Using Design Patterns and Frameworks to Develop
Object-Oriented Communication Systems

Douglas C. Schmidt
www.cs.wustl.edu/~schmidt/
schmidt@cs.wustl.edu
Washington University, St. Louis

Motivation

- Developing efficient, robust, extensible, portable, and reusable communication software is hard
- It is essential to understand successful techniques that have proven effective to solve common development challenges
- Design patterns and frameworks help to capture, articulate, and instantiate these successful techniques

Observations

- Developers of communication software confront recurring challenges that are largely application-independent
  - e.g., service initialization and distribution, error handling, flow control, event demultiplexing, concurrency control
- Successful developers resolve these challenges by applying appropriate design patterns
- However, these patterns have traditionally been either:
  1. Locked inside heads of expert developers
  2. Buried in source code

Design Patterns

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - i.e., “Patterns = problem/solution pairs in a context”
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs
• **Intent:** provide a surrogate for another object that controls access to it

---

### More Observations

- Reuse of patterns alone is not sufficient
  
  - Patterns enable reuse of architecture and design knowledge, but not code (directly)

- To be productive, developers must also reuse detailed designs, algorithms, interfaces, implementations, etc.

- Application **frameworks** are an effective way to achieve broad reuse of software

---

### Frameworks

- A framework is:
  
  - "An integrated collection of components that collaborate to produce a reusable architecture for a family of related applications"

- Frameworks differ from conventional class libraries:
  
  1. Frameworks are ‘semi-complete’ applications
  2. Frameworks address particular application domains
  3. Frameworks provide “inversion of control”

- Typically, applications are developed by *inheriting* from and *instantiating* framework components
Differences Between Class Libraries and Frameworks

(A) CLASS LIBRARY ARCHITECTURE

(B) APPLICATION FRAMEWORK ARCHITECTURE

Tutorial Outline

- Outline key challenges for developing communication software
- Present the key reusable design patterns and framework components in high-performance Web clients and servers
  - Both single-threaded and various multi-threaded solutions are presented
  - The patterns and frameworks covered generalize to other communication software systems
    - e.g., ORBs, video-on-demand, medical imaging
- Discuss lessons learned from using patterns and frameworks on production software systems
  - e.g., telecom, avionics, medical systems

Stand-alone vs. Distributed Application Architectures

(1) STAND-ALONE APPLICATION ARCHITECTURE

(2) DISTRIBUTED APPLICATION ARCHITECTURE

Concurrency vs. Parallelism

CONCURRENT SERVER

PARALLEL SERVER
Sources of Complexity

- Distributed application development exhibits both inherent and accidental complexity

Inherent complexity results from fundamental challenges, e.g.,
- Distributed systems
  * Latency
  * Error handling
  * Service partitioning and load balancing
- Concurrent systems
  * Race conditions
  * Deadlock avoidance
  * Fair scheduling
  * Performance optimization and tuning

Sources of Complexity (cont’d)

- Accidental complexity results from limitations with tools and techniques, e.g.,
  - Low-level tools
    * e.g., Lack of type-secure, portable, re-entrant, and extensible system call interfaces and component libraries
  - Inadequate debugging support
  - Widespread use of algorithmic decomposition
    * Fine for explaining network programming concepts and algorithms but inadequate for developing large-scale distributed applications
  - Continuous rediscovery and reinvention of core concepts and components

OO Contributions

- Communication software has traditionally been performed using low-level OS mechanisms, e.g.,
  - fork/exec
  - Shared memory
  - Signals
  - Sockets and select
    - POSIX pthreads, Solaris threads, Win32 threads

- OO design patterns and frameworks elevate focus to application concerns, e.g.,
  - Service functionality and policies
  - Service configuration
  - Concurrent event demultiplexing and event handler dispatching
  - Service concurrency and synchronization

Concurrent Web Client/Server Example

- The following example illustrates a concurrent OO architecture for a high-performance Web client/server

