OpenWebServer: An Adaptive Web Server Using Software Patterns

Junichi Suzuki and Yoshikazu Yamamoto

Keio University

ABSTRACT The explosive growth of the Web requires servers to be extensible and configurable. This article describes our adaptive Web server, OpenWebServer, which uses the Reflection architectural pattern. The server supports the dynamic adoption of functionality, such as introducing additional protocols, modifying execution policies, and tuning system performance. This is achieved by specifying and coordinating metaobjects that represent various aspects within the Web server. We present a Java version of OpenWebServer, and describe its design using Reflection and other design patterns: Singleton, Bridge, Mediator, Observer, and Decorator. These patterns provide a better-factored design and allow the Web server to evolve continually beyond static and monolithic servers.

As the Internet becomes more ubiquitous, the Web is increasingly used for many purposes. The explosive growth of the Web places larger and more challenging demands on servers. An effective design for Internet-based servers is required. This article describes our experience in designing a Web server that addresses many of the current challenges facing Web servers.

CHALLENGES FACING WEB SERVERS

Current Web servers must:

- Connect with various systems such as groupware, database management systems, mobile agent engines, and transaction processing monitors.
- Integrate generic communication environments including Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM)
- Extend server functionality by introducing additional network protocols or content data types
- Change server execution policies; for example, optimizing concurrency, connection management, request handling, and cache management.
- Adapt to execution environments such as ATM networks and electronic devices such as network routers, printers, or copiers

Every user may not require the same functionality in a Web server. Therefore, a Web server should be flexible enough to meet a wide range of requirements on demand. Most Web servers, unfortunately, are monolithic. They provide a fixed and limited set of capabilities. When additional functionality is required, the usual solution involves shutting down the system, modifying one or more components, integrating them with existing components, and restarting the system. This solution is often difficult or expensive to maintain, and does not allow the server to be used during the upgrade. Therefore, it is typical to take the "scrap-and-build" approach for a given requirement, where the software is rewritten from scratch because it may be more economically feasible. A dynamically adaptable Web server architecture based on reusable components is an attractive alternative for extensive and intrusive changes.

ADAPTABILITY OF WEB SERVERS

Most Web servers lack the adaptability to enable the system's evolution. Designers cannot know or predict all possible uses of the system. A service or configuration that is appropriate at one point may not be useful later, and the system may not evolve transparently.

The *Reflection* architectural pattern enables the system to maintain information about itself and use this information to remain changeable and extensible [1]. This article addresses the problem of adaptability and describes our use of patterns to build an adaptive Web server. We consider the use of Reflection for specifying various aspects (e.g., structure and/or behavior) of an open-ended system that can be dynamically adapted. OpenWebServer is developed on top of the Adaptive Internet Server Framework (AISF) [2]. This article presents a Java version of OpenWebServer.

The remainder of this article is organized as follows. The next section introduces the concept and benefits of Reflection. We then present the advantage of developing a Web server based on design patterns. The article goes on to present some applications of the adaptability of OpenWebServer. We describe design patterns used in the implementation phase, and conclude with a note on the current status of the project and future work.

THE REFLECTION ARCHITECTURAL PATTERN

In general, the meta-architecture that Reflection embodies introduces the notion of *object/metaobject separation*. A metaobject or *metalevel object* contains information about the internal structure and/or behavior of one or more *baselevel objects* or *baseobjects*, which includes the application logic. In other words, metaobjects can track and control certain aspects (e.g., structure and/or behavior) of baseobjects. A set of metaobjects is called a *metaspace* or *metalevel*. A set of baseobjects is called a *baselevel*.

Reflection is the ability of a program to manipulate as data something that represents the state of the program to adjust to changing requirements. The goal of reflection is to allow a baseobject to reflect on its own execution state and eventually alter it to change its meaning during execution. In contrast to reflection, *reification* is the process of making something accessible that is normally unavailable in the baselevel (e.g., programming environment) or hidden from the programmer. For the execution of a baseobject to be supervised, it must first be reified into the corresponding metalevel. The interfaces a baseobject uses to access its metalevel are called *metaobject protocols* (MOPs). The relationships are illustrated in Fig. 1.

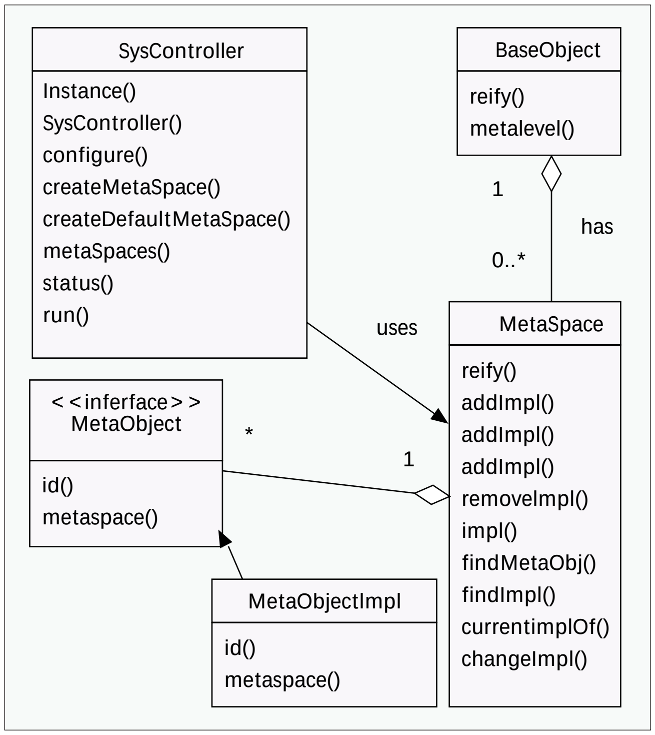
The Conceptual Framework and Advantages of OpenWebServer

