Multiple Inheritance and Interfaces

Overview

• Multiple inheritance and its problems
  – i.e. creeping featuritis

• Java interfaces, C++ mix-ins, and “orthogonal properties”

• Specification and implementation classes
Multiple inheritance

• Basically, *multiple inheritance* (MI) means that a class inherits member variable and method *implementations* directly from more than one parent class.
  
  – A more limited version (*i.e.*, what Java does) allows inheriting implementations from only one parent, but only abstract methods from multiple parents (Java interfaces).
  
• This is not usually considered to be MI, strictly speaking.
Common uses of MI

1. Modelling real-world situations
   – Multiple related but distinct roles in the real-world
   – Often, these are mostly “passive” classes
     • *i.e.*, a class that only contains data with corresponding access methods.
     • Thus, there’s little implementation inheritance.

2. “Polymorphic hijacking”
   – There’s a neat set of classes for dealing with objects of type A, but for some reason you must root your hierarchy at B.
   – Create an ABC that inherits from both A and B.
   – Your objects are then treated polymorphically as either As or Bs.
Common uses of MI

3. “Cheap pickup of functionality”
   a) “Mix-ins”
      – Some class defines some useful routines intended for general use.
      – You can “mix-in” this functionality into your class by simply inheriting from the *mix-in* class.
      – Mix-ins are usually small, single purpose classes.
   b) Lazy design
      – You don’t (or can’t) redesign your class hierarchy
Problems of MI

• Most of the technical problems of MI boil down to trying to find the appropriate implementation of a method.
  – Instead, you end up finding more than one.

• Also:
  – Adding MI to an OOPL makes the language much more complicated.
    • It’s also complex to implement support for MI in the compiler.
  – As with all of OOP, misuse makes for monstrously complicated application code.
    • C++ systems in particular have a bad reputation for poor use of MI.
Name clashes

Q: Which `draw()` does `GraphicalCowboy` inherit by default?

C++ solutions:
- Must insist child to define a `draw()` method to disambiguate, or
- Don’t use `GraphicalCowboy:::draw()`
Inheritance from a common ancestor

- We probably don’t want two copies of the common features.

- C++ solution:
  - Use renaming if you want two copies
  - Use `virtual` inheritance in parent if not.
Creeping Featuritis (CF)

35 methods

LinkedList

Stack

8 methods

ListStack

ArrayStack

N methods including
MoveListPtrToNthElt

java.util.Vector

Java 1.1 library

(old)
Eiffel library

java.util.Stack
Creeping Featuritis

- Bad habit commonly observed in C++ world:
  - Want to adapt several ideas into one new monster class
  - Therefore, just use multiple inheritance! Right??
- Better idea:
  - Put some thought into how to design your classes.
  - Break into manageable, distinct, *essential* pieces.
  - Design the interfaces, think out the abstract relationships.
  - Compose via instantiation, use of containers, brokers, parameterization, *etc*.
- Importantly, sometimes you have to ask yourself if there is a conflict between an *is-a* and a *has-a* relationship.
What is Creeping Featuritis?

- Creeping featuritis (a real term!) causes *design rot* over time.

  1. Describes a systematic tendency to load more features onto systems at the expense of whatever elegance they may have possessed when originally designed.

  2. More generally, the tendency for anything complicated to become even more complicated because people keep saying "Gee, it would be even better if it had this feature too".
Creeping Featuritis (ctd.)

• Usually the best solution is to inherit from one class and instantiate as attributes from others.

• When you are considering whether class B should inherit from class A, ask yourself:
  – *Is each B also someone an A?*
    - Is a Rectangle really a Figure?
    - Is a ListStack really a LinkedList in its heart?
  – … or does each B really *contain* an A to aid in the implementation?
    - A Cowboy really *has-a* SixShooter, even if you can use the draw() method by inheriting from it.
Solving CF

1. **Exploit static typing**
   - Declare all stacks of static type `Stack` and then instantiate to `ListStack` (or `ArrayStack`).
   - Static typing will ensure that only features of `Stack` can be accessed by the instance
     - … unless of course the client downcasts
   - Awkward, requires programmer discipline, difficult to enforce
Solving CF

2. “Selective inheritance”
   • Hide individual features you don’t want clients to see by declaring them as private in the new class.
   • Fairly common in C++ world.
   • Awkward, requires lots of typing.
     • However, clients can defeat it by type trickery.
3. private inheritance

```cpp
class ListStack : public Stack, private LinkedList {
    // ...
};
```

- All of the public and protected features of the parent become private in the child
- This (intentionally) breaks polymorphism!
  - You cannot treat a ