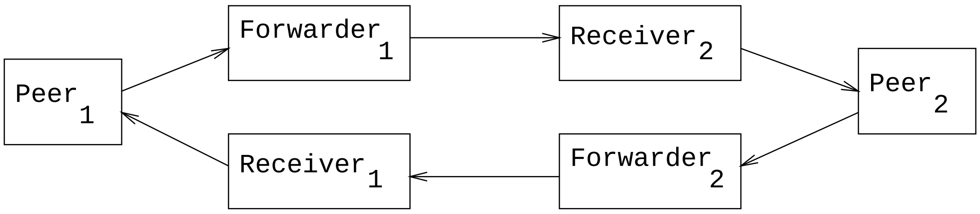


Figure 2.8: Forwarder-Receiver Interaction

Another variant of Producer-Consumer is *Producer-Intermediary-Consumer*. This pattern places an intermediary between the Producer and the Consumer. This intermediary may be a passive object such as a monitor, or an active object such as a task. An example of this pattern is Doble's Shopper who obtains items for a Consumer from a Producer [38].

Doug Lea defines a Producer-Consumer variant, the *Flow Network*. A Flow Network is a collection of objects that pass one way messages from sources (Producers) to sinks (Consumers). Examples of Flow Networks are: avionics control systems, assembly systems, data flow systems, work flow systems, and event systems [76]. Andersen refers to this as a *Network* since there may be multiple input sources, and multiple output sinks [6].

Tee and Join Pipeline Systems listed as variants by [25] allow more than one input and/or output.

Program Chaining divides each piece of the program into an individual component that invokes the next one in sequence. The main purpose of this pattern is to reduce memory requirements by using secondary storage (a mass storage device such as a floppy disk, hard disk, tape drive) to store the components until they are required. Each newly invoked phase of the program and its run-time data are loaded from secondary storage into main memory, completely replacing the invoking executable [88].

Chain of Responsibility allows a task to send a request to another task, which is implicitly at the head of a chain of other tasks. The request is passed through the chain, and any member of the chain can fulfill the request, depending upon run-time conditions. The number of participants in the chain is unbounded, and participants in the chain can be selected at run-time. This design pattern is also known as *Event Handler*, *Bureaucrat*, and *Responder* [47].

Two patterns that rely heavily upon Chain of Responsibility are: Matcher-Handler, and Bureaucracy. The *Matcher-Handler* design pattern has traits of the Observer, Chain of Responsibility, and Strategy design patterns. It delivers data to implicitly specified receivers, of whom more than one may handle the request. The set of receivers can also be dynamically specified. These qualities let it behave like Chain of Responsibility.

Since a data event may need to be processed independently and simultaneously by more than one task, Matcher-Handler has traits of the Observer design pattern, except that the handlers are only notified when the information they are specifically waiting for has arrived.

In order to reduce the impact of changing the matching criteria over time, the Strategy pattern isolates the matching behaviour; however, the purpose of Strategy is to provide selectable behaviour, particularly at runtime. The end result is the same, though the purposes differ [84].

The other design pattern, *Bureaucracy*, is based on the Composite[9], Observer, and Chain of Responsibility design patterns. The Bureaucracy design pattern lets developers build self-contained, hierarchical structures that can interact with clients on every level. No external control is necessary, and the structures can maintain their own inner consistency. Chain of Responsibility forwards requests until a task is reached that can fully execute the request. The previous tasks may have partially executed the request, reinterpreted it, or cancelled it. Because the work can be handled by a task low in the hierarchy, which would not have notified higher-level tasks since it knew how to process the request, the higher-level tasks still need to be warned of the state change. Therefore the Observer is responsible for notifying tasks of the

[9] *GoF* define the Composite pattern as composing objects into tree structures to represent hierarchies (either part of, or the entire hierarchy). Clients can treat individual objects and compositions of objects uniformly.

hierarchy of state changes. The Composite pattern allows the building of complex structures [99].

2.4.2 Update Patterns

These patterns are responsible for maintaining the state information in a system by ensuring that all participants are notified of changes.

Observer defines and maintains a dependency among objects. As an object changes, all clients who have registered an interest with the Observer are notified of the changes. This design pattern is also known as: *Broadcaster/Listener* [1], *Caller/Provider* [3], *Provider/Observer* [43], *Subscriber/Publisher* [70], *Announcer/Listener* [79], *Dependents*, *Publisher-Subscriber* [24, 25, 31], *Update*, and *Listener* [47, 123, 100, 124, 129]. Wolf and Liu refer to it as a "dependency" and note that Coad refers to this as a "broadcast" [133]. Feiler and Tichy classify it as a special case of the Propagator pattern [44]. Kim and Benner provide several implementation design patterns for the Observer design pattern [71].

There are a number of related design patterns, including the Spy, the Notification Server, the Notifier, the Handler, Broadcasting Sequential Processes, the Component Bus, and the ValueModel.

In the *Spy* design pattern, the Spy task monitors the progress of a parallel program by examining shared global memory. It can consolidate, process, and report the gathered information [121].

Hirschfeld and Eastman's *Notification Server* inverts the Observer's roles. The Notification Server takes on the role of the observed subject though it is really the shared resource that is being observed, while the registered clients are the observers. Also unlike the Observer pattern, the Notification Server is not notified of the changes, only the registered clients are notified [64]. The *Notifier* notifies the administrator when an event has occurred; thus, it could be used as part of the Observer implementation [50].

Berczuk defines a *Handler* to process items when the end-to-end system requirements have not been fully specified yet. When an item to be processed is created, it registers itself with the handler. This pattern is related to the Observer pattern except that the observer is the class of object created, and the event that triggers notification is the creation of an object of a given class [17].

Broadcasting Sequential Processes (BSP) uses the Publisher-Subscriber pattern. A message broadcast (published) by one task can be received by all other tasks (subscribers). Thus, programs are collections of loosely coupled tasks cooperating to accomplish a common goal [48].

The *Component Bus* allows tasks to communicate indirectly. It manages the routing of information dynamically, as tasks can dynamically register interest in the required information [41]. The Component Bus is also referred to as a Message Bus. It (or a Broker) is used to implement the *Event-Based Integration* design pattern [124].

Woolf's *ValueModel* framework is another variant of the Observer design pattern. It contains a value, and it informs its registered dependents when the value changes [134].

Propagator is a family of patterns for consistently updating objects in a dependency network [44]. All of these patterns support smart propagation for avoiding redundant work as well as concurrent updates. It is also known as *Cascaded Update*. There are four main Propagator patterns:

Strict Propagator always performs a complete update. No indication of success or failure is given. It only keeps track of its dependents for the purpose of propagation. The changed predecessor is passed with the update method and is thus accessible to the dependents, allowing them to identify which predecessor has changed. This design pattern combines the methods of the subject and object classes of Observer and makes the notification method recursive. It is also known as the *Forward*, *Immediate*, or *Eager* Propagator.

Strict Propagator with Failure is the same as the Strict Propagator, except that an object is marked as invalid if the update failed. It is also known as an *Optimistic* Propagator.

Lazy Propagator only updates objects if it can determine that they are changed. Objects in the dependency network only keep track of their direct predecessors. It is also known as *Update on Demand*, or the *Backward* Propagator.