OpenWebServer contains a metalevel that supports a wide range of aspects of Web servers. It provides a set of fine-grained metaobjects and supplemental utility objects to support the writing of both the baselevel and metalevel. OpenWebServer is implemented within the programmable metalevel and can dynamically adjust itself so that it is executed in the best-tuned condition for a given requirement.

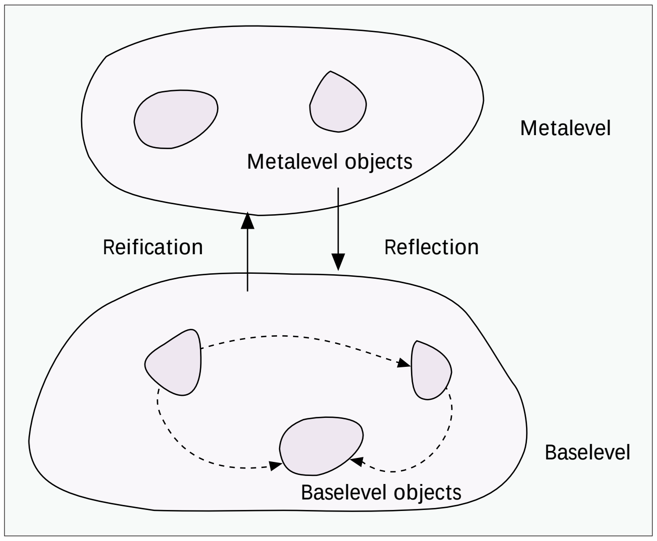
Although modern Web servers provide some extension mechanisms such as Common Gateway Interface (CGI), server-side APIs, and server-side scripting capability, their extensibility is restricted within the application level. In contrast, OpenWebServer provides a uniform platform where a variety of requirements can be specified from low-level services, such as connection management, request handling, and cache management, into application-level services without breaking the single framework. In other words, the metalevel in OpenWebServer plays the role of a generic "change absorber" for the Web server.

The Reflection pattern provides the following advantages:

- Separation of concerns: In conventional Web servers, the system's basic mechanisms are complicated by policy specifications. This makes it difficult to understand and maintain the system. The separation of the reflective facilities from the underlying mechanisms allows the reuse of feasible policies.
- Improved adaptability and configurability: The metalevel keeps the baselevel open-ended for extension. New requirements can be implemented with relatively minor modifications to baseobjects within the original system. This eliminates the "scrap-and-build" solution in system development.
- Transparency: Metaobject protocols decouple the metalevel and baselevel. This indirection reduces the constraints on both, so the levels can be developed independently. This transparency allows the metalevel to evolve while maintaining backward compatibility with the baselevel.



■ **Figure 2.** *Classes in the* `jp.ac.keio.ows.kernel` *package.*



■ **Figure 1.** *A typical reflective architecture.*

OpenWebServer is named after the Open-Closed Principle (OCP), which was originally proposed by Bertrand Meyer and states that software entities should be open for extension but closed for internal modification. This means we should design systems so that objects can be extensible by adding new objects via inheritance or composition, not by modifying the object's internal working code. OpenWebServer is intended to apply the Open-Closed Principle so that the system can evolve by adding new metaobjects or reorganizing them.

The Metalevel Design of OpenWebServer

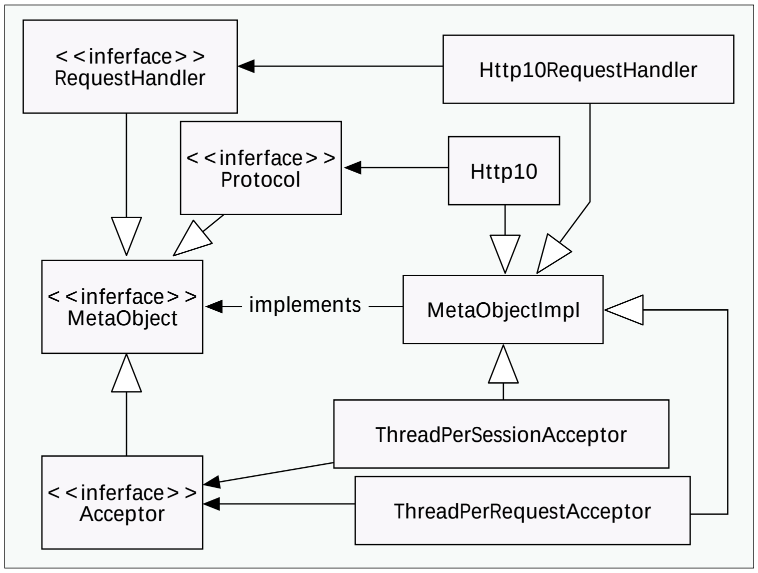
This section presents an architectural overview of OpenWebServer and describes the design patterns used to construct the metalevel.

Architectural Overview

OpenWebServer consists of three packages. The `jp.ac.keio.ows.kernel` package contains the foundation objects to construct and maintain the metalevel. The `jp.ac.keio.ows.meta` package contains series of metaobjects and is controlled by the `kernel` package. The `jp.ac.keio.ows.utility` package contains utility objects to support the writing of both the baselevel and metalevel. It is used by `kernel` and `meta` packages.

The `jp.ac.keio.ows.kernel` package is organized as shown in Fig. 2, and contains the following objects:

- `SysController` — Starts the system by creating appropriate metaspaces and metaobjects with a configuration file. It also keeps track of system status and configuration. This object is also responsible for stopping and resuming the system. It is an active object executed on a root thread in the thread hierarchy.
- `MetaSpace` — Represents a metalevel or metaspace. Multiple metalevels can exist within the system, although it usually has a single metalevel. It references every metaobject and coordinates the interaction between them to meet a given requirement. This object is discussed later.
- `MetaObject` — Specifies the interfaces for all metaobjects. This is a base interface class for them. MetaObjectImpl and its subclasses provide the implementations of this object. The relationship between MetaObject and MetaObjectImpl is discussed next.



■ **Figure 3.** *Classes in the* `jp.ac.keio.ows.meta` *package.*

- `MetaObjectImpl` — Provides an implementation for a MetaObject. Multiple implementations can be defined for a single MetaObject.
- `BaseObject` — Specifies the interfaces needed to all baseobjects. This is a base class for them.

`SysController` is a Singleton, since it has exactly one instance in the system. The Singleton pattern ensures that a class has just one instance and provides controlled access to it [3]. The following shows its selected interfaces.

```
public class SysController implements Runnable {
  static protected SysController Instance = null;
  protected Thread thread;
  public static SysController Instance(String param){
    if(Instance == null){
      Instance = new SysController(param);
    }
      return Instance;
  }
  protected SysController(String param) {
    // ...
    thread = new Thread(this);
    thread.start();
  }
}
```

The `static Instance()` method is used to instantiate or return the unique instance. The constructor is protected and called by `Instance()`. `SysController` creates one or more instances of `MetaSpace` and holds a set of references to them.

`MetaSpace` is an entry point from the baselevel to metalevel. Baseobjects can access `MetaSpace` when communicating with their metalevels. Each baseobject can have zero or more instances of `MetaSpace`. In turn, each `MetaSpace` aggregates zero or more metaobjects (Fig. 2).

Baseobjects do not have to be attached to a metalevel, but can be dynamically attached on demand. Attachment on demand is known as lazy reification. Each baseobject can be reified with its method `reify()`, and access its metalevel with the method `metalevel()` (Fig. 2). Once a baseobject is reified, it becomes aware of its metalevel, and the corresponding metaobjects are then instantiated by the class `MetaSpace`. The number and type of created metaobjects depends on what the users wish to do.

The `jp.ac.keio.ows.meta` package consists of a collection of metaobjects (Fig. 3). To specify metaobjects, we identified services and entities by looking for typical events or constructs found during the execution of Web servers. Abstracting these events and constructs, OpenWebServer provides the following metaobjects by default:

- `Initializer` — Initializes the network facility along with the current configuration. A typical task is to create one or more sockets according to the current communication protocol and concurrency policy, and then instantiate an `Acceptor`.
- `Acceptor` — Waits for and accepts incoming requests. It encapsulates the different concurrency policies for simultaneous access (discussed later). Once it obtains a request, it asks a `RequestHandler` to process the request given the current configuration (i.e., protocol and concurrency policy).
- `RequestHandler` — Deals with requests from an `Acceptor`. It encapsulates the different policies for interpreting a request, based on the kind of request and target content. It is created on a per-request basis when an `Acceptor` accepts a request. It is executed on a separate thread or the same thread on which an Acceptor runs.

`Protocol` — Defines protocol-specific information on a per-protocol basis. It is referenced by a `RequestHandler`.
- `ContentFinder` — Finds a target resource (e.g., HTML/XML files or data within a back-end repository) passed by a `RequestHandler`.
- `Logger` — Records accesses to the Web server. `RequestHandler` typically calls it.
- `ExecManager` — Executes external entities like CGI scripts.

These metaobjects represent typical aspects of Web servers. They are objects that affect the behavior of other objects, and have the following basic responsibilities [1]:
- Encapsulating system internals that may change
- Providing an interface to facilitate modifications to the metalevel
- Controlling baselevel behavior

As shown in Fig. 3, all metaobjects are interface classes derived from `MetaObject` in the `kernel` package. We can add a new metaobject depending on the requirements by deriving from `MetaObject`. Implementations of a metaobject are provided by implementation classes derived from `MetaObjectImpl`. Figure 3 shows two implementations attached to `Acceptor`. `ThreadPerRequestAcceptor` and `ThreadPerSessionAcceptor` implement different concurrency policies of thread per request and thread per session, respectively. Implementation classes are also contained in the meta package.

OpenWebServer eliminates the explicit level shifting from baselevel to metalevel, and the explicit reflective function calls found in the traditional reflective programming languages. This is similar to the approach introduced by CodA, a Smalltalk-based meta-architecture [4]. Metaobjects are just like other objects in the system. Reflective computation is executed by direct interaction with the desired metaobjects. Methods of every metaobject within the `kernel` and `meta` packages are considered MOPs.

