Overview of Patterns

- Present solutions to common software problems arising within a certain context

- Help resolve key software design forces
  - Flexibility
  - Extensibility
  - Dependability
  - Predictability
  - Scalability
  - Efficiency

- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs

- Generally codify expert knowledge of design strategies, constraints & “best practices”

The Proxy Pattern

```
Client
  \  /  \
/   \   /
|     |   |  Proxy
|     |   |  Service
\  /  \
|   |   |  service
|   |   |  service
```

AbstractService

```
Client
  \  /  \
/   \   /
|     |   |  Proxy
|     |   |  Service
\  /  \
|   |   |  service
|   |   |  service
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
```

```
abstract
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Idioms</strong></td>
<td>Restricted to a particular language, system, or tool</td>
<td>Scoped locking</td>
</tr>
<tr>
<td><strong>Design patterns</strong></td>
<td>Capture the static &amp; dynamic roles &amp; relationships in solutions that occur repeatedly</td>
<td>Active Object, Bridge, Proxy, Wrapper Façade, &amp; Visitor</td>
</tr>
<tr>
<td><strong>Architectural patterns</strong></td>
<td>Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, &amp; include the rules and guidelines for organizing the relationships between them</td>
<td>Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, &amp; Reactor</td>
</tr>
<tr>
<td><strong>Optimization principle patterns</strong></td>
<td>Document rules for avoiding common design &amp; implementation mistakes that degrade performance</td>
<td>Optimize for common case, pass information between layers</td>
</tr>
</tbody>
</table>
Benefits of Patterns

- Enables reuse of software architectures & designs
- Improves development team communication
- Convey “best practices” intuitively
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

www.cs.wustl.edu/~schmidt/patterns.html
Limitations of Patterns

- Require significant tedious & error-prone human effort to handcraft pattern implementations
- Can be deceptively simple
- Leaves some important details unresolved

www.cs.wustl.edu/~schmidt/patterns.html
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Idioms</strong></td>
<td>Restricted to a particular language, system, or tool</td>
<td>Scoped locking</td>
</tr>
<tr>
<td><strong>Design patterns</strong></td>
<td>Capture the static &amp; dynamic roles &amp; relationships in solutions that occur repeatedly</td>
<td>Active Object, Bridge, Proxy, Wrapper Façade, &amp; Visitor</td>
</tr>
<tr>
<td><strong>Architectural patterns</strong></td>
<td>Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, &amp; include the rules and guidelines for organizing the relationships between them</td>
<td>Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, &amp; Reactor</td>
</tr>
<tr>
<td><strong>Optimization principle patterns</strong></td>
<td>Document rules for avoiding common design &amp; implementation mistakes that degrade performance</td>
<td>Optimize for common case, pass information between layers</td>
</tr>
</tbody>
</table>
Legacy Avionics Architectures

Key System Characteristics
• Hard & soft real-time deadlines
  • ~20-40 Hz
• Low latency & jitter between boards
  • ~100 usecs
• Periodic & aperiodic processing
• Complex dependencies
• Continuous platform upgrades

Avionics Mission Computing Functions
• Weapons targeting systems (WTS)
• Airframe & navigation (Nav)
• Sensor control (GPS, IFF, FLIR)
• Heads-up display (HUD)
• Auto-pilot (AP)

4: Mission functions perform avionics operations
3: Sensor proxies process data & pass to missions functions
2: I/O via interrupts
1: Sensors generate data
Legacy Avionics Architectures

Key System Characteristics
- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 usecs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

Limitations with Legacy Avionics Architectures
- Stovepiped
- Proprietary
- Expensive
- Vulnerable
  - **Tightly coupled**
  - **Hard to schedule**
  - **Brittle & non-adaptive**
Decoupling Avionics Components

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• I/O driven DRE application</td>
<td>• Tightly coupled components</td>
<td>• Apply the Publisher-Subscriber architectural pattern to distribute</td>
</tr>
<tr>
<td>• Complex dependencies</td>
<td>• Hard to schedule</td>
<td>periodic, I/O-driven data from a single point of source to a collection</td>
</tr>
<tr>
<td>• Real-time constraints</td>
<td>• Expensive to evolve</td>
<td>of consumers</td>
</tr>
</tbody>
</table>

**Structure**

- Publisher
  - produce
- Event Channel
  - attachPublisher
  - detachPublisher
  - attachSubscriber
  - detachSubscriber
  - pushEvent
- Subscriber
  - consume

