

Pra 03

Constructing a Web-Servers with Patterns

Michael Stal
Siemens Corporate Technology



Michael Stal – Aufbau eines Web-Servers mit Hilfe von Patterns

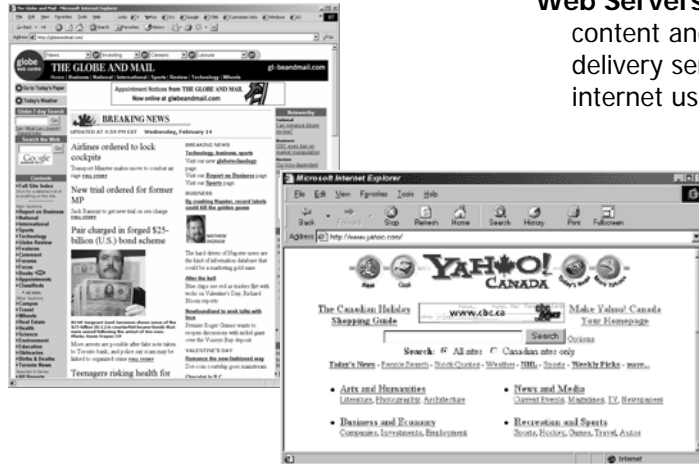
1



Case Study

A High-Performance
Web Server

Web Server: The Domain



Web Servers provide content and content delivery services to internet users.

Web Server: Key Challenges

Different end user needs and traffic workloads require configurable web server architectures regarding the following aspects:

- Concurrency models: thread per request, thread pool, ...
- Event demultiplexing models: synchronous or asynchronous
- File caching models: least-recently used (LRU), least-frequently used (LFU), ...
- Content delivery protocols: HTTP/1.0, HTTP/1.1, HTTP-NG
- Operating system: UNIX, Win32, ...



Event Demultiplexing (I)

A Web server must demultiplex and process multiple types of event, such as `CONNECT` and `HTTP GET` requests.

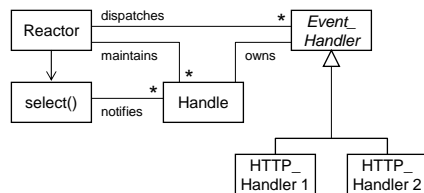
To be adaptable towards different usage scenarios, its concrete event demultiplexing and dispatching architecture should:

- Support changing the event demultiplexing and dispatching code independently of event processing code.
- Allow the integration of new or improved versions of the event processing services.
- Support a high throughput.
- Be simple to use.



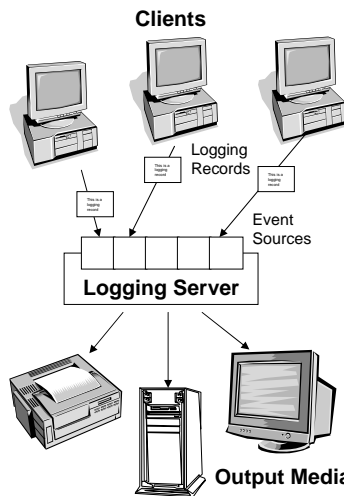
Event Demultiplexing (II)

Separate event demultiplexing and dispatching from event processing via Reactor:



- Event handlers (`HTTP_Handler`) represent the web server's services for processing events that can occur on an associated event source.
- Handles (`Handle`) represent event sources whose events must be processed.
- An event demultiplexer (`select()`) discovers and demultiplexes events that occur on handles.
- A reactor (`Reactor`) provides the web server's event loop and dispatches events that occurred on `Handles` to their associated `HTTP_Handler`.

Reactor: Problem

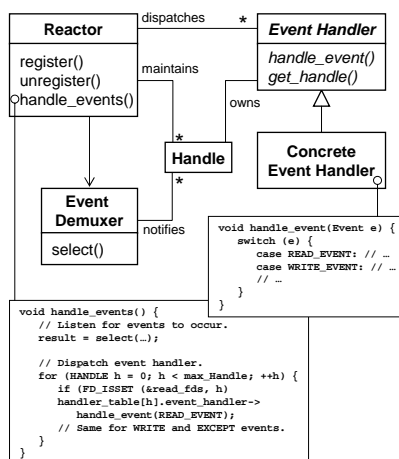


Reactor (POSA2)

Problem: Event-driven applications must be prepared to receive multiple service requests simultaneously, but to simplify programming, these requests should be processed serially. How can we provide such an infrastructure so that:

- an application does not block on any single event source and exclude other event sources from being serviced.
- application developers are shielded from low-level event detection, de-multiplexing, and dispatching mechanisms.
- an application is extensible with new event handling components.
- a high throughput can be achieved.

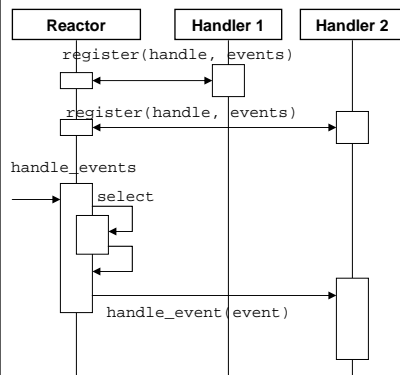
Reactor: Structure



Solution Structure: Separate event demultiplexing and dispatching from event processing.

- The OS provides mechanisms to represent event sources (*Handles*) and to detect and demultiplex events that occur on these event sources (*Event Demultiplexers* like `select()`).
- A *Reactor* encapsulates the OS event demultiplexing mechanisms and provides functionality to dispatch events to application services.
- *Event Handlers* provide functionality to process certain types of event that occur on a certain event source. They register with the reactor to get dispatched when the respective events occur on their event source.

Reactor: Dynamics



Solution Dynamics: Dispatch an event handler whenever an event occurs on its handle for which it has registered.

- The *Concrete Event Handlers* register their *Handle* and all events of interest with the *Reactor*.
- The application's main event loop calls the *Reactor* to listen for events to occur on all registered *Handles*.
- Once one or more events occurred on the *Handles*, the *Reactor* serially dispatches their *Event Handlers* to process the occurred events, if the *Handlers* had registered for them.

Reactor: Consequences



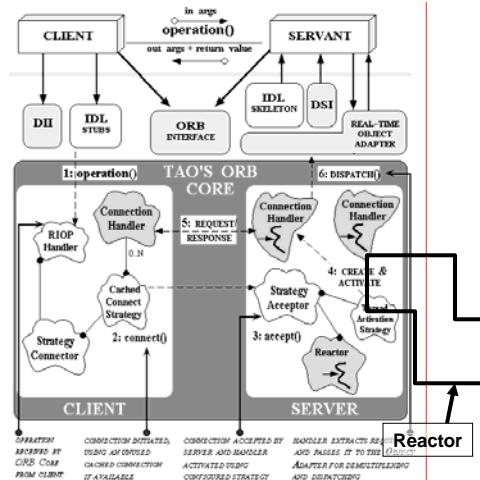
Benefits

- Separation of concerns: event de-multiplexing and dispatching infrastructure is separated from application-specific event processing.
- Reusability: the event de-multiplexing and dispatching infrastructure is a reusable component.
- Extensibility: new or improved event handlers can be added easily.
- Ease of use: a Reactor provides a simple, serial event handling architecture.

Liabilities

- Restricted applicability: requires appropriate OS support.
- Non-pre-emptive: in single-threaded applications, long running event handlers can prevent the processing of other events.

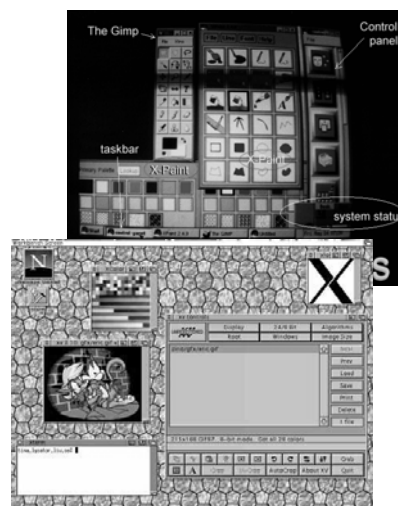
Reactor: Example



TAO: The high-performance, real-time CORBA ORB
TAO, The ACE ORB, uses a multi-threaded Reactor design as its central server-side ORB core event handling infrastructure.



Reactor: Known Uses



Object Request Brokers: The ACE ORB (TAO) provides Reactor-based event handling infrastructures.

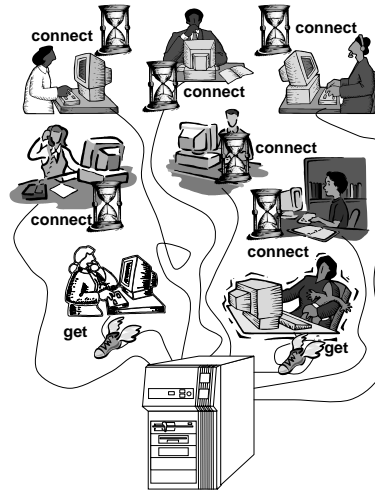
GUI infrastructures: MFC and Win Forms provide Reactor-based main event loops.

ACE: The open source ADAPTIVE Communication Environment provides several reusable Reactor frameworks for different types of event demultiplexers, such as `WaitForMultipleObjects()` and `select()`, and concurrency models, such as Leader/Followers.

Connection Establishment (I)

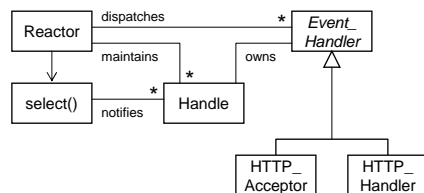
The two key activities performed by a web server are accepting client connection requests and processing subsequent HTTP requests.

- To provide a high throughput and quality of service to clients, processing a particular HTTP request must not prevent the web server from accepting new client connection requests.
- To support the configuration of the web server towards different environments and usage scenarios, it should be possible to modify connection establishment code independently of HTTP request processing code.



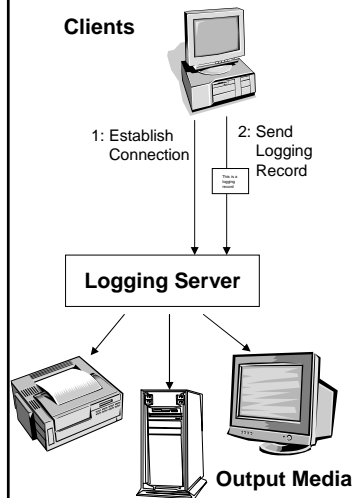
Connection Establishment (II)

Separate the establishment of connections from processing events via Acceptor-Connector:



- Acceptors (HTTP_Acceptor) listen for incoming client connection request and create/initialize service handlers to process subsequent HTTP requests.
- Service handlers (HTTP_Handler) are created by an acceptor to process a specific client's HTTP requests.

Acceptor-Connector: Problem

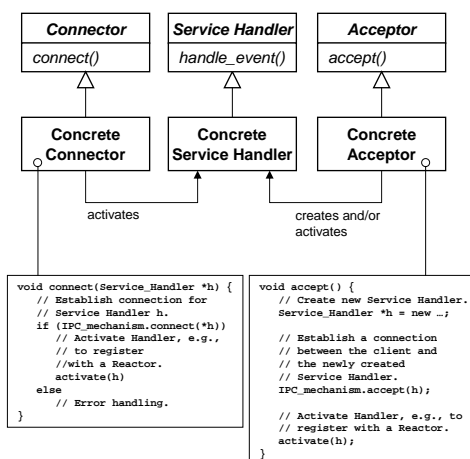


Acceptor-Connector (POSA2)

Problem: In networked systems, components must first connect to one another before they can interact. However:

- in many systems components can act both as clients that initiate a connection to a remote component or as servers that accept connection requests from remote clients.
- the code for connection establishment is largely independent of service processing code and can also alter independently.
- if connection establishment code is interwoven with service processing code, components cannot handle new connection requests while they are processing service requests.

