Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Designing Classes

Design for Reuse

Charlie Garrod  Michael Hilton
Administrivia

- HW2 due Thursday Sep 14, 11:59 p.m.
Readings

• Due today:
  – Item 15: Minimize mutability
  – Item 39: Make defensive copies when needed

• Thursday optional reading due:
  – Item 16: Favor composition over inheritance
  – Item 17: Design and document for inheritance or else prohibit it
  – Item 18: Prefer interfaces to abstract classes

• Tuesday required readings due:
  – Chapter 9. Use-Case Model: Drawing System Sequence Diagrams
  – Chapter 10. Domain Model: Visualizing Concepts
Today: Class-level reuse with delegation and inheritance

- Delegation
- Inheritance
  - Java-specific details for inheritance
- Later in the course:
  - System-level reuse with libraries and frameworks
The promise of reuse:

| Products | Cost
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>With reuse</td>
<td></td>
</tr>
<tr>
<td>Without reuse</td>
<td></td>
</tr>
</tbody>
</table>
COMPOSITION AND DELEGATION
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
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
IMPLEMENTATION INHERITANCE AND ABSTRACT CLASSES
Variation in the real world: types of bank accounts

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

public interface SavingsAccount {
    public long getBalance();
    public void deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public double getInterestRate();
}
```
Interface inheritance for an account type hierarchy

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:

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

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

Implementation inheritance allows for code reuse by allowing protected elements to be visible in subclasses and abstract methods to be implemented in a subclass.
Inheritance: a glimpse at the hierarchy

- Examples from Java
  - java.lang.Object
  - Collections library
compact1, compact2, compact3

java.util

**Class LinkedList<E>**

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

**Type Parameters:**
E - the type of elements held in this collection

**All Implemented Interfaces:**
Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```java
public class LinkedList<E>
  extends AbstractSequentialList<E>
  implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.
Aside: A glimpse at the hierarchy

Excerpt from Java Collections API

Collection

- Collection
- List
- Set
- AbstractCollection
- AbstractList
- AbstractSequentialList
- ArrayList
- LinkedList
- HashSet
- AbstractSet
- Vector
- Linkedlist
- Cloneable

- Interface
- Class
- AbstractClass

“implements”

“extends”

interfaces
Aside: Inheritance and Class Hierarchies

• All Java classes are arranged in a hierarchy
  – Object is the superclass of all Java classes

• Inheritance and hierarchical organization capture idea:
  – One thing is a refinement or extension of another

• Benefits of inheritance:
  – Fundamentally enables reuse
  – And modeling flexibility

• A Java aside:
  • Each class can directly extend only one parent class
  • A class can implement multiple interfaces
Inheritance and subtyping

• Inheritance is for polymorphism and 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
class A implements I
```
Typical roles for interfaces and classes

• An interface defines expectations / commitments for clients
• A class fulfills the expectations of an interface
  – An abstract class is a convenient hybrid
  – A subclass specializes a class's implementation
Java details: extended reuse with super

```java
public abstract class AbstractAccount implements Account {
    protected long balance = 0;
    public boolean withdraw(long amount) {
        // withdraws money from account (code not shown)
    }
}
```

```java
public class ExpensiveCheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public boolean withdraw(long amount) {
        balance -= HUGE_ATM_FEE;
        boolean success = super.withdraw(amount);
        if (!success)
            balance += HUGE_ATM_FEE;
        return success;
    }
}
```

Overrides withdraw but also uses the superclass withdraw method
Java details: constructors with this and super

```java
public class CheckingAccountImpl
    extends AbstractAccount implements CheckingAccount {

    private long fee;

    public CheckingAccountImpl(long initialBalance, long fee) {
        super(initialBalance);
        this.fee = fee;
    }

    public CheckingAccountImpl(long initialBalance) {
        this(initialBalance, 500);
    }

    /* other methods... */
```
Java details: final

• A final field: prevents reassignment to the field after initialization
• A final method: prevents overriding the method
• A final class: prevents extending the class
  – e.g., public final class CheckingAccountImpl { …
Note: type-casting in Java

• Sometimes you want a different type than you have
  – e.g., double pi = 3.14;
    int indianaPi = (int) pi;

• Useful if you know you have a more specific subtype:
  – e.g.,
    Account acct = ...;
    CheckingAccount checkingAcct =
    (CheckingAccount) acct;
    long fee = checkingAcct.getFee();
  – Will get a ClassCastException if types are incompatible

• Advice: avoid downcasting types
  – Never(?) downcast within superclass to a subclass
Note: `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.
Note: 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();
      } else if (acct instanceof InterestCheckingAccount) {
          icAccount = (InterestCheckingAccount) acct;
          adj = icAccount.getInterest();
          adj -= icAccount.getFee();
      }
      ...
  }
  ```

Warning: This code is bad.
Avoiding `instanceof`

```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

```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();
    ...
}
```
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
Open/Closed Principle

• Open for extension
  – Extend behavior for new functionality

• Closed for modification
  – Do no modify existing modules
Cautionary tale: left-pad

```javascript
module.exports = leftpad;
function leftpad (str, len, ch) {
    str = String(str);
    var i = -1;
    if (!ch && ch !== 0) ch = ' ';
    len = len - str.length;
    while (++i < len) {
        str = ch + str;
    }
    return str;
}
```
Left-pad incident

• Author un-published left-pad from npm repository over name dispute
• Cascading effect broke many widely used packages such as React (used by Facebook), Bable, and more
• Over 1000 projects affected
Introduction to UML

• 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)
Diagrams

Are often useful when you need to:

Communicate,

Visualize or

Analyze

something, especially something with some structure.
http://www.uh.edu/engines/epi1712.htm
“Democracy is the worst form of government, except for all the others”

-Allegedly Winston Churchill
UML: Unified Modeling Language
UML: Unified Modeling Language
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)

```
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() { ... }
}