Pattern-Oriented Software Architecture

CORBA Design Patterns

Arvind S Krishna
Jaiganesh Balasubramanian
http://www.doc.ece.edu/

Electrical & Computing Engineering Department
The Henry Samueli School of Engineering
University of California, Irvine

Friday, November 08, 2002

Motivation for Patterns and Frameworks

- Developing software is hard
- Developing reusable software is even harder
- Proven solutions include Patterns and frameworks

Overview of Patterns and Frameworks

- Patterns support reuse of software architecture and design
  - Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain

- Frameworks support reuse of detailed design and code
  - A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications.

- Together, design patterns and frameworks help to improve software quality and reduce development time
  - e.g., reuse, extensibility, modularity, performance

Pattern Definition

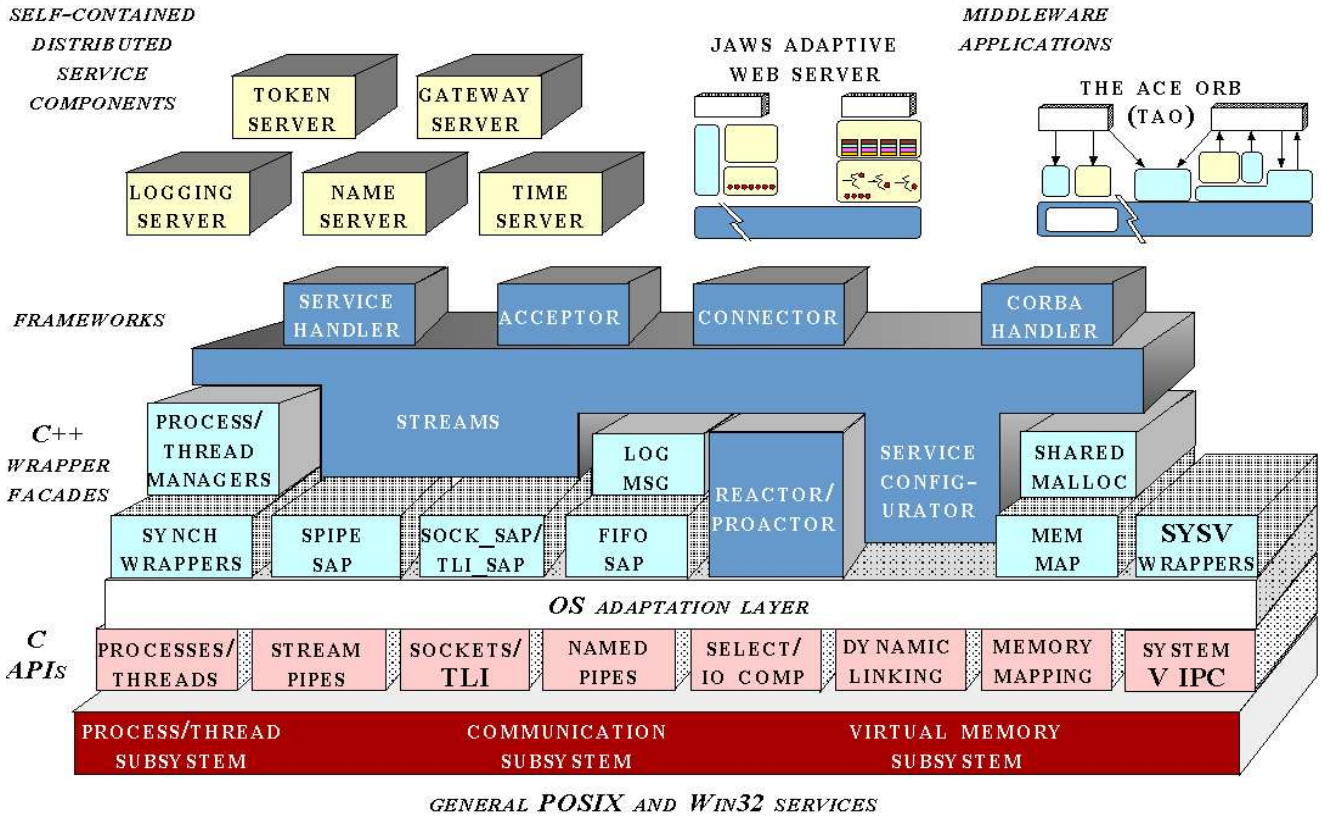
- Present solutions to common software problems arising within a certain context
- Help resolve key design forces
- Capture recurring structures and dynamics among software participants to facilitate reuse of successful designs.
- Generally codify expert knowledge of design constraints and "best practices"

ACE Introduction

- Motivation for ACE (Adaptive Communication Environment)
  - ACE was first started in 1991 at UCI where Dr Schmidt was a grad student in the ICS department.
  - It was necessary to revert to using C function APIs and procedural design when we needed to access system resources like Threads, Sockets that were available in common OS platforms
  - It was needed to keep rewriting the code to handle common network programming tasks such as connection establishment, event demultiplexing and distribution
- ACE was created to resolve these two frustrations

Overview of the ACE Framework

- Open-source
- 200,000+ lines of C++
- 30+ person-years of effort
- Ported to Win32, UNIX, & RTOSs
  - *e.g.,* VxWorks, pSoS, LynxOS, Chorus, QNX



- **Large open-source user community**
  - www.cs.wustl.edu/~schmidt/ACE-users.html

ACE benefits

•ACE contains a higher-level network programming framework that integrates and enhances the lower-level C++ wrapper facades. This framework supports the dynamic configuration of concurrent distributed services into applications.