Design Patterns for Modeling Metaobjects

***Bridge: Decoupling an Metaobject and Its Implementations* —** The metaobjects described in the previous section are abstractions of typical aspects of Web servers, and define their interface and semantics. Their concrete implementations are prepared as classes that implement them and derive from MetaObjectImpl (Fig. 3 and 4). The relationship between a metaobject and its implementations is based on *Bridge* [3]. The pattern explicitly decouples an interface and its implementation. Permanent binding between them should be avoided in OpenWebServer. The implementation must be dynamically chosen or changed for system adaptability. Also, both the interface and the implementations should be independently extensible; changes in an implementation should have no impact on clients of the interface.

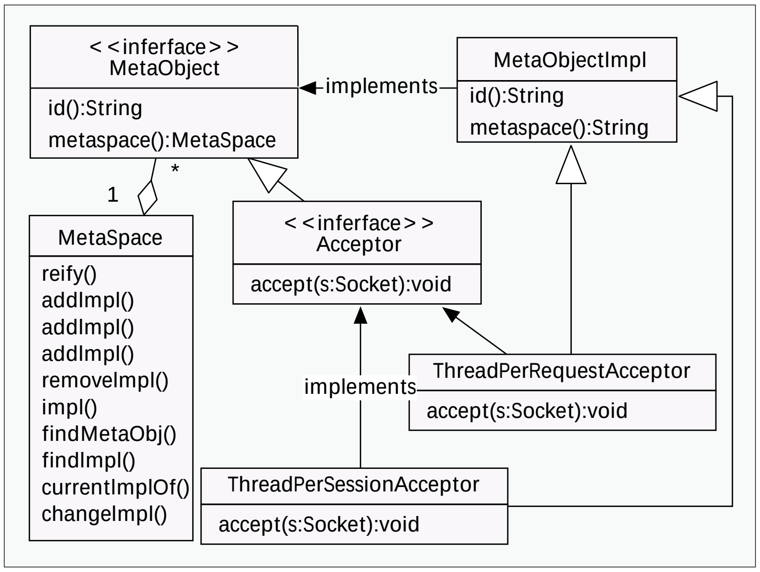
Bridge separates an interface and its implementation using object composition instead of inheritance, which is typically used for incremental modifications. Object composition should be favored over inheritance, since object composition decouples the early and permanent binding between an object and the semantics of its behavior, and provides more runtime flexibility [2, 3].

OpenWebServer uses a variant of Bridge which includes the interface-implementation relationship, provided by Java, between the interface and its implementation. Figure 4 depicts a class diagram where this pattern is applied to Acceptor. ThreadPerRequestAcceptor and ThreadPerSessionAcceptor are implementation classes of Acceptor. These classes implement accept() differently, which contains an infinite loop to wait for incoming requests and dispatches them to a RequestHandler along with the current concurrency policy.

In our variant of Bridge, MetaSpace aggregates implementation objects so that it can change the metaobject's implementation dynamically. It can inspect the current implementation with its currentImpl(), and change it using changeImpl(). The next section describes how MetaSpace maintains a set of metaobjects and their implementations internally.

Bridge allows changing the metaobject's implementation at runtime, depending on the desired execution policy in the Web server. It allows dynamic adaptation without system shutdown. Bridge has the following benefits for OpenWebServer:

- Decoupling abstraction and implementations.
- Layered architecture that separates the core mechanism and its policy explicitly. This leads to a system that is better factored and easier to maintain.
- Improved adaptability.

