Generalization

• Creation of super and subtype entities
• May be program and design classes, concepts, use cases, etc.
• For programs, generalization corresponds to class inheritance
• Subtypes can modify, add to, and access properties of supertypes
Kinds of Generalization

• specialization
• specification
• construction
• extension
• limitation
• combination
Specialization

• Re-defines defined parent properties and methods. Also called refinement.
• E.g. User-defined class constructor always calls the supertype constructor, either explicitly or automatically
• E.g. Suppose we have a stack class where a message is returned when an item is pushed on the stack
  • success if pushed, or error if stack is full
  • change this so stack-is-full warning is returned when stack becomes full, i.e. last item is pushed
Specification

• Used to specify behavior, but not to define it
• Subtype defines the behavior
• E.g. ActionListener interface in Java GUI. Specifies behavior (methods) for listener classes which implement ActionListener
Construction

• For the purposes of re-use of methods and data structures
• Defining new methods and data structures in terms of inherited ones
• Child may have no logical relationship with parent(s)
• E.g. Use of Vector to define a stack class
Extension

• Adding new properties and methods
• No modification of parent properties
• Similar to construction but has the “is a property” which construction may not have
• E.g. adding button declarations to the Dialog class in a new extends class
Limitation

• Override some methods with blank/null methods
• Limits access to certain behaviors
• May be used with construction to block methods that have no meaning in the subtype
• E.g. WindowAdpator class which is a limitation of WindowListener interface
Combination

• Uses multiple inheritance for two or more applications of generalization

• DS Java example

  MessageDialog extends (i) Dialog implements (ii) ActionListener
  (i) extension: add buttons, panels
  (ii) specification: define actionPerformed method()
Generalization Correctness Rules
(From earlier lecture)

• *is-a*  An instance of a subtype is also an instance of the supertype. e.g. an administrator is a DS user

• *Substitutability*  Suppose B is a subtype of A. It should be possible to substitute an instance of B any place that requires something of type A.

• *100% rule*  All of the supertype’s definition should also apply to the subtype (i.e. its attributes, associations)
Correct Generalization Uses

• Specialization – is-a
• Specification – is-a, substitutability, 100%
• Extension – is-a, substitutability, 100%
• Combination (sometimes)
Polymorphism

- Literally, an entity that can take different “forms”
- Programs
  - A variable that can have values of different types
  - A method that can take arguments of different types
Polymorphic Variables

• Dynamically typed languages (e.g. Smalltalk)
  – any variable may hold items of any type, provided expected operations are defined

• Statically typed languages (e.g. Java)
  – variable takes on values of differently typed subtypes of variable’s class
Polymorphic Methods

- Method can have arguments of different types
- Any type
- Subtypes of declared supertypes
Polymorphic Design Pattern and Generalization

- Based on the specification variation of the uses of generalization
- Subtypes give different method definitions for the more general method specification
- Eg. In DS can be in member or admin state, for which there are different behaviors. Design a state class with abstract methods whose substate behavior is defined in the subtype classes
Why Polymorphic Design?

• Subtypes encapsulate behavior alternatives for specification and documentation
• Promotes cohesion and reuse
• Follows the basic expert pattern
  – creates an expert which will be given the responsibility for the behavior for which it is the expert
Functional Versus Polymorphic Alternative Behavior

- **Functional:**
  conditional statement calls the appropriate function/subroutine or executes the substatement that exhibits the appropriate behavior.

- **Polymorphic:**
  send message to an object of a single type (supertype) which will result in the appropriate behavior. Object will be an instance of the associated supertype.
Overloading – Early Uses

• Use of the same name for different operations/methods
• Idea inherited from mathematics
• e.g. overloading of * operation
  x * y for integers x and y
  x * y for complex numbers x and y
  x * y for p × n and n × q matrices x and y
Overloading and Method Names

• Different methods have the same name
  – methods in the same class
  – methods in different classes related by subtyping
  – methods in different classes not related by subtyping
    • logically and non-logically related

• Need a way to disambiguate an overloaded method reference
Overload Disambiguation

• Class disambiguation
  – method reference is disambiguated by the class of the object whose method is being invoked
    • subtypes of supertype
    • unrelated classes
      – e.g. isEmpty() is understood by Vector and Hashtable classes

• Parameter disambiguation
  – methods in the same class with the same name must have different parameter lists
  – e.g. multiple constructors for a class
Overloading and Programming Languages

• Requires that variables be typed, so that compiler can find and incorporate the correct code
Polymorphism and Overloading

• Polymorphism => overloading
• Suppose the superclass has a method m() that is defined in the subtypes
• m() overloaded in the subtypes
• Uses class disambiguation
Non-Polymorphic Overloading