•**Event demultiplexing components** -- The ACE Reactor and Proactor are extensible, object-oriented demultiplexers that dispatch application-specific handlers in response to various types of I/O-based, timer-based, signal-based, and synchronization-based events.

•**Service initialization components** -- The ACE Acceptor and Connector components decouple the active and passive initialization roles, respectively, from application-specific tasks that communication services perform once initialization is complete.

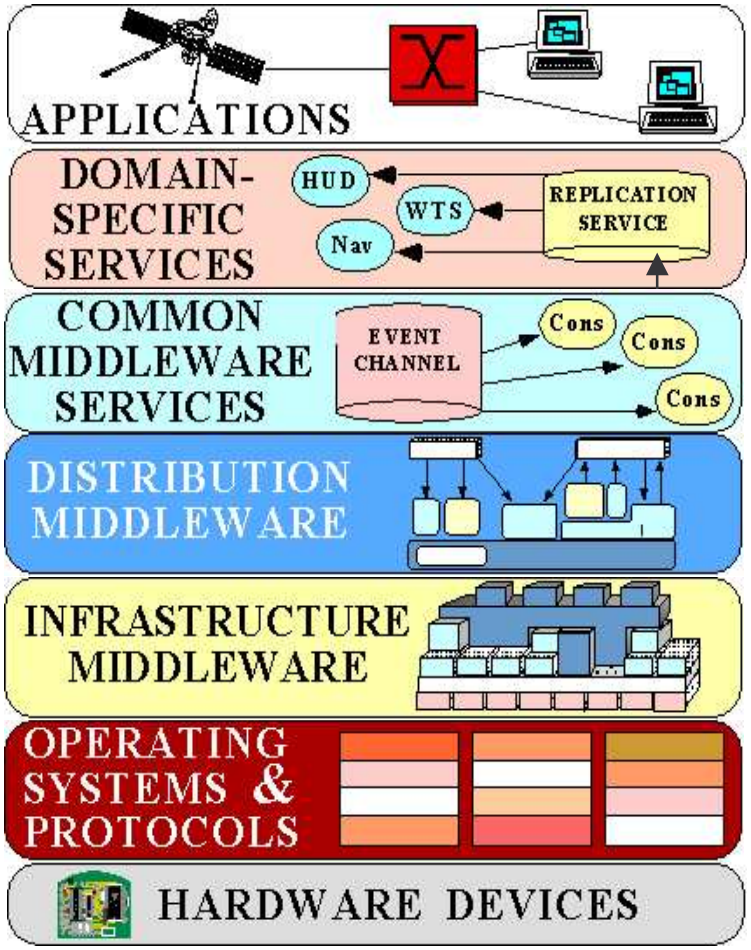
•**Service configuration components** -- The ACE Service Configurator supports the configuration of applications whose services may be assembled dynamically at installation-time and/or run-time.

•**Hierarchically-layered stream components** -- The ACE Streams components simplify the development of communication software applications, such as user-level protocol stacks, that are composed of hierarchically-layered services.

•**ORB adapter components** -- ACE can be integrated seamlessly with single-threaded and multi-threaded CORBA implementations via its ORB adapters.

TAO (The ACE ORB)

- Motivation for TAO
  - Provide a high quality, freely available, open source, standards compliant CORBA middleware platform that could be used effectively by both researchers and developers.
  - Help to mature middleware standards
  - Combine optimizations for OS level networking sub systems with optimizations for ORBs (Object Request Broker) to provide ORB end systems that can support end to end throughput latency jitter and reliability QoS guarantees.



Strategic and Tactical Patterns

- *Strategic Patterns* have an extensive impact on the software architecture.

  - Typically oriented to solutions in a particular do main

  e.g., Reactor and Acceptor pattern in the domain of event-driven, connection-oriented communication software