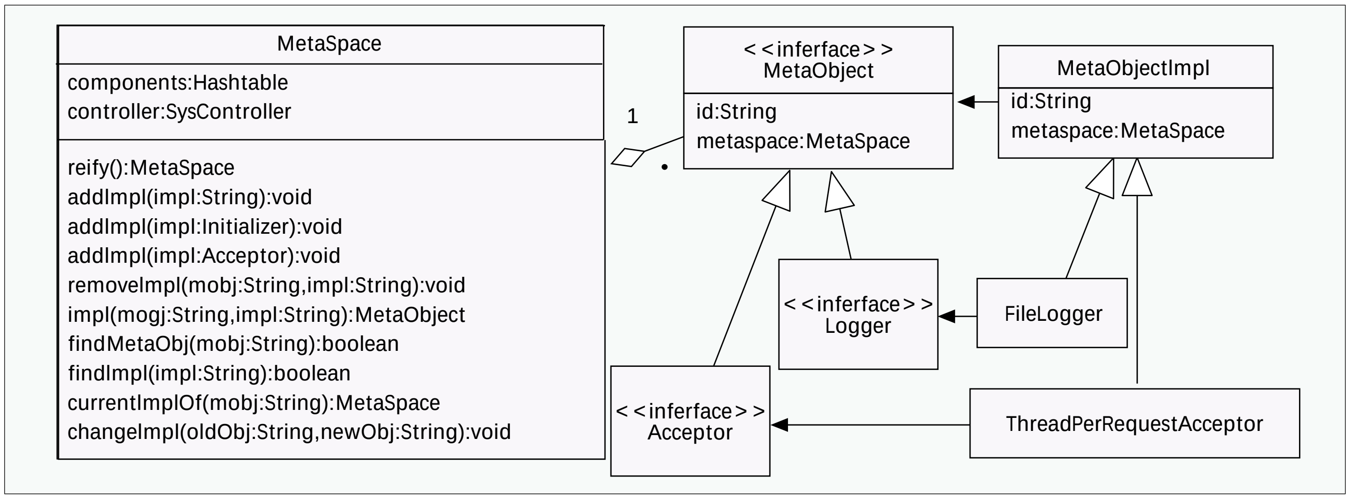


■ **Figure 4.** *Class structure of a metaobject and its implementations.*

Design Patterns for Coordinating Metaobjects

OpenWebServer encourages metaobjects to specify aspects in Web servers and interact with each other to meet design demands. This often causes multiple associations among metaobjects. In the worst case, every metaobject should know all others. Although partitioning a system into well-defined objects generally enhances reusability, lots of interconnections make it less likely that a metaobject can work without the support of others. As a result, the system behaves as if it were monolithic. Moreover, it would be difficult to change system's behavior transparently due to the spaghetti associations.

Mediator: Isolating the Associations Between Metaobjects — To avoid this situation, we use *Mediator* [3], which encapsulates the interaction of a set of objects and facilitates loose coupling among them by keeping the reference to every object within a mediator object. In OpenWebServer, MetaSpace is a mediator object. It controls and coordinates interactions among a set of metaobjects. It acts as an intermediary among the metaobjects in the metalevel. Each metaobject knows only MetaSpace, not any other metaobject, thereby reducing the number of interconnections.



MetaSpace
components:Hashtable
controller:SysController
reify():MetaSpace
addImpl(impl:String):void
addImpl(impl:Initializer):void
addImpl(impl:Acceptor):void
removeImpl(mobj:String,impl:String):void
impl(mogj:String,impl:String):MetaObject
findMetaObj(mobj:String):boolean
findImpl(impl:String):boolean
currentImplOf(mobj:String):MetaSpace
changeImpl(oldObj:String,newObj:String):void
1
•
<<inferface>>
MetaObject
id:String
metaspace:MetaSpace
MetaObjectImpl
id:String
metaspace:MetaSpace
<<inferface>>
Logger
FileLogger
<<inferface>>
Acceptor
ThreadPerRequestAcceptor

■ **Figure 5.** *A class structure of MetaSpace, metaobjects, and their implementations based on Mediator. MetaSpace is an intermediary among metaobjects and encapsulates their interactions, instead of metaobjects being connected to each other directly.*

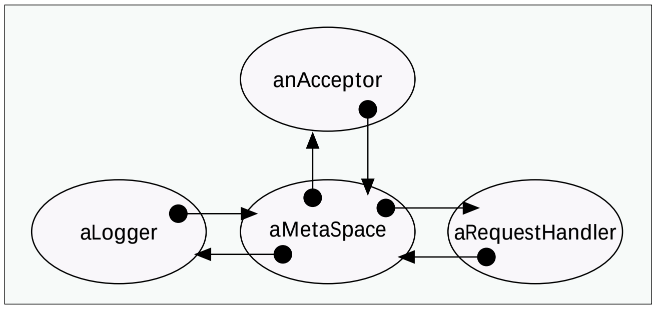


Figure 6. *The runtime object structure in the Mediator-based metalevel. Here, three metaobjects interconnect indirectly via `MetaSpace`.*

Figure 5 shows how Mediator is applied to the metalevel of OpenWebServer. In this figure, `MetaSpace` is an intermediary among `Acceptor` and `Logger`. Each metaobject has an attribute metaspace to refer to the `metaspace` to which it belongs, and can call `curentImplOf()` of `MetaSpace` to get the current implementation of a desired metaobject. Figure 6 depicts a runtime object structure where three metaobjects indirectly interconnect with `MetaSpace`. `MetaSpace` contains every metaobject and its implementations in the instance variable `components`, typed `Hashtable`. This variable has the mapping of a string entry (key) and `Vector` type value. The former is used to assign the string name of a metaobject, and the latter is for the sequence of implementation objects. The current implementation is assigned to the first element of the vector. The method `currentImplOf()` returns the first element, and `changeImpl()` changes the order of elements in the vector. To add an implementation object, the method `addImpl()` is used, which is prepared for every type of metaobject.

Mediator provides the following benefits for OpenWebServer:
• Encapsulation of the interactions between metaobjects
• Loose connections between metaobjects and simplified protocols between them
• Centralized control for metaobjects

Mediator+Observer: Propagating the Change of Metaobjects — Decoupling a metaobject and its implementations causes the changes in each metaobject to be localized when the metaobject dynamically alters its implementations. Such changes, however, will occasionally affect other metaobjects and require the event to be transferred to them. For example, a change in the implementation of the `Protocol` metaobject requires reconfiguration of `RequestHandler`. Therefore, `MetaSpace` must propagate change events to the metaobjects that are interested in the change. Every metaobject should not know the metaobjects that depend on its change.

In this situation, *Observer* [3] can be used for metaobject-mediator communication. This pattern encapsulates the interactions between objects by using a subject to store the state information for observer objects. Figure 7 shows that metaobjects (and their implementations virtually) are observers of `MetaSpace`. `Observable` is a subject class, and `Observer` is an observer interface class. Whenever the configuration of a metaobject changes, `MetaSpace` propagates the event to other metaobjects using its method `notifyObservers()`. Metaobjects responds to the event with `update()`. Mediator is also an observer for `SysController`, because `SysController` sends the system status to `MetaSpace`.

