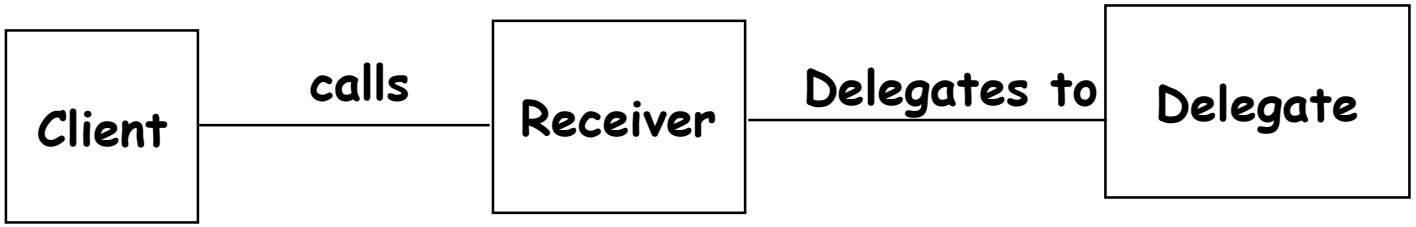
Design Patterns concluded

Delegation Pattern

- aka Chain-of-Responsibility (GoF, p 223)
- A receiver of a request that cannot handle the request directly passes it to another object to see if the latter can handle it,
- and so on, until some object does handle it,
- or until the end of the chain is reached.

Delegation



Delegation vs. Inheritance

Delegation

Pro:

Flexibility: Any object can be replaced at run time by the one to which it delegates.

Con:

Inefficiency: Extra step is involved.

Inheritance

Pro:

Straightforward to use

Supported by many programming languages

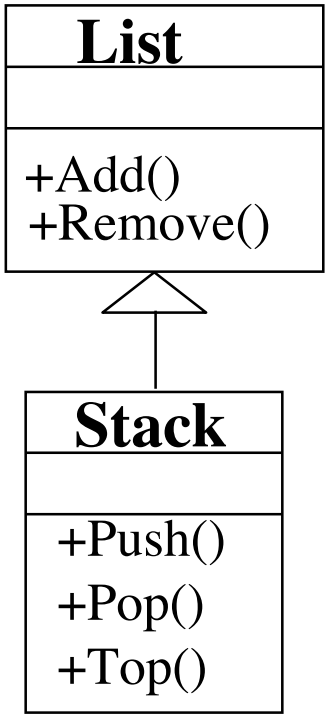
Easy to implement new functionality

Con:

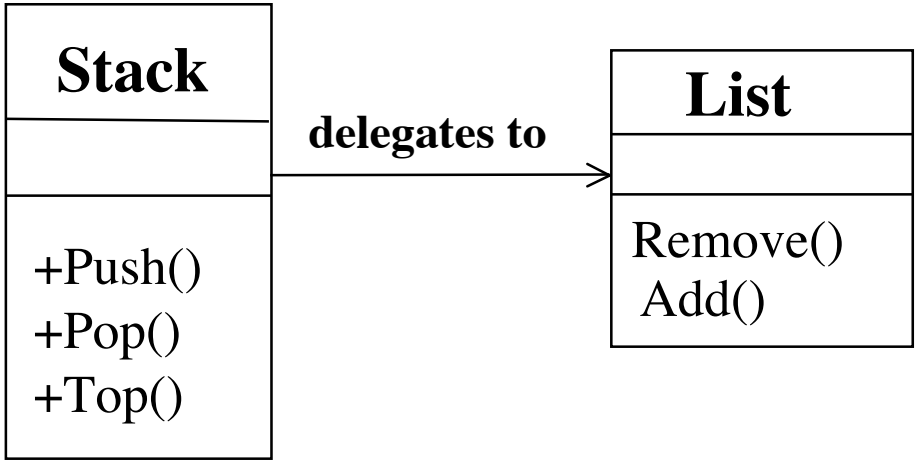
Inheritance may expose a subclass to the details of its parent class. Any change in the parent class implementation forces the subclass to change (which requires recompilation of both); See Open/Closed Principle: Don't make such changes.