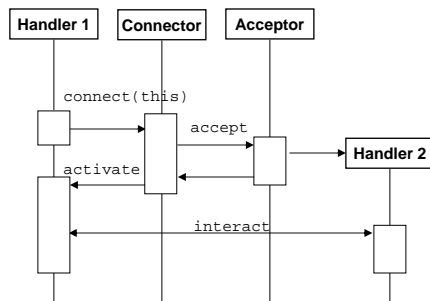
Acceptor-Connector: Structure



Solution Structure: Separate connection establishment from service processing.

- *Service Handlers* implement the components that want to interact.
- *Connectors* actively establish new connections on behalf of a Service Handler to a remote Acceptor.
- *Acceptors* passively accept connection requests from remote Connectors and initialize a Service Handler to service the connection.
- *Note: both synchronous and asynchronous connection establishment can be supported.*

Acceptor-Connector: Dynamics



Solution Dynamics: Let Connector and Acceptor establish a connection and pass them on to interacting Service Handlers.

- A *Concrete Service Handler* asks a *Concrete Connector* to establish a connection to a specific remote *Concrete Service Handler*.
- The *Connector* sends a connection request to a *Concrete Acceptor* residing on the respective server.
- *Connector* and *Acceptor* establish the connection and pass them on to 'their' *Concrete Service Handlers*, which then interact.

Acceptor-Connector: Consequences



Benefits

- Separation of concerns: connection establishment is separated from application-specific service processing.
- Reusability: connectors and acceptors are reusable components.
- Efficiency: while service handlers still interact, connectors and acceptors can already establish a new connection.

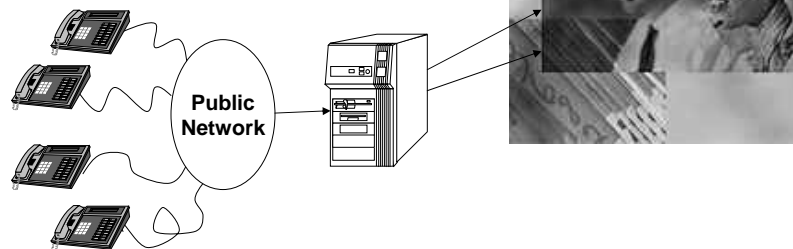


Liabilities

- Complexity: complex infrastructure and complex implementation for the asynchronous connection establishment.

Acceptor-Connector: Example

Call center applications, such as the Ericsson EOS Call Center Management System or Siemens FlexRouting, use Acceptor-Connector configurations to establish connections actively with passive supervisors (or agents).



Acceptor-Connector: Known Uses

Web Browsers: Internet Explorer, Netscape to connect to servers associated with images embedded in html pages.

Object Request Brokers: The ACE ORB (TAO) and Orbix 2000 to establish connections to ORB services.

ACE: provides a reusable Acceptor-Connector framework

Call Center Management Systems: Ericsson, Siemens to establish connections to supervisors.



Concurrency (I)

A web server must serve multiple clients simultaneously and at a high-quality of service.

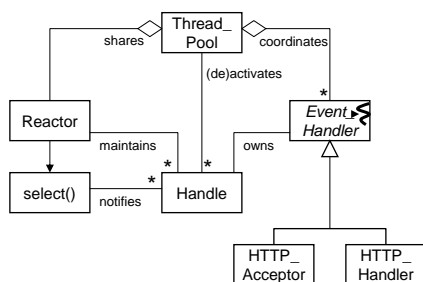
Multi-threaded, concurrent event processing is a typical response to this requirement. Yet:

- Thread management overhead should be minimized to not degrade performance unnecessarily.
- Resource consumption should be predictable to avoid server saturation.
- Throughput is important. All events are therefore of identical priority, they are processed in the order of their arrival.



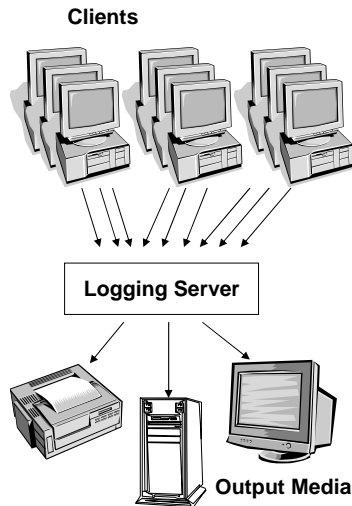
Concurrency (II)

Let multiple Leader/Followers service threads share a common set of event sources by taking turns discovering demultiplexing, dispatching, and processing events:



- A thread pool (**Thread_Pool**) coordinates a group of event handler threads to share a common set of event sources to process events.
- Event handlers (**HTTP_Handler**, **HTTP_Acceptor**) join the thread pool to process client HTTP requests.
- A Reactor (**Reactor**) maintains the set of shared event sources.
- Handles (**Handle**) represent the set of event sources shared by the event handler threads.

Leader/Followers: Problem

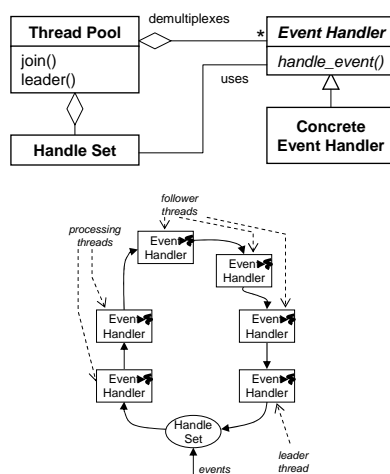


Leader/Followers (POSA2)

Problem: An event-driven application may use multi-threading to serve multiple client requests simultaneously. However:

- Concurrency overhead, such as thread-creation, context switching, thread coordination, should not outweigh the benefits of multi-threading.
- Resource consumption should be predictable to avoid saturation of the event-driven application.
- Some concurrent application also trade general throughput and performance over scheduling and prioritizing.
- Requests are structured, repetitive, require immediate response, and can be served via short, atomic, and isolated processing actions.

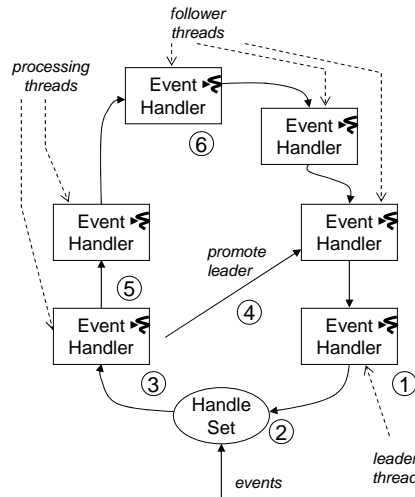
Leader/Followers: Structure



Solution Structure: Introduce a thread pool in which multiple service instances share a common event source.

- Events arrive on a *Handle Set* shared by all threads.
- *Event Handler* threads implement application services:
 - A *Leader* Thread gets exclusive access to the *Handle Set*, and blocks until an event arrives.
 - *Follower* Threads queue up behind the leader and wait until it is their turn to be the leader.
 - *Processing* Threads are processing an event they received when being the leader.
- A *Thread Pool* coordinates the *Event Handler* threads.

Leader/Followers: Dynamics



Solution Dynamics: Let the Event Handler threads take turns on the shared Handle Set.

- All threads are spawned. One thread obtains access to the *Handle Set* and becomes the *Leader* thread (1), all other threads become *Followers*.
- An event occurs on the *Handle Set* and is received by the *Leader* thread (2).
- The *Leader* thread changes its role to a *Processing* thread and processes the request (3), the *Thread Pool* promotes one *Follower* to become the new *Leader* (4).
- While the event is processed (5), steps (2) - (4) recur.
- When the event processing finished the *Processing* thread changes its role to a *Follower* and waits until it becomes the new *Leader* again (6).

Leader/Followers: Consequences



Benefits

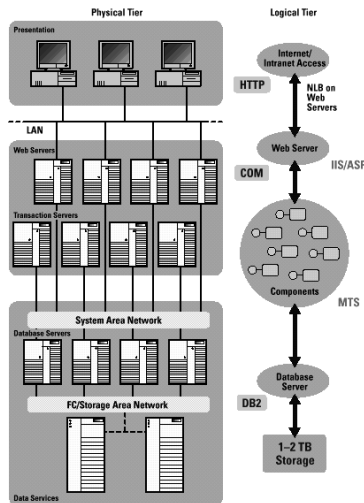
- Simple thread management: threads coordinate themselves via the pool; all threads are spawned at system start-up.
- Predictable resource consumption: the thread pool is of fixed size
- Performance: One of the fastest—if not the fastest—concurrency architecture.



Liabilities

- Most suitable for applications with a small set of atomic services.
- Only suitable for systems whose services have a small memory footprint.
- No scheduling / priority handling possible.

Leader/Followers: Example



On-Line Transaction Processing (OLTP)
Systems, such as for travel agencies, financial services, or broker information, often use the a Leader/Followers concurrency architecture to perform high-volume transaction processing on backend database servers.

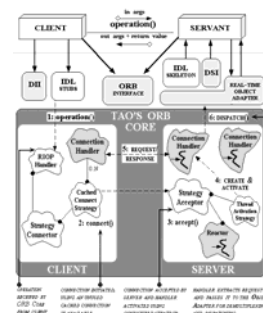


Leader/Followers: Known Uses

Object Request Brokers: The ACE ORB (TAO) and to handle network events.

Application Servers: Application Server to handle network events.

Transaction Monitors: Tuxedo, MTS
Transaction Service to process transactions.



TAO Features

- Open-source
- 500+ classes
- 500,000+ lines of C++
- ACE/patterns-based
- 30+ person-years of effort
- Ported to UNIX, Win32, MVS, and many RT & embedded OSs (e.g., VxWorks, LynxOS, Chorus, QNX)

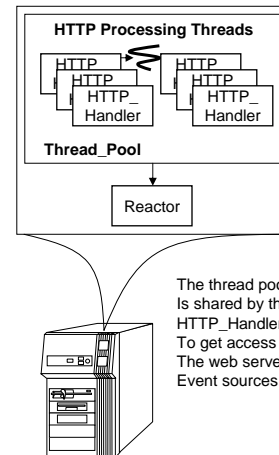


Concurrency Coordination (I)

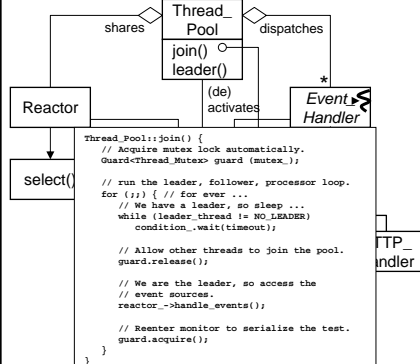
To avoid concurrency hazards like race conditions, the access sequence of threads to shared web server components must be coordinated.

Some of these shared components—such as the `Thread_Pool`—are relatively small-sized objects with a fairly simple, straight-forward logic.

- It is crucial that the `Thread_Pool` executes efficiently—it is the heart of the web servers event handling infrastructure. Therefore, the thread coordination mechanism should be efficient.
- Expensive thread coordination overhead should be avoided.