- Key system requirements are:
  1. Robust implementation of HTTP protocol
     - i.e., resilient to incorrect or malicious Web clients/servers
  2. Extensible for use with other protocols
     - e.g., DICOM, HTTP 1.1, SFP
  3. Leverage multi-processor hardware and OS software
     - e.g., Support various concurrency models
General Web Client/Server Interactions

Web Server Software Architecture

- **Event Dispatcher**
  - Encapsulates Web server concurrency and dispatching strategies

- **HTTP Handlers**
  - Parses HTTP headers and processes requests

- **HTTP Acceptor**
  - Accepts connections and creates HTTP Handlers

Design Patterns in the Web Server Implementation

- **Proxy**
  - "Provide a surrogate or placeholder for another object to control access to it"

- **Strategy**
  - "Define a family of algorithms, encapsulate each one, and make them interchangeable"

- **Adapter**
  - "Convert the interface of a class into another interface client expects"

- **Singleton**
  - "Ensure a class only has one instance and provide a global point of access to it"

- **State**
  - "Allow an object to alter its behavior when its internal state changes"
Event Handling Patterns

- **Reactor**
  - “Decouples synchronous event demultiplexing and event handler initiation dispatching from service(s) performed in response to events”

- **Proactor**
  - “Decouples asynchronous event demultiplexing and event handler completion dispatching from service(s) performed in response to events”

- **Asynchronous Completion Token**
  - “Efficiently associates state with the completion of asynchronous operations”

Concurrency Patterns

- **Active Object**
  - “Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads”

- **Half-Sync/Half-Async**
  - “Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency”

- **Double-Checked Locking Optimization Pattern**
  - “Ensures atomic initialization of objects and eliminates unnecessary locking overhead on each access”

Concurrency Architecture Patterns

- **Thread-per-Request**
  - “Allows each client request to run concurrently in a separate thread”

- **Thread Pool**
  - “Allows up to N requests to execute concurrently within a pool of threads”

- **Thread-per-Connection**
  - “Allows each client connection to run concurrently”
    - S suited for HTTP 1.1, but not HTTP 1.0

Service Initialization Patterns

- **Connector**
  - “Decouples active connection establishment from the service performed once the connection is established”

- **Acceptor**
  - “Decouples passive connection establishment from the service performed once the connection is established”

- **Service Configurator**
  - “Decouples the behavior of network services from point in time at which services are configured into an application”
Selecting the Server’s Concurrency Architecture

- **Problem**
  - A very strategic design decision for high-performance Web servers is selecting an efficient concurrency architecture.

- **Forces**
  - No single concurrency architecture is optimal.
  - Key factors include OS/hardware platform and workload.

- **Solution**
  - Understand key alternative concurrency patterns.

Alternative Web Server Concurrency Patterns

- The following example illustrates the design patterns (and framework components) in an OO implementation of a concurrent Web server.

- The following are the key concurrency pattern alternatives:
  1. Reactive
  2. Thread-per-request
  3. Thread-per-connection
  4. Synchronous Thread Pool
  5. Asynchronous Thread Pool

### Alternative Web Server Concurrency Patterns

**Reactive Web Server**

1. Connect
2. Handle Input
3. Create Handler
4. Accept Connection
5. Activate Handler
6. Process HTTP Request

**Thread-per-Request Web Server**

1. Connect
2. Handle Input
3. Create Handler
4. Accept Connection
5. Spawn Thread
6. Process HTTP Request
The ADAPTIVE Communication Environment (ACE)

A set of C++ wrappers and frameworks based on common communication software design patterns.