`SysController`, `MetaSpace`, and some metaobjects are usually executed on different threads. Therefore, it is possible to cause deadlock if an `Observable` issues a change notification in a thread while an `Observer` is trying to check the `Observable`'s instance variable. To avoid the potential deadlock, `MTObservable` is introduced (Fig. 7). It uses `Notifier` to issue the event notification in a new thread. A `Notifier` is created for a single notification to an observer. This concurrent variant of Observer is borrowed from [5]. Note that

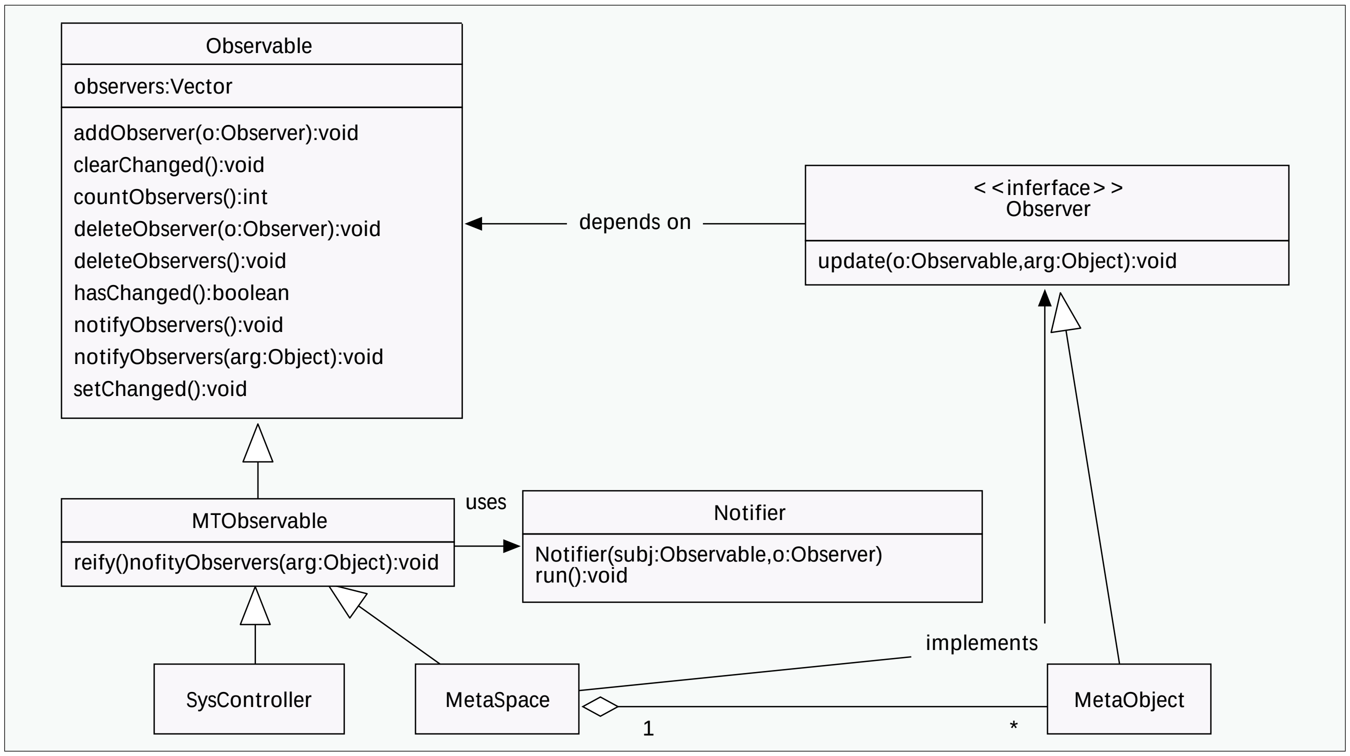


Figure 7. *A refined relationship for event notification between metaobjects and `MetaSpace` with Observer.*

`Observable` and `Observer` are not those in the `java.util` package, because the instance variable `observers` of `Observable` in JDK is private instead of protected; thus, it cannot be used from `MTObservable`.

Observer provides the following benefits for OpenWebServer:
- Loose coupling between metaobjects and the mediator
- Centralized control to propagate the change events
- Simplified and more abstract protocol between the mediator and metaobjects

DESIGN PATTERNS USED FOR UTILITY OBJECTS

OpenWebServer provides supplemental utility objects that allow the asynchronous I/O capability used by a concurrency policy called *single-threaded with I/O multiplexing* (described later). These objects are plugged into a Java package handling I/O streams named `java.io`. This package uses *Decorator* [3]. Figure 8 shows that `AsyncInputStream` and `NonBlockingStream` are derived from `java.io.FilterInputStream`. The difference in these classes is whether the method `available()` of `java.io.InputStream` is used or not. This method returns the number of bytes that can be read from the stream without blocking. `NonBlockingStream` precedes all `read*()` methods with a call to `available()` to ensure that there are data available. The method `available()`, however, does not work well with certain mechanisms like the network socket, and `read()` may not block if `available()` returns 0 [6]. `AsyncInputStream` reports the correct number of bytes that can actually be read asynchronously without using the derived `available()`. It can get available data in a nonblocking manner by spawning a thread that calls blocking `read()` exclusively.

