CSE 331

Design Patterns 1:
Iterator, Adapter, Singleton, Flyweight

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer
http://www.cs.washington.edu/331/

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: Singleton

A class that has only a single instance

Creational Patterns

- Constructors in Java are inflexible:
  - Can't return a subtype of the class they belong to.
  - Always returns a fresh new object; can never re-use one.
- Creational factories:
  - Factory method
  - Abstract Factory object
  - Prototype
  - Dependency injection
- Sharing:
  - Singleton
  - Interning
  - Flyweight

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 lots of 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?

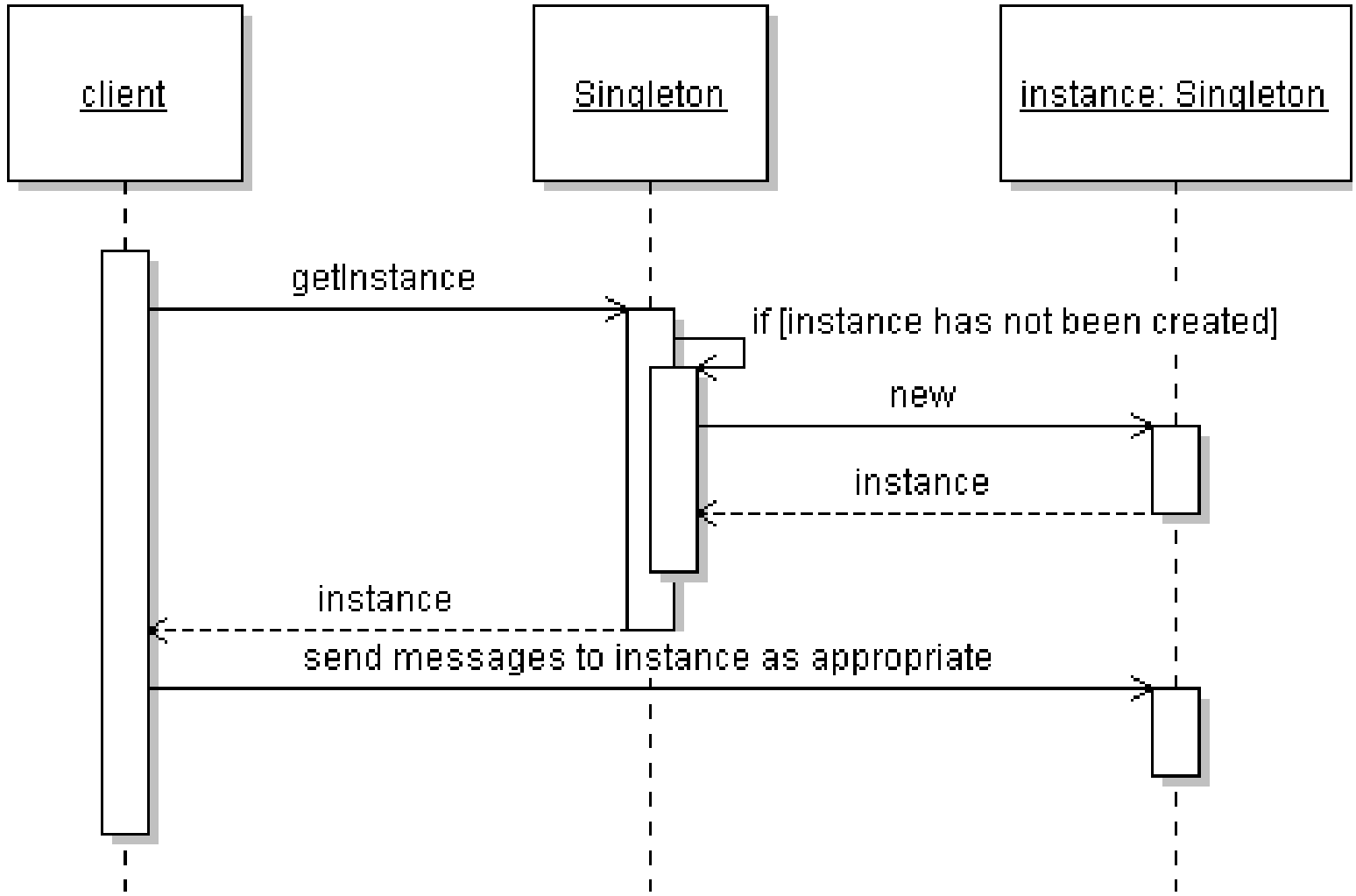
- *Disadvantage*: Lacks flexibility.
  - Static methods can't be passed as an argument, nor returned.