Delegation vs. Inheritance

Inheritance



Delegation



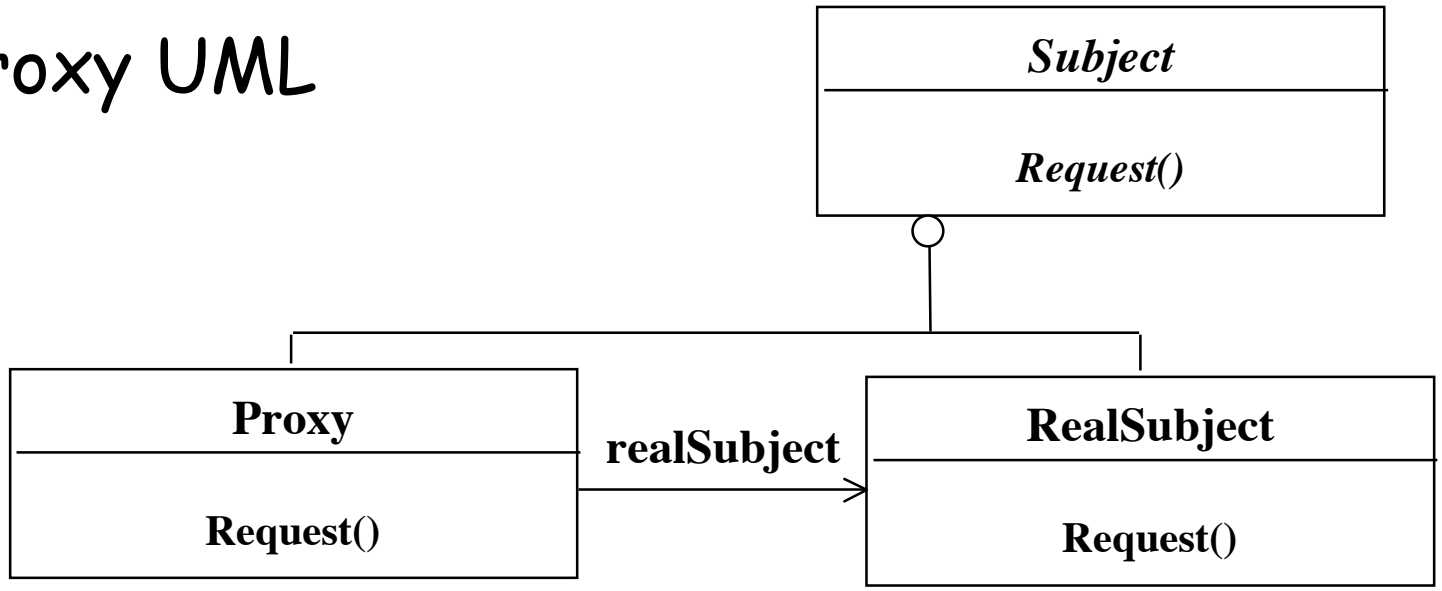
Proxy Pattern

- (GoF, p 207)
- Use a simple object as a placeholder for an expensive-to-create or very complex object.
- Visual example: An icon for a file or window

Proxy

- What is expensive?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
  - Uses another object ("the proxy") that acts as a stand-in for the real object
  - The proxy creates the real object only if the user asks for it (lazy evaluation).

Proxy UML



- Interface inheritance is used to specify the interface *shared* by **Proxy** and **RealSubject**.

- *Delegation* is used to catch accesses to the Proxy and forward them to the **RealSubject**

More Proxy Uses

- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often.
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
  - Proxy is a stand-in
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document.

Lazy Evaluation

- Sub-case of the proxy pattern
- The proxy initially has a method that is run for **creating/computing** and installing the object to which it ultimately points.
- This obviates the creation of objects that aren't used in a particular run.
- This provides added economy: the creation is done either:
  - once, if needed, or
  - not at all

Lazy Evaluation Example

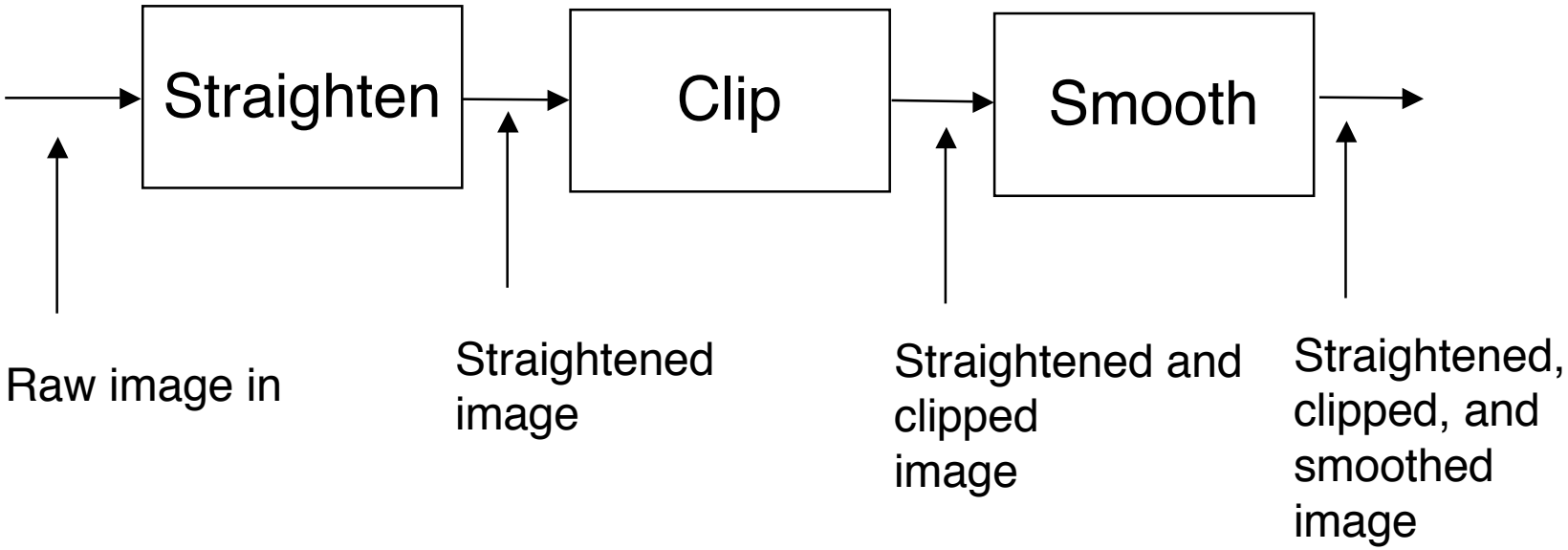
- Copying a file:
  - If the file is only being **read**, there is no need to make a separate copy.
  - Making a copy can be **deferred** until one of the users *modifies* the file; the actual copying is triggered by an **attempted** modification.

