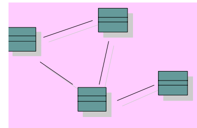
SE203b: OO Design for Software Engineers

W7: ***OO Design Approach***

Architecture & Design patterns

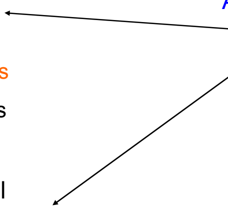


The Road Map

- ✓ Introduction to Software Design
- ✓ Software Design Approaches
- ✓ Introduction to OO Paradigm
- ➢ Software Design with OO Paradigm
- Patterns in Design and Architecture
- Selected Design Topics

OO Software Requirements & Design Process

- ✓ Requirement capture using Use Cases
  - ✓ Use Case Diagrams
  - ✓ Use Case Description
- ➢ Analysis & Design Model
  - ✓ Class Diagrams
    - ✓ Data Dictionary
  - ✓ Interaction Diagrams
  - Statechart Diagrams
  - Activity Diagrams
- Implementation Model
- Deployment Model

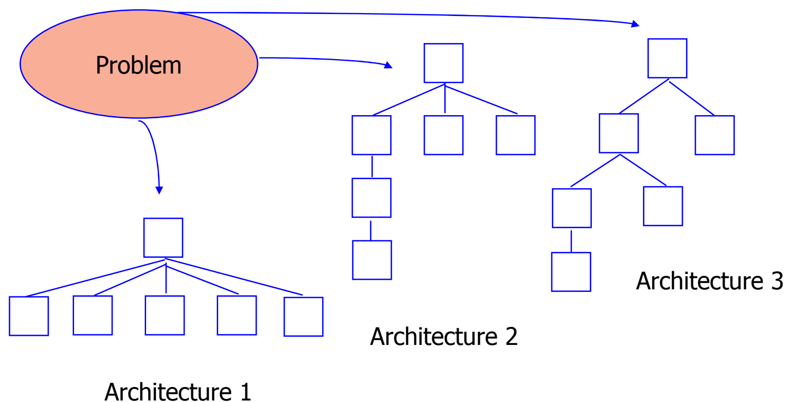


Architecture-centric
- Patterns
- Frameworks

Architecture

System architecture describes

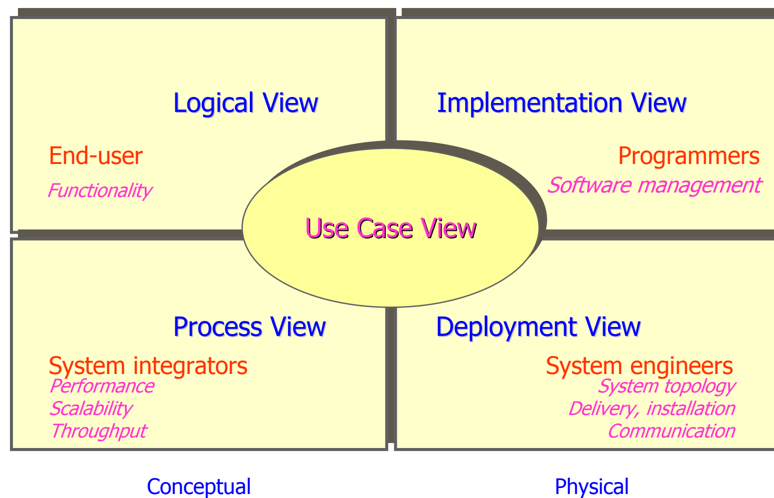
the pattern of the interconnection of the system's components



Software Architecture

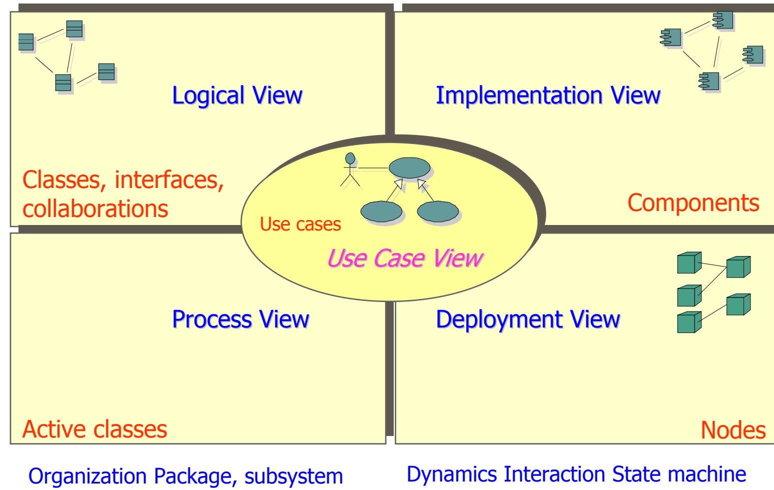
- Software architecture encompasses the set of significant decisions about the organization of a software system
  - composition of the structural and behavioral elements into larger subsystem
    - structural elements and their interfaces by which a system is composed
    - behavior as specified by the interaction or patterns among those elements
  - architectural style/pattern that guides this organization

