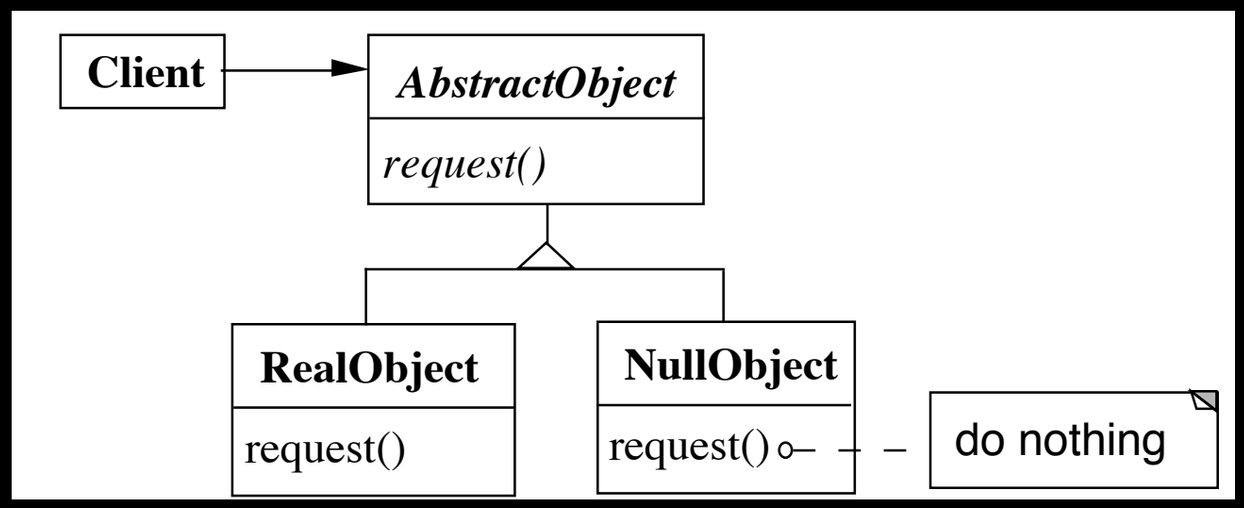
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2018
Doc 15 Null Object, Chain of Responsibility, Pipes and Filters
Nov 1, 2018

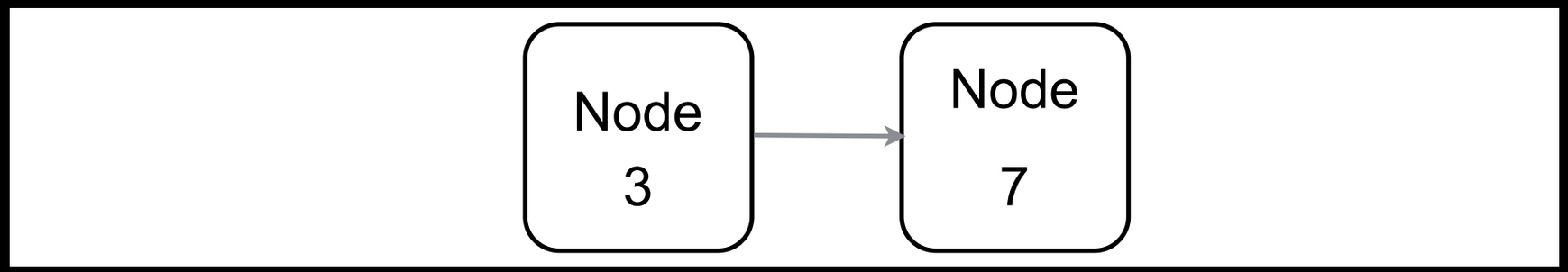
Null Object

Null Object



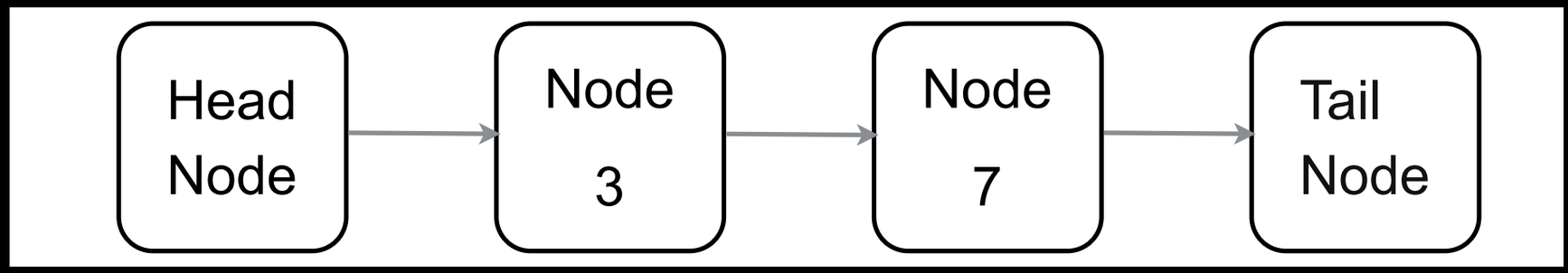
NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing




```
class LinkedList {
  Node head;

  public toString() {
    if (head == nil) {
      return “()”;
    }
    String listAsString = “(;
    Node current = head;
    while (current != null) {
      listAsString += current.value() + “, “;
      current = current.next;
    }
    listAsString = removetail(listsString, 2);
    return listAsString + “)”;
  }
```




```
class LinkedList {
  Node head;

  public toString() {
    return head.toString();
    }
```

```
class Node {
   public String toString() {
      return " " + element + next.toString();
   }
}
```

```
class HeadNode {
   public String toString() {
      return "(" + next.toString();
   }
}
```

```
class TailNode {
   public String toString() {
      return " )";
   }
}
```

Applicability - When to use Null Objects

Some collaborator instances should do nothing

You want clients to ignore the difference between a collaborator that does something and one that does nothing

Client does not have to explicitly check for null or some other special value

You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way

Applicability -When not to use Null Objects

Very little code actually uses the variable directly

The code that does use the variable is well encapsulated

The code that uses the variable can easily decide how to handle the null case and will always handle it the same way

Consequences

Advantages

Uses polymorphic classes

Simplifies client code

Encapsulates do nothing behavior

Makes do nothing behavior reusable

Disadvantages

Forces encapsulation

Makes it difficult to distribute or mix into the behavior of several collaborating objects

May cause class explosion

Forces uniformity

Is non-mutable

Implementation

Too Many classes

Multiple Do-nothing meanings

Try Adapter pattern

Transformation to RealObject

Try Proxy pattern

Refactoring: Introduce Null Object

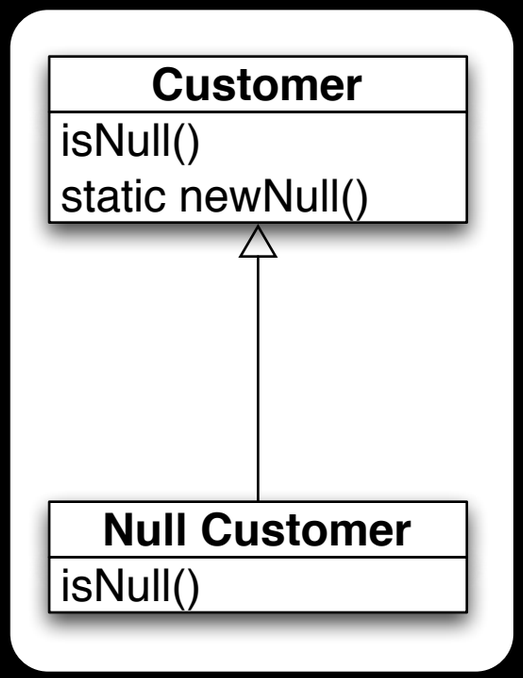
You have repeated checks for a null value

Replace the null value with a null object

```
if (customer == null)
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

Create Null Subclass




```
public boolean isNull() { return false;}
public static Customer newNull() { return new NullCustomer();}
```

```
boolean isNull() { return true;}
```

Compile

Replace all nulls with null object

```
class SomeClassThatReturnCustomers {

   public Customer getCustomer() {
      if (_customer == null )
         return Customer.newNull();
      else
         return _customer;
      }
      etc.
}
```

Compile

Replace all null checks with isNull()

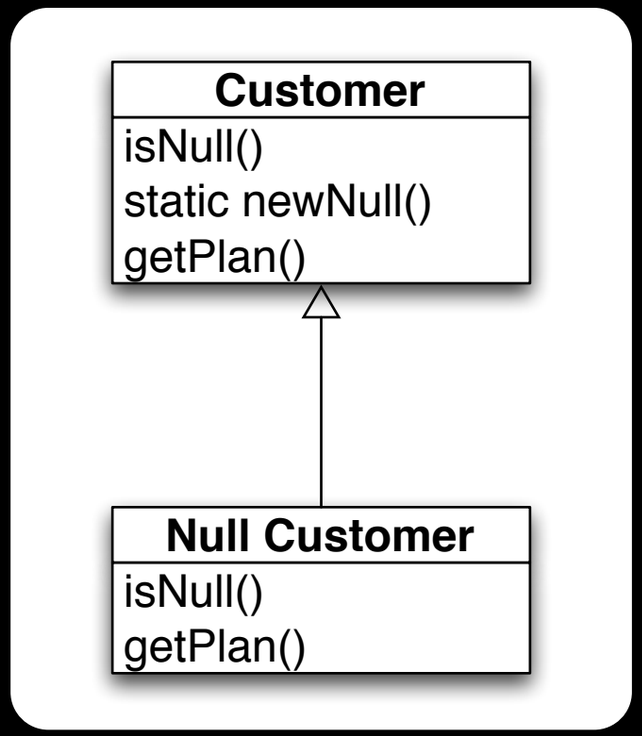
```
if (customer == null)
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```

```
if (customer.isNull())
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```

Compile and test

Find an operation clients invoke if not null
Add Operation to Null class

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```