Lazy Evaluation

- aka Caching, Memoization [*sic*]
- Applicable if the method is a true function on its arguments(no side-effects).
- The returned value of a method, for arguments on which it is called, is remembered, and returned immediately when there is a subsequent call to the function for those same arguments.

Stream Pattern (not GoF)

- aka "pipes and filters"
- Process data in streams, demanding elements as needed
- Use lazy-evaluation in stream construction



Flyweight Pattern

- (GoF, p 196)
- Create light-weight objects that represent **shareable** big objects.

Flyweight Example

- In a word processor, rather than replicate information about chunks of text, such as:
  - Font family
  - Font size
  - Font attributes (boldness, italic, etc.)
  - Color

  in every chunk
- create a package of that particular set of information and share the packages among chunks.
- (This package could be identified as a "style".)

Interpreter Pattern

- (GoF, p 243)
- Define an interpreter for a language
- Sub-abstractions:
  - Command-line interpreter
  - S-expression interpreter
    - Reusable parser

Exercise

- How might an interpreter make good use of the Composite Pattern?

Memento Pattern

- Remember previous states of a sub-system, e.g. for purpose of implementing ***undo*** functionality.
- **Checkpoints** state using *Memento objects*.
- Only the sub-system can use a Memento; the using application cannot examine them internally.

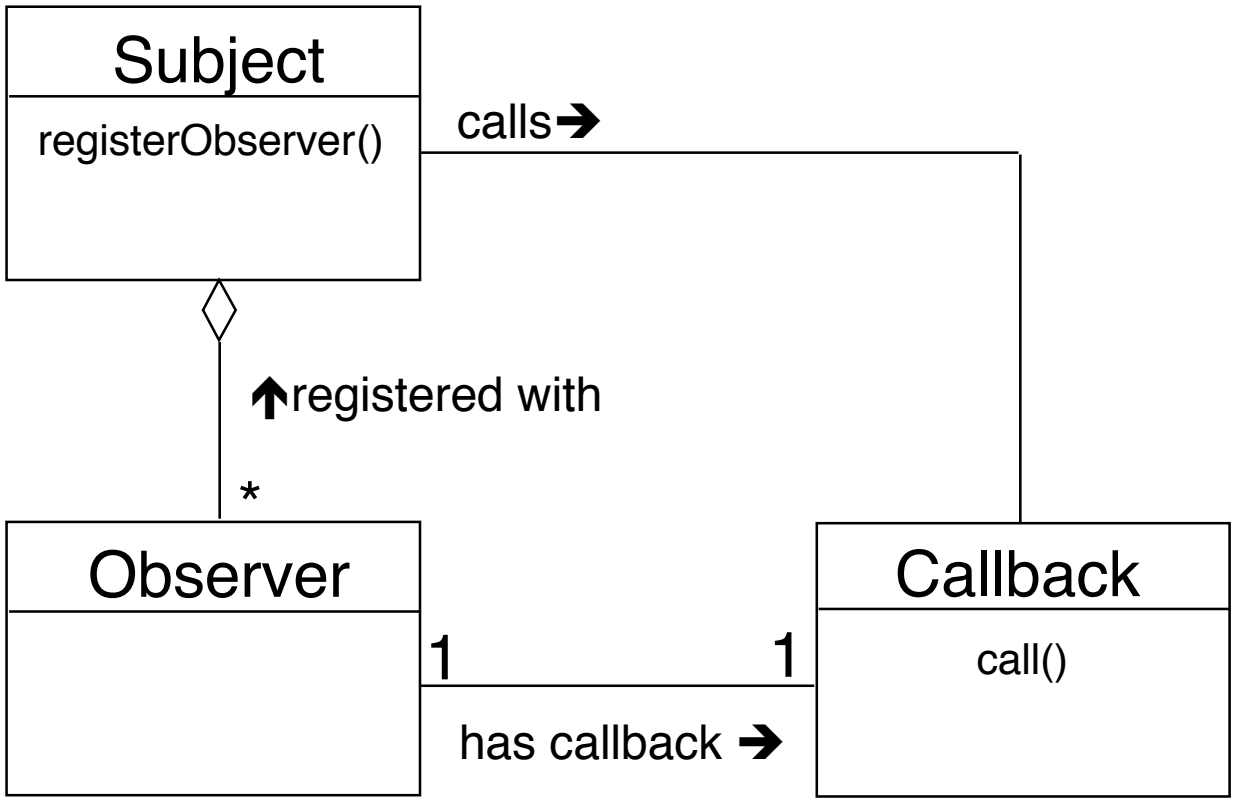
Observer Pattern

- (GoF, p 293)
- aka Listener, Callback, Publish-Subscribe
- **Register** observer objects with an subject to be observed.
- Whenever specified types of changes in the object occur, **all observers are notified** by running a *pre-specified* method on them.
- This method is sometimes called a **callback**.