**Dynamics**

- :Publisher
  - produce
  - pushEvent
  - event
- :Event Channel
  - attachSubscriber
  - pushEvent
  - event
  - detachSubscriber
- :Subscriber
  - consume
- :Event
  - create
  - receive
Applying the Publisher-Subscriber Pattern to Bold Stroke

Bold Stroke uses the **Publisher-Subscriber** pattern to decouple sensor processing from mission computing operations:

- Anonymous publisher & subscriber relationships
- Group communication
- Asynchrony

Considerations for implementing the **Publisher-Subscriber** pattern for mission computing applications include:

- **Event notification model**
  - Push control vs. pull data interactions

- **Scheduling & synchronization strategies**
  - e.g., priority-based dispatching & preemption

- **Event dependency management**
  - e.g., filtering & correlation mechanisms

Diagram:

- **1: Sensors generate data**
- **2: I/O via interrupts**
- **3: Sensor publishers push events to event channel**
- **4: Event Channel pushes events to subscribers(s)**
- **5: Subscribers perform avionics operations**
Pros & Cons of Pub/Sub Pattern

This pattern provides the following **benefits**: 

- **Separation of concerns**  
  - This pattern decouples application-independent dissemination from application-specific functionality

- **Flexibility on data dissemination**  
  - The Pub/Sub pattern supports aggregating, filtering, and prioritizing of data

- **Scalability**  
  - Since senders and receivers are decoupled, applications can scale in the number of receivers and senders

This pattern also incur **liabilities**: 

- **Complexity of debugging & testing**  
  - Applications written with this pattern can be hard to debug due its transparency

- **Added overhead**  
  - A pub/sub architecture can increase overhead of system management and data delivery
Ensuring Platform-neutral & Network-transparent Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Mission computing requires remote IPC | Applications need capabilities to:  
• Support remote communication  
• Provide location transparency  
• Handle faults  
• Manage end-to-end QoS  
• Encapsulate low-level system details | Apply the Broker architectural pattern to provide platform-neutral communication between mission computing boards |

**Dynamics**

- **Client**
  - operation (params)
  - marshal
  - receive_reply
  - result

- **Client Proxy**
  - connect
  - send_request

- **Broker**
  - register_service
  - dispatch
  - unmarshal
  - result

- **Object Adapter**
  - assigned port

- **Server**
  - start_up
Pros & Cons of Broker Pattern

This pattern provides the following **benefits:**

- **Separation of concerns**
  - This pattern decouples application-independent object location & dispatching mechanisms from application-specific functionality

- **Application programming simplicity**
  - The Broker pattern simplifies the programming of business logic for the application

- **Reuse**
  - Since it’s application independent the implementation can be reused in various application domains or subsystems of the same application

This pattern also incur **liabilities:**

- **Complexity of debugging & testing**
  - Applications written with this pattern can be hard to debug due its indirection and transparency

- **Added level of indirection**
  - A brokered architecture can be less efficient than a monolithic architecture
Separating Concerns Between Tiers

**Context**
- Distributed systems are now common due to the advent of
  - The global Internet
  - Ubiquitous mobile & embedded devices

**Problem**
- It’s hard to build distributed systems due to the complexity associated with many capabilities at many levels of abstraction

**Solution**
- Apply the *Layers* pattern (P1) to create a multi-tier architecture that separates concerns between groups of tasks occurring at distinct layers in the distributed system

**Database Tier**
- e.g., persistent data

**Middle Tier**
- e.g., common business logic

**Presentation Tier**
- e.g., thin client displays

Services in the *middle tier* participate in various types of tasks, e.g.,
- Workflow of integrated “business” processes
- Connect to databases & other backend systems for data storage & access
Applying the Layers Pattern to Image Acquisition

Presentation Tier
- e.g., radiology clients

Middle Tier
- e.g., image routing, security, & image transfer logic

Database Tier
- e.g., persistent image data

Diagnostic & clinical workstations are presentation tier entities that:
- Typically represent sophisticated GUI elements
- Share the same address space with their clients
- Their clients are containers that provide all the resources
- Exchange messages with the middle tier components

Image servers are middle tier entities that:
