Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing classes

Behavioral subtyping

Michael Hilton  Bogdan Vasilescu
Administrivia

• Homework 1 due tonight 11:59 p.m.
  – Everyone must read and sign our collaboration policy
• Reading due Tuesday: Effective Java, Items 17 and 50
• Homework 2 due next Thursday at 11:59 p.m.
Key concepts from Tuesday
Key concepts from Tuesday

• Information hiding: Design for change, design for reuse
  – Encapsulation: Visibility modifiers in Java
  – Interface types vs. class types
• Functional correctness
  – JUnit and friends
Today

- Functional correctness, continued
- Behavioral subtyping
  - Liskov Substitution Principle
  - The java.lang.Object contracts
Unit testing

• Tests for small units: methods, classes, subsystems
  – Smallest testable part of a system
  – Test parts before assembling them
  – Intended to catch local bugs

• Typically written by developers
• Many small, fast-running, independent tests
• Few dependencies on other system parts or environment
JUnit

- A popular, easy-to-use, unit-testing framework for Java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    @Test
    public void testSanityTest() {
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        assertEquals(true, g1.addVertex(s1));
        assertEquals(true, g1.addVertex(s2));
        assertEquals(true, g1.addEdge(s1, s2));
        assertEquals(s2, g1.getNeighbors(s1)[0]);
    }

    @Test
    public void test...
    }

    private int helperMethod...

}
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

/**
   * 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
 * 0.0 to obtain the specified length
 * @throws NegativeArraySizeException if newLength is negative
 * @throws NullPointerException if original is null
 */
int [] copyOf(int[] original, int newLength);
Test organization conventions

• Have a test class FooTest for each public class Foo
• Separate source and test directories
  – FooTest and Foo in the same package
Testable code

• Think about testing when writing code
  – Modularity and testability go hand in hand
• Same test can be used on all implementations of an interface!
• Test-driven development
  – Writing tests before you write the code
  – Tests can expose API weaknesses
Writing testable code

// 700 LOC
public boolean foo() {
    try {
        synchronized () {
            if () {
                } else {
                }
            for () {
                if () {
                    if () {
                        if () {
                            if () {
                                if () {
                                    for () {
                                        }
                                    }
                                } else {
                                    if () {
                                        for () {
                                            }
                                        }
                                    } else {
                                        if () {
                                            for () {
                                                }
                                            }
                                        } else {
                                            if () {
                                                for () {
                                                    }
                                                }
                                            } else {
                                            if () {
                                                for () {
                                                    }
                                                }
                                            } else {
                                            if () {
                                                for () {
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
} else {
    if () {
        for () {
            if () {
                if () {
                    if () {
                        for () {
                            }
                        }
                    }
                }
            } else {
                if () {
                    for () {
                        }
                    }
                } else {
                }
            } else {
            }
        }
    }
} else {
}
Run tests frequently

• Run tests before every commit
  – Do not commit code that fails a test
• If entire test suite becomes too large and slow:
  – Run local package-level tests ("smoke tests“) frequently
  – Run all tests nightly
  – Medium sized projects easily have 1000s of test cases
• Continuous integration servers scale testing
Continuous integration: Travis CI
Continuous integration: Travis CI build history
When should you stop writing tests?
When should you stop writing tests?

• When you run out of money...
• When your homework is due...
• When you can't think of any new test cases...
• The *coverage* of a test suite
  – Trying to test all parts of the implementation
  – Statement coverage: percentage of program statements executed
    • Compare to: method coverage, branch coverage, path coverage
DESIGN FOR REUSE
The promise of reuse:

![Graph showing the comparison of cost with and without reuse. The x-axis represents the number of products, and the y-axis represents cost. The graph shows a linear increase in cost with more products, with the line for 'Without reuse' having a steeper slope than the line for 'With reuse'.]
Reuse: Family of development tools
Reuse: Web browser extensions
Reuse and variation: Flavors of Linux
Today: Class-level reuse with inheritance

• Inheritance
  – Java-specific details for inheritance
• Behavioral subtyping: Liskov's Substitution Principle

• Next week:
  – Delegation
  – Design patterns for improved class-level reuse
• Later in the course:
  – System-level reuse with libraries and frameworks
IMPLEMENTATION INHERITANCE AND ABSTRACT CLASSES
Variation in the real world: types of bank accounts

<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>
Better: Interface inheritance for an account type hierarchy

```java
«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

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

Multiple interface extension

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

If we know we have a CheckingAccount, additional methods are available.
```
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 { }
The power of object-oriented interfaces

- Subtype polymorphism
  - Different kinds of objects can be treated uniformly by client code
  - Each object behaves according to its type
    - e.g., if you add new kind of account, client code does not change:

```java
If today is the last day of the month:
  For each acct in allAccounts:
    acct.monthlyAdjustment();
```
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
```

```
SavingsAccountImpl
...
...
```

```
InterestCheckingAccountImpl
...
...
```
Implementation inheritance for code reuse

What’s wrong with this design?
Implementation inheritance for code reuse

Code duplication
Better: Reuse abstract account code

