Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Design patterns for reuse

Charlie Garrod **Bogdan Vasilescu**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 3 due Thursday 11:59 p.m.
- Required reading due today: UML & Patterns Ch 9 and 10
  - Intro to domain modeling & System sequence diagrams
- (Optional) reading for Thursday:
  - UML & Patterns Ch 17
  - EJ 49, 54, 69
- Required reading for next week: UML & Patterns Ch 14—16
- **Midterm exam Thursday next week (Feb 15)**
  - Review session and practice exam (coming soon)

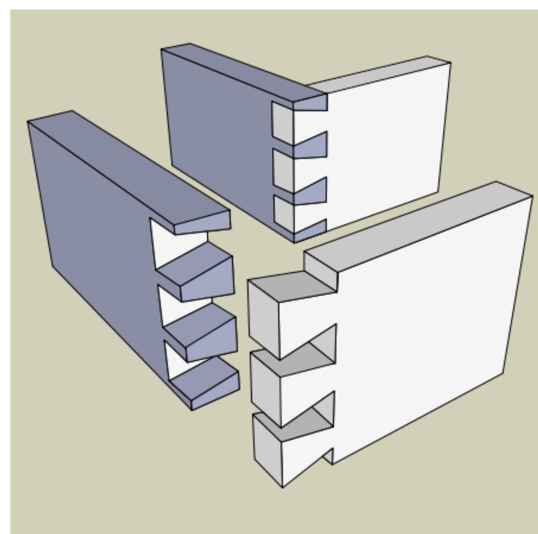
Key concepts from Thursday

UML you should know

- Interfaces vs. classes
- Fields vs. methods
- Relationships:
  - "extends" (inheritance)
  - "implements" (realization)
  - "has a" (aggregation)
  - non-specific association
- Visibility: + (public) - (private) # (protected)

Design patterns

- Carpentry:
  - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
  - "Is a strategy pattern or a template method better here?"



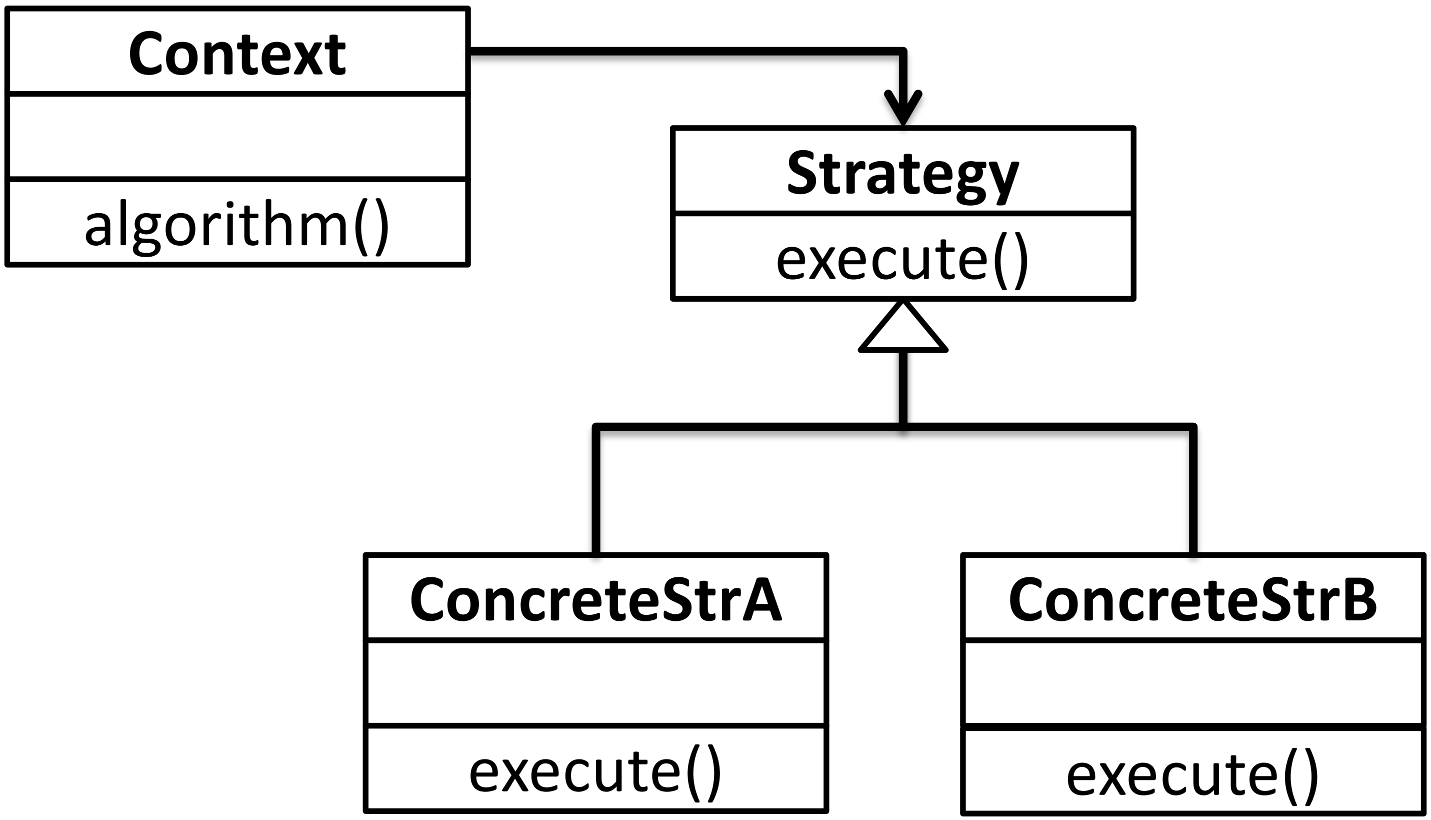
Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