Demultiplexing and Dispatching Events

- **Problem**
  - Web servers must process several different types of events simultaneously

- **Forces**
  - Multi-threading is not always available
  - Multi-threading is not always efficient
  - Tightly coupling general event processing with server-specific logic is inflexible

- **Solution**
  - Use the Reactor pattern to decouple generic event processing from server-specific processing

The Reactor Pattern

- **Intent**
  - "Decouples synchronous event demultiplexing and event handler initiation dispatching from service(s) performed in response to events"

- **This pattern resolves the following forces for synchronous event-driven software:**
  - How to demultiplex multiple types of events from multiple sources of events synchronously and efficiently within a single thread of control
  - How to extend application behavior without requiring changes to the event dispatching framework
Structure of the Reactor Pattern

- Participants in the Reactor pattern

Collaboration in the Reactor Pattern

A Single-threaded Reactive Web Server

An Integrated Reactive/Active Web Server
The HTTP_Handler Public Interface

- The HTTP_Handler is the Proxy for communicating with clients
  - Along with Reactor, this class implements the asynchronous task part of Half-Sync/Half-Async

```cpp
// Reusable base class.
template <class PEER_ACCEPTOR>
class HTTP_Handler :
  public Svc_Handler<PEER_ACCEPTOR::PEER_STREAM,
                   NULL_SYNCH> {
public:
  // Entry point into HTTP_Handler, called by
  // HTTP_Acceptor.
  virtual int open (void *) {
    // Register with Reactor to handle client input.
    Reactor::instance ()->register_handler (this, READ_MASK);
    // Register timeout in case client doesn't
    // send any HTTP requests.
    Reactor::instance ()->schedule_timer
      (this, 0, ACE_Time_Value (HTTP_CLIENT_TIMEOUT));
  }
};
```

The HTTP_Handler Protected Interface

- The following methods are invoked by callbacks from the Reactor

```cpp
protected:
  // Reactor notifies when client's timeout.
  virtual int handle_timeout (const Time_Value &,
   const void *) {
    // Remove from the Reactor.
    Reactor::instance ()->remove_handler
      (this, READ_MASK);
  }
  // Reactor notifies when HTTP requests arrive.
  virtual int handle_input (HANDLE);
    // Receive/frame client HTTP requests (e.g., GET).
    int recv_request (Message_Block &*);
};
```

Integrating Multi-threading

- **Problem**
  - Multi-threaded Web servers are needed since Reactive Web servers are often inefficient, non-scalable, and non-robust

- **Forces**
  - Multi-threading can be very hard to program
  - No single multi-threading model is always optimal

- **Solution**
  - Use the Active Object pattern to allow multiple concurrent server operations using an OO programming style

The Active Object Pattern

- **Intent**
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"

- **This pattern resolves the following forces for concurrent communication software:**
  - **How to allow blocking read and write operations on one endpoint that do not detract from the quality of service of other endpoints**
  - **How to simplify concurrent access to shared state**
  - **How to simplify composition of independent services**
Structure of the Active Object Pattern

- The Scheduler determines the sequence that Method Objects are executed.

Collaboration in the Active Object Pattern

Using the Active Object Pattern in the Web Server

The HTTP_Processor Class

- Processes HTTP requests using the "Thread Pool" concurrency model.

```cpp
class HTTP_Processor : public Task {
public:
    // Singleton access point.
    static HTTP_Processor *instance (void);

    // Pass a request to the thread pool.
    virtual int put (Message_Block *);

    // Event loop for the pool thread.
    virtual int svc (int) {
        Message_Block *mb = 0; // Message buffer.
        // Wait for messages to arrive.
        for (;;) {
            getq (mb); // Inherited from class Task;
            // Identify and perform WWW Server
            // request processing here...
        }
    }

    protected:
    HTTP_Processor (void); // Constructor.
};
```
Using the Singleton Pattern

- The HTTP.Processor is implemented as a Singleton that is created “on demand”

```cpp
HTTP_Processor * HTTP_Processor::instance (void) {
    // Beware of race conditions!
    if (instance_ == 0)
        instance_ = new HTTP_Processor;
    return instance_;}
```

- Constructor creates the thread pool

```cpp
HTTP_Processor::HTTP_Processor (void) {
    // Inherited from class Task.
    activate (THR_NEW_LWP, num_threads);
}
```

Subtle Concurrency Woes with the Singleton Pattern

- **Problem**
  - The canonical Singleton implementation has subtle “bugs” in multi-threaded applications

- **Forces**
  - Too much locking makes Singleton too slow...
  - Too little locking makes Singleton unsafe...