- Provide server-side functionality
  - e.g., they are responsible for scalable concurrency & networking
- Can run in their own address space
- Are integrated into containers that hide low-level OS platform details
## Pros & Cons of the Layers Pattern

This pattern has four **benefits**:  

- **Reuse of layers**  
  - If an individual layer embodies a well-defined abstraction & has a well-defined & documented interface, the layer can be reused in multiple contexts

- **Support for standardization**  
  - Clearly-defined & commonly-accepted levels of abstraction enable the development of standardized tasks & interfaces

- **Dependencies are localized**  
  - Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed

- **Exchangeability**  
  - Individual layer implementations can be replaced by semantically-equivalent implementations without undue effort

This pattern also has **liabilities**:  

- **Cascades of changing behavior**  
  - If layer interfaces & semantics aren’t abstracted properly then changes can ripple when behavior of a layer is modified

- **Higher overhead**  
  - A layered architecture can be less efficient than a monolithic architecture

- **Unnecessary work**  
  - If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, performance can suffer

- **Difficulty of establishing the correct granularity of layers**  
  - It’s important to avoid too many & too few layers
Scaling Up Performance via Threading

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 & 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

• 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

• 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
The Half-Sync/Half-Async Pattern

Solution

• Apply the Half-Sync/Half-Async architectural pattern (P2) 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.

This solution yields two benefits:
1. Threads can be mapped to separate CPUs to scale up server performance via multi-processing.
2. Each thread blocks independently, which prevents a flow-controlled connection from degrading the QoS that other clients receive.
• This pattern defines two service processing layers—one async & one sync—along with a queueing layer that allows services to exchange messages between the two layers

• The pattern allows sync services, such as HTTP protocol processing, to run concurrently, relative both to each other & to async services, such as event demultiplexing
Pros & Cons of Half-Sync/Half-Async Pattern

This pattern has three **benefits**:

- **Simplification & performance**
  - The programming of higher-level synchronous processing services are simplified without degrading the performance of lower-level system services

- **Separation of concerns**
  - Synchronization policies in each layer are decoupled so that each layer need not use the same concurrency control strategies

- **Centralization of inter-layer communication**
  - Inter-layer communication is centralized at a single access point, because all interaction is mediated by the queueing layer

This pattern also incurs **liabilities**:

- **A boundary-crossing penalty may be incurred**
  - This overhead arises from context switching, synchronization, & data copying overhead when data is transferred between the sync & async service layers via the queueing layer

- **Higher-level application services may not benefit from the efficiency of async I/O**
  - Depending on the design of operating system or application framework interfaces, it may not be possible for higher-level services to use low-level async I/O devices effectively

- **Complexity of debugging & testing**
  - Applications written with this pattern can be hard to debug due its concurrent execution
Drawbacks with Half-Sync/Half-Async

Problem

• Although Half-Sync/Half-Async threading model is more scalable than the purely reactive model, it is not necessarily the most efficient design.

• e.g., passing a request between the Reactor thread & a worker thread incurs:
  • Dynamic memory (de)allocation,
  • Synchronization operations,
  • A context switch, &
  • CPU cache updates

• This overhead makes JAWS’ latency unnecessarily high, particularly on operating systems that support the concurrent accept() optimization.

Solution

• Apply the Leader/Followers architectural pattern (P2) to minimize server threading overhead.
The Leader/Followers architectural pattern (P2) provides an efficient concurrency model where multiple threads take turns sharing event sources to detect, demux, dispatch, & process service requests that occur on the event sources.

This pattern eliminates the need for—and the overhead of—a separate Reactor thread & synchronized request queue used in the Half-Sync/Half-Async pattern.