Representing System Architecture



Conceptual Physical

Architecture and the UML



Organization Package, subsystem

Dynamics Interaction State machine

The importance of software architecture

- To enable everyone to better understand the system
- To allow software developers to work on individual pieces of the system in isolation
- To prepare for extension of the system
- To facilitate reuse and reusability

Design Stable Architecture

- To ensure the maintainability and reliability of a system,
  - The architectural model of the system must be designed to be stable
  - In other words,
    - new features can be easily added
      - with only small changes to the architecture

To be a Master …

- Always study the designs of other Masters
  - There are thousands of existing design patterns, which must be
    - understood, memorized, and applied

"Each pattern describes a problem which occurs over and over again in our environment
and then describes the core of the solution to that problem,
in such a way that you can reuse this solution
a million times over,
without ever doing it the same way twice"

… Christopher Alexander

Patterns for Design Problems:
Non-functional

- How recursive, tree like structures be modeled?
- How can we reduce the interconnection between classes,
  - especially between classes that belong to different modules or subsystems?
- How can additional functionality be attached to an object dynamically?

Design with Patterns

- A description of a pattern involves four items:
  - The name of the pattern
  - The purpose of the pattern, the problem it solves
  - How we could accomplish this
  - The constraints and forces we have to consider in order to accomplish it

- **The greatest influential work on this fledging community was**

Design Patterns: Elements of Reusable Object-Oriented Software by *Gamma, Helm, Johnson, Vlissides , 1995*In recognition of their important work, these four authors are commonly and affectionately known as the "Gang of Four"

Kinds of Software Design Patterns

- Architectural Patterns
  - a fundamental structural organization for software systems
    - It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them

- Design Patterns
  - commonly recurring structure of communicating subsystems or components that solves a general design problem within a particular context
    - a scheme for refining the subsystems or components of a software system, or the relationships between them

- Idioms (coding) patterns
  - a low-level pattern specific to a programming language
  - how to implement particular aspects of components or the relationships between them using the features of the given language

Patterns: Design Principles

- All the design patterns are based on two design concepts
  1. programming to an interface
  2. using indirection or composition with delegation

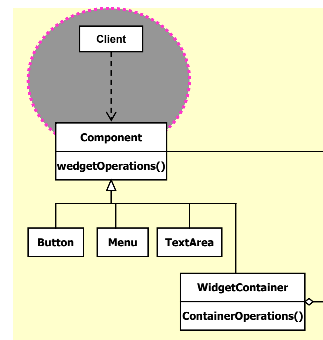
Program to an interface Not an implementation

Inheritance: *white-box* reuse

- Inheritance permits defining a family of objects with identical interfaces
  - *Polymorphism* depends on this!
- All derived classes share the base class interface
  - Subclasses extend (add) or override operations,
    - **But NOT** to block (hiding) operations
- All subclasses respond to requests in the interface of the abstract class

Inheritance for *Program to an interface*

- Clients should be **unaware** of types of objects:
  - "explicit case analysis on the type of an object is usually an error. The designer should use polymorphism".
- Clients only know about the abstract classes defining the interface.
  - : "All base classes should be abstract".
- This reduces implementation dependencies

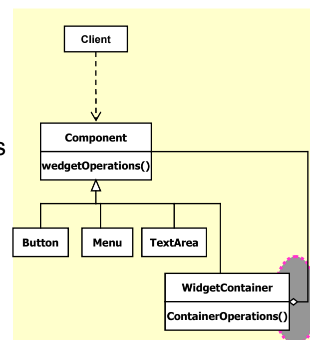


However,
Inheritance for reuse

- Parent classes often define part of their subclasses physical representation
  - Inheritance exposes the parent implementation
    - →it's said to "break encapsulation"
- Change in parent → change in subclass
- Can't change inherited implementation at run-time

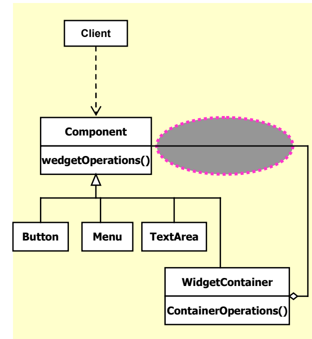
Object Composition: Black-box Reuse

- Object composition: *black-box reuse*
- In object composition
  - objects are accessed solely through their interfaces
    - no break of encapsulation
  - any object can be replaced by another at runtime
    - as long as they are the same type

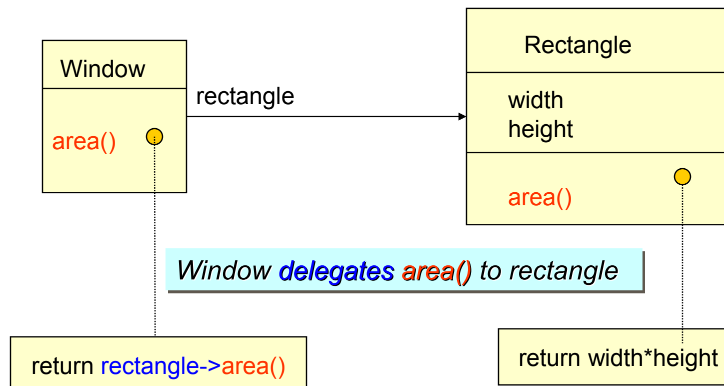


Delegation

- Delegation is "a way of making composition as powerful for reuse as inheritance" through:
  - The **receiver** passes itself to the **delegate** to let the delegated operation refer to the receiver!
  - These objects handle a request
    - analogous to subclass deferring request to parent
- Can change behaviors at run-time
    - For example: button can become text at run-time by replacing Button with TextArea, (assuming Button & TextArea are same type!)
    - However, there are run-time costs.