Review

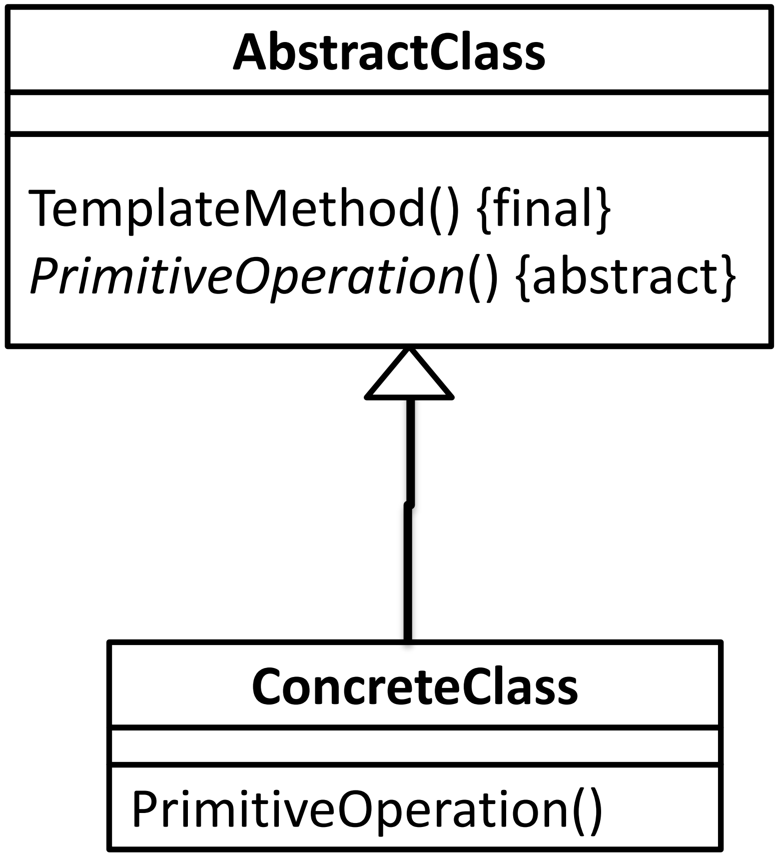
The Strategy Design Pattern

Problem: Clients need different variants of an algorithm



Template method design pattern

- **Problem**: An algorithm consists of customizable parts and invariant parts



Review

Avoiding `instanceof` with the template method pattern

```
public void doSomething(Account acct) {
  float adj = 0.0;
  if (acct instanceof CheckingAccount) {
    checkingAcct = (CheckingAccount) acct;
    adj = checkingAcct.getFee();
  } else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
    adj = savingsAcct.getInterest();
  }
  …
}
```

Instead:

```
public void doSomething(Account acct) {
  long adj = acct.getMonthlyAdjustment();
  …
}
```

Review

institute for SOFTWARE RESEARCH

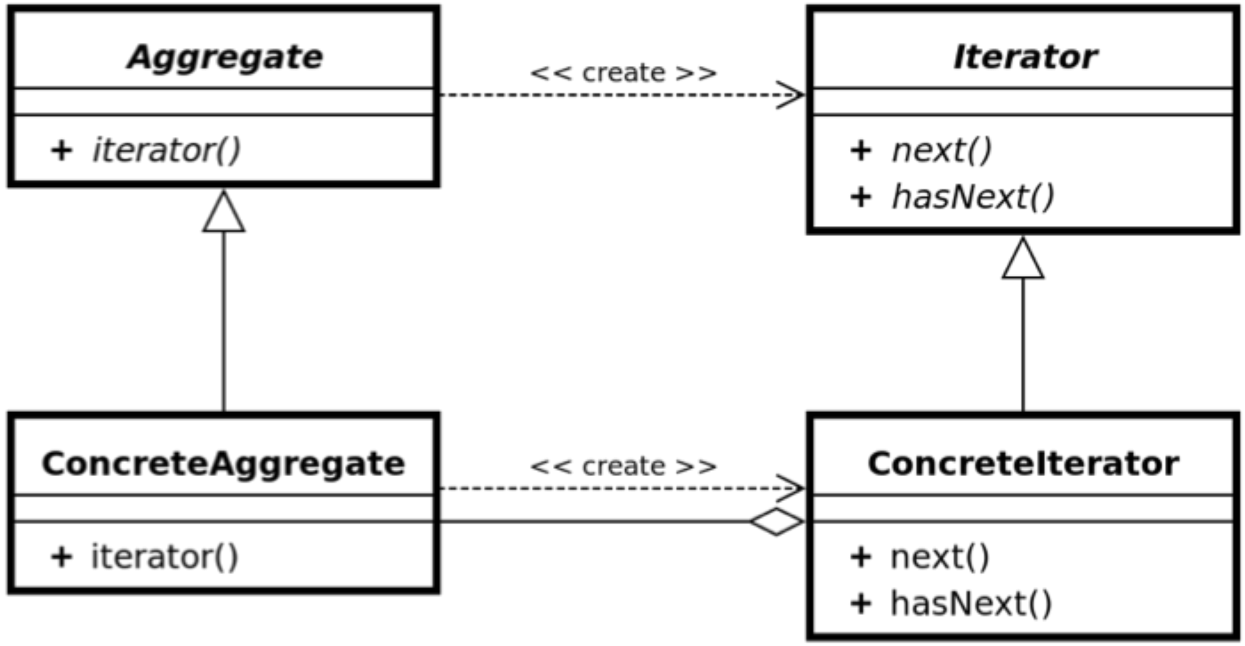
Quiz: Strategy pattern vs Template method pattern

Today

- More design patterns for reuse:
  - The Iterator design pattern (left overs from Thu)
  - The Composite design pattern
  - The Decorator design pattern

