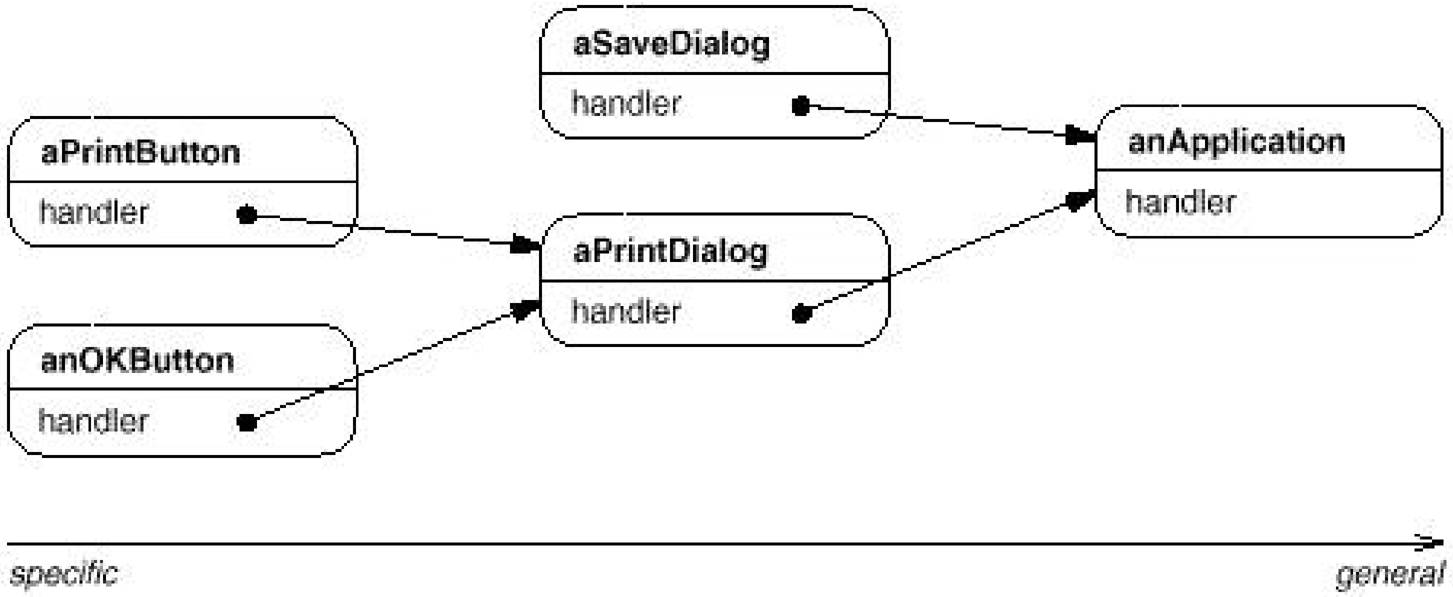
The
Chain of Responsibility
Pattern

The Chain of Responsibility Pattern

- Intent
  - ➪ Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Motivation
  - ➪ Consider a context-sensitive help system for a GUI
  - ➪ The object that ultimately provides the help isn't known explicitly to the object (e.g., a button) that initiates the help request
  - ➪ So use a chain of objects to decouple the senders from the receivers. The request gets passed along the chain until one of the objects handles it.
  - ➪ Each object on the chain shares a common interface for handling requests and for accessing its successor on the chain

