Review -- an Exercise

Use the Elevator example as a context, develop a:
• Use case diagram
• Class diagram
• Sequence diagram
• Start Chart diagram
Two software design problems
DNA_ANALYSIS_normal (String DNA_sequence)
DNA_ANALYSIS_detailed (String DNA_sequence)

Two problems

DNAAnalysis (int type, String DNA_sequence)
Complaints from the Biologist

• We have agreed on the set of methods, and my whole application was developed based on this set of methods (that we agreed on). It is time consuming and error prone for us to make extensive changes of our application code.

• What happens if another set of methods are used in the future? Does it mean we need to change our code again?
Interface – an Important Concept in Design Pattern

DNA_ANALYSIS_normal (String DNA_sequence)
DNA_ANALYSIS_detailed (String DNA_sequence)

BioInform_App

DNA_Interface

COTS_Component

DNA_Component

DNAAnalysis (int type, String DNA_sequence)
1. What is Interface?

2. What is the relationship between the interface and the DNA components:
   1. Inheritance?
   2. Delegation?

3. How to solve incompatible interfaces
   1. enforce before design?
   2. “fix” after design?

4. How to support future extensions with new DNA components?
Design pattern

• Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over.

• A design pattern has the four essential elements:
  – Pattern name
  – Problem
  – Solution
  – Consequences
# Design patterns

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Command</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Iterator</td>
</tr>
<tr>
<td>Prototype</td>
<td>Composite</td>
<td>Observer</td>
</tr>
<tr>
<td>Singleton</td>
<td>Façade</td>
<td>Strategy</td>
</tr>
<tr>
<td>etc.</td>
<td>Decorator</td>
<td>Interpreter</td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>Façade</td>
<td>etc.</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

---

Dr. Xiaolin Hu, CS Department, GSU
Understand the fundamentals

- Class
- Interface
- Dynamic binding
Specify object interface

- Object in OO programming
  - An object packages both data and the procedure (typically called methods or operations)
  - An object performs an operation when it receives a request (or message) from a client.
- The set of all signatures defined by an object’s operations is called the interface
- Objects are known only through their interface. An object’s interface says nothing about its implementation – encapsulation
  - How about public variables that can be accessed directly?
- Two objects having completely different implementations can have identical interfaces.
Signature

<table>
<thead>
<tr>
<th>TarifSchedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table zone2price</td>
</tr>
<tr>
<td>Enumeration getZones()</td>
</tr>
<tr>
<td>Price getPrice(Zone)</td>
</tr>
</tbody>
</table>

Hashtable

- numElements:int
- put(key:Object, entry:Object)
- get(key:Object):Object
- remove(key:Object)
- containsKey(key:Object):boolean
- size():int
Dynamic Binding

• Recall what happens in a method call in OO
• Dynamic binding – the run-time association of a request to an object and one of its operations
• Dynamic binding means that issuing a request doesn’t commit you to a particular implementation until run-time.
• This is done to support the cases where the appropriate method can't be determined at compile-time (i.e. statically)
An Example

Public Class classA{
    classC b;
    b = new classC();
    b.getName();
}

Public Class classB {
    Public string getName(){
        Return “b_class”;  
    }
}

Public Class classC extends classB{
    Public string getName(){
        Return “C_class”;  
    }
    Public void newMethod1()  
        {....}
}

Public Class classD extends classB{
    Public string getName(){
        Return “D_class”; 
    }
    Public void newMethod2()  
        {....}
}

Can this be changed to
ClassB b;
?
An Example

```java
Public Class classA {
    classB b;

    getUserInput( );
    If(userInput ==“B”) b = new classB();
    If(userInput ==“C”) b = new classC();
    If(userInput ==“D”) b = new classD();

    b.getName();
}

Public Class classB {
    Public string getName(){
        Return “b_class”; 
    }
}

Public Class classC extends classB{
    Public string getName(){
        Return “C_class”; 
    }
}

Public Class classD extends classB{
    Public string getName(){
        Return “D_class”; 
    }
}
```

Dynamic binding – the run-time association of a request to an object and one of its operations.
1) Static binding in Java occurs during Compile time while Dynamic binding occurs during Runtime.

2) private, final and static methods and variables uses static binding and bonded by compiler while virtual methods are bonded during runtime based upon runtime object.

3) Static binding uses Type(Class in Java) information for binding while Dynamic binding uses Object to resolve binding.

3) Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.
public class StaticBindingTest {
    public static void main(String args[]) {
        Collection c = new HashSet();
        StaticBindingTest et = new StaticBindingTest();
        et.sort(c);
    }

    //overloaded method takes Collection argument
    public Collection sort(Collection c) {
        System.out.println("Inside Collection sort method");
        return c;
    }

    //another overloaded method which takes HashSet argument which is sub class
    public Collection sort(HashSet hs) {
        System.out.println("Inside HashSet sort method");
        return hs;
    }
}

Output: Inside Collection sort method
Dynamic Binding

• With dynamic binding
  – You can write a program that expects an object with a particular interface
  – You can substitute objects that have identical interfaces for each other at runtime
  – This substitutability is known as polymorphism

• Polymorphism
  – In computer science, **polymorphism** is a programming language feature that allows values of different data types to be handled using a uniform interface. The concept of polymorphism applies to both data types (e.g., a list with elements of arbitrary type) and functions.
Object and Class

• Objects are created by **instantiating** a class
• The object is said to be an **instance** of the class
• New class can be defined in terms of existing classes using class inheritance
• Subclass, parent class
• Abstract class, abstract operation, concrete class
• Difference and similarity between abstract class and interface.
  – When to use abstract class and when to use interface.
• Override
Graphic Notation

- Some graphic notation on book P15, 16, 20, 23
- Instantiating
- Inheritance
- Uses (a class keeps a reference to an instance of another class)
- Aggregation
- Abstract class, interface
- Abstract operation
- Pseudo code
An Object’s Class and its Type

- A type refers to the interface
- Subtype (inheriting) supertype
- A class defines an object’s internal state and the implementation of its operations
- In contrast, an object’s type only refers to its interface – the set of its requests to which it can respond.
- An object can have many types – it implements multiple interfaces.
- Objects of different classes can have the same type
Is the following allowed?

```java
InterfaceE myClass = new ClassB();
InterfaceE myClass = new ClassD();
InterfaceE myClass = new ClassA();
InterfaceE myClass = new ClassC();
ClassA myClass = new ClassB();
ClassB myClass = new ClassA();
```
Understand the major concepts

1. Implementation inheritance versus interface inheritance
2. Inheritance versus delegation
Question

- I need to implement a **Stack** class.
- How?
- I already have (implemented and tested) a **List** class.

<table>
<thead>
<tr>
<th>List</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add()</td>
<td>Push()</td>
</tr>
<tr>
<td>Remove()</td>
<td>Pop()</td>
</tr>
<tr>
<td></td>
<td>Top()</td>
</tr>
</tbody>
</table>
Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.

Example: I have a List class, I need a Stack class. How about subclassing the Stack class from the List class and providing three methods, Push() and Pop(), Top()?

How to implement push(), pop(), and top()?
Implementation Inheritance vs Interface Inheritance

• Implementation inheritance
  – Also called class inheritance
  – Goal: Extend an applications’ functionality by reusing functionality in parent class
  – Inherit from an existing class with some or all operations already implemented

• Interface inheritance
  – Also called subtyping
  – Inherit from an abstract class with all operations specified, but not yet implemented
Interface Inheritance

Public Class classA {
    ......
    classB b;

    String userInput = getUserInput();
    If(userInput =="C") b = new classC();
    If(userInput =="D") b = new classD();

    b.getName();
    ......
}

Public Class classB {
    Public virtual string getName();
}

Public Class classC : classB {
    Public string getName(){
        Return "C_class";
    }
    Public void otherMethod1() { }
}

Public Class classD: classB {
    Public string getName(){
        Return "D_class";
    }
    Public void otherMethod2() { }
}

Dynamic binding – the run-time association of a request to an object and one of its operations
Interface Inheritance – in JAVA

```java
Public Class classA {
    ......
    InterfaceB b;
    If(userInput =="C") b = new classC();
    If(userInput =="D") b = new classD();
    b.getName();
    ......
}

Public interface InterfaceB {
    Public string getName();
}

Public Class classC implements InterfaceB {
    Public string getName(){
        Return "C_class";
    }
    Public void otherMethod() { }
}

Public Class classD implements InterfaceB {
    Public string getName(){
        Return "D_class";
    }
    Public void otherMethod2() { }
}

classA still depends on classC and classD (for example, classA cannot be compiled without classC and classD)!

Go back to the BioInf_App example. This means the programmer of BioInf_App needs to have the classes in DNACOMPONENT in order to compile.
```
Discussion

• How to make classA not depend on classC and classD?

• classA expose itself to the other classes (how?). And provide a method, e.g., setClass(B_interface temp) to be called by other classes.

classA{
    ...
    setClass(InterfaceB temp){
        b=temp;
    }
    ...
}
classB{
    ...
    a.setClass(this);
    ...
}
To establish their connections, another class (let’s call it `myApp`) creates `classA`, `classC` and `classD`, and then set up the connections.