```
class NullCustomer {
   public BillingPlan getPlan() {
      return BillingPlan.basic();
   }
```

Remove the Condition Check

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

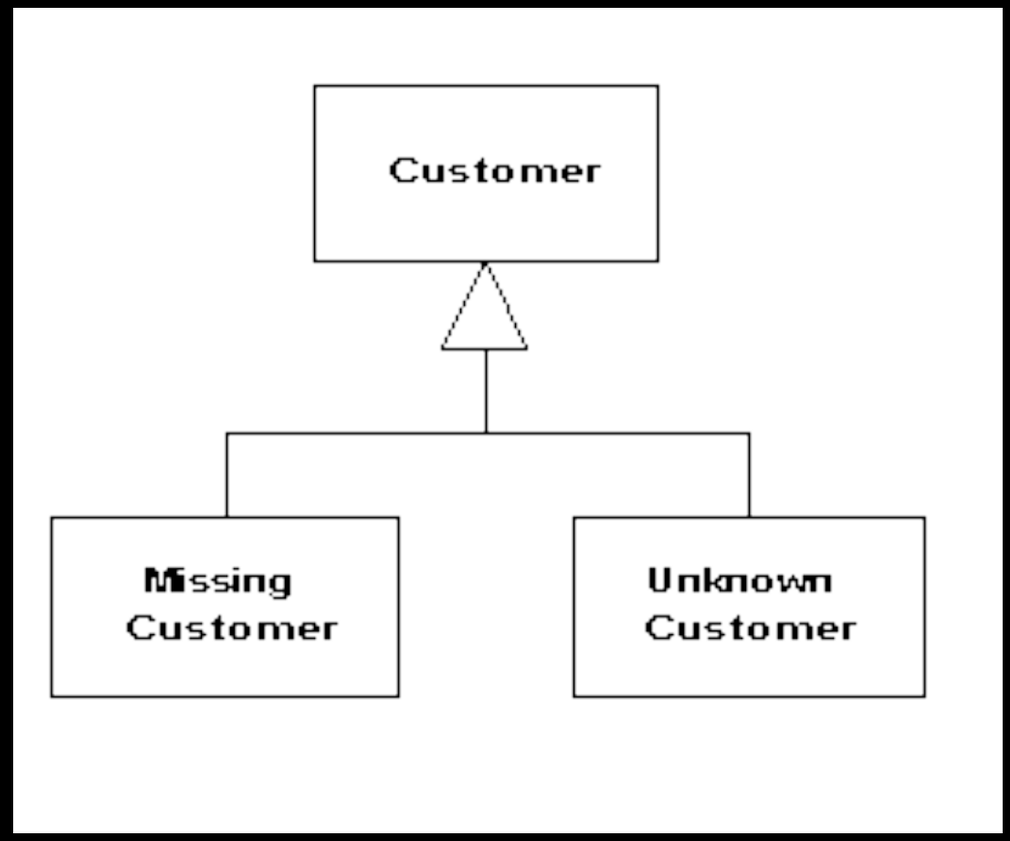
Compile & Test

Repeat last two slides for each operation
clients check if null

Special Case

Special Case

Represent special cases by a subclass



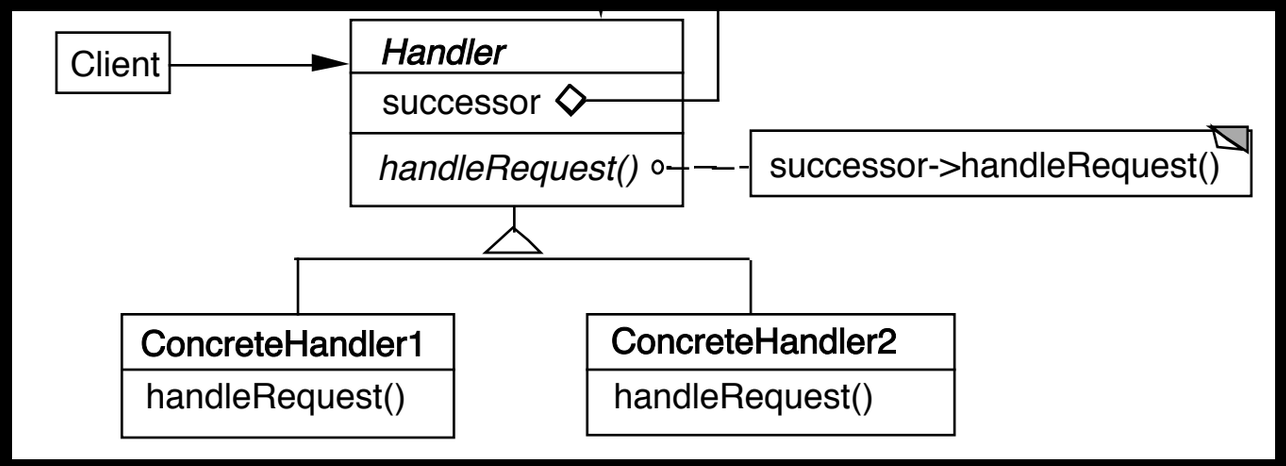
Use when multiple places that have same behavior

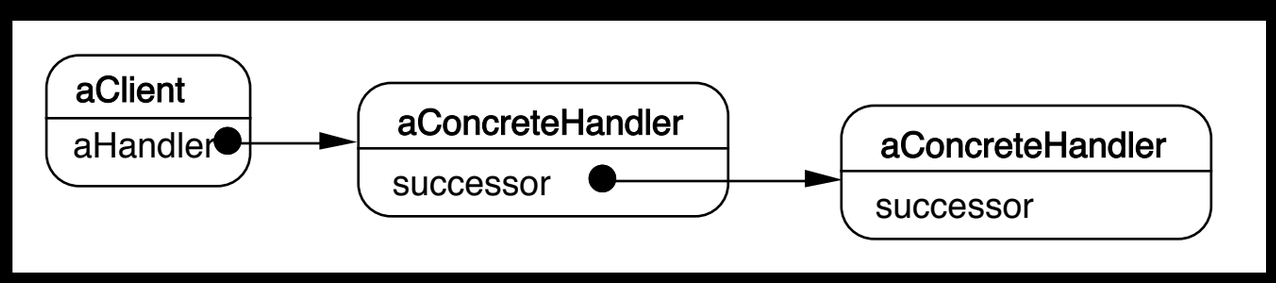