Adaptive Propagator is a compromise between the Strict Propagator and the Lazy Propagator. It immediately forward propagates only the invalid marker and separately propagates updates optimistically, periodically, or on demand. The forward propagation phase performs no updates, and stops quickly in the case of successive waves, since it encounters already marked objects. The actual updates take place in the backward phase, which can even be run simultaneously with the forward phase. In this fashion, it can avoid successive waves of updates (Strict Propagator), or having to traverse the entire network backwards to the roots and check time stamps (Lazy Propagator).

Chapter 3

Conclusion

In the realm of design patterns, there is no satisfactory taxonomy that meets the needs of everyone. This is amply evident from the proliferation of possibilities. A large part of this is due to the inability to firmly classify each pattern into solely one nomenclature (architectural pattern versus design pattern), category of functionality, or purpose. This process is rendered even more complicated by the quantity of published patterns, whose names and purposes overlap, as is seen throughout Chapter 2. The scope of the problem is magnified and driven home once the reader realizes that concurrency patterns are only a small portion of patterns applicable to a restricted area of computer science.

Though no system of classification is perfect, I believe that a good start has been made by identifying the basic, underlying metapatterns of concurrent programming. The three main divisions are: synchronization, mutual exclusion, and client-server. Synchronization covers the mechanisms for synchronizing tasks as well as patterns in communication while mutual exclusion discusses various lock patterns.

Within the category of client-server, design patterns can be further subdivided into client-side patterns, server-side patterns, and client-server interactions. Client-side patterns involve the introduction of an intermediary (passive or active, implicit or explicit) between the client and server. The three main metapatterns are proxies, mediators, and brokers. Server-side metapatterns are those of the proprietor and the administrator. Since the administrator uses worker tasks, the patterns involving workers are also examined, yielding the basic patterns of independent workers, cooperative workers, and workers communicating directly with a client.

The final category of client-server interactions neatly covers the areas missing from the previous two, subsuming both delegation and publication patterns. These divisions account for both the common and exotic patterns found in concurrent programming, without descending to the level of idioms such as semaphores or monitors.

Appendix A

Sample Design Pattern

This pattern is taken from pages 293 to 303 of the *GoF* design patterns text [47].

A.1 OBSERVER Object Behavioral

A.1.1 Intent

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

A.1.2 Also Known As

Dependents, Publish-Subscribe

A.1.3 Motivation

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency among related objects. You do not want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data [KP88, LVC89, P+88, WGM88]. Classes defining application data and presentations are independently reused. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data-object using different presentations (see Figure A.1). The spreadsheet and the bar chart do not know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.

This behavior implies that the spreadsheet and bar chart are data views dependent of the data object, and therefore, are notified of any change in its state. There is no reason to limit the number of dependent objects to two; any number of different user interfaces for the same data could exist.

The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer queries the subject to synchronize its state with the subject's state.

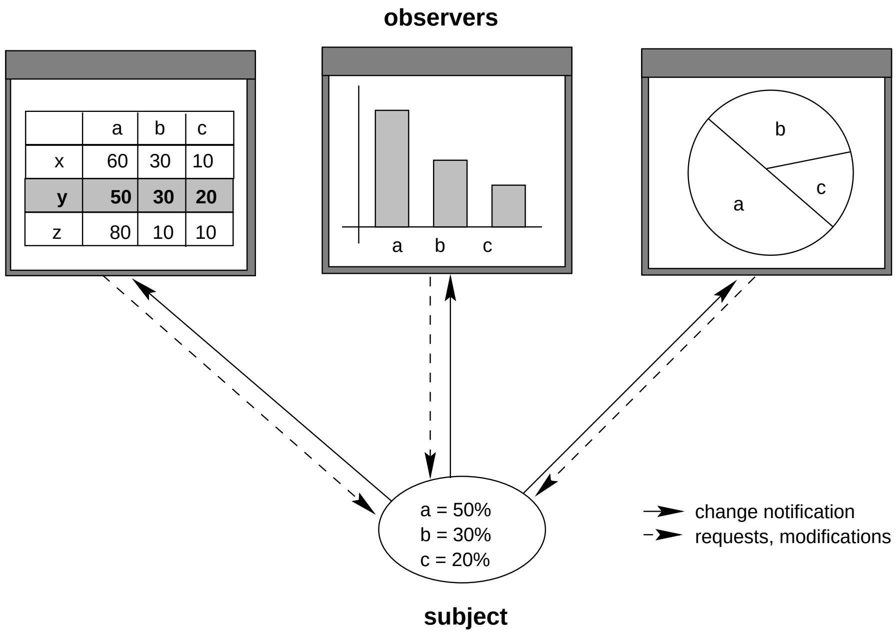


Figure A.1: Observer Design Pattern Example

This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know the types or identities of the observers. Any number of observers can subscribe to receive notifications.

A.1.4 Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you do not know statically how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about the kind of objects these are. In other words, you do not want the objects tightly coupled.

A.1.5 Structure

See Figure A.2.

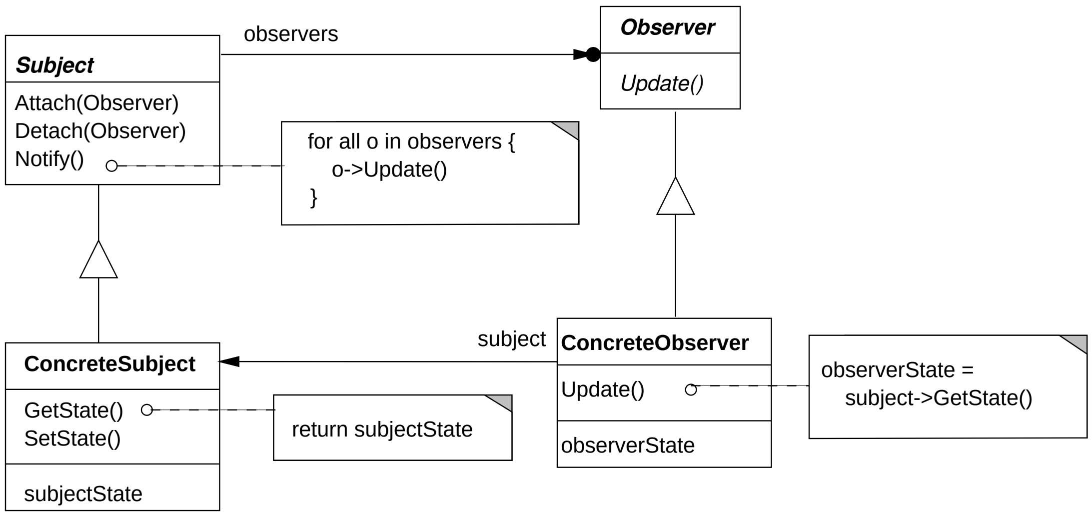


Figure A.2: The Observer Design Pattern Structure in UML.

A.1.6 Participants

A.1.6.0.4 Subject

- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

A.1.6.0.5 Observer

- defines an updating interface for objects that are notified of changes in a subject.

A.1.6.0.6 ConcreteSubject

- stores state of interest to `ConcreteObserver` objects.

- sends a notification to its observers when its state changes.

