Software Design Patterns

Lecture 12

Based on slides from Marty Stepp, University of Washington

And Jesse Hartloff, cse.buffalo.edu

Design Challenges

- Designing software for reuse is hard. One must find:
  - a good problem decomposition, and the right software
  - a design with flexibility, modularity and elegance
- Designs often emerge from trial and error
- Successful designs do exist
  - two designs they are almost never identical
  - they exhibit some recurring characteristics
- Can designs be described, codified or standardized?
  - this would short circuit the trial and error phase
  - produce "better" software faster

Design Patterns

- A pattern provide a solution to a common software problem in a context
  - describes a recurring software structure
  - is abstract from programming language
  - identifies classes and their roles in the solution to a problem
  - patterns are not code or designs; must be instantiated/applied

- example: Iterator pattern
  - The Iterator pattern defines an interface that declares methods for sequentially accessing the objects in a collection.
  - Recall Collections in Java

Benefits of Using Patterns

- Common design vocabulary
  - allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation
  - embodies a culture; domain-specific patterns increase design speed
- Capture design expertise and allow that expertise to be communicated
  - promotes design reuse and avoid mistakes
  - Makes it easier for other developers to understand a system.
- Improve documentation (less is needed)
  - patterns are described well once

Design Patterns

- Not really needed for course projects
  - Small projects
  - Requirements don't change
  - No design patterns needed
  - Can get away without proper OO usage
- Necessary when the projects are large
- Large projects with poor design
  - Do you want to refactor 10,000 lines of code?
  - 100,000 lines?
- Design is HARD and patterns can capture lessons learned.

Gang of Four

- Gamma, Helm, Johnson, and Vlissides
  - Wrote seminal book on design patterns:
    Design Patterns: Elements of Reusable Object-Oriented Software
  - Describe 23 different (classic) design patterns, dividing them into three different classes of patterns
    - *Creational design patterns*: Dealing with when and how objects are created, these patterns typically create objects for you, relieving you of the need to instantiate those objects directly.
    - *Structural design patterns*: Describe how objects are composed into larger groups.
    - *Behavioral design patterns*: Generally talk about how responsibilities are distributed in the design and how communication happens between objects.

Gang of Four (GoF) patterns

- **Creational Patterns**
  (abstracting the object-instantiation process)
  - *Factory Method* Abstract Factory *Singleton*
  - Builder Prototype

- **Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - *Adapter* Bridge Composite
  - *Decorator* Facade Flyweight
  - Proxy

- **Behavioral Patterns**
  (communication between objects)
  - Command Interpreter *Iterator*
  - Mediator *Observer* State
  - *Strategy* Chain of Responsibility Visitor
  - Template Method

Describing a pattern

- *Problem:* In what situation should this pattern be used?
- *Solution:* What should you do? What is the pattern?
  - describe details of the objects/classes/structure needed
  - should be somewhat language-neutral
- *Advantages:* Why is this pattern useful?
- *Disadvantages:* Why might someone not want this pattern?

Let’s focus on these (most useful?)

- Behavioral Patterns
  - Iterator
  - Strategy
  - Observer pattern
- Structural Patterns
  - Adapter Pattern
  - Decorator Pattern
- Creation Patterns
  - Singleton
  - Factory Method Pattern

Gang of Four (GoF) patterns

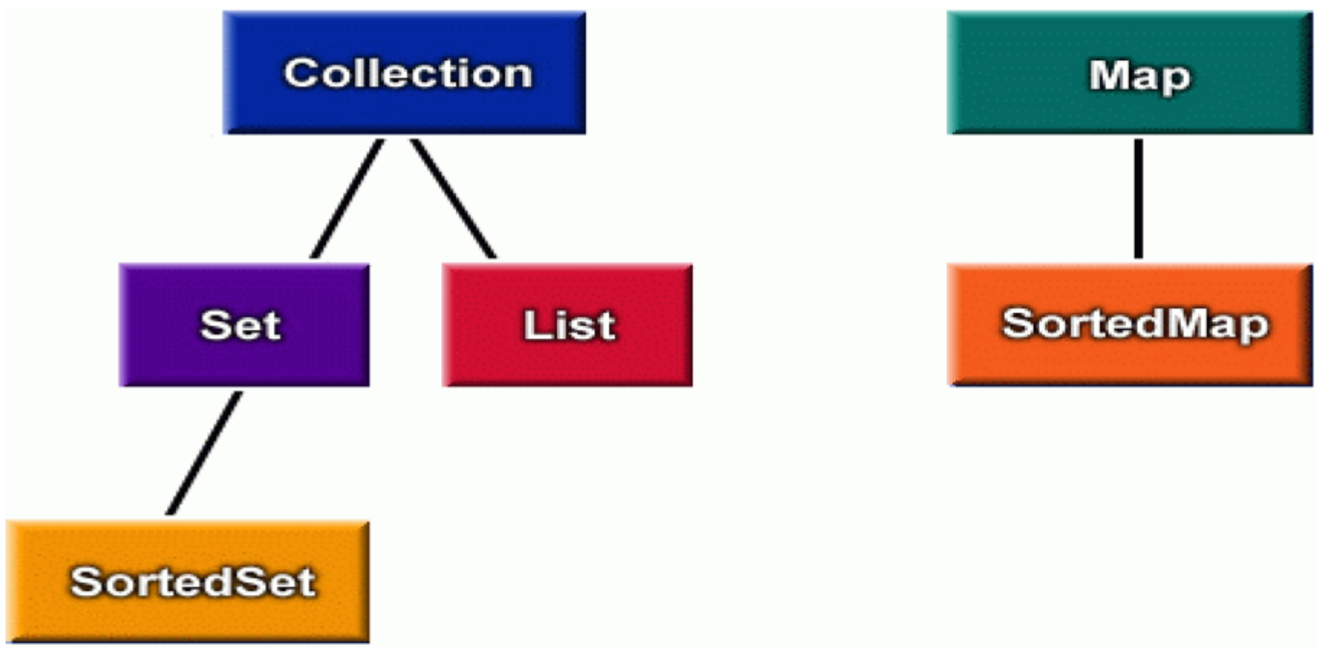
- **Creational Patterns**
  (abstracting the object-instantiation process)
  - *Factory Method* Abstract Factory *Singleton*
  - Builder Prototype

- **Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - *Adapter* Bridge Composite
  - *Decorator* Facade Flyweight
  - Proxy

- **Behavioral Patterns**
  (communication between objects)
  - Command Interpreter *Iterator*
  - Mediator *Observer* State
  - *Strategy* Chain of Responsibility Visitor
  - Template Method

Pattern: Iterator

objects that traverse collections



Iterator pattern

- *Problem:* To access all members of a collection, must perform a specialized traversal for each data structure.
  - Introduces undesirable dependences.
  - Does not generalize to other collections.

- *Solution:*
  - Provide a standard *iterator* object supplied by all data structures.
  - The implementation performs traversals, does bookkeeping.
    - The implementation has knowledge about the representation.
  - Results are communicated to clients via a standard interface.

- *Disadvantages:*
  - Iteration order is fixed by the implementation, not the client.
  - Missing various potentially useful operations (add, set, etc.).