- *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.
- 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() {}

    ...
}
```

Lazy initialization

- Can wait until client asks for the instance to create it:

```
public class RandomGenerator {
    private static RandomGenerator gen = null;

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

    private RandomGenerator() {}

    ...
}
```

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: Flyweight

a class that has only one instance for each unique state

Redundant objects

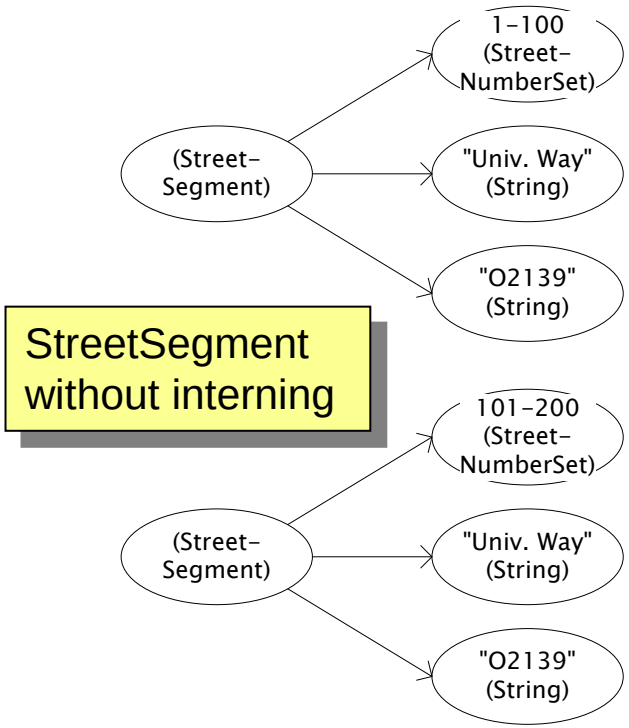
- *Problem*: Redundant objects can bog down the system.
  - Many objects have the same state.
  - example: `File` objects that represent the same file on disk
    - `new File("mobydick.txt")`
    - `new File("mobydick.txt")`
    - `new File("mobydick.txt")`
    - …
    - `new File("notes.txt")`
  - example: `Date` objects that represent the same date of the year
    - `new Date(4, 18)`
    - `new Date(4, 18)`

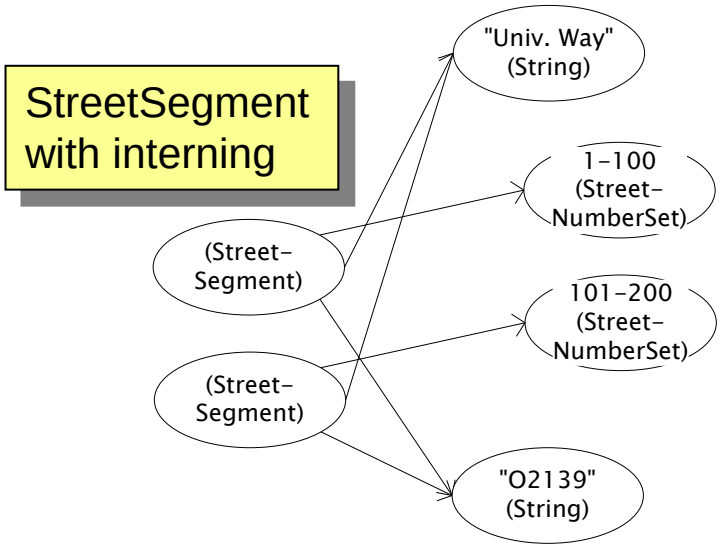
Flyweight pattern

- **flyweight**: An assurance that no more than one instance of a class will have identical state.
  - Achieved by caching identical instances of objects.
  - Similar to singleton, but one instance for each unique object state.
  - Useful when there are many instances, but many are equivalent.
  - Can be used in conjunction with Factory Method pattern to create a very efficient object-builder.
  - Examples in Java: `String`, `Image`, `Toolkit`, `Formatter`, `Calendar`, `JDBC`

Flyweight diagram

- Flyweighting shares objects and/or shares their internal state
  - saves memory
  - allows comparisons with == rather than `equals` (why?)





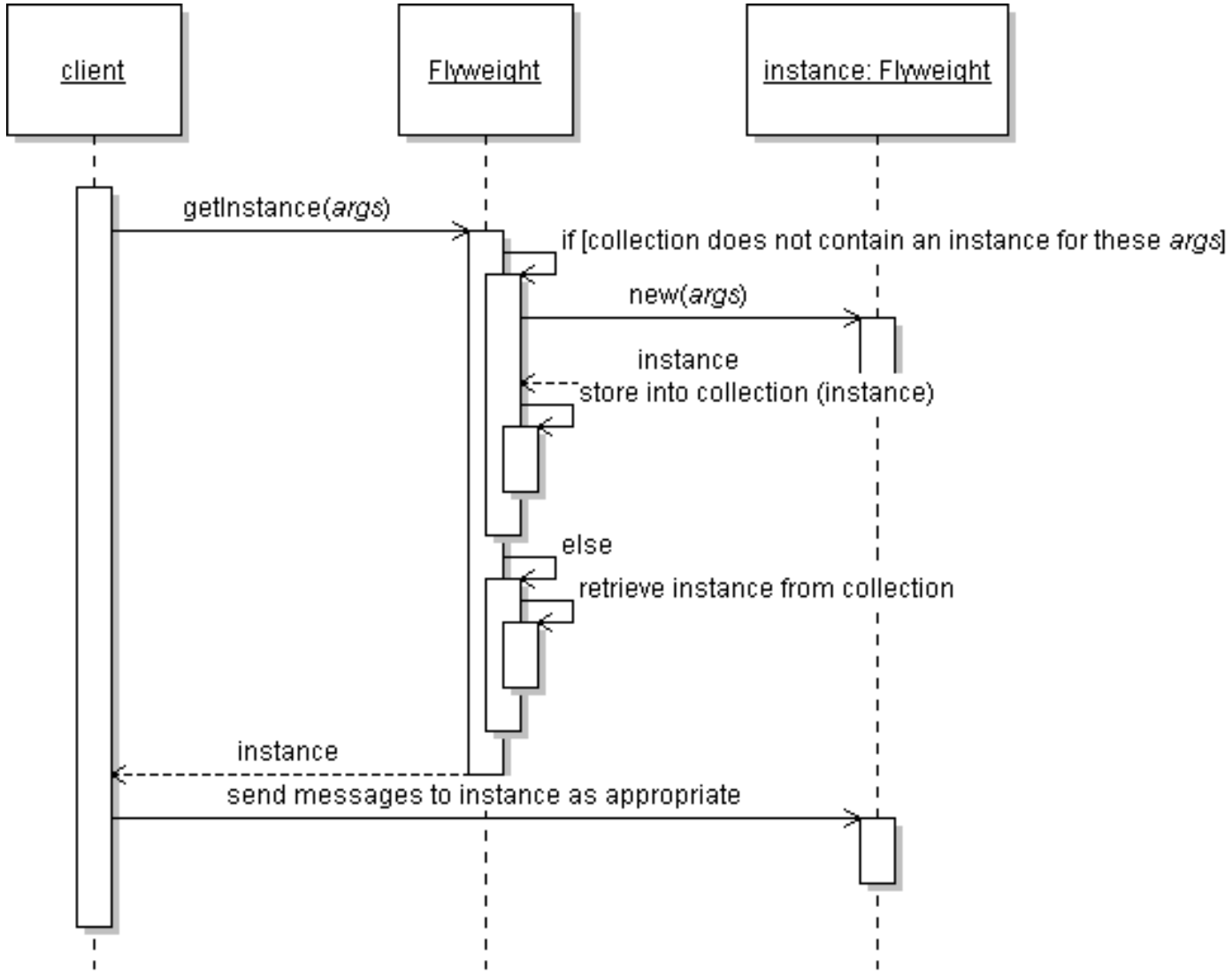
Implementing a Flyweight

- Flyweighting works best on *immutable* objects. (Why?)

- Class pseudo-code sketch:

```
public class Name {
  • static collection of instances
  • private constructor
  • static method to get an instance:
    if (we have created this kind of instance before):
        get it from the collection and return it.
    else:
        create a new instance, store it in the collection and return it.
}
```

Flyweight sequence diagram



Implementing a Flyweight

```
public class Flyweighted {
    private static Map<KeyType, Flyweighted> instances
            = new HashMap<KeyType, Flyweighted>();

    private Flyweighted(...) { ... }

    public static Flyweighted getInstance(KeyType key) {
        if (!instances.contains(key)) {
            instances.put(key, new Flyweighted(key));
        }
        return instances.get(key);
    }
}
```

Class before flyweighting

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

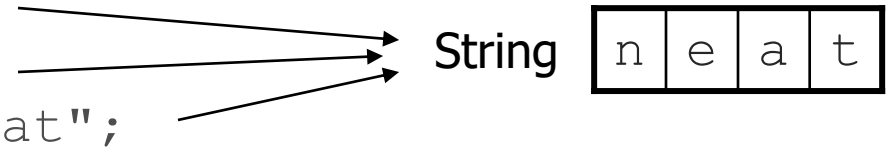
Class after flyweighting

```
public class Point {
    private static Map<String, Point> instances =
        new HashMap<String, Point>();

    public static Point getInstance(int x, int y) {
        String key = x + ", " + y;
        if (!instances.containsKey(key)) {
            instances.put(key, new Point(x, y));
        }
        return instances.get(key);
    }

    private final int x, y;  // immutable

    private Point(int x, int y) {
        ...
```

String flyweighting

- **interning**: Synonym for flyweighting; sharing identical instances.
  - Java `String` objects are automatically interned (flyweighted) by the compiler whenever possible.
  - If you declare two string variables that point to the same literal.
  - If you concatenate two string literals to match another literal.