A.1.6.0.7 ConcreteObserver

- maintains a reference to a `ConcreteSubject` object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

A.1.7 Collaborations

- `ConcreteSubject` notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- A `ConcreteObserver` object, once informed of changes in the concrete subject, may query the subject for information. `ConcreteObserver` uses this information to reconcile its state with that of the subject. The interaction diagram in Figure A.3 illustrates the collaborations among a subject and two observers:

The collaboration starts with an observer telling the subject to change its state by invoking the `SetState` method that may implicitly invoke the `Notify` method. Once the subject determines that a change has occurred, it notifies all observers by invoking their respective `Update` methods.

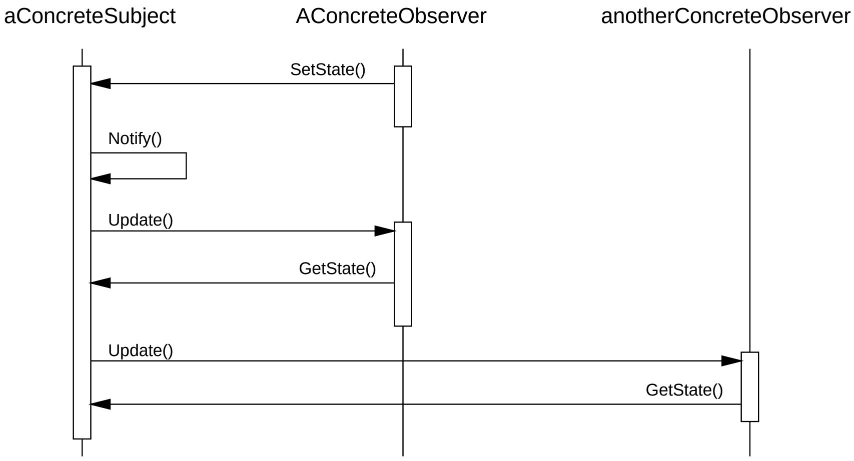


Figure A.3: Observer Design Pattern Interaction Diagram

Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject. `Notify` is not always called by the subject. An observer, or another kind of object entirely, can call it. The Implementation subsection discusses some common variations.

A.1.8 Consequences

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1. *Abstract coupling between Subject and Observer.* All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract

Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling among subjects and observers is abstract and minimal.

Because Subject and Observer are not tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, then the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction).

2. *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends need not specify its receiver (presumably since it might be directed to a `ChangeManager` who is responsible for broadcasting the message to registered observers). The notification is broadcast automatically to all interested objects that subscribed to it. The subject does not care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It is up to the observer to handle or ignore a notification.

3. *Unexpected updates.* Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that are not well-defined or maintained usually lead to spurious updates, which can be hard to track down.

This serious problem is aggravated by the fact that the simple update protocol provides no details on what changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

A.1.9 Implementation

Several issues related to the implementation of the dependency mechanism are discussed in this subsection.

1. *Mapping subjects to their observers.* The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. However, such storage may be too expensive when there are many subjects and few observers. One solution is incur a time penalty and reduce the required amount of space by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping. Thus a subject with no observers does not incur storage overhead. On the other hand, this approach increases the cost of accessing the observers by adding in the cost of the look-up.

2. *Observing more than one subject.* It might make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may depend on more than one data source. It is necessary to extend the Update interface in such cases to let the observer know which subject is sending the notification. The subject can simply pass itself as a parameter

in the `Update` operation, thereby letting the observer know which subject to examine.

3. *Who triggers the update?* The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls Notify to trigger the update? Here are two options:

   (a) Have state-setting operations on Subject call `Notify` after they change the subject's state. The advantage of this approach is that observers do not have to remember to call `Notify` on the subject. The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient.

   (b) Make subjects responsible for calling `Notify` at the right time. The advantage here is that the subject can wait to trigger the update until after a series of state changes has occurred, thereby avoiding needless intermediate updates. The disadvantage is that subjects have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call `Notify`.

4. *Dangling references to deleted subjects.* Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it. In general, simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

5. *Making sure Subject state is self-consistent before notification.* It is important to make sure Subject state is self-consistent before calling `Notify`, because observers query the subject for its current state in the course of updating their own state.

   This self-consistency rule is easy to violate unintentionally when Subject subclass operations call inherited operations. For example, the notification in the following code sequence is triggered when the subject is in an inconsistent state:

```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
        // trigger notification

    _myInstVar += newValue;
        // update subclass state (too late!)
}
```

   You can avoid this pitfall by sending notifications from template methods (Template Method (325)) in abstract Subject classes. Define a primitive operation for subclasses to override, and make `Notify` the last operation in the template method, which will ensure that the object is self-consistent when subclasses override Subject operations. An example of this is the template method `Cut` containing the primitive operation `ReplaceRange` which is overridden by the subclasses.

```
void Text::Cut (TextRange r) {
    ReplaceRange(r);            // redefined in subclasses
    Notify();
}
```

By the way, it is always a good idea to document which Subject operations trigger notifications.

6. *Avoiding observer-specific update protocols: the push and pull models.* Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely.

   At one extreme, which we call the **push model**, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the **pull model**; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

   The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that are not always true. The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. On the other hand, the pull model is potentially inefficient, because Observer classes must ascertain what changed without help from the Subject.

7. *Specifying modifications of interest explicitly.* You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those observers that have registered interest in that event. One way to support this uses the notion of **aspects** for Subject objects. To register interest in particular events, observers are attached to their

subjects using

```
void Subject::Attach(Observer*, Aspect& interest);
```

where interest specifies the event of interest. At notification time, the subject supplies the changed aspect to its observers as a parameter to the Update operation. For example:

```
void Observer::Update(Subject*, Aspect& interest);
```

8. *Encapsulating complex update semantics.* When the dependency relationship among subjects and observers is particularly complex, an object that maintains these relationships is possibly required. We call such an object a `ChangeManager`. Its purpose is to minimize the work required to make observers reflect a change in their subject. For example, if an operation involves changes to several interdependent subjects, you may have to ensure that their observers are notified only after all the subjects were modified to avoid notifying observers more than once.

   `ChangeManager` has three responsibilities:

   (a) It maps a subject to its observers and provides an interface to maintain this mapping. This eliminates the need for subjects to maintain references to their observers and vice versa.

   (b) It defines a particular update strategy.

   (c) It updates all dependent observers at the request of a subject.

Figure A.4 depicts a simple `ChangeManager`-based implementation of the Observer pattern. There are two specialized `ChangeManager`s. `SimpleChangeManager` is naive in that it always updates all observers of each subject. In contrast, `DAGChangeManager` handles directed-acyclic graphs of dependencies among subjects and their observers. A `DAGChangeManager` is preferable to a `SimpleChangeManager` when an observer observes more than one subject. In that case, a change in two or more subjects might cause redundant updates. The `DAGChangeManager` ensures the observer receives just one update. `SimpleChangeManager` is fine when multiple updates are not an issue.