Iterator pattern

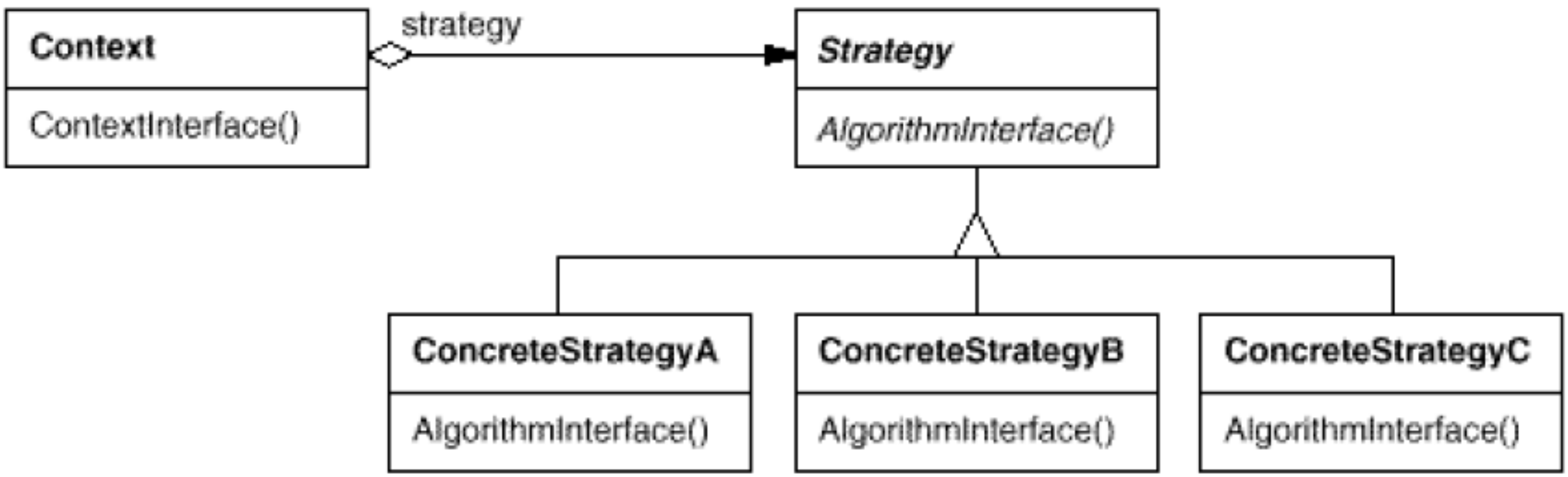
- **iterator**: an object that provides a standard way to examine all elements of any collection
  - uniform interface for traversing many different data structures
  - supports concurrent iteration and element removal

```
for (Iterator<Account> itr = list.iterator(); itr.hasNext(); ) {
    Account a = itr.next();
    System.out.println(a);
}
```

```
set.iterator()
map.keySet().iterator()
map.values().iterator()
```

Pattern: Strategy

objects that hold alternate algorithms to solve a problem



Strategy pattern

- **strategy**: an algorithm separated from the object that uses it, and encapsulated as its own object
  - each strategy implements one behavior, one implementation of how to solve the same problem
  - separates algorithm for behavior from object that wants to act
  - allows changing an object's behavior dynamically without extending / changing the object itself

- examples:
  - file saving/compression
  - layout managers on GUI containers
  - AI algorithms for computer game players

Strategy example: Card player

```
/*
 * Strategy hierarchy parent
 *(an interface or abstract class)
 */
public interface Strategy {
    public Card move();
}

// setting a strategy
player1.setStrategy(new SmartStrategy());

// using a strategy
Card p1move = player1.move();  // uses strategy
```

Pattern: Observer

objects that listen for updates to the state of others

Model and view

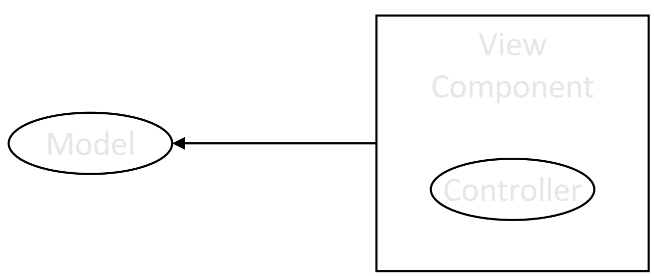
- **model**: Classes in your system that are related to the internal representation of the state and behavior of the system.
  - often part of the model is connected to file(s) or database(s)
  - examples (card game): Card, Deck, Player
  - examples (bank system): Account, User, UserList

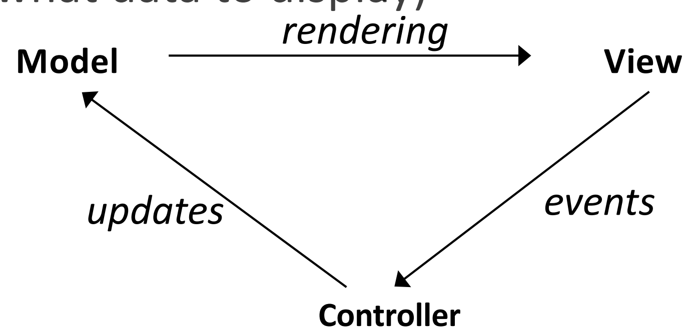
- **view**: Classes in that display the state of the model to the user.
  - generally, this is your GUI (could also be a text UI)
  - should not contain crucial application data
  - Different views can represent the same data in different ways
    - Example: Bar chart vs. pie chart
  - examples: PokerGUI, PacManCanvas, BankApplet

Model-view-controller

- **model-view-controller (MVC)**: Design paradigm for graphical systems that promotes strict separation between model and view.

- **controller**: classes that connect model and view
  - defines how user interface reacts to user input (events)
  - receives messages from view (where events come from)
  - sends messages to model (tells what data to display)





Model/view separation

- Your model classes should NOT:
  - import graphical packages (java.awt.*, javax.swing.*)
  - store direct references to GUI classes or components
  - know about the graphical classes in your system
  - store images, or names of image files, to be drawn
  - drive the overall execution of your program

- Your view/controller classes should:
  - store references to the model class(es)
  - call methods on the model to update it when events occur

- *Tricky part:* Updating all aspects of the view properly when the state of the model changes...

Observer pattern

- **observer**: An object that "watches" the state of another object and takes action when the state changes in some way.

- *Problem:* You have a model object with a complex state, and the state may change throughout the life of your program.
  - You want to update various other parts of the program when the object's state changes.

- *Solution:* Make the complex model object observable.

- **observable** object: An object that allows observers to examine it (notifies its observers when its state changes).
  - Permits customizable, extensible event-based behavior for data modeling and graphics.

