CS4227 SOFTWARE ARCHITECTURE & DESIGN
CS5722 SOFTWARE ARCHITECTURE

Lecture D: Interceptor Architectural Pattern

J.J. Collins
Dept of CSIS
University of Limerick

Reading

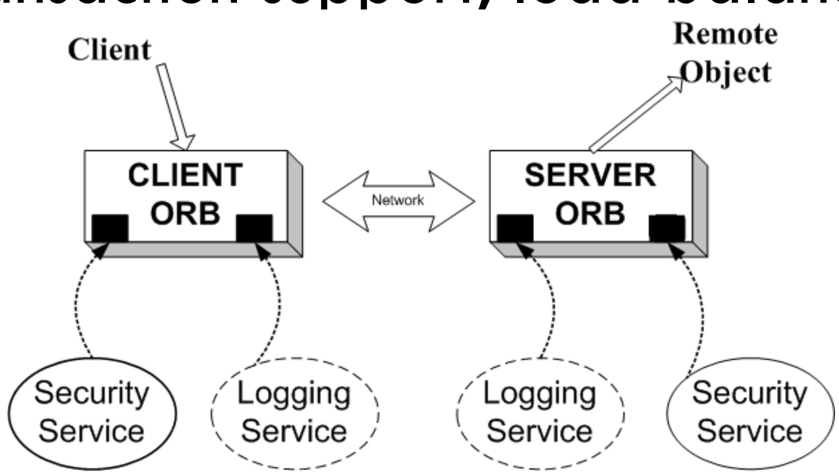
- **<u>This lecture is based on</u> excerpts from:**

Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture , Volume 2: Patterns for Concurrent and Networked Objects. 2000. Wiley.

  - Text referred to as POSA2

The Interceptor Architectural Pattern

- A software house developing an Object Request Broker (ORB) **middleware framework**, based on broker architectural framework.
- In addition to **marshalling and unmarshalling** service requests, middleware provides a basic set of services such as connection management and security.
- Applications using the framework may require **additional services** such transaction support, load balancing, trader, logging, etc.



Extending the Framework

<u>Options</u>

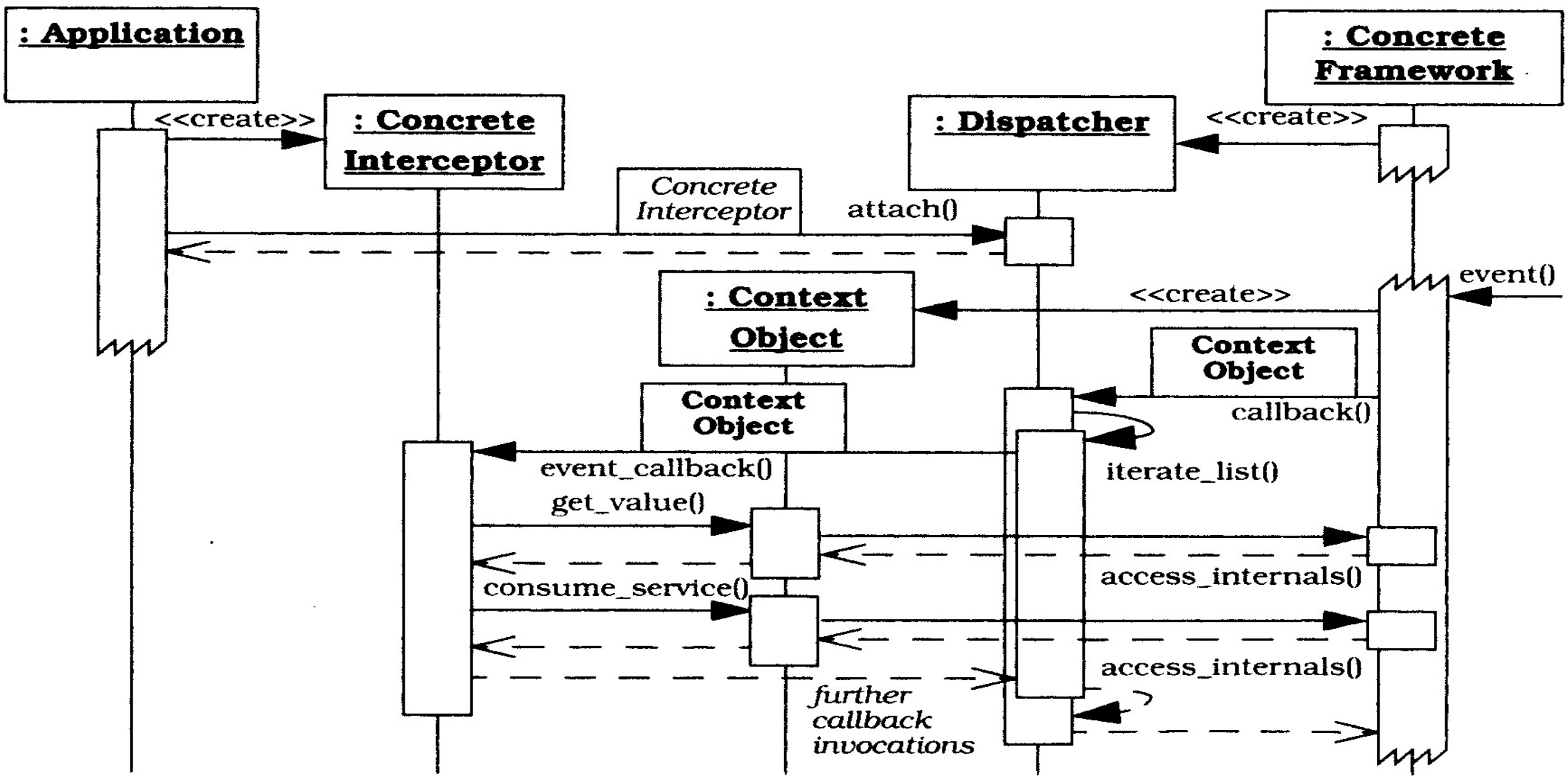
1. Framework developers **include all possible services**
   - **Problem → Infeasible to integrate all services at design time.**
2. **Client developers include these additional/out-of-band services in their application** that uses the framework API
   - **Problem ???**
3. **Alternatively, middleware framework is a kernel supporting core (basic) services that also facilitates transparent extension**
   - Through integration of services from 3rd parties.