Delegation: *Example*



Design Pattern Template

- Intent
  - short description of pattern and its purpose
- Motivation
  - motivation scenario demonstrating pattern's use
- Applicability
  - circumstances in which pattern applies
- Structure
  - graphical representation of the pattern
- Participants
  - participating classes and/or objects and their responsibilities

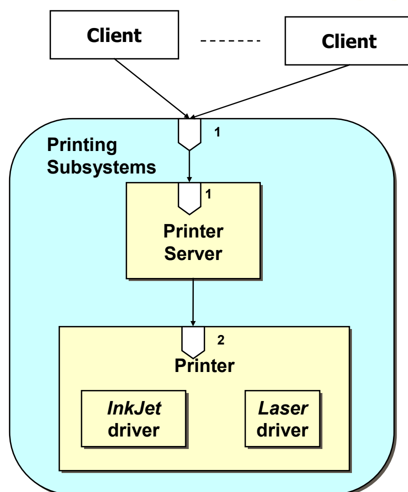
Some Reusable OO Design Patterns

- Façade
- Composite
- Adapter
- Proxy
- Observer
- Abstract Factory

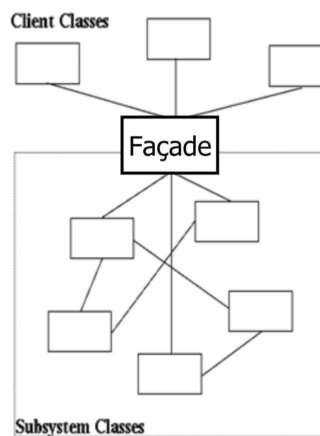
The Façade Pattern

- Provides a unified interface to a set of interfaces in a subsystem
  - Façade defines a higher-level interface
    - that makes the subsystem easier to use
    - Use a Façade object to provide a single, simplified interface
      - to the more general facilities of a subsystem
  - Common design goal
    - to minimize the communication and dependencies between subsystems
- How do you simplify the view that programmers have of a complex package?

Façade Pattern: Subsystems



Structure of the Façade Pattern

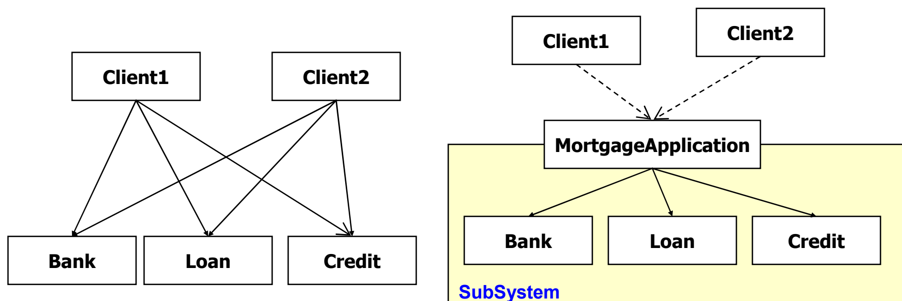


Participants of the Façade Pattern

- Façade
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem Classes
  - Implement subsystem functionality
  - Handle work assigned by the façade object
  - No need to have knowledge of the façade
    - i.e., they *keep no references* to it

Façade:

Mortgage Application Example



- Delegation is used to **bind** **MortgageApplication** with **Bank, Loan,** and **Credit**

Façade – Code Sample

```
// "SubSystem ClassA"
class Bank
{
  public boolean SufficientSavings( Customer c ) {
    System.out.println("Check bank for " + c.getName());
    return true;
  }
}
```

```
// "SubSystem ClassC"
class Loan
{
  public boolean GoodLoan( Customer c ) {
    System.out.println( "Check loan for " + c.getName());
    return true;
  }
}
```

```
// "SubSystem ClassB"
class Credit
{
  public boolean GoodCredit( int amount, Customer c ) {
    System.out.println( "Check credit for " + c.getName());
    return true;
  }
}
```


Façade – Code Sample

```
// "Facade"
class MortgageApplication {
  int amount;
  private Bank bank = new Bank();
  private Loan loan = new Loan();
  private Credit credit = new Credit();

  public MortgageApplication(int amount){this.amount = amount;}

  public boolean IsEligible( Customer c )
  {
    // Check creditworthyness of applicant
    if( !bank.SufficientSavings( c ) )
      return false;
    if( !loan.GoodLoan( c ) )
      return false;
    if( !credit.GoodCredit( amount, c ))
      return false;

    return true;
  }
}
```

```
/// Facade Client
public class FacadeApp
{
  public static void main(String[] args)
  {
    // Create Facade
    MortgageApplication mortgage =
            new MortgageApplication( 125000 );
    // Call subsystem through Facade
    mortgage.IsEligible(
          new Customer( "Gabrielle McKinsey" ) );
  }
}
```