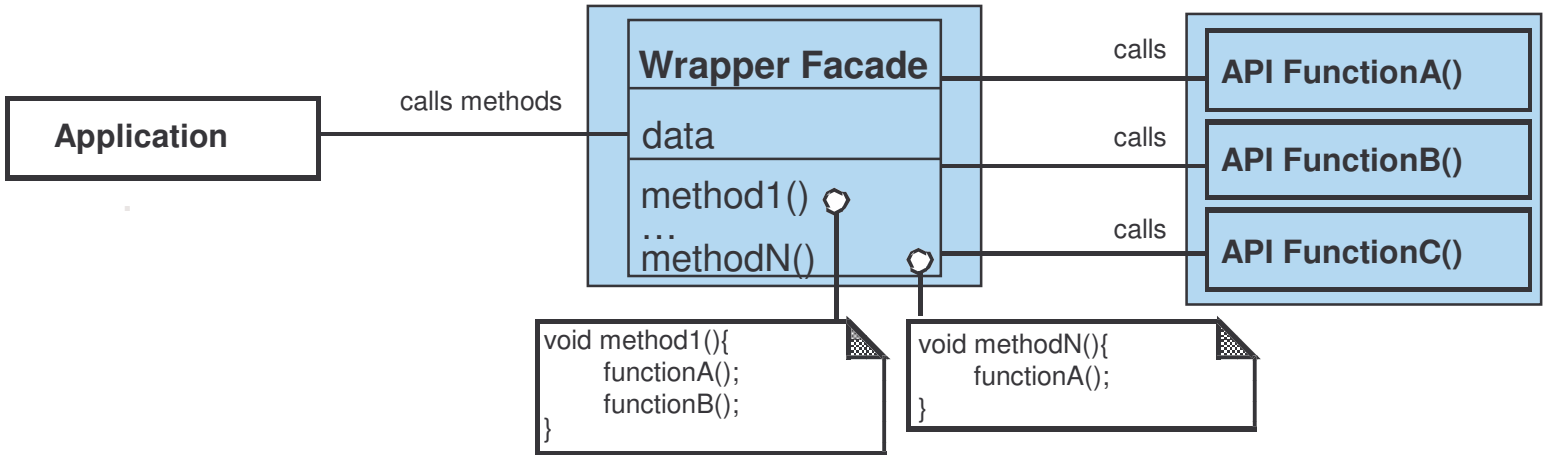
- *Tactical design patterns* have a relatively localized impact on a software architecture

  - Typically domain-independent

  e.g., Wrapper, Adapter, Bridge, Factory Method, and Strategy

Wrapper Façade Pattern

- Context
  - A Web server must manage a variety of OS services, including processes, threads, Socket connections, virtual memory, & files. Most operating systems provide low-level APIs written in C to access these services

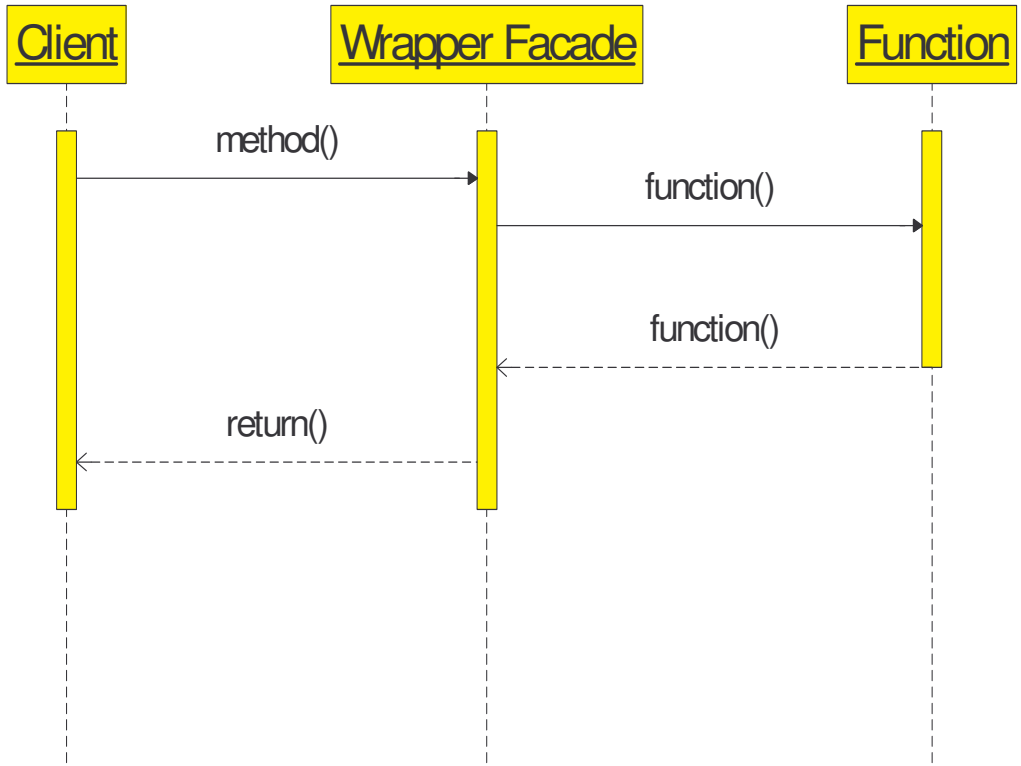


Wrapper Facade

- Problem
  - **Implementing advanced services requires utilizing lower-level functionality.**
  - **Many low-level APIs differ in syntax, even if they share semantics.**
  - **Nonetheless**,
    - **Applications should be portable to different platforms**
    - **Direct dependencies from APIs should be avoided.**
    - **Applications should use these APIs correctly and consistently.**

```
// Handle UNIX/Win32 portability.
#if defined (_WIN32)
static CRITICAL_SECTION lock;
#else
static mutex_t lock;
#endif /* _WIN32 */
...
#if defined (_WIN32)
SOCKET h;
DWORD t_id;
#else
int h;
thread_t t_id;
#endif /* _WIN32 */
...
#if defined (_WIN32)
ThreadCreate( ... );
#else
thr_create( ... );
#endif /* _WIN32 */
```