- **Solution**
  - Use the Double-Checked Locking optimization pattern to minimize locking and ensure atomic initialization

The Double-Checked Locking Optimization Pattern

- **Intent**
  - “Ensures atomic initialization of objects and eliminates unnecessary locking overhead on each access”

- This pattern resolves the following forces:

  1. *Ensures atomic initialization or access to objects, regardless of thread scheduling order*
  2. *Keeps locking overhead to a minimum*
      - e.g., only lock on creation

- Note, this pattern assumes atomic memory access...

Using the Double-Checked Locking Optimization Pattern for the Web Server

```cpp
if (instance_ == NULL) {
    mutex_acquire ();
    if (instance_ == NULL)
        instance_ = new HTTP_Processor;
    mutex_release ();
}
return instance_;
Integrating Reactive and Multi-threaded Layers

- **Problem**
  - Justifying the hybrid design of our Web server can be tricky

- **Forces**
  - Engineers are never satisfied with the status quo ;-)  
  - Substantial amount of time is spent re-discovering the intent of complex concurrent software design

- **Solution**
  - Use the Half-Sync/Half-Async pattern to explain and justify our Web server concurrency architecture

Half-Sync/Half-Async Pattern

- **Intent**
  - “Decouples synchronous I/O from asynchronous I/O in a system to simplify programming effort without degrading execution efficiency”

- This pattern resolves the following forces for concurrent communication systems:
  - How to simplify programming for higher-level communication tasks
    - These are performed synchronously
  - How to ensure efficient lower-level I/O communication tasks
    - These are performed asynchronously

Structure of the Half-Sync/Half-Async Pattern

**Collaborations in the Half-Sync/Half-Async Pattern**

- This illustrates input processing (output processing is similar)
Using the Half-Sync/Half-Async Pattern in the Web Server

Joining Async and Sync Tasks in the Web Server

- The following methods form the boundary between the Async and Sync layers

```c
int HTTP_Handler::handle_input (void)
{
    Message_Block *mb = 0;
    // Try to receive and frame message.
    if (recv_request (mb) == HTTP_REQUEST_COMPLETE) {
        Reactor::instance ()->remove_handler
            (this, READ_MASK);
        Reactor::instance ()->cancel_timer (this);
        // Insert message into the Queue.
        HTTP_Processor<FA>::instance ()->put (mb);
    }
}

// Task entry point.
HTTP_Processor::put (Message_Block *msg) {
    // Insert the message on the Message_Queue
    // (inherited from class Task).
    putq (msg);
}
```

Optimizing Our Web Server for Asynchronous Operating Systems

- **Problem**
  - Synchronous multi-threaded solutions are not always the most efficient

- **Forces**
  - Purely asynchronous I/O is quite powerful on some OS platforms
    - *e.g.*, Windows nt 4.x
  - Good designs should be adaptable to new contexts

- **Solution**
  - Use the Proactor pattern to maximize performance on Asynchronous OS platforms

The Proactor Pattern

- **Intent**
  - "Decouples asynchronous event demultiplexing and event handler completion dispatching from service(s) performed in response to events"

- **This pattern resolves the following forces for asynchronous event-driven software:**
  - How to demultiplex multiple types of events from multiple sources of events asynchronously and efficiently within a minimal number of threads
  - How to extend application behavior without requiring changes to the event dispatching framework
Structure of the Proactor Pattern

- Participants in the Proactor pattern

Client Connects to a Proactive Web Server

Client Sends Request to a Proactive Web Server
Structuring Service Initialization

- **Problem**
  - The communication protocol used between clients and the Web server is often orthogonal to the initialization protocol

- **Forces**
  - Low-level connection establishment APIs are tedious, error-prone, and non-portable
  - Separating initialization from use can increase software reuse substantially

- **Solution**
  - Use the Acceptor pattern to decouple passive service initialization from run-time protocol

The Acceptor Pattern

- **Intent**
  - "Decouples passive initialization of a service from the tasks performed once the service is initialized"

- **This pattern resolves the following forces for network servers using interfaces like sockets or TLI:**
  1. How to reuse passive connection establishment code for each new service
  2. How to make the connection establishment code portable across platforms that may contain sockets but not TLI, or vice versa
  3. How to ensure that a passive-mode descriptor is not accidentally used to read or write data
  4. How to enable flexible policies for creation, connection establishment, and concurrency

Structure of the Acceptor Pattern

- **Acceptor** is a factory that creates, connects, and activates a Svc_Handler

Collaboration in the Acceptor Pattern
The Acceptor Class

- The Acceptor class implements the Acceptor pattern