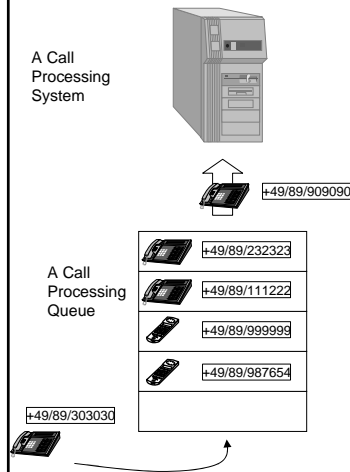
Concurrency Coordination (II)



Coordinate the access to the `Thread_Pool` by implementing it as a Monitor Object:

- The thread pool (`Thread_Pool`) is a monitor object.
- The thread pool's services (`join()`, `leader()`) are synchronized methods.
- The thread pool implementation uses a monitor lock (`mutex_`) and a condition variable (`condition_`) to synchronize the access to its services transparently for the HTTP processing threads (`HTTP_Acceptor`, `HTTP_Handler`).

Monitor Object: Problem

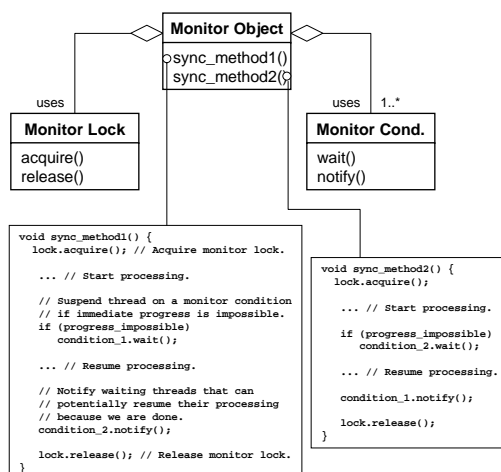


Monitor Object (POSA2)

Problem: Concurrent access to 'small' concurrent objects such as a queues or stacks should be as simple as possible:

- Synchronization should be easy to implement.
- Threading and synchronization overhead should be minimized or, even better, avoided.
- Yet clients should not notice that the object is shared.

Monitor Object: Structure




Solution Structure: Synchronize an object's methods so that only one method can execute at any one time:

- A *Monitor Object* is an object shared among multiple threads.
- *Synchronized Methods* implement thread-safe functions of the Monitor Object.
- One or more *Monitor Conditions* allow the Synchronized Methods to schedule their execution.
- A *Monitor Lock* ensures a serialized access to a Monitor Object.


Solution Dynamics: Let the Monitor Object schedule its own method execution sequence.

- A Client Thread invokes a *Synchronized Method*. The method acquires the *Monitor Lock* and starts executing.
- At some point, the method cannot proceed, therefore it suspends itself on the *Monitor Condition*, which automatically releases the lock and puts the calling thread to sleep.
- Another thread can execute a *Synchronized Method* on the monitor object. Before terminating, this method notifies the *Monitor Condition* on which the first thread is waiting.
- If the *Monitor Condition* evaluates to true, it automatically re-acquires the *Monitor Lock* and resumes the suspended thread as well as the suspended *Synchronized Method*.



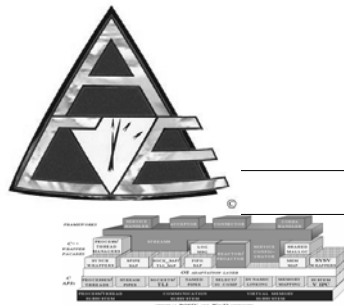
Benefits

- Simple yet effective: Simplification of concurrency control; simplification of scheduling method execution.



Liabilities

- Limited scalability: clients can block.
- Hard to extend: functionality and synchronization logic are tightly coupled.
- Nested Monitor Lockout Problem.



Efficient Content Access (I)

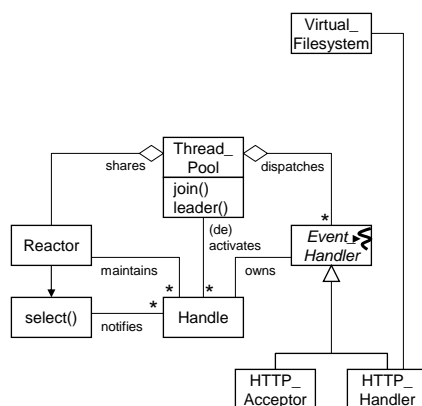
To achieve a high throughput and a good quality of service, a web server must access URLs requested by clients efficiently.

Retrieving the URLs from the file system for every access to them introduces performance penalties, however:

- Some URLs are more frequently accessed than others.
- Individual clients often return to an URL they already visited before.



Efficient Content Access (II)



Introduce a virtual file system to have the content of frequently accessed URLs readily accessible via Caching:

- A cache (Virtual_Filesystem) keeps URL content in memory once it got accessed and loaded from the file system for the first time. For subsequent accesses, this content is readily available

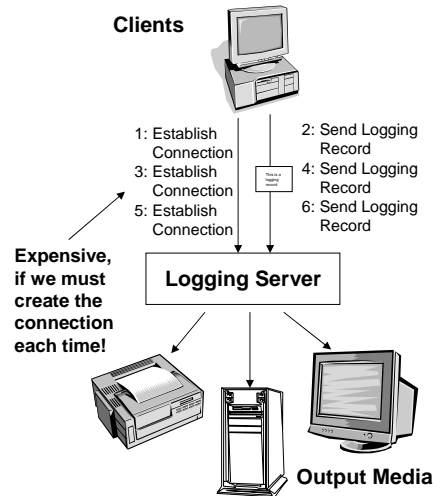
HTTP_Handlers first check the cache for URL availability before they load it from the file system.

Caching: Problem

Caching (POSA3)

Problem: Repetitious acquisition, initialization, and release of resources causes unnecessary performance overhead:

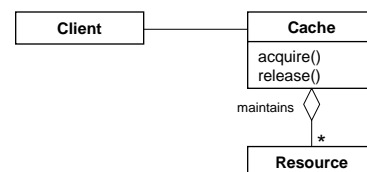
- The costs of acquisition, initialization, and release of frequently used resources should be minimized.
- Access to the resources should be simple.



Caching: Structure

Solution Structure: Instead of releasing a resource after its usage, temporarily store it in an in-memory buffer:

- *Resources* represent entities that application functionality uses to execute its business logic, such as threads, network connections, file handles, memory, etc.
- A *Cache* keeps currently unused resources of a specific type readily available for subsequent use.
- *Clients* use the Resources maintained by the Cache to execute their own functionality.



Caching: Dynamics

```
sequenceDiagram
    participant Client
    participant Cache
    participant Resource



    Client->>Cache: acquire
    Cache->>Resource: create
    Resource-->>Cache: 
    Cache-->>Client: Resource
    Client->>Resource: use
    Resource-->>Cache: 
    Client->>Cache: release
    Cache->>Resource: 
    Resource-->>Cache: Resource
    Cache-->>Client: Resource
    Client->>Cache: acquire
    Cache->>Resource: 
    Resource-->>Cache: Resource
    Cache-->>Client: Resource
    Client->>Resource: use
    Resource-->>Cache: 
```

The diagram illustrates the interaction between a Client, a Cache, and a Resource. The Client requests a Resource from the Cache (acquire). The Cache checks if the Resource is available; if not, it creates one (create) and returns it to the Client (Resource). The Client uses the Resource (use). The Client releases the Resource back to the Cache (release). The Cache keeps the Resource for subsequent uses. The Client requests the Resource from the Cache again (acquire). The Cache returns the already existing Resource (Resource). The Client re-uses the Resource (use).

Solution Dynamics: Create resources once and re-cycle them often:

- A *Client* requests a *Resource* from the *Cache*.
- The *Cache* checks if the *Resource* is available and creates one, if not
- The *Client* uses the *Resource*.
- The *Client* releases the resource by passing it back to the *Cache*.
- The *Cache* keeps the *Resources* for subsequent uses.
- The *Client* requests the *Resource* from the *Cache* again.
- The *Cache* returns the already existing *Resource*.
- The *Client* ,re-uses' the *Resource*.

Caching: Consequences



Benefits

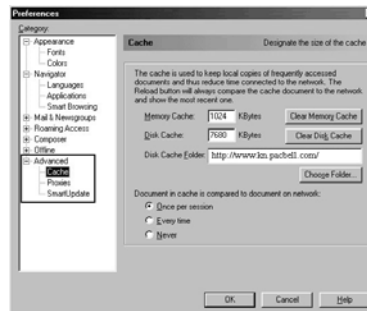
- Performance: Quick access to resources, minimized resource acquisition and release costs.
- Predictability: Explicit control of resource usage possible.

Liabilities

- Cache maintenance overhead: If the cache is full, its 're-organization' can cause performance penalties in accessing and using resources.

Caching: Example

Web Browsers: Web Browsers like Netscape and Internet Explorer use a cache to keep local copies of frequently accessed documents.



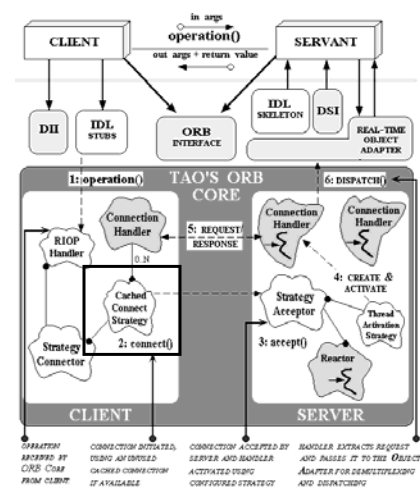
Caching: Known Uses

Web Browsers: Internet Explorer and Netscape cache frequently accessed documents.

Web Servers: Apache and Jaws cache frequently requested documents.

Object Request Brokers:
The ACE ORB (TAO) caches unused network connections and re-cycles them for new client connections.

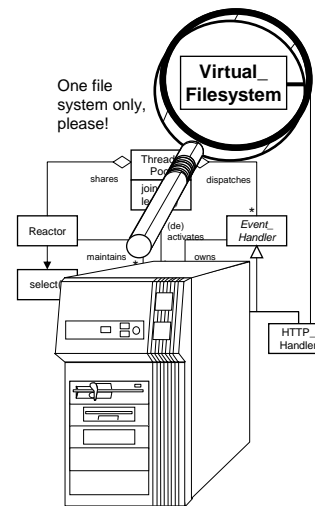
High-performance applications:
Almost every high-performance software uses caches to speed-up performance.



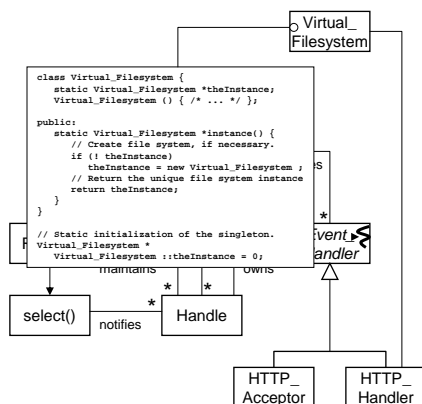
One Cache Only (I)

To achieve predictability regarding resource consumption, there should only be one virtual file system in the web server.

However, OO languages do not support an instance-controlled object creation.



One Cache Only (II)



Ensure that the web server's **Virtual_Filesystem** can be instantiated just once by making it a **Singleton**:

- A singleton (**Virtual_Filesystem**) keeps its constructors and destructors private so that only itself can create instances of itself.