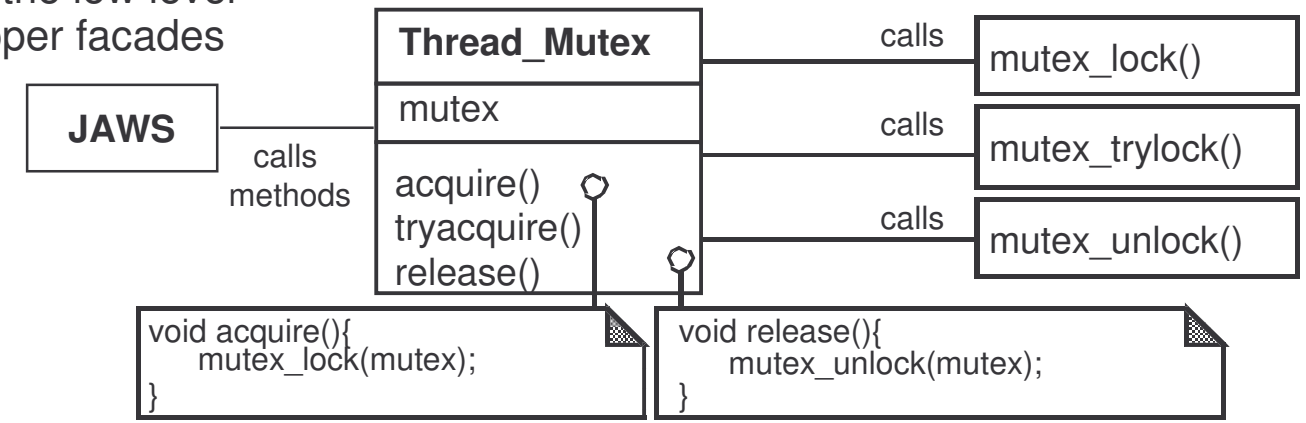
- Solution
  - Use the Wrapper Façade Pattern to avoid accessing low level operating system APIs directly.
- Definition
  - **The *Wrapper Facade* design pattern encapsulates the functions and data provided by existing non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces.**
- Solution Dynamics
  - A Client invokes a method on the wrapper façade.
  - The wrapper delegates the execution of this method to a function that represents a low level API



Wrapper Façade: Finale

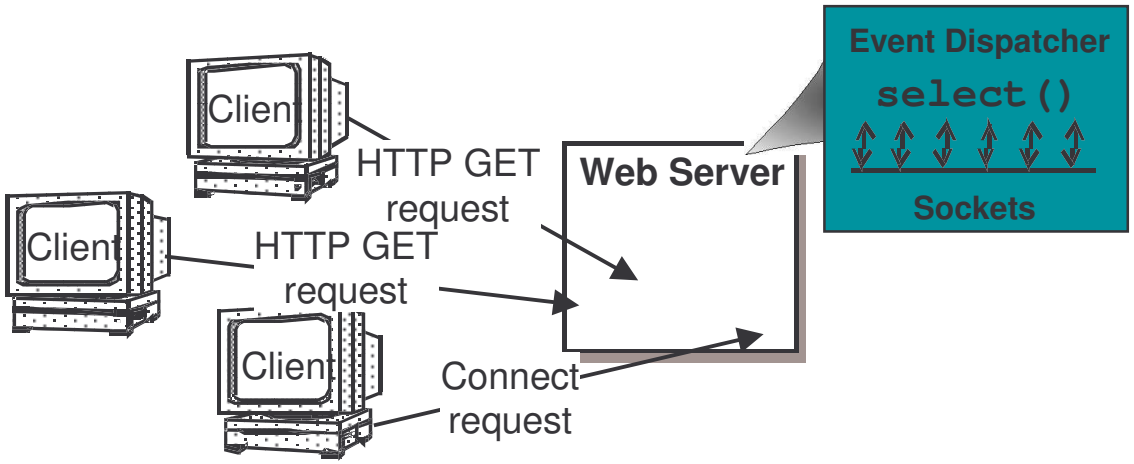
- Solution Structure
  - Encapsulate functions and low-level data structures within classes
  - Functions are existing low-level functions and data structures.
  - Wrapper Facades shield Clients from dependencies to the functions by providing methods that forward client invocations to the functions.
  - A Client accesses the low level functions via Wrapper facades
- Benefits
  - Robustness
  - Portability
  - Modularity
- Liability
  - Additional indirection

ACE defines several wrapper facades to ensure its framework components can run on many operating systems, including Windows, UNIX, & many real-time operating systems