```
String a = "neat";
String b = "neat";
String c = "n" + "eat";
```



- So why doesn't == always work with `Strings`?

Limits of String flyweight

```
String a = "neat";
Scanner console = new Scanner(System.in);
String b = console.next();  // user types "neat"
if (a == b) { ...           // false
```

- There are many cases the compiler doesn't / can't flyweight:
  - When you build a string later out of arbitrary variables
  - When you read a string from a file or stream (e.g. `Scanner`)
  - When you build a new string from a `StringBuilder`
  - When you explicitly ask for a `new String` (bypasses flyweighting)

- You can force Java to flyweight a particular string with `intern`:

```
b = b.intern();
if (a == b) { ...           // true
```

String interning questions

```
String fly  = "fly";  String weight  = "weight";
String fly2 = "fly";  String weight2 = "weight";
```

- Which of the following expressions are `true`?
  a) `fly == fly2`
  b) `weight == weight2`
  c) `"fly" + "weight" == "flyweight"`
  d) `fly + weight == "flyweight"`

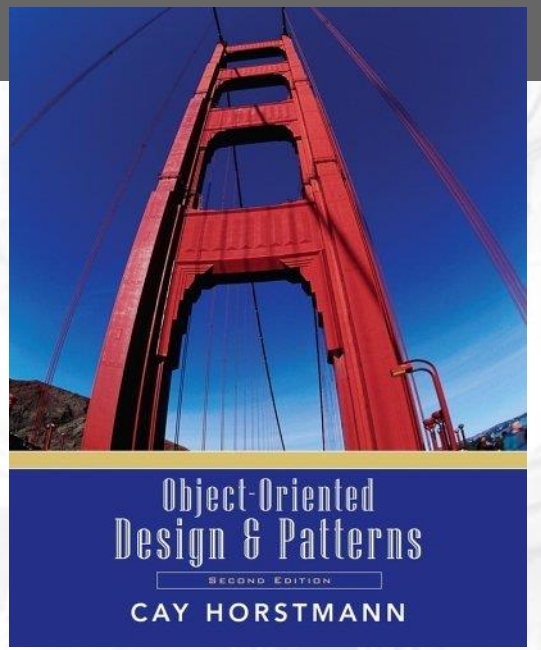
```
String flyweight = new String("fly" + "weight");
```

  e) `flyweight == "flyweight"`