Benefits of observer

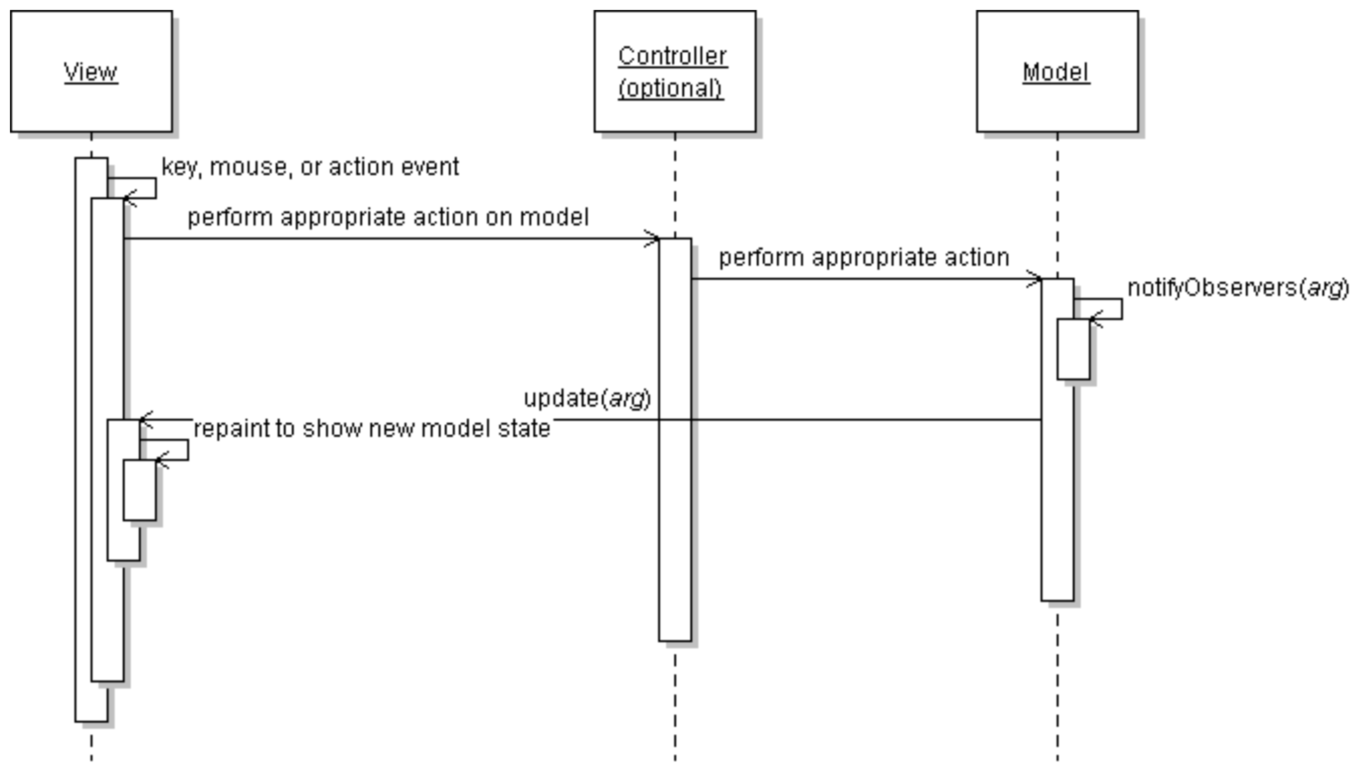
- Abstract (loose) coupling between subject and observer; each can be extended and reused individually.

- Dynamic relationship between subject and observer; can be established at run time (can "hot-swap" views, etc) gives more programming flexibility.

- Broadcast communication: Notification is broadcast automatically to all interested objects that subscribed to it.

- Can be used to implement model-view separation in Java easily.

Observer sequence diagram



Observer interface

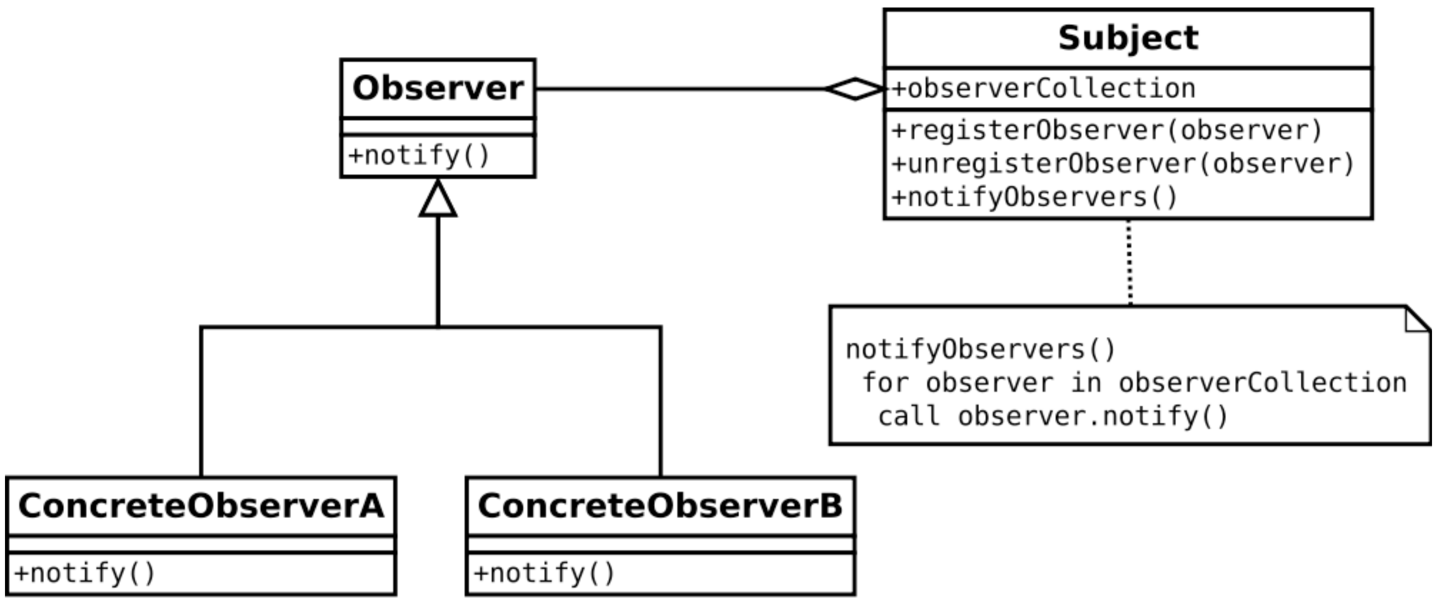
```
// import java.util.*;

public interface Observer {
    public void update(Observable o, Object arg);
}

public class Observable { ... }
```

- Basic idea:
  - Make your view code implement `Observer`.
  - Make your main model class extend `Observable`.
  - Attach the view to the model as an observer.
  - The view's `update` method will be called when the observable model changes, so write code to handle the change inside `update`.

Observable class

Method name	Description
`addObserver(`**Observer**`)`	adds an Observer to this object; its update method is called when notifyObservers is called
`deleteObserver(`**Observer**`)`	removes an Observer from this object
`notifyObservers()` `notifyObservers(`**arg**`)`	inform all observers about a change to this object; can pass optional object with more information
`setChanged()`	flags that this object's state has changed; *must* be called prior to each call to notifyObservers

Observer Pattern



Multiple views

- Make an `Observable` model.