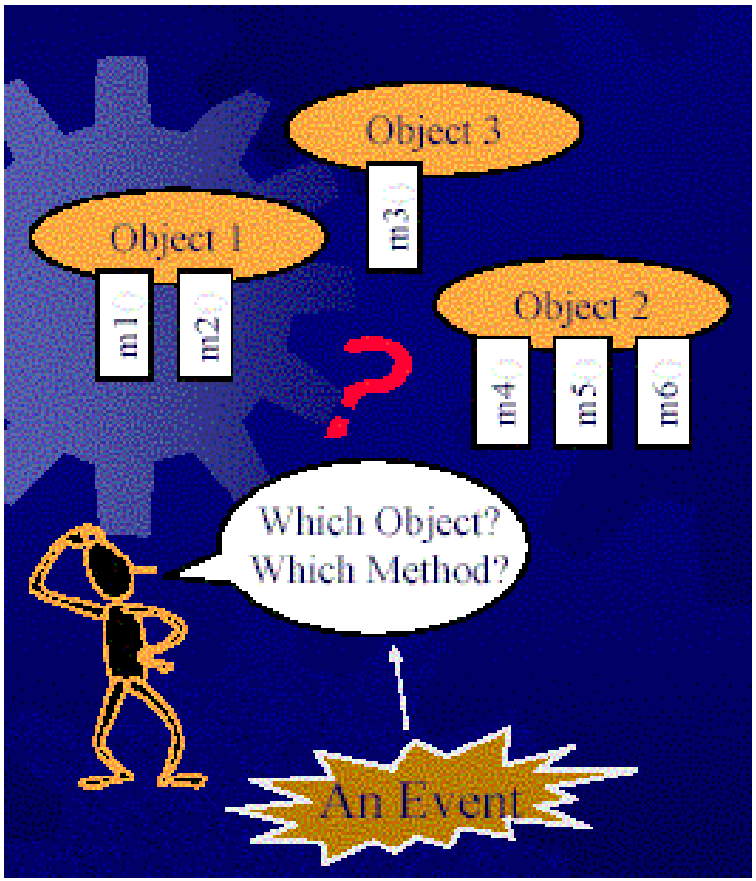
Reactor Architectural Pattern

- Context
  - Web servers can be accessed simultaneously by multiple clients
  - They must demux & process multiple types of indication events arriving from clients concurrently
  - A common way to demux events in a server is to use **`select()`**



Reactor Pattern

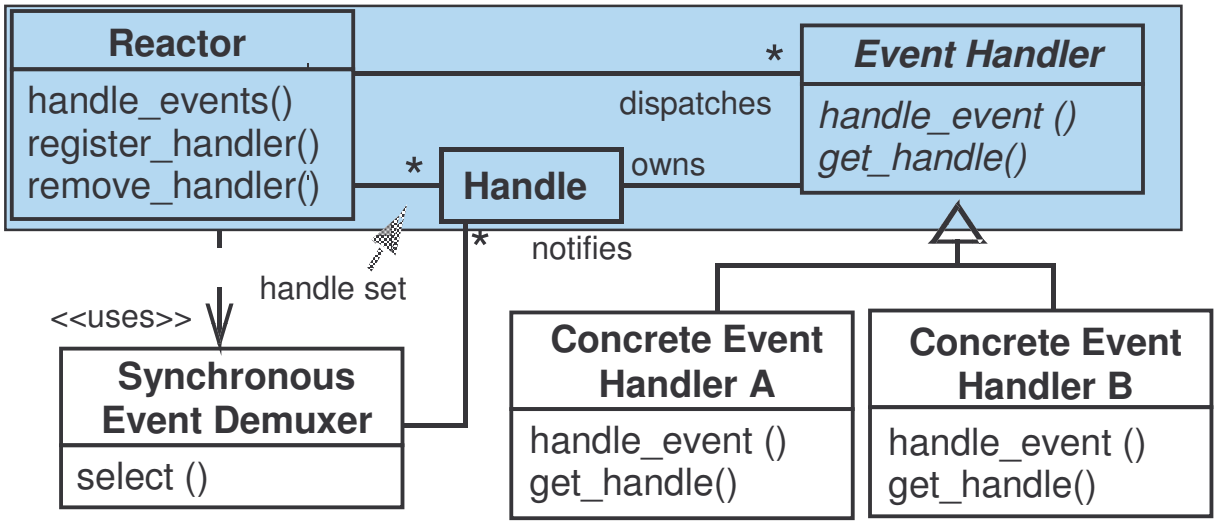
- Problem
  - In distributed systems, servers must handle client request effectively.
  - They must be able to handle request even if they are waiting for other requests to arrive.
  - Developers often couple event-demuxing & connection code with protocol-handling code.
  - This code cannot then be reused directly by other protocols or by other middleware & applications
  - Thus, changes to event-demuxing & connection code affects the server protocol code directly & may yield subtle bugs