After conditional check for particular class instance

Or same behavior after a null check

Chain of Responsibility

Chain of Responsibility





Dynamically create chain of handlers

Multiple handlers may be able to handle a request

Only one handler actually handles the request

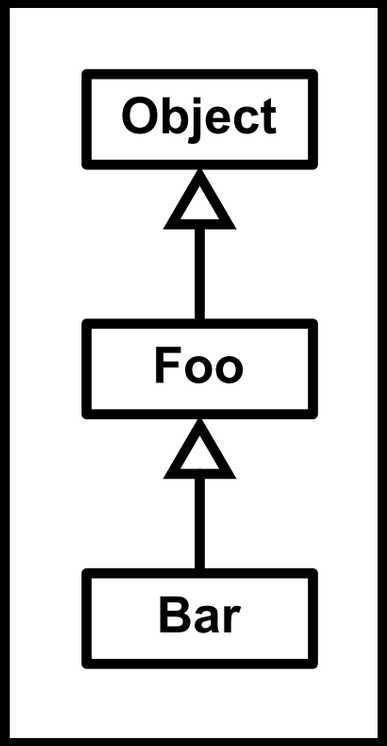
Consequences

Reduced coupling

Added flexibility in assigning responsibilities to c

Not guaranteed that request will be handled

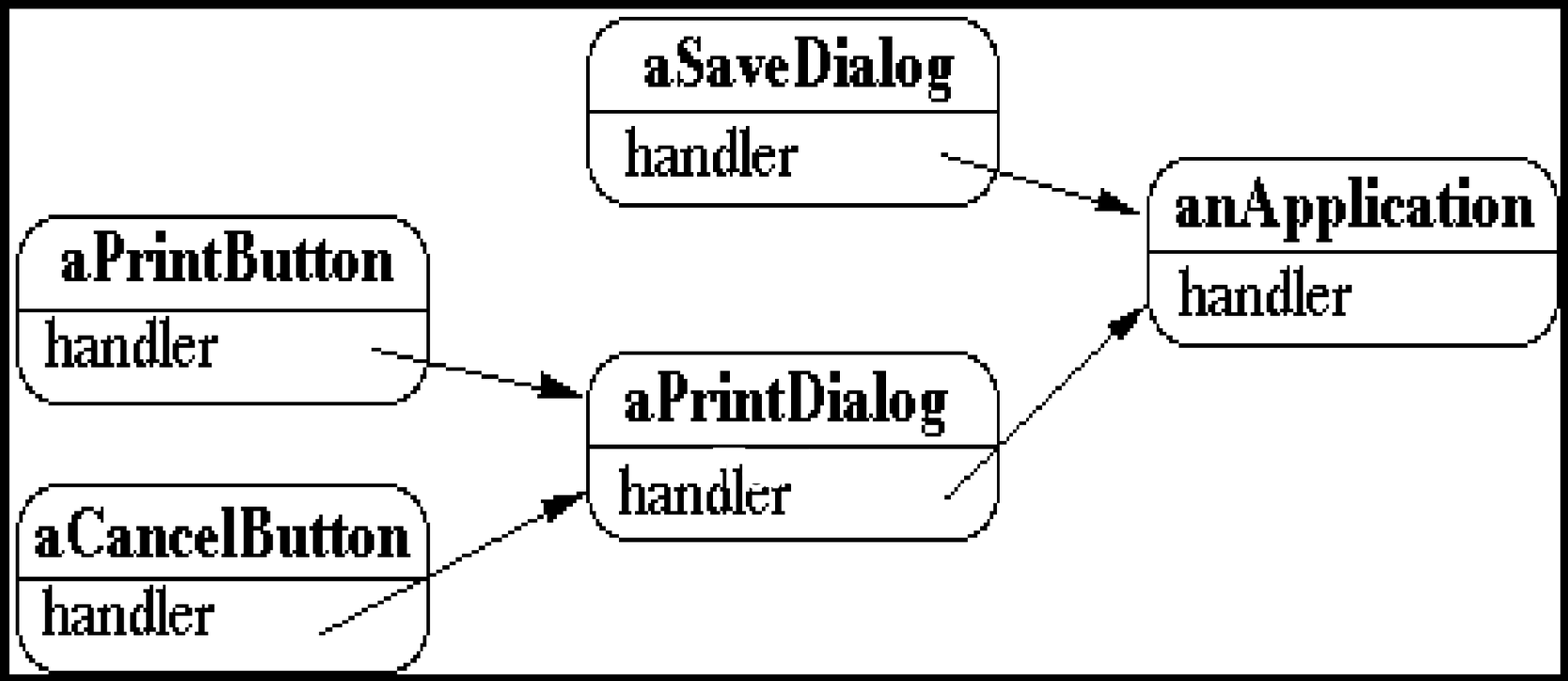
Finding Methods