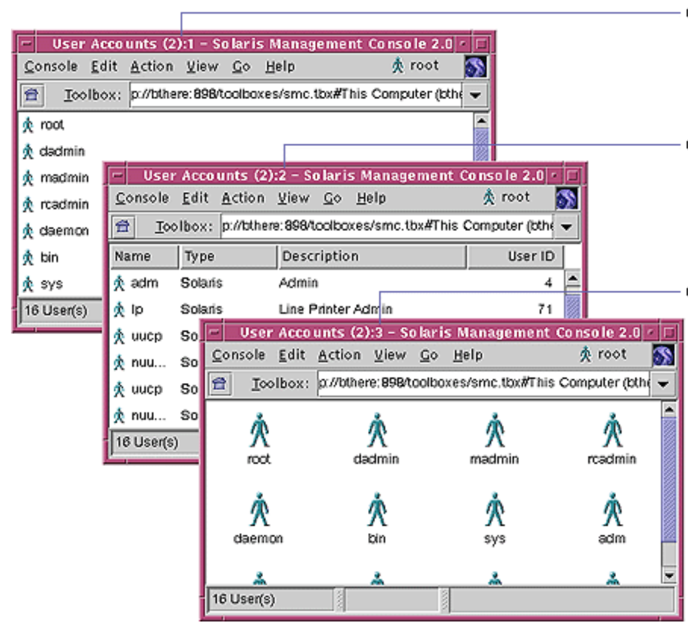
- Write an abstract `View` superclass which is a `JComponent`.
  - make `View` an observer

- Extend `View` for all of your actual views.
  - Give each its own unique inner components and code to draw the model's state in its own way.

- Provide a mechanism in GUI to set the view (perhaps via menus).
  - To set the view, attach it to observe the model.

Multiple views examples

- File explorer (icon view, list view, details view)
- Games (overhead view, rear view, 3D view)
- Graphs and charts (pie chart, bar chart, line chart)



Gang of Four (GoF) patterns

- **Creational Patterns**
  (abstracting the object-instantiation process)
  - *Factory Method*
  - Abstract Factory
  - *Singleton*
  - Builder
  - Prototype

- **Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - *Adapter*
  - Bridge
  - Composite
  - *Decorator*
  - Facade
  - Flyweight
  - Proxy

- **Behavioral Patterns**
  (communication between objects)
  - Command
  - Interpreter
  - *Iterator*
  - Mediator
  - *Observer*
  - State
  - *Strategy*
  - Chain of Responsibility
  - Visitor
  - Template Method

Pattern: Adapter

an object that fits another object into a given interface

Adapter pattern

- *Problem:* We have an object that contains the functionality we need, but not in the way we want to use it.
  - Cumbersome / unpleasant to use. Prone to bugs.

- *Example:*
  - We are given an Iterator, but not the collection it came from.
  - We want to do a for-each loop over the elements, but you can't do this with an Iterator, only an `Iterable`:

```
public void printAll(Iterator<String> itr) {
    // error: must implement Iterable
    for (String s : itr) {
        System.out.println(s);
}   }
```

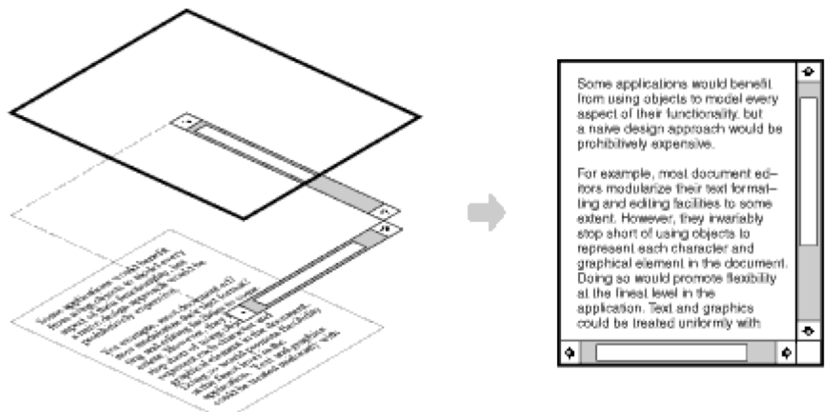
Adapter in action

- *Solution:* Create an **adapter object** that bridges the provided and desired functionality.

```
public class IterableAdapter implements Iterable<String> {
    private Iterator<String> iterator;

    public IterableAdapter(Iterator<String> itr) {
        this.iterator = itr;
    }

    public Iterator<String> iterator() {
        return iterator;
    }
}
...
public void printAll(Iterator<String> itr) {
    IterableAdapter adapter = new IterableAdapter(itr);
    for (String s : adapter) { ... }  // works
}
```

Pattern: Decorator

objects that wrap around other objects to add useful features



Decorator pattern

- **decorator**: an object that modifies behavior of, or adds features to, another object
  - decorator must maintain the common interface of the object it wraps up
  - used so that we can add features to an existing simple object without needing to disrupt the interface that client code expects when using the simple object
  - the object being "decorated" usually does not explicitly know about the decorator

- examples in Java:
  - multilayered input streams adding useful I/O methods
  - adding scroll bars to GUI controls

Decorator Pattern

- Add features by adding wrapper classes
- Outer object interacts with the world
- Original/inner object is hidden
- Especially useful when original class is in a library and lacks needed functionality

Decorator example: I/O

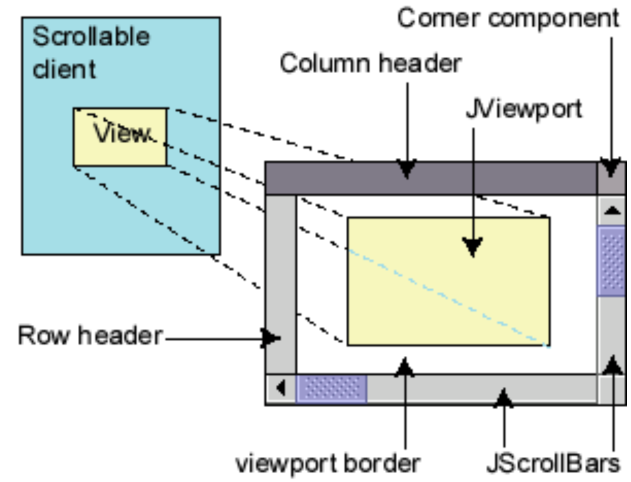
- normal `InputStream` class has only `public int read()` method to read one letter at a time
- decorators such as `BufferedReader` or `Scanner` add additional functionality to read the stream more easily

```
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);

// because of decorator streams, I can read an
// entire line from the file in one call
// (InputStream only provides public int read() )
String wholeLine = br.readLine();
```

