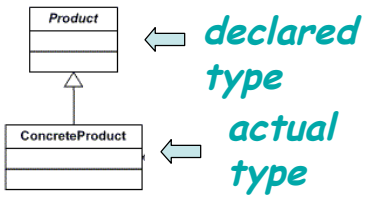
Design Patterns2

- More design patterns (GoF)
  - Structural: Adapter, Bridge, Façade
  - Creational: Abstract Factory, Singleton
  - Behavioral: Observer, Iterator, State, Visitor

Factory Method

- Isolate the creation of certain objects into a separate method
  - The method "manufactures" the objects
- Declared return type: a supertype of the created object
  - The caller does not know the exact type of the new object



Example - Original Class

```
Pizza orderPizza(String type){
   Pizza pizza;
   if (type.equals("cheese")) { pizza= new CheesePizza();
   }  else if (type.equals("greek") {pizza= new GreekPizza();
   }  else if (type.equals("pepperoni") {pizza= new PepperPizza();
   }  else …
   }
   pizza.prepare();  pizza.bake();  pizza.cut(); pizza.box();
   return pizza:
}
```

IDEA: want to separate creation of pizza objects from pizza operations, for ease of accommodating changes in types of pizza in the future; need to abstract out shaded code into a Factory to create the pizzas

Example

```
public class SimplePizzaFactory{
   public Pizza createPizza(String type){
         Pizza pizza = null;
         if (type.equals("cheese")) { pizza= new CheesePizza();
         }  else if (type.equals("greek") {pizza= new GreekPizza();
         }  else if (type.equals("pepperoni") {pizza= new PepperPizza();
         }  else …
         }  return pizza;
   }
}
```

First, create the Factory class which will be responsible for creating pizza objects

Example - Separate object creation from client code

Each PizzaStore object has-a SimplePizzaFactory object

```
public class PizzaStore{
   SimplePizzaFactory factory;
   public PizzaStore(SimplePizzaFactory factory){
        this.factory = factory;
   }
   public Pizza orderPizza(String type){
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();  pizza.bake();  pizza.cut(); pizza.box();
   return pizza;}
}
```

Second, replace new() with createPizza() call, having received factory as parameter to PizzaStore constructor; now have separated pizza creation from client (PizzaStore);

Example - Abstract Factory - Creation by subclass

```
public abstract class PizzaStore{
        public Pizza orderPizza(String type){
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();  pizza.bake();  pizza.cut(); pizza.box();
    return pizza;}
    abstract Pizza createPizza(String type);
}
```

Now the concrete subclasses of PizzaStore (e.g., NYPizzaStore, ChicagoPizzaStore) become the Factory for this class; they each have different specialized creation methods;

Factory Method

abstract Pizza createPizza(String type);

abstract means subclasses will implement

Pizza is the product to be created

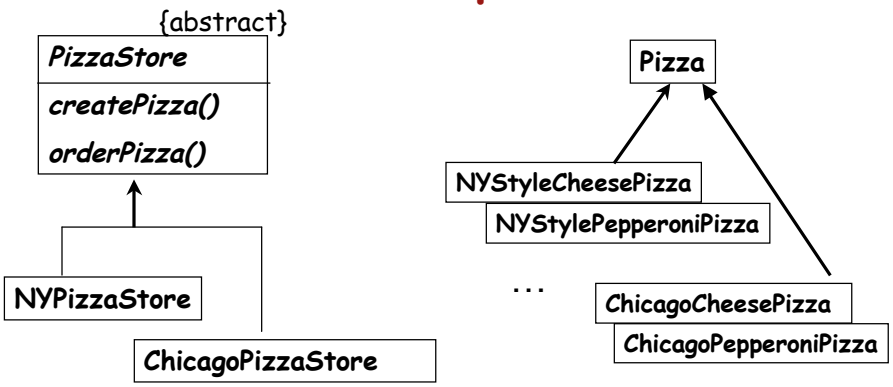
createPizza shows name of product in factory name, while isolating client code in superclass from knowing actual kinds of product there are

type allows parameterization to select from among product varieties

Creation by Subclasses

- The superclass (PizzaStore) does not know which Pizza is being instantiated; it just uses the Pizza object returned to it
- Need subclasses of both Pizza and PizzaStore for client-specific behavior
- Sometimes: in the superclass, have a "default" factory implementation
  - Subclasses can override it if they need to