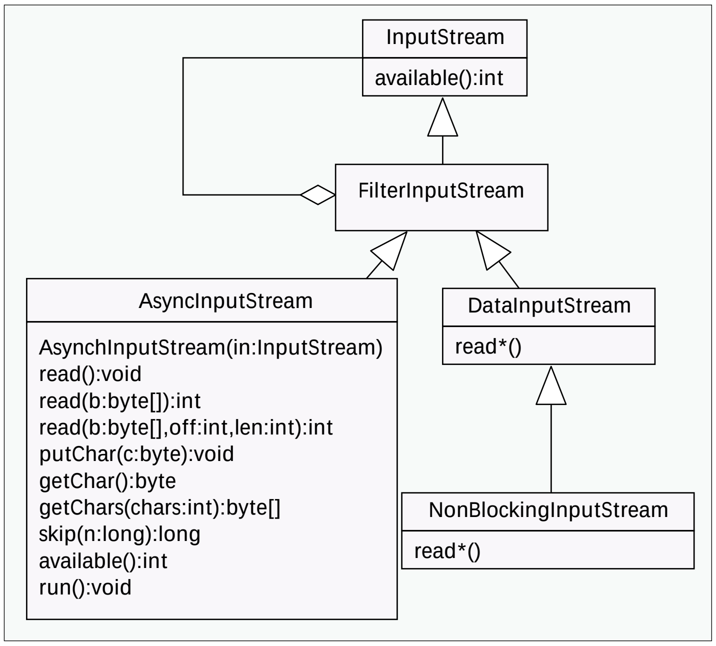


Figure 8. *`AsyncInputStream` and `NonBlockingStream` are used for the asynchronous I/O capability. They are plugged into the Decorator class structure.*

APPLICATIONS

Our first application is to support dynamic reconfiguration of concurrency policy. OpenWebServer provides a series of implementations for `Acceptor`, and can tune the policy at runtime. It supports the following policies:
- Process per request: A simple model to implement, but resource-intensive and requires too much overhead.
- Process pool: Alleviates the overhead in the above forking model; requires mutual exclusion.
- Thread-per-request: Much faster than the forking policy. However, it is not portable for different platforms.
- Thread-per-session: Less resource-intensive than the thread-per-request policy. However, it requires that endpoints use HTTP 1.1 or that the server detect the location of clients for every incoming request.
- Thread pool: Alleviates the overhead of the threading policy; requires mutual exclusion.
- Single-threaded I/O multiplexing: Conserves resources and is highly portable; also has less overhead. However, it is fault-sensitive, and the number of simultaneous connections is limited.

For example, OpenWebServer can start as a single-threaded server with I/O multiplexing, and then change itself into a threaded server when the workload (i.e., the access rate) of the server exceeds the predefined threshold.

The second application is dynamic adaptation of communication protocols. OpenWebServer provides configurations for HTTP 0.9, 1.0, and 1.1 as implementation classes of `Protocol`.

OpenWebServer also has application-level components. We have prepared a CGI capability as an implementation class for `ExecManager`.

Also, it has been integrated with a computer-supported collaborative work (CSCW) environment where the software design information is shared within the distributed development team [7]. OpenWebServer prepares an implementation for `ContentFinder`, which finds a requested Extensible Markup Language (XML) document. It also provides utility objects that parse Hypertext Markup Language (HTML)/XML documents, create their syntax trees, and retrieves elements and attributes in the parsed tree.

The personalization service of HTML/XML documents is implemented with OpenWebServer [8]. The dynamic generation of document presentation is performed based on the context of client-side environment, users, and user behaviors. The implementations for `RequestHandler`, `ContentFinder`, and `CacheManager` are provided for this purpose.

Our last application is to integrate OpenWebServer with the Document Object Model (DOM) interface and CORBA [7]. DOM is a standard of the World Wide Web Consortium, which provides a series of interfaces for accessing and manipulating XML documents.

DISCUSSION AND RELATED WORK

To evolve the metalevel of OpenWebServer, metaobjects should be fine-grained and blackbox so that they are plug-compatible each other. *Evolving Framework* [9] is a pattern language, a set of patterns that describes how to start building a reusable framework, refactor it and support the development process with appropriate tools. It includes a pattern, *Three examples*, which should be applied to the development of a framework at first. This pattern enables a developer to find hidden abstractions and rebuild proper abstractions by developing at least three applications. We have developed six applications, as described earlier, and have tried to be sure the metalevel contains proper abstractions (i.e., metaobjects). We plan to develop other applications that are even more different from previous ones. *Evolving Framework* also includes *Hot spots*, a pattern to identify hot spots and encapsulate them with design patterns. Hot spots are the aspects of a problem domain that must be kept flexible. In OpenWebServer, a series of metaobjects represent hot spots and encapsulate some types of changes. Developing its subsequent applications would allow us to find hidden hot spots and refine the metalevel. Currently, we are applying *Fine-grained objects* and *Black-box framework*.