The Interceptor Architectural Pattern

- **Context:** developing frameworks that can be extended transparently
- **Problem:**
  - A framework should allow for the integration of additional services without requiring a restructuring of the system architecture.
  - Should not impact existing services or applications that use the framework
  - Applications using a framework may need to monitor and control the behaviour of the framework platform
    - For example, some applications may use the Reflection pattern to monitor fault tolerance strategies, and modify these strategies using the Strategy pattern.

The Interceptor Architectural Pattern

Dynamics



The Interceptor Architectural Pattern

Solution

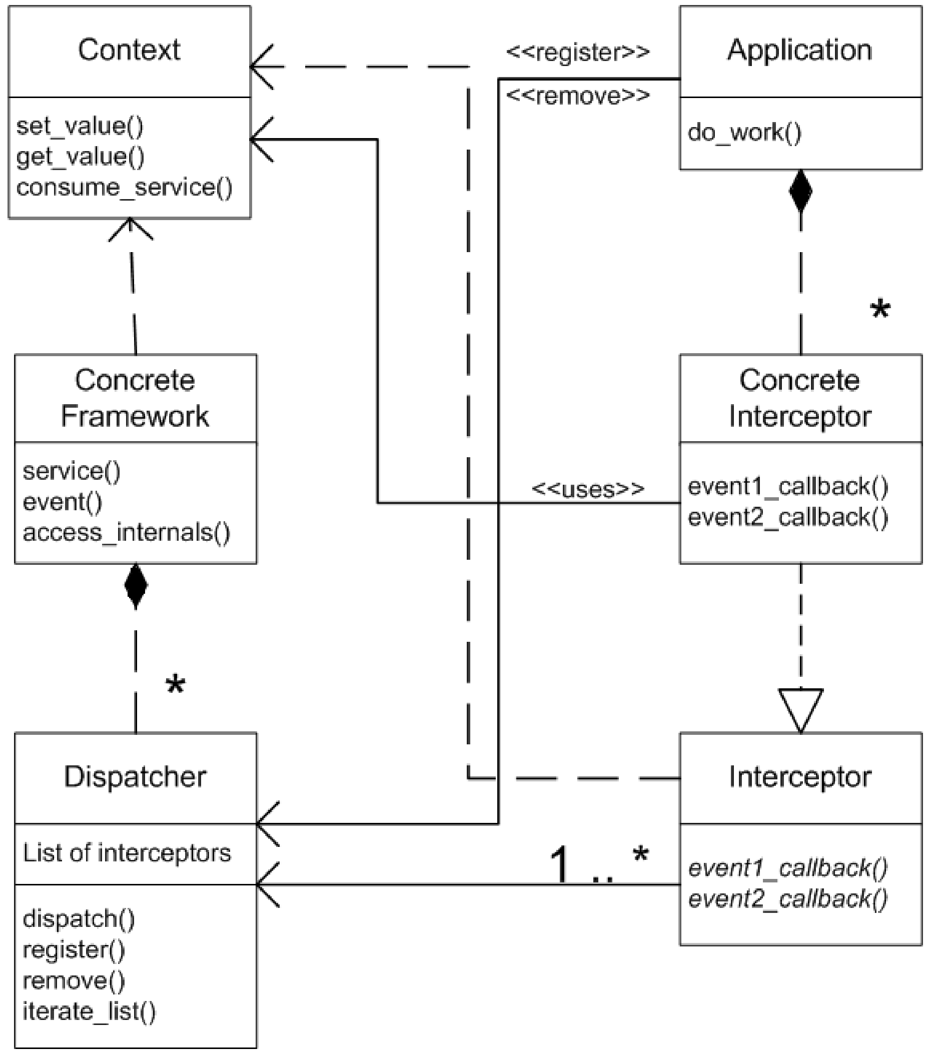
- "**Out-of-band**" services register with the framework by predefined interfaces.
- Framework triggers these out-of-band services when certain events fire.
- Open the framework's implementation so that **out-of-band services can access and control certain aspects of the framework's behaviour**.
- For a specified set of events that are processed by framework, define and expose an interceptor callback interface
- Applications can derive concrete interceptors from this interface to implement out-of-band services that process these said events.
- Provide a dispatcher that allows applications to register their concrete interceptors with the framework.