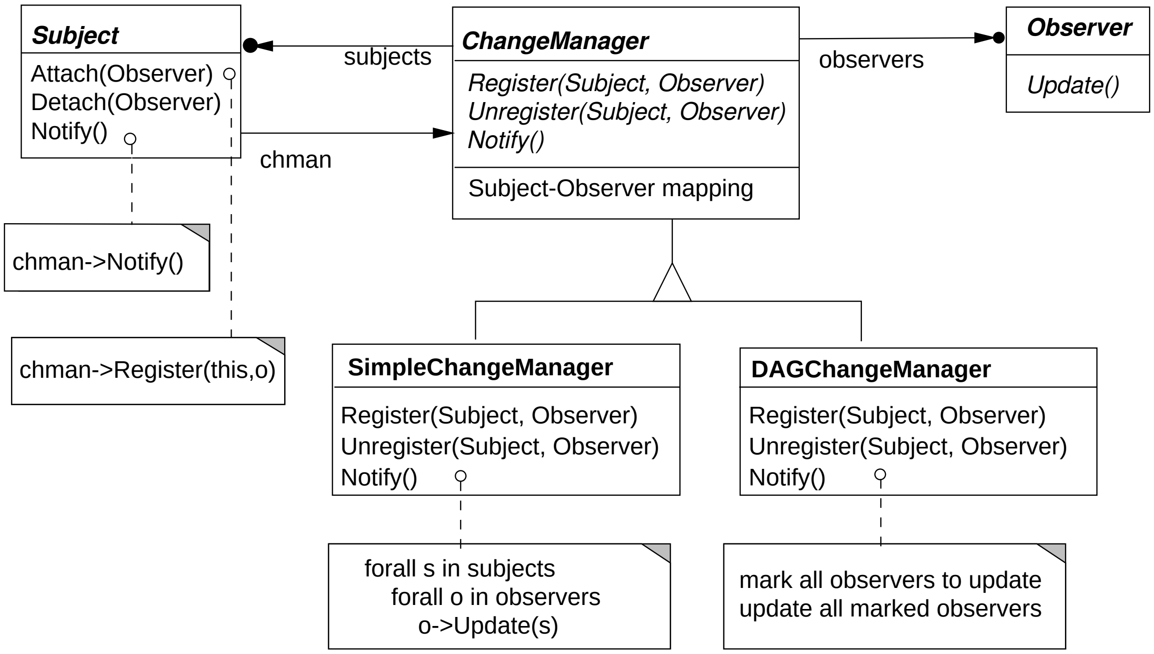


Figure A.4: Observer Design Pattern Change Manager

`ChangeManager` is an instance of the Mediator (273) pattern. In general there is only one `ChangeManager`, and it is known globally. The Singleton (127) pattern is potentially useful here.

9. *Combining the Subject and Observer classes.* Class libraries written in languages that lack multiple inheritance (like Smalltalk) generally do not define separate Subject and Observer classes. One proposed solution is the combination of their interfaces into one class. That lets you define an object that acts as both a subject and an observer without multiple inheritance. In Smalltalk, for example, the Subject and Observer interfaces are defined in the root class `Object`, making them available to all classes.

A.1.10 Sample Code

An abstract class defines the Observer interface:

```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

This implementation supports multiple subjects for each observer. The subject passed to the `Update` operation lets the observer determine which subject changed when it observes more than one.

Similarly, an abstract class defines the Subject interface:

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

`ClockTimer` is a concrete subject for storing and maintaining the time of day. It notifies its observers every second. `ClockTimer` provides an interface for retrieving individual time units such as the hour, minute, and second.

```
class ClockTimer : public Subject {
public:
    ClockTimer();
```

```
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick();
    };
```

The `Tick` operation gets called by an internal timer at regular intervals to provide an accurate time base. `Tick` updates the `ClockTimer`'s internal state and calls `Notify` to inform observers of the change:

```
    void ClockTimer::Tick () {
        // update internal time keeping state
        // ...
        Notify();
}
```

Now we can define a class `DigitalClock` that displays the time. It inherits its graphical functionality from a `Widget` class provided by a user interface toolkit. The Observer interface is mixed into the `DigitalClock` interface by inheriting from `Observer`.

```
    class DigitalClock: public Widget, public Observer {
    public:
        DigitalClock(ClockTimer*);
        virtual ~DigitalClock();

        virtual void Update(Subject*);
            // overrides Observer operation

        virtual void Draw();
            // overrides Widget operation;
            // defines how to draw the digital clock
    private:
        ClockTimer* _subject;
```

```
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}
```

Before the `Update` operation draws the clock face, it checks to make sure the notifying subject is the clock's subject:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        Draw();
    }
}

void DigitalClock::Draw () {
    // get the new values from the subject

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // etc.

    // draw the digital clock
}
```

An `AnalogClock` class is similarly defined.

