An Introduction to Object Orientation and CBD

References

• http://www.javasoft.com/docs/index.html
  – Provides on-line tutorials and specifications of java language and frameworks

Objectives

• Provide a Working Knowledge of OOD
  – Definitions of OO Concepts and Terms
  – Examples in Java
  – Specifications Examples in UML
  – Discussion and Homework for Learning
• Establish a basis for understanding CBD and reuse issues.

Overview

• Inheritance
  – Descendent and Ancestors
  – Invariant Inheritance
  – Inheritance and Creation
• Polymorphism
  – Attachment, Data Structures
Overview

• Typing for Inheritance
  – Limits on Polymorphism
  – Instances with Polymorphism
  – Static and Dynamic Type
• Dynamic Binding
• Deferred Features and Classes

OOD/CBD Desired Results

• Extensibility
  – The ease of adapting software products to changes of specification
  – Simplicity and Decentralization
• Reusability
  – The ability of software elements to serve for the construction of different applications

OOD/CBD Desired Results

• Compatibility
  – The ease of combining software elements with other elements
  – CORBA, OLE-COM (ActiveX)
  – SQL Call Level Interface, ODBC (Microsoft), JDBC (Java)
  – Sockets, Remote Method Invocation
  – OOD Specification of Interfaces to Entities

OOD/CBD Desired Results

• Efficiency
  – The ability of a software system to place as few demands as possible on hardware resources, such as processor time, memory space, and communication bandwidth
  – Dr. Abstract and Mr. Microsecond
  – Hard Deadlines, Response Time, throughput budget, Latency Times, ...
OOD/CBD Desired Results

- Portability
  - The ease of transferring software products to various hardware and software environments
- Ease of Use
  - Ease with which people of various backgrounds and qualifications can learn to apply software products to solve their problems

OOD/CBD Desired Results

- Timeliness
  - Estimation and Incremental Development
- Low Cost
  - Competitive Market Pressures 30-60% Reductions
  - Minimal Discrepancies at Integration Stops Re-Work
  - Reuse Allows Less Code and Maintenance

Reuse and Extension of Systems

Requires Modularity

- Modularity provides reuse and extension when modules are self contained and organized in a stable architecture
- The software construction method must produce a system made of autonomous elements connected by a coherent, simple structure

Reuse and Extension of Systems--Criteria

- Modular Composability - Software elements may be freely combined with each other to produce a new system
- Modular Understandability - A human can understand each module without having to know many others
Reuse and Extension of Systems Criteria

- Modular Continuity - In the software architecture, a small change in a problem specification will trigger a change in just one module, or a small number of modules.
- Modular Protection - An architecture in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module.

Modular Software Construction Rules

- Direct Mapping - The modular structure devised in the process of building the software remains compatible with the modular structure devised in the modeling of the problem domain.

Modular Software Construction Rules

- Few Interfaces - Every Module Should Communicate with As Few As Possible.
- Small Interfaces - If two modules communicate, they should exchange as little information as possible (from continuity and protection).

Modular Software Construction Rules

- Explicit Interfaces - Whenever two modules A and B communicate, this must be obvious from the text of A or B or both:
  - Need information to Compose or Decompose
  - Analyze the continuity when modules change
  - Understanding
Modular Software Construction

Rules

• Information Hiding - The designer must select a subset of the module’s properties as the official information about the module, to be made available to authors of client modules
  – Public Part and Secret Part
  – Clients cannot be written to depend on the secret part of a module

Open-Closed Principle

• Open-Closed principle - Modules should be both open and closed
  – Open means it is available for extension such as adding fields, expanding operations, etc
  – Closed means it is available for use by other modules with a well defined interface that is stable
  – Closed means it is compiled and released
Java Classes A and A'prime

class A {
    public void service1 {
        /*method code*/
    }
    public void service2 {
        /*method code*/
    }
}
class A'prime extends A {
    public void newserv3 {
        /*method code*/
    }
    public void newserv4 {
        /*method code*/
    }
}

Reuse of Designs

- Reducing the gap between design and implementation (understandable OO code) leads to reusing implementations
- Components are mostly reusable OO designs manifested in code
- OO, Design Patterns, Frameworks, and Components are Closely related (more later)

A Stack Class Specification

(JavaDoc Description)

Class java.util.Stack
java.lang.Object
    +----java.util.Vector
        +----java.util.Stack

public class Stack
    extends Vector
The Stack class represents a last-in-first-out (LIFO) stack of objects.

