Lecture 14: Advanced Analysis (Part 2)

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2002

Credit where Credit is Due

- Some material presented in this lecture is taken from section 5 of Maciaszek's "Requirements Analysis and System Design". © Addison Wesley, 2000

Goals for this Lecture

- Continue to cover the material presented in Section 5 of the textbook
  - Introduce
    - Advanced Generalization and Inheritance
    - Advanced Aggregation and Delegation

Advanced Generalization and Inheritance

- Generalization and Inheritance are related but not the same
  - Generalization is a semantic relationship between classes
    - It states that the interface of the subclass must include all (public and protected) properties of the superclass
  - Inheritance is the mechanism by which more specific elements incorporate structure and behavior defined by more general elements
    - (more on this later)

Generalization and Substitutability

- Semantically, generalization
  - introduces new classes into a model
  - categorizes them into generic and more specific classes
  - and establishes superclass-subclass relationships
- The benefits of generalization arise from the substitutability principle
  - a subclass object can be used in place of a superclass object in any part of the code where the superclass object is accessed
  - This allows for instance aggregation and associations to be defined on a superclass yet apply to its subclasses
- Unfortunately, inheritance can defeat substitutability benefits

Encapsulation vs. Inheritance

- Encapsulation demands that an object's attributes are only accessible through the operations in an object's interface
  - If enforced, encapsulation leads to a high level of data independence; data structures can change within a class and not impact other classes (in general, non-functional changes can invalidate this statement)
- Inheritance, unfortunately, compromises encapsulation, since subclasses can directly access protected attributes without using operations

Interface Inheritance

- One type of inheritance involves just the interface of a class; used to support substitutability
  - A subclass inherits attribute types and operation signatures from its superclass or from an explicitly defined interface class (such as the interface mechanism in Java)
    - A subclass is said to "support" or "implement" the interface; the subclass does not inherit the implementations of the operations from a superclass
  - Abstract classes are similar to interfaces, except that an abstract class can define implementations for some operations that are inherited by its subclasses

More on Interfaces

- An interface is a collection of operations (not data) that specifies a particular service of a class or a component
  - For instance, lists, queues, stacks, and trees typically provide an Iterator interface that allows other classes to cycle through their elements

UML Notation

- The most simple notation for an interface is a labeled circle

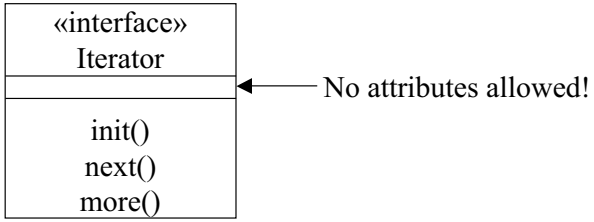
Iterator

Interface names can be grouped using packages

Java::Collection::Iterator

UML Notation

- However, a full class diagram can be used to specify the particular operations associated with an interface

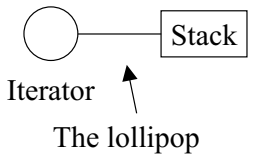


How interfaces are used

- You cannot instantiate an instance of an interface, instead other classes (and thus their objects) choose to implement certain interfaces
  - An interface can act as a type, so you can declare variables that have, for instance, the Iterator type
  - This allows you to point at a class who implements the Iterator interface without knowing (or caring) about what its actual type is

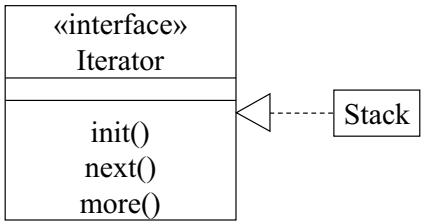
UML Notation

- To indicate that a class implements a particular interface, use the “lollipop” notation
- This is also called “realization”



UML Notation, continued

- When drawing an interface using a class diagram, realization is shown using the following notation



The fact that realization has two notations is, in my opinion, unfortunate.

Implementation Inheritance

- Generalization can be used to imply substitutability which can be realized by interface inheritance
  - Generalization can also be used to imply code reuse (the default) which is realized by implementation inheritance
  - Maciaszek points out that this is a (dangerously) powerful interpretation of generalization; lets see why

Implementation Inheritance

- Implementation inheritance allows subclasses to override the implementation of operations provided by a superclass
  - It can support both
    - calling the superclass implementation and extending it (good)
    - replacing the superclass implementation entirely (bad, why? hint: substitutability)

Extension Inheritance

- UML advocates the use of extension inheritance
  - A subclass has more properties (attributes and/or methods) than its superclass
  - The overriding of properties should be used with care
    - A subclass can only be allowed to make a property more specific (e.g. to constrain values or to make methods more efficient) not to change the meaning of a property
      - because if the meaning is changed, then substitutability is lost

Restriction Inheritance

- Restriction inheritance occurs when inherited properties are suppressed by subclasses ("pay no attention to the man behind the curtain")
  - See examples on page 182
- This runs counter to the notion of generalization and requires developers to realize that certain properties in the subclass must be ignored (even though they are present) which can lead to maintenance problems

Convenience Inheritance

- Inheritance that neither extends or restricts is "bad news"
  - Such inheritance can occur when two or more classes have similar implementations but no semantic (or taxonomic) relationships conceptually
    - For instance, a line segment is not a point but it can reuse some of the features of point (but only after extensive overriding of its operations)
    - See page 183

Problems of Implementation Inheritance

- Fragile Base class
  - allowing the implementation of a superclass to evolve after subclasses have been defined
    - changing the signature of a method
    - splitting a single method into one or more methods
    - joining multiple methods into a single method

Problems of Implementation Inheritance

- Overriding and Callbacks
  - Five types of overriding
    - reuse inherited method without change
    - call inherited method within own method (extension)
    - ignore inherited method (override)
    - inherited method is empty; supply own implementation
    - inherit interface only (interface inheritance)
  - These mechanisms set-up a web of calls that can occur between classes; a message passed to one object might be sent up or down the inheritance hierarchy in complex ways; see page 185

Problems of Implementation Inheritance

- Multiple Implementation Inheritance
  - Only a problem with languages that support multiple inheritance
  - multiple interface inheritance causes problems since multiple interface contracts must be merged into a single object
  - multiple implementation inheritance causes problems when multiple implementation fragments are merged into a single object

Advanced Aggregation and Delegation

- Maciaszek begins section 5.4 by reintroducing his variations to aggregation
  - ExclusiveOwns
  - Owns
  - Has
  - Member
- I will not repeat this material

Aggregation and Generalization

- More interesting is section 5.4.2
  - Aggregation as alternative to generalization
    - Generalization is superclass-subclass
    - Aggregation is superset-subset
      - but generalization can be represented as an aggregation, see page 190
  - Main distinction is that
    - generalization is predicated on the notion of class
      - inheritance implements semantics
    - while aggregation is centered on the notion of objects
      - delegation implements semantics

Delegation

- Whenever a composite object cannot handle a task by itself, it delegates the responsibility of completing the task to one of its component objects
- Delegation versus Inheritance
  - Delegation can be used to model inheritance and vice versa
  - In delegation, outer objects reuse the methods of inner objects; the difference is who retains control
    - a message sent to a subclass can be passed to the superclass via inheritance, but control returns to the superclass; in delegation control stays with the inner object
  - Delegation has one advantage over inheritance in that sharing and reuse can be done at run-time; reuse in inheritance is specified statically and cannot change