The Adapter Pattern

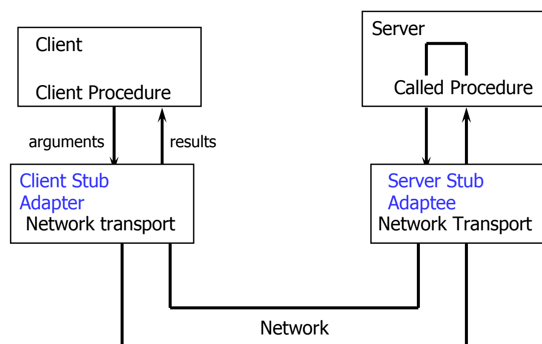
- It converts the interface of a class into another interface
  - in a way that allow us
    - to use an existing class with an interface does not match the one we need
    - to create a reusable class that cooperates with unrelated or unforeseen classes
      - i.e., classes that don't necessarily have compatible interfaces
    - to use several existing subclasses
      - but it's impractical to adapt their interface by sub-classing everyone
      - An object adapter can adapt the interface of its parent class

- How to effectively use polymorphism when reusing a class whose methods
  - have the same function
  - but *not* the same signature

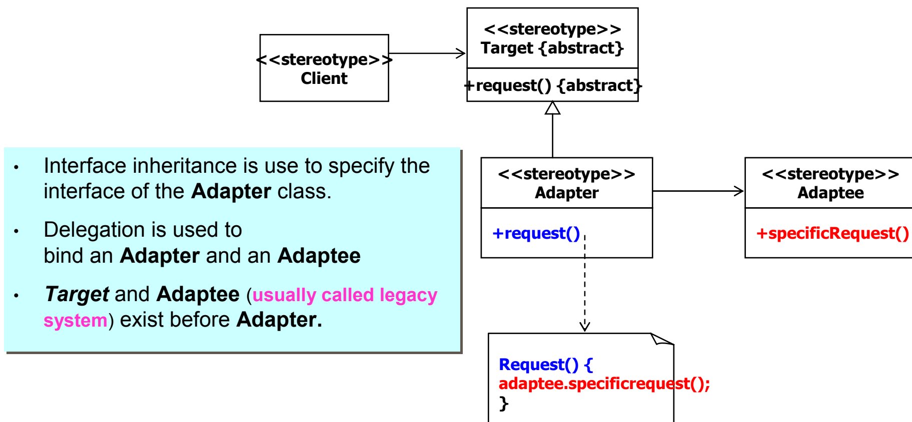
  as the other methods in the hierarchy?

Adapter Pattern

Example: Remote Procedure Call (RPC)



Adapter Pattern : The Structure



Participants of the Adapter Pattern

- Client
  - Collaborates with objects conforming to the target interface
- Target
  - Defines the application-specific interface that clients use
- Adapter
  - Adapts the interface of the adaptee to the target interface
- Adaptee
  - Defines an existing interface that needs adapting

Subsystem Design with

Façade and Adapter

- The ideal structure of a subsystem consists of
  - an interface object
  - a set of application domain objects of existing systems
    - Some of the application domain objects are interfaces to existing systems
  - one or more control objects

- use design patterns to realize this subsystem structure
  - Façade:

    Realization of the Interface Object
    - Provides the interface to the subsystem and act as controller
  - Adapter:

    Interface to existing systems
    - Provides the interface to existing system (legacy system)
    - The existing system is not necessarily object-oriented!

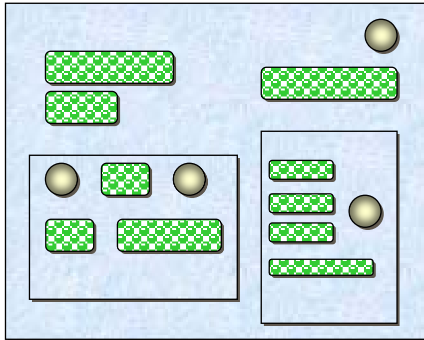
Composite Pattern

- Intent
  - to represent part-whole hierarchies of objects
  - to treat individual objects and compositions of objects uniformly
    - i.e., to be able to ignore the difference between compositions of objects and individual objects

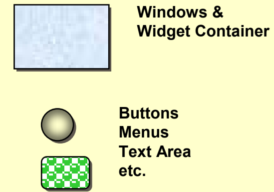
- Motivation
  - If not used, client-object must treat primitive and container classes differently
    - Thus, making the application more complex than is necessary

Consider…

Application Window

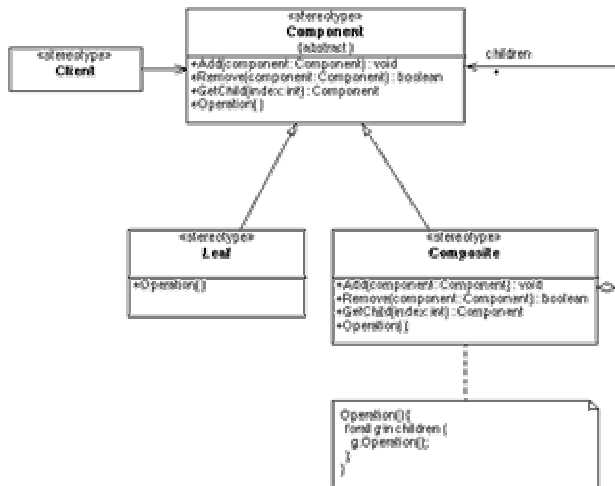


Legend



How does the window hold and deal with the different items it has to manage?