Reactor Architectural Pattern

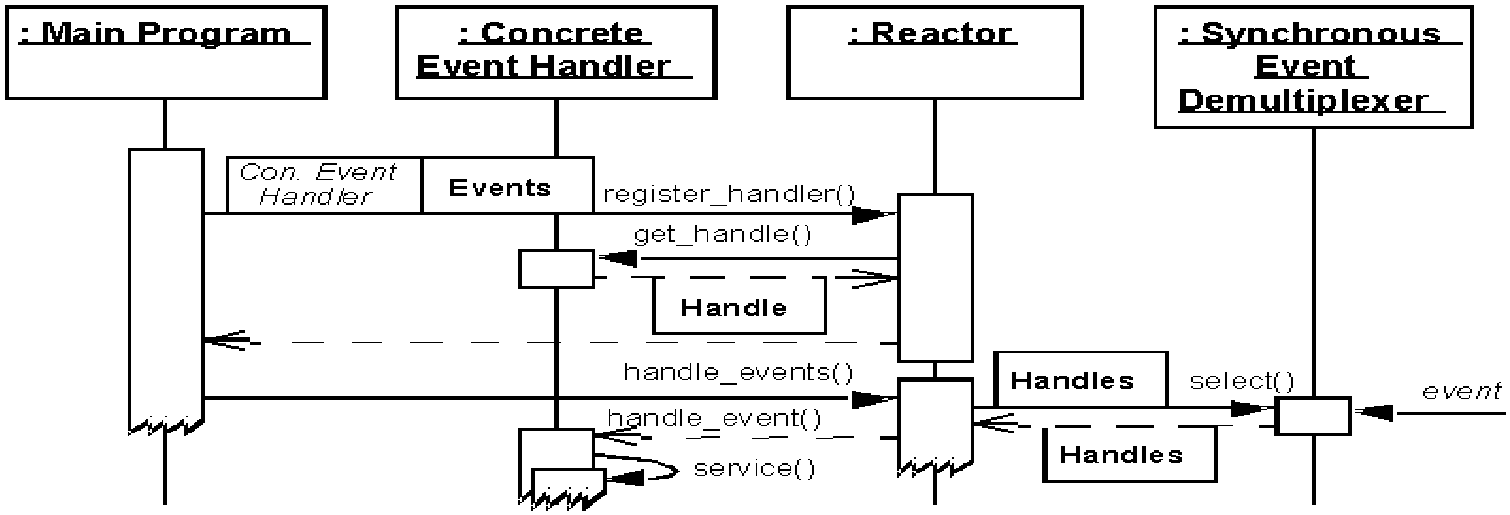
- Solution

  - The *Reactor* architectural pattern allows event-driven applications to demultiplex & dispatch service requests that are delivered to an application from one or more clients.



Reactor Architectural Pattern: Dynamics

- Solution Dynamics
  - **The reactor requests each event handler to pass back its handle**
  - **The event handling is started. The Reactor calls the synchronous event demultiplexer to wait for events to occur on the handles.**
  - **The synchronous even demultiplexer notifies the Reactor when the handle becomes ready. The reactor triggers the event handler represented by the handle to process the event**
  - **The event handler processes the event.**



Reactor Architectural Pattern: Finale

- Benefits
  - Separation of concerns
  - Configurability and Extensibility with new services
  - Portability

- Liabilities
  - Restricted applicability since it requires the underlying operating system to support handles.
  - Event handlers are not preempted while they are executing
  - Hard to debug.

The Half-Sync/Half-Async Pattern: Intro

Context

- HTTP runs over TCP, which uses flow control to ensure that senders do not produce data more rapidly than slow receivers or congested networks can buffer and process
- Since achieving efficient end-to-end *quality of service* (QoS) is important to handle heavy Web traffic loads, a Web server must scale up efficiently as its number of clients increases

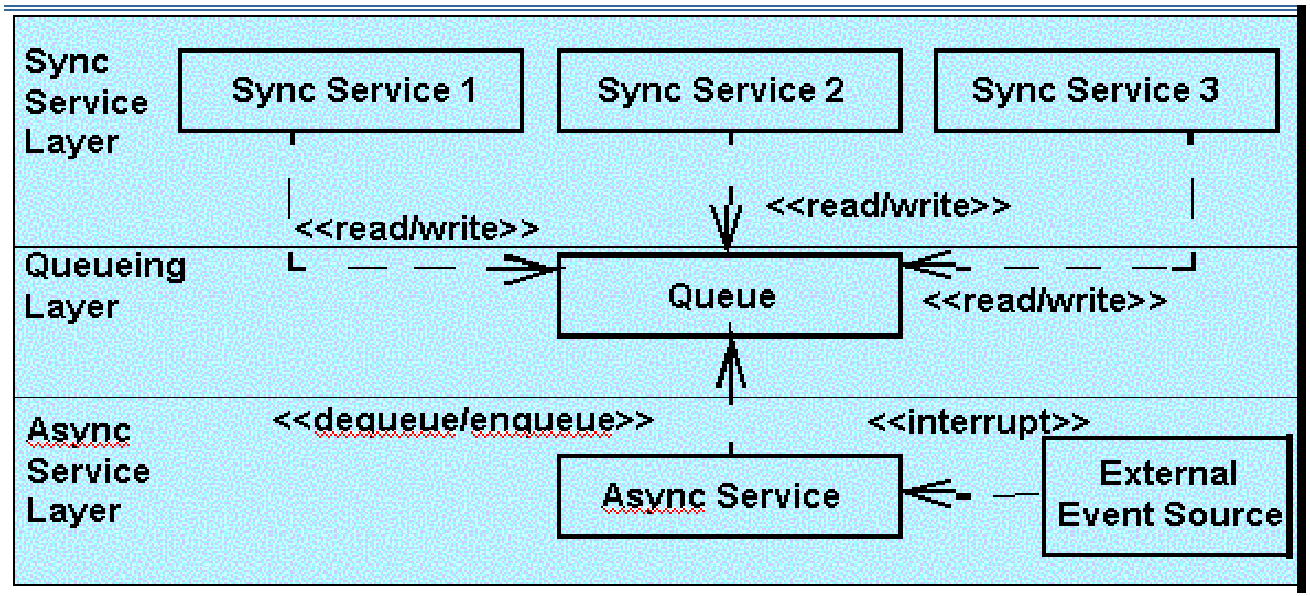
Problem

- Processing all HTTP GET requests reactively within a single-threaded process does not scale up, because each server CPU time-slice spends much of its time blocked waiting for I/O operations to complete
- Similarly, to improve QoS for all its connected clients, an entire Web server process must not block while waiting for connection flow control to abate so it can finish sending a file to a client