The Interceptor Architectural Pattern

- When an event occurs, the framework notifies the appropriate dispatcher to invoke the callback of the registered concrete interceptor.
- The Command design pattern (behaviour) is an object-oriented pattern for callbacks.
- Define context objects to allow a concrete interceptor to introspect and control certain aspects of the framework.
- Context objects provide methods to access and modify a framework's internal state.
- Can be passed to concrete interceptors when they are dispatched by the framework.

The Interceptor Architectural Pattern

STRUCTURE



The Interceptor Architectural Pattern

- **Class: Concrete Framework**
- **Collaborator:** Dispatcher
- **Responsibility:**
  - Defines application services.
  - Integrates Dispatchers that allow applications to intercept events.
  - Delegates events to associated dispatchers.

- **Class: Interceptor**
- **Collaborator:**
- **Responsibility:**
  - Defines an interface for out-of-band services.

The Interceptor Architectural Pattern

- **Class: Concrete Interceptor**
- **Collaborator:** Context Object
- **Responsibility:**
  - Implements a specific out-of-band service.
  - Uses context object to control the concrete framework.

- **Class: Dispatcher**
- **Collaborator:** Interceptor, Application
- **Responsibility**
  - Allows applications to register and remove concrete interceptors.
  - Dispatches registered concrete interceptor callbacks when events occur.

The Interceptor Architectural Pattern

- **Class: Context Object**
- **Collaborator:** Concrete Framework
- **Responsibility:**
  - Allows services to obtain state information from the concrete framework.
  - Allows services to control certain behaviours in the concrete framework.

- **Class: Application**
- **Collaborator:** Dispatcher, Concrete Interceptor
- **Responsibility:**
  - Runs on top of the concrete framework.
  - Instantiates concrete interceptors and registers them with dispatchers.

The Interceptor: Consequences

Benefits:

- **Extensibility & flexibility**
  - Interceptors allow an application to evolve without breaking existing APIs & components
- **Separation of concerns**
  - Interceptors decouple the "functional" path from the "meta" path
- **Support for monitoring & control of frameworks**
  - e.g., generic logging mechanisms can be used to unobtrusively track application behaviour
- **Layer symmetry**
  - Interceptors can perform transformations on a client-side whose inverse are performed on the server-side
- **Reusability**
  - Interceptors can be reused for various general-purpose behaviours

The Interceptor: Consequences

Liabilities:

- **Complex design issues**
  - Determining interceptor APIs & semantics is non-trivial
- **Malicious or erroneous interceptors**
- **Potential interception cascades**
  - Interceptors can result in infinite recursion