Composite Pattern: The Structure

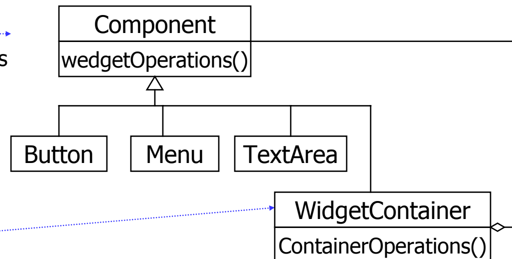


A Possible Solution

Component
implements default behavior for widgets
when possible

Button, Menu, etc.
overrides Component methods
when needed

WidgetContainer
will have to override all WidgetOperations



A container is a subclass of component, but
a container retains a list of other components that it can display inside of itself

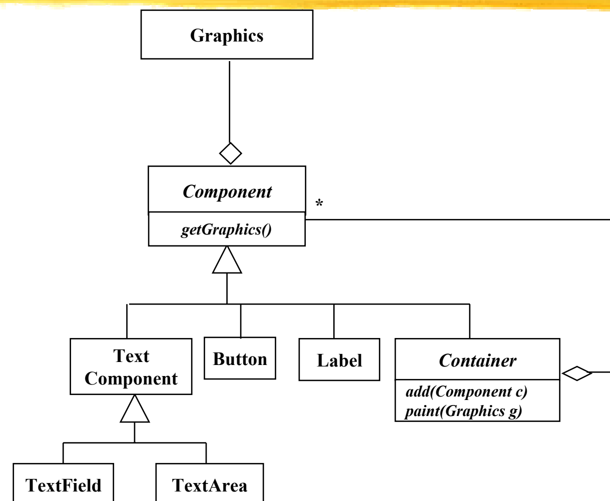
Participants of Composite Pattern

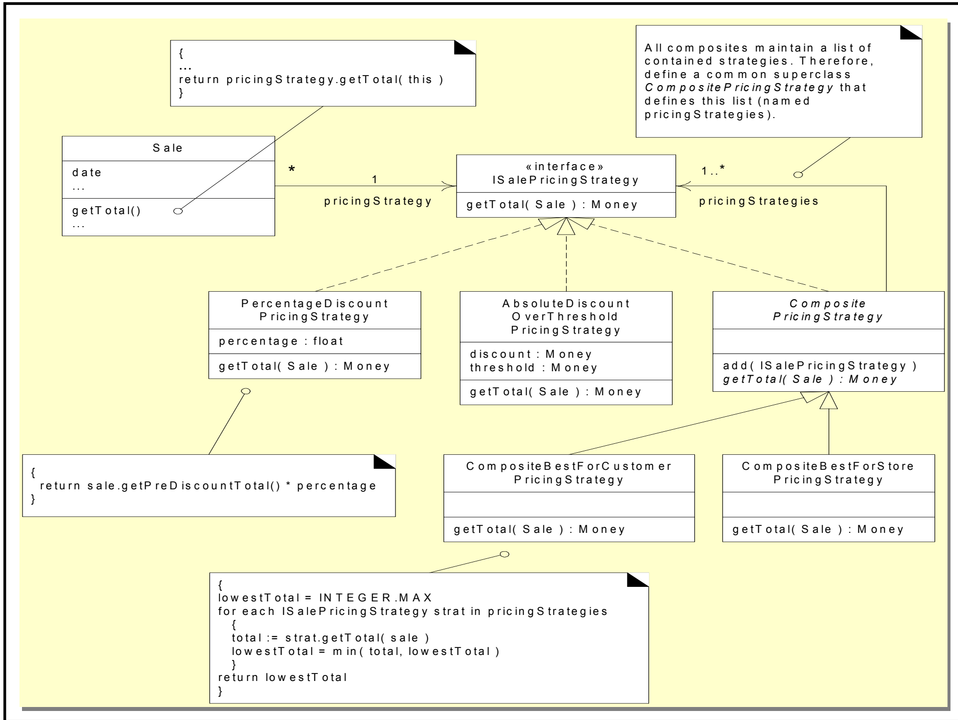
- Component
  - Declares the interface for objects in the composition
  - Implements default behavior for the interface common to all classes
- Leaf
  - Represents leaf objects in the composition
  - Defines behavior for primitive objects in the composition
- Composite
  - Defines behavior for components having children
  - Implements child-related operations in the component interface
  - Stores child components
- Client
  - Manipulates objects in the composition through the component interface