Leader/Followers Pattern Dynamics

1. Leader thread demuxing
   - Thread 1
   - Thread 2
   - Thread Pool
   - Handle Set
   - Concrete Event Handler

   - Thread 1 joins
   - Thread 2 sleeps until it becomes the leader
   - Thread 2 waits for a new event, thread 1 processes current event
   - Thread 1 sleeps until it becomes the leader

2. Follower thread promotion
   - Thread 1
   - Thread 2
   - Handle
   - Event
   - Event Handler

   - Thread 2 promotes new leader
   - Handle events

3. Event handler demuxing & event processing
   - Thread 1
   - Thread 2
   - Handle
   - Event
   - Event Handler

   - Thread 2 waits for a new event, thread 1 processes current event
   - Thread 1 joins

4. Rejoining the thread pool
   - Thread 1
   - Thread 2
   - Handle
   - Event
   - Event Handler

   - Thread 1 promotes new leader
   - Handle events
   - Thread 2 deactivates handle

   - Thread 2 waits for a new event, thread 1 processes current event
   - Thread 1 joins
Pros & Cons of Leader/Followers Pattern

This pattern provides two **benefits**:  
* **Performance enhancements**  
  - This can improve performance as follows:  
    - It enhances CPU cache affinity & eliminates the need for dynamic memory allocation & data buffer sharing between threads  
    - It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization  
    - It can minimize priority inversion because no extra queueing is introduced in the server  
    - It doesn’t require a context switch to handle each event, reducing dispatching latency  
  
  * **Programming simplicity**  
    - The Leader/Follower pattern simplifies the programming of concurrency models where multiple threads can receive requests, process responses, & demultiplex connections using a shared handle set

This pattern also incur **liabilities**:  
* **Implementation complexity**  
  - The advanced variants of the Leader/ Followers pattern are hard to implement  
* **Lack of flexibility**  
  - In the Leader/ Followers model it is hard to discard or reorder events because there is no explicit queue  
* **Network I/O bottlenecks**  
  - The Leader/Followers pattern serializes processing by allowing only a single thread at a time to wait on the handle set, which could become a bottleneck because only one thread at a time can demultiplex I/O events
Decoupling Event Demuxing, Connection Management, & Protocol Processing (1/2)

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()`

Problem

• 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 server protocol code directly & may yield subtle bugs, e.g., when porting to use TLI or `WaitForMultipleObjects()`
Solution

Apply the Reactor architectural pattern (P2) & the Acceptor-Connector design pattern (P2) to separate the generic event-demultiplexing & connection-management code from the web server’s protocol code.
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 Pattern Dynamics

Observations

- Note inversion of control
- Also note how long-running event handlers can degrade the QoS since callbacks steal the reactor’s thread!
Pros & Cons of the Reactor Pattern

This pattern offers four **benefits**:

- **Separation of concerns**
  - This pattern decouples application-independent demuxing & dispatching mechanisms from application-specific hook method functionality

- **Modularity, reusability, & configurability**
  - This pattern separates event-driven application functionality into several components, which enables the configuration of event handler components that are loosely integrated via a reactor

- **Portability**
  - By decoupling the reactor’s interface from the lower-level OS synchronous event demuxing functions used in its implementation, the Reactor pattern improves portability

- **Coarse-grained concurrency control**
  - This pattern serializes the invocation of event handlers at the level of event demuxing & dispatching within an application process or thread

This pattern can incur **liabilities**:

- **Restricted applicability**
  - This pattern can be applied efficiently only if the OS supports synchronous event demuxing on handle sets

- **Non-pre-emptive**
  - In a single-threaded application, concrete event handlers that borrow the thread of their reactor can run to completion & prevent the reactor from dispatching other event handlers

- **Complexity of debugging & testing**
  - It is hard to debug applications structured using this pattern due to its inverted flow of control, which oscillates between the framework infrastructure & the method callbacks on application-specific event handlers
Using Asynchronous I/O Effectively

**Context**
- Synchronous multi-threading may not be the most scalable way to implement a Web server on OS platforms that support async I/O more efficiently than synchronous multi-threading.
- For example, highly-efficient Web servers can be implemented on Windows NT by invoking async Win32 operations that perform the following activities:
  - Processing indication events, such as TCP CONNECT & HTTP GET requests, via `AcceptEx()` & `ReadFile()`, respectively.