```
test = new Bar();
test.toString();
```

Context Help System

User clicks on component for help



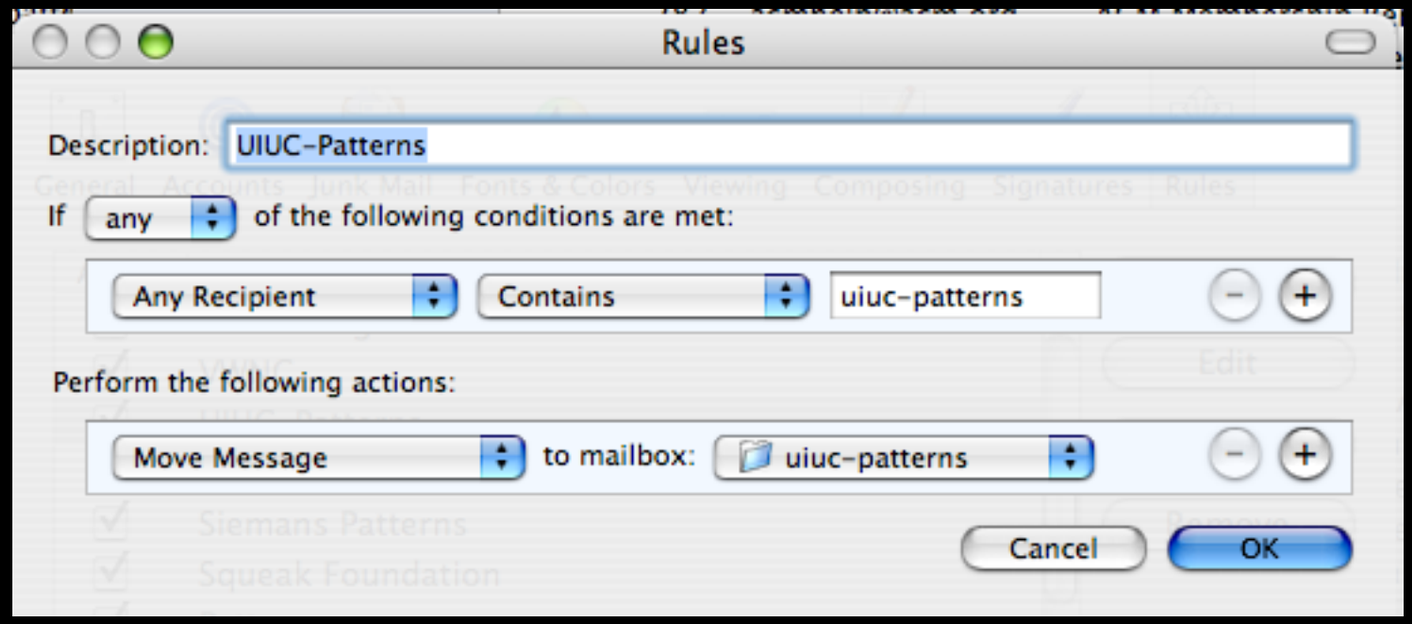
Tree of handlers

From specific to general

Email Filters in Mail Client

User creates a set of rules
- delete
- move
- modify

Chain the rules
- First rule that applies handles the mail



Other Examples

Java 1.0 AWT action(Event)

http://wiki.cs.uiuc.edu/PatternStories/JavaAWT

javax.servlet.Filter

http://tomcat.apache.org/tomcat-4.1-doc/servletapi/javax/servlet/Filter.html

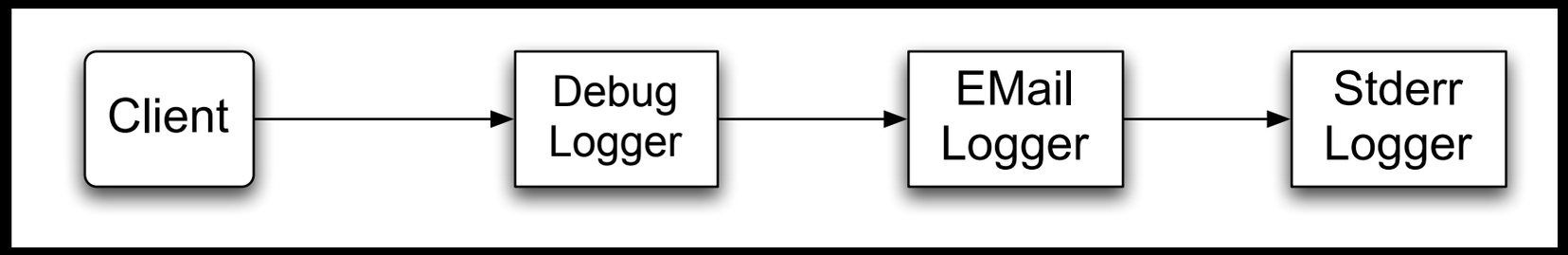
Microsoft Windows global keyboard events

http://www.javaworld.com/javaworld/jw-08-2004/jw-0816-chain.html

Apache Commons Chain

http://commons.apache.org/chain/

Logger Example