Example



Creator classes

Product classes

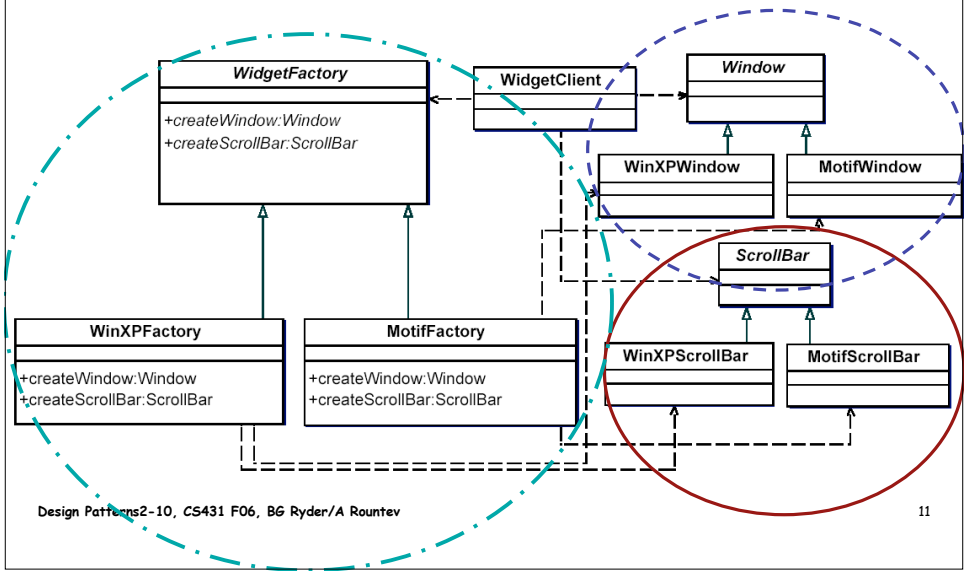
Actually an example of parallel class hierarchies.

Summary - Abstract Factory

- A generalization of the "factory" idea
  - Put factory method(s) in a separate class
- Usually: several categories of objects
  - e.g., GUI elements: windows, scroll bars, etc.
- Separate factory method for each category
  - createWindow():Window
  - createScrollBar():ScrollBar
  - Window and ScrollBar are abstract classes

Another Factory Example

Abstract factory + concrete factories



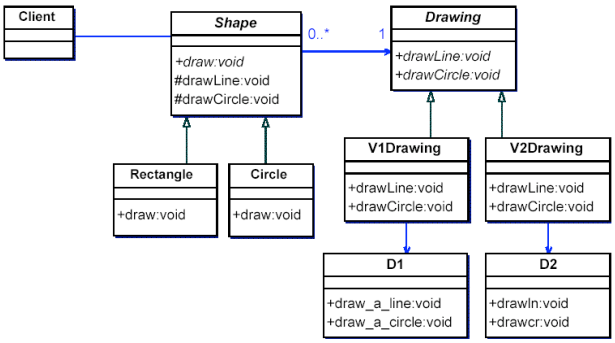
Observations

- Creation of entire families of objects can be controlled transparently

```
AbstractFactory f = new ConcreteFactory1();
                    ⇩
AbstractFactory f = new ConcreteFactory2();
```

- Separation of responsibilities
  - Factory: decides which objects are needed
  - Client: uses objects through their supertypes

Combining Bridges and Factories



- Factory method: e.g., static method createDrawing() in Drawing; called by Shape
- Concrete factory class DrawingFactory with method createDrawing(); called by Shape (and its subclasses)

Singleton Pattern

- Goal: to ensure that there is only one instance of a given class X
- Step 1: make all of X's constructors private
- Step 2: add in X a private static field
  - Usually called instance
- Step 3: add a public static method that returns the static field
  - Usually called getInstance()