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
```

```
    // ...
};
```

The following code creates an `AnalogClock` and a `DigitalClock` that always show the same time:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Whenever the `timer` ticks, the two clocks are updated and redisplay themselves appropriately.

A.1.11 Known Uses

The first and perhaps best-known example of the Observer pattern appears in Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalk environment [KP88]. MVC's `Model` class plays the role of Subject, while View is the base class for observers. Smalltalk, ET++ [WGM88], and the THINK class library [Sym93b] provide a general dependency mechanism by putting Subject and Observer interfaces in the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern are InterViews [LVC89], the Andrew Toolkit [P+88], and Unidraw [VL90]. InterViews defines Observer and Observable (for subjects) classes explicitly. Andrew calls them "view" and "data object," respectively. Unidraw splits graphical editor objects into View (for observers) and Subject parts.

A.1.12 Related Patterns

Mediator (273): By encapsulating complex update semantics, the `ChangeManager` acts as mediator between subjects and observers.

Singleton (127): The `ChangeManager` may use the Singleton pattern to make it unique and globally accessible.

Bibliography

[1] Amund Aarsten, Davide Brugali, and Giuseppe Menga. Designing concurrent and distributed control systems. *Communications of the ACM*, 39(10):50–58, 1996.

[2] Amund Aarsten, Davide Brugali, and Giuseppe Menga. Patterns for cooperation. In *Proceedings of the Third Joint Pattern Languages of Programs, Distribution Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/amund1.ps.gz`.

[3] Amund Aarsten, Gabriele Elia, and Giuseppe Menga. G++: A pattern language for the object-oriented design of concurrent and distributed information systems, with Applications to computer integrated manufacturing. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, volume 1 of *Software Patterns Series*. Addison-Wesley, 1995. Retrieved January 1, 2000 from Pattern Languages of Programs Design database on the World Wide Web: `ftp://galileo.polito.it/articles/gpp/plop94.ps`.

[4] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984. Retrieved January 9, 2001 from the MIT Press database on the World Wide Web: `http://mitpress.mit.edu/sicp/full-text/sicp/book/book.html`.

[5] Gul Agha, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research directions in concurrent object-oriented programming.* MIT, 1993.

[6] Henning Andersen. Network: A pattern for composing computation. In *Proceedings of the Second European Conference on Pattern Languages of Programs, General Design Patterns*, Munich, Germany: Siemens, 1997. (EuroPLoP'97) Siemens Technical Report 120/SW1/FB. Retrieved September 19, 2000 from the EuroPLoP^{TM} 1997 database on the World Wide Web: `http://www.riehle.org/events/europlop-1997/p15final.pdf`.

[7] Gregory Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[8] Gregory Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[9] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1), March 1983.

[10] Anonymous. Demon, July 1993. Retrieved December 20, 2000 from the Hacker Dictionary database on the World Wide Web: `http://www.lysator.liu.se/hackdict/split2/demon.html`.

[11] Anonymous. Glossary of tech support terms associated with version 5.0.x of the raptor firewall, 1998. Retrieved January 12, 2001 from AXENT Technologies Technical Support Group for Raptor products database on the World Wide Web: `http://www.raptor.com/cs/FAQ/eagle5glossary.html`.

[12] Anonymous. *Self-Addressed Stamped Envelope.* Portland Pattern Repository, 4 September 2000. `http://c2.com/cgi/wiki?SelfAddressedStampedEnvelope`.

[13] Brad Appleton. What is a pattern anyway? *Patterns and Software: Essential Concepts and Terminology*, 14 February 2000. `http://www.enteract.com/~bradapp/docs/patterns-intro.html`.

[14] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG.* Prentice Hall, Second edition, 1996.

[15] Jean Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems.* Addison Wesley, Second edition, 1998.

[16] Rajive Bagrodia. Synchronization of asynchronous processes in CSP. *The Association for Computing Machinery Transactions on Programming Languages and Systems*, 11(4), 1989. Retrieved September 28,

2000 from the The Association for Computing Machinery Digital Library database on the World Wide Web: `http://www.acm.org/pubs/citations/journals/toplas/1989-11-4/p585-bagrodia`.

[17] Stephen Berczuk. Organizational multiplexing: Patterns for processing satellite telemetry with distributed teams. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

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

[19] K. S. Booth, W. M. Gentleman, and J. Schaeffer. Anthropomorphic Programming. Technical Report CS-82-47, Department of Computer Science, University of Waterloo, 1984.

[20] Per Brinch Hansen. Concurrent programming concepts. *Software—Practice and Experience*, 5(4):223–245, December 1973.

[21] F. Lee Brown, Jr., James DiVietri, Graziella Diaz de Villegas, and Eduardo B. Fernandez. The authenticator pattern. In *Proceedings of the Sixth Pattern Languages of Programs, The Group 3 Workshop*. Pattern Languages of Programs, 1999. Retrieved January 5, 2001 from the PLoP^{TM} 1999 database on the World Wide Web: `http://st-www.cs.uiuc.edu/ plop/plop99/proceedings/Fernandez4/Authenticator3.PDF`.

[22] Alan Burns and Geoff Davies. *Concurrent Programming*. Addison-Wesley, 1993.

[23] Frank Buschmann. The master-slave pattern. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, volume 1 of *Software Patterns Series*. Adddison-Wesley, 1995.

[24] Frank Buschmann and Regine Meunier. A system of patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, volume 1 of *Software Patterns Series*. Adddison-Wesley, 1995.

[25] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley and Sons Ltd, 1996.

[26] Nicholas Carriero and David Gelertner. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3), September 1989. Retrieved September 28, 2000 from the ACM Digital Library database on the World Wide Web: `http://dev.acm.org/pubs/citations/journals/surveys/1989-21-3/p323-carriero/`.

[27] José Cela and José Alfonso. Parallelization of the spai preconditioner in a master-slave configuration. In *Third European PVM Conference Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[28] Arthur Charlesworth. The multiway rendezvous. *Transactions on Programming Languages and Systems*, 9(2), July 1987.

[29] David R. Cheriton. Multi-process structuring and the thoth operating system. Technical Report CS-79-19, Department of Computer Science, University of Waterloo, 1979.

[30] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3), March 1988.

[31] Peter Coad, David North, and Mark Mayfield. *Object Models : Strategies, Patterns and Applications*. Prentice Hall, Second edition, 1997.

[32] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[33] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Software Patterns Series. Addison-Wesley, 1995.

[34] Fernando Das Neves and Alejandra Garrido. Warden: A pattern for object distribution. In *Proceedings of the Third Joint Pattern Languages of Programs, Distribution Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/ PLoP-96/warden.ps.gz`.

[35] Andrew Davison. *A Survey of Logic Programming-Based Object-Oriented Languages*. Massachusetts Institute of Technology, 1993.

[36] Dennis de Champeaux, Douglas Lea, and Penelope Faure. *Object-Oriented System Development*. Addison-Wesley, HTML edition, 1993. Retrieved September 20, 2000 from Doug Lea's home page on the World Wide Web: `http://g.oswego.edu/dl/oosdw3/index.html`.

[37] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In C.A.R. Hoare and R.H. Perrott, editors, *Operating Systems Techniques*, pages 72–93. 1972.

[38] Jim Doble. Shopper. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[39] Stephen H. Edwards. Streams: A pattern for pull-driven processing. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, volume 1 of *Software Patterns Series*. Adddison-Wesley, 1995.