Decorator example: GUI

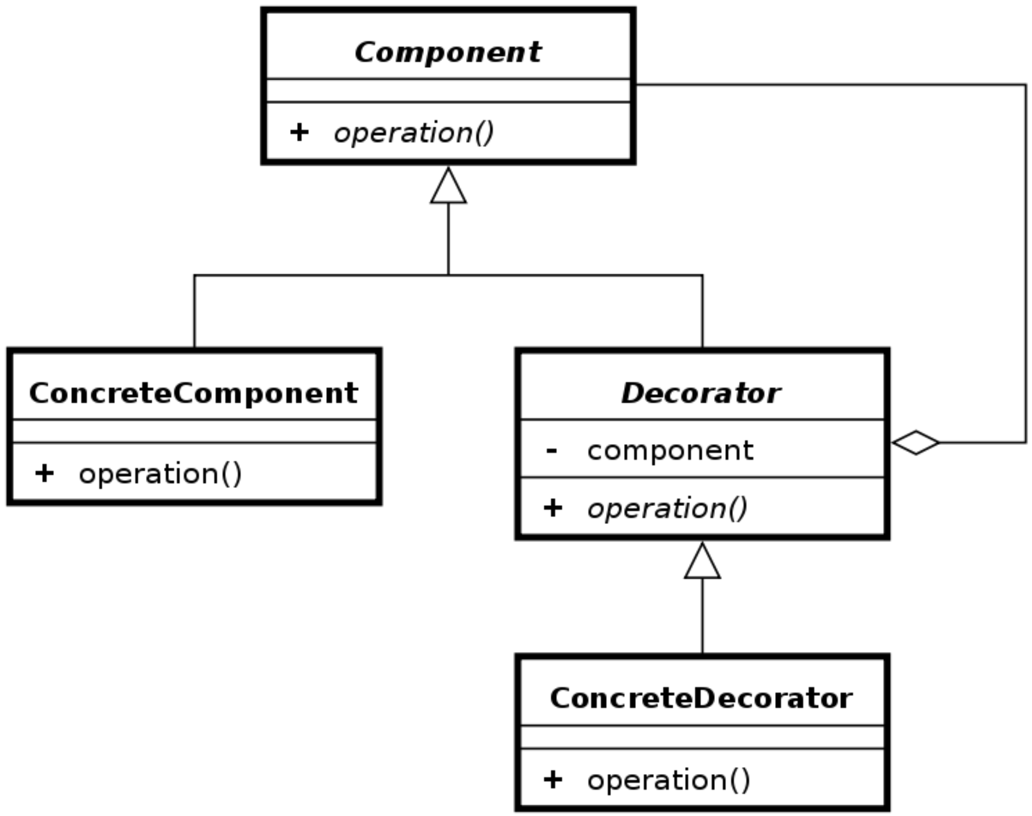
- normal GUI components don't have scroll bars
- JScrollPane is a container with scroll bars to which you can add any component to make it scrollable

```
// JScrollPane decorates GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane = new JScrollPane(area);
contentPane.add(scrollPane);
```

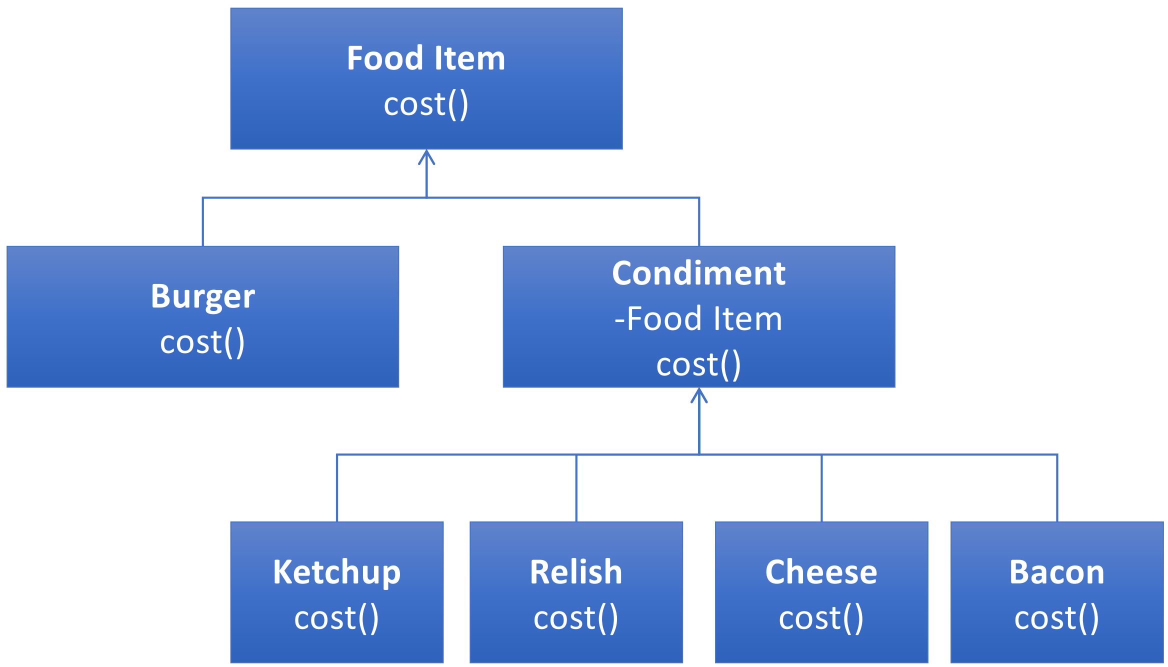


  - JComponents also have a `setBorder` method to add a "decorative" border. Is this another example of the Decorator pattern? Why or why not?

Decorator Pattern



Decorator Pattern



Gang of Four (GoF) patterns

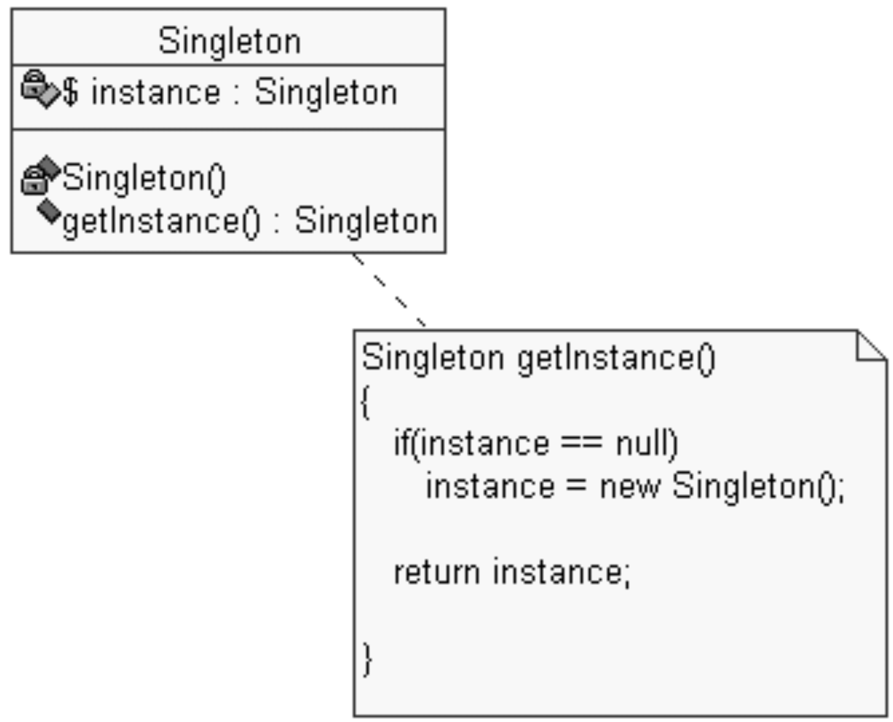
- **Creational Patterns**
  (abstracting the object-instantiation process)
  - *Factory Method* Abstract Factory *Singleton*
  - Builder Prototype

- **Structural Patterns**
  (how objects/classes can be combined to form larger structures)
  - *Adapter* Bridge Composite
  - *Decorator* Facade Flyweight
  - Proxy

- **Behavioral Patterns**
  (communication between objects)
  - Command Interpreter *Iterator*
  - Mediator *Observer* State
  - *Strategy* Chain of Responsibility Visitor
  - Template Method

Pattern: Singleton

a class that has only one instance



Restricting object creation

- *Problem:* Sometimes we really only ever need (or want) one instance of a particular class.
  - Examples: keyboard reader, bank data collection, game, UI
  - We'd like to make it illegal to have more than one.