```
class ChainOfResponsibilityExample {
   public static void main(String[] args) {
      // building the chain of responsibility
      Logger l = new DebugLogger(Logger.DEBUG).setNext(
                  new EMailLogger(Logger.ERR).setNext(
                  new StderrLogger(Logger.NOTICE) ) );

      l.message("Entering function x.", Logger.DEBUG);    // handled by DebugLogger
      l.message("Step1 completed.", Logger.NOTICE);       // handled by Debug- and
StderrLogger
      l.message("An error has occurred.", Logger.ERR);    // handled by all three Logger
   }
}
```

First Attempt

```
abstract class Logger {
   public static int ERR = 3;
   public static int NOTICE = 5;
   public static int DEBUG = 7;
   protected int mask;

   protected Logger next;
   public Logger setNext(Logger l) {
      next = l;
      return this; }

   abstract public void message(String msg, int priority);
}

class DebugLogger extends Logger {
   public DebugLogger(int mask) {
      this.mask = mask; }

   public void message(String msg, int priority) {
      if (priority <= mask) debug log here
      if (next != null) next.message(msg, priority);
   }
}
```

```
class EMailLogger extends Logger {
   public EMailLogger(int mask) { this.mask = mask; }

   public void message(String msg, int priority) {
      if (priority <= mask) send email here;
      if (next != null) next.message(msg, priority);
   }
```

Improved Logger

```
abstract class Logger {
   public static int ERR = 3;
   public static int NOTICE = 5;
   public static int DEBUG = 7;
   protected int mask;

   protected Logger next;
   public Logger setNext(Logger l) {
     next = l;
    return this; }
   public void message(String msg, int priority) {
      if (priority <= mask) log(msg);
      if (next != null) next.message(msg, priority);
   }

   abstract void log(String message);
}

class StderrLogger extends Logger {
   public StderrLogger(int mask) { this.mask = mask; }

   void message(String msg, int priority) { send to err}
}
```

```
class EMailLogger extends Logger {
   public EMailLogger(int mask) { this.mask = mask; }

    void message(String msg, int priority) { email here}
   }
}

class DebugLogger extends Logger {
   public DebugLogger(int mask) { this.mask = mask; }