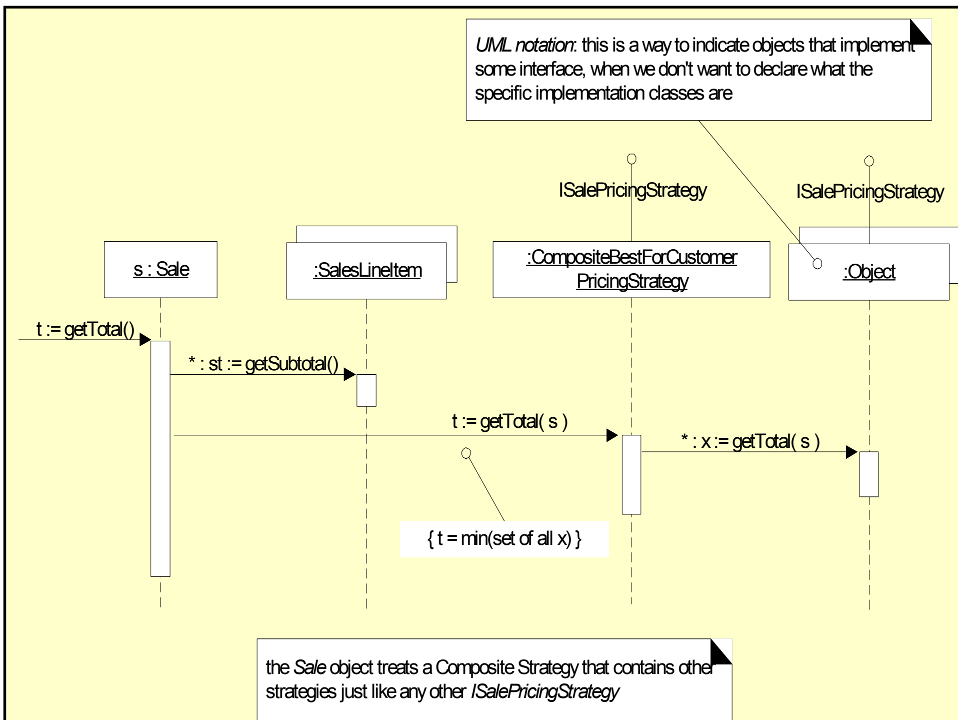
Application Examples

- File systems
  - Individual files are leaves
  - directories and subdirectories are composite files (that contain files)
- Most WEB URL's are composites that contain many other URL's
  - Some URL's are leaves
    - documents without links; images; email addresses
- Java AWT is based on the Composite pattern

Java AWT library with the component pattern




{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass CompositePricingStrategy that defines this list (named pricingStrategies).
Sale
date
...
getTotal()
...
*
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1..*
pricingStrategies
PercentageDiscount PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
{
return sale.getPreDiscountTotal() * percentage
}
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
{
total := strat.getTotal(sale)
lowestTotal = min(total, lowestTotal)
}
return lowestTotal
}

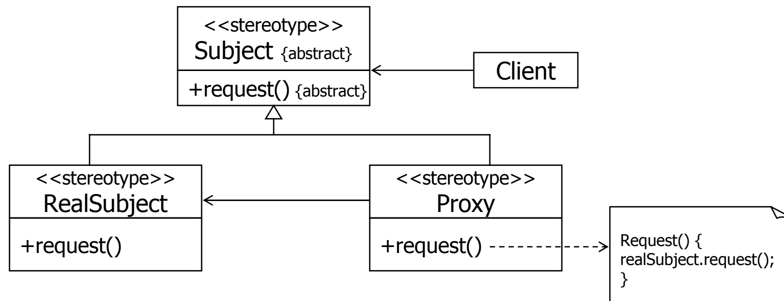

UML notation: this is a way to indicate objects that implement some interface, when we don't want to declare what the specific implementation classes are
ISalePricingStrategy
ISalePricingStrategy
s : Sale
:SalesLineItem
:CompositeBestForCustomer PricingStrategy
:Object
t := getTotal()
* : st := getSubtotal()
t := getTotal(s)
* : x := getTotal(s)
{ t = min(set of all x) }
the Sale object treats a Composite Strategy that contains other strategies just like any other ISalePricingStrategy

```
public abstract class CompositePricingStrategy implements ISalePricingStrategy {
        protected List pricingStrategies = new ArrayList();
        public void add( ISalePricingStrategy s ) { pricingStrategies.add( s ); }
        public abstract Money getTotal( Sale sale );
} // end of class

public class CompositeBestForCustomerPricingStrategy
                                         extends CompositePricingStrategy
{
        public Money getTotal( Sale sale ) {
          Money lowestTotal = new Money( Integer.MAX_VALUE );

          for( Iterator i = pricingStrategies.iterator(); i.hasNext(); )
          {
            ISalePricingStrategy strategy = (ISalePricingStrategy)i.next();
            Money total = strategy.getTotal( sale );
            lowestTotal = total.min( lowestTotal );
          }
          return lowestTotal;
        }
} // end of class
```

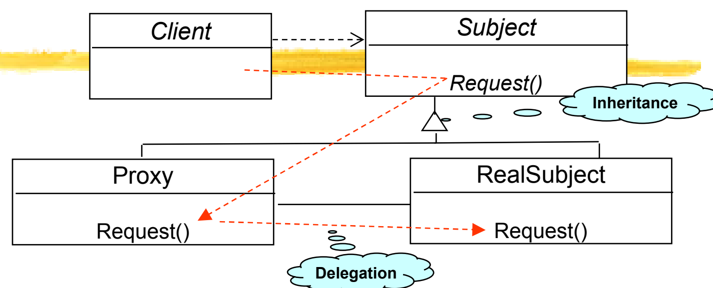

Proxy Pattern

- A proxy acts on behalf of a real object
  - *both* have the same interface
  - to improve the security of the system
    - by **checking access** before loading an object to the memory
  - to improve the performance of a system
    - by delaying expensive computations and **using memory only when needed**
  - to have a stand-in for the real object
    - **to control how** the real object behaves
- How to reduce the need to create instances of a heavyweight class?