- *Issues:*
  - Creating many objects can take a lot of time.
  - Extra objects take up memory.
  - It is a pain to deal with different objects floating around if they are essentially the same.
  - Multiple objects of a type intended to be unique can lead to bugs.
    - What happens if we have more than one game UI, or account manager?

Singleton pattern

- **singleton**: An object that is the only object of its type.
  (one of the most known / popular design patterns)

  - Ensuring that a class has at most one instance.
  - Providing a global access point to that instance.
    - e.g. Provide an accessor method that allows users to see the instance.

- *Benefits:*
  - Takes responsibility of managing that instance away from the programmer (illegal to construct more instances).
  - Saves memory.
  - Avoids bugs arising from multiple instances.

Restricting objects

- One way to avoid creating objects: use static methods
  - Examples: `Math`, `System`
  - Is this a good alternative choice? Why or why not?

- *Disadvantage*: Lacks flexibility.
  - Static methods can't be passed as an argument, nor returned.

- *Disadvantage*: Cannot be extended.
  - Example: Static methods can't be subclassed and overridden like an object's methods could be.

Implementing Singleton

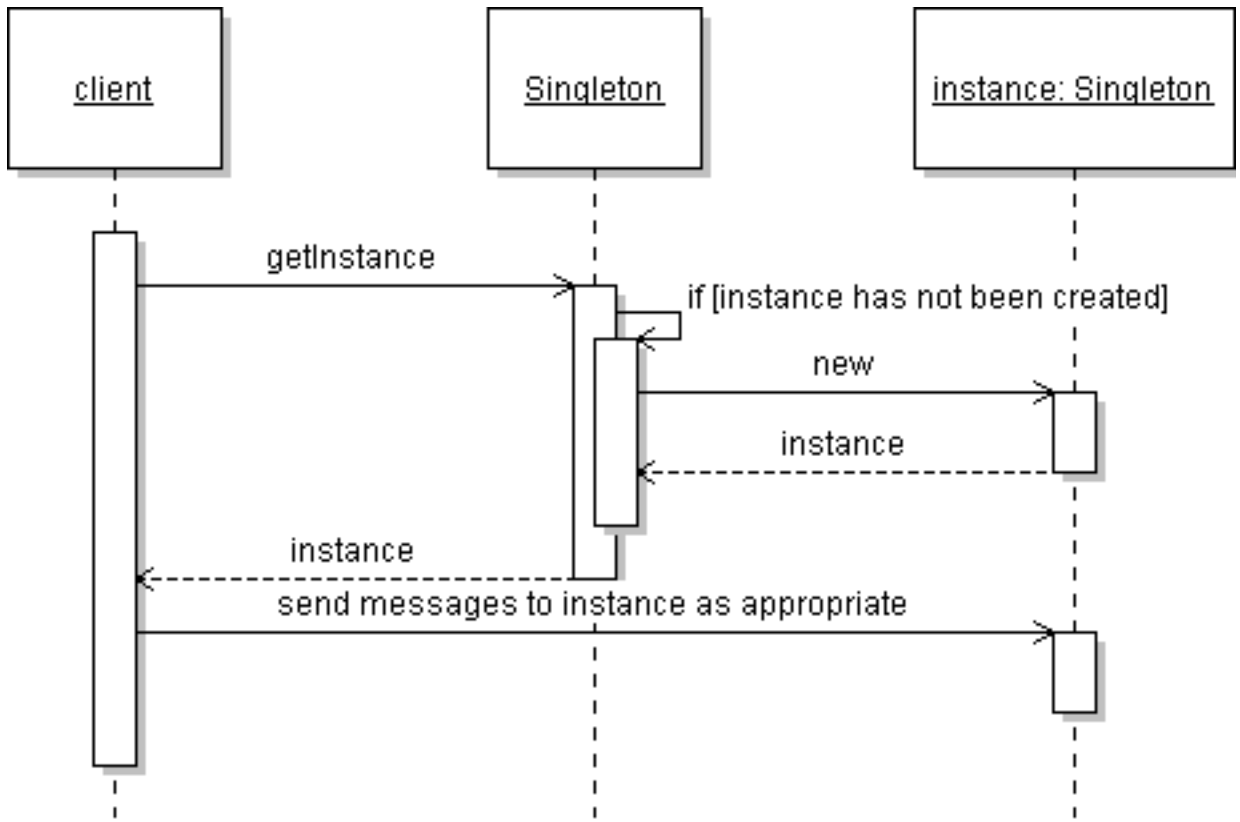
- Make constructor(s) `private` so that they can not be called from outside by clients.
  - I.e., Client *Doesn't use* new to do the instantiation
  - Instantiate the object from within the class definition itself!

- Declare a single `private static` instance of the class.

- Write a public `getInstance()` or similar method that allows access to the single instance.

  - May need to protect / synchronize this method to ensure that it will work in a multi-threaded program.

Singleton sequence diagram



Singleton example

- Class `RandomGenerator` generates random numbers.

```
public class RandomGenerator {
    private static final RandomGenerator gen =
              new RandomGenerator();

    public static RandomGenerator getInstance() {
        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

- Possible problem: always creates the instance, even if it isn't used

Singleton example 2

- variation: don't create the instance until needed

```
// Generates random numbers.
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static RandomGenerator getInstance() {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }

    ...
}
```

- What could go wrong with this version?

Singleton example 3

- variation: solve concurrency issue by locking

```
// Generates random numbers.
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static synchronized RandomGenerator getInstance()
  {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }

    ...
}
```

- Is anything wrong with this version?

Singleton example 4

- variation: solve concurrency issue without unnecessary locking