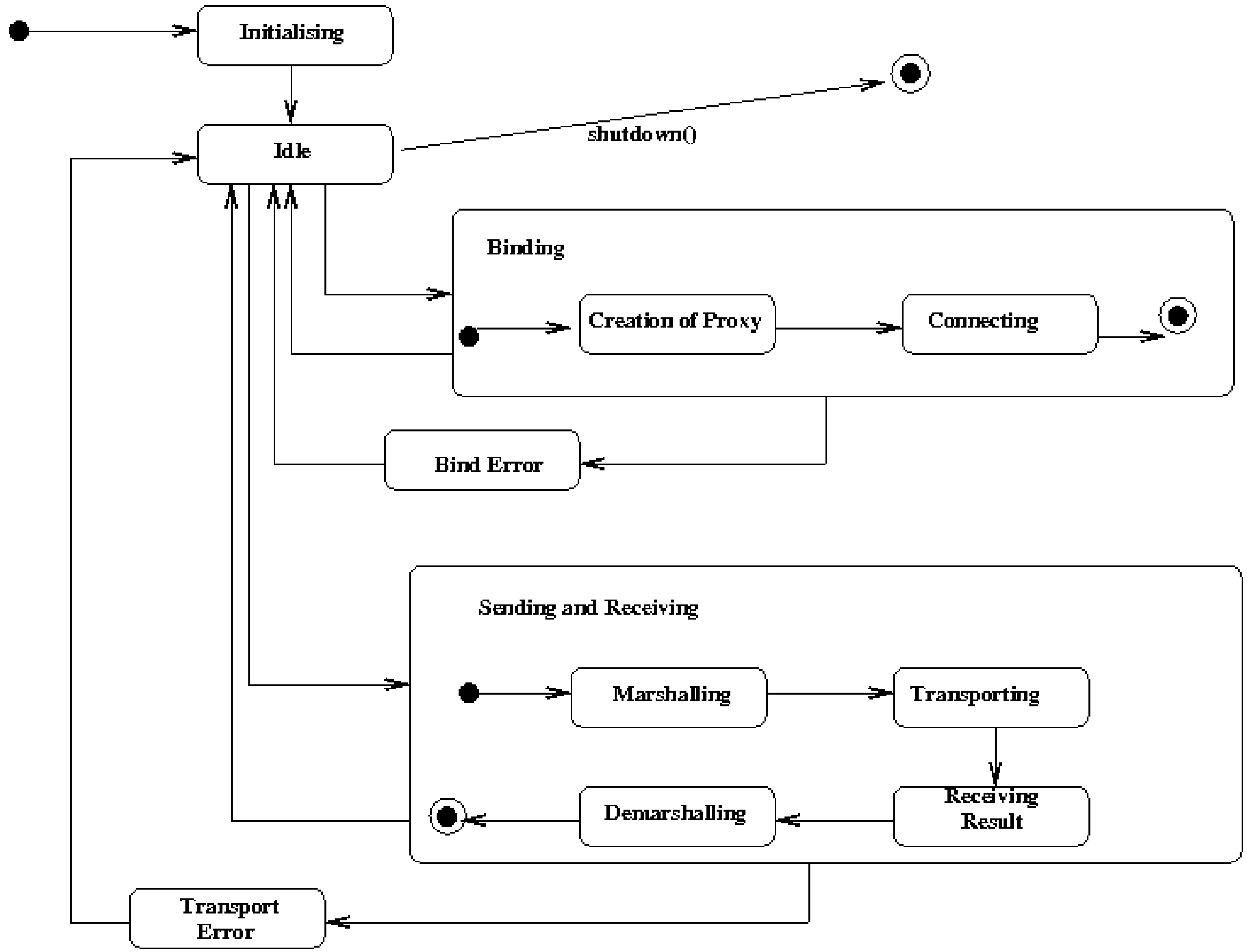
Known Uses

- **All middleware platforms**
- Glassfish
  - https://glassfish.java.net/docs/v3/api/javax/interceptor/package-summary.html
- Apache Tomcat
  - http://tomcat.apache.org/tomcat-6.0-doc/config/cluster-interceptor.html
- Document Interceptor for Sharepoint
  - http://www.archibus.com/ai/abizfiles/v19.3_help/system-management-help/Content/web_services/webservices_example7.htm
- Microsoft Internet Explorer to register plugins for specific media types.

Step 1: Model the Internal Behaviour of the Framework

- Create a composite state machine.

State machine for client side middleware platform – ORB in this example.



Capture the aspects that are subject to interception.

Step 2: Identify and Model Interception Points

(a) First, identify concrete framework state transitions that may not be visible to external applications, but are subject to interception.

- Example:
  - A client may wish to intercept an outgoing request so that it may add functionality such as logging, or modify the parameters dynamically such as Quality of Service arguments.
- These state transitions constitute interception points.

Step 2: Identify and Model Interception Points

(b) Partition interception points into Reader and Writer sets.

- Shown for client side only

Interception Point	Description	Reader / Writer
Binding	The client binds to a remote object. The framework creates a proxy and establishes communication. A monitoring service might intercept this event to visualise load.	Reader
PreMarshalOutRequest	Interceptors might change the target object to support load balancing, validate preconditions, or encrypt parameters.	Reader + Writer
PostMarshalOutRequest	A client might start a timer to measure latency	Reader
PreMarshalInReply	The reply has just arrived. A client may be interested in stopping a timer that measures latency	Reader
PostMarshalInReply	Evaluate post-conditions, decryption, validation of result, etc	Reader + Writer

Step 2: Identify and Model Interception Points

(c) Integrate interception points into the state machine model.