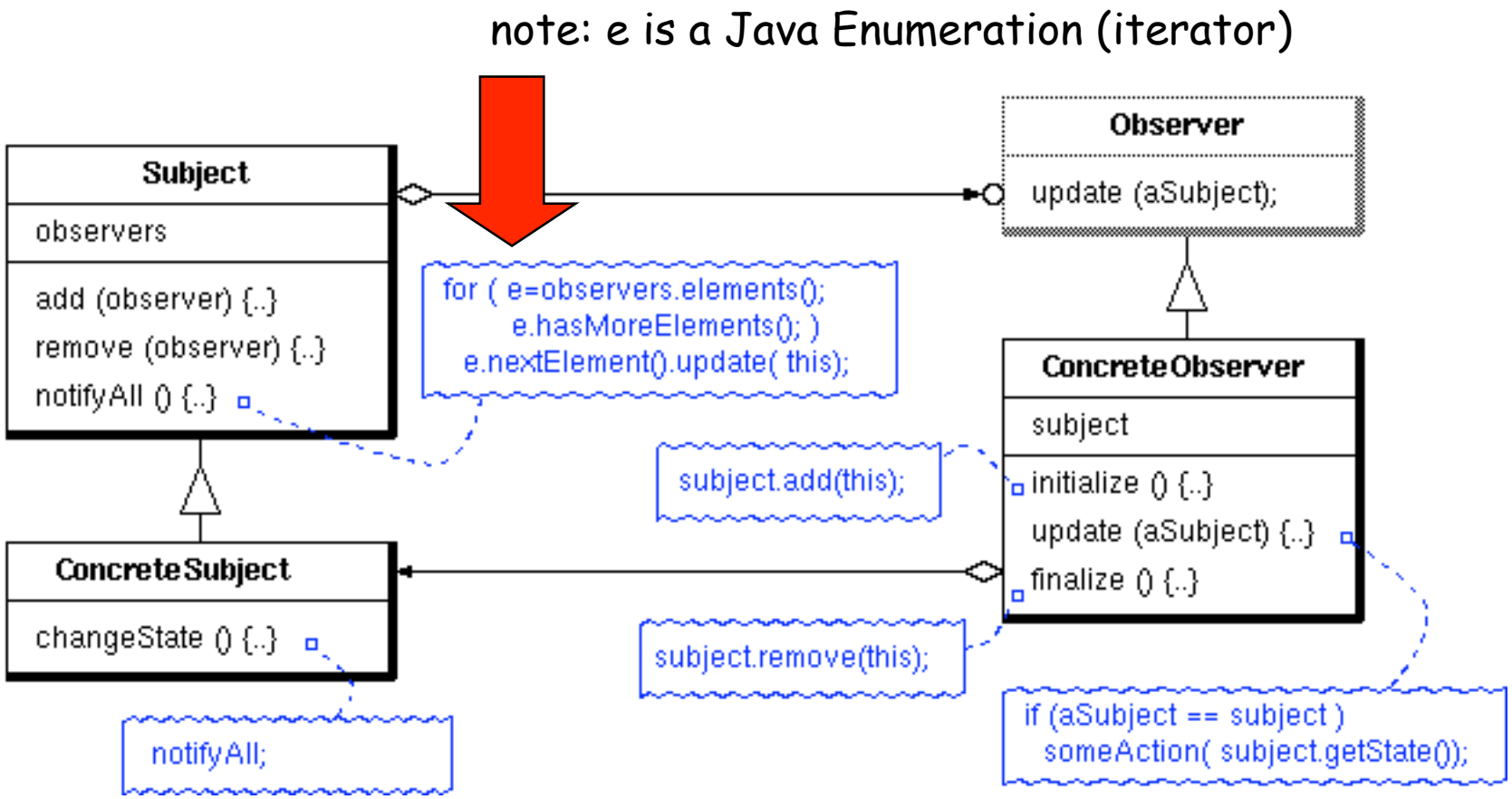
Observer Usage

- Simulation programs
- Real-time monitoring programs (e.g. involving sensors, timers)

Observer UML



More Detailed Observer for Java



Java 1.2 Observable Base class

void **addObserver**(Observer o)

Adds an observer to the set of observers for this object, provided that it is not already in the set.

void **notifyObservers**()

If this object has changed, as indicated by hasChanged(), then notify all of its observers and then call clearChanged() to indicate that this object has no longer changed.

protected void **setChanged**()

Marks this Observable object as having been changed; hasChanged() will now return true and notifyObservers() will be called.

protected void **clearChanged**()

Indicates that this object is regarded as not having changed, or that it has already notified all of its observers of its most recent change.

Java 1.2 Observer Interface

public void **update**(Observable observable, Object arg)

Called by an Observable on which this Observer is registered.

Example in Java (1)

```
import java.util.Observable;
import java.util.Observer;

class MyObservable extends Observable
{
private int value = 0;

public void setValue(int value)
  {
  this.value = value;
  setChanged();
  notifyObservers("new value = " + value);
  }

public int getValue()
  {
  return value;
  }
}
```

both needed

```
class MyObserver implements Observer
{
int index;

MyObserver(int index)
  {
  this.index = index;
  }


public void update(Observable observable,
                   Object arg)
  {

System.out.println("Observed by MyObserver "
                   + index + ": " + arg);
  }
```

interface

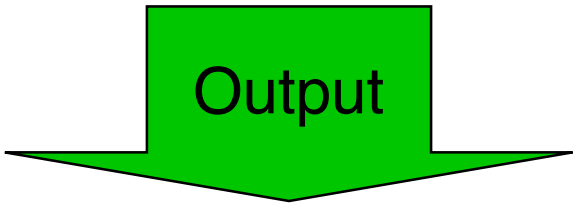
required method

Example in Java (2)

```
public static void main(String arg[])
 {
 MyObservable observable = new MyObservable();

 for( int i = 0; i < Parameters.numObservers; i++ )
  {
  observable.addObserver(new MyObserver(i));
  }

 observable.setValue(1);
 observable.setValue(2);
 observable.setValue(3);
 }
```




```
Observed by MyObserver 4: new value = 1
Observed by MyObserver 3: new value = 1
Observed by MyObserver 2: new value = 1
Observed by MyObserver 1: new value = 1
   …
Observed by MyObserver 2: new value = 3
Observed by MyObserver 1: new value = 3
Observed by MyObserver 0: new value = 3
```

Observer Variants

- **Push** variant: When updated, the subject **passes** all relevant **information** to the observer's callback.