  - Transmitting requested files to clients asynchronously via `WriteFile()` or `TransmitFile()`.
- When these async operations complete, WinNT delivers the associated completion events containing their results to the Web server.
  1. Delivers the associated completion events containing their results to the Web server.
  2. Processes these events & performs the appropriate actions before returning to its event loop.
The Proactor Pattern

Problem
• Developing software that achieves the potential efficiency & scalability of async I/O is hard due to the separation in time & space of async operation invocations & their subsequent completion events.

Solution
• Apply the Proactor architectural pattern (P2) to make efficient use of async I/O.

This pattern allows event-driven applications to efficiently demultiplex & dispatch service requests triggered by the completion of async operations, thereby achieving the performance benefits of concurrency without incurring its many liabilities.
Proactor Pattern Dynamics

1. Initiate operation
2. Process operation
3. Run event loop
4. Generate & queue completion event
5. Dequeue completion event & perform completion processing

Note similarities & differences with the Reactor pattern, e.g.:
- Both process events via callbacks
- However, it’s generally easier to multi-thread a proactor
Pros & Cons of Proactor Pattern

This pattern offers five benefits:

- **Separation of concerns**
  - Decouples application-independent async mechanisms from application-specific functionality

- **Portability**
  - Improves application portability by allowing its interfaces to be reused independently of the OS event demuxing calls

- **Decoupling of threading from concurrency**
  - The async operation processor executes long-duration operations on behalf of initiators so applications can spawn fewer threads

- **Performance**
  - Avoids context switching costs by activating only those logical threads of control that have events to process

- **Simplification of application synchronization**
  - If concrete completion handlers spawn no threads, application logic can be written with little or no concern for synchronization issues

This pattern incurs some liabilities:

- **Restricted applicability**
  - This pattern can be applied most efficiently if the OS supports asynchronous operations natively

- **Complexity of programming, debugging, & testing**
  - It is hard to program applications & higher-level system services using asynchrony mechanisms, due to the separation in time & space between operation invocation & completion

- **Scheduling, controlling, & canceling asynchronously running operations**
  - Initiators may be unable to control the scheduling order in which asynchronous operations are executed by an asynchronous operation processor
Architectural Patterns Resources

• Books

• Web sites

http://www.enterpriseintegrationpatterns.com/ - patterns for enterprise systems and integrations

http://www.cs.wustl.edu/~schmidt/POSA/ - patterns for distributed computing systems

http://www.hillside.net/patterns/ - a catalog of patterns and pattern languages

Layers Pattern Revisited

**Context**
- A large system that requires decomposition

**Problem**
- Managing a “sea of classes” that addresses various levels of abstraction

**Solution**
- Aggregate classes at the same level of abstraction into layers.
Applying the Layers Pattern to Image Acquisition

Presentation Tier
• e.g., radiology clients

Middle Tier
• e.g., image routing, security, & image transfer logic

Database Tier
• e.g., persistent image data

Image servers are middle tier entities that:
• Provide server-side functionality
  • e.g., they are responsible for scalable concurrency & networking
• Can run in their own address space
• Are integrated into containers that hide low-level OS platform details

Diagnostic & clinical workstations are presentation tier entities that:
• Typically represent sophisticated GUI elements
• Share the same address space with their clients
  • Their clients are containers that provide all the resources
• Exchange messages with the middle tier components
Model View Controller Revisited

**Context**
- Interactive applications with a flexible human-computer interface

**Problem**
- Managing different & changing presentations of the same data
- Updating the presentations when the data changes

**Solution**
- Decouple core data and functionality from output representations or input behavior
Applying the Layers & MVC Patterns to Image Acquisition

**Layer 1**
Database Tier
- e.g., persistent image data

**Layer 2**
Middle Tier
- e.g., image routing, security, & image transfer logic

**Layer 3**
Presentation Tier
- e.g., radiology clients

Model in MVC pattern
- Diagnostic Workstations
- Clinical Workstations

Views/Controllers in MVC pattern

- Image Servers
- Image Database
- Patient Database

- Views/Controllers in MVC pattern
- Model in MVC pattern
Patterns Are More Than Structure

Pattern A?

