Principles of Software Construction: Objects, Design, and Concurrency

Designing classes

Introduction to design patterns

Josh Bloch    Charlie Garrod
Administrivia

• Homework 2 due tonight
• Reading due next Tuesday
  – Effective Java: #21, 38, 39, 40, 41, 44
• Homework 3 due Sunday, September 25th
  SEND
  + MORE
  --------
  MONEY
• Human cryptarithmetic solving by Josh
  – Monday, September 19th 6:30 p.m. (location TBD)

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);
    }
}
```

From An Evening Of Puzzlers by Josh Bloch
What does it print?

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

```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: 0.8999999999999999

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

```java
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));
    }
}
```
What does it print?

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

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));
    }
}
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
public BigDecimal(double val)
```

Translates a double into a BigDecimal which is the exact decimal representation of the double's binary floating-point value.

```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));
    }
}

How do you fix it?

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...
Using delegation to extend functionality

• One solution:

```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 LoggingList is composed of a List, and delegates (the non-logging) functionality to that List
Implementation inheritance for code reuse

```
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() { ... }
}
```
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 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 factor > 0;
    void scale(int factor) {
        w=w*factor;
        h=h*factor;
    }

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

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?
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;
    }

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

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

← Invalidates stronger invariant (w==h) in subclass

Yes?! (But the Square is not a square...
Today: Introduction to Design Patterns

• A Java Puzzler
• Introduction to UML class diagrams
• Introduction to design patterns
  – Strategy pattern
• Specific design patterns for change and reuse:
  – Template method pattern
  – Iterator pattern
  – Decorator pattern (next Tuesday)
UML: Unified Modeling Language
UML: Unified Modeling Language

Diagram:
- Citizen
- Foreigner
- Person
  - Nationality
  - Role
- Male
- Female
- Student
  - Degree type
    - Undergrad Student
    - Masters Student
    - PHD Student
- Employee
- Professor
- Registrar
- Associate Professor
- Tenured Professor
UML: Unified Modeling Language
UML: Unified Modeling Language
UML in this course

• UML class diagrams
• UML interaction diagrams
  – Sequence diagrams
  – Communication diagrams
UML class diagrams (interfaces and inheritance)

```java
public interface Account {
    public long getBalance();
    public void deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount extends CheckingAccount, SavingsAccount {
}
```
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() { ... }
}
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...
UML advice

- Best used to show the big picture
  - Omit unimportant details
    - But show they are there: ...

- Avoid redundancy
  - e.g., bad:

  ![Bad UML example](image)

  good:
Today: Introduction to Design Patterns

- A Java Puzzler
- Introduction to UML class diagrams
- Introduction to design patterns
  - Strategy pattern
- Specific design patterns for change and reuse:
  - Template method pattern
  - Iterator pattern
  - Decorator pattern (next Tuesday)
One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.
Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.
A third design scenario

- Suppose we need to sort a list in different orders...

```java
interface Comparator {
    boolean compare(int i, int j);
}

final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
    ...
    boolean mustSwap =
        cmp.compare(list[i], list[j]);
    ...
}
```
Design patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

– Christopher Alexander, Architect (1977)
How not to discuss design (from Shalloway and Trott)

• Carpentry:
  – How do you think we should build these drawers?
  – Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...

• Software Engineering:
  – How do you think we should write this method?
  – I think we should write this if statement to handle ... followed by a while loop ... with a break statement so that...
Discussion with design patterns

• Carpenter:
  – "Is a dovetail joint or a miter joint better here?"

• Software Engineering:
  – "Is a strategy pattern or a template method better here?"
History: *Design Patterns* (1994)
Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences
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
Patterns are more than just structure

• Consider: A modern car engine is constantly monitored by a software system. The monitoring system must obtain data from many distinct engine sensors, such as an oil temperature sensor, an oxygen sensor, etc. More sensors may be added in the future.
Design pattern conclusions

• Provide shared language
• Convey shared experience
• Can be system and language specific
Today: Introduction to Design Patterns

• A Java Puzzler
• Introduction to UML class diagrams
• Introduction to design patterns
  – Strategy pattern
• Specific design patterns for change and reuse:
  – Template method pattern
  – Iterator pattern
  – Decorator pattern (next Tuesday)
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();
    }
    ...
}
```

- 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) {
        CheckingAccount acct;
        adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        SavingsAccount acct;
        adj = savingsAcct.getInterest();
    } else if (acct instanceof InterestCheckingAccount) {
        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 abstract `java.util.AbstractList<E>`

```java
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);
...
Template method pattern

• Problem: An algorithm consists of customizable parts and invariant parts

• Solution: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations

• Consequences
  – Code reuse for the invariant parts of algorithm
  – Customization is restricted to the primitive operations
  – Inverted (Hollywood-style) control for customization
Template method vs. the strategy pattern

- Both support variation in a larger 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: Introduction to Design Patterns

• A Java Puzzler
• Introduction to UML class diagrams
• Introduction to design patterns
  – Strategy pattern
• Specific design patterns for change and reuse:
  – Template method pattern
  – Iterator pattern
  – Decorator pattern (next Tuesday)
Traversing a collection

- Since Java 1.0:
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); ++i) {
    System.out.println(arguments.get(i));
  }
  ```
- Java 1.5: for-each loop
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
    System.out.println(s);
  }
  ```
- For-each loop 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.:
  ```java
  List<String> arguments = ...;
  for (Iterator<String> it = arguments.iterator();
      it.hasNext(); ) {
      String s = it.next();
      System.out.println(s);
  }
  ```
Getting an Iterator

```java
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

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

```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) { ... }
}
```
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
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();
    }
    ```
Conclusions

- UML can be useful
- Design patterns facilitate communication and learning
- Today's design patterns facilitate reuse and change:
  - Strategy pattern
  - Template pattern
  - Iterator pattern