```
// Generates random numbers.
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static RandomGenerator getInstance() {
        if (gen == null) {
            synchronized (RandomGenerator.class) {
                // must test again -- can you see why?
                // sometimes called test-and-test-and-set (TTS)
                if (gen == null) {
                    gen = new RandomGenerator();
                }
            }
        }
        return gen;
    }
}
```

Singleton Comparator

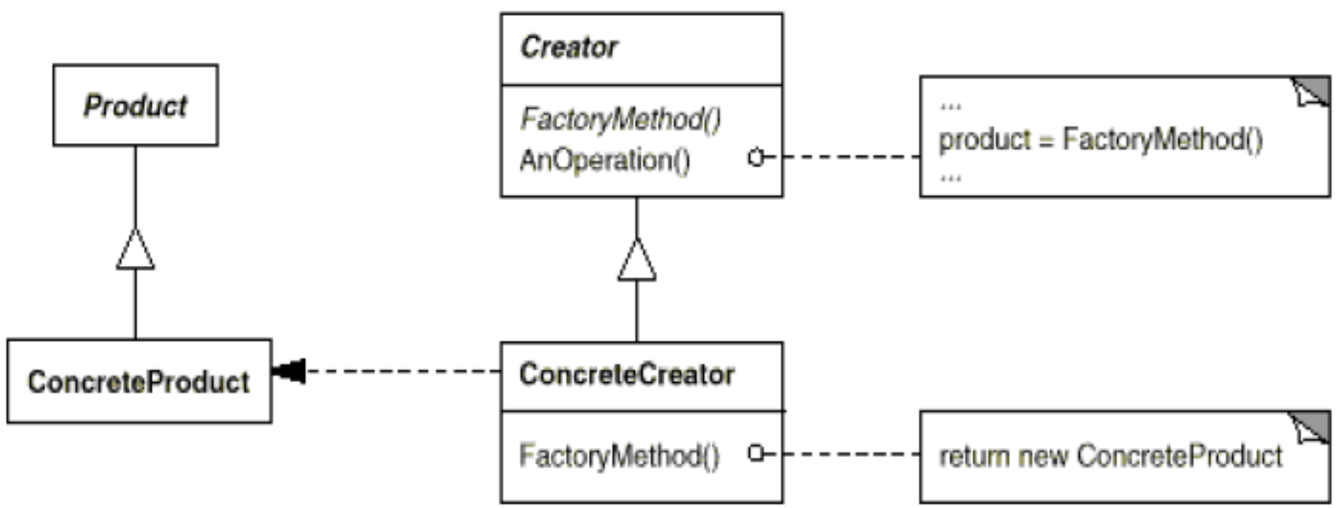
- Comparators make great singletons because they have no state:

```
public class LengthComparator
        implements Comparator<String> {
    private static LengthComparator comp = null;

    public static LengthComparator getInstance() {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }

    private LengthComparator() {}

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Pattern: Factory

(a variation of Factory Method, Abstract Factory)

a class or method used to create objects easily



Problem: Bulky GUI code

- GUI code to construct many components quickly becomes redundant (here, with menus):

```
home homestarItem = new JMenuItem("Homestar Runner");
starItem.addActionListener(this);
viewMenu.add(homestarItem);

crapItem = new JMenuItem("Crappy");
crapItem.addActionListener(this);
viewMenu.add(crapItem);
```

- another example (with buttons):

```
button1 = new JButton();
button1.addActionListener(this);
button1.setBorderPainted(false);

button2 = new JButton();
button2.addActionListener(this);
button2.setBorderPainted(false);
```

Factory pattern

- **factory**: A class whose job is to easily create and return instances of other classes.
  - a *creational* pattern; makes it easier to construct complex objects
  - instead of calling a constructor, use a static method in a "factory" class to set up the object
  - saves lines, complexity to quickly construct / initialize objects
  - examples in Java: borders (BorderFactory), key strokes (KeyStroke), network connections (SocketFactory)

Using factories in Java

- Setting borders on buttons and panels:
  - use BorderFactory class

```
myButton.setBorder(
  BorderFactory.createRaisedBevelBorder());
```

- Setting hot-key "accelerators" on menus:
  - use KeyStroke class

```
menuItem.setAccelerator(
  KeyStroke.getKeyStroke('T', KeyEvent.ALT_MASK));
```

Factory implementation

When implementing a factory of your own:

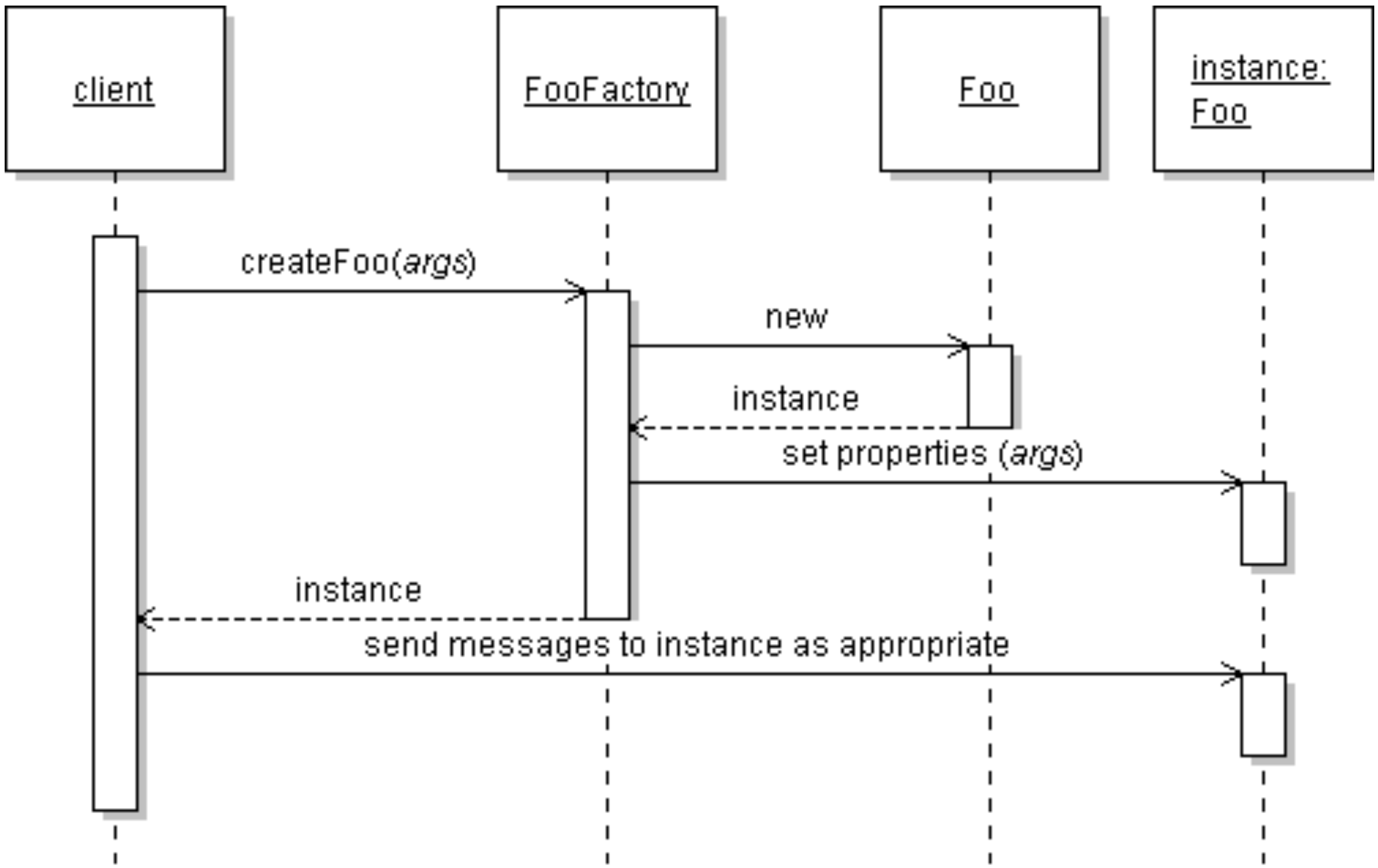
- The factory itself should not be instantiated.
  - make constructor private

- The factory uses static methods to construct components.