```
String interned1 = (fly + weight).intern();
String interned2 = flyweight.intern();
```

  f) `interned1 == "flyweight"`
  g) `interned2 == "flyweight"`

UNIVERSITY OF WASHINGTON



CSE 403

Design Patterns and GUI Programming

Reading:

Object-Oriented Design and Patterns, Ch. 5 (Horstmann)

Benefits of using patterns

- patterns are a 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
- patterns capture design expertise and allow that expertise to be communicated
  - promotes design reuse and avoid mistakes
- improve documentation (less is needed) and understandability (patterns are described well once)

Gang of Four (GoF) patterns

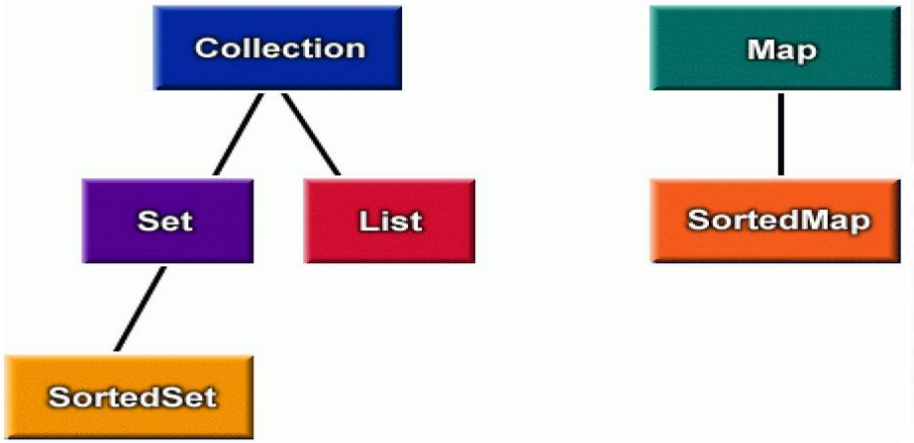
- **Creational Patterns**
  (abstracting the object-instantiation process)
  - Factory Method
  - Abstract Factory
  - Singleton
  - Builder
  - Prototype

- **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

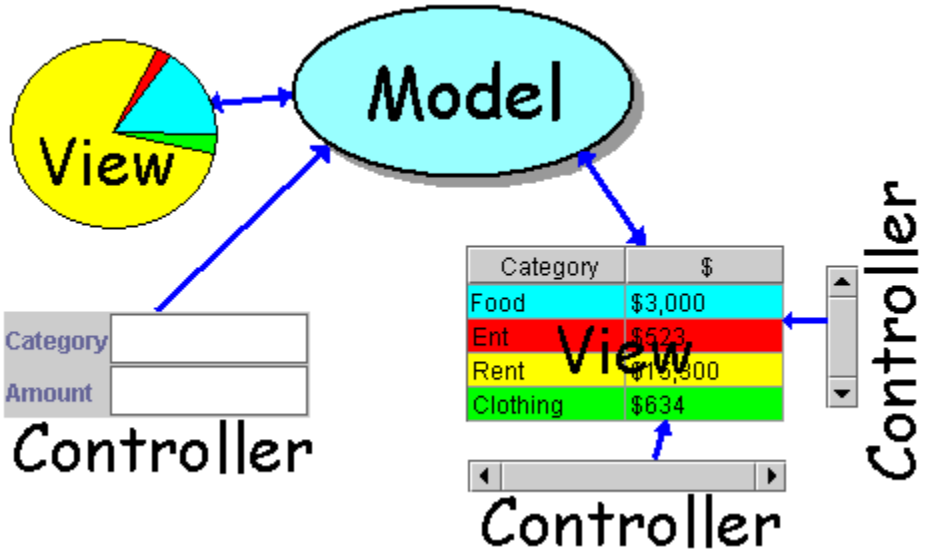
- **iterator**: an object that provides a standard way to examine all elements of any collection
  - uniform interface for traversing many different data structures
  - supports concurrent iteration and element removal

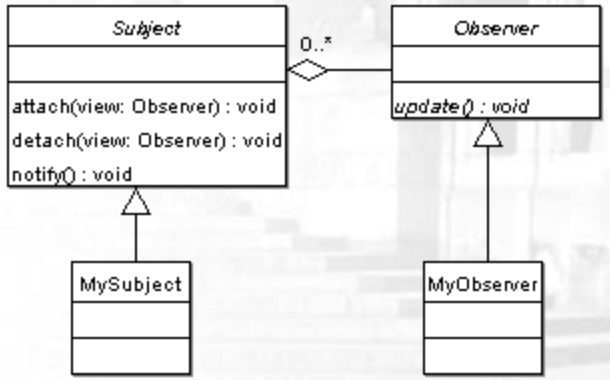
```
for (Iterator<Account> itr = list.iterator(); itr.hasNext(); ) {
    Account a = itr.next();
    System.out.println(a);
}
```

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

Pattern: Observer

objects whose state can be watched





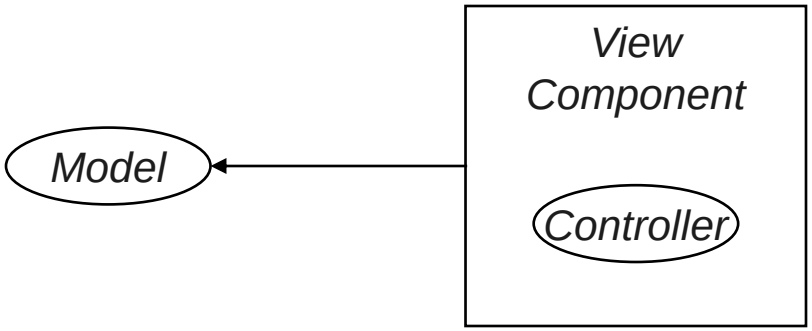