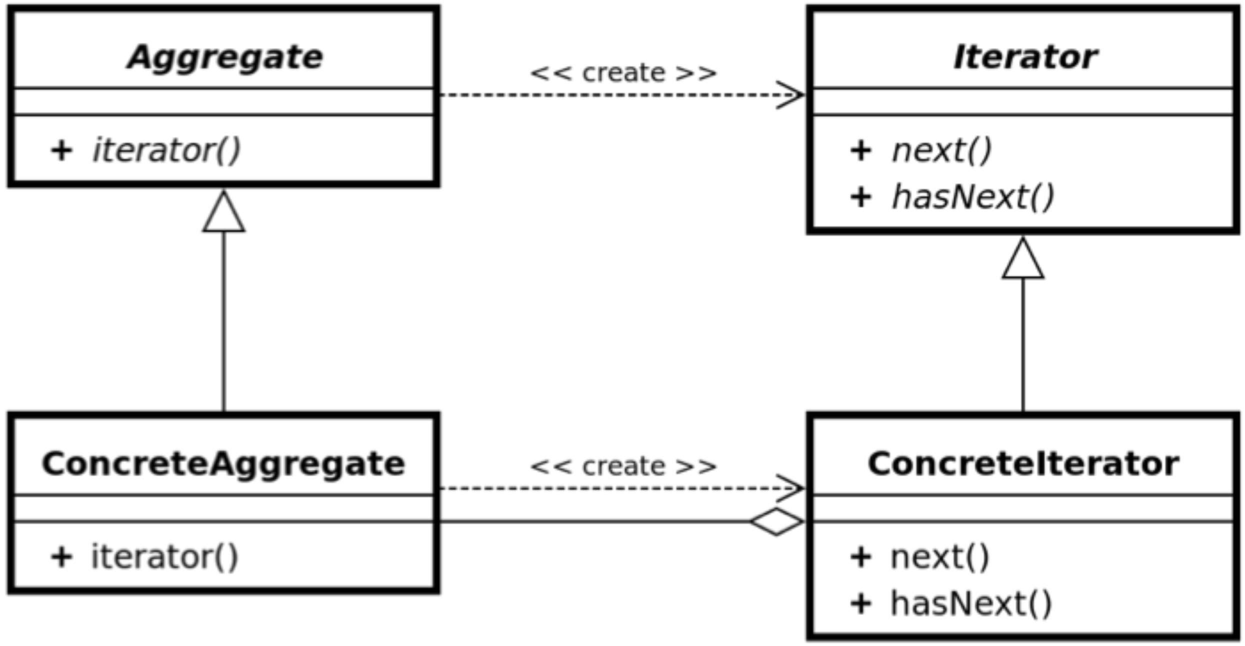
- Design goals and design principles

The Iterator Pattern



- **Problem**: Clients need uniform strategy to access all elements in a container, independent of the container type
  - All items in a list, set, tree; the Fibonacci numbers; all permutations of a set
  - Order is unspecified, but access every element once
- **Solution**: A strategy pattern for iteration
- **Consequences**:
  - Hides internal implementation of underlying container
  - Easy to change container type
  - Facilitates communication between parts of the program

The Iterator Pattern



- Interface:
  - hasNext()
  - next()

Example: `while (i.hasNext()) { x = i.next(); process(x); }`
or `for (x : i) { process(x); }`

Traversing a collection in Java

- Old-school Java for loop for ordered types

```
List<String> arguments = …;
for (int i = 0; i < arguments.size(); i++) {
  System.out.println(arguments.get(i));
}
```

- Modern standard Java for-each loop

```
List<String> arguments = …;
for (String s : arguments) {
  System.out.println(s);
}
```

Works for every implementation of `Iterable`:

```
public interface Iterable<E> {
   public Iterator<E> iterator();
}
```

Iterators in Java

```
interface Iterable<E> {//implemented by most collections
        Iterator<E> iterator();
}
interface Iterator<E> {
        boolean hasNext();
        E next();
        void remove();          // removes previous returned item
}                               // from the underlying collection
```

An Iterator implementation for Pairs

```
public class Pair<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second=s;}

}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second=s;}
  public Iterator<E> iterator() {
    return new PairIterator();
  }
  private class PairIterator implements Iterator<E> {
    private boolean seenFirst = false, seenSecond = false;
    public  boolean hasNext() { return !seenSecond; }
    public  E next() {
      if (!seenFirst)  { seenFirst  = true; return first;  }
      if (!seenSecond) { seenSecond = true; return second; }
      throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

Fibonacci Iterator

```
class FibIterator implements Iterator<Integer> {
      public  boolean hasNext() {             }


      public  Integer next() {






      }
      public void remove() {
         throw new UnsupportedOperationException();
      }
}
```

Fibonacci Iterator

```
class FibIterator implements Iterator<Integer> {
      public  boolean hasNext() { return true; }
      private int a = 1;
      private int b =1;
      public  Integer next() {
            int result = a;
            a = b;
            b = a + result;
            return result;
      }
      public void remove() {
         throw new UnsupportedOperationException();
      }
}
```

Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`

Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
  - If you simply want to remove an item:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  if (s.equals("Charlie"))
    arguments.remove("Charlie"); // runtime error
}
```

Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable…
- …but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
  - If you simply want to remove an item:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  if (s.equals("Charlie"))
    it.remove();
}
```

Today

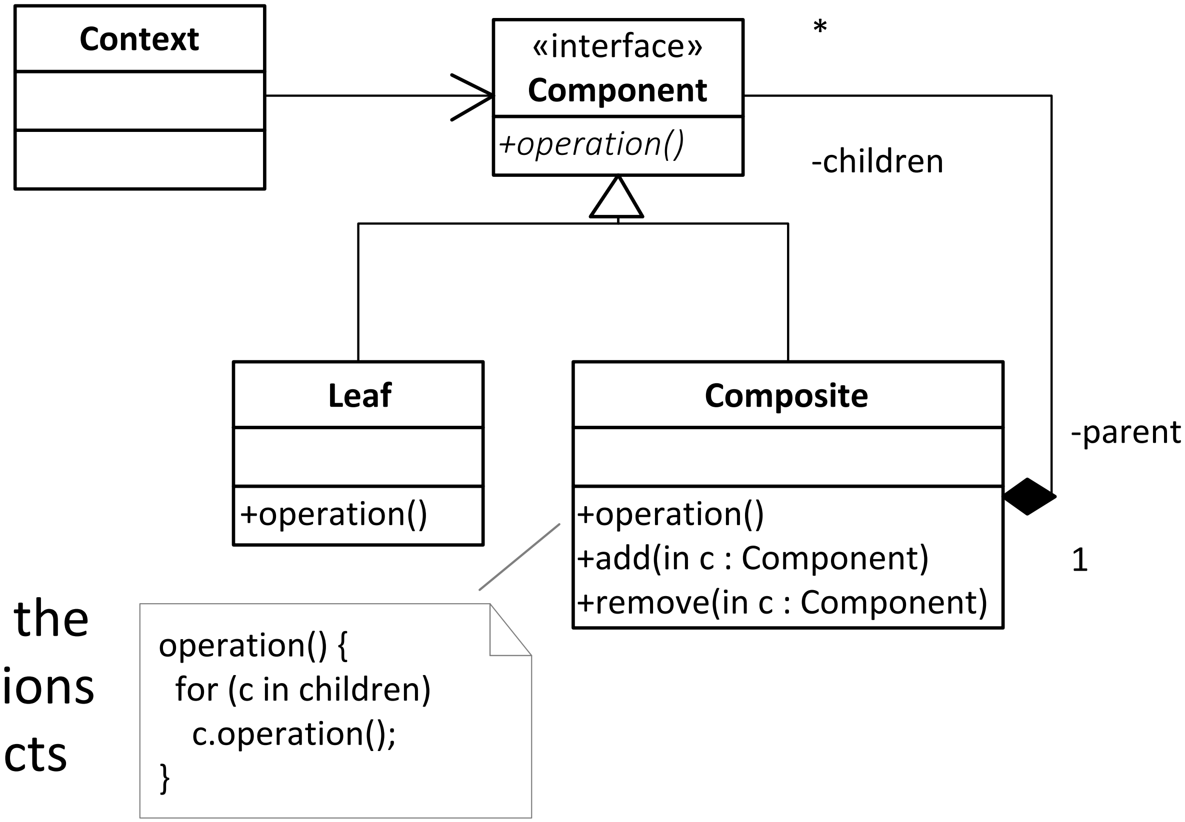
- More design patterns for reuse:
  - ~~The Iterator design pattern (left overs from Thu)~~
  - The Composite design pattern
  - The Decorator design pattern

- Design goals and design principles

Design Exercise (on paper)

- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

The Composite Design Pattern



Applicability

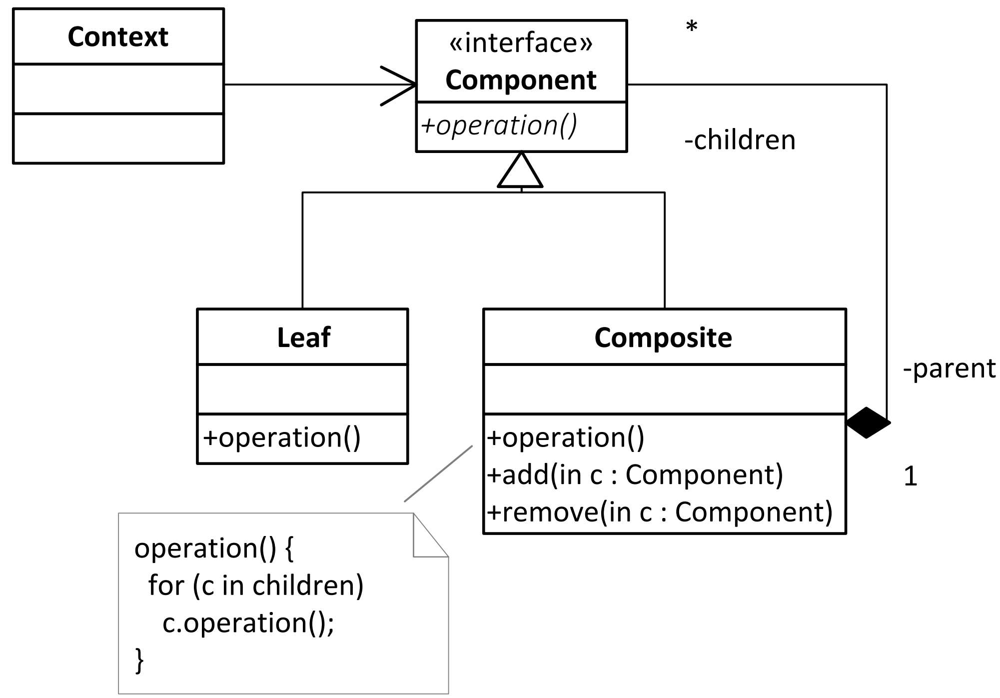
- You want to represent part-whole hierarchies of object
- You want to be able to ignore the difference between compositions of objects and individual objects

- **Consequences**
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components