Example

```
class Logger {
  private Logger() { }
  private static Logger instance = null;
   public static Logger getInstance() {
        if (instance == null) instance = new Logger();
        return instance;
   }
}
```

---

client code: Logger.getInstance().writeLog(…)

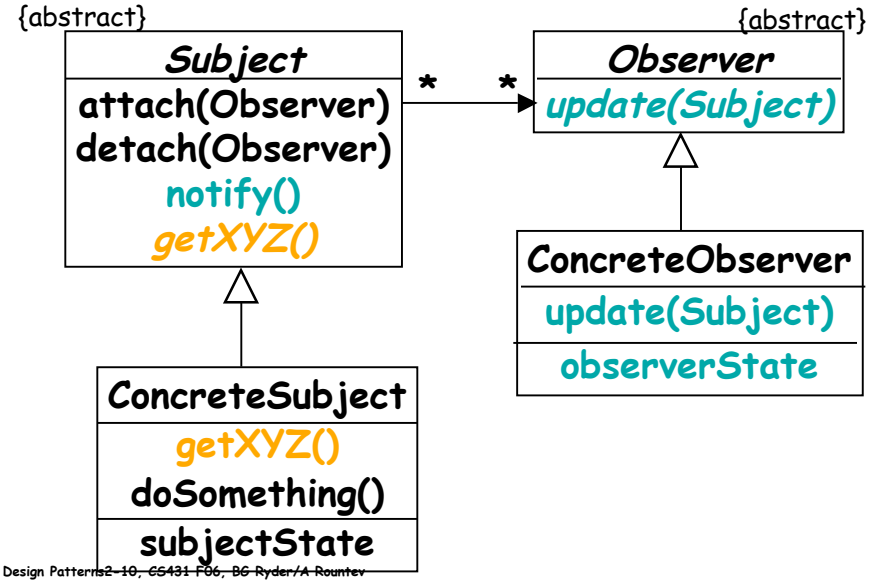
Observations

- On-demand creation
  - For performance reasons
  - Alternative: upfront creation, by initializing the static field
- Common reasons for using it
  - Unique resources: file system, window manager, printer spooler, factory, …
- Problems for multi-threaded programs
    - What if 2 threads try to do getInstance() simultaneously? Can we get 2 instances?
      - Make getInstance() synchronized?
      - Alternatives?

Observer Pattern

- Goal: when one object X changes state, all its dependent objects Y_i are notified and updated automatically
- Want to reduce coupling
  - Do not hard-code calls from X to Y_i
  - Want if we want to add a new Y_i?
- Abstract class Observer, subclassed by concrete observer classes
- X has a list of all observer objects

GoF Formulation (Modified)



Interactions

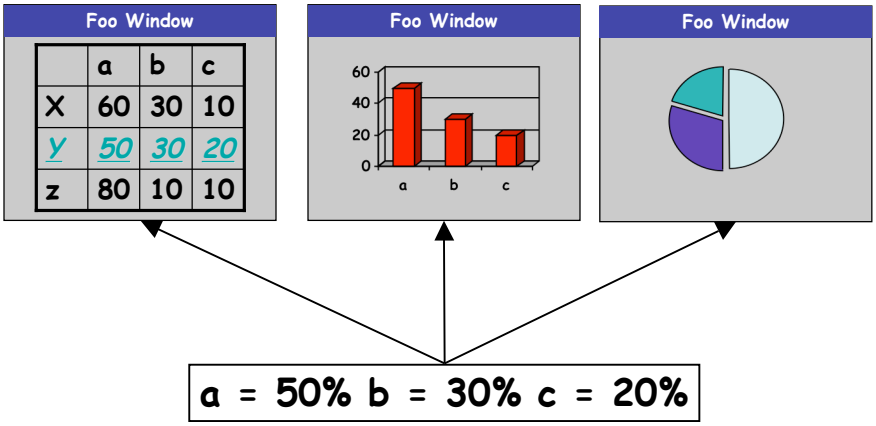
- doSomething() changes the state of the concrete subject, and invokes notify()
- notify() goes through all attached observers and invokes update(this) on each one
  - this call invokes the corresponding update method in the concrete observers
- update(Subject s) calls s.getXYZ() to get details about the state change in order to update its own state

