Principles of Software Construction: Objects, Design, and Concurrency

Designing classes

Design patterns for reuse

Josh Bloch  Charlie Garrod
Administrivia

• Homework 3 due Sunday, February 7\textsuperscript{th}
  
  SEND
  
  + MORE
  
  \hline
  
  MONEY

• Human cryptarithm solving by Josh
  – Sunday, January 31\textsuperscript{st}, 3:30 p.m. in Wean 5302

• Upcoming office hours...
Key concepts from Tuesday...
Inheritance

```java
public abstract class AbstractAccount implements Account {
    protected long balance = 0;
    public long getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public long getFee() { ... }
}
```

```
«interface» Account
getBalance() : long
deposit(amount : long)
withdraw(amount : long) : boolean
transfer(amount : long,
    target : Account) : boolean
monthlyAdjustment()

«interface» CheckingAccount
getFee() : long

AbstractAccount
# balance : long
+ getBalance() : long
+ deposit(amount : long)
+ withdraw(amount : long) : boolean
+ transfer(amount : long,
    target : Account) : boolean
+ monthlyAdjustment()

CheckingAccountImpl
monthlyAdjustment()
getFee() : long
```

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```
What Does It Print?

(a) 0.9  
(b) 0.90  
(c) It varies  
(d) None of the above

public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}

What Does It Print?

(a) 0.9
(b) 0.90
(c) It varies
(d) None of the above: 0.8999999999999999

Decimal values can't be represented exactly by float or double
Another Look

```
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```
How Do You Fix It?

// You could fix it this way...
import java.math.BigDecimal;
public class Change {
    public static void main(String args[]) {
        System.out.println(
            new BigDecimal("2.00").subtract(
                new BigDecimal("1.10")));
    }
}

// ...or you could fix it this way
public class Change {
    public static void main(String args[]) {
        System.out.println(200 - 110);
    }
}
The Moral

• Avoid float and double where exact answers are required
  – For example, when dealing with money
• Use BigDecimal, int, or long instead
2. “A Change is Gonna Come”

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}

(a) 0.9
(b) 0.90
(c) 0.8999999999999999
(d) None of the above
What Does It Print?

(a) 0.9
(b) 0.90
(c) 0.8999999999999999
(d) None of the above:
0.899999999999999911182158029987476766109466552734375

We used the wrong BigDecimal constructor
Another Look

The spec says:

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```

The spec says:

```java
    public BigDecimal(double val)
    Translates a double into a BigDecimal which is
    the exact decimal representation of the double's
    binary floating-point value.
```
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal("2.00");
        BigDecimal cost = new BigDecimal("1.10");
        System.out.println(payment.subtract(cost));
    }
}

Prints 0.90
The Moral

- Use `new BigDecimal(String)`, not `new BigDecimal(double)`
- `BigDecimal.valueOf(double)` is better, but not perfect
  - Use it for non-constant values.
- For API designers
  - Make it easy to do the commonly correct thing
  - Make it possible to do exotic things
Key concepts from Tuesday...
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);
    }
    ...
}
Design with inheritance (or not)

• Favor composition over inheritance
  – Inheritance violates information hiding

• Design and document for inheritance, or prohibit it
  – Document requirements for overriding any method
Behavioral subtyping

Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be provable for objects $y$ of type $S$ where $S$ is a subtype of $T$.  

Barbara Liskov

• e.g., Compiler-enforced rules in Java:
  – Subtypes can add, but not remove methods
  – Concrete class must implement all undefined methods
  – Overriding method must return same type or subtype
  – Overriding method must accept the same parameter types
  – Overriding method may not throw additional exceptions

• Also applies to specified behavior:
  – Same or stronger invariants
  – Same or stronger postconditions for all methods
  – Same or weaker preconditions for all methods

This is called the *Liskov Substitution Principle*. 
Behavioral subtyping (Liskov Substitution Principle)

class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant fuel >= 0;
    //@ requires fuel > 0 && !engineOn;
    //@ ensures engineOn;
    void start() { … }

    void accelerate() { … }
    //@ requires speed != 0;
    //@ ensures speed < old(speed)
    void brake() { … }
}

class Hybrid extends Car {
    int charge;
    //@ invariant charge >= 0;
    //@ requires (charge > 0 || fuel > 0) && !engineOn;
    //@ ensures engineOn;
    void start() { … }

    void accelerate() { … }
    //@ requires speed != 0;
    //@ ensures speed < old(speed)
    //@ ensures charge > \old(charge)
    void brake() { … }
}

Subclass fulfills the same invariants (and additional ones)
Overridden method start has weaker precondition
Overridden method brake has stronger postcondition
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    int h, w;

    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }

    //methods
}

class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
    int h, w;
    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }
    //methods
}

class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}

Is this Square a behavioral subtype of Rectangle? (Yes.)
Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //methods
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

Is this Square a behavioral subtype of Rectangle? (Yes.)
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //@ requires factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }
}

//@ invariant h>0 && w>0;
//@ invariant h==w;
class Square extends Rectangle {
    Square(int w) {
        super(w, w);
    }
}
```

Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }

    //@ requires factor > 0;
    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

Is this Square a behavioral subtype of Rectangle? (Yes.)
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
   //@ invariant h>0 && w>0;
   int h, w;

   Rectangle(int h, int w) {
      this.h=h; this.w=w;
   }

   //@ requires neww > 0;
   //@ ensures w = neww;
   //@ ensures h = \old(h);
   void setWidth(int neww) {
      w = neww;
   }

   }
}

class Square extends Rectangle {
   //@ invariant h>0 && w>0;
   //@ invariant h==w;
   Square(int w) {
      super(w, w);
   }

   //@ requires neww > 0;
   //@ ensures w = neww;
   //@ ensures h = neww;
   void setWidth(int neww) {
      w = neww;
      h = neww;
   }

   }

Is this Square a behavioral subtype of Rectangle?
Behavioral subtyping (Liskov Substitution Principle)

class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h = h; this.w = w;
    }

    //@ requires neww > 0;
    //@ ensures w = neww;
    //@ ensures h = \old(h);
    void setWidth(int neww) {
        w = neww;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }

    //@ requires neww > 0;
    //@ ensures w = neww;
    //@ ensures h = neww;
    void setWidth(int neww) {
        w = neww;
        h = neww;
    }
}

Is this Square a behavioral subtype of Rectangle?
No: Overridden setWidth violates postcondition
Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h=h; this.w=w;
    }

    //@ requires neww > 0;
    //@ ensures w = neww;
    //@ ensures h = \old(h);
    void setWidth(int neww) {
        w = neww;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

Behavioral subtyping (Liskov Substitution Principle)

```java
class Rectangle {
    //@ invariant h>0 && w>0;
    int h, w;

    Rectangle(int h, int w) {
        this.h = h; this.w = w;
    }

    //@ requires neww > 0;
    //@ ensures w = neww;
    //@ ensures h = \old(h);
    void setWidth(int neww) {
        w = neww;
    }
}

class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}

class GraphicProgram {
    void scaleW(Rectangle r, int factor) {
        r.setWidth(r.getWidth() * factor);
    }
}

Yes? (But the Square is not actually a square...)
```

← Invalidates stronger invariant (w==h) in subclass
Behavioral subtyping summary

- Applying the Liskov Substitution Principle can be subtle
- Design goal: simplicity
Learning goals for today

• Be able to motivate, describe, and use the template method and decorator patterns
Today: Design patterns for class-level reuse

• An Evening of Puzzlers
• Behavioral subtyping: Liskov's Substitution Principle
• Design patterns for class-level reuse
  – Template method pattern
  – Decorator pattern
  – (time permitting) Iterator pattern
Recall instanceof

• Operator that tests whether an object is of a given class
  public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
      checkingAcct = (CheckingAccount) acct;
      adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
      savingsAcct = (SavingsAccount) acct;
      adj = savingsAcct.getInterest();
    }
    ...
  }

