
Class-level reuse with inheritance
Behavioral subtyping

Charlie Garrod  Bogdan Vasilescu
Administrivia

• Homework 1 due tonight 11:59pm
  – Everyone must read and sign our collaboration policy before we can grade
• Reading assignments:
  – Optional due today (UML class diagrams)
  – Mandatory due Tuesday (immutability, defensive copying)
• Homework 2 released tomorrow
Key concepts from Tuesday
Key concepts from Tuesday

• Design for Change such that
  – Classes are *open for extension* and modification without invasive changes
  – Classes encapsulate details likely to change behind (small) stable interfaces

• Design for Division of Labor such that
  – Internal parts can be *developed* independently
  – Internal details of other classes do not need to be *understood*, contract is sufficient
  – Test classes and their contracts separately (unit testing)
Subtype Polymorphism

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

• Use interfaces to separate expectations from implementation

• 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
interface Animal {
    void makeSound();
}
class Dog implements Animal {
    public void makeSound() { System.out.println("bark!"); }
}
class Cow implements Animal {
    public void makeSound() { mew(); }
    public void mew() {System.out.println("Mew!"); }
}

1 Animal a = new Animal();
2 a.makeSound();

3 Dog d = new Dog();
4 d.makeSound();

5 Animal b = new Cow();
6 b.makeSound();
7 b.mew();
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();
```
JUnit

```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...
}
```
Write testable code

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

Unit testing as design mechanism

* Code with low complexity

* Clear interfaces and specifications

Source: http://thedailywtf.com/Articles/Coding-Like-the-Tour-de-France.aspx
Test organization

• Conventions (not requirements)
• Have a test class FooTest for each public class Foo
• Have a source directory and a test directory
  – Store FooTest and Foo in the same package
  – Tests can access members with default (package) visibility
Selecting test cases: common strategies

• Read specification
• Write tests for
  – Representative case
  – Invalid cases
  – Boundary conditions
• Are there difficult cases? (error guessing)
  – Stress tests?
  – Complex algorithms?
• Think like an attacker
  – The tester’s goal is to find bugs!
• Prevent regressions
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
    • Execute every statement, ideally
    • Compare to: method coverage, branch coverage, path coverage
Code coverage metrics

• Method coverage – coarse
• Branch coverage – fine
• Path coverage – too fine
  – Cost is high, value is low
  – (Related to *cyclomatic complexity*)
Method Coverage

• Trying to execute each method as part of at least one test

```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
public boolean isZero() {
```
Structure of Code Fragment to Test

Flow chart diagram for junit.samples.money.Money.equals
Statement Coverage

• Statement coverage
  – What portion of program statements (nodes) are touched by test cases

```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```
Branch Coverage

- Branch coverage
  - What portion of condition branches are covered by test cases?
  - Multicondition coverage – all boolean combinations of tests are covered

```java
public boolean equals(Object anObject) {
    if (isZero())
        return ((IMoney)anObject).isZero();
    if (anObject instanceof IMoney)
        return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```
Path Coverage

• Path coverage
  – What portion of all possible paths through the program are covered by tests?

```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    if (anObject instanceof Money) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```
### Coverage Report - All Packages

<table>
<thead>
<tr>
<th>Package</th>
<th># Classes</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Packages</td>
<td>55</td>
<td>75%</td>
<td>64%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.ant</td>
<td>11</td>
<td>52%</td>
<td>43%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.check</td>
<td>3</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.coveragedata</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.instrument</td>
<td>10</td>
<td>90%</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.merge</td>
<td>1</td>
<td>86%</td>
<td>88%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting</td>
<td>4</td>
<td>87%</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html</td>
<td>1</td>
<td>87%</td>
<td>77%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html.files</td>
<td>1</td>
<td>87%</td>
<td>62%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.xml</td>
<td>1</td>
<td>100%</td>
<td>95%</td>
<td></td>
</tr>
<tr>
<td>net.sourceforge.cobertura.util</td>
<td>9</td>
<td>60%</td>
<td>69%</td>
<td></td>
</tr>
<tr>
<td>someotherpackage</td>
<td>1</td>
<td>83%</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.
Check your understanding

• Write test cases to achieve 100% line coverage but not 100% branch coverage

```c
void foo(int a, int b) {
    if (a == b) {
        a = a * 2;
    }
    if (a + b > 10) {
        return a - b;
    }
    return a + b;
}
```
Check your understanding

• Write test cases to achieve 100% line coverage and also 100% branch coverage

```c
void foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```
Check your understanding

• Write test cases to achieve 100% line coverage and 100% branch coverage and 100% path coverage

```c
void foo(int a, int b) {
    if (a == b)
        a = a * 2;
    if (a + b > 10)
        return a - b;
    return a + b;
}
```
Coverage metrics: useful but dangerous

• Can give false sense of security
• Examples of what coverage analysis could miss
  – Data values
  – Concurrency issues – race conditions etc.
  – Usability problems
  – Customer requirements issues
Testing, Static Analysis, and Proofs

• Testing
  – Observable properties
  – Verify program for one execution
  – Manual development with automated regression
  – Most practical approach now
  – Does not find all problems (unsound)

• Static Analysis
  – Analysis of all possible executions
  – Specific issues only with conservative approx. and bug patterns
  – Tools available, useful for bug finding
  – Automated, but unsound and/or incomplete

• Proofs (Formal Verification)
  – Any program property
  – Verify program for all executions
  – Manual development with automated proof checkers
  – Practical for small programs, may scale up in the future
  – Sound and complete, but not automatically decidable

What strategy to use in your project?
DESIGN FOR REUSE
The limits of exponentials

Computing capability

Human capacity

capability

time
The promise of reuse:

Cost

Without reuse

With reuse

# Products
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>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>getFee() : float</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>«interface» SavingsAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBalance() : float</td>
</tr>
<tr>
<td>deposit(amount : float)</td>
</tr>
<tr>
<td>withdraw(amount : float) : boolean</td>
</tr>
<tr>
<td>transfer(amount : float, target : Account) : boolean</td>
</tr>
<tr>
<td>getInterestRate() : float</td>
</tr>
</tbody>
</table>
Better: 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
```

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

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

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

Multiple interface extension
Interface inheritance for an account type hierarchy

```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 {
}
```
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?

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

«interface» CheckingAccount
getFee() : float

CheckingAccountImpl
...
...

«interface» SavingsAccount
getInterestRate() : float

SavingsAccountImpl
...
...

«interface» InterestCheckingAccount

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

Code duplication

- «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
  - ...
  - getBalance()
  - ...

- SavingsAccountImpl
  - ...
  - getBalance()
  - ...

- InterestCheckingAccountImpl
  - ...
  - getBalance()
  - ...

- «interface» InterestCheckingAccount
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 */ }
}
```

### Diagram

![Diagram showing the relationship between the AbstractAccount and CheckingAccount interfaces and classes.](image-url)
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 */
    }
}
```

- An abstract class is missing the implementation of one or more methods.
- An abstract method is left to be implemented in a subclass.
- Protected elements are visible in subclasses.
- 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
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
CLASS INVARIANTS
Recall: Data Structure Invariants (cf. 122)

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

```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 $T$.

Barbara Liskov

- Applies to specified behavior:
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions 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.
Behavioral subtyping in a nutshell

• If `Cowboy.draw()` overrides `Circle.draw()` somebody gets hurt!
Car is a behavioral subtype of Vehicle

abstract class Vehicle {
  int speed, limit;
  //@ invariant speed < limit;

  //@ requires speed != 0;
  //@ ensures speed < old(speed)
  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

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