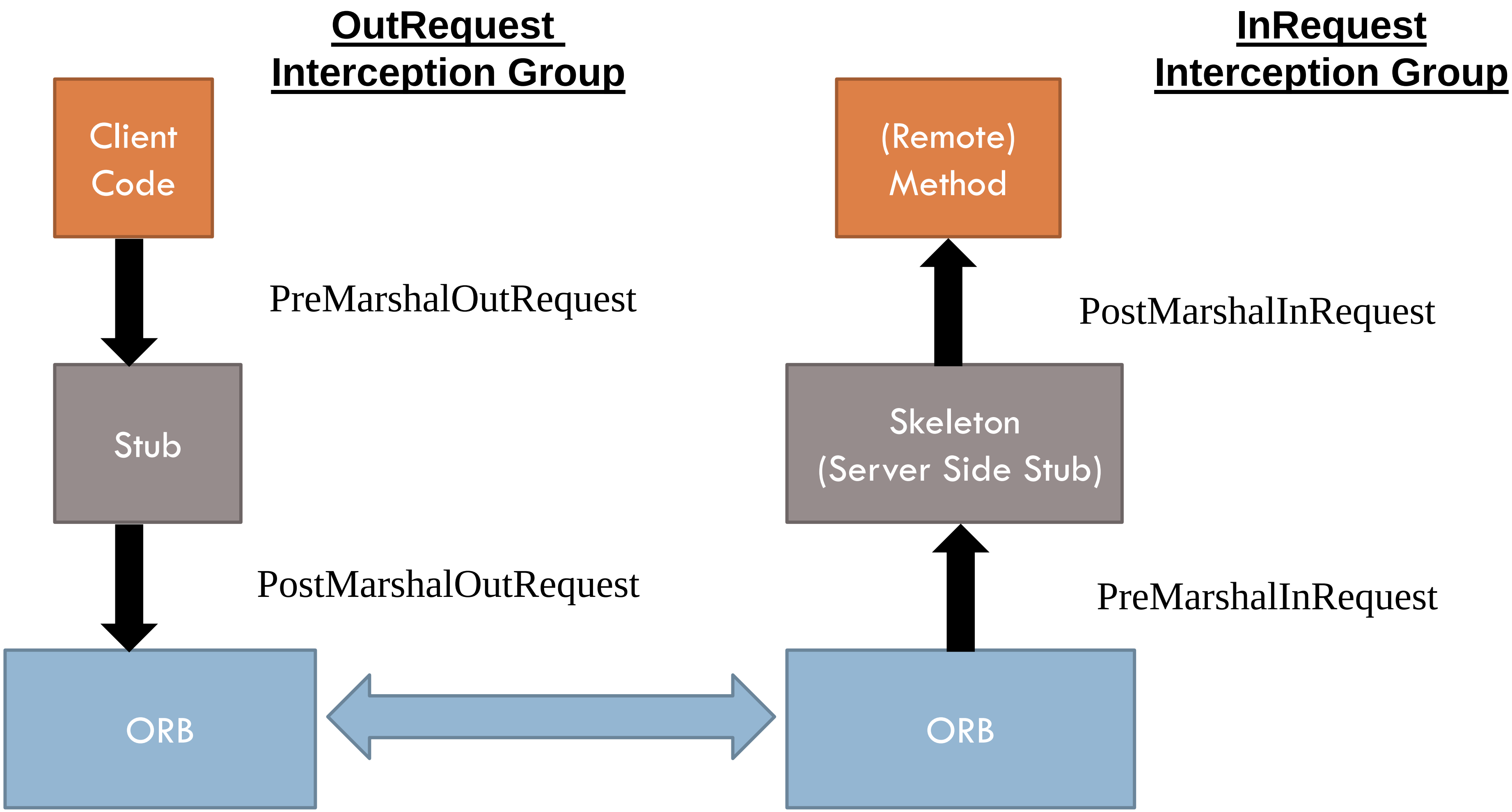
- By introducing intermediary states
- If a state transition is subject to interception, place a new transition between the source and sink states of the original transition.

(d) Partition into disjoint interception groups:

- By analysing the state machine for interception points in the same area.