Recall: model and view

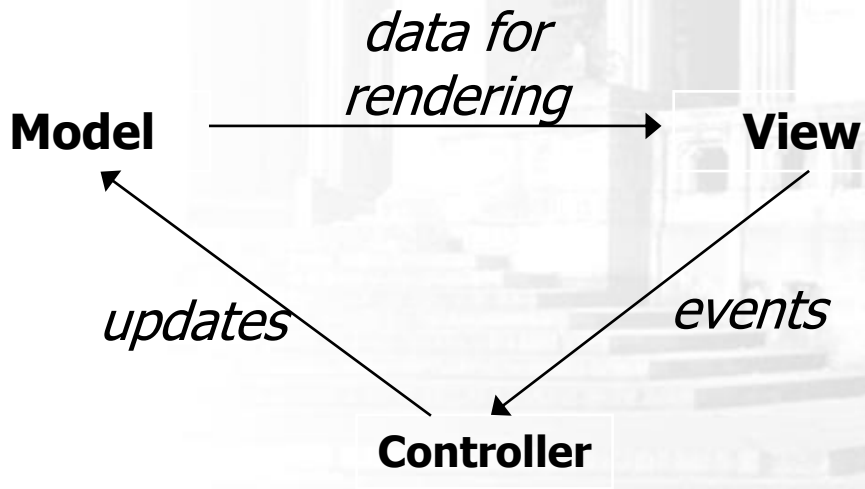
- **model**: classes in your system that are related to the internal representation of the state of the system
  - often part of the model is connected to file(s) or database(s)
  - examples (card game): Card, Deck, Player
  - examples (bank system): Account, User, UserList

- **view**: classes in your system 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: PokerPanel, BankApplet

Model-view-controller

- **model-view-controller (MVC)**: common design paradigm for graphical systems
- **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)
  - sometimes part of view (see left)





Observer pattern

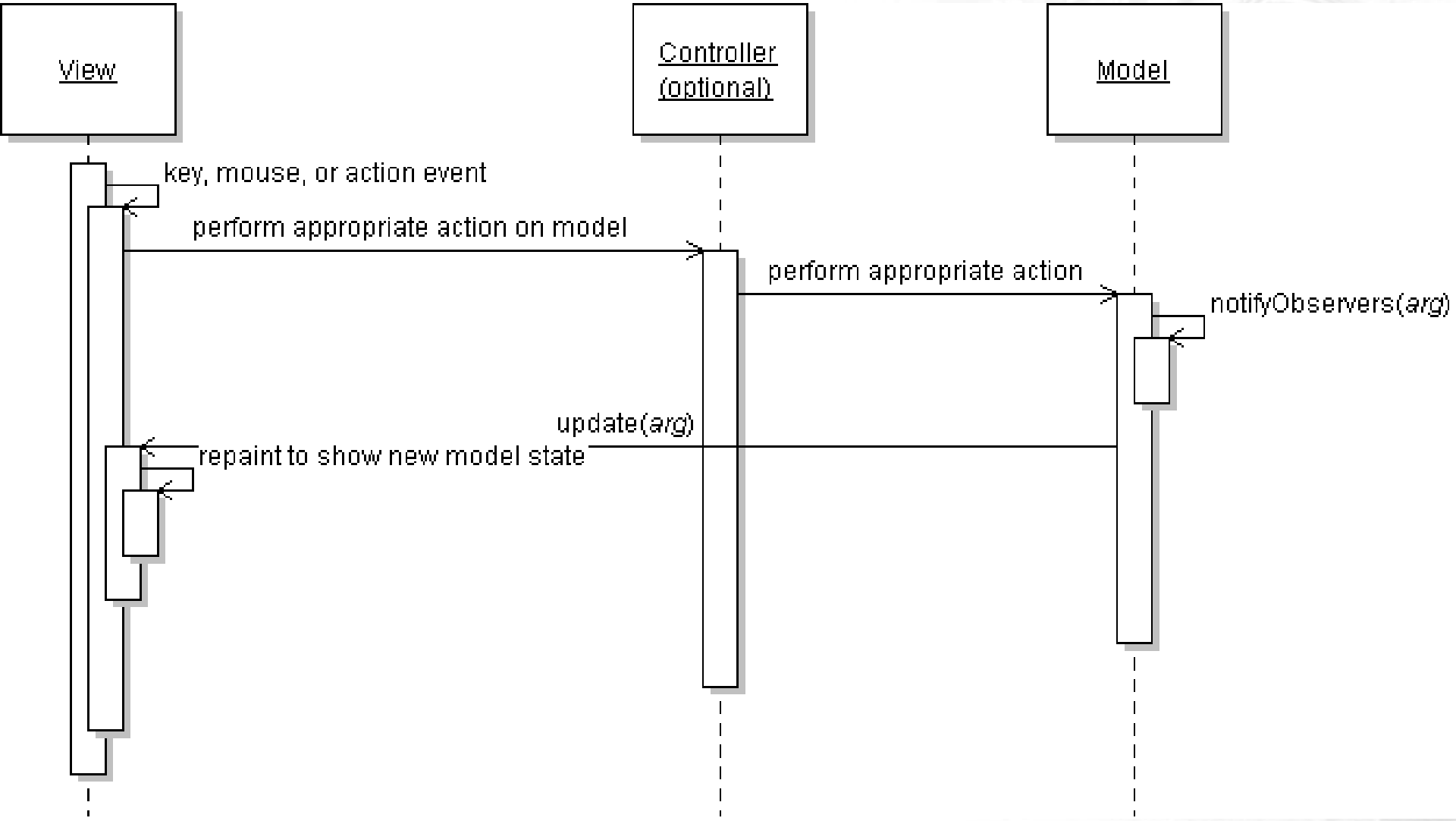
- **observer**: an object that "watches" the state of another object and takes action when the state changes in some way
  - examples in Java: event listeners; `java.util.Observer`

- **observable** object: an object that allows observers to examine it (often the observable object notifies the observers when it changes)
  - permits customizable, extensible event-based behavior for data modeling and graphics

Benefits of observer

- abstract 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 a lot more programming flexibility
- broadcast communication: notification is broadcast automatically to all interested objects that subscribed to it
- Observer can be used to implement model-view separation in Java more easily

Observer sequence diagram



Observer interface