Proxy: The Structure



Proxy:
The Structure & Behavior



- Interface inheritance is use to specify the interface of the proxy

  through class Subject

  - In Java Subject can be implemented with an interface

  Proxy is a subclass of the abstract class Subject

The Client always calls Request() in Proxy

The Implementation of Request() in class Proxy then uses *Delegation*, to access Request() in RealSubject

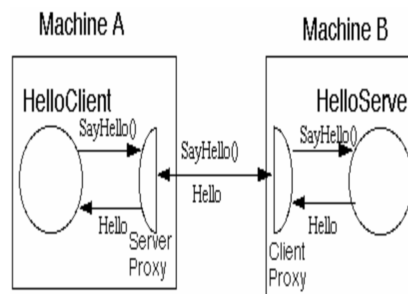
RealSubject ist also subclass of the abstract class Subject

Proxy Pattern
Applications

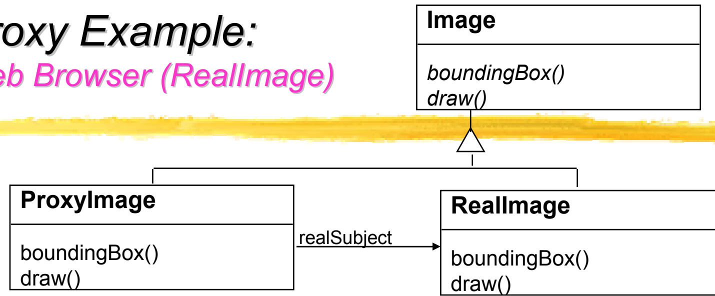
- Distributed Programming:
  - Reduction of the access cost for remote objects
  - Virtual Objects
- Authentication:
  - Checking the access rights of a caller

Categories of Proxy

- Remote Proxy
  - The actual object is on a remote machine (remote address space)
  - Hide real details of accessing the object
    - Used in CORBA, Java RMI



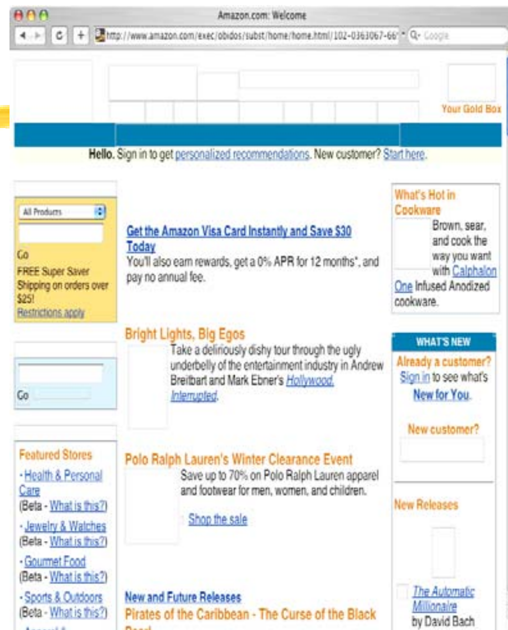
Proxy Example:
Web Browser (RealImage)



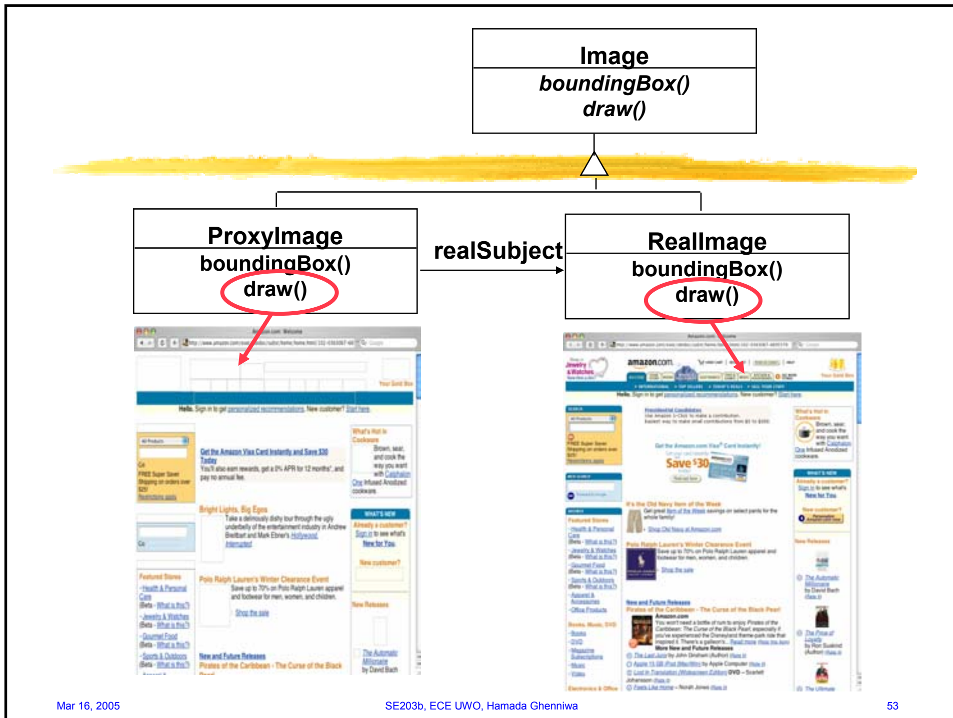
- **Images** are stored and displayed separately from **Text**
- The client cannot tell, if it is using **ProxyImage** instead of **RealImage**
- The draw() can be implemented differently in **ProxyImage** and in **RealImage**
  - **ProxyImage** draws an empty rectangle
  - **RealImage** draws the full picture