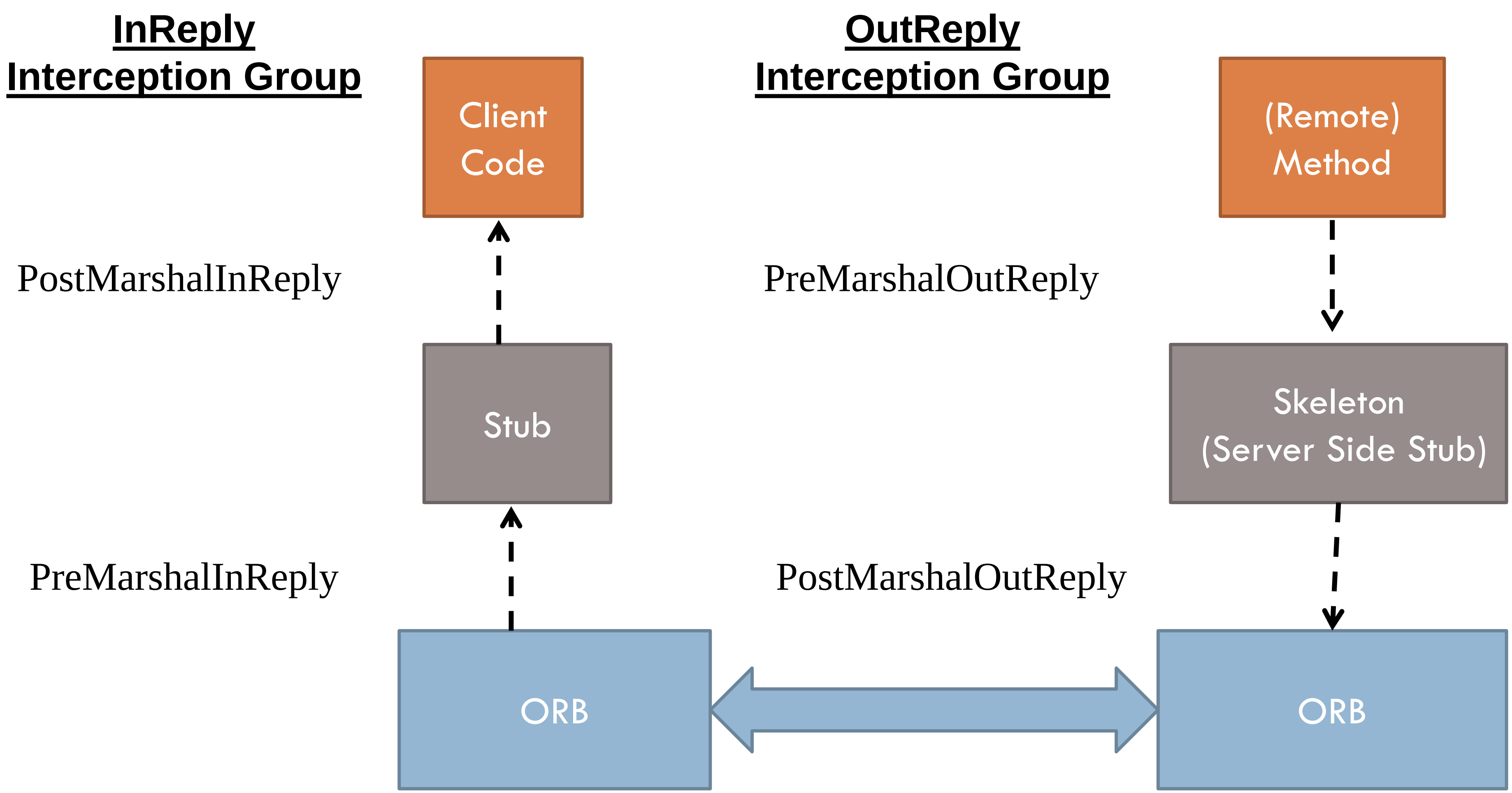
Step 2: Identify and Model Interception Points

- **Organise into Interception Groups**



Step 2: Identify and Model Interception Points

- **Organise into Interception Groups**



Step 2: Identify & Model Interception Points

- The disjoint interception groups:
  - **OutRequest** Interception Group
    - PreMarshalOutRequest
    - PostMarshalOutRequest
  - **InReply** Interception Group
    - PreMarshalInReply
    - PostMarshalInReply
- InRequest and OutReply on server side

Step 3: Specify the Context Objects

3.1 Determine the context object semantics

- If an interception point belongs to a reader set, determine what information should be provided.
- For writers, determine how to open the framework.
  - For example, methods that can modify a parameter.
  - Balance "open extensibility" versus "error prone interception code" that introduce vulnerabilities.

3.2 Determine the number of context objects

- Multiple interfaces: different context objects for a diverse set of interception points, facilitates fine grained control, more work for client developers.
- Single interface
  - May result in a bloated context interface that is difficult to understand

Step 3: Specify the Context Objects

- Example: intercepting outgoing events
  - Reading and changing the target object to support load balancing
  - Reading and modifying parameters to support encryption, validation, or change behaviour reflectively.
  - Adding new data to the request such as security tokens or transaction contexts to add out of band information to the request.
  - Integrating custom marshalling and unmarshalling components.

Step 3: Specify the Context Objects

- Correspond to activities associated with
  - `PreMarshalOutRequest` and
  - `PostMarshalOutRequest`.
- Introduce two context objects types
  - `UnmarshaledRequest`
  - `MarshaledRequest`

Step 3: Specify the Context Objects


```
public interface UnmarshaledRequest {
  public String getHost();
  public void setHost(String host);
  public long getPort();
  public long setPort(long newPort);
  public String getObjName();
  public void setObjName(String newName);
  public String getMethod();
  public void getMethod(String name);
  public void addInfo(Object info);
  // .............
}
```

Step 3: Specify the Context Objects

3.3 Determine how to pass context objects

- Per Registration: a context object is passed once and once only to a concrete interceptor upon registration
- Per-event

- Determined usually by number of interfaces.
- Per event has higher overheads due to repeated creation and deletion.