[40] Björn Eiderbäck and Jiarong Li. Undertaker. In *Proceedings of the Second European Conference on Pattern Languages of Programs, General Design Patterns*, Munich, Germany: Siemens, 1997. (EuroPLoP'97) Siemens Technical Report 120/SW1/FB. Retrieved September 19, 2000 from the EuroPLoP^{TM} 1997 database on the World Wide Web: `http://www.riehle.org/events/europlop-1997/p17final.pdf`.

[41] Philip Eskelin. Component interaction patterns. In *Proceedings of the Sixth Pattern Languages of Programs, The Group 1 Workshop.* Pattern Languages of Programs, 1999. Retrieved September 14, 2000 from the PLoP^{TM} 1999 database on the World Wide Web: `http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/Eskelin1/ComponentInteractionPatterns.PDF`.

[42] Philip Eskelin. Interruptible command. In *Proceedings of the Fifth Pattern Languages of Programs, The Group 2 Network of Learning Workshop.* Pattern Languages of Programs, 1999. Retrieved September 20, 2000 from the PLoP^{TM} 1998 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P46.pdf`.

[43] Ted Faison. Interaction patterns for communicating processes. In *Proceedings of the Fifth Pattern Languages of Programs, Four-Story Limit Workshop*, 1998. Retrieved September 19, 2000 from the PLoP^{TM} 1998 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P02.pdf`.

[44] Peter Feiler and Walter Tichy. Propagator: A family of patterns. In *Proceedings of the Third Joint Pattern Languages of Programs, System*

Configuration and Resource Management Workshop, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/ PLoP-96/tichy.ps.gz`.

[45] Sebastian Fischmeister and Wolfgang Lugmayr. The supervisor-worker pattern. In *Proceedings of the Sixth Pattern Languages of Programs, The Group 5 Workshop*. Pattern Languages of Programs, 1999. Retrieved September 14, 2000 from the PLoP^{TM} 1999 database on the World Wide Web: `http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/fischmeister/pattern-times.pdf`.

[46] Robert Flanders and Eduardo B. Fernandez. Data filter architecture pattern. In *Proceedings of the Sixth Pattern Languages of Programs, The Group 7 Workshop*. Pattern Languages of Programs, 1999. Retrieved January 5, 2001 from the PLoP^{TM} 1999 database on the World Wide Web: `http://st-www.cs.uiuc.edu/ plop/plop99/proceedings/Fernandez5/Flanders3.PDF`.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[48] Narain Gehani. Broadcasting sequential processes (bsp). In Narain Gehani and Andrew D. McGettrick, editors, *Concurrent Programming*, pages 234–255. Addison-Wesley, 1988.

[49] Narain Gehani and Andrew McGettrick, editors. *Concurrent Programming*. International Computer Science Series. Addison-Wesley, 1988.

[50] Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software—Practice and Experience*, 11(5), 1981.

[51] Morven Gentleman, Terry Shepard, and Douglas Thoreson. Administrators and multiprocessor rendezvous mechanisms. *Software—Practice and Experience*, 22(1), 1992.

[52] Ramiro González Maciel. The emissary design pattern. In *Proceedings of the Fifth Pattern Languages of Programs, Agricultural Valleys Workshop*, 1998. Retrieved September 19, 2000 from the PLoP^{TM} 1998 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P57.pdf`.

[53] Mark Grand. *Patterns in Java: a catalog of reusable design patterns*, volume 1. John Wiley and Sons, Inc., 1998.

[54] Mark Grand. *Patterns in Java: a catalog of reusable design patterns*, volume 2. John Wiley and Sons, Inc., 1999.

[55] Ennio Grasso. Synchronizer—an object behavioral pattern for concurrent programming. In *Proceedings of the Second European Conference on Pattern Languages of Programs, Distribution Patterns*, Munich, Germany: Siemens, 1997. (EuroPLoP'97) Siemens Technical Report 120/SW1/FB. Retrieved September 19, 2000 from the EuroPLoP^{TM} 1997 database on the World Wide Web: `http://www.riehle.org/events/europlop-1997/p3final.pdf`.

[56] Timothy Harrison, Douglas C. Schmidt, and Irfan Pyarali. Asynchronous completion token: An object behavioural pattern for efficient asynchronous event handling. In *Pattern Languages of Programs Design*, volume 3 of *Software Patterns Series*. Addison-Wesley, 1997. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/ACT.ps.gz`.

[57] Stephen Hartley. *Concurrent Programming: the Java programming language.* Oxford University Press, 1998.

[58] Viviane Hays, Marc Loutrel, and Eduardo B. Fernandez. The object filter and access control framework. In *Proceedings of the Seventh Pattern Languages of Programs, The Office Connections Workshop.* Pattern Languages of Programs, 2000. Retrieved December 13, 2000 from the PLoP^{TM} 2000 database on the World Wide Web: `http://jerry.cs.uiuc.edu/ plop/plop2k/Fernandez3/Fernandez3.pdf`.

[59] Richard Helm and Erich Gamma. Patterns and software design: The courier pattern. *Dr. Dobb's Sourcebook*, pages 55–59, January/February 1996.

[60] K. Hendrikx, E. Duval, and H. Olivié. Managing shared resources. In *Proceedings of the Fifth European Conference on Pattern Languages of Programs, Design and Programming Workshop*, 2000. Retrieved October 21, 2000 from the EuroPLoP^{TM} 2000 database on the World Wide Web: `http://www.coldewey.com/europlop2000/papers/hendrikx.zip`.

[61] Michi Henning and Steve Vinoski. *Advanced CORBA® Programming with C++*. Addison-Wesley, 1999.

[62] Mark Heuser and Eduardo Fernandez. RPC client: A pattern for the client-side implementation of a pipelined request/response protocol. In *Proceedings of the Sixth Pattern Languages of Programs, Group Five Workshop*. Pattern Languages of Programs, 1999. Retrieved September 14, 2000 from the PLoP^{TM} 1999 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/fernandezA/heuser003.PDF`.

[63] Robert Hirschfeld and Jeff Eastman. Lock server. In *Proceedings of the Fifth Pattern Languages of Programs, Four-Story Limit Workshop*, 1998. Retrieved September 19, 2000 from the PLoP^{TM} 1998 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P18.pdf`.

[64] Robert Hirschfeld and Jeff Eastman. Notification server. In *Proceedings of the Fifth Pattern Languages of Programs, Four-Story Limit Workshop*, 1998. Retrieved September 19, 2000 from the PLoP^{TM} 1998 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P20.pdf`.

[65] James C. Hu and Christopher D. Gill. Patterns in flexible server application frameworks. In *Proceedings of the Seventh Pattern Languages of Programs, The Unselfconscious Process Workshop*. Pattern Languages of Programs, 2000. Retrieved December 13, 2000 from the PLoP^{TM} 2000 database on the World Wide Web: `http://jerry.cs.uiuc.edu/ plop/plop2k/Hu/Hu.pdf`.

[66] Prashant Jain and Michael Kircher. Leasing. In *Proceedings of the Seventh Pattern Languages of Programs, Quiet Backs Workshop*. Pattern Languages of Programs, 2000. Retrieved October 21, 2000 from the PLoP^{TM} 2000 database on the World Wide Web: `http://jerry.cs.uiuc.edu/ plop/plop2k/Jain-Kircher/Jain-Kircher.pdf`.

[67] Prashant Jain and Michael Kircher. Lookup. In *Proceedings of the Fifth European Conference on Pattern Languages of Programs, Architecture and Design Workshop*, 2000. Retrieved October 21, 2000 from the EuroPLoP^{TM} 2000 database on the World Wide Web: `http://www.coldewey.com/europlop2000/papers/jain+kircher.zip`.

[68] Prashant Jain and Douglas C. Schmidt. Service configurator – a pattern for dynamic configuration and reconfiguration of commu-

nication services. In *Proceedings of the Third Pattern Languages of Programs, System Configuration and Resource Management Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/Service-Configurator.ps.gz`.

[69] Jean-Marc Jézéquel, Michel Train, and Christine Mingins. *Design Patterns and Contracts*. Addison-Wesley, 2000.

[70] Raman Kannan. Managing continuous data feed with subscriber/publisher pattern. *SIGPLAN Notices*, 30(10), October 1995. Retrieved September 20, 2000 from the OOPSLA 1995 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers/part.ps.gz`.

