Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Delegation and inheritance

Michael Hilton       Bogdan Vasilescu
Administrivia

• Homework 1 graded soon
• Reading assignment due today: Effective Java Items 17 + 50
  – Optional reading due Thursday
  – Required reading due next Tuesday
• Homework 2 due Thursday 11:59 p.m.
Take out a piece of paper...
In 3 minutes:

1. What is your Andrew ID?
2. What is the main point of one of today's reading assignments (e.g., Effective Java 17)?
3. What is the main point of the other of today's reading assignments (e.g., Effective Java 50)?
1. What is your Andrew ID?

+2 any answer
2. and 3. What are the main points of the reading?

+2 Minimize mutability
+2 Make defensive copies

Write "Graded by" and your Andrew ID on the bottom.
Write their score (x/6) near the top of their paper.
Design goals for your Homework 1 solution?

Functional correctness  Adherence of implementation to the specifications
Robustness            Ability to handle anomalous events
Flexibility           Ability to accommodate changes in specifications
Reusability           Ability to be reused in another application
Efficiency            Satisfaction of speed and storage requirements
Scalability           Ability to serve as the basis of a larger version of the application
Security              Level of consideration of application security

Source: Braude, Bernstein, Software Engineering. Wiley 2011
One Homework 1 solution...

class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double similarityTo(Document d) {
        ... ourText = download(url);
        ... theirText = download(d.url);
        ... ourFreq = computeFrequencies(ourText);
        ... theirFreq = computeFrequencies(theirText);
        return cosine(ourFreq, theirFreq);
    }
    ...
}
Compare to another Homework 1 solution...

class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
    }

    public double similarityTo(Document d) {
        String ourText = download(url);
        String theirText = download(d.url);
        Map<String, Integer> ourFreq = computeFreq(ourText);
        Map<String, Integer> theirFreq = computeFreq(theirText);
        return cosine(ourFreq, theirFreq);
    }
}

class Document {
    private final String url;
    public Document(String url) {
        this.url = url;
        String ourText = download(url);
        frequencies = computeFrequencies(ourText);
        return cosine(frequencies, d.frequencies);
    }
}

Using the Document class

For each url:
    Construct a new Document

For each pair of Documents d1, d2:
    Compute d1.similarityTo(d2)
    ...

• What is the running time of this, for n urls?
Latency Numbers Every Programmer Should Know

*Jeff Dean, Senior Fellow, Google*

<table>
<thead>
<tr>
<th>PRIMITIVE</th>
<th>LATENCY:</th>
<th>ns</th>
<th>us</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10,000</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read 4K randomly from SSD*</td>
<td>150,000</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
<td>500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD*</td>
<td>1,000,000</td>
<td>1,000</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
<td>10,000</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
<td>20,000</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
<td>150,000</td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>
The point

- Constants matter
- Design goals sometimes clearly suggest one alternative
Key concepts from last Thursday
Key concepts from last Thursday

- Testing
- Specifying program behavior: contracts
- Behavioral subtyping
- The `java.lang.Object` contracts
Reminder: Subtype Polymorphism

• A type (e.g. Point) can have many forms (e.g., CartesianPoint, PolarPoint, ...)

• Use interfaces to separate expectations from implementation
Creating Objects

```java
interface Point {
    int getX();
    int getY();
}
Point p = new Point() {
    int getX() { return 3; }
    int getY() { return -10; }
}
```
Classes as Object Templates

```java
interface Point {
    int getX();
    int getY();
}

class CartesianPoint implements Point {
    int x,y;
    Point(int x, int y) {this.x=x; this.y=y;}
    int getX() { return this.x; }
    int getY() { return this.y; }
}

Point p = new CartesianPoint(3, -10);
```
interface Point {
    int getX();
    int getY();
}

class PolarPoint implements Point {
    double len, angle;
    PolarPoint(double len, double angle)
        {this.len=len; this.angle=angle;}
    int getX() { return this.len * cos(this.angle);}
    int getY() { return this.len * sin(this.angle); }
    double getAngle() {...}
}

Point p = new PolarPoint(5, .245);
Reminder: Subtype Polymorphism (cont’d)

• When invoking a method \( p.x() \) the specific implementation of \( x() \) from object \( p \) is executed
  – The executed method depends on the actual object \( p \), i.e., on the runtime type
  – It does not depend on the static type, i.e., how \( p \) is declared

• All implementations of an interface can be used interchangeably

• This allows flexible change (modifications, extensions, reuse) later without changing the client implementation, even in unanticipated contexts
Static types vs dynamic types

• Static type: how is a variable declared
• Dynamic type: what type has the object in memory when executing the program (we may not know until we execute the program)

```java
Point createZeroPoint() {
    if (new Math.Random().nextBoolean())
        return new CartesianPoint(0, 0);
    else    return new PolarPoint(0,0);
}
Point p = createZeroPoint();
p.getX();
p.getAngle();
```
Behavioral subtyping summary

- When subtyping, design and implement carefully
  - Subtype must be substitutable anywhere the supertype could be used