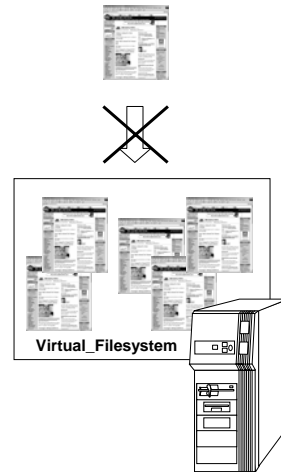
A class access method ensures that only one instance of a singleton class can be created and offers clients an access to this unique instance.

Note that we trade safety for flexibility here!

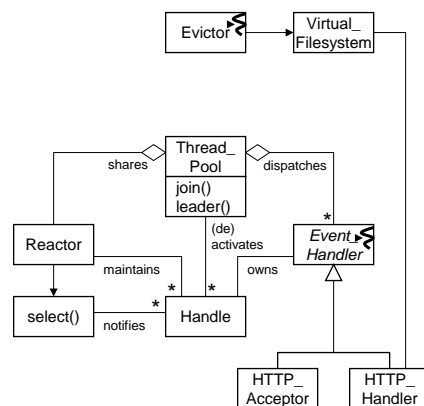
Cache Management (I)

If we only load URLs into the web server's cache, it will be full at some point in time and we cannot load any further URLs.

- The less frequently the loaded URLs are accessed, the more the performance and throughput of the web server degrades.
- Any run-time flushing of the cache **must not** degrade the operational performance of the web server.



Cache Management (II)



Manage the content of the web server's **Virtual_Filesystem** and release unused URLs via an **Evictor**:

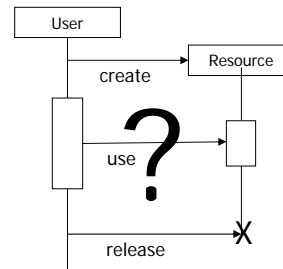
- An evictor (**Evictor**) runs in its own thread of control. Once the occupation of the **Virtual_Filesystem** exceeds a certain limit, it notifies the evictor to remove less frequently used URLs from it.

Evictor: Problem

Evictor (POSA3)

Problem: Distributed systems need to manage resources such as file handles or CPU time efficiently. Over time resources that are not required any more should be freed. Immediate resource release, however, might cause expensive re-acquisitions:

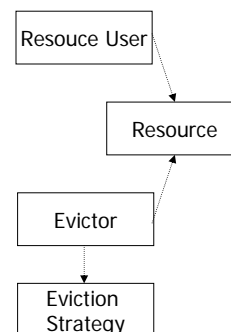
- The frequency of resource usage should influence its lifecycle.
- Resource release should be parametrized by issues such as CPU time.
- The solution should be transparent to the user.



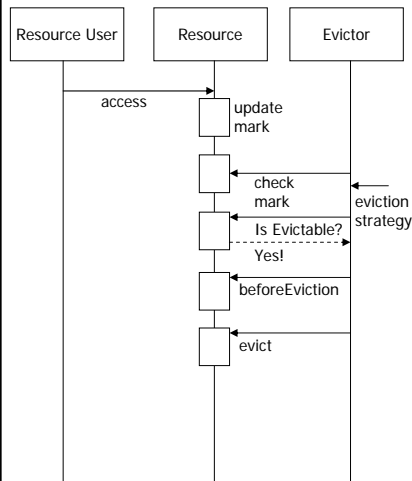
Evictor: Structure

Solution Structure: Monitor resources and control their lifecycles using a strategy such as LRU. Each resource is marked on usage. Recently or infrequently used resources do not get marked. The application periodically or on demand selects unmarked resources for eviction:

- *Resource users* represent applications or OS that use resources.
- A *Resource* provides some local or remote service.
- An *Evictor* releases resources depending on one or more eviction strategies.
- An *Eviction Strategy* describes the strategy used to determine whether a resource should be evicted.



Evictor: Dynamics



Solution Dynamics: Register resources with the evictor. Evictor checks resources for usage marks. If strategy tells evictor to free resource evictor asks resource whether it is evictable. If evictable resource is informed and released.

Evictor: Consequences



Benefits

- Scalability: Evictor helps to keep upper bound of resources. Applications can scale without memory impact.
- Low-memory footprint: Eviction strategy helps to keep only required resources in memory, thus reducing memory footprint.
- Transparency: User does not see implementation of evictor.
- Stability: Reduces propability of resource exhaustion.



Liabilities

- Overhead: Additional logic to determine which resources to evict and to implement eviction strategy.
- Re-Acquisition Penalty: Resurrecting an already evicted resource might be expensive. Finetune eviction strategy to prevent this.

Evictor: Known Uses

.NET: Garbage Collection.

EJB: Release of enterprise beans.

Weak References in .NET and Java:

Using weak references disposable objects can be used and resurrected.

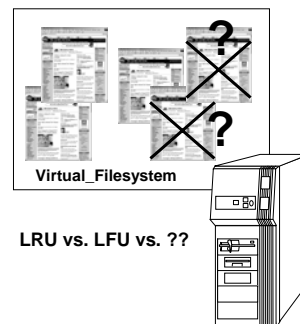
CORBA: Servant Manager uses eviction to get rid of servants.

Operating Systems: Swapping implemenations use the Evictor pattern.

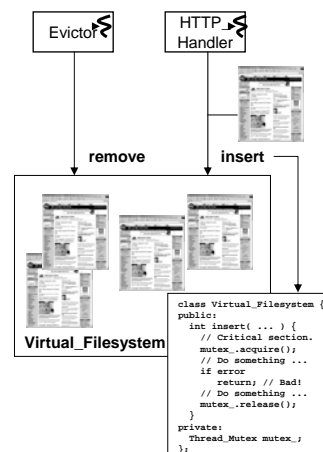
Flexible Cache Management (I)

Different web server usage scenarios could require different cache eviction policies.

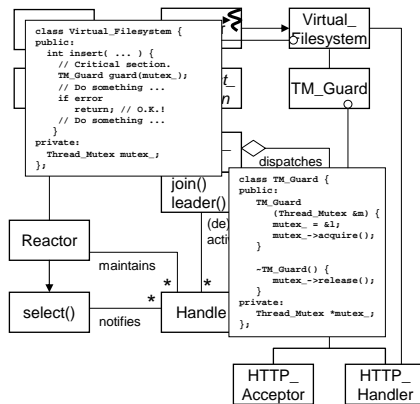
- Hard-coding cache eviction policies into the evictor pollutes its code with many conditional statements, which ultimately impede performance.
- Hard-coding cache eviction policies into the evictor degrades its maintainability and extensibility.
- Changing the current cache eviction policy must be possible at run-time to avoid shutting down the web-server (means taking it out of service) unnecessarily.



- ## Configure the Evictor with cache eviction policies dynamically via Strategies:
- An abstract strategy (`Abstract_Eviction`) defines the interface for different cache eviction policies.
 - Concrete Strategies (`LFU`, `LRU`) implement the different cache eviction policies.
 - A context (`Evictor`) is (dynamically) configured with a particular concrete strategy. This strategy is called whenever the context wants to execute the respective type of algorithm.



Reliable Synchronization (II)



Protect critical regions in the Virtual_Filesystem via Scoped Locking:

- A guard class (TM_Guard) automates the acquisition and release of a given type of lock.
- The shared component (Virtual_Filesystem) uses the guard to protect its critical section from concurrent access.

Scoped Locking: Problem

Scoped Locking (POSA2)

Problem: Protecting critical sections from concurrent access proxies prevents damaging state in an object. However:

- Locks that protect the critical sections should always be acquired and released properly.
- It is hard to ensure that locks are released in all exit paths from the critical section.

```
class Foo {
    ThreadMutex lock;

public:
    void bar() {
        // Do something ...

        // Acquire lock to serialize the
        // access to the critical section.
        lock.acquire();

        // Do something ...
        if (error)
            return;

        // Release lock.
        lock.release();

        // Do something ...
    }
}
```

BAD!

Scoped Locking: Structure

Solution Structure: Scope the critical section and acquire and releases a lock automatically.

- A *Guard* acquires a lock in its constructor and releases this lock in its destructor.
- A scoped *Critical Section* acquires a guard object to serialize access to it.

```
class Guard {
    ThreadMutex *lock;

public:
    Guard (ThreadMutex &l) {
        lock = &l;
        lock->acquire();
    }

    Guard() {
        lock->release();
    }
}
```

```
class Foo {
    ThreadMutex lock;

public:
    void bar() {
        // Do something ...

        { // Begin of critical section.
            Guard guard(lock);

            // Do something ...
            if (error)
                return;
        } // End of critical section.

        // Do something ...
    }
}
```

O.K.!

Scoped Locking: Dynamics

Solution Dynamics: Acquire a lock explicitly and let the run-time environment release it properly:

- Creating the *Guard* instance on the stack after entering the scope of the *Critical Section* acquires the lock.
- Leaving the *Critical Section* means leaving its scope, which lets the run-time environment call the destructor of the *Guard* on the stack, which releases the lock.

```
class Foo {
    ThreadMutex lock;

public:
    void bar() {
        // Do something ...

        { // Begin of critical section.
            1a Guard guard(lock);

            // Do something ...
            if (error)
                return;
        } 2a // End of critical section.

        // Do something ...
    }
}
```

```
class Guard {
    ThreadMutex *lock;

public:
    1b Guard (ThreadMutex &l) {
        lock = &l;
        lock->acquire();
    }

    2b ~Guard() {
        lock->release();
    }
}
```

Scoped Locking: Consequences



Benefits

- Robustness: Locks are acquired and released properly—regardless of the exit path from the critical section.
- In Java: Built-in language feature.



Liabilities

- For example, in C++ Language Feature Dependent: Works only “inside” true C++. For instance, calling the C function `longjmp()` does not call destructors of C++ objects when the stack unwinds.

Scoped Locking: Example

.NET defines a feature called a
synchronized block that implements
Scoped Locking.

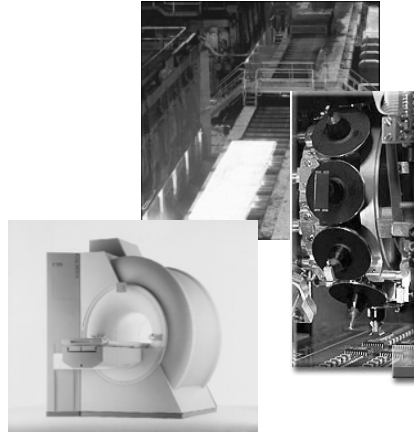
```
class Dispatcher
{
    ...
    public void changeHandler(Handler newHandler)
    {
        lock(eventSource)
        {
            oldHandler = eventSource.getHandler(event);
            eventSource.installHandler(event, newHandler);
        }
    }
    ...
    private EventSource eventSource;
    private Event event;
    private Handler oldHandler;
}
```

Scoped Locking: Known Uses

Concurrent software: Almost all concurrent software systems use scoped locking to ensure a proper acquisition and release of locks that protect critical regions.

Class libraries: Class libraries like ACE, threads.h++, and the Booch components define a set of guard classes that follow scoped locking.

.NET, Java: the synchronized blocks provide built-in scoped locking.