```cpp
// Reusable Factor
template <class SVC_HANDLER>
class Acceptor :
    public Service_Object // Subclass of Event_Handler.
{
    public:
        // Notified by Reactor when clients connect.
        virtual int handle_input (void)
        {
            // The strategy for initializing a SVC_HANDLER.
            SVC_HANDLER *sh = new SVC_HANDLER;
            peer_acceptor_.accept (sh->peer ());
            sh->open ();
        }
        // ...

    protected:
        // IPC connection factory.
        SOCK_Acceptor peer_acceptor_;  // ...
}
```

The HTTP_Acceptor Class Interface

- The HTTP_Acceptor class accepts connections and initializes HTTP Handlers

```cpp
class HTTP_Acceptor
    : public Acceptor<HTTP_Handler>
{
    public:
        // Hook called automatically when HTTP_Acceptor
        // is dynamically linked.
        virtual int init (int argc, char *argv[]);

        // Hook called automatically when HTTP_Acceptor is
        // dynamically unlinked.
        virtual int fini (void);

        // ...
}
```

Putting the Pieces Together at Run-time

- Problem
  - Prematurely committing ourselves to a particular Web server configuration is inflexible and inefficient

- Forces
  - Certain server configuration decisions can’t be made efficiently until run-time
  - Forcing users to pay for components they don’t use is undesirable

- Solution
  - Use the Service Configurator pattern to assemble the desired Web server components dynamically
**The Service Configurator Pattern**

- **Intent**
  - "Decouples the behavior of communication services from the point in time at which these services are configured into an application or system."

- This pattern resolves the following forces for highly flexible communication software:
  - How to defer the selection of a particular type, or a particular implementation, of a service until very late in the design cycle
    - i.e., at installation-time or run-time
  - How to build complete applications by composing multiple independently developed services
  - How to optimize, reconfigure, and control the behavior of the service at run-time

---

**Structure of the Service Configurator Pattern**

---

**Collaboration in the Service Configurator Pattern**

---

**Using the Service Configurator Pattern in the Web Server**

- Existing Web server is based on Half-Sync/Half-Async pattern

- Other versions could be single-threaded, could use other concurrency strategies, and other protocols
The HTTP_Acceptor Class Implementation

```c
int HTTP_Acceptor::init (int argc, char **argv[]) {
    Options::instance ()->parse_args (argc, argv);
    // Set the endpoint into listener mode.
    Acceptor::open (local_addr);
    // Initialize the communication endpoint.
    Reactor::instance ()->register_handler (this, ACCEPT_MASK);
}
```

```c
int HTTP_Acceptor::fini (void) {
    // Unblock threads in the pool so they will
    // shutdown correctly.
    HTTP_Processor::instance ()->close ();
    // Wait for all threads to exit.
    Thread_Manager::instance ()->wait ();
}
```

Main Program for Web Server

- Dynamically configure and execute the Web Server
  - Note that this is totally generic!