```
interface Item {
    double getWeight();
}

class Letter implements Item {
    double weight;
    double getWeight() {...}
}

class Box implements Item {
    ArrayList<Item> items=new ArrayList<>();
    double getWeight() {
        double weight = 0.0
        for(Item item : items) {
            weight += item.getWeight();
        }
    }
    void add(Item item){
        items.add(item);
    }
}
```



Today

- More design patterns for reuse:
  - ~~The Iterator design pattern (left overs from Thu)~~
  - ~~The Composite design pattern~~
  - The Decorator design pattern

- Design goals and design principles

Limitations of inheritance

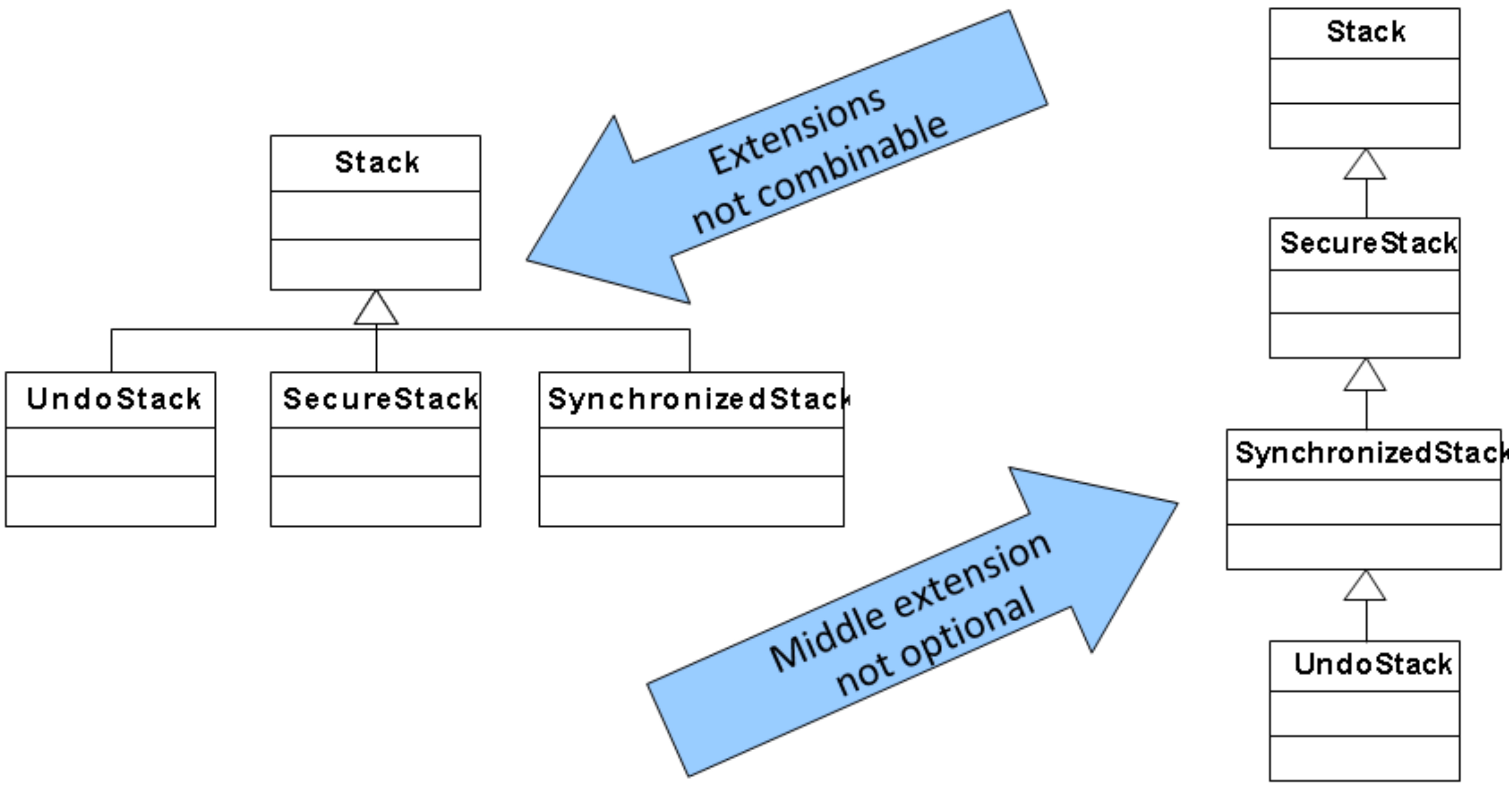
- Suppose you want various extensions of a `Stack` data structure…
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses

Limitations of inheritance

- Suppose you want various extensions of a `Stack` data structure…
  - `UndoStack`: A stack that lets you undo previous push or pop operations
  - `SecureStack`: A stack that requires a password
  - `SynchronizedStack`: A stack that serializes concurrent accesses
  - `SecureUndoStack`: A stack that requires a password, and also lets you undo previous operations