Web page access via RealImage

Web page access via *ProxyImage*



Activation of the Proxy Class



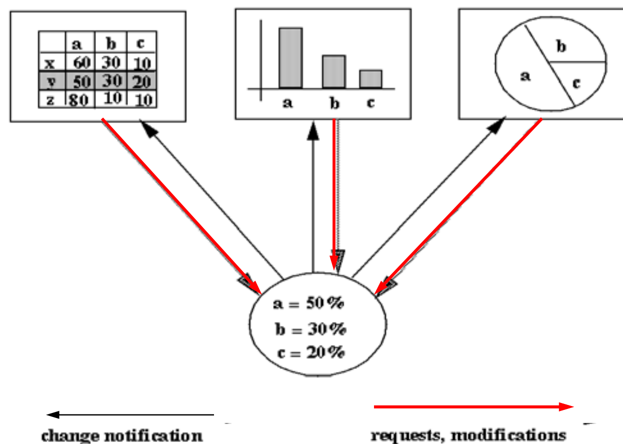
Categories of Proxy *(Cont.)*

- Virtual Proxy
  - Provides different objects with different levels of access to original object
  - Creates/accesses expensive objects on demand
    - to delay creating an expensive object until it is really accessed
- Cache Proxy (Server Proxy)
  - Multiple local clients can share results from expensive operations
    - remote accesses or long computations
- Firewall Proxy
  - Protect local clients from outside world

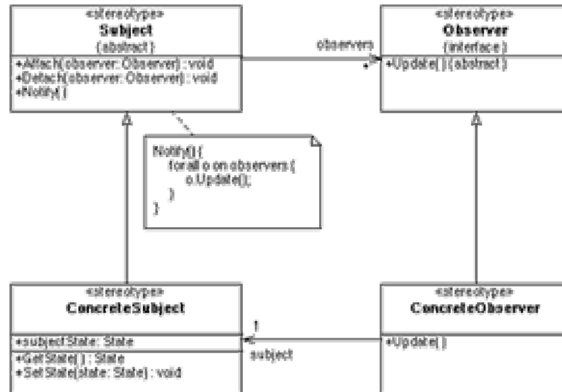
The Observer Pattern

- Defines a one-to-many dependency between objects
  - When one object changes state
    - all its dependents are notified and updated automatically
  - A common side-effect of partitioning a system into a collection of cooperating classes
    - the need to maintain consistency between related objects
      - without making the classes tightly coupled
- How to reduce the interconnection between classes,
  - especially between classes that belong to different modules or subsystems

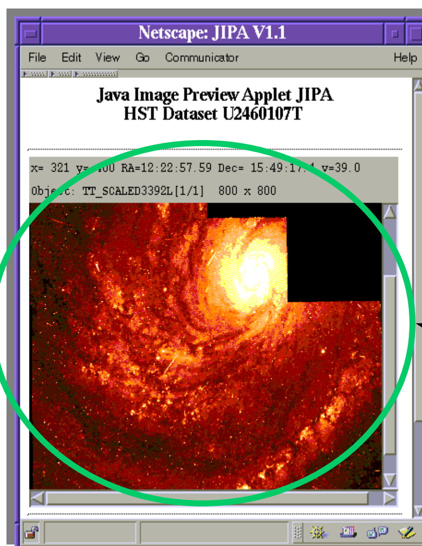
Observer Pattern: Example



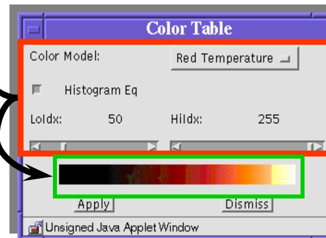
Structure of the Observer Pattern



Example



- The dialog frame below defines a color model (red marked).
  - Whenever the color model is changed
    - the dependant colorbar as well as the image (green marked) are notified.
- Java supports this concept by introducing Observable and Observer classes



Participants of the Observer Pattern

- Subject
  - Knows its observers
  - Provides an interface for attaching and detaching observers
  - Sends a notification to its observers when its state changes
- Concrete Subject
  - Stores state of interest to concrete observers
- Observer
  - Defines an updating interface for concrete observers
- Concrete Observer
  - Maintains a reference to a concrete subject object
  - *Stores state* that should stay consistent with the subject's
  - Implements the updating interface