```c
int main (int argc, char *argv[]) {
    Service_Config daemon;
    // Initialize the daemon and dynamically
    // configure the service.
    daemon.open (argc, argv);
    // Loop forever, running services and handling
    // reconfigurations.
    daemon.run_event_loop ();
    /* NOTREACHED */
}
```

The OO Architecture of the JAWS Framework

- WWW.cs.wustl.edu/~jxh/research/

Configuring the Web Server with the Service Configurator

- The concurrent Web Server is configured and initialized via a configuration script

```bash
% cat /svc.conf
dynamic TP_WWW_Server Service_Object *
www_server.dll:make_TP_WWW_Server ()
"-p $PORT -t $THREADS"
```

- Factory function that dynamically allocates a Half-Sync/Half-Async Thread Pool Web Server

```c
extern "C" Service_Object *make_TP_WWW_Server (void); Service_Object *make_TP_WWW_Server (void) {
    return new HTTP_Acceptor;
    // ACE dynamically unlinks and deallocates this object.
}
```
Web Server Optimization Techniques

- Use lightweight concurrency
- Minimize locking
- Apply file caching and memory mapping
- Use “gather-write” mechanisms
- Minimize logging
- Pre-compute HTTP responses
- Avoid excessive time calls
- Optimize the transport interface

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures
- Patterns explicitly capture expert knowledge and design tradeoffs
- Patterns help improve developer communication
- Patterns help ease the transition to object-oriented technology

Applying Patterns to CORBA ORBs

Drawbacks to Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a software development process is a human-intensive activity
Suggestions for Using Patterns Effectively

- Do not recast everything as a pattern
  - Instead, develop strategic domain patterns and reuse existing tactical patterns

- Institutionalize rewards for developing patterns

- Directly involve pattern authors with application developers and domain experts

- Clearly document when patterns apply and do not apply

- Manage expectations carefully

Lessons Learned using OO Frameworks

- Benefits of frameworks
  - Enable direct reuse of code (cf patterns)
  - Facilitate larger amounts of reuse than stand-alone functions or individual classes

- Drawbacks of frameworks
  - High initial learning curve
    - Many classes, many levels of abstraction
  - The flow of control for reactive dispatching is non-intuitive
  - Verification and validation of generic components is hard

Patterns and Framework Literature

- Books
  - Gamma et al., “Design Patterns: Elements of Reusable OO Software” AW, 1994
  - Siemens, Pattern-Oriented Software Architecture, Wiley, 1996

- Special Issues in Journals
  - October ’96 “Communications of the ACM” (eds: Douglas C. Schmidt, Ralph Johnson, and Mohamed Fayad)
  - October ’97 “Communications of the ACM” (eds: Douglas C. Schmidt and Mohamed Fayad)

- Magazines
  - C++ Report and JOOP, columns by Coplien, Vlissides, Vinoski, Schmidt, and Martin

Conferences and Workshops on Patterns

- Pattern Language of Programs Conferences
  - September, 1998, Monticello, Illinois, USA
  - st-www.cs.uiuc.edu/users/patterns/patterns.html

- The European Pattern Languages of Programming conference
  - July, 1998, Kloster Irsee, Germany
  - www.cs.wustl.edu/~schmidt/patterns.html

- USENIX COOTS
  - April 27–30, 1998, Santa Fe, New Mexico
  - www.usenix.org/events/coots98/
Obtaining ACE and JAWS

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns
  - JAWS is both a Web server framework and a high-performance Web server

- All source code for ACE and JAWS is freely available
  - www.cs.wustl.edu/~schmidt/ACE.html

- Mailing lists
  * ace-users@cs.wustl.edu
  * ace-users-request@cs.wustl.edu
  * ace-announce@cs.wustl.edu
  * ace-announce-request@cs.wustl.edu

- Newsgroup
  - comp.soft-sys.ace