  - `SynchronizedUndoStack`: A stack that serializes concurrent accesses, and also lets you undo previous operations
  - `SecureSynchronizedStack`: …
  - `SecureSynchronizedUndoStack`: …

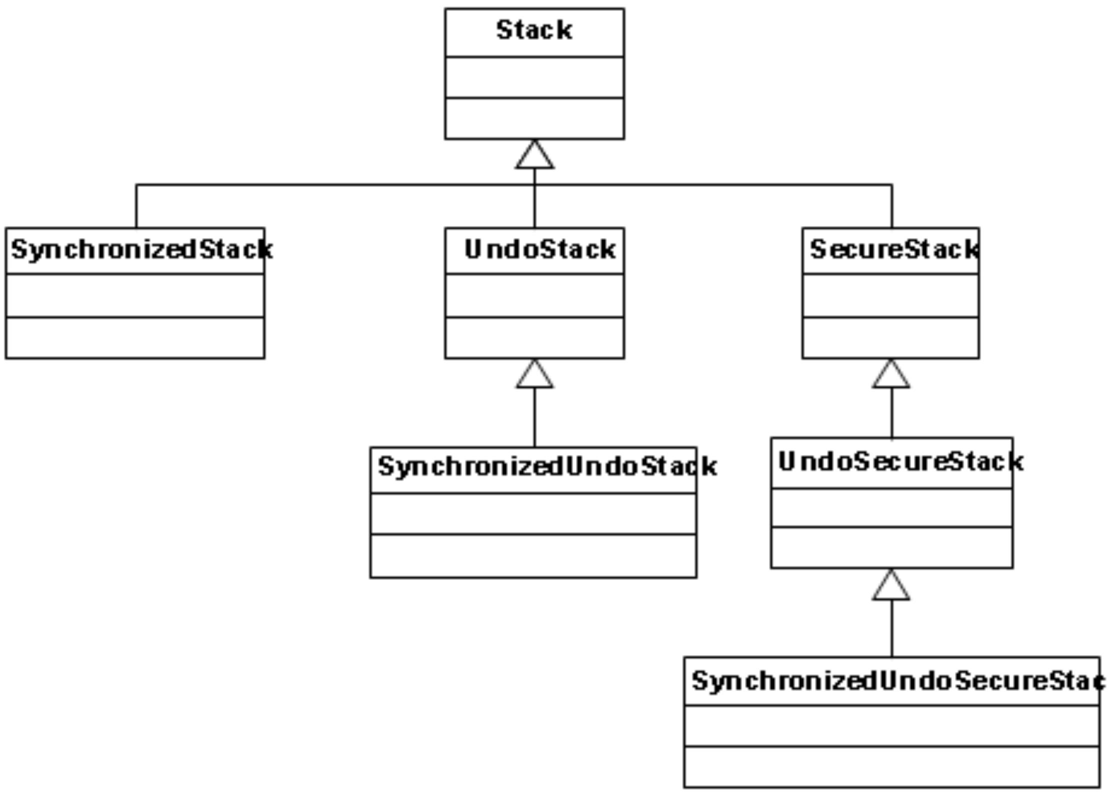
Goal: arbitrarily composable extensions

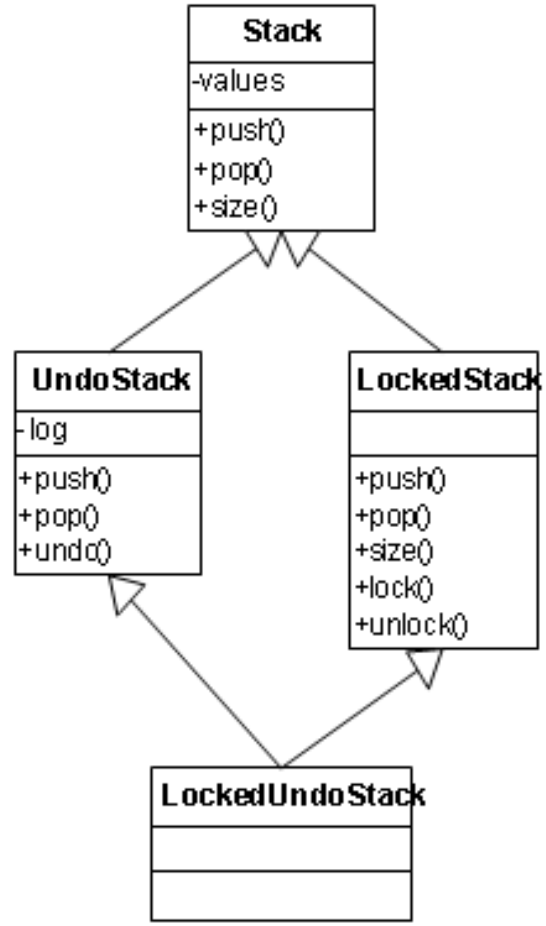
Limitations of inheritance



Workarounds?

- Combining inheritance hierarchies
  - Combinatorical explosion
  - Massive code replication

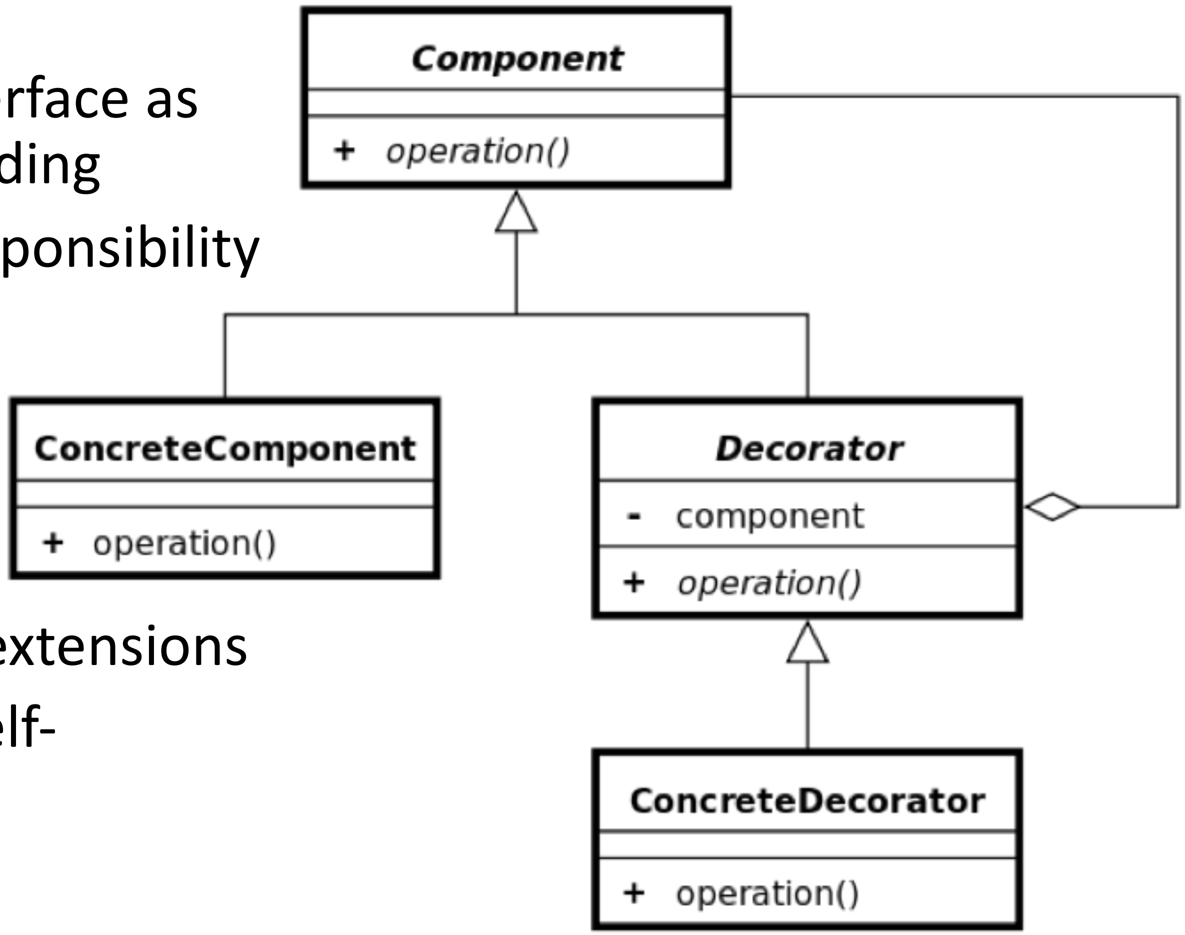




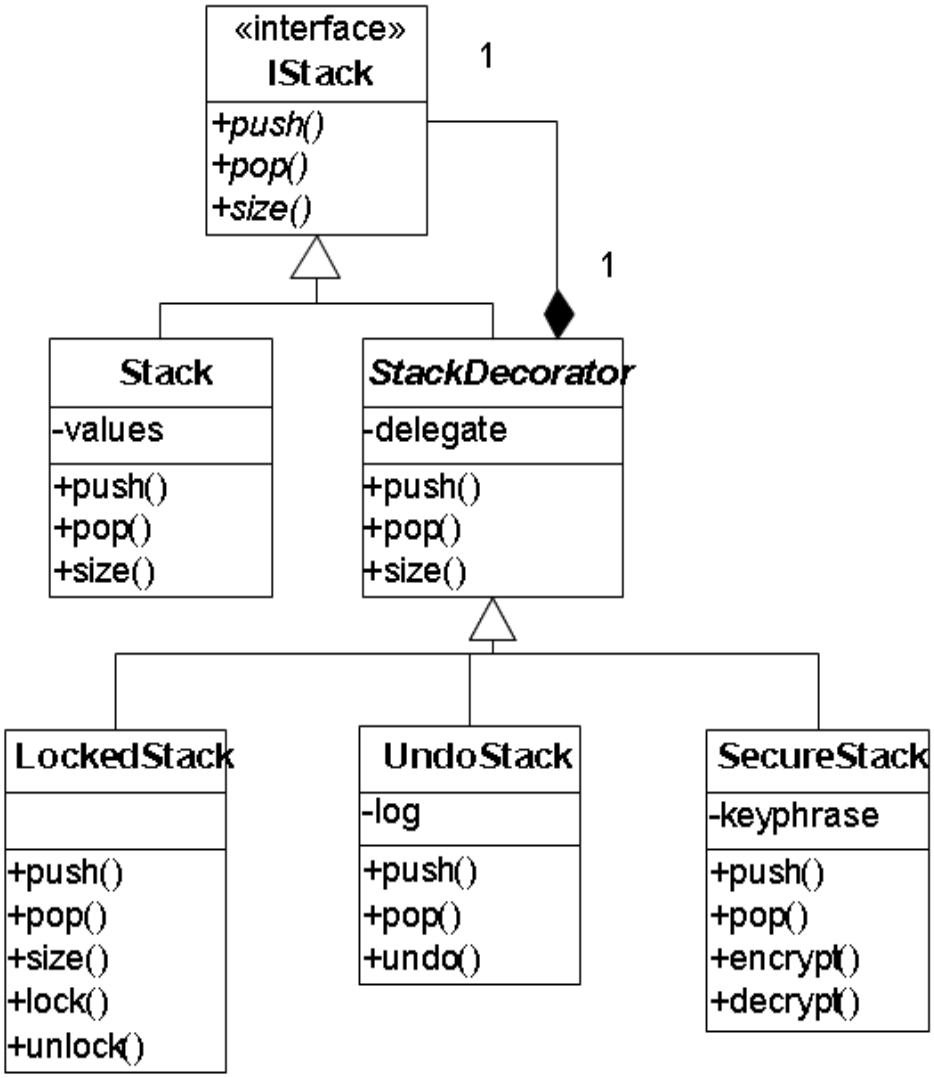
Multiple inheritance
- Diamond problem

The *Decorator* design pattern

- **Problem**: Need arbitrary / dynamically composable extensions to individual objects.
- **Solution**:
  - Implement common interface as the object you are extending
  - But delegate primary responsibility to an underlying object.
- **Consequences**:
  - More flexible than static inheritance
  - Customizable, cohesive extensions
  - Breaks object identity, self-references



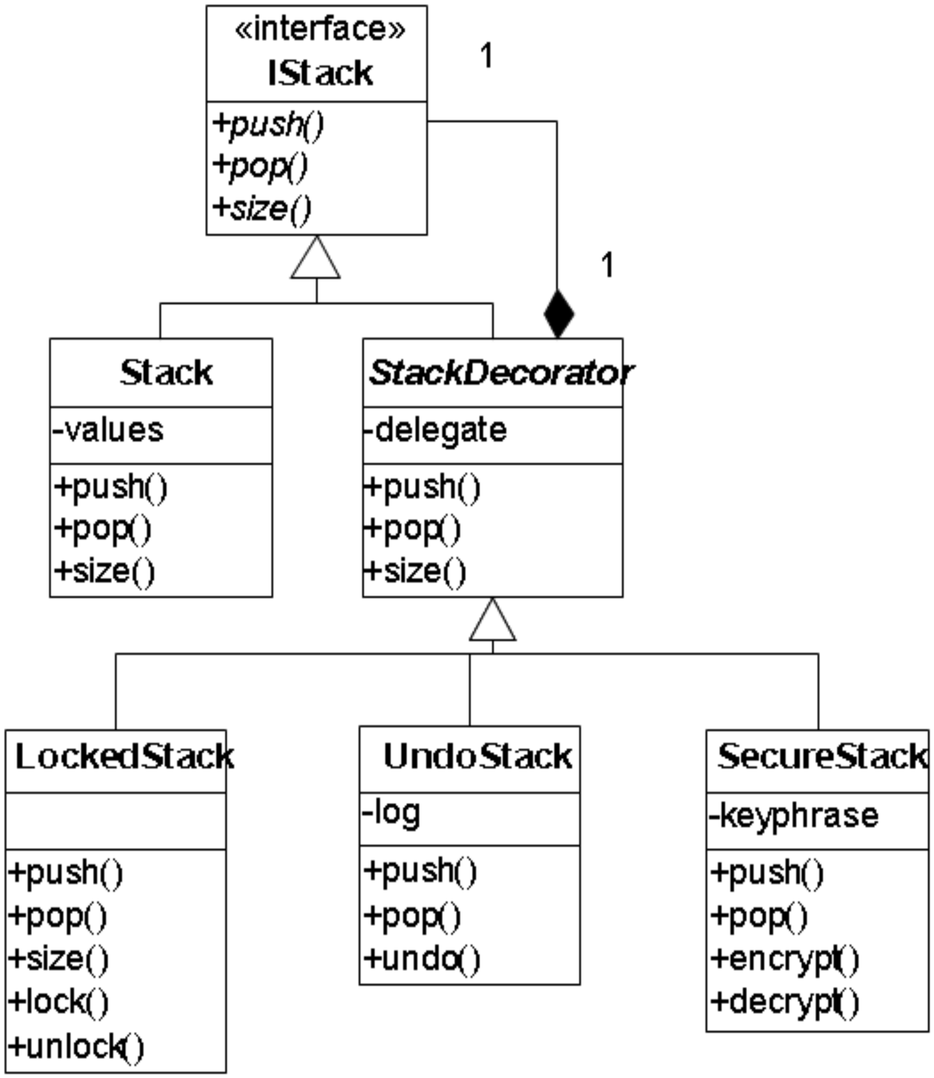
Using the Decorator for our Stack example



The abstract forwarding class