   void message(String msg, int priority) { debug stuff}
   }
}
```

Is this the Chain of Responsibility?

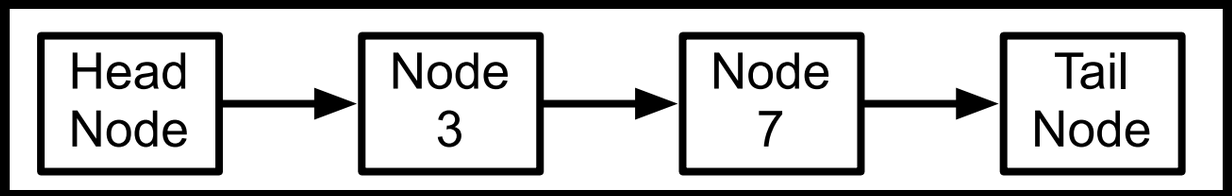
Object-Oriented Recursion

A method polymorphically sends its message to a different receiver

Eventually a method is called that performs the task

The recursion then unwinds back to the original message send

Linked List toString



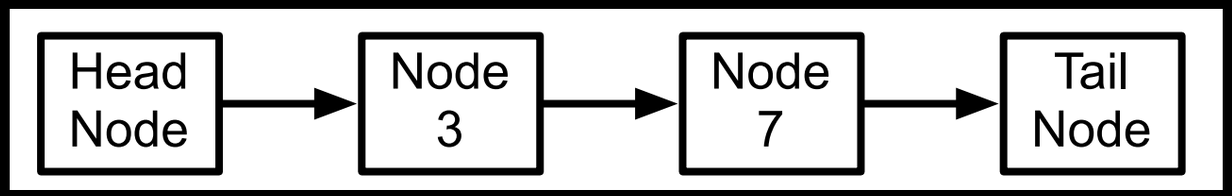
(3 7)

```
class HeadNode {
   public String toString() {
      return "(" + next.toString();
   }
}
```

```
class TailNode {
   public String toString() {
      return " )";
   }
}
```

```
class Node {
   public String toString() {
      return " " + element + next.toString();
   }
}
```

Linked List add




```
class HeadNode {
   public void add(int value) {
      next.add(value);
   }
}
```

```
class TailNode {
   public void add(int value) {
      prependNode(value);
   }
}
```