- **Pull** variant: The subject merely **notifies** the observer, which then **extracts** the needed **information** from the subject.

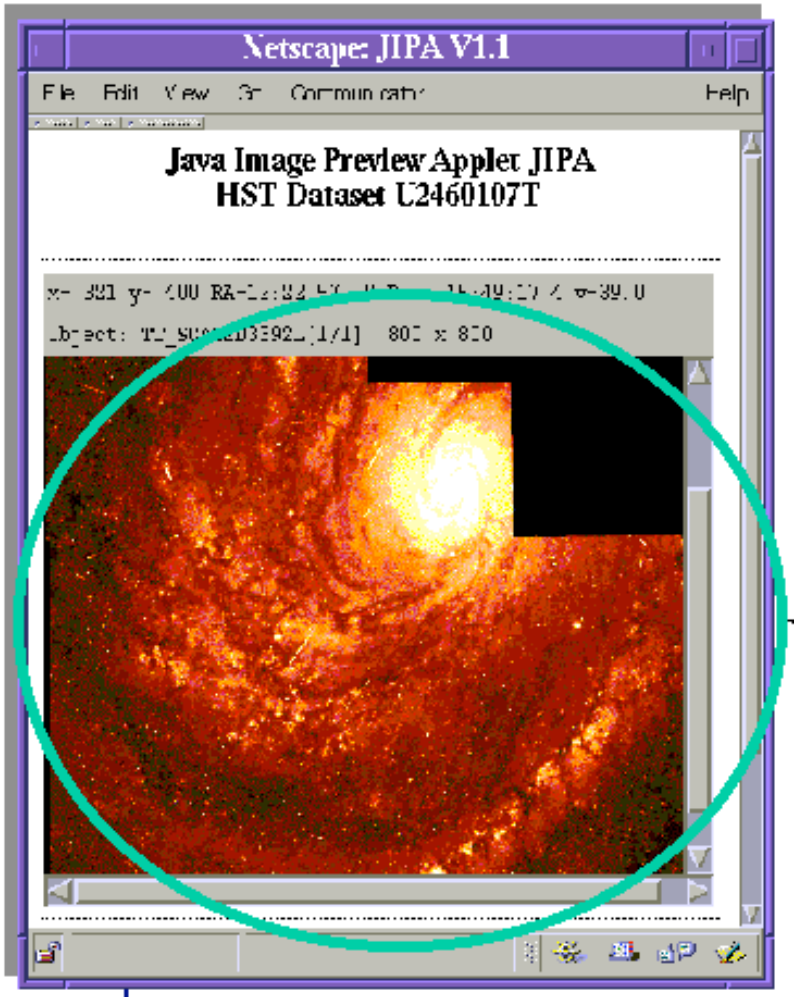
Model-View-Controller Pattern (pre-dated GOF)

- Separates graphical-user-interface applications into three non-overlapping parts:
  - The **Model**, governing only the ***content*** of what's being displayed,
  - The **View**, which defines ***how*** the information is displayed, and
  - The **Controller**, which deals with user interactions (mouse, keyboard, etc.)

MVC vs. Observer

- How can the Model-View-Controller pattern exploit the Observer pattern?

Observer Pattern in Action



Intent

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and update automatically [Sch98].

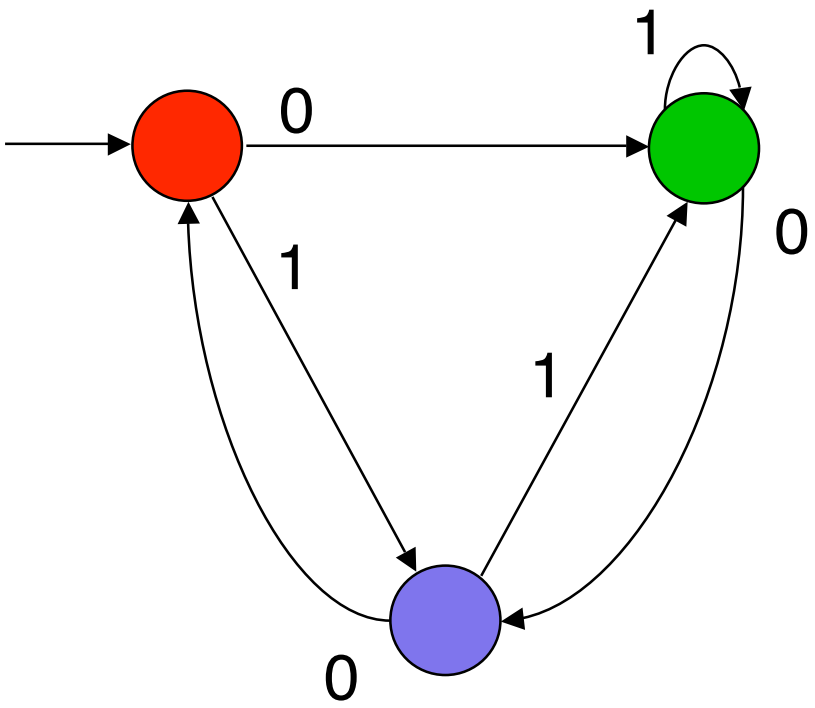
Example

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

State Pattern

- In a state-table based system, use a **different class for each state**, to reflect state-dependent behavior,
- rather than encoding all behavior into one class with a state variable.