- 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
Contracts and Interfaces

• All objects implementing an interface must adhere to the interface’s contracts
  – Objects may provide different implementations for the same specification
  – Subtype polymorphism: Client only cares about interface, not about the implementation

    p.getX()          s.read()

=> Design for Change
Methods common to all Objects

- equals: returns true if the two objects are “equal”
- hashCode: returns an int that must be equal for equal objects, and is likely to differ for unequal objects
- toString: returns a printable string representation
What does the following code print?

```java
public final class Name {
    private final String first, last;

    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }

    public boolean equals(Name o) {
        return first.equals(o.first) && last.equals(o.last);
    }

    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }

    public static void main(String[] args) {
        Set<Name> s = new HashSet<>();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(s.contains(new Name("Mickey", "Mouse")));
    }
}
```

(a) true  
(b) false  
(c) It varies  
(d) None of the above
What does it print?

(a) true
(b) false
(c) It varies
(d) None of the above

The `Name` class overrides `hashCode` but not `equals`!

The two `Name` instances are thus unequal.
What does the following code print?

```java
public final class Name {
    private final String first, last;

    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first;
        this.last = last;
    }

    public boolean equals(Name o) {
        // Accidental overloading
        return first.equals(o.first) && last.equals(o.last);
    }

    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }

    public static void main(String[] args) {
        Set<Name> s = new HashSet<>();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
```
A correct equals implementation

```java
@Override
public boolean equals(Object o) {
    if (!(o instanceof Name))
        return false;
    Name n = (Name) o;
    return n.first.equals(first) && n.last.equals(last);
}
```
Today

• Design for reuse: delegation and inheritance
Recall our earlier sorting example:

Version A:
```java
static void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] < list[j];
    } else {
        mustSwap = list[i] > list[j];
    }
    ...
}
```

Version B':
```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]);
    ...
}
```
Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
  - e.g. here, the Sorter is delegating functionality to some Comparator
- Judicious delegation enables code reuse

```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]);
    ...
}
```
Delegation

- **Delegation** is simply when one object relies on another object for some subset of its functionality
  - e.g. here, the Sorter is delegating functionality to some Comparator
- Judicious delegation enables code reuse
  - Sorter can be reused with arbitrary sort orders
  - Comparators can be reused with arbitrary client code that needs to compare integers

```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]);
    ...
}
```
Using delegation to extend functionality

- Consider the `java.util.List` (excerpted):

  ```java
  public interface List<E> {
    public boolean add(E e);
    public E remove(int index);
    public void clear();
    ...
  }
  ```

- Suppose we want a list that logs its operations to the console...
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.
Delegation and design

• Small interfaces with clear contracts
• Classes to encapsulate algorithms, behaviors
  – E.g., the Comparator
Recall: bank accounts example

<table>
<thead>
<tr>
<th>«interface» CheckingAccount</th>
<th>«interface» SavingsAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>getFee() : float</td>
<td>getInterestRate() : float</td>
</tr>
</tbody>
</table>
Interface inheritance for an account type hierarchy

```
«interface» Account

getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
monthlyAdjustment()

«interface» CheckingAccount

getFee() : float

«interface» SavingsAccount

getInterestRate() : float

«interface» InterestCheckingAccount

SavingsAccount is a subtype of Account. Account is a supertype of SavingsAccount.

CheckingAccount extends Account. All methods from Account are inherited (copied to CheckingAccount).

If we know we have a CheckingAccount, additional methods are available.

Multiple interface extension
```
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 { }
Recall: Implementation inheritance for code reuse

```
<?interface Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
monthlyAdjustment()

<?interface CheckingAccount
getFee() : float

<?interface SavingsAccount
getInterestRate() : float

CheckingAccountImpl
...
...

<?interface InterestCheckingAccount

InterestCheckingAccountImpl
...
...

SavingsAccountImpl
...
...
```
Implementation inheritance for code reuse

Code duplication
public abstract class AbstractAccount implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}

Better: Reuse abstract account code
Challenge: Can we get good code reuse without inheritance?

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

«interface» CheckingAccount
getFee() : float

«interface» SavingsAccount
getInterestRate() : float

«interface» InterestCheckingAccount
```
Yes! (Reuse via composition and delegation)

```
public class CheckingAccountImpl
    implements CheckingAccount {
    BasicAccountImpl basicAcct = new(...);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // ...
```

CheckingAccountImpl is composed of a BasicAccountImpl
Design alternatives: delegation vs. inheritance

class BasicAccount
    implements Account {
    private long balance = 0;
    public long getBalance() {
        return balance;
    }
    // other methods...
}

class CheckingAccountImpl
    implements CheckingAccount {
    private BasicAccount account;
    public long getBalance() {
        return account.getBalance();
    }
    public void monthlyAdjustment() {
        account.setBalance(
            account.getBalance() - getFee());
    }
    public long getFee() { ... }
}
Design discussion: Delegation vs. inheritance

• Inheritance can improve modeling flexibility
  – protected hooks / helper methods
  – Test with subclasses

• But usually, favor composition/delegation over inheritance
  – Inheritance violates information hiding
  – Delegation supports information hiding

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

• See (excellent) optional reading for Thursday