The Half-Sync/Half-Async Pattern

Solution

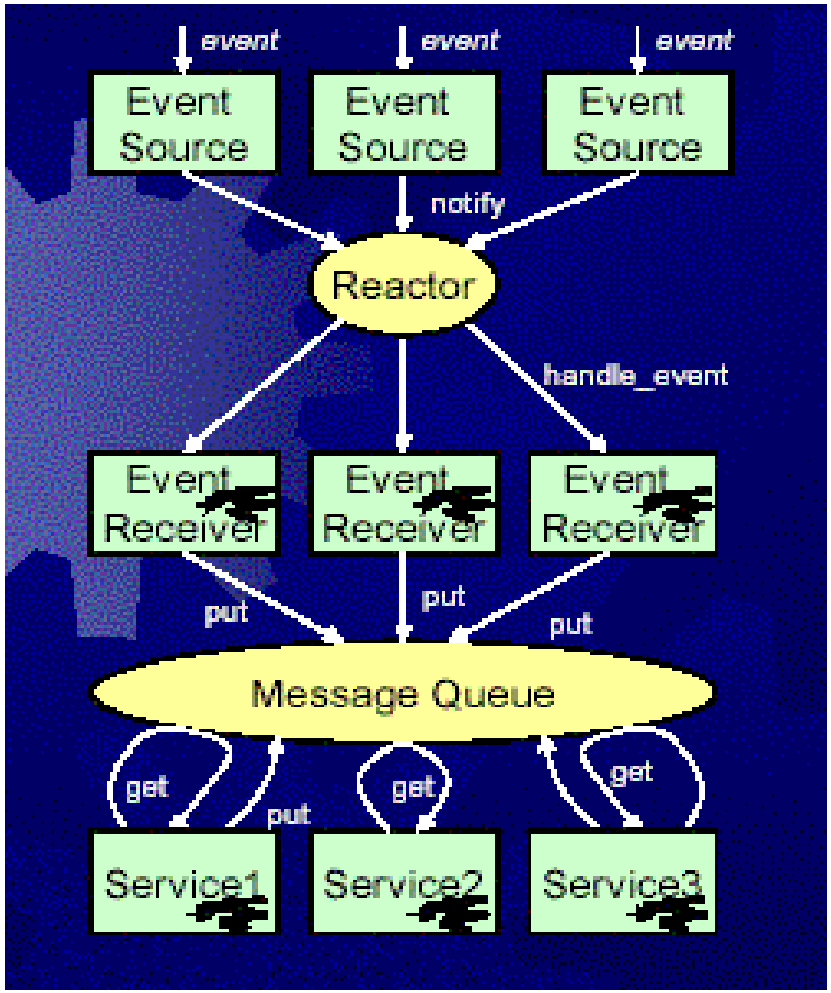
- Apply the *Half-Sync/Half-Async* architectural pattern to scale up server performance by processing different HTTP requests concurrently in multiple threads.
-The *Half-Sync/Half-Async* architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



The Half-Sync/Half-Async Pattern

Solution Dynamics

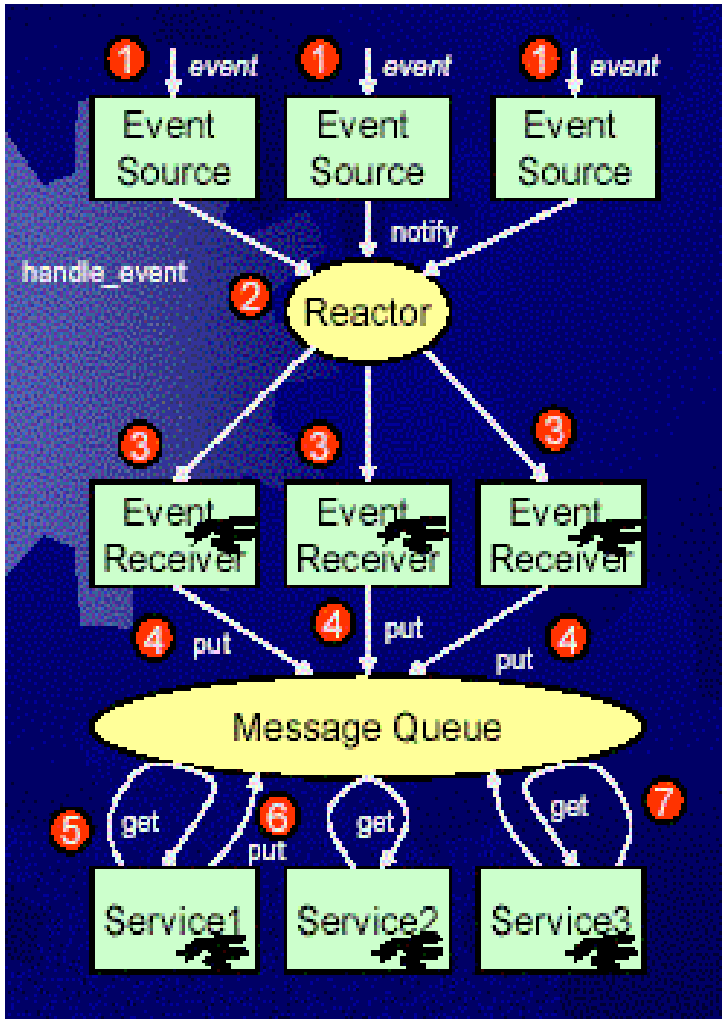
- Events arriving at Event sources asynchronously are read by asynchronous event receivers that insert them into the Message queue.
- A pool of synchronous application services retrieve these Events and process them, each in its own thread of control.