Possible Implementation in Java

```
class Subject {
   private HashSet oset = new HashSet();
   public attach(Observer o) { oset.add(o); }
   public detach(Observer o) { oset.remove(o); }
   public notify() {
       Iterator it = oset.iterator();
       while (it.hasNext()) {
              Observer o = (Observer) it.next();
              o.update(this);
       } } ... }
```

Spreadsheet Example

- In a spreadsheet, we may have multiple GUI views of the same data



Why Use It?

- Changes in object state in one object require state changes in other objects
- Do not know in advance the dependent objects
- To decouple the objects
  - In anticipation of future changes
  - For reuse: can reuse the subjects without the observers, and vice versa
- The subject is simplified
  - Only responsibility: broadcast to observers

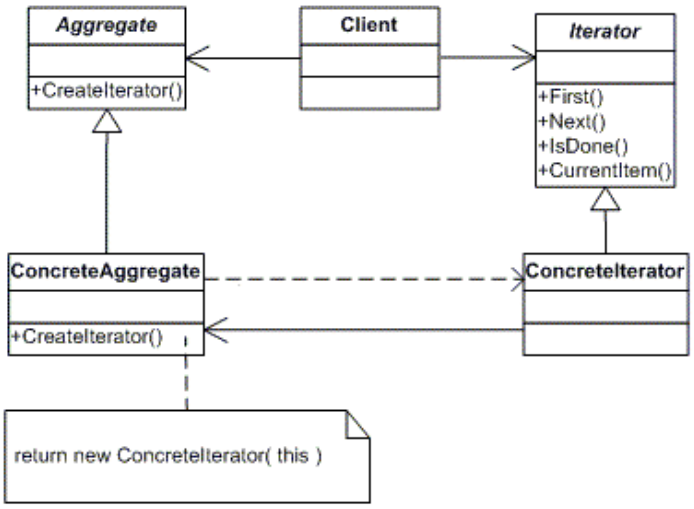
Example: standard package java.util

```
class Observable { // subject
   public void addObserver(Observer o) {…}
   public void deleteObserver(Observer o) {…}
   public void deleteObservers() {…}
   public int countObservers() {…}
   public void notifyObservers()
        { notifyObservers(null); }
   public void notifyObservers(Object arg) {…}
}
interface Observer {
  public void update(Observable o, Object arg); }
```


Iterator

- Goal: access the elements of an aggregate object without exposing its underlying representation
  - e.g. elements of a List, Set, Table, …
- Use separate "iterator" classes
  - Each iterator object corresponds to a particular traversal of the elements
- Used very often, especially in standard libraries (C++, Java, etc.)

General Form



Example: java.util.ArrayList

- ArrayList: subclass of AbstractList
  - add(index, Object), isEmpty(), …
  - Factory method: Iterator iterator()
    - Inherited from AbstractList
- Interface Iterator
  - Methods hasNext() and next()
- iterator() in AbstractList returns an instance of an internal class that:
  - implements the Iterator interface
  - is specific for Lists

Sample Client Code

```
ArrayList ar;
...
Iterator iter = ar.iterator();
while ( iter.hasNext() ) {
  Object element = iter.next();
  // do something with the list element
}
```


Observation

- An iterator is associated with a particular aggregate object
- There may be several active iterators for the same object at the same time
- Iterate over different aggregates
  - AbstractList has subclasses LinkedList, ArrayList, Vector, and Stack
- Are iterators allowed to change the aggregate object?
  - Sometimes yes, but should be done carefully
  - Often is a PL design question