- The factory should offer as simple an interface to client code as possible.
  - Don't demand lots of arguments; possibly overload factory methods to handle special cases that need more arguments.

- Factories are often designed for reuse on a later project or for general use throughout your system.

Factory sequence diagram

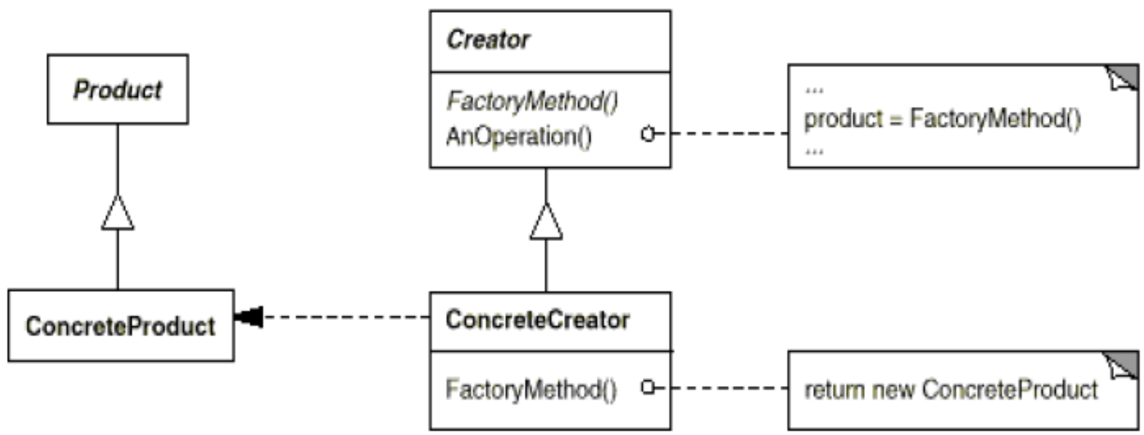


Factory example

```
public class ButtonFactory {
    private ButtonFactory() {}

    public static JButton createButton(
            String text, ActionListener listener,
            Container panel) {
        JButton button = new JButton(text);
        button.setMnemonic(text.charAt(0));
        button.addActionListener(listener);
        panel.add(button);
        return button;
    }
}
```

GoF's variations on Factory

- **Factory Method pattern**: A factory object that can be constructed and has an overridable method to create its objects
  - can be subclassed to make new kinds of factories
- **Abstract Factory pattern**: When the topmost factory class and its creational method are abstract (can be overridden)



Factory Method Example

```
abstract class SalesTax {
    protected double rate;
    abstract void getRate();

    public void calculateTax(double amount) {
        System.out.printf("$%6.2f\n", amount * (1.0 +rate));
    }

}
```

See SWD pages 147-151 for full details and extension

Factory Method Example cont.

```
public class BostonTax extends SalesTax {
    public void getRate() {
        rate = 0.0875;
    }
}

public class ChicagoTax extends SalesTax {
    public void getRate() {
        rate = 0.075;
    }
}

public class StLouisTax extends SalesTax {
    public void getRate() {
        rate = 0.05;
    }
}
```

Factory Method Example cont.

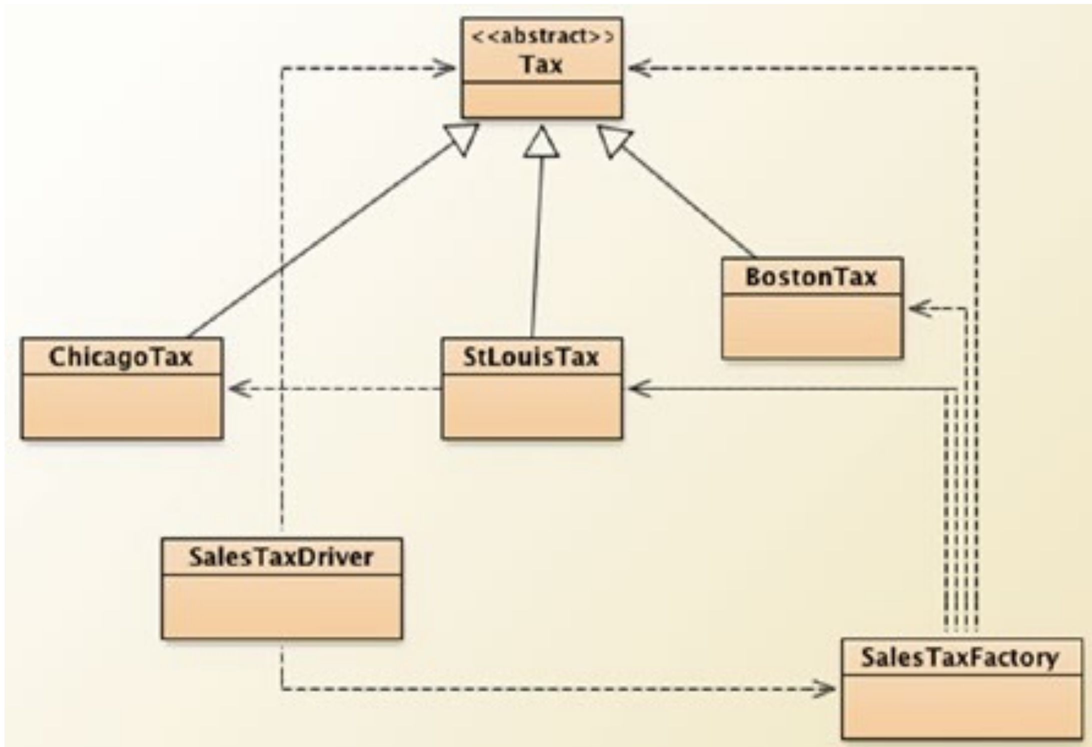
```
public class SalesTaxFactory {
    /**
     * use the makeTaxObject() method to get object of type SalesTax
     */
    public SalesTax makeTaxObject(String location) {

        if(location == null) {
            return null;
        } else if(location.equalsIgnoreCase("boston")) {
            return new BostonTax();
        } else if(location.equalsIgnoreCase("chicago")) {
            return new ChicagoTax();
        } else if(location.equalsIgnoreCase("stlouis")) {
            return new StLouisTax();
        }

        return null;
    }
}
```

Factory Method Example cont.

```java
/**
 * Test the Factory Method pattern.
 * We use the SalesTaxFactory to get the object of concrete classes
 */

import java.io.*;
import java.util.Scanner;

public class SalesTaxDriver {

    public static void main(String args[])throws IOException {
        Scanner stdin = new Scanner(System.in);

        SalesTaxFactory salesTaxFactory = new SalesTaxFactory();
        //get an object of type SalesTax and call its getTax()method.

        System.out.print("Enter the location (boston/chicago/stlouis): ");
        String location= stdin.nextLine();

        System.out.print("Enter the dollar amount: ");
        double amount = stdin.nextDouble();

        SalesTax cityTax = salesTaxFactory.makeTaxObject(location);

        System.out.printf("Bill amount for %s of $%6.2f is: ", location, amount);
        cityTax.getRate();
        cityTax.calculateTax(amount);
    }
}
```


See SWD pages 147-151 for full details and extension

Factory Method Example cont.



UML for SalesTaxFactory example