Now `classA` has no knowledge of `classC` and `classD` at all. It depends only on `InterfaceB`. 
Another Way

```java
Public Class classA {
    InterfaceB b;
    
    myFactory = new factory();
    b = myFactory.createClass();
    
    b.getName();
    
}

Public interface InterfaceB {
    Public string getName();
}

Public Class classC implements InterfaceB {
    
    Public string getName(){
        Return “C_class”; 
    }
}

Public Class classD implements InterfaceB {
    
    Public string getName(){
        Return “D_class”; 
    }
}

Public Class factory {
    
    Public InterfaceB createClass(){
        getUserInput();
        If(userInput ==“C”) return new classC();
        If(userInput ==“D”) return new classD();
    }
}
```

The factory class separates the client (classA) and service provider (classC and classD). Itself can be developed by a third party.

Dr. Xiaolin Hu, CS Department, GSU
Example of Interface Inheritance

- The DrawingEditor does not need to know the specific objects it uses. It treats all as Shape.
- In the future, new types of shapes are likely supported. When that happens, there is no need to change the DrawingEditor.
Interface inheritance

- All subclasses respond to the requests in the interface of this abstract class, making them all subtype of the abstract class.

- Benefits:
  - Clients remain unaware of the specific types of objects they use. (note that an object may have many types)
  - Clients remain unaware of the class that implement these objects. Clients only know about the abstract class defining the interface.

- Reduce implementation dependencies between subsystems

- *Program to an interface, not an implementation* – don’t declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class.
Question

- I need to implement a **Stack** class.
- How?
- I already have (implemented and tested) a **List** class.
Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.

  - Example: I have a List class, I need a Stack class. How about subclassing the Stack class from the List class and providing three methods, Push() and Pop(), Top()?

How to implement push(), pop(), and top()?

What are the problems with this approach?
Pros and cons

Pros: reuse, straightforward, fast execution

Cons: break the rule of encapsulation

- In the previous example, some inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?
  -- is there a way to avoid this?
  * No. Unless you can modify the parent class, e.g., to change the methods to protected.
- Tight coupling between parent class and child class – a change in the parent class will force the child class to change.
  – what happens if the Add() method is changed?
Problem with implementation inheritance

- How to avoid the following problem?
  - Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?

An example – Hashtable and mySet
Why Delegation? (cont.)

- The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they’re composed.
- Delegation also makes it possible to integrate two behaviors together without combining them together – each of them can be extended independently.
- The window rectangle example on page 20.

Instead of a Window being a Rectangle, it would have a Rectangle.
Inheritance vs. Composition

- The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition.
- Class inheritance – white box reuse, defined statically at compile time.
- Object composition – black box reuse, defined dynamically at run time.
- Composition: new functionality is obtained by assembling or composing objects to get more complex functionality.
  - similar to assembling a car using parts.
- Compare the two approaches by considering building a car that can fly;
- Another example is to build a new department in a company that has both functions of IT and marketing.
Delegation as alternative to Class Inheritance

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance.
- In Delegation two objects are involved in handling a request:
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object.

- Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.
- In what situations does it make better sense to use inheritance instead of delegation?
Delegation instead of Class Inheritance

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.

Going one step further: add an interface that is called by the client!
1. Delegation

Client \rightarrow MyStack

\begin{itemize}
  \item \texttt{Add()}
  \item \texttt{Remove()}
\end{itemize}

\texttt{List}

2. Interface inheritance

Client \rightarrow Stack

\begin{itemize}
  \item \texttt{Push()}
  \item \texttt{Pop()}
  \item \texttt{Top()}
\end{itemize}

Remember this structure!!
To Summarize: Delegation vs Class Inheritance

- **Delegation**
  - **Pro:**
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - **Con:**
    - Inefficiency: Objects are encapsulated.
    - Dynamic, highly parameterized software is harder to understand than more static software.

- **Inheritance**
  - **Pro:**
    - Defined statically at compile time and straightforward to use
    - Supported by many programming languages
    - Easy to implement if simply adding new functionality.
  - **Con:**
    - Inheritance exposes a subclass to the details of its parent class
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)
    - “Inheritance breaks encapsulation”
Relating Run-Time and Compile-Time Structures

• An object-oriented program’s runtime structure often bears little resemblance to its code structure

• Aggregation and acquaintance – they are implemented in the same way in most programming languages

• Aggregation
  – implies that one object owns or is responsible for another object
  – One object *having* or being *part of* another object
  – Implies that an aggregate object and its owner have identical lifetime

• Acquaintance
  – implies that one object merely knows of another object
  – *Association* or the *using* relationship
  – Acquainted objects may request operations of each other, but they aren’t responsible for each other
  – Is a weaker relationship than aggression and suggests much looser coupling between objects

• Example: car and engine, tires, gasoline; car and road, driver
Some rules

Program to an interface, not an implementation.

