Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Design patterns for reuse (continued) and design heuristics

Charlie Garrod       Michael Hilton
Administrivia

- Reading due today: UML and Patterns Chapters 9 and 10
- Optional reading for Thursday:
  - UML and Patterns Chapter 17
  - Effective Java items 39, 43, and 57ff
- Homework 3 due Thursday at 11:59 p.m.
- Midterm exam next Thursday (September 28th)
  - Review session, practice exam info 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)
• Basic best practices...
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?"
Strategy pattern

- **Problem:** Clients need different variants of an algorithm
- **Solution:** Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- **Consequences:**
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes:
    - Code can be harder to understand
    - Lots of overhead if the strategies are simple
Avoiding `instanceof` with the template method pattern

```java
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:

```java
public void doSomething(Account acct) {
    long adj = acct.getMonthlyAdjustment();
    ...
}
```
The abstract java.util.AbstractList<E>

abstract T get(int i);
abstract int size();
boolean set(int i, E e); // pseudo-abstract
boolean add(E e); // pseudo-abstract
boolean remove(E e); // pseudo-abstract
boolean addAll(Collection<? extends E> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
boolean contains(E e);
boolean containsAll(Collection<?> c);
void clear();
boolean isEmpty();
abstract Iterator<E> iterator();
Object[] toArray()
<T> T[] toArray(T[] a);
...

Today

• More design patterns for reuse
  – Iterator pattern
  – Decorator pattern

• Design goals and design principles
Traversing a collection

• Old-school Java for loop for ordered types
  
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); i++) {
    System.out.println(arguments.get(i));
  }
  ```

• Modern standard Java for-each loop
  
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
    System.out.println(s);
  }
  ```

Works for every implementation of Iterable:

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

```java
public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
    void remove();  // removes previous returned item
}  // from the underlying collection

• To use explicitly, e.g.:
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
}
```
Iterator design pattern

• Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  – 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
Getting an Iterator

public interface Collection<E> extends Iterable<E> {
    boolean add(E e);
    boolean addAll(Collection<? extends E> c);
    boolean remove(Object e);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    boolean contains(Object e);
    boolean containsAll(Collection<?> c);
    void clear();
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    ...
}

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.
An Iterator implementation for Pairs

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

```java
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
```
An Iterator implementation for Pairs

```java
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) { ... }
```
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:
    ```java
    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:
    ```java
    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
  – Iterator pattern
  – Decorator 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

Extensions not combinable

Middle extension not optional
Workarounds?

- Combining inheritance hierarchies?
- Multiple inheritance?
The decorator design pattern

- **Problem:** You need arbitrary or dynamically composable extensions to individual objects.
- **Solution:** Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object.
- **Consequences:**
  - More flexible than static inheritance
  - Customizable, cohesive extensions
  - Breaks object identity, self-references
Decorators use both subtyping and delegation

```java
public class LoggingList<E> implements List<E> {
    private final List<E> list;
    public LoggingList<E>(List<E> list) { this.list = list; }
    public boolean add(E e) {
        System.out.println("Adding " + e);
        return list.add(e);
    }
    public E remove(int index) {
        System.out.println("Removing at " + index);
        return list.remove(index);
    }
    ...
```
The AbstractStackDecorator forwarding class

```java
public abstract class AbstractStackDecorator
    implements Stack {
    private final Stack stack;
    public AbstractStackDecorator(Stack stack) {
        this.stack = stack;
    }
    public void push(Item e) {
        stack.push(e);
    }
    public Item pop() {
        return stack.pop();
    }
    ...
}
```
The concrete decorator classes

```java
public class UndoStack extends AbstractStackDecorator
    implements Stack {
    private final UndoLog log = new UndoLog();
    public UndoStack(Stack stack) {
        super(stack);
    }
    public void push(Item e) {
        log.append(UndoLog.PUSH, e);
        super.push(e);
    }
    ...
}
```
Using the decorator classes

• To construct a plain stack:
   `Stack stack = new ArrayStack();`

• To construct an undo stack:
Using the decorator classes

- To construct a plain stack:
  ```java
  Stack stack = new ArrayStack();
  ```
- To construct an undo stack:
  ```java
  UndoStack stack = new UndoStack(new ArrayStack());
  ```
Using the decorator classes

• To construct a plain stack:
  Stack stack = new ArrayStack();
• To construct an undo stack:
  UndoStack stack = new UndoStack(new ArrayStack());
• To construct a secure synchronized undo stack:
Using the decorator classes

- To construct a plain stack:
  Stack $s =$ new ArrayStack();
- To construct an undo stack:
  UndoStack $s =$ new UndoStack(new ArrayStack());
- To construct a secure synchronized undo stack:
  SecureStack $s =$ new SecureStack(new SynchronizedStack(
    new UndoStack(new ArrayStack())));
Decorators from java.util.Collections

- Turn a mutable collection into an immutable collection:
  
  ```java
  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:
  
  ```java
  static List<T> synchronizedList(List<T> lst);
  static Set<T> synchronizedSet(Set<T> set);
  static Map<K,V> synchronizedMap(Map<K,V> map);
  ```
The UnmodifiableCollection (simplified excerpt)

```java
public static <T> Collection<T> unmodifiableCollection(Collection<T> c)
    return new UnmodifiableCollection<>(c);
}
class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    final Collection<E> c;
    UnmodifiableCollection(Collection<> c) {this.c = c; }
    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) {throw new UnsupportedOperationException();}
    public boolean containsAll(Collection<?> coll) {return c.containsAll(coll);
    public boolean addAll(Collection<? extends E> coll) {throw new UnsupportedOperationException();
    public boolean removeAll(Collection<?> coll) {throw new UnsupportedOperationException();
    public boolean retainAll(Collection<?> coll) {throw new UnsupportedOperationException();
    public void clear() {throw new UnsupportedOperationException();
```
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 is conceptually difficult
Today

• More design patterns for reuse
  – Iterator pattern
  – Decorator pattern

• Design goals and design principles
Metrics of software quality

- **Sufficiency / functional correctness**
  - Fails to implement the specifications ... Satisfies all of the specifications

- **Robustness**
  - Will crash on any anomalous event ... Recovers from all anomalous events

- **Flexibility**
  - Must be replaced entirely if spec changes ... Easily adaptable to changes

- **Reusability**
  - Cannot be used in another application ... Usable without modification

- **Efficiency**
  - Fails to satisfy speed or storage requirement ... satisfies requirements

- **Scalability**
  - Cannot be used as the basis of a larger version ... is an outstanding basis...

- **Security**
  - Security not accounted for at all ... No manner of breaching security is known

Source: Braude, Bernstein, Software Engineering. Wiley 2011
Design principles

- Low coupling
- Low representational gap
- High cohesion
A design principle for reuse: low coupling

- Each component should depend on as few other components as possible

- Benefits of low coupling:
  - Enhances understandability
  - Reduces cost of change
  - Eases reuse
Design principles to be continued Thursday..