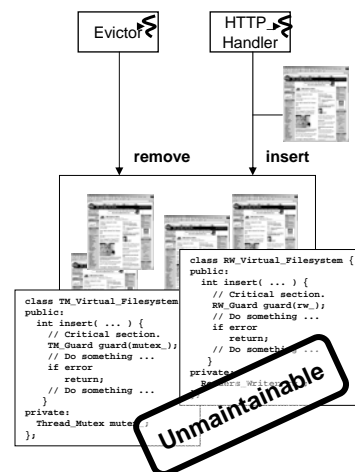


Efficient Synchronization (I)

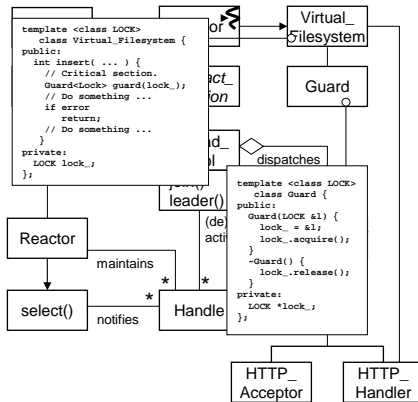
A web server should be able to run on multiple platforms, ranging from single to multi-processor platforms.

On different platform, different types of lock provide different performance. For instance, a mutex is most suitable for single processor platforms whereas a readers/writer lock is often most efficient for multi-processor platforms.

However, maintaining multiple variants of every shared object that differ only in the lock type is very ineffective.



Efficient Synchronization (II)



Configure the type of lock used by the `Virtual_Filesystem` via **Strategized Locking**:

- A guard class (`Guard`) automates the acquisition and release of a configurable type of lock.
- The shared component (`Virtual_Filesystem`) uses the guard to protect its critical section from concurrent access and configures it with the appropriate type of lock.

Strategized Locking: Problem

Strategized Locking (POSA2)

Problem: The most optimal locking mechanisms for a piece of code can vary dependent on the platform used. However:

- Maintaining multiple variants of code that differs only in the lock type used is very ineffective.



```
class singleton {
    static singleton *theInstance;
    singleton () { /* ... */ }

public:
    static singleton *getInstance() {
        if (! theInstance) {
            // Acquire lock to serialize the
            // access to the critical section.
            lock.acquire();

            // Create singleton, if necessary.
            if (! theInstance)
                theInstance = new singleton;

            // Release lock.
            lock.release();
        }
        // Return the singleton instance.
        return theInstance;
    }
}

// Static singleton initialization.
singleton *singleton::theInstance = 0;
```

Thread mutex
on single-processor
platforms.

Readers/Writer
lock on multi-processor
platforms.



Strategized Locking: Structure

```
template <class LOCK> class Singleton {
    static Singleton *theInstance;
    Singleton () { /* ... */ };
    LOCK lock;

public:
    static Singleton *getInstance() {
        if (! theInstance) {
            // Acquire lock to serialize the
            // access to the critical section.
            lock.acquire();

            // Create singleton, if necessary.
            if (! theInstance)
                theInstance = new Singleton;

            // Release lock.
            lock.release();
        }
        // Return the singleton instance.
        return theInstance;
    }
}
```

Solution Structure: Modify the code so that it can be configured with the appropriate type of lock, either statically using generic types or dynamically using polymorphism.

Strategized Locking: Dynamics

```
template <class LOCK> class Singleton {
    static Singleton *theInstance;
    Singleton () { /* ... */ };
    LOCK lock;

public:
    static Singleton *getInstance() {
        if (! theInstance) {
            // Acquire lock to serialize the
            // access to the critical section.
            lock.acquire();

            // Create singleton, if necessary.
            if (! theInstance)
                theInstance = new Singleton;

            // Release lock.
            lock.release();
        }
        // Return the singleton instance.
        return theInstance;
    }
}
```

```
Singleton<ThreadMutex> *Singleton<ThreadMutex>::theInstance = 0;
```

Solution Dynamics: Configure the code with the appropriate type of lock during its initialization.

Strategized Locking: Consequences



Benefits

- Flexibility: Critical sections can be configured with the most appropriate type of lock, dependent on usage scenarios and platforms on which the code runs.
- Maintenance: Code becomes easier to maintain and (re-)use because it becomes independent on locking mechanisms.



Liabilities

- Visible Locking: If templates are used it is visible to application code what locking mechanism is used in a piece of code.

Strategized Locking: Example

ACE: The open source ADAPTIVE Communication Environment provides a reusable lock guard class that uses templated strategized locking.



```
template <class LOCK> class Guard {
public:
    Guard(LOCK &l) : lock_ (&lock), owner_ (false) {
        lock_.acquire();
        owner_ = true;
    }

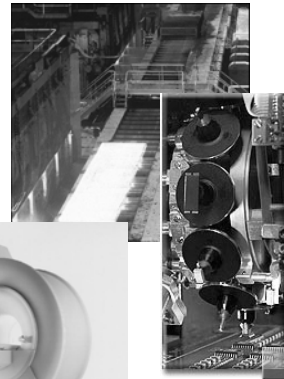
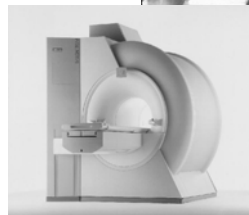
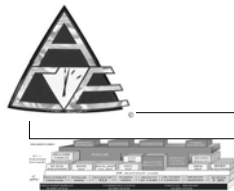
    ~Guard() {
        if (owner_) lock_.release();
    }

private:
    LOCK *lock_;
    bool owner_
};
```

Strategized Locking: Known Uses

Concurrent software: Almost all concurrent software systems use strategized locking to ensure a proper acquisition and release of locks that protect critical regions.

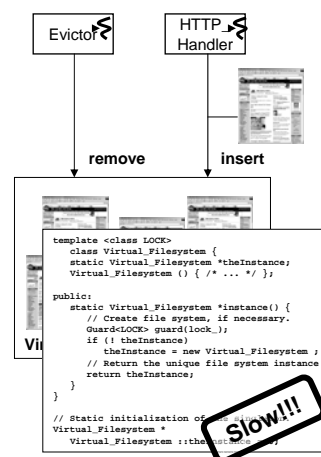
Class libraries: Class libraries like ACE use strategized locking.



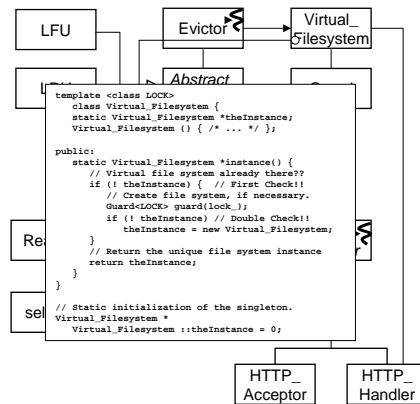
Efficient Synchronization (III)

The web server's virtual file system is Singleton. It is also a component shared by multiple threads, thus we must prevent an accidental double instantiation, due to race conditions.

However, all locking and synchronization overhead should be minimized.



Efficient Synchronization (IV)



Introduce a flag that provides a hint whether it is necessary to execute a critical section before acquiring the lock that guards it, using Double-Checked Locking.

Double-Checked Locking: Problem

Double-Checked Locking (POSA2)

Problem: Sometimes, critical sections are not always executed, but only conditionally. Some such critical sections are even executed just once. However:

- Despite the need for serializing the access to the critical section, there should be no locking overhead for those situations in which the critical section is not executed.

```
class Singleton {
    static Singleton *theInstance;
    Singleton () { /* ... */ };

public:
    static Singleton *getInstance() {
        // Acquire lock to serialize the
        // access to the critical section.
        lock.acquire();

        // Create singleton, if necessary.
        if (! theInstance)
            theInstance = new Singleton;

        // Release lock.
        lock.release();

        // Return the singleton instance.
        return theInstance;
    }
}

// Static singleton initialization.
Singleton *Singleton::theInstance = 0;
```

SLOW!

Double-Checked Locking: Structure

```
class Singleton {
    static Singleton *theInstance;
    Singleton () { /* ... */ };

public:
    static Singleton *getInstance() {
        if (! theInstance) { 1st Check
            // Acquire lock to serialize the
            // access to the critical section.
            lock.acquire();

            // Create singleton, if necessary.
            if (! theInstance) 2nd Check
                theInstance = new Singleton;

            // Release lock.
            lock.release();
        }
        // Return the singleton instance.
        return theInstance;
    }
}

// Static singleton initialization.
Singleton *Singleton::theInstance = 0;
```

Solution Structure:

Double-Check:

- Introduce a *flag* that provides a hint whether it is necessary to execute a critical section before acquiring the lock that guards it.

Double-Checked Locking: Dynamics

Solution Dynamics: Skip the critical section if it does not need to be executed:

- If the first check reveals that the critical section can be skipped. No locking overhead occurs.
- If the critical section needs to be executed, a lock is acquired to prevent race conditions if two or more threads passed the first check.
- Once the lock is acquired, the need for executing the critical section is checked again, so that it is executed only when needed.

```
class Singleton {
    static Singleton *theInstance;
    Singleton () { /* ... */ };

public:
    static Singleton *getInstance() {
        ① if (! theInstance) {
            // Acquire lock to serialize the
            // access to the critical section.
            ② lock.acquire();

            // Create singleton, if necessary.
            ③ if (! theInstance)
                theInstance = new Singleton;

            // Release lock.
            lock.release();
        }
        // Return the singleton instance.
        return theInstance;
    }
}

// Static singleton initialization.
Singleton *Singleton::theInstance = 0;
```

Double-Checked Locking: Consequences



Benefits

- Avoids locking overhead: Locks are acquired only when necessary, the most common cases perform fast.



Liabilities

- Awkward implementation: Can require processor-specific instructions on certain platforms due to multi-processor cache coherency.
- **Does not work in Java!**

Double-Checked Locking: Example

Solaris: The implementation of `errno` on Solaris uses double-checked locking.

```
#define errno (*(__errno()))

int *__errno () {
    static pthread_mutex_t keylock;
    static pthread_key_t key;
    static int once;
    int *error_number = 0;

    if (once) {
        pthread_mutex_lock (&keylock);
        if (once) {
            pthread_key_create (&key, free);
            once = 1;
        }
        pthread_mutex_unlock (&keylock);
    }

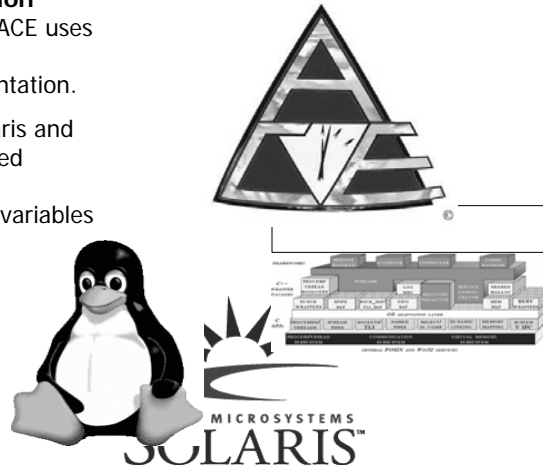
    pthread_getspecific (key, &error_number);
    if (error_number == 0) {
        error_number = (int *) malloc (sizeof (int));
        pthread_setspecific (key, error_number);
    }
    return error_number;
}
```

Double-Checked Locking: Known Uses

ADAPTIVE Communication

Environment (ACE): ACE uses double-checked locking throughout its implementation.

Operating Systems: Solaris and Linux use double-checked locking, for instance to implement POSIX once variables or POSIX Pthreads.



Accessing OS APIs (I)

Developing portable web server is hard if developers must wrestle with low-level operating system APIs.

