Principles of Software Construction: Objects, Design, and Concurrency

Interaction Diagrams and Introduction to Inheritance

Charlie Garrod  Jonathan Aldrich
Administrivia

- Homework 1 due Thursday
- Homework 2 coming soon!
  - Due next Thursday
Key concepts from last Thursday
The design process

1. Object-oriented analysis
   - Understand the problem
   - Identify the key concepts and their relationships
   - Build a (visual) vocabulary
   - Create a domain model (aka conceptual model)

2. Object-oriented design
   - Identify software classes and their relationships with class diagrams
   - Assign responsibilities (attributes, methods)
   - Explore behavior with interaction diagrams
   - Explore design alternatives
   - Create an object model (aka design model and design class diagram) and interaction models

3. Implementation
   - Map designs to code, implementing classes and methods
Documenting a domain model

• **Typical: UML class diagram**
  - Simple classes without methods and essential attributes only
  - Associations, inheritances, etc. as needed
  - Do not include implementation-specific details, e.g., types, method signatures
  - Include notes as needed

• Complement with examples, glossary, etc. as needed

• Formality depends on size of project

• Expect revisions
Today

- **Visualizing dynamic behavior**
  - Interaction diagrams

- **Inheritance and polymorphism**
  - For maximal code re-use
  - Design ideas: hierarchical modeling
  - Inheritance and its alternatives
  - Java details related to inheritance
Our design trajectory

Problem Space
Domain Model

- Real-world concepts
- Requirements, concepts
- Relationships among concepts
- Solving a problem
- Building a vocabulary

Solution Space
Object Model

- System implementation
- Classes, objects
- References among objects and inheritance hierarchies
- Computing a result
- Finding a solution
How do we bridge the representational gap?

• Domain model with a UML class diagram
  ▪ Simple classes without methods and essential attributes only
  ▪ Associations, inheritances, etc. as needed
  ▪ Do not include implementation-specific details, e.g., types, method signatures
  ▪ Include notes as needed

• Complement with examples, glossary, etc. as needed

• Formality depends on size of project

• Expect revisions
One tool: Interaction diagrams

- An *interaction diagram* is a picture that shows, for a single scenario of use, the events that occur across the system’s boundary or between subsystems.

- Clarifies interactions:
  - Between the program and its environment
  - Between major parts of the program

- For this course, you should know:
  - Communication diagrams
  - Sequence diagrams
E.g., Lifecycle of Jonathan's simulation framework

1. Select and create agents

2. Add agents to framework

3. Invoke simulate() on the framework

4. Invoke timestep() on each agent

5. Update agent-specific state in timestep()

6. Invoke logState() on each agent

7. Repeat 4-6 until done

Simulation Framework

- Lodgepole agent
- Infestation agent
- Management agent
- Douglas Fir agent
- Observation agent
- ...
Creating a communication diagram
Communication diagram example, complete

1: prepare()
2*: [for all order lines]: prepare
3: hasStock := check()
4: [hasStock]: remove()
5: needsReorder := needToReorder()
6: [needsReorder]: new
7: [hasStock]: new

: Order Entry Window
: Order
: Macallan line: Order Line
: Macallan stock: Stock Item
: Delivery Item
: Reorder Item
(An annotated communication diagram)
Constructing a sequence diagram
Sequence diagram example, complete
(An annotated sequence diagram)
What kind of interaction diagram is this?

Simulation Framework

1. Select and create agents

Simulation Driver

2. Add agents to framework

3. Invoke simulate() on the framework

4. Invoke timestep() on each agent

5. Update agent-specific state in timestep()

6. Invoke logState() on each agent

7. Repeat 4-6 until done

Lodgepole agent

Infestation agent

Management agent

Douglas Fir agent

Observation agent

...
What kind of interaction diagram is this?

Simulation Framework

1. Select and create agents
2. Add agents to framework
3. Invoke simulate() on the framework
4. Invoke timestep() on each agent
5. Update agent-specific state in timestep()
6. Invoke logState() on each agent
7. Repeat 4-6 until done

Lodgepole agent
Infestation agent
Management agent
Douglas Fir agent
Observation agent
...