Describe a Finite-State Machine using State pattern



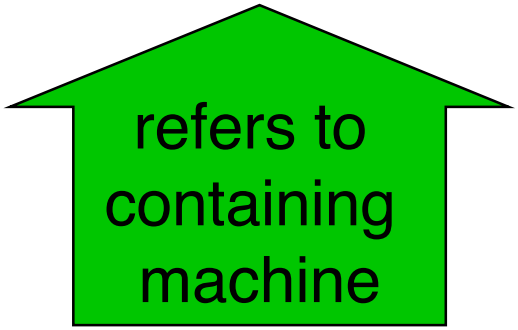

```
class Machine
{
protected State current;
public handle (int input)
 {
 current = current.handle(input);
 }
}
```

dispatch

```
class MyMachine extends Machine
{
State red = new Red(this);
State blue = new Blue(this);
State green = new Green(this);
current = red;
}
```

State Classes

```
class State
{
protected Machine machine;
}
```




```
class Red extends State
{
Red( Machine m) { this.machine = machine; }

State handle(int input)
 {
 switch( input )
  {
   case 0: return machine.blue;
   case 1: return machine.red;
  }
 }
}
```

similarly for Blue, Green

Command Pattern

- Similar idea to State pattern, except that a different class is used for each kind of command.

Factory-Method Pattern

- (GoF, p 107)
- A **Factory-method** is a method is used to create other objects (as opposed to directly calling the **constructor** for the objects).

- The factory method might produce objects of several *different* derived classes, if desired.

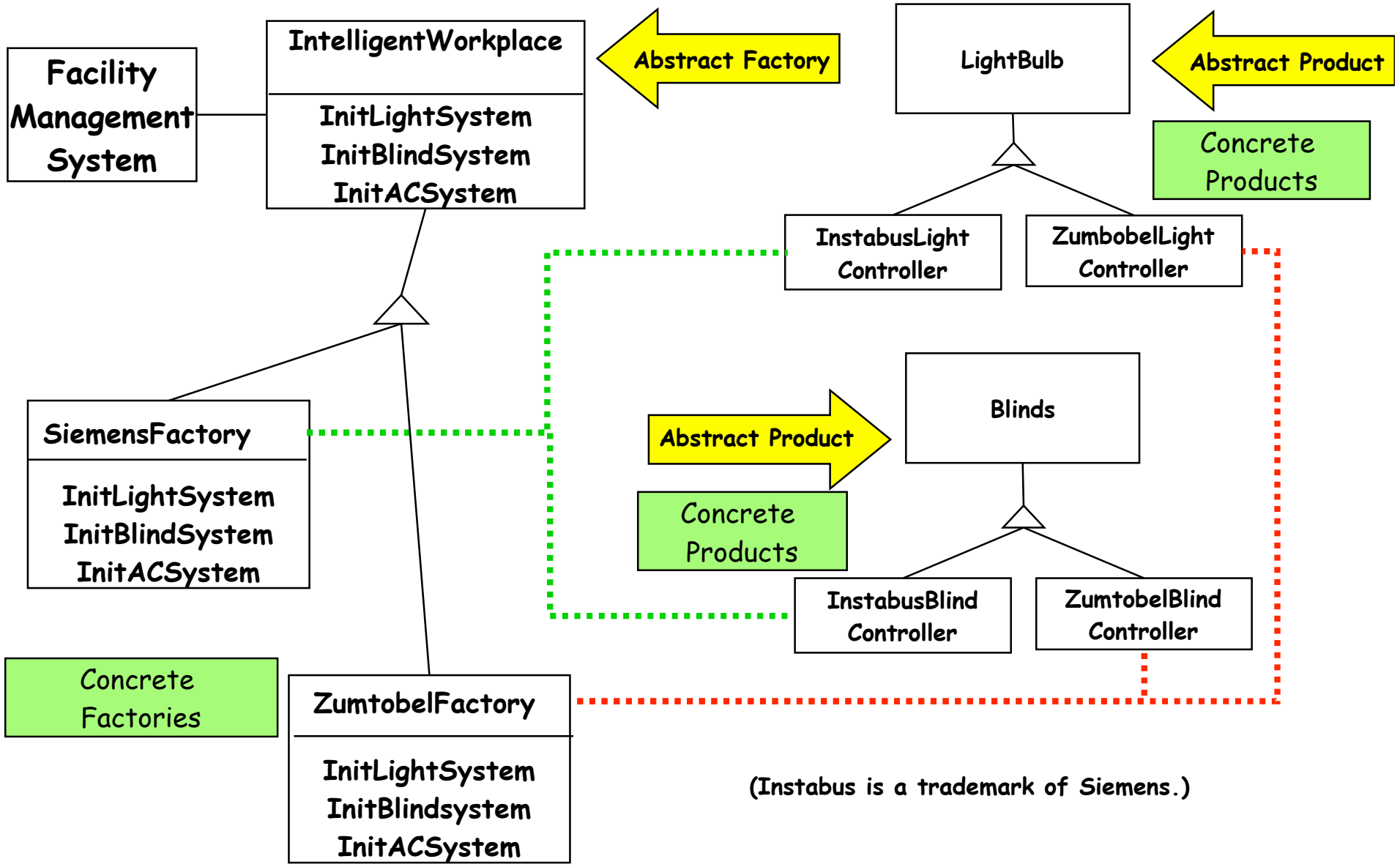
Abstract Factory Pattern

- (GoF, p 87)
- Provide an **interface for creating** families of related objects without specifying their concrete classes.
- All objects created are derived from the same *abstract* class.