```
package java.util;

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

- Idea: The `update` method will be called when the observable model changes, so put the appropriate code to handle the change inside `update`

Observable class

- `public void addObserver(Observer o)`
- `public void deleteObserver(Observer o)`
  Adds/removes `o` to/from the list of objects that will be notified (via their update method) when notifyObservers is called.

- `public void notifyObservers()`
- `public void notifyObservers(Object arg)`
  Inform all observers listening to this Observable object of an event that has occurred. An optional object argument may be passed to provide more information about the event.

- `public void setChanged()`
  Flags the observable object as having changed since the last event; must be called each time before calling notifyObservers.

Common usage of Observer

1. write a model class that extends `Observable`
   - have the model notify its observers when anything significant happens

2. make all views of that model (e.g. GUI panels that draw the model on screen) into observers
   - have the panels take action when the model notifies them of events (e.g. `repaint`, play sound, show option dialog, etc.)

Using multiple views

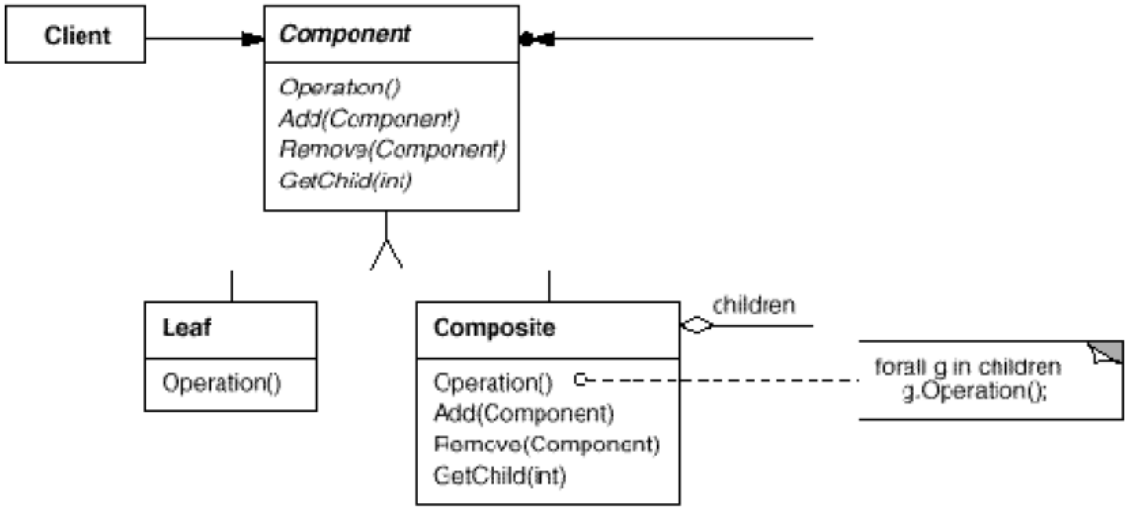
- make an Observable model
- write a View interface or abstract class
  - make View an observer
- extend/implement View for all actual views
  - give each its own unique inner components and code to draw the model's state in its own way
- provide mechanism in GUI to set view (perhaps through menus)
  - to set view, attach it to observe the model

Example: changing views

```
// in the frame's action listener:
// hide old view; show new one

model.deleteObserver(view1);
model.addObserver(view2);
view1.setVisible(false);
view2.setVisible(true);
```