Step 4: Specify the Interceptors

- An interceptor defines a generic interface that a concrete framework uses to invoke concrete interceptors via dispatchers when interception points are triggered.
- An interceptor is defined for each interception group.
- A designated callback hook method (signature) is specified for each interception point in a group.
- The interceptor corresponds to the observer participant in the Observer Design Pattern.
- Designated callback hook mechanism plays the role of "*update()*"
- Interface for OutRequest Interception Group

```
public interface ClientRequestInterceptor {
  public void onPreMarshalRequest (UnmarshaledRequest context);
  public void onPostMarshaledRequest (MarshaledRequest context);
}
```

Step 5: Specify the Dispatchers

5.1 Specify the Interceptor registration interface.

- To implement different callback policies, applications can pass a priority token when registering concrete interceptors with the dispatcher.
  - Determines invocation order when multiple interceptors registered with same dispatcher
- Helper class can automate registration process

Step 5: Specify the Dispatchers


```
public class ClientRequestDispatcher {

    Vector interceptors_;

    synchonized public void registerClientRequestInterceptor
                                        (ClientRequestInterceptor i) {
            interceptors_.addElement(i);
    }

    synchonized public void removeClientRequestInterceptor
                                        (ClientRequestInterceptor i) {
            interceptors_.removeElement(i);
    }
}
```

Step 5: Specify the Dispatchers

5.2 Specify the Dispatcher callback interface

- When an interception event occurs, the framework callback to the dispatcher.
- The dispatcher then invokes the designated callback hook method of its registered concrete interceptors.
- A dispatcher often provides the same interface to the concrete framework that its associated interceptor offers to the dispatcher.

```
public class ClientRequestDispatcher implements ClientRequestInterceptor {
    ...............
}
```

- Benefits
  - Streamlines performance through delegation
  - Localises the modifications required if dispatcher interface changes.
    - Example: new interception point

Step 6: Implement the Callback Mechanisms in the Concrete Framework.

- Use Observer Design Pattern
- Dispatchers register with the concrete framework.
- When an interception event occurs, the framework calls back to the designated method in the dispatcher.
- Can pass the context as a parameter, or use a preallocated singleton.
- Dispatcher can use strategy to allow applications to use different callback strategies – FIFO/LIFO, priority, Chain of Responsibility.

Step 6: Implement the Callback Mechanisms in the Concrete Framework.


```
public class ClientRequestDispatcher {
    // ......
    public void dispatchClientRequestInterceptorPreMarshal
                        ( UnMarshaledRequest context) {
       Vector interceptors;
        synchronized (this) {
              interceptors = (Vector) interceptors.clone();
        }
       for(int i = 0; i < interceptors.size(); i++) {
             ClientRequestInterceptor ic =
                        (ClientRequestInterceptor)interceptors.elementAt(i);
             ic.onPreMarshalRequest(context);
        }
    }
    // .............
}
```

Step 7: Implement the Concrete Interceptors


```
public class Client {

   static final void main(Strings args[]) {

         ClientRequestInterceptor myInterceptor = new ClientRequestInterceptor () {
               public void onPreMarshalRequest(UnMarshaledRequest context) {
                     System.out.println(context.getObj() + “called”);
                     // …………
               }
               public void onPostMarshalRequest(MarshaledRequest context) {
                     // …………
               }
         };

         ClientRequestDispatcher.theInstance().registerClientRequestInterceptor
                           (myInterceptor);
         // Do Work
   }
}
```

The Interceptor Architectural Pattern

- The dispatcher appears to act the role of a dynamic proxy i.e. introduces another level of indirection that decouples clients from servers that are linked through callbacks.
- This is found in the abstract client pattern that supports callbacks.
- Patterns found in Gamma et al. including Bridge, Chain of Responsibility, Observer, and Strategy uses the Abstract Client. The Reactor pattern [Schmidt], the Completion Callback pattern [Lea], the Conduits+ framework [Huni] also use the Abstract Client.
- This pattern is central to OO programming, and is the easiest callback pattern to apply.
- Java provides an interface facility for specifying Abstract Client interfaces.
- In C, operational equivalence is facilitated by using pointers to functions.
- For a layman's overview, see Paul Jakubik (1997). Callback Implementations in C++.
- Abstract client is further generalised by the adapter pattern.