```java
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 */ }
}
```

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

"interface" CheckingAccount
- getFee() : float

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

CheckingAccountImpl
- monthlyAdjustment()
- getFee() : float
Better: Reuse abstract account code

```java
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 */ }
}
```

- **Abstract class** is missing the implementation of one or more methods.
- **Protected elements** are visible in subclasses.
- **Abstract method** is left to be implemented in a subclass.
- No need to define `getBalance()` – the code is inherited from `AbstractAccount`. 
Interfaces vs Abstract Classes vs Concrete Classes

• An **interface** defines expectations / commitment for clients
  – Java: can declare methods but cannot implement them
  – Methods are **abstract methods**

• An **abstract class** is a convenient hybrid between an interface and a full implementation. Can have:
  – Abstract methods (no body)
  – Concrete methods (w/ body)
  – Data fields
Interfaces vs Abstract Classes vs Concrete Classes

• Unlike a concrete class, an **abstract class** ...
  – *Cannot be instantiated*
  – *Can declare abstract methods*
    • Which *must* be implemented in all *concrete* subclasses

• An abstract class may **implement** an interface
  – But need not define all methods of the interface
  – Implementation of them is left to subclasses

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

«interface» CheckingAccount
getFee() : float

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

CheckingAccountImpl
monthlyAdjustment()
getFee() : float
```
Aside: Inheritance and subtyping

• Inheritance is for **code reuse**
  – Write code once and only once
  – Superclass features implicitly available in subclass

• Subtyping is for **polymorphism**
  – Accessing objects the same way, but getting different behavior
  – Subtype is substitutable for supertype

```java
class A extends B
```

```java
class A implements I
```
Interfaces vs Abstract Classes vs Concrete Classes

• A class can extend 0 or 1 superclass
  – Called single inheritance

• An interface cannot extend a class at all
  – (Because it is not a class)

• A class or interface can implement 0 or more interfaces
  – Closest thing to multiple inheritance
The class hierarchy

• The root is Object (all non-primitives are Objects)
• All classes except Object have one parent class
  – Specified with an extends clause:
    ```java
class Guitar extends Instrument { ... }  
```
  – If extends clause is omitted, defaults to Object
• A class is an instance of all its superclasses
CLASS INVARIENTS
Recall: Data Structure Invariants (cf. 122)

```c
struct list {
    elem data;
    struct list* next;
};

struct queue {
    list front;
    list back;
};

bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
```
Recall: Data Structure Invariants (cf. 122)

- Properties of the Data Structure
- Should always hold before and after method execution
- May be invalidated temporarily during method execution

```c
void enq(queue Q, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{ ... }
```
Class Invariants

• Properties about the fields of an object
• Established by the constructor
• Should always hold before and after execution of public methods
  – May be invalidated temporarily during method execution
Class Invariants

• Properties about the fields of an object
• Established by the constructor
• Should always hold before and after execution of public methods

```java
public class SimpleSet {
    int contents[];
    int size;

    //@ ensures sorted(contents);
    SimpleSet(int capacity) { ... }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean add(int i) { ... }

    //@ requires sorted(contents);
    //@ ensures sorted(contents);
    boolean contains(int i) { ... }
}
```
BEHAVIORAL SUBTYPING

“SHOULD I BE INHERITING FROM THIS TYPE?”
Behavioral subtyping (Liskov Substitution Principle)

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 $T$.

- Applies to specified behavior:
  - Same or stronger invariants
  - Same or weaker preconditions for all methods
  - Same or stronger postconditions for all methods

- 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

This is called the Liskov Substitution Principle.
Barbara Liskov

- First Woman to earn PhD in CS in US
- Turing Award (2008)
- MIT Professor
abstract class Vehicle {
    int speed, limit;

    //@ invariant speed < limit;

    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() { ... }

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

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

```java
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() { ... }
}
```

```java
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
1) An operation is happy if it can be substituted for its super type’s method and sad if it cannot.
2) A smile is wider at the top than at the bottom; a frown is the opposite.
3) Preconditions are at the top of the smile/frown, postconditions are at the bottom of the smile/frown.
4) Ergo: Preconditions can be wider (looser/weaker), postconditions can be narrower (tighter/stronger).
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.)
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?

(Yes.)

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?

```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?

(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;
   }

   //@ 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

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
    //@ 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;
    @Override
    void setWidth(int neww) {
        w=neww;
        h=neww;
    }
}
Today

• Functional correctness, continued
• Behavioral subtyping
  – Liskov Substitution Principle
  – The java.lang.Object contracts
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@659e0bfd
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

```java
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;
    }

    ...
}
```
An Object method exercise

Provide all code needed for a reasonable equals method:

```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;
    }
    ...
}
```
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);
}
```
Summary

• Please complete the course reading assignments
• Test early, test often!
• Subtypes must fulfill behavioral contracts
• Always override `hashCode` if you override `equals`
• Always use `@Override` if you intend to override a method
  – Or let your IDE generate these methods for you...