Pattern: Composite

objects that can serve as containers, and can hold other objects like themselves



Composite pattern

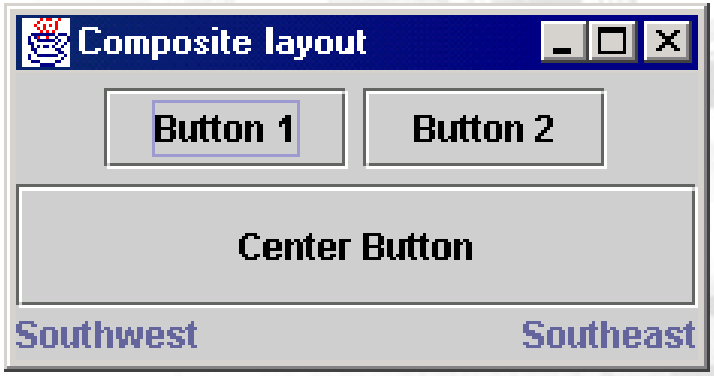
- **composite:** an object that is either an individual item or a collection of many items
  - composite objects can be composed of individual items or of other composites
  - recursive definition: objects that can hold themselves

  - often leads to a tree structure of leaves and nodes:
    - **<node>** ::= **<leafnode>** | **<compositenode>**
    - **<compositenode>** ::= **<node>***

- examples in Java:
  - collections (a List of Lists)
  - GUI layout (panels containing panels containing buttons, etc.)

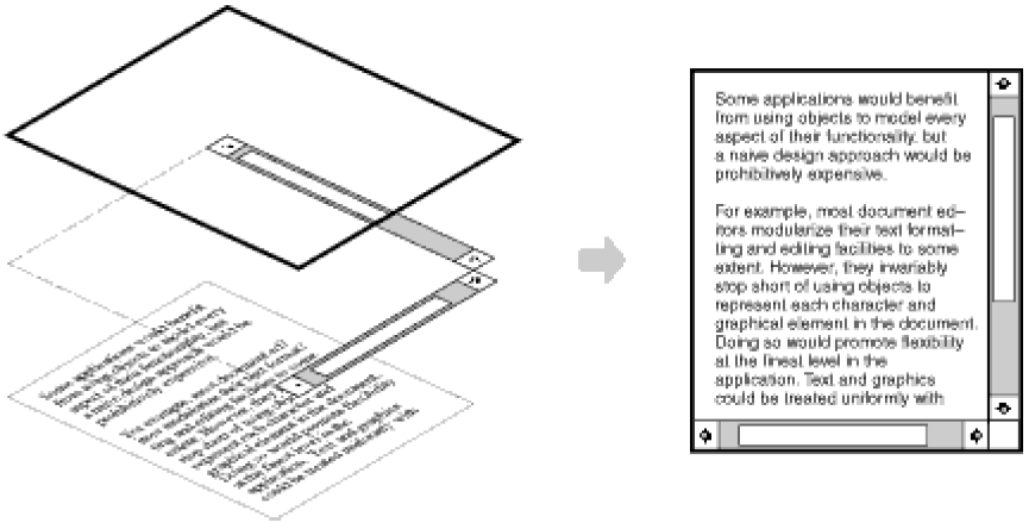
Composite example: panels

```
Container north = new JPanel(new FlowLayout());
north.add(new JButton("Button 1"));
north.add(new JButton("Button 2"));

Container south = new JPanel(new BorderLayout());
south.add(new JLabel("Southwest"), BorderLayout.WEST);
south.add(new JLabel("Southeast"), BorderLayout.EAST);

// overall panel contains the smaller panels (composite)
JPanel overall = new JPanel(new BorderLayout());
overall.add(north, BorderLayout.NORTH);
overall.add(new JButton("Center Button"), BorderLayout.CENTER);
overall.add(south, BorderLayout.SOUTH);
frame.add(overall);
```



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 example: I/O

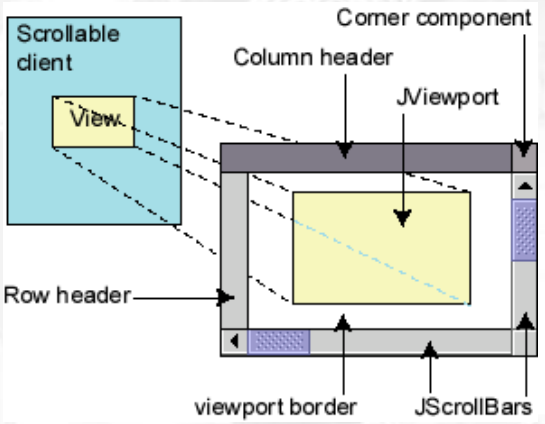
- normal `InputStream` class has only `public int read()` method to read one letter at a time
- decorators such as `BufferedReader` or `Scanner` add additional functionality to read the stream more easily

```
// 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?