Stack Class JavaDoc Excerpt

Stack()
    Constructs an empty stack
empty()
    Tests if this stack is empty.
peek()
    Looks at the object at the top of this stack without removing it from the stack.
pop()
    Removes the object at the top of this stack and returns that object as the value of this function.
push(Object)
    Pushes an item onto the top of this stack.
search(Object)
    Returns where an object is on this stack.
Some OO Definitions

- An **object** is a run-time instance of some class
- A **reference** is a run-time value which is either void or attached. If attached, the reference identifies a single object
- The **direct dependents** of an object are the objects attached to its reference fields
- The **dependents** of an object are the object itself and (recursively) the dependents of its direct dependents
- **Persistence** is the conservation of objects between computing sessions. **Persistence closure** conserves all dependents of the object

Inheritance Terminology

- A **descendant** of a class \( C \) is any class that inherits directly or indirectly from \( C \), including \( C \) itself. (Formally: either \( C \) or, recursively, a descendant of an heir of \( C \).)
- A **proper descendant** of \( C \) is a descendant other than \( C \) itself.
- An **ancestor** of \( C \) is a class \( A \) such that \( C \) is a descendant of \( A \).
- A **proper ancestor** of \( C \) is a class \( A \) such that \( C \) is a proper descendant of \( A \).

```java
public class Stack extends Vector {
    public Object push(Object item) {
        addElement(item);
        return item;
    }
    public synchronized Object pop() {
        Object obj;
        int len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
    }
    ...
}
```
Invariant Inheritance Rule

The invariant property of a class is the boolean and of the assertions appearing in its invariant clause and of the invariant properties of its parents if any.

Creation Inheritance rule

An inherited feature’s creation status in the parent class (that is to say, whether or not it is a creation procedure) has no bearing on its creation status in the heir.

Polymorphism

- **Polymorphism** means the ability to take several forms.
- In object-oriented development what may take several forms is a variable entity or data structure element
- An entity has the ability, at run time, to become attached to objects of different types, all controlled by a static declaration.
Polymorphic Attachment

```java
class PolyMatcher is {
    Polygon p;
    public line longestSide(Polygon p) {
        line side = p.getSide[1];
        line side2;
        scount = p.getSideCount;
        // calculate longest side of the polygon
        for (int i=2, i<=scount, i++) {
            side2 = p.getSide[i];
            if (side2 > side) side = side2;
        }
        return side;
    }
}
```

Typing for Inheritance

**Feature Call rule**

In a feature call $x.f$, where the type of $x$ is based on a class $C$, feature $f$ must be defined in one of the ancestors of $C$.

This rule is static - checked on text not run-time dependent

Polygon p = new polygon (plist);
p.side1; // illegal operation found by compiler

Typing for Inheritance

A **direct instance** of a class $C$ is an object produced according to the exact definition of $C$, through a creation instruction $C x = \text{new } C$.

An **instance** of $C$ is a direct instance of a descendant of $C$.

**Static-dynamic type consistency**

An entity declared of a type $T$ may at run time only become attached to instances of $T$.

The Many Faces of Inheritance

(Meyer, B. IEEE Computer, May 1996)
**Classification of Inheritance**

- **Model Inheritance**
  - “is-a” relations between abstractions in the model
- **Software Inheritance**
  - relations within the software with no obvious counterpart in the model
- **Variation Inheritance**
  - describes the differences between classes

**Subtype Inheritance**

Subtype inheritance applies if $A$ and $B$ represent certain sets $A'$ and $B'$ of external objects such that $B'$ is a subset of $A'$ and the set modeled by any other subtype heir of $A$ is disjoint from $B$. $A$ must be deferred.

- **Examples**
  - File is an heir of Device in an operating system
  - Taxonomies in Zoology or botany, e.g. mammal is an heir of vertebrate

**Restriction Inheritance**

Restriction inheritance applies if the instances of $B$ are those instances of $A$ that satisfy a certain constraint, expressed if possible as part of the invariant of $B$ and not included in the invariant of $A$. Any feature introduced by $B$ should be a logical consequence of the added constraint. $A$ and $B$ should be both deferred or both effective.

- **Examples**
  - Square is a descendant of Rectangle where the extra constraint is $side1 = side2$ (included in the invariant of SQUARE)

**Extension Inheritance**

Extension inheritance applies when $B$ introduces features not present in $A$ and not applicable to direct instances of $A$. Class $A$ must be effective.

Examples

- Class Chapter inherits from class Document
- MovingPoint inherits from Point
Variation Inheritance

- Variation inheritance applies if B redefines some features of A; A and B are either both deferred or both effective, and B must not introduce any features except for the direct needs of the redefined features. There are two cases:
  - Functional variation inheritance: some of the redefinitions affect feature bodies, rather than just their signatures.
  - Type variation inheritance: all redefinitions are signature redefinitions.
- public void segment perpendicular(){…} changed to
- public void dottedSegment perpendicular() {…}

Uneffecting Inheritance

Uneffecting inheritance applies if B redefines some of the effective features of A into deferred (i.e. abstract) features.
- You may find a reusable class that is too concrete for your purposes, although the abstraction it describes serves your needs.