[71] Jung Kim and Kevin Benner. Implementation patterns for the observer pattern. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[72] Charles D. Knutson, Timothy A. Budd, and Curtis R. Cook. Multiparadigm patterns of thought and design. In *Joint Pattern Languages of Programs Conference, Potpourri Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/knutson.ps.gz`.

[73] Martin Kobetič and Peter Neurath. Survey of object-oriented concurrent programming - focus on actors, 1995. Retrieved January 14, 2000 from the Comenius University database on the World Wide Web: `http://object.dcs.fmph.uniba.sk/object/uploads/Diploma_theses/1995_Kobetic_Neurath/www/soocp/Soocp.htm`.

[74] R. Lavender and Douglas C. Schmidt. Active object: An object behavioural pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[75] Doug Lea. *Concurrent Programming in Java^{TM}: Design Principles and Patterns*. The Java^{TM} Series. Addison-Wesley Longman, Inc., 1997.

[76] Doug Lea. *Concurrent Programming in Java^{TM}: Design Principles and Patterns*. The Java^{TM} Series. Addison-Wesley, second edition, 1999.

[77] Doug Lea. Patterns—Discussion FAQ. *Doug Lea's Home Page*, December 1999. http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html.

[78] S.A. MacKay, W. M. Gentleman, D.A. Stewart, and M. Wein. Harmony as an object-oriented operating system. Technical Report NRC 29636, National Research Council of Canada, September 1988. Retrieved February 19, 2001 from the NRC database on the World Wide Web: http://wwwsel.iit.nrc.ca/abstracts/NRC29636.abs.

[79] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs.* John Wiley & Sons, 1999.

[80] Klaus Marquardt. Patterns for object transport. In *Proceedings of the Fifth European Conference on Pattern Languages of Programs, Design and Programming Workshop*, 2000. Retrieved October 21, 2000 from the EuroPLoP^{TM} 2000 database on the World Wide Web:http://www.coldewey.com/europlop2000/papers.html.

[81] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Forkjoin. Retrieved January 5, 2001 from the Pattern Language for Parallel Application Programming database on the World Wide Web: http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/SupportingStructures/ForkJoin.htm.

[82] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs, The Group 1 Workshop.* Pattern Languages of Programs, 1999. Retrieved September 14, 2000 from the PLoP^{TM} 1999 database on the World Wide Web: http://st-www.cs.uiuc.edu/ plop/plop99/proceedings/massingill/massingill.pdf.

[83] Paul McKenney. Selecting locking primitives for parallel programming. *Communications of the The Association for Computing Machinery*, 39(10):75–82, 1996.

[84] Frank Metayer. Matcher-handler. In *Proceedings of the Sixth Pattern Languages of Programs, The Group 2 Workshop.* Pattern Languages of Programs, 1999. Retrieved January 5, 2001 from the PLoP^{TM} 1999 database on the World Wide Web: http://st-www.cs.uiuc.edu/ plop/plop99/proceedings/Metayer/MatcherHandler.pdf.

[85] Regine Meunier. Pipes and filters architecture. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, volume 1 of *Software Patterns Series*. Adddison-Wesley, 1995.

[86] James Noble. Found objects. a pattern language for finding objects from within designs. In *Proceedings of the First European Conference on Pattern Languages of Programs, Pattern Language Workshop*, 1996. Retrieved January 21, 2000 from the EuroPLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/europlop-96/ww1-papers.html`.

[87] James Noble. Classifying relationships between object-oriented design patterns. In *Australian Software Enginnering Conference (ASWEC'98)*, May 1998. Retrieved January 16, 2001 from James Noble's draft paper database on the World Wide Web: `http://www.mri.mq.edu.au/ kjx/drafts.html`.

[88] James Noble and Charles Weir. Proceedings of the memory preservation society. In *Proceedings of the Third European Conference on Pattern Languages of Programs, Patterns of Design Workshop*, 1998. Retrieved September 14, 2000 from the EuroPLoP^{TM} 1998 database on the World Wide Web: `http://www.coldewey.com/europlop98/Program/Papers/Weir.ps.gz`.

[89] Jean-Lin Pacherie and Jean-Marc Jézéquel. The operator design pattern application to parallel computation. In *Proceedings of the Third Joint Pattern Languages of Programs, Concurrency and Operating Systems Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/jezequel.ps.gz`.

[90] Marta Patiño, Francisco Ballesteros, Ricard Jiménez, Sergio Arévalo, Fabio Kon, and Roy Campbell. `CompositeCalls`: A design pattern for efficient and flexible client-server interaction. In *Proceedings of the Sixth Pattern Languages of Programs, Group Seven Workshop*. Pattern Languages of Programs, 1999. Retrieved September 14, 2000 from the PLoP^{TM} 1999 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/ballesteros/interpfim.pdf`.

[91] Luís Moniz Pereira, Luís Monteiro, José Cunha, and Joaquin N. Aparício. Delta Prolog: A distributed backtracking extension with events. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.

[92] Dorina Petriu and Gurudas Somadder. A pattern language for improving the capacity of layered client/server systems with multi-threaded servers. In *Proceedings of the Second European Conference on Pattern Languages of Programs, Distribution Patterns*, Munich, Germany: Siemens, 1997. (EuroPLoP'97) Siemens Technical Report 120/SW1/FB. Retrieved September 19, 2000 from the EuroPLoP^{TM} 1997 database on the World Wide Web: `http://www.riehle.org/events/europlop-1997/p23final.pdf`.

[93] Wolfgang Pree. *Design Patterns for Object-Oriented Software*. Addison-Wesley, 1995.

[94] Nat Pryce. Idiom or pattern. *Portland Pattern Repository*, 8 June 1999. `http://c2.com/cgi/wiki?IdiomOrPattern`.

[95] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas Jordan. Proactor: An architectural pattern for demultiplexing and dispatching handlers for asynchronous events. In Brian Foote, Neil Harrison, and Hans Rohnert, editors, *Pattern Languages of Programs Design*, volume 4 of *Software Patterns Series*, 1999. Retrieved January 20, 2000 from the PLoP^{TM} 1997 database on the World Wide Web: `http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/Proceedings/pyarali.proactor.pdf`.

[96] Irfan Pyarali, Carlos O'Ryan, and Douglas C. Schmidt. Patterns for efficient, predictable, scalable, and flexible dispatching components. In *Proceedings of the Seventh Pattern Languages of Programs, The Network of Learning Workshop*. Pattern Languages of Programs, 2000. Retrieved December 13, 2000 from the PLoP^{TM} 2000 database on the World Wide Web: `http://jerry.cs.uiuc.edu/ plop/plop2k/Pyarali/Pyarali.pdf`.

[97] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *Operating Systems Review*, 25:47–50, April 1991.

[98] John Hamilton Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1992. Retrieved January 18, 2000 from the NECI Scientific Literature Digital Library database on the World Wide Web: `http://citeseer.nj.nec.com/104521.html`.

[99] Dirk Riehle. Bureaucracy-a composite pattern. In *Proceedings of the First European Conference on Pattern Languages of Programs, Other Patterns Workshop*, 1996. Retrieved January 21, 2000 from the EuroPLoP^{TM}

1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/europlop-96/ww3-papers.html`.

[100] Linda Rising. *The Patterns Handbook: Techniques, Strategies, and Applications.* Cambridge University Press, 1998. Collected and Introduced by Linda Rising.

[101] António Rito Silva, João Pereira, and José Alves. Object synchronization patterns. In *Proceedings of the First European Conference on Pattern Languages of Programs, Distribution Workshop*, 1996. Retrieved January 21, 2000 from the EuroPLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/europlop-96/papers/paper09.ps`.

[102] António Rito Silva, João Pereira, and Pedro Sousa. Local serialization pattern. In *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30(10). SIGPLAN Notices, October 1995. Retrieved September 20, 2000 from the OOPSLA 1995 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers/atomobj.ps.gz`.