```
public abstract class StackDecorator
           implements IStack {
  private final IStack stack;
  public StackDecorator(IStack stack) {
      this.stack = stack;
  }
  public void push(Item e) {
        stack.push(e);
  }
  public Item pop() {
        return stack.pop();
  }
  …
}
```

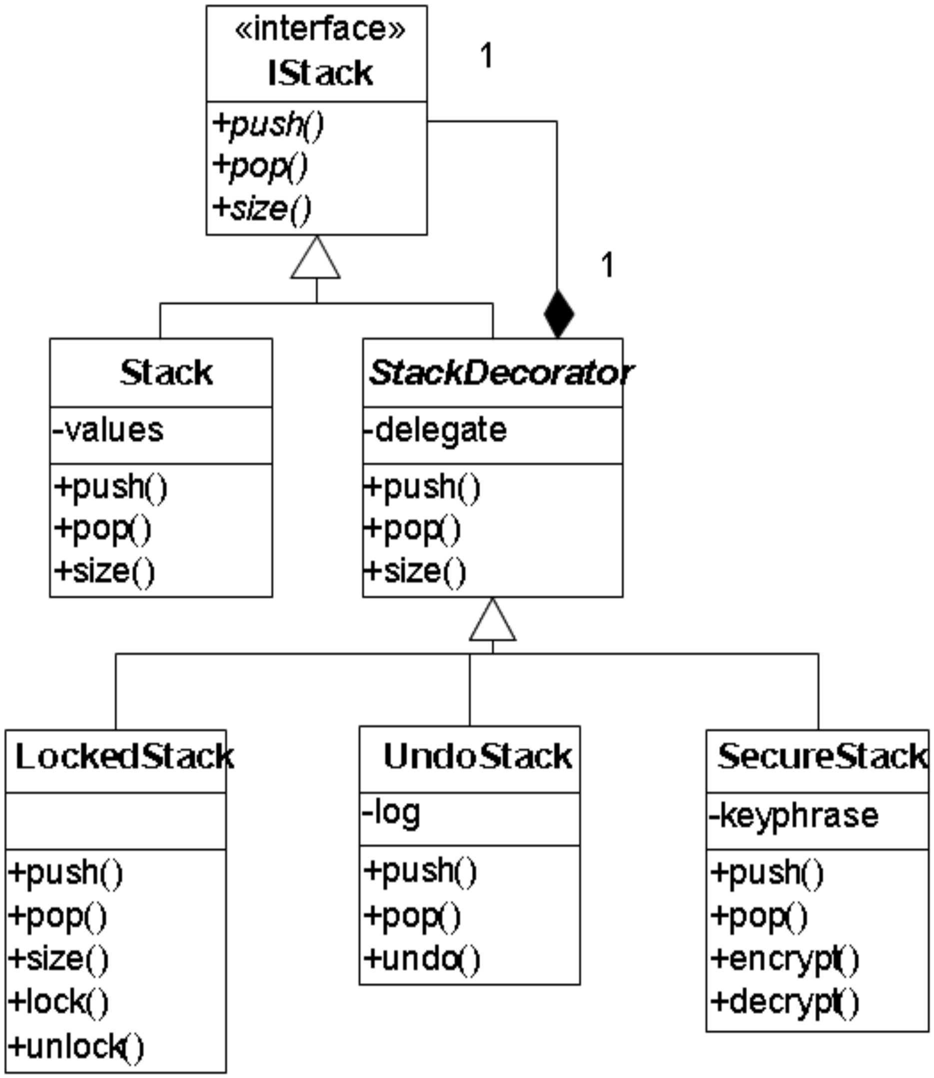
Using the Decorator for our Stack example



A concrete decorator class