```
class Node {
   public void add(int value) {
      if (element > value)
         prependNode(value);
      else
         next.add(value);
   }
}
```

OO Recursion

Decorator

Chain of Responsibility

Pipes and Filters

Pipes & Filters

ls | grep -i b | wc -l

Context
Processing data streams

Problem
Building a system that processes or transforms a stream of data

Forces
Small processing steps are easier to reuse than large components

Non-adjacent processing steps do not share information

System changes should be possible by exchanging or recombining processing steps, even by users

Final results should be presented or stored in different ways

Solution

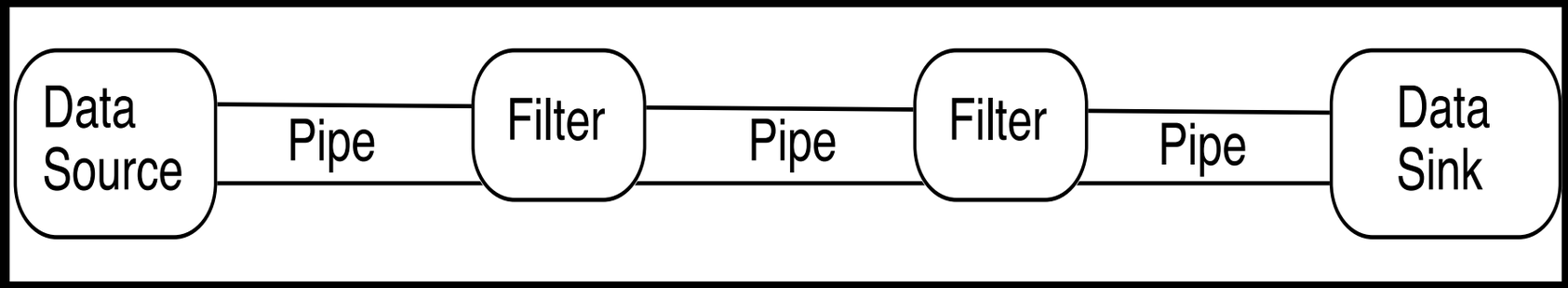
Divide task into multiple sequential processing steps or filter components

Output of one filter is the input of the next filter

Filters process data incrementally

Filter does not wait to get all the data before processing

Solution Continued



Data source – input to the system

Data sink – output of the system

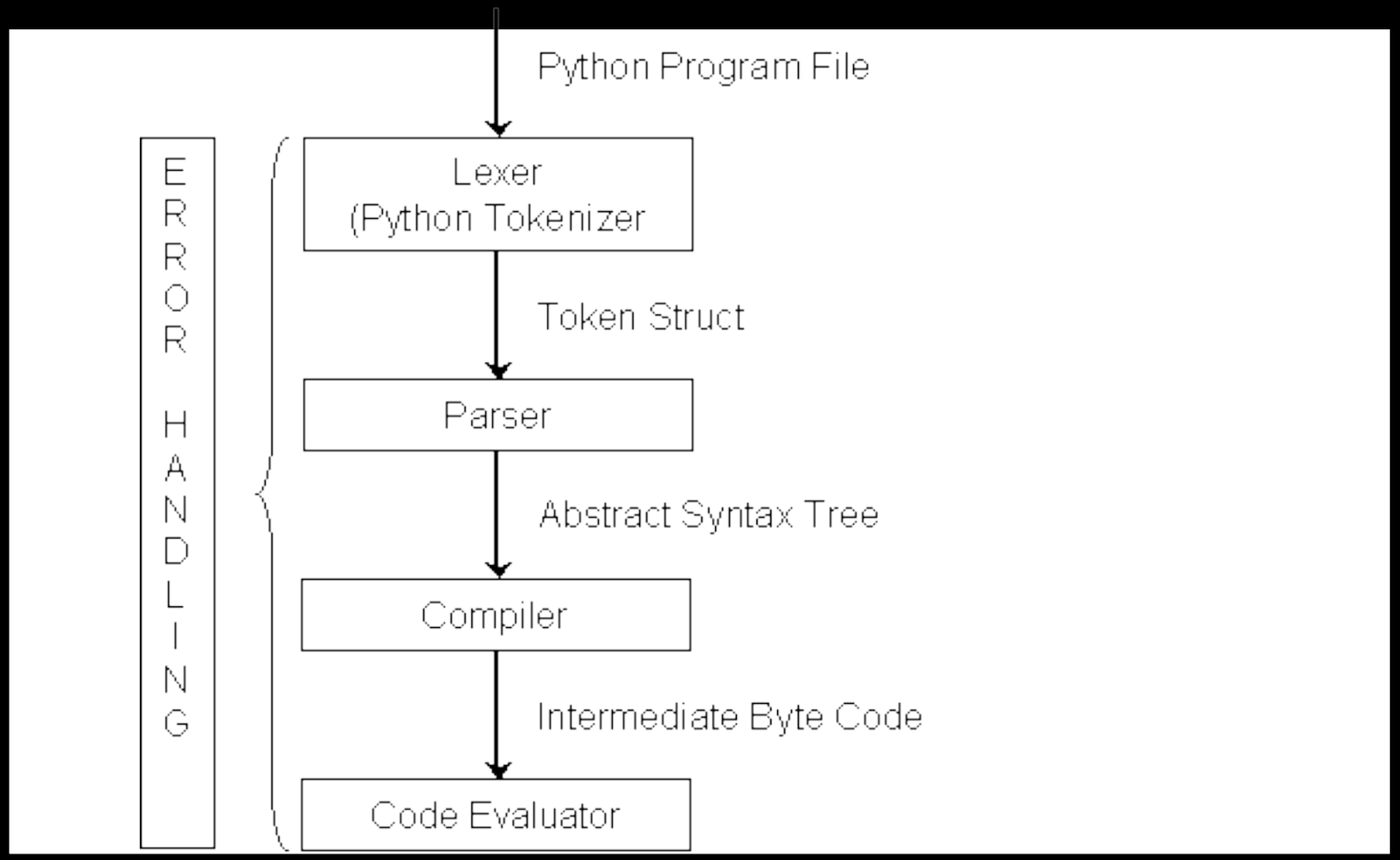
Pipes - connect the data source, filters and data sink

Pipe implements the data flow between adjacent processes steps

Processing pipeline – sequence of filters and pipes

Pipeline can process batches of data

Python Interpreter



http://wiki.cs.uiuc.edu/cs427/Python+-+Batch+Sequential

Intercepting Filter - Problem

Preprocessing and post-processing of a client Web request and response

A Web request often must pass several tests prior to the main processing

- Has the client been authenticated?
- Does the client have a valid session?
- Is the client's IP address from a trusted network?
- Does the request path violate any constraints?
- What encoding does the client use to send the data?
- Do we support the browser type of the client?

Nested if statements lead to fragile code

Intercepting Filter - Forces

Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.

Centralization of common logic is desired.

Services should be easy to add or remove unobtrusively without affecting existing components, so that they can be used in a variety of combinations, such as

Logging and authentication

Debugging and transformation of output for a specific client

Uncompressing and converting encoding scheme of input