(It is a communication diagram.)
Sequence vs. communication diagrams

- Relative advantages and disadvantages?
Today

- **Visualizing dynamic behavior**
  - Interaction diagrams

- **Inheritance and polymorphism**
  - For maximal code re-use
  - Design ideas: hierarchical modeling
  - Inheritance and its alternatives
  - Java details related to inheritance
An introduction to inheritance

- **A dog of an example:**
  - Dog.java
  - AbstractDog.java
  - Chihuahua.java
  - GermanShepherd.java

- **Typical roles:**
  - An interface define expectations / commitment for clients
  - An *abstract class* is a convenient hybrid between an interface and a full implementation
  - Subclass *overrides* a method definition to specialize its implementation
Inheritance: a glimpse at the hierarchy

• Examples from Java
  ▪ java.lang.Object
  ▪ Collections library
Java Collections API (excerpt)

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

interfaces
Benefits of inheritance

• Reuse of code

• Modeling flexibility

• A Java aside:
  • Each class can directly extend only one parent class
  • A class can implement multiple interfaces
Aside: more UML class diagram notation

«interface» Dog
- getName() : String
- getBreed() : String
- bark() : String
- setName(name : String)
- toString() : String

AbstractDog
- name : String
- breed : String
+ getName() : String
+ getBreed() : String
+ bark() : String
+ setName(name : String)
# setBreed(breed : String)
+ toString() : String

GermanShephard
- bark() : String
- play()
Another example: different kinds 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>
A better design: 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
```

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
A better design: An account type hierarchy

```java
public interface CheckingAccount
    extends Account {
    ...
}
```

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

```java
public interface InterestCheckingAccount
    extends CheckingAccount, SavingsAccount {
    ...
}
```

Account are inherited (copied to CheckingAccount)

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

Multiple interface extension
The power of object-oriented interfaces

• Polymorphism
  ▪ Different kinds of objects can be treated uniformly by client code
    • e.g., a list of all accounts
  ▪ Each object behaves according to its type
    • If you add new kind of account, client code does not change
  ▪ Consider this pseudocode:

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

• See the DogWalker example
One implementation: Just use interface inheritance

```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

CheckingAccountImpl
...

«interface» InterestCheckingAccount

SavingsAccountImpl
...

InterestCheckingAccountImpl
...
```
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...
}

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 */ }
}
```

- 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.
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
Challenge: Is inheritance necessary?

- Can we get the same amount of 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
```
Reuse via composition and delegation

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

CheckingAccountImpl is composed of a BasicAccountImpl
Java details: extended re-use with super

```java
public abstract class AbstractAccount implements Account {
    protected float balance = 0.0;
    public boolean withdraw(float amount) {
        // withdraws money from account (code not shown)
    }
}

public class ExpensiveCheckingAccountImpl extends AbstractAccount implements CheckingAccount {
    public boolean withdraw(float 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.
public class CheckingAccountImpl
    extends AbstractAccount implements CheckingAccount {

    private float fee;

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

    public CheckingAccountImpl(float initialBalance) {
        this(initialBalance, 5.00);
    }

    /* other methods... */
}

Invokes another constructor in this same class
Invokes a constructor of the superclass. Must be the first statement of the constructor.
Java details: final

- A final class: prevents extending the class
  - e.g., public final class CheckingAccountImpl {
    ...

- A final method: prevents overriding the method

- A final field: prevents assignment to the field
  - (except to initialize it)

- Why might you want to use final in each of the above cases?
Note: type-casting in Java

- Sometimes you want a different type than you have
  - e.g., `float 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;`  
    `float 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) {
    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();
    }
    ...
}
```

- Advice: avoid `instanceof` if possible
  - Never(?) use `instanceof` in superclass to check type against subclass
Avoiding `instanceof` with the Template Method pattern

```java
public interface Account {
    ...
    public float getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    ...
    public float getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    ...
    public float getMonthlyAdjustment() {
        return getInterest();
    }
}
```
Avoiding `instanceof` with the Template Method pattern

```java
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
float adj = acct.getMonthlyAdjustment();
```
Thursday…

• Behavioral contracts and subtyping rules