Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Behavioral subtyping (continued), then Delegation and inheritance

Charlie Garrod       Bogdan Vasilescu
Administrivia

- 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.
Key concepts from last Thursday
Key concepts from last Thursday

• Testing
  – Statement, branch, and path coverage

• Inheritance
  – Implementation inheritance, abstract classes

• Behavioral Subtyping: Liskov Substitution Principle
Selecting test cases

- Write tests based on the specification, for:
  - Representative cases
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker
- Other tests: performance, security, system interactions, ...
A testing example

```java
/**
 * computes the sum of the first len values of the array
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the first len array values
 * @throws NullPointerException if array is null
 * @throws ArrayIndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */

int partialSum(int array[], int len);
```
A testing example

```java
/**
 * computes the sum of the first len values of the array
 * @param array array of integers of at least length len
 * @param len number of elements to sum up
 * @return sum of the first len array values
 * @throws NullPointerException if array is null
 * @throws ArrayIndexOutOfBoundsException if len > array.length
 * @throws IllegalArgumentException if len < 0
 */

int partialSum(int array[], int len);
```

- Test null array
- Test length > array.length
- Test negative length
- Test small arrays of length 0, 1, 2
- Test long array
- Test length == array.length
- Stress test with randomly-generated arrays and lengths
A testing exercise

/**
 * Copies the specified array, truncating or padding with zeros so the copy has the specified length. For all indices that are valid in both the original array and the copy, the two arrays will contain identical values. For any indices that are valid in the copy but not the original, the copy will contain 0. Such indices will exist if and only if the specified length is greater than that of the original array.
 *
 * @param original the array to be copied
 * @param newLength the length of the copy to be returned
 * @return a copy of the original array, truncated or padded with zeros to obtain the specified length
 * @throws NegativeArraySizeException if newLength is negative
 * @throws NullPointerException if original is null
 */
int [] copyOf(int[] original, int newLength);
Today

• Behavioral subtyping (continued)
  – Liskov Substitution Principle
  – The java.lang.Object contracts

• Design for reuse: delegation vs. inheritance
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. Subtypes must have:
  - 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*. 
LSP example: Car is a behavioral subtype of Vehicle

abstract class Vehicle {
    int speed, limit;
    //@ invariant speed < limit;
    abstract void brake();
}

class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant speed < limit;
    //@ invariant fuel >= 0;
    //@ requires fuel > 0
    && !engineOn;
    //@ ensures engineOn;
    void start() { ... }
    void accelerate() { ... }
    void brake() { ... }
}

Subclass fulfills the same invariants (and additional ones)
Overridden method has the same pre and postconditions
LSP example: Hybrid is a behavioral subtype of Car

class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //@ invariant speed < limit;
    //@ 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;
    //@ invariant ... 
    //@ 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
Is this Square a behavioral subtype of Rectangle?

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.)

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);
  }
}

(Yes.)
Is this Square a behavioral subtype of Rectangle?

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?

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

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

   //methods
}

(Yes.)

(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?

```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;
    }
}

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?

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);
    }
}

(Yes.)
Is this Square a behavioral subtype of Rectangle?

```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;
    }

    //@ 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?

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);
    }
}

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

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

(Yes! But the Square is not a square...)}
This Square is not a behavioral subtype of Rectangle

```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;
    }

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

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
The built-in `java.lang.Object` implementations

- Provide identity semantics:
  - `equals(Object o)`: returns true if `o` refers to this object
  - `hashCode()`: returns a near-random int that never changes
  - `toString()`: returns a string consisting of the type and hash code
    - For example: `java.lang.Object@659e0bfa`
The `toString()` specification

- Returns a concise, but informative textual representation
- Advice: Always override `toString()`, e.g.:

```java
final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;
    ...
    @Override public String toString() {
        return String.format("(%03d) %03d-%04d", areaCode, prefix, lineNumber);
    }
}
```

```java
Number jenny = ...;
System.out.println(jenny);
Prints: (707) 867-5309
```
The `equals(Object)` specification

- Must define an equivalence relation:
  - Reflexive: For every object `x`, `x.equals(x)` is always true
  - Symmetric: If `x.equals(y)`, then `y.equals(x)`
  - Transitive: If `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`
- Consistent: Equal objects stay equal, unless mutated
- "Non-null": `x.equals(null)` is always false
An equals(Object) example

```java
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof PhoneNumber))  // Does null check
            return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNumber == lineNumber
            && pn.prefix == prefix
            && pn.areaCode == areaCode;
    }

    ...
}
```
The `hashCode()` specification

- Equal objects must have equal hash codes
  - If you override `equals` you must override `hashCode`
- Unequal objects should usually have different hash codes
  - Take all value fields into account when constructing it
- Hash code must not change unless object is mutated
A hashCode() example

public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        int result = 17;  // Nonzero is good
        result = 31 * result + areaCode;  // Constant must be odd
        result = 31 * result + prefix;  //
        result = 31 * result + lineNumber;  //
        return result;
    }

    ...
}

...
Today

• Behavioral subtyping (continued)
  – Liskov Substitution Principle
  – The java.lang.Object contracts

• Design for reuse: delegation vs. 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: Implementation inheritance for code reuse

```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() { ... }
}
```
Design alternatives: delegation vs. inheritance

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

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

- Inheritance can improve modeling flexibility
- 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
Summary

• Behavioral subtyping: Must conform to specification, even if not enforced by compiler
• java.lang.Object contracts critical for basic Java use
• Design alternatives: Favor delegation over inheritance