- OS APIs differ from platform to platform regarding both syntax and semantic.
- OS dependencies in system code result in non-portable code.
- There is a mismatch between procedural programming style of OS APIs and the object paradigm used in the component's implementation.
- Programming low-level APIs is inherently error-prone.

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



```

classDiagram
    class LFU
    class LRU
    class Evictor
    class Virtual_Filesystem
    class Guard
    class Abstract_Eviction
    class Thread_Pool {
        join()
        leader()
    }
    class Reactor
    class select_["select()"]
    class Handle
    class Event_Handler {
        <<abstract>>
    }
    class HTTP_Acceptor
    class HTTP_Handler

    LFU --> Evictor
    LRU --> Abstract_Eviction
    Evictor --> Virtual_Filesystem
    Virtual_Filesystem --> Guard
    Abstract_Eviction --> Thread_Pool
    Thread_Pool --> Reactor : shares
    Thread_Pool --> Event_Handler : dispatches
    Reactor --> select_
    Reactor --> Handle : maintains
    select_ --> Handle : * notifies
    Thread_Pool --> Handle : *
    Handle --> Event_Handler : * owns
    Event_Handler <|-- HTTP_Acceptor
    Event_Handler <|-- HTTP_Handler
  
```

UML class diagram illustrating the libevent architecture components and their relationships:

- LFU** (Least Frequently Used) and **LRU** (Least Recently Used) are associated with **Evictor** and **Abstract_Eviction** respectively.
- Evictor** is associated with **Virtual_Filesystem**.
- Virtual_Filesystem** is associated with **Guard**.
- Abstract_Eviction** is associated with **Thread_Pool**.
- Thread_Pool** (containing `join()` and `leader()`) is associated with **Reactor** (via `shares`) and **Event_Handler** (via `dispatches`).
- Reactor** is associated with **select()** and **Handle** (via `maintains`).
- select()** is associated with **Handle** (via `* notifies`).
- Thread_Pool** is associated with **Handle** (via `*`).
- Handle** is associated with **Event_Handler** (via `* owns`).
- Event_Handler** is an abstract class with **HTTP_Acceptor** and **HTTP_Handler** as subclasses.

Legend:

- Socket** (represented by a rounded rectangle)
- Thread** (represented by a rectangle)
- Thread_Mutex** (represented by a rectangle with a wavy line)
- Condition_Variable** (represented by a rectangle with a wavy line)

- A set of wrapper facades (`Socket`, `Thread`, `Thread_Mutex`, `Condition_Variable`) provide a portable, robust, and modular access to low-level OS APIs.
- Application classes (our web server classes) use the wrapper facades in their implementation.

```
// Main driver function for a logging server.
int main (int argc, char *argv[]) {
    struct sockaddr_in sock_addr;
    // Handle UNIX/Win32 portability.
    #if defined (_WIN32)
        SOCKET acceptor;
        WORD version_requested = MAKEWORD(2, 0);
        WSADATA wsa_data;
        int error = WSAStartup(version_requested, &wsa_data);
        if (error != 0) return -1;
    #else
        int acceptor;
    #endif // _WIN32 */
    // Create a local endpoint of communication.
    acceptor = socket (PF_INET, SOCK_STREAM, 0);

    // Set up the address to become a server.
    memset (&sock_addr, 0, sizeof sock_addr);
    sock_addr.sin_family = PF_INET;
    sock_addr.sin_port = htons (LOGGING_PORT);
    sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);

    // Associate address with endpoint.
    bind (acceptor, reinterpret_cast<struct sockaddr *>
        (&sock_addr), sizeof sock_addr);

    // Make endpoint listen for connections.
    listen (acceptor, 5);

    // Main server event loop.
    for (;;) {
        // Handle UNIX/Win32 portability.
        #if defined (_WIN32)
            SOCKET h;
            DWORD t_id;
        #else
            int h;
            thread_t t_id;
        #endif // _WIN32 */
    }
}
```

- applications should be portable.
- direct dependencies of application code to these APIs should be avoided.
- applications should use these APIs correctly and consistently.

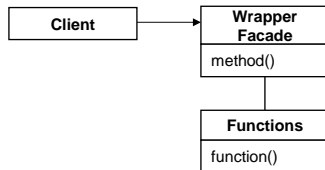
```

    // Block waiting for clients to connect.
    h = accept(acceptor, 0, 0);

    // Spawn a new thread that runs the <server> entry point.
    if defined (_WIN32)
        CreateThread(0, 0, LPTHREAD_START_ROUTINE(logging_handler),
                    reinterpret_cast void* > (h), 0, &tid);
    #else
        thr_create(0, 0, logging_handler, reinterpret_cast void* > (h),
                  THR_DETACHED, &tid);
    #endif /* _WIN32 */
    return 0;
}

```

Wrapper Facade: Structure



```
class INET_Addr {
public:
    INET_Addr (u_short port, u_long addr = 0) {
        // Set up the address to become a server.
        memset (&addr_, 0, sizeof addr_);
        addr_.sin_family = PF_INET;
        addr_.sin_port = htons (port);
        addr_.sin_addr.s_addr = htonl (addr);
    }

    u_short get_port () const
    { return ntohs (addr_.sin_port); }
    u_long get_ip_addr () const
    { return ntohl (addr_.sin_addr.s_addr); }

    sockaddr *addr () const
    { return reinterpret_cast <sockaddr *> (&addr_); }

    size_t size () const { return sizeof (addr_); }

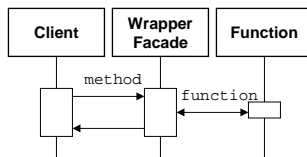
    // ...
private:
    sockaddr in_addr;
};
```

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.

```
class SOCK_Acceptor {
public:
    // Initialize a passive-mode acceptor socket.
    SOCK_Acceptor (const INET_Addr &addr) {
        // Create a local endpoint of communication.
        handle_ = socket (PF_INET, SOCK_STREAM, 0);
        // Associate address with endpoint.
        bind (handle_, addr.addr (), addr.size ());
        // Make endpoint listen for connections.
        listen (handle_, 5);
    };
    // ...
};
```

Wrapper Facade: Dynamics



Solution Dynamics: Forward method invocations on a Wrapper Facade to the appropriate low-level Functions:

- A *Client* invokes a method on a *Wrapper Facade*.
- The *Wrapper Facade* executes one or more low-level *Functions* in the right order, thereby performing all necessary data conversions and error handling.

Wrapper Facade: Consequences



Benefits

- Robustness: Low-level APIs are used correctly.
- Portability: An application depends on unified, semantically expressive interfaces, not on low-level API syntax.
- Modularity: Wrapper Facades are low-level, self-contained building blocks.



Liabilities

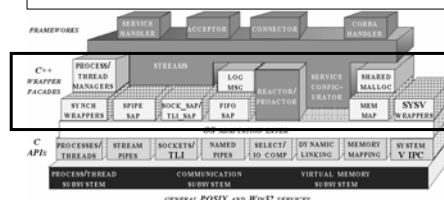
- Indirection: Wrapper Facades add a level of indirection between application code and API code.

Wrapper Facade: Example



The open source
**ADAPTIVE
Communication
Environment
(ACE)**

ACE Class	Description
ACE_Addr	The root of the ACE network addressing hierarchy.
ACE_INET_Addr	Encapsulates the Internet-domain address family.
ACE_IPC_SAP	The root of the ACE IPC wrapper facade hierarchy.
ACE_SOCKET	The root of the ACE Socket wrapper facade hierarchy.
ACE_SOCKET_Connector	A factory that connects to a peer acceptor and then initializes a new endpoint of communication in an ACE_SOCKET_Stream object.
ACE_SOCKET_IO	Encapsulate the data transfer mechanisms supported by data-mode sockets.
ACE_SOCKET_Stream	A factory that initializes a new endpoint of communication in an ACE_SOCKET_Stream object in response to a connection request from a peer connector.

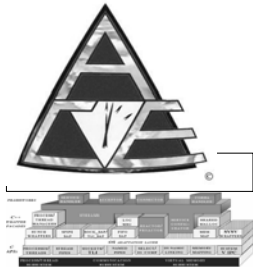


www.cs.wustl.edu/~schmidt/ACE.html

ACE provides a rich set of Wrapper Facades that abstract from more than 40 operating systems:

- Sockets
- Threads
- Processes
- Messages
- Synchronization
- ...

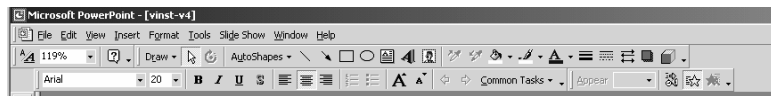
Wrapper Facade: Known Uses



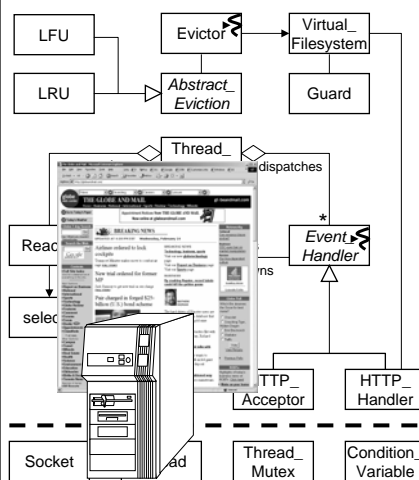
Class libraries and frameworks: Rouge Wave Net .h++, Rouge Wave Threads .h++, ACE, to abstract from operating system APIs.

.NET and Java Virtual Machines to abstract from the underlying OS.

Graphical User Interfaces Libraries: Win Forms, Smalltalk, MacApp, InterViews, Unidraw, NIHCL, ET++, MFC, Swing to abstract from low-level GUI libraries.



Wrap-Up



Patterns helped us realizing a highly efficient yet flexible web server:

- *Reactor* defines the fundamental event handling architecture of the web server.
- *Acceptor-Connector* improves availability by separating connection establishment from HTTP processing
- *Leader/Followers* and *Monitor Object* provide a high-performance concurrency architecture.
- *Caching* and an *Evictor* with associated eviction *Strategies* speed up accessing frequently used URLs.
- *Singleton* ensures the uniqueness of the server's virtual file system.
- *Scoped Locking*, *Strategized Locking*, and *Double-Checked Locking* ensure that the virtual file system is thread-safe without incurring locking overhead.
- *Wrapper Facade* ensures portability.

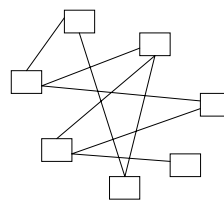


Further Refinements

Abstractions (I)

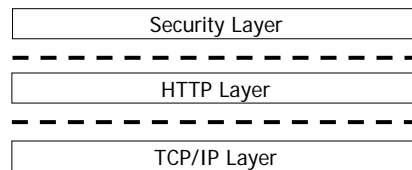
When developing a web server different layers of abstractions are available. For example:

- Handling network protocols from the physical medium up to its handling in the web server.
- The functionality in the web server covers different concerns such as communication, media handling, file system access, security issues,
- If all concerns are interwoven it is difficult to change, maintain, or extend the web server because no one knows what implications a change might have.
- Putting everything in components is not sufficient



This was straightforward to build, but how can I now change it?