```
public class UndoStack
        extends StackDecorator
        implements IStack {
  private final UndoLog log = new UndoLog();
  public UndoStack(IStack stack) {
        super(stack);
  }
  public void push(Item e) {
        log.append(UndoLog.PUSH, e);
        super.push(e);
  }
  …
}
```

Using the Decorator for our Stack example



Using the decorator classes

- To construct a plain stack:

```
Stack s = new Stack();
```

- To construct an plain undo stack:

```
UndoStack s = new
    UndoStack(new Stack());
```

- To construct a secure synchronized undo stack:

```
SecureStack s = new
    SecureStack(new
    SynchronizedStack(new
    UndoStack(new Stack())));
```

Decorators from `java.util.Collections`

- Turn a mutable list into an immutable list:

```
static List<T>  unmodifiableList(List<T>  lst);
static Set<T>   unmodifiableSet( Set<T>   set);
static Map<K,V> unmodifiableMap( Map<K,V> map);
```

- Similar for synchronization:

```
static List<T>  synchronizedList(List<T>  lst);
static Set<T>   synchronizedSet( Set<T>   set);
static Map<K,V> synchronizedMap( Map<K,V> map);
```

The decorator pattern vs. inheritance

- Decorator composes features at run time
  - Inheritance composes features at compile time
- Decorator consists of multiple collaborating objects
  - Inheritance produces a single, clearly-typed object
- Can mix and match multiple decorations
  - Multiple inheritance has conceptual problems

Summary

- Use UML class diagrams to simplify communication
- Design patterns…
  - Convey shared experience, general solutions
  - Facilitate communication
- Specific design patterns for reuse:
  - Strategy
  - Template method
  - Iterator
  - Composite
  - Decorator