The Half-Sync/Half-Async Pattern

Solution Dynamics

-Events arriving at event sources asynchronously (1) are dispatched to appropriate event receivers by the Reactor (2).
-The Event Receivers read the events (3) and insert them into a Message Queue (4).
-Steps 1 to 4 happen independently of any service processing.
-Application services run in their own thread of control and block on the message queue until an event they can process arrives (5).
-The application services process the events and generate output that is either returned to the sender of the original event or to a designated receiver. (6 to 8).



The Half-Sync/Half-Async Pattern

Benefits

- Thread Management Overhead is minimized and resources are not wasted.
-Application services can run independently.
-Threads coordinate via Message Queues.
-Fast event reception (async) and simplified service implementation (sync).
-Implementation is easy.

Liabilities

-Processing an event by multiple services in cooperation may incur context switching overhead.

Observation

•*Failure rarely results from unknown scientific principles, but from failing to apply proven engineering practices & patterns*

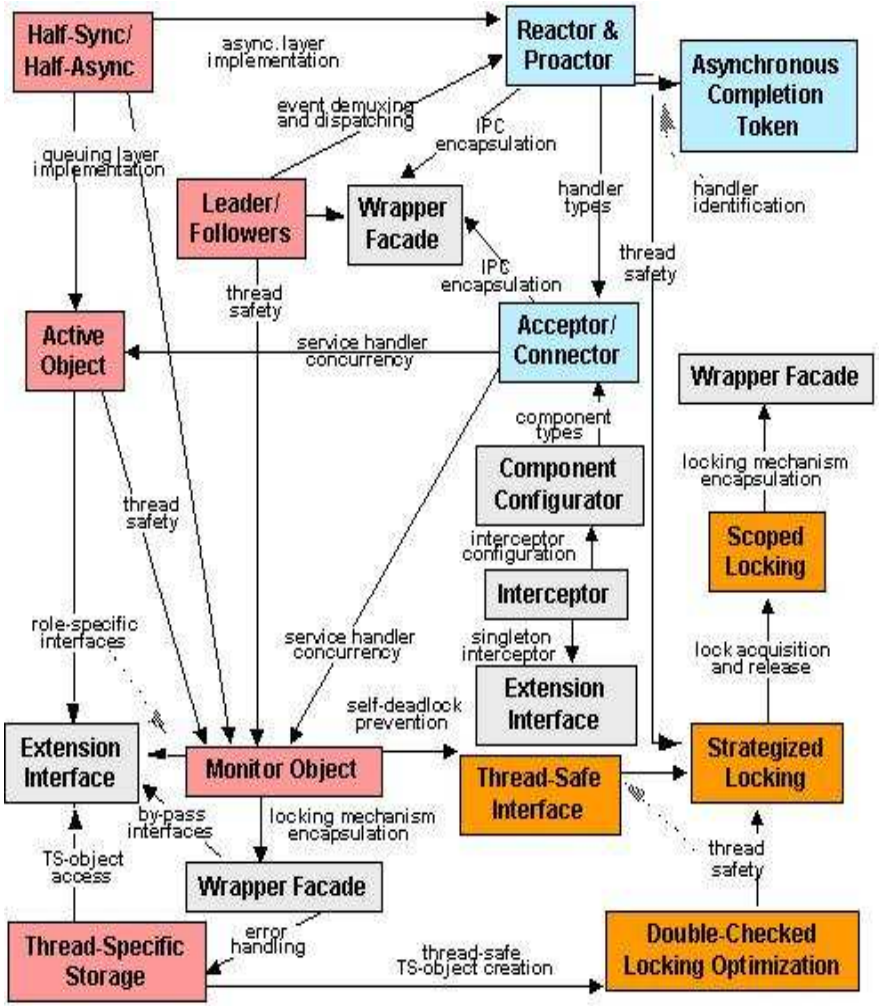
Benefits of POSA2 Patterns

•Preserve crucial design information used by applications & underlying frameworks/components

•Facilitate design reuse

•Guide design choices for application developers

URL for POSA Books
www.posa.uci.edu



Summary and Conclusions

- Mature engineering disciplines have handbooks that describe successful solutions to known problems
- e.g., automobile designers don't design cars using the laws of physics, they adapt adequate solutions from the handbook known to work well enough.
- The extra few percent of performance available by starting from scratch typically isn't worth the cost
- Patterns can form the basis for the handbook of software engineering.
- If software is to become an engineering discipline, successful practices must be systematically documented and widely disseminated.