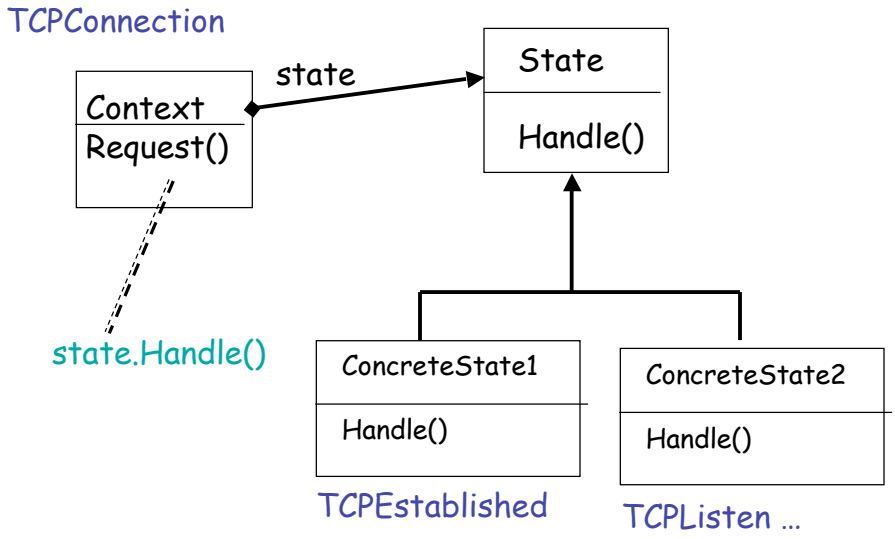
State

- Useful if object should change its behavior when its internal state changes
  - Encapsulate state into separate classes and delegate to the object representing the current state

Example

Imagine a TCPConnection object that can be in one of several states
  - TCPEstablished, TCPListen, TCPClosed
  - The object responds differently to requests, depending on what state it is in
  - Think of each TCPConnection having a state field that has-a relation to different TCPState objects during a connection; then behavior can be delegated to the proper TCPState object
- When to use?
  - When object's behavior depends on its state and must change at runtime.
  - Operations have large multipart conditionals, depending on object state

State Pattern



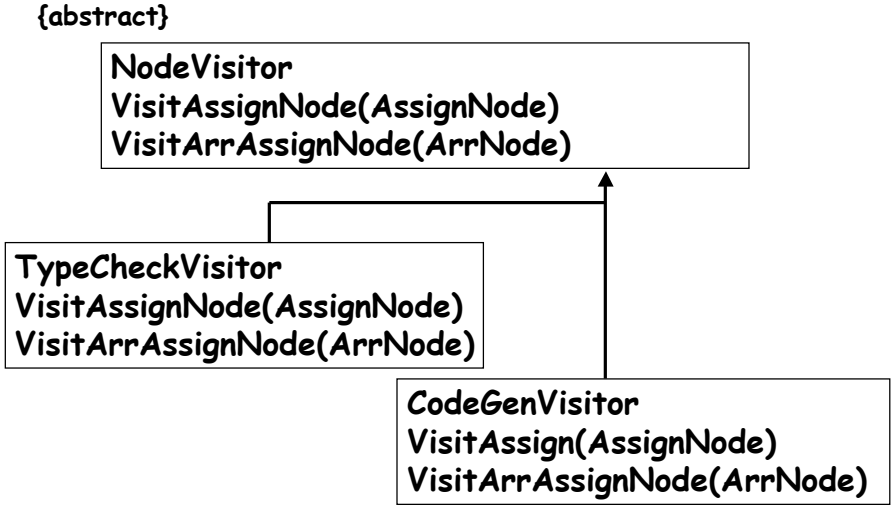
Another Example, Intuition

- Gumball machine with NoQuarter, HasQuarter, Sold, SoldOut states
  - Think of how the GumballMachine object responds differently to the command turnCrank() depending on which of these states it is in
- Often this pattern is an alternative to using lots of conditionals in your code

Visitor Pattern

- Given a composite structure (e.g., AST) and operations on each element (e.g., type checking a node), pattern separates the operations from the structure classes
  - Gathers all related operations in Visitor class
  - Eases addition of new operations
  - Allows collection of state information
  - But exposes structure to Visitor class, and sometimes breaks encapsulation

Example - Visitor Hierarchy



Example - Element Hierarchy