The Chain of Responsibility Pattern

- Motivation

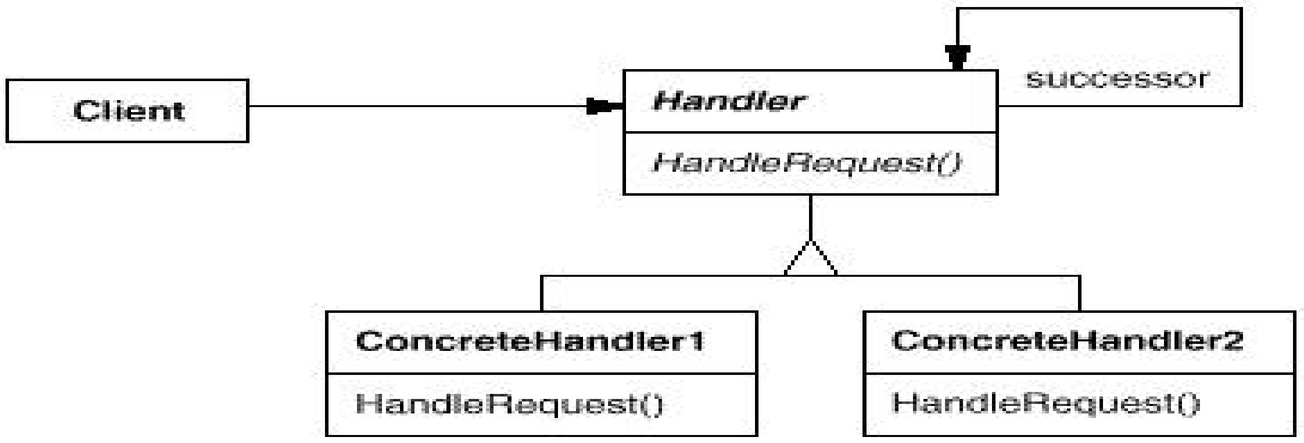


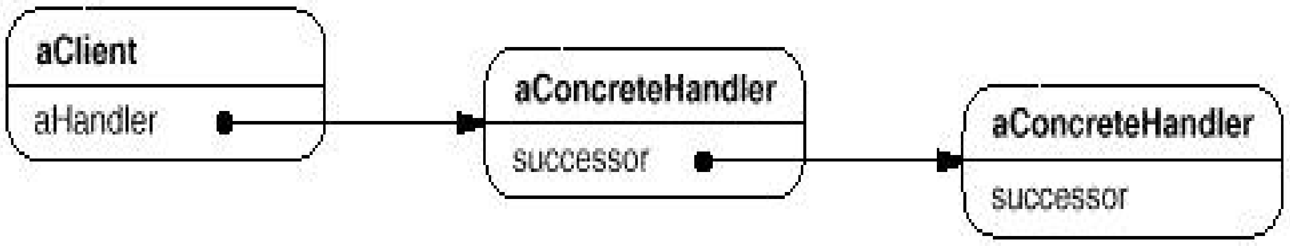
The Chain of Responsibility Pattern

- Applicability
  - Use Chain of Responsibility:
    - When more than one object may handle a request and the actual handler is not know in advance
    - When requests follow a “handle or forward” model - that is, some requests can be handled where they are generated while others must be forwarded to another object to be handled
- Consequences
  - Reduced coupling between the sender of a request and the receiver - the sender and receiver have no explicit knowledge of each other
  - Receipt is not guaranteed - a request could fall off the end of the chain without being handled
  - The chain of handlers can be modified dynamically

The Chain of Responsibility Pattern

- Structure





Chain of Responsibility Example 1

- Scenario: The designers of a set of GUI classes need to have a way to propagate GUI events, such as MOUSE_CLICKED, to the individual component where the event occurred and then to the object or objects that are going to handle the event
- Solution: Use the Chain of Responsibility pattern. First post the event to the component where the event occurred. That component can handle the event or post the event to its container component (or both!). The next component in the chain can again either handle the event or pass it up the component containment hierarchy until the event is handled.
- This technique was actually used in the Java 1.0 AWT

Chain of Responsibility Example 1 (Continued)

- Here's a typical Java 1.0 AWT event handler:

```
public boolean action(Event event, Object obj) {
  if (event.target == test_button)
    doTestButtonAction();
  else if (event.target == exit_button)
    doExitButtonAction();
  else
    return super.action(event,obj);
  return true;  // Return true to indicate the event has been
                // handled and should not be propagated further.
}
```

Chain of Responsibility Example 1 (Continued)

- In Java 1.1 the AWT event model was changed from the Chain of Responsibility (CoR) pattern to the Observer pattern. Why?
  - Efficiency: GUIs frequently generate many events, such as MOUSE_MOVE events. In many cases, the application did not care about these events. Yet, using the CoR pattern, the GUI framework would propagate the event up the containment hierarchy until some component handled it. This caused the GUI to slow noticeably.
  - Flexibility: The CoR pattern assumes a common Handler superclass or interface for all objects which can handle chained requests. In the case of the Java 1.0 AWT, every object that could handle an event had to be a subclass of the Component class. Thus, events could not be handled be non-GUI objects, limiting the flexibility of the program.

Chain of Responsibility Example 1 (Continued)

- The Java 1.1 AWT event model uses the Observer pattern. Any object that wants to handle an event registers as an event listener with a component. When an event occurs, it is posted to the component. The component then dispatches the event to any of its listeners. If the component has no listeners, the event is discarded. For this application, the Observer pattern is more efficient and more flexible!