• Advice: avoid instanceof if possible
  – Never(?) use instanceof in a superclass to check type against subclass

Warning: This code is bad.
Recall `instanceof`

- Operator that tests whether an object is of a given class

```java
public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
        adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
        adj = savingsAcct.getInterest();
    } else if (acct instanceof InterestCheckingAccount) {
        icAccount = (InterestCheckingAccount) acct;
        adj = icAccount.getInterest();
        adj -= icAccount.getFee();
    }
    ...
}
```

**Warning:** This code is bad.
Avoiding instanceof with the template method pattern

```java
public interface Account {
    ...
    public long getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    ...
    public long getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    ...
    public long getMonthlyAdjustment() {
        return getInterest();
    }
}```
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 template method design pattern
The template method design pattern

• **Applicability**
  – Algorithm consists of customizable and invariant parts
  – Limits subclass extensions to specific operations

• **Consequences**
  – Code reuse for invariant parts of algorithm
  – Inverted Hollywood-style control for customization
Template method vs. the strategy pattern

- Both support variations in larger common context
- Template method uses inheritance + an overridable method
- Strategy uses an interface and polymorphism (via composition)
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class
Today: Design patterns for class-level reuse

• An Evening of Puzzlers
• Behavioral subtyping: Liskov's Substitution Principle
• Design patterns for class-level reuse
  – Template method pattern
  – Decorator pattern
  – (time permitting) Iterator pattern
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 or LockedStack: 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 or LockedStack: 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
  – Combinatorial explosion
  – Massive code replication

• Multiple inheritance
  – Diamond problem
The decorator design pattern
Decorators use both subtyping and delegation

```java
public class LoggingList<E> implements List<E> {
    private final List<E> list;
    public LoggingList(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);
    }
    ...
```
Using the decorator pattern for the stack

```
<<interface>>
IStack
+push()
+pop()
+size()

Stack
-values
+push()
+pop()
+size()

StackDecorator
-delegate
+push()
+pop()
+size()

LockedStack

UndoStack
-log
+push()
+pop()
+undo()

SecureStack
-keyphrase
+push()
+pop()
+encrypt()
+decrypt()
```
The StackDecorator forwarding class

    public abstract class StackDecorator implements IStack {
        private final IStack delegate;
        public StackDecorator(IStack d) { this.delegate = d; }
        public void push(Object e) {
            delegate.push(e);
        }
        public Object pop() {
            return delegate.pop();
        }
        ...
    }
The concrete decorator classes

public class UndoStack extends StackDecorator
    implements IStack {
    private final UndoLog log = new UndoLog();
    public UndoStack(IStack d) { super(d); }
    public void push(Object e) {
        log.append(UndoLog.PUSH, e);
        super.push(e);
    }
    ... 
}
Using the decorator classes

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

• To construct an undo stack:
Using the decorator classes

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

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

• To construct a plain stack:
  \[
  \text{Stack } s = \text{new } \text{Stack}();
  \]

• To construct an undo stack:
  \[
  \text{UndoStack } s = \text{new } \text{UndoStack(new Stack())};
  \]

• To construct a secure synchronized undo stack:
  \[
  \text{SecureStack } s = \text{new } \text{SecureStack(new SynchronizedStack(new UndoStack(new Stack())))};
  \]
The decorator design pattern

• **Applicability**
  – To dynamically add responsibilities to individual objects
  – For responsibilities that can be withdrawn
  – When extension by subclassing is impractical

• **Consequences**
  – More flexible than static inheritance
  – Avoids monolithic classes
  – Breaks object identity, self-references
  – Lots of little objects
Decorators from java.util.Collections

- Turn a mutable list into an immutable list:
  ```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) {return false;}
    public boolean addAll(Collection<? extends E> coll) { return false; }
    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 single, clearly-typed object
• Can mix and match multiple decorations
  – Multiple inheritance is conceptually difficult
Summary: Design patterns for class-level reuse

- Behavioral subtyping can be hard
- Template method pattern to create customizable algorithms
- Decorator pattern to mix and match decorations