Reification Inheritance

Reification inheritance applies if A represents a general kind of data structure, and B represents a partial or complete choice of implementation for data structures of that kind. A is deferred; B may still be deferred, leaving room for further reification through its own heirs, or it may be effective.
- Dispenser is parent of Queue and of Stack
- Table has heirs SequentialTable and HashTable
- Final Reification of SequentialTable leads to ArrayedTable, LinkedTable, and FileTable

Structure Inheritance

Structure inheritance applies if A, a deferred class, represents a general structural property and B, which may be deferred or effective, represents a certain type of objects possessing that property.
- Usually A represents a mathematical property that a certain set of objects may possess;
  - For example, A may be the class Comparable
  - A has operations such as infix "<" and infix ">",
  - A class like STRING will inherit from Comparable.
Implementation Inheritance

Implementation inheritance applies if $B$ obtains from $A$ a set of features (other than constant attributes and once functions) necessary to the implementation of the abstraction associated with $B$. Both $A$ and $B$ must be effective.

- Class Stack inherits operations from class Array and uses the operations of Array to implement the stack operations.

Facility Inheritance

Facility inheritance applies if $A$ exists solely for the purpose of providing a set of logically related features for the benefit of heirs such as $B$. Two common variants are:

- Constant inheritance in which the features of $A$ are all constants or once functions describing shared objects.
- Machine inheritance in which the features of $A$ are routines, which may be viewed as operations on an abstract machine.

Substitution Principle

References


Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

- How to maintain the open-closed principle
Substitution Principle
B. Liskov
If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \) then \( S \) is a subtype of \( T \).

Lesson Learned
- When considering whether a particular design is appropriate or not, one must not view the solution in isolation.
- One must view it in terms of the reasonable assumptions that will be made by the user of that design.

Behavior as Is-A
- The substitution principle makes it clear that the Is-A relationship pertains to behavior
- All descendants must conform to the behavior that clients expect of the parent class
Substitution and Design by Contract

• Post of Rectangle.setWidth(double w)
  itsWidth == w && itsHeight == oldItsHeight
  Note: oldItsHeight is the value before setWidth is executed
• Post of Square.setWidth(double w)
  itsWidth == w && itsHeight == w
• Post of Square.setWidth is noncomparable to the post of Rectangle.setWidth since it does not conform to itsHeight == oldItsHeight

Substitutability

• For a descendant class B of a class A, B is substitutable as a type for A when for each operation in B, fb, that overrides an operation in A, fa, we have
  – if different, the precondition of fb is weaker than the precondition of fa.
    • Mathematically: Pre(fa) => Pre(fb)
  – if different, the postcondition of fb is stronger than the postcondition of fa (in the context of the precondition of fa)
    • Mathematically: (Pre(fa) AND Post(fb)) => Post(fa)

TextModel Example

```java
interface TextModel {
  int max (); //maximum length this text can have
  int length (); //current length
  char read (int pos);  //character at position pos
  void write (int pos, char ch); //insert character ch at position pos
  // 
  // pre len := this.length(); (all i: 0 <= i < len: txt[i] := this.read(i));
  // len < this.max() and 0 <= pos <= len
  // post this.length() = len + 1
  // and (all i: 0 <= i < pos: this.read(i) = txt[i])
  // and this.read(pos) = ch
  // and (all i:pos < i < this.length(): this.read(i) = txt[i - 1])
  // ]... ]
```
GreatTextModel Example

class GreatTextModel implements TextModel {
    ...
    
    void write(int pos, char ch); // insert character ch at position pos
    // [ len:int; txt: array of char •
    // pre len := this.length(); (all i: 0 <= i < len: txt[i] := this.read(i));
    // len < this.max() and 0 <= pos <= this.max();
    // post this.length() = max(len, pos) + 1
    // and (all i: 0 <= i < pos: this.read(i) = txt[i]);
    // and this.read(pos) = ch
    // and (all i: pos < i < len: this.read(i) = txt[i - 1]);
    // and (all i: len < i < pos: this.read(i) = " ")
    //]
    }

Weaken the Precondition

des.length() < des.max() and 0 <= pos <= des.length
=> des.length() < des.max() and 0 <= pos < des.max()

Types

- The types of input and in-out parameters form a part of an operation’s preconditions
- The types of output and in-out parameters plus the type of returned values form a part of an operation’s postconditions
- Types do not make up the entire contract other pre and post conditions are not part of the types
Covariance

- A provider may establish more than is specified by a contract
- Hence, a subtype interface can replace the types of output parameters by more specific subtypes (i.e. descendants of the original type)
- As the types of output parameters and return values can thus be varied in the same direction as the types of the containing interfaces, this is called *covariance*

Contravariance

- A provider may expect less than is specified in the precondition
- Hence, a subtype interface could replace the types of input parameters by something more general (i.e. from original type to one of its ancestors)
- As the types of the input parameters can be varied in the opposite direction of the type of the containing interfaces, this is called *contravariance*

Function g could be substituted for function f in terms of domain and range

Covariance and Contravariance