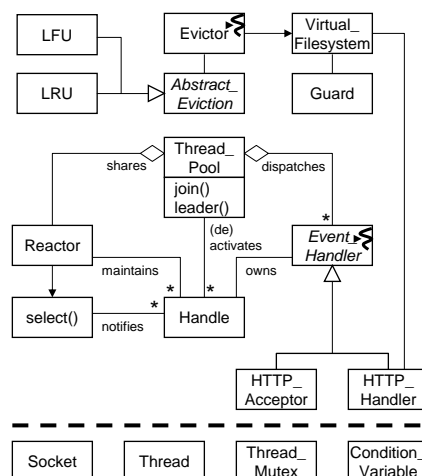
Abstractions (II)



Structure the web server implementation into different abstraction using Layers:

- Layers of abstraction such as the TCP/IP layer are separated into different layers of implementation.
- This way, you might even change the protocol stack implementation without impact on other system parts.
- The Wrapper Facades introduced in previous slides would also build one or more layers of abstraction

Abstractions (II)



Shield the web server implementation from low-level OS-APIs via a set of Wrapper Facades:

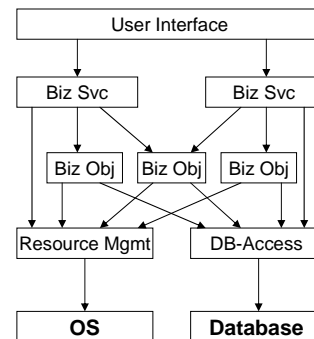
- A set of wrapper facades (Socket, Thread, Thread_Mutex, Condition_Variable) provide a portable, robust, and modular access to low-level OS APIs.
- Application classes (our web server classes) use the wrapper facades in their implementation.

Layers: Problem

Layers (POSA1)

Problem: Some applications consist of services that reside at different abstraction levels, with the higher-level services building upon the lower-level services. However:

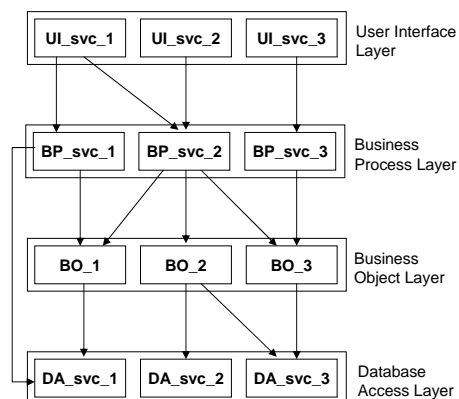
- Despite their cooperation relationships and dependencies, services of different abstraction levels should be decoupled as much as possible.
- Yet not all services can exist in isolation. Instead they strongly complement one another.
- Services at a particular level of abstraction may also evolve but changes should not ripple through the entire application and all its abstraction levels.



Layers: Structure

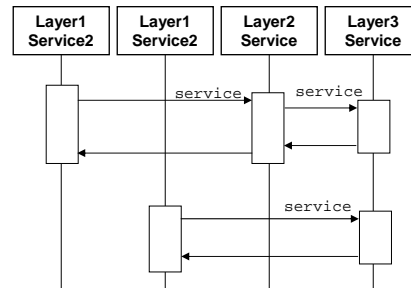
Solution Structure: Decouple the abstraction levels of an application by splitting it into several loosely connected layers—one layer for each abstraction level.

- *Services* realize the application's functionality.
- *Layers* comprise all services of a particular abstraction level.



Layers: Dynamics

Solution Dynamics: Let services of a particular layer call only services of lower-level layers.



Layers: Consequences



Benefits

- Loose coupling: Layers are decoupled from one another as much as possible.
- Exchangeability: Layer implementations can be exchanged transparently for other layers.
- Reusability: Layers and their services are units of reuse.

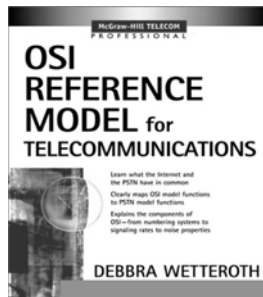


Liabilities

- Lower efficiency: The more layers there are, and the more decoupled these layers are, the less efficient is the system!
- Layer granularity: Finding the right layer granularity is often non-trivial.

Layers: Example

OSI 7-Layer Model: The International Standardization Organization (ISO) defined a 7-layer architecture for networking protocols.



Application

Layer 7: Provides miscellaneous protocols for common activities

Presentation

Layer 6: Structures information and attaches semantics

Session

Layer 5: Provides dialog control and synchronization

Transport

Layer 4: Breaks messages into packets and guarantees delivery

Network

Layer 3: Selects a route from sender to receiver

Data Link

Layer 2: Detects and corrects errors in bit sequences

Physical

Layer 1: Transmits bits

Layers: Known Uses

OSI 7-Layer Model for networking protocols.

N-Tier applications follow a layered architecture where layers are distributed across multiple network nodes.

Windows NT: NT defines five layers: system services, resource management, kernel, HAL (Hardware Abstraction Layer), Hardware

Presentation Tier

DOC-Middleware

Business Tier

Bizz
Comp1

Bizz
Comp2

Bizz
Comp3

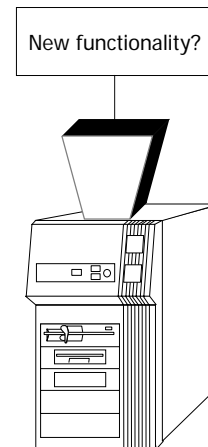
DOC-Middleware

Database Tier

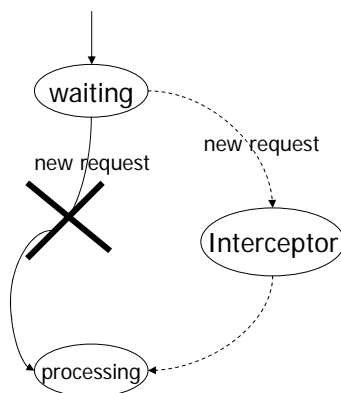
Out-of-the-band Functionality (I)

When developing a web server we cannot anticipate what kind of functionality might required:

- People might want to add additional media formats.
- Other network protocols such as SOAP might be required.
- Plug-Ins should be supported.
- The processing flow should be be modifyable.
- One way would be to let the development team of the web-server add those things exclusively.
- In this scenario, we end up with a whole ball of mud since the possible extensions would flood the system.
- Otherwise, we could stay with minimal extensions, but then people would very likely switch to other products.



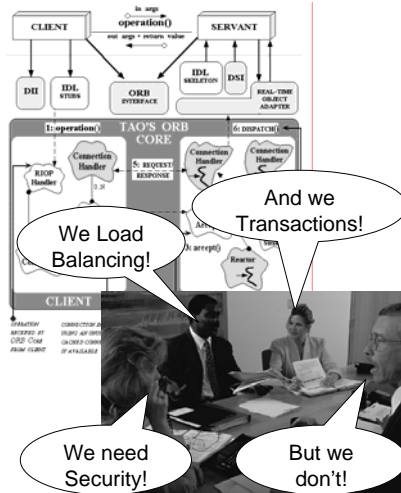
Out-of-the-Band Functionality (II)



Open the web server implementation according to the Open-Close principle using Interceptors:

- Define hooks in the processing state machine where 3rd parties might add code.
- Registered interceptors get automatically by the web server when specific events occur.
- Users might add their interceptors to midify and extend the processing logic for adding new media formats, protocols, plug-ins, security features, ...
- Interceptors might be combined by Pipes & Filters or Chain of Responsibility.

Interceptor: Problem



Interceptor (POSA2)

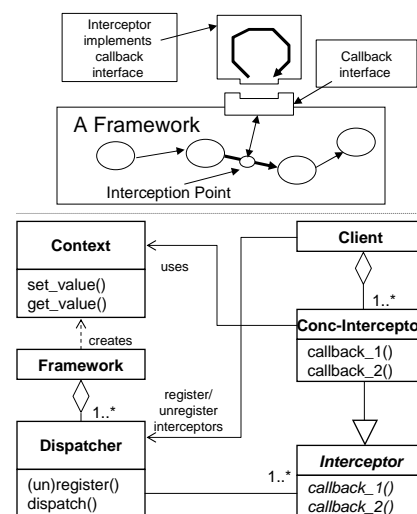
Problem: Some client applications may require more functionality from an application than it originally includes or is intended to include. However:

- It is impossible to anticipate specific requirements of applications a priori, specifically for application frameworks and product families.
- The implementation of a customer-specific system version should not pollute the code of other system versions that not need its specific services.

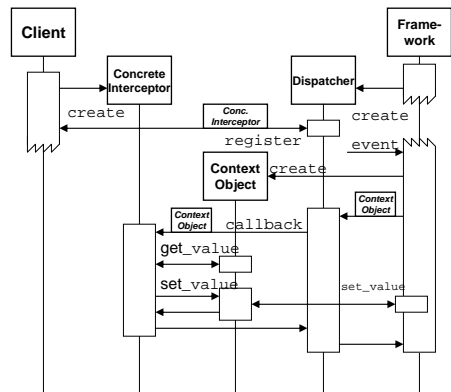
Interceptor: Structure

Solution Structure: Open an implementation so that external services can react on the occurrence of internal events:

- A *Framework* provides application functionality and specifies internal events on which occurrence add-on functionality can be triggered.
- An *Interceptor* declares a general interface for handling these events.
- *Concrete Interceptors* implement the interceptor interface in a client-specific way.
- *Dispatchers* allow to register interceptors with the framework and notify them when events occur.
- *Context Objects* provide interceptors with information about an event occurred.



Interceptor: Dynamics



Solution Dynamics: Call back all registered interceptors whenever a framework-internal event of interest occurs:

- A *Client* creates a *Concrete Interceptor* and registers it with a *Dispatcher*.
- In the *Framework*, an event occurs that is passed via the *Dispatcher* to the *Concrete Interceptor*, along with a *Context Object* containing information about the event.
- The *Concrete Interceptor* uses the information it receives from the *Context Object* to execute its functionality, if necessary by modifying the *Framework*'s state through the *Context Object*'s accessor methods.

Interceptor: Consequences



Benefits

- Extensibility and Flexibility: A design can be prepared for unexpected extensions without destabilizing it.
- Separation of concerns: Core functionality is separated from out-of-band functionality.
- Control: Ideal support for debugging, monitoring, tracing, logging etc.
- Reusability: Lean functionality is better reusable than fat functionality.

Liabilities

- Complexity: High realization effort.
- Stability: If interceptors break, the framework can break too; malicious interceptors.
- Limited power in most languages: OO is not the most ideal programming paradigm.

Interceptor: Example

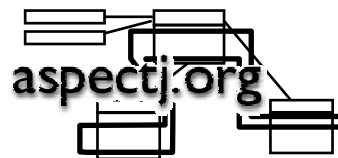
.NET Remoting specifies Interceptors that are designed to intercept the flow of a request/reply sequence through the runtime at specific points so that services can query the request information and manipulate the service contexts which are propagated between clients and servers.

Interceptor: Known Uses

.NET Remoting, CORB specify portable interceptors to intercept the ORB communication between clients and servers.

Reflective Systems: Interceptors intercept important events occurring in the base-level, inspect state information from the base-level, and influence its subsequent behavior.