• Methods in unrelated classes may have the same name
• E.g. Many classes have an isValueOf() operator that will behave slightly differently for each class
• The classes are not subtypes of a more general type
Coercion and Deferment

- Deferred method definition
  - abstract method in superclass
  - another kind of “downward” overloading
- Coercion
  - types of parameters are coerced to a standard type, not really overloading
  - e.g. x*y is always floatingpoint for both integer and real numbers
Overloading Examples

• DS: execute() methods for logOn, DateRequest classes

• Multiple constructors for the same class
States and Alternative Behavior

• Context: object can be in several states, and it behaves differently depending on which state it is in

• E.g. DS
  – states reflect type of current user (member, administrator, unauthorized)
  – response to request, such as addMember() will be different in each state
State Classes

• Object \( x \) from class \( A \) can be in different states

• Create a state class \( S \)
  – \( S \) has abstract methods that mirror the state sensitive methods in \( A \)
  – define subtypes of \( S \) for each of the states of \( A \), that have definitions that describe the different state sensitive behavior
State Pattern

• Suppose A: x is a state sensitive object
• Define a corresponding state class S for A
• Include a state variable S: s in A whose value will be an instance of the appropriate subtype of S
• When a method x.m() in x is invoked, it executes s.m()
DS - State Class for DL

- Domain Logic subsystem has an interface class called DomainLogic.
- DomainLogic contains an inner class that defines a state class DomainLogicState.
- DomainLogicState has subtypes for each type of user.
- DomainLogic has a class variable called currentState that is set to an instance of a subtype of DomainLogicState.
DS – Setting the State Variable

• GUI calls logOn(name) in the DomainLogic instance dL
• dL.logOn(name) creates a LogOn instance which is then executed
• logOn.execute() determines user type based on
  – name (administrator has a special name) and
  – whether is a member with this name in member data base
• execute() sets the userType and the currentState variables for the DomainLogic instance
public int execute()
{
    if (userName.equals(administrator))
    {
        userType = Constants.ADMIN;
        currentState = new DomainLogicAdminState();
        return(Constants.ADMIN);
    }

    b = dataBase.isMember(userName);
    if (dataBase.isMember(userName))
    {
        userType = Constants.MEMBER;
        currentState = new DomainLogicMemberState();
        return Constants.MEMBER;
    }
    else
    {
        userType = Constants.UNAUTH;
        currentState = new DomainLogicUnauthState();
        return (Constants.UNAUTH);
    }
}
DS - State Dependent Behavior

• Suppose dL gets a message (i.e. one of its methods is performed), such as deleteMember(name)

• dL calls the corresponding state method
Abstract class and methods:

DomainLogic
- state
+ getADate()

DomainLogicState
+ getADate()

MemberState
+ getADate()

AdminState
+ getADate()

UnauthUserState
+ getADate()

getADate(props) → getADate(props, dL) →
Object1 dL: DomainLogic : DomainLogicState

Method of current substate subtype object is invoked:
public boolean deleteMember(String name) {
    return currentState.deleteMember(this, name);
}

Note: a parameter has been added to the method for the subtype in order to pass along the identity of the object that can be in different states
DS – Code for MemberState Subtype

• Members are not allowed to delete members

```java
public boolean deleteMember(DomainLogic dl, String name)
{
    return false;
}
```
DS – Code for AdminState Subtype

• Assume DataBase is built with a delete capability so DL just passes it through

```java
public boolean deleteMember(DomainLogic dl, String name)
{
    return dl.dataBase.deleteMember(name);
}
```
Alternative Object Behaviors

- Mechanisms: Polymorphic pattern, states, decorator wrapper
- Issues:
  i) how are the alternatives defined?
  ii) set of peer alternatives, or alternative to a given existing behavior?
  iii) when is an alternative behavior established for an object?
  iv) does the new behavior require the creation of a new object?
<table>
<thead>
<tr>
<th>Polymorphic</th>
<th>State Based</th>
<th>Decorator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative subtypes for</td>
<td>Alternative subtypes for associated state</td>
<td>New class definition using object of interest</td>
</tr>
<tr>
<td>object of interest</td>
<td>variable</td>
<td>in the definition</td>
</tr>
<tr>
<td>Related set of alternatives</td>
<td>Related set of alternatives</td>
<td>Single new kind</td>
</tr>
<tr>
<td>Set at object creation time</td>
<td>Changed during run time</td>
<td>Changed during run time</td>
</tr>
<tr>
<td>New object with selected behavior</td>
<td>Same object with new behavior</td>
<td>New object with altered behavior</td>
</tr>
</tbody>
</table>

Copyright W. Howden