Abstract Factory Pattern

- Consider a facility management system for an **intelligent house** that supports different control system families
- How can you construct a single control system factory that is independent from the manufacturer?

OWL System for the The Intelligent Workplace at Carnegie Mellon University



(Instabus is a trademark of Siemens.)

Applicability for Abstract Factory Pattern

- Independence from initialization or representation:
  - The client system should be independent of how its products are created, composed or represented.
- Manufacturer Independence:
  - A system must be configured with one of multiple families of products.
  - You want to provide a class library for a customer ("facility management library"), but you don't want to reveal what particular product you are using.
- Constraints on related products:
  - A family of related products is designed to be used together and you need to enforce this constraint.
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

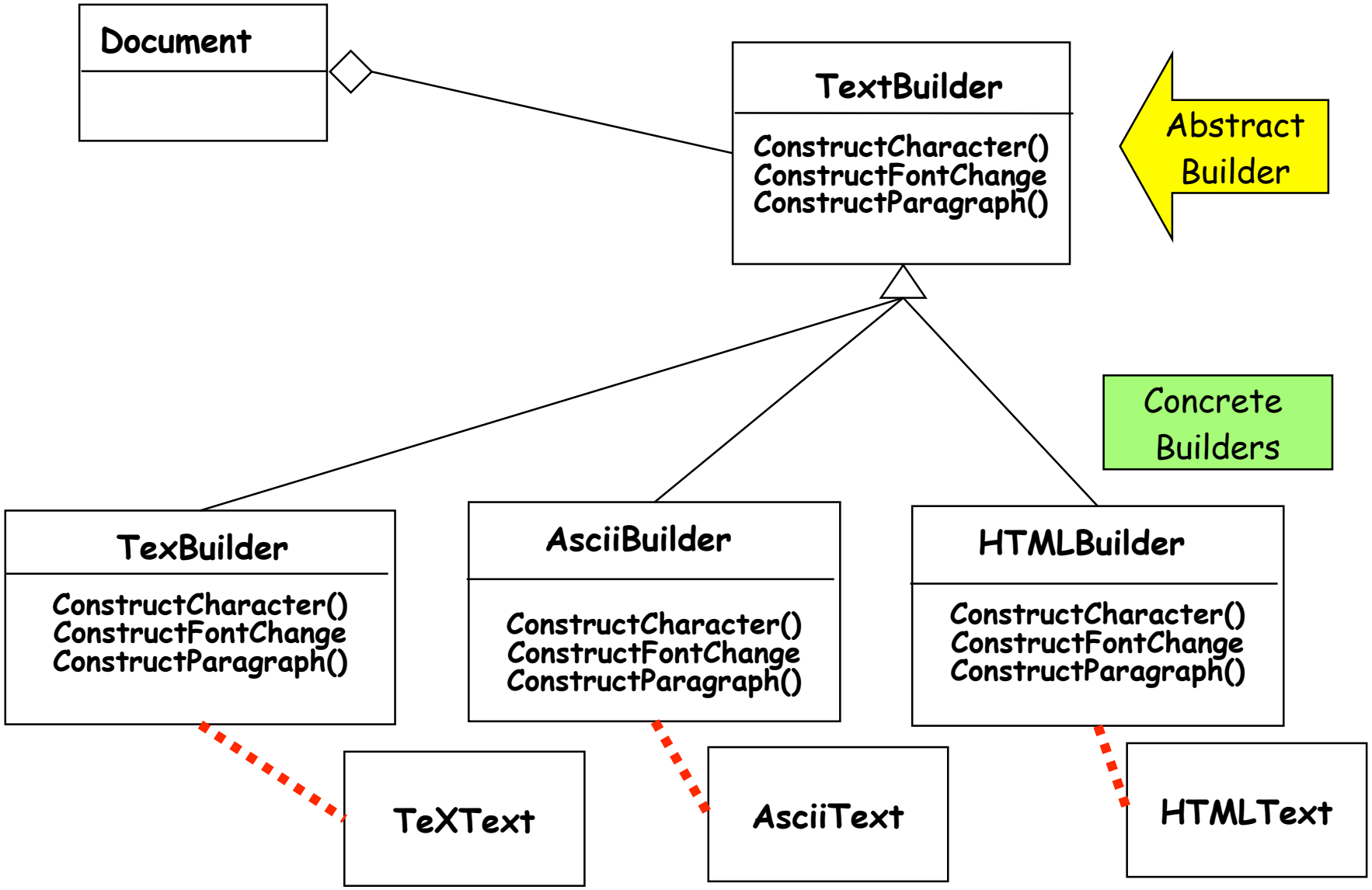
Builder Pattern

- (GoF, p 97)
- Separate the construction of a complex object from its **representation**, so that the same construction interface can create different representations.
- The different representations would be derived from a **common base class**.

Builder Pattern Example

- Consider an application that creates textual documents.
- We may wish to render these documents in numerous formats (MS Word, RTF, TeX, PostScript, FrameMaker, HTML, …)
- Constructing general Constructors between such formats is expensive.
- Instead, build the document "in the abstract" and have specific builder methods for each format.