Favor object composition over class inheritance

Interface inheritance + delegation → design pattern

Loose coupling → reuse
Designing for change

• The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements in designing your systems so that they can evolve accordingly.
• Here are some common causes of redesign
Designing for change (cont.)

- Creating an object by specifying a class explicitly – commits you to a particular implementation instead of a particular interface.
- Dependence on hardware and software platform → middleware
- Dependent on object representation or implementation
- Algorithmic dependencies – algorithms that are likely to change should be isolated.
- Tight coupling – the system becomes a dense mass that’s hard to learn, port, and maintain.
- Extending functionality by subclassing
- Inability to alter classes conveniently.
Some Other Concepts

• Application programs
  – Design priorities are: internal reuse, maintainability, and extension

• Toolkit
  – A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality.
  – For example: the C++ I/O stream library, or an image processing toolkit
  – Toolkit don’t impose a particular design on your application
  – Toolkit emphasize code reuse. They are the OO equivalent of subroutine libraries.

• Framework
  – A framework is a set of cooperating classes that make up a reusable design for a specific class of software.
  – For example: a framework for building financial modeling application
  – The framework may impose a particular design on your application. It dictates the architecture of your application.
  – Frameworks emphasize design reuse over code reuse.
Framework

• How is a framework used?
  – You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework.

• Toolkit and framework – reuse leads to an inversion of control
  – When you use a toolkit, you write the main body of the application and call the code you want to reuse
  – When you use a framework, you reuse the main body and write the code it calls

• With framework – faster development, similar structure, easier to maintain. But you lose some creative freedom.

• Design of framework is the hardest. A framework designer gambles that one architecture will work for all applications in the domain.
  – It is imperative to design the framework to be as flexible and extensible as possible.
A Framework Example - JNI

- The Java Native Interface (JNI) is a programming framework that allows Java code running in the Java virtual machine (JVM) to call and be called by native applications and libraries written in other languages, such as C, C++.
- The JNI framework lets a native method utilize Java objects in the same way that Java code uses these objects.
- In JNI, native functions are implemented in a separate .c or .cpp files. When the JVM invokes the function, it passes a JNIEnv pointer, a jobject pointer, and any Java arguments declared by the Java method. A JNI function may look like this:
  ```c
  JNIEXPORT void JNICALL Java_ClassName_MethodName (JNIEnv *env, jobject obj) {
      //Implement Native Method Here
  }
  ```

Question: how to design/implement the JNI framework, such as the JNIEnv?
Framework and Design Pattern

• Mature frameworks usually incorporate several design patterns

• Frameworks and design patterns are different
  – Design patterns are more abstract than frameworks. Frameworks can be embodied in code, but only examples of patterns can be embodied in code.
  – Design patterns are smaller architectural elements than frameworks
  – Design patterns are less specialized than frameworks.

• Large object-oriented applications will end up consisting of layers of frameworks that cooperate with each other.
Chapter 2: A case study

Designing a document editor

A sample document in Word

Let’s carry on the design together (draw class diagram) – hopefully you will get the idea why design pattern is useful!
Design problems

Document Structure – the choice of internal representation for the document

- We should treat text and graphics uniformly
- We shouldn’t have to distinguish between single elements and groups of elements in the internal representation.
- However, there is a need to analyze the text for such things as spelling errors (no need for analyzing an image).
Design problems

Formatting – encapsulating the formatting algorithms and interact with the document’s internal representation

- It must break text into lines, lines into columns, and so on, taking into account the user’s desires such as margin width, indentation, single or double space, and many other formatting constrains.
- People have come up with a variety of formatting algorithms with different strengths and weaknesses, e.g., trade-off between formatting quality and formatting speed.
- We want to add a new kind of Glyph subclass without regard to the formatting algorithm. Conversely, adding a new formatting algorithm shouldn’t require modifying exiting glyphs.
Design problems

Embellishing the user interface – transparent enclosure

• Add a border around the text editing area
  – Inheritance: BorderedComposition?

• Add a scrollbar around the text editing area
  – Inheritance: ScrollbarComposition?

• Both border and scrollbar around the text editing area
  – Inheritance: BorderedScrollbarComposition?

• We need transparent enclosure
Design problems

Supporting multiple look-and-feel standards
– Abstract object creation
  • We want to be able to change the look-and-feel style at runtime. And we want to replace a entire widget, such as button, toolbar, header, set easily.
  • Problem with hard coded implementation
    – Does not support runtime change
    – Have to track down and change every constructor if a new style is added.
Design problems

User Operations

• We don’t want to associate a particular user operation with a particular user interface, such as a particular menu item.
• We want to support undo and redo of most but not all its functionality.
• Solution: treat each operation as an object!
Group Formation and Advanced Topic Review