The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
13 April 2003

Notes – Multiple Inheritance

I. Concept
   a. It's not necessary to have just one superclass.
   b. Every instance of a subclass must behave like an instance of *each* superclass.
   c. It's mostly a simple concept,
   d. Take the union of all behaviors of all superclasses, and that defines the subclass.
   e. Representation is union of the superclasses' representation.
   f. Example: See CS100-31-19
   g. What about cases where representation overlaps?
      i. Has to be done by name.
      ii. Keep both sets? If so, what should they be called?
      iii. Keep just one? If so, which one?
   h. It's a little easier with methods
      i. Private methods aren't important.
      ii. The subclass can be required to write an implementation if it's not clear which should be used; that would replace implementation from both parents.
      iii. Could union both methods (do printing for student and teacher together, to replace that for Student and that for Teacher).
      iv. If it's a value-returning function, and two different return types are given, there's a problem.
   i. Diamond Inheritance
      i. What if both superclasses inherit from the same common super-superclass?
      ii. StudentTeacher is a subclass of Student and Teacher
      iii. Both Student and Teacher are subclasses of Person
      iv. This causes even more complications.

II. Object-Oriented Language Differences
   a. The biggest difference among object oriented languages is how they handle multiple inheritance.
   b. Java
      i. Supports the minimum.
      ii. One class can implement multiple interfaces.
      iii. No multiple inheritance with classes.
      iv. Eliminates a lot of the problems.
   c. C++
      i. Originally C++ did not support multiple inheritance at all.
      ii. People complained, so now it supports virtually everything (almost no restrictions are in place).
      iii. The Rub
         1. With pointer arithmetic, it should be possible to get from a pointer to the object to any field in that object.
         2. See CS100-32-8
         3. If two classes (A, C) are combined together to make a subclass (B), fields in C won't be the right distance from the beginning of B.
         4. It's not easy to assign C* when the C variables aren't first in memory. It has to add the size of A every time.
         5. Consider assigning two pointers of different types to the same supertype pointer – it has to yield two different addresses!
      iv. Virtual Calls
         1. The *this* pointer needs to point to the superclass, so it should be downcast when working with C variables, but ONLY if the instance is of the superclass!
         2. The compiler creates an extra function that takes the subclass pointer and downcasts it.

III. Overlapping Names

a. Java
    i. Methods with the same name are assumed to be the same.
    ii. Signatures MUST have the same return type.
    iii. The subclass exceptions must be a subset of the intersection of the superclass exceptions.
b. C++
    i. Fields are all considered distinct
    ii. If you have two fields with the same name, always use A::fieldname and B::fieldname
    iii. (That's always legal, but not usually helpful outside of this context.)
    iv. If no superclass function is virtual, the subclass is distinct.
    v. Use `b.A::f1();`
    vi. That's an explicit call, so even if A's function is virtual, it still gets called!
    vii. Virtual Functions
        1. If both supers inherit from the same parent, the subclass just replaces that one method.
        2. If not, see CS100-32-18
    viii. If more than one direct superclass gives an implementation, the subclass MUST redefine it or there's now ay to know which should be chosen.

IV. Diamond Inheritance
a. Not an issue in Java
b. HUGE issue in C++
c. Consider X as the parent of A and C, and those as parents of B.
d. The problems come back to pointer arithmetic again.
e. Virtual Inheritance
    i. Works similar to virtual functions.
    ii. If A and C have virtual inheritance, B gets only one copy of X.
    iii. That makes it diamond inheritance.
    iv. Always use virtual inheritance if you're using multiple inheritance.

V. The Middle Ground
a. Java is minimalist
b. C++ is insane
c. There are others, like CLOS (Common Lisp Object System) in the middle.
d. The General Case
    i. Have no public fields. Let all fields be used for representation only.
    ii. A good compiler can eliminate the overhead involved with the extra methods required for that.
    iii. Like Java, methods with the same name must be the same.
    iv. Require an implementation to be given by the subclass where both parents give one.
    v. The Rub
        1. Arranging things in memory is problematic.
        2. Define a hidden method like C++
        3. Java avoids this by simply not having state allowed in interfaces.
e. Another Idea
    i. Let interfaces define methods, but no representation.
    ii. Definitely let interfaces define *static* methods.
    iii. That avoids the replication of code in multiple children, but doesn't violate any of the rules already dictated about interfaces.

VI. Summary
a. Use multiple inheritance sparingly, particularly in C++
b. There are situations where the most natural solution involves multiple inheritance.
c. Design for the clean, simple case, and then fight with the language to get it done.
d. Whatever hoop-jumping the language requires should be hidden; keep the same specification even if there's a lot more to it behind the scenes.