Example UML


Document
TextBuilder
ConstructCharacter()
ConstructFontChange
ConstructParagraph()
Abstract Builder
Concrete Builders
TexBuilder
ConstructCharacter()
ConstructFontChange
ConstructParagraph()
AsciiBuilder
ConstructCharacter()
ConstructFontChange
ConstructParagraph()
HTMLBuilder
ConstructCharacter()
ConstructFontChange
ConstructParagraph()
TeXText
AsciiText
HTMLText

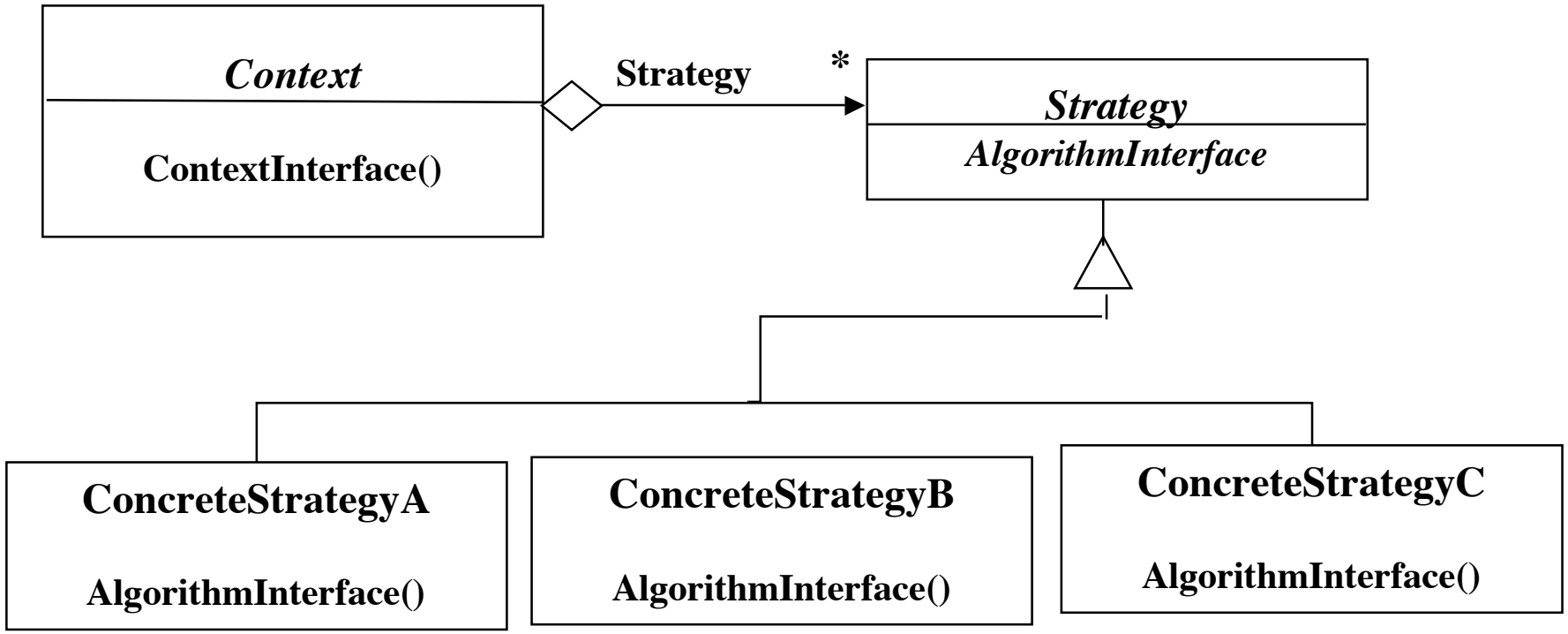
Prototype Pattern

- (GoF, p 117)
- Create objects by **cloning** prototype, then possibly specializing the resulting object, as opposed to direct call of constructor.
- Reduces the number of parameters that need to be passed to a constructor.

Strategy Pattern

- (GoF, p 315)
- Encapsulate each of a family of **algorithms** for solving the same problem
- Provide a *common abstract interface* or base class.
- Related: Template-Method Pattern (GoF, p 325), for **methods** used in algorithms.

Strategy Pattern



Classifying Patterns

- **Creational**: Patterns involving creating objects
- **Structural**: Patterns involving creating relationships between objects
- **Behavioral**: Patterns involving the functions that objects perform

Pattern Taxonomy

- **Creational** Patterns
  - Abstract the instantiation process.
  - Make a system independent from the way its objects are created, composed and represented.
- **Structural** Patterns
  - Adapters, Bridges, Facades, and Proxies are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce an abstract class to enable future extensions
    - Encapsulate complex structures
- **Behavioral** Patterns
  - Concerned with algorithms and the assignment of responsibilies between objects: Who does what?
  - Characterize complex control flow that is difficult to follow at runtime.

Exercise: Unscramble the following Classifications

- **Creational**
  - Builder
  - Command
  - Composite
  - Factory-Method
  - State
  - Strategy
  - Visitor
- **Structural**
  - Abstract Factory
  - Adapter
  - Bridge
  - Façade
  - Memento
  - Observer
  - Proxy
- **Behavioral**
  - Decorator
  - Delegation
  - Flyweight
  - Interpreter
  - Prototype
  - Singleton

Additional

- See course web page for links to source code for GoF, etc.
- For on-line book that has patterns with code examples, see:
  - www.mindview.net/Books/TIPatterns/
- For a different taxonomy, see
  - http://wwwipd.ira.uka.de/~tichy/patterns/overview.html