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Let the algorithm vary independently from clients that use it.

Pattern B?

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Patterns Are Abstract

The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn’t describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

- Design Patterns: Elements of Reusable Object-Oriented Software
## Taxonomy of Patterns & Idioms

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Idioms</strong></td>
<td>Restricted to a particular language, system, or tool</td>
<td>Scoped locking</td>
</tr>
<tr>
<td><strong>Design patterns</strong></td>
<td>Capture the static &amp; dynamic roles &amp; relationships in solutions that occur repeatedly</td>
<td>Active Object, Bridge, Proxy, Wrapper Façade, &amp; Visitor</td>
</tr>
<tr>
<td><strong>Architectural patterns</strong></td>
<td>Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, &amp; include the rules and guidelines for organizing the relationships between them</td>
<td>Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, &amp; Reactor</td>
</tr>
<tr>
<td><strong>Optimization principle patterns</strong></td>
<td>Document rules for avoiding common design &amp; implementation mistakes that degrade performance</td>
<td>Optimize for common case, pass information between layers</td>
</tr>
</tbody>
</table>
Seminal Design Patterns Book

Design Patterns: Elements of Reusable Object-Oriented Software
by Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides (“Gang of Four”)

Written in 1995

Documents 23 design patterns outlining:
  • Intent
  • Motivation
  • Applicability
  • Structure
  • Collaborations
  • Consequences
  • Implementation
  • Known uses
  • Related patterns

Patterns grouped as:
  • Creational,
  • Structural, or
  • Behavioral
Managing Global Objects Effectively

Goals:
- Centralize access to objects that should be visible globally, e.g.:
  - command-line options that parameterize the behavior of the program
  - The object (Reactor) that drives the main event loop

Constraints/forces:
- Only need one instance of the command-line options & Reactor
- Global variables are problematic in C++

% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit

% tree-traversal
> 1+4*3/2
7
Rather than using global variables, create a central access point to global instances, e.g.:

```c
int main (int argc, char *argv[]) {
    // Parse the command-line options.
    if (!Options::instance ()->parse_args (argc, argv))
        return 0;

    // Dynamically allocate the appropriate event handler
    // based on the command-line options.
    Expression_Tree_Event_Handler *tree_event_handler =
        Expression_Tree_Event_Handler::make_handler
            (Options::instance ()->verbose ());

    // Register event handler with the reactor.
    Reactor::instance ()->register_input_handler
        (tree_event_handler);
    // ...
```
Singleton Description (1/2)

Intent

ensure a class only ever has one instance & provide a global point of access

Applicability

– when there must be exactly one instance of a class, & it must be accessible from a well-known access point
– when the sole instance should be extensible by subclassing, & clients should be able to use an extended instance without modifying their code

Structure

```
If (uniqueInstance == 0)
    uniqueInstance = new Singleton;
    return uniqueInstance;
```

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>static instance()</td>
</tr>
<tr>
<td>singletonOperation()</td>
</tr>
<tr>
<td>getSingletonData()</td>
</tr>
<tr>
<td>static uniqueInstance</td>
</tr>
<tr>
<td>singletonData</td>
</tr>
</tbody>
</table>

Singleton Description (2/2)

Singleton

**Consequences**

+ reduces namespace pollution
+ makes it easy to change your mind & allow more than one instance
+ allow extension by subclassing
- same drawbacks of a global if misused
- implementation may be less efficient than a global
- concurrency pitfalls strategy creation & communication overhead

**Implementation**

– static instance operation
– registering the singleton instance
– deleting singletons

<table>
<thead>
<tr>
<th>Singleton</th>
<th>object creational</th>
</tr>
</thead>
</table>

**Known Uses**

– Unidraw's Unidraw object
– Smalltalk-80 ChangeSet, the set of changes to code
– InterViews Session object

**See Also**

– Double-Checked Locking Optimization pattern from POSA2
– “To Kill a Singleton”
  www.research.ibm.com/designpatterns/pubs/ph-jun96.txt