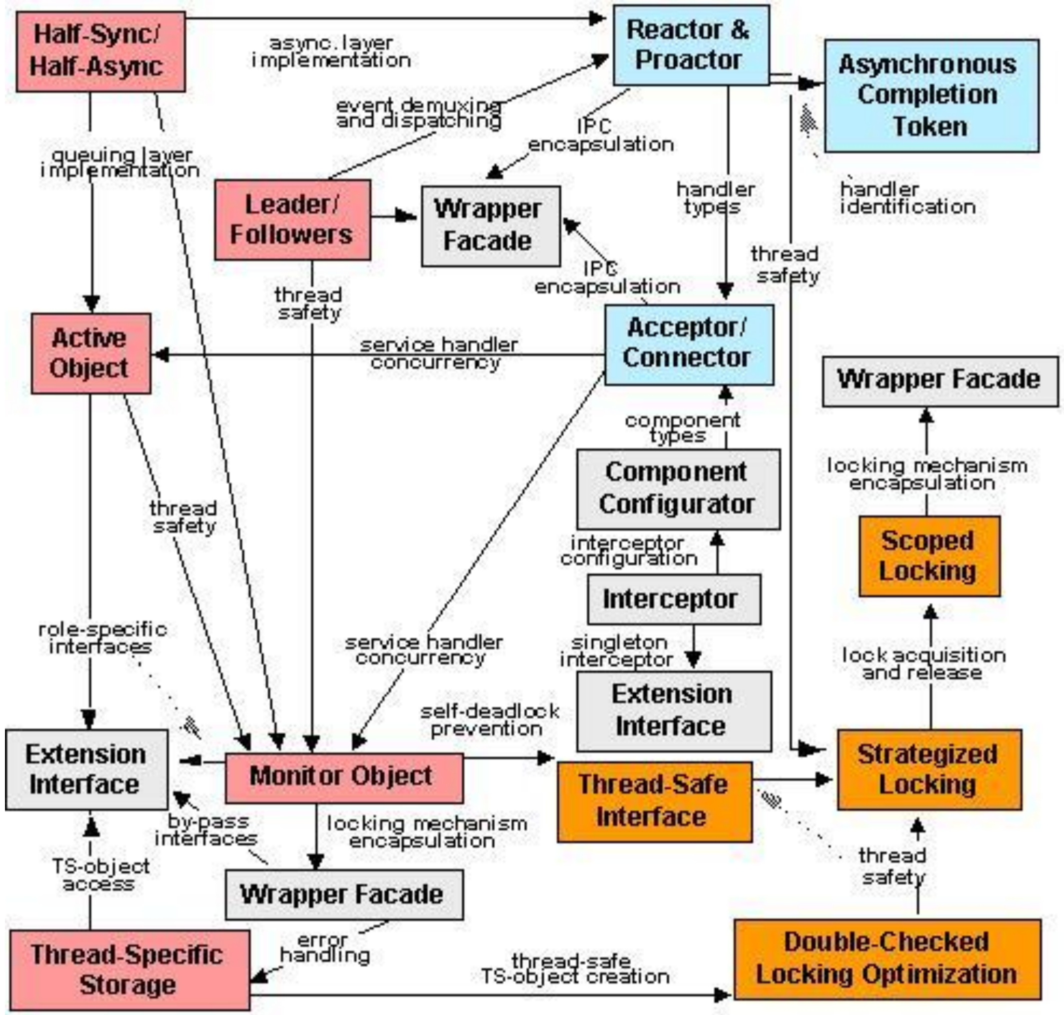
The Interceptor Architectural Pattern

- Interception is commonly used to handle security & transactions transparently.
- It can also enable performance enhancement strategies
  - e.g., just-in-time activation, object pooling, load balancing, & caching
- Interceptor are a "meta-programming mechanism," along with
  - Smart proxies
  - Pluggable protocols
  - Gateways/bridges
  - Interface repositories
- Provide building-blocks to handle extensions, often unanticipated.
- More on meta-programming:
  - www.cs.wustl.edu/~schmidt/PDF/IEEE.pdf

The Interceptor Architectural Pattern

- Variants:
  - Interceptor Proxy variant (also known as Delegator).
  - Often used on the server side of a distributed system to intercept remote operations.
- Interceptor facilitates service extensions to frameworks.
- Used with **Acceptor-Connector**, **Component Configurator**, **Active-Object**, **Monitor Object**, **Wrapper façade** and other design patterns.
- See Schmidt et al (2004). Pattern-Oriented Software Architecture – Volume 2:

POSA 2 Catalogue



Architectural Patterns

- Architectural and design patterns weaved together in a pattern language for middleware and applications
- The **Leader/Followers** architectural pattern provides a concurrency model that allows multiple threads to share a set of event sources.
- The **Reactor** architectural pattern structures event–driven applications, particularly servers that receive requests from multiple clients concurrently, but processes them iteratively.
- The **Proactor** architectural pattern structures event-driven applications that receive service requests from concurrent clients.