Chain of Responsibility Example 2

- Scenario: We are designing the software for a security monitoring system. The system uses various sensors (smoke, fire, motion, etc.) which transmit their status to a central computer. We decide to instantiate a sensor object for each physical sensor. Each sensor object knows when the value it is sensing exceeds some threshold(s). When this happens, the sensor object should take some action. But the action that should be taken may depend on factors other than just the sensor's value. For example, the location of the sensor, the value of the data or equipment located at the sensor’s position, the value of the data or equipment in other locations near the sensor’s position, etc. We want a very scalable solution in which we can use our physical sensors and sensor objects in any environment. What can be do??

Chain of Responsibility Example 2 (Continued)

- Solution: Use the Chain of Responsibility pattern. Aggregate sensor objects in a containment hierarchy that mirrors the required physical zones (areas) of security. Define wall, room, floor, area, building, campus and similar objects as part of this containment hierarchy. The alarm generated by a sensor is passed up this hierarchy until some containment object handles it.

Chain of Responsibility Example 3

- Scenario: We are designing the software for a system that approves purchasing requests. The approval authority depends on the dollar amount of the purchase. The approval authority for a given dollar amount could change at any time and the system should be flexible enough to handle this situation.
- Solution: Use the Chain of Responsibility pattern. PurchaseRequest objects forward the approval request to a PurchaseApproval object. Depending on the dollar amount, the PurchaseApproval object may approve the request or forward it on to the next approving authority in the chain. The approval authority at any level in the chain can be easily modified without affecting the original PurchaseRequest object.

Handling Requests

- Notice in the basic structure for the CoR pattern that the Handler class just has one method, handleRequest(). Here it is as a Java interface:

```
public interface Handler {
  public void handleRequest();
}
```

- What if we want to handle different kinds of requests, for example, help, print and format requests?

Handling Requests (Continued)

- Solution 1:We could change our Handler interface to support multiple request types as follows:

```
public interface Handler {
  public void handleHelp();
  public void handlePrint();
  public void handleFormat();
}
```

- Now any concrete handler would have to implement all of the methods of this Handler interface.

Handling Requests (Continued)

- Here's an example of a concrete handler for this new Handler interface:

```
public class ConcreteHandler implements Handler {
  private Handler successor;

  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }

  public void handleHelp() {
    // We handle help ourselves, so help code is here.
  }

  public void handlePrint() {
    successor.handlePrint();
  }
```

Handling Requests (Continued)

```
    public void handleFormat() {
      successor.handleFormat();
    }
  }
```

- Of course, if we add a new kind of request we need to change the interface which means that all concrete handlers need to be modified!

Handling Requests (Continued)

- Solution 2: Another solution is to have separate handler interfaces for each type of request. For example, we could have:

```
public interface HelpHandler {
  public void handleHelp();
}

public interface PrintHandler {
  public void handlePrint();
}

public interface FormatHandler {
  public void handleFormat();
}
```

Handling Requests (Continued)

- Now a concrete handler can implement one (or more) of these interfaces. The concrete handler must have successor references to each type of request that it deals with, in case it needs to pass the request on to its successor.

Handling Requests (Continued)

- Here's a concrete handler which deals with all three request types:

```
public class ConcreteHandler
  implements HelpHandler, PrintHandler, FormatHandler {

  private HelpHandler helpSuccessor;
  private PrintHandler printSuccessor;
  private FormatHandler formatSuccessor;

  public ConcreteHandler(HelpHandler helpSuccessor
                         PrintHandler printSuccessor,
                         FormatHandler formatSuccessor) {
    this.helpSuccessor = helpSuccessor;
    this.printSuccessor = printSuccessor;
    this.formatSuccessor = formatSuccessor;
  }
```

Handling Requests (Continued)

```
    public void handleHelp() {
      // We handle help ourselves, so help code is here.
    }

    public void handlePrint() {printSuccessor.handlePrint();}

    public void handleFormat() {formatSuccessor.handleFormat();}

  }
```

Handling Requests (Continued)

- Solution 3: Still another approach to handling different kinds of requests is to have a single method in the Handler interface which takes an argument describing the type of request. For example, we could describe the request as a String:

```
public interface Handler {
  public void handleRequest(String request);
}
```

Handling Requests (Continued)

- And now our concrete handler looks like:

```
public class ConcreteHandler implements Handler {
  private Handler successor;

  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }

  public void handleRequest(String request) {
    if (request.equals("Help")) {
      // We handle help ourselves, so help code is here.
    }
    else
      // Pass it on!
      successor.handle(request);
  }
}
```