[103] Hans Rohnert. The proxy design pattern revisited. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[104] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[105] Aamod Sane and Roy Campbell. Composite messages: A structural pattern for communication between components. In *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30(10). SIGPLAN Notices, October 1995. Retrieved September 20, 2000 from the OOPSLA 1995 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers/aamod.ps.gz`.

[106] Aamod Sane and Roy Campbell. Resource exchanger: A behavioural pattern for low-overhead concurrent resource management. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*, pages 461–473. Addison-Wesley, 1996.

[107] Douglas C. Schmidt. Family of design patterns for application-level gateways. *Theory and Practice of Object Systems*, 2(1), December 1996. Special issue on Patterns and Pattern Languages. Retrieved January 20, 2000 from the ACE database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PDF/TAPOS-00.pdf`.

[108] Douglas C. Schmidt. Applying patterns and frameworks to develop object-oriented communication software. In Peter Salus, editor, *Handbook of Programming Languages*, volume 1. MacMillan Computer Publishing, 1997.

[109] Douglas C. Schmidt. Strategized locking, thread-safe interface, and scoped locking: Patterns and idioms for simplifying multi-threaded C++ components. *C++ Report*, 11(9), September 1999. Retrieved January 20, 2000 from the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PDF/locking-patterns.pdf`.

[110] Douglas C. Schmidt. Monitor object—an object behavioral pattern for concurrent programming. *C++ Report*, 2000. Retrieved September 20, 2000 from the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PDF/monitor.pdf`.

[111] Douglas C. Schmidt and Timothy Harrison. Double-checked locking. an object behavioral pattern for initializing and accessing thread-safe objects efficiently. In Robert C. Martin, Dirk Riehle, Frank Buschmann, and John Vlissides, editors, *Pattern Languages of Programs Design*, volume 3 of *Software Patterns Series*. Addison-Wesley, 1997. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/DC-Locking.ps.gz`.

[112] Douglas C. Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann. Leader/followers: A design pattern for efficient multi-threaded event demultiplexing and dispatching. In *Proceedings of the Seventh Pattern Languages of Programs, The Network of Learning Workshop*. Pattern Languages of Programs, 2000. Retrieved December 13, 2000 from the PLoP^{TM} 2000 database on the World Wide Web: `http://jerry.cs.uiuc.edu/ plop/plop2k/ORyan/ORyan.pdf`.

[113] Fred Schneider. *On Concurrent Programming*. Springer-Verlag, 1997.

[114] Jean-Francois Selber and Gilles Le Goff. Task manager design pattern. In *Proceedings of the Fourth European Conference on Pattern Languages*

of Programs, Patterns of Design Workshop, 1999. Retrieved September 14, 2000 from the EuroPLoP^{TM} 1999 database on the World Wide Web: `http://www.argo.be/europlop/Papers/Final/Goff.doc`.

[115] Mary Shaw. Some patterns for software architecture. In John M. Vlissides, James O. Coplien, and Normal L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[116] J. A. Simpson and E. S. C. Weiner, editors. *The Oxford English Dictionary*. Oxford University Press, 2 edition, 1989. Retrieved November 6, 2000 from The Oxford English Dictionary database on the World Wide Web: `http://daisy.uwaterloo.ca/~fwtompa/oed/oed-local/lookup.cgi`.

[117] Peter Sommerlad. Command processor. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[118] Peter Sommerlad and Frank Buschmann. The manager design pattern. In *Joint Pattern Languages of Programs Conference, System Configuration and Resource Management Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/ sommerlad.ps.gz`.

[119] Peter Sommerlad and Michael Stal. The client-dispatcher-server design pattern. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Programs Design*, volume 2 of *Software Patterns Series*. Addison-Wesley, 1996.

[120] Michael Stal. The broker architectural framework. In *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30(10). SIGPLAN Notices, October 1995. Retrieved September 20, 2000 from the OOPSLA 1995 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers/broker.ps.gz`.

[121] Darlene Stewart and W. Gentleman. Non-stop monitoring and debugging on shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97)*. Institute of Electrical and Electronics Engineers, Inc., 1997. Retrieved January 19, 2000 from the National Research Council of Canada database on the World Wide Web: `http://wwwsel.iit.nrc.ca/abstracts/NRC40147.abs`.

[122] P.P. Tanner, S.A. MacKay, D.A. Stewart, and M. Wein. A multi-taasking switchboard approach to user interface management. In *Proceedings of the 13th annual conference on Computer graphics (SIGGRAPH '86), Computer Graphics*, volume 20, 1986. Retrieved March 1, 2001 from the ACM Digital Library database on the World Wide Web: `http://info.acm.org/pubs/citations/proceedings/graph/15922/p241-tanner`.

[123] Jean Tessier and Rudolf Keller. Manager-agent and remote operation: Two key patterns for network management interfaces. In *Proceedings of the Third Joint Pattern Languages of Programs, Frameworks and Architectures Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/keller.ps.gz`.

[124] Walter F. Tichy. Essential software design patterns. Retrieved November 10, 2000 from the World Wide Web: `http://wwwipd.ira.uka.de/~tichy/patterns/concurrency.html`.

[125] Walter F. Tichy. A catalogue of general-purpose software design patterns. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 23)*. IEEE Computer Society, 1998. (invited paper).

[126] Allan Vermeulen. An asynchronous design pattern. *Dr. Dobb's Journal*, 21(6), 1996.

[127] Allen Vermeulen, Gabe Beged-Dov, and Patrick Thompson. The pipeline design pattern. In *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30(10). SIGPLAN Notices, October 1995. Retrieved September 20, 2000 from the OOPSLA 1995 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers/quilt.ps.gz`.

[128] Mauricio J. Vianna e Silva, Sergio Carvalho, and John Kapson. Patterns for layered object-oriented applications. In *Proceedings of the Second European Conference on Pattern Languages of Programs, Distribution Patterns Workshop.* (EuroPLoP'97) Siemens Technical Report 120/SW1/FB, 1997. Retrieved October 21, 2000 from the EuroPLoP^{TM} 1997 database on the World Wide Web: `http://www.riehle.org/events/europlop-1997/p5final.pdf`.

[129] John Vlissides. *Pattern Hatching: Design Patterns Applied.* The Software Pattern Series. Addison-Wesley, 1998.

[130] Eugene Wallingford. The sponsor-selector pattern. In *Proceedings of the Third Joint Pattern Languages of Programs, System Configuration and Resource Management Workshop*, 1996. Retrieved January 20, 2000 from the PLoP^{TM} 1996 database on the World Wide Web: `http://www.cs.wustl.edu/~schmidt/PLoP-96/wallingford.ps.gz`.

[131] Charles Weir. Architectural styles for distribution. In *Proceedings of the Second European Conference on Pattern Languages of Programs, Distribution Workshop*, 1997. Retrieved January 21, 2000 from the EuroPLoP^{TM} 1997 database on the World Wide Web: `http://www.riehle.org/events/europlop-1997/p21final.pdf`.

[132] Gregory V. Wilson. *Practical Parallel Programming.* MIT Press, 1995.

[133] Kirk Wolf and Chamond Liu. New clients with old servers. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, volume 1 of *Software Patterns Series.* Adddison-Wesley, 1995.

[134] Bobby Woolf. The object recursion pattern. In *Proceedings of the Fifth Pattern Languages of Programs, Site Repair Workshop*, 1998. Retrieved October 21, 2000 from the PLoP^{TM} 1998 database on the World Wide Web: `http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P21.pdf`.

[135] Walter Zimmer. Relationships between design patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Programs Design*, number 1 in Software Patterns Series, pages 345–364. Addison-Wesley Publishing Company, Inc., 1995.

Index