Aspect weavers weave aspects into code that (conceptually) intercept the system's control flow at certain join-points to execute additional behavior.



```
aspect PublicTracing {
    Log log = new Log();

    pointcut publicMethodCalls():
        calls(public * com.expressions.*(..));

    before(): publicMethodCalls() {
        Trace.traceEntry
            (thisJoinPointStaticPart.getSignature().toString());
    }

    after(): publicMethodCalls() {
        Trace.traceExit
            (thisJoinPointStaticPart.getSignature().toString());
    }
}
```

Joinpoint
Declaration

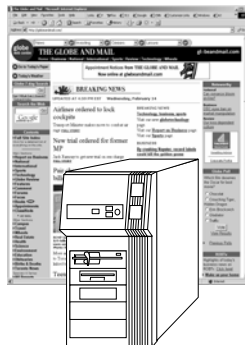
Advice refers
to joinpoint

Runtime Configuration (I)

A web server implements different strategies which might be exchangeable such as the concurrency or dispatching models used:

- It should be possible to exchange strategies at runtime.
- It would also be advantageous to add functionality dynamically.
- Seldomly used functionality could be dynamically added or removed.
- However, these changes should have no impact on the runtime execution.

Runtime Configuration (II)



Add new functionality by dynamically reconfiguring the web server using the Component Configurator:

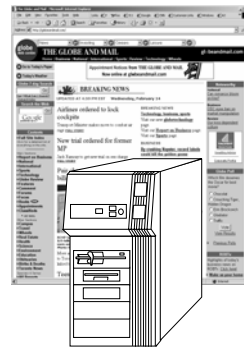
- Exchangeable components provide a specific interface and can be dynamically added/removed.
- The lifecycle of these components is controlled by a component configurator within the web server.
- Now, we can easily change strategies if we implement them by components.

Component Configurator: Problem

Component Configurator (POSA2)

Problem: It may be necessary to re-configure a system with a different set of service components than done in its original configuration. However:

- The re-configuration must happen at run-time.
- Services must be initialized, suspended, resumed, and terminated dynamically.



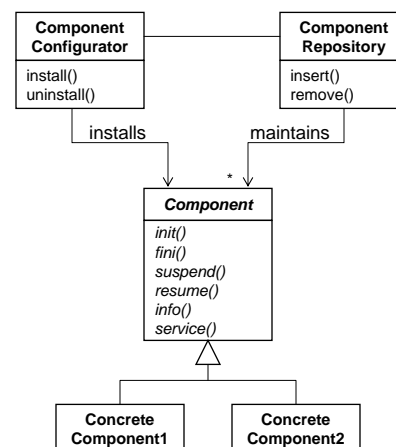
Different end user needs and traffic workloads require configurable web server architectures regarding many aspects:

- Concurrency models: thread per request, thread pool, ...
- Event demultiplexing models: synchronous or asynchronous
- File caching models: LRU, LFU, ...
- Content delivery protocols: HTTP/1.0, HTTP/1.1, HTTP-NG

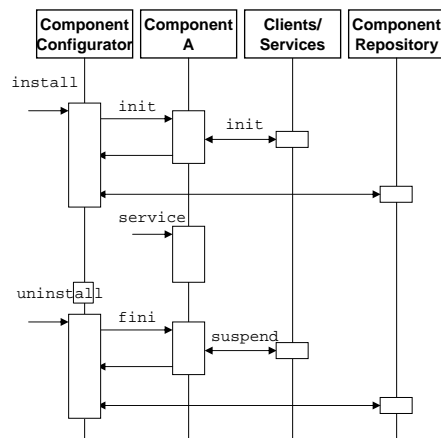
Component Configurator: Structure

Solution Structure: Separate the configuration of components from the point in time they are used.

- A *Component* defines a common interface for all configurable components.
- *Concrete Components* implement specific configurable components.
- A *Component Repository* maintains the currently configured and loaded components.
- A *Component Configurator* can dynamically load components to, and unload from, the component repository, and execute them.



Component Configurator: Dynamics



Solution Dynamics: Let the Component Configurator control the (re-) configuration of an application:

- The *Component Configurator* loads all *concrete components* of a particular configuration, initializes them and inserts them into the *Component Repository*.
- The *Concrete Components* serve client requests.
- When re-configuring an application, the *Component Configurator* terminates all *Concrete Components* and removes them from the *Component Repository*.

Component Configurator: Consequences



Benefits

- Centralized administration: All components are managed centrally.
- Increased configuration dynamism: Components can be (re-) configured without shutting a system down and at almost any point in time during system execution.
- Availability: During (re-) configuration an application is still partially available.



Liabilities

- Lack of determinism: Configuration is done at run-time.
- Structural and behavioral overhead: A Component Configurator arrangement results in a more complex system and component architecture and in more complex behavior.

[illegible]

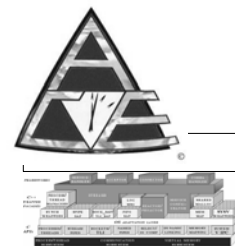
JAWS uses the Component Configurator pattern to dynamically optimize, control, and reconfigure the behavior of its Web server strategies at installation-time or during run-time. For example, JAWS applies the Component Configurator pattern to configure its various Cached Virtual Filesystem strategies, such as least-recently used (LRU) or least-frequently used (LFU).

```

classDiagram
    class ACE_Event_Handler
    class ACE_Service_Config
    class ACE_Service_Object
    class ACE_Service_Repository
    class ACE_Service_Repository_Iterator
    class Application_Service

    ACE_Event_Handler <|-- ACE_Service_Object
    ACE_Service_Object <|-- Application_Service
    ACE_Service_Repository <|-- ACE_Service_Repository_Iterator
    ACE_Service_Repository *-- ACE_Service_Object
  
```

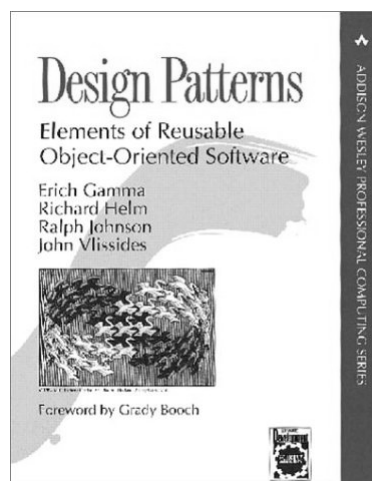
ACE Class	Description
ACE_Service_Object	Defines a uniform interface that the ACE Service Configuration framework uses to configure and control a service implementation. Concrete operations include initializing, negotiating, running, and terminating a service.
ACE_Service_Repository	A central repository for all services managed using the ACE Service Configuration framework. It provides methods for locating, reporting on, and controlling all of an application's configured services.
ACE_Service_Repository_Iterator	A portable mechanism for iterating through the members in a repository.
ACE_Service_Config	Provides an interface for parsing and editing source scripts, specifying which services to configure into an application, by loading and unloading DLLs and which services to





Annotated References

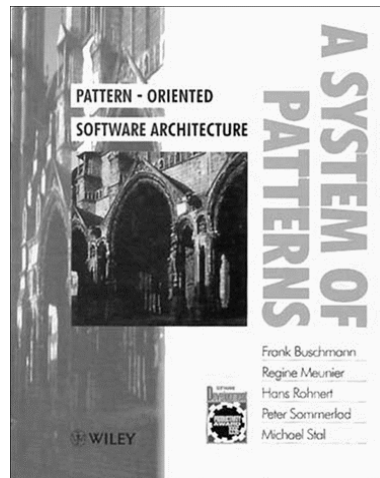
The Gang of Four Book



The Gang Of Four Book is the first, and still the most popular pattern book. It contains 23 general purpose design patterns and idioms for

- *Object creation*: Abstract Factory, Builder, Factory Method, Prototype, and Singleton
- *Structural Decomposition*: Composite and Interpreter
- *Organization of Work*: Command, Mediator, and Chain of Responsibility
- *Service Access*: Proxy, Facade, and Iterator
- *Extensibility*: Decorator and Visitor
- *Variation*: Bridge, Strategy, State, and Template Method
- *Adaptation*: Adapter
- *Resource Management*: Memento and Flyweight
- *Communication*: Observer

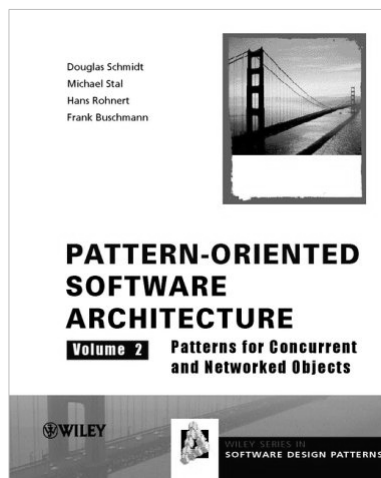
A System Of Patterns



A System Of Patterns is the first volume of the POSA series and the second most popular pattern book. It contains 17 general purpose architectural patterns, design patterns and idioms for:

- *Structural Decomposition*: Layers, Blackboard Pipes and Filters, and Whole Part
- *Distributed Systems*: Broker, Forwarder-Receiver and Client-Dispatcher-Server
- *Interactive Systems*: Model-View-Controller and Presentation-Abstraction-Control
- *Adaptive Systems*: Microkernel, Reflection
- *Organization of Work*: Master Slave
- *Service Access*: Proxy
- *Resource Management*: Counted Pointer, Command Processor, View Handler
- *Communication*: Publisher-Subscriber

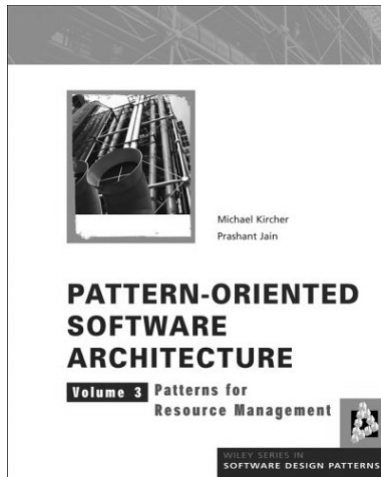
Patterns For Concurrent and Networked Objects



Patterns For Concurrent And Networked Objects is the second volume of the POSA series. It contains 17 architectural patterns, design patterns and idioms for concurrent, and networked systems:

- *Service Access and Configuration*: Wrapper Facade, Component Configurator, Interceptor and Extension Interface
- *Event Handling*: Reactor, Proactor, Asynchronous Completion Token and Acceptor-Connector
- *Synchronization*: Scoped Locking, Double-Checked Locking, Strategized Locking, and Thread-Safe Interface
- *Concurrency*: Active Object, Monitor Object, Leader/Followers, Thread-Specific Storage Half-Sync/Half-Async

Patterns for Resource Management



Patterns For Resource Management covers patterns helpful for resource acquisition and management. It is structured as a pattern language and includes patterns such as

- Evictor
- Lease
- Lazy Evaluation
- Caching
- Lookup
- ...

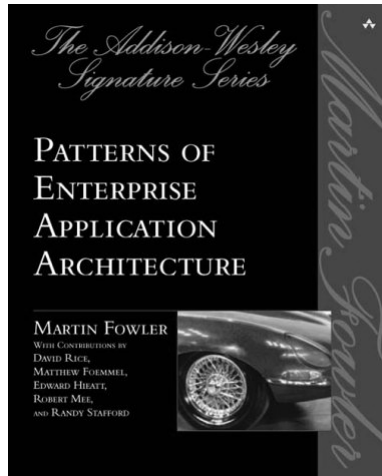
Server Component Patterns



Server Component Patterns includes a pattern language that illustrates core design concepts for containers as well as fundamental design criteria for components.

- For each pattern is outlined how it is implemented in EJB, CCM, and COM+
- A separate part in the book describes the EJB implementation in full depth.

Enterprise Architecture Patterns



Patterns of Enterprise Architecture includes a pattern language with about 50 patterns that illustrates how to design 3-tier enterprise business information applications.

A core strength of the book is its fine collection of patterns to map from an object-oriented application to a relational database.

Bemerkungen