Despite its flexibility, Reflection has some liabilities [1]:

- Modifications at the metalevel may cause damage: The robustness of a metalevel is quite important, because incorrect modifications of the metalevel may cause serious damage to the system. The current OpenWebServer cannot detect potential errors that might be caused by the change specifications. Metalevels that allow the safe deletion of metaobjects are of particular interest.
- Increased number of components: The greater the number of aspects defined at the metalevel, the greater the number of metaobjects. This may add unnecessary complexity for a simple Web server and increase the difficulty of maintaining the metalevel. We are investigating a mechanism that keeps the metalevel lightweight by allowing the safe deletion of metaobjects.
- Lower efficiency: Reflective systems are slower than nonreflective systems, because a single task requires interaction between the baselevel and metalevel. Thus, OpenWebServer is slower than static and monolithic servers. However, we can adjust the trade-off between dynamic adaptability and performance with the mechanism of lazy reification.

Current Project Status and Future Work

The current OpenWebServer includes eight metaobjects, 29 implementation objects, and 30 utility objects. It was initially implemented with the Python programming language and later with Java. We are investigating other languages to illustrate that the architectural design of OpenWebServer does not depend on a specific language.

As for the metalevel in OpenWebServer, we are aggressively making metaobjects fine-grained. At present, we are dividing `Acceptor` into different metaobjects that deal with concurrency and I/O, as described in [10], because the current `Acceptor` is somewhat coarse. Also, we are experimenting with an alternative mechanism to coordinate metaobjects. As described earlier, OpenWebServer incorporates some design patterns to achieve system evolution. However, composing and coordinating metaobjects to introduce new system behavior requires precise knowledge of the interfaces and implementations of all the metaobjects. Few methodologies have been proposed for metaobject composition. We are developing a framework that enables the relationships between baselevel and metalevel as well as among metaobjects to be more configurable, and to hide the details of the coordination process. We are particularly interested in the deletion of metaobjects from the metalevel. The deletion of metaobjects often brings unexpected fatal errors, while the addition of metaobjects can be handled seamlessly. Our goal is to provide a highly scalable metalevel.

We are also developing applications for OpenWebServer to demonstrate the power of its metalevel and improve it. We plan to introduce additional communication protocols such as Lightweight Directory Access Protocol and Simple Network Management Protocol. We also plan to provide real-time streaming functionality for continuous media using Real-time Transport Protocol or Real-Time Streaming Protocol. New underlying environments of OpenWebServer are also planned including CORBA, embedded environments, and real-time operating systems.

Conclusion

This article addresses how Web servers can meet diverse requirements and describes the advantage of using software patterns that make them adaptable and configurable. We use *Reflection*, *Singleton*, *Bridge*, *Mediator*, *Observer*, and *Decorator* to achieve it. With these patterns, OpenWebServer makes its aspects open-ended for extension, and allows itself to continually evolve beyond the static and monolithic servers of today. The information on our project is maintained at www.yy.cs.keio.ac.jp/~suzuki/project/aisf/. The original version of this article can be also found at this URL, which does not remove some sentences, figures, and references due to editorial limits.

Acknowledgments

We would like to thank Linda Rising, Haruo Akimaru, and Shigeki Yamada for their invitation to this special issue. We are also grateful to Mike Duell, Jeff Garland, and John Goodsen for their helpful comments and suggestions for earlier drafts.

References

[1] F. Buschmann *et al.*, *A System of Patterns: Pattern-Oriented Software Architecture*, Wiley, 1996.
[2] J. Suzuki and Y. Yamamoto, “Building an Adaptive Web Server with a Meta-Architecture: AISF approach,” *Proc. SPA ’98*, Mar. 1998.
[3] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[4] J. McAffer, “Engineering the Meta Level,” *Proc. Reflection ’96*, 1996.
[5] D. Lea, *Concurrent Programming in Java: Design Principle and Patterns*, Addison-Wesley, 1997.
[6] S. Oaks and H. Wong, *Java Threads*, O’Reilly, 1997.
[7] J. Suzuki and Y. Yamamoto, “Toward the Interoperable Software Design Models: Quartet of UML, XML, DOM and CORBA,” submitted to IEEE ISESS ’99, 1999.
[8] J. Suzuki and Y. Yamamoto, “Document Brokering with Agents: Persona Approach,” *Proc. WISS ’98*, Dec. 1998.
[9] D. Roberts and R. Johnson, “Patterns for Evolving Frameworks,” *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.
[10] J. C. Hu and D. C. Schmidt, “Developing Flexible and High-Performance Web Servers with Frameworks and Patterns,” *ACM Comp. Surveys*, May 1998.

Biographies

Junichi Suzuki [M] (suzuki@yy.cs.keio.ac.jp) received B.S. and M.S. degrees in computer science from Keio University, Japan. He is currently completing his Ph.D. degree at the Department of Computer Science at Keio University. His research interests include object-oriented patterns and frameworks, reflection, object-oriented development methodology, distributed object computing, intelligent user interface, agent communication, and computational biology. He is a member of ACM, IPSJ, and JSSST.

Yoshikazu Yamamoto [M] (yama@cs.keio.ac.jp) received B.S., M.S., and Ph.D. degrees in administration engineering from Keio University, Tokyo. He is currently an associate professor in the Department of Computer and Information Science at Keio University. He worked at Linkoping University, Sweden, as a visiting professor from 1981 to 1983. His current research interests include distributed discrete event simulation and modeling, OOP, agent programming, intelligent interface, and documentation. He is a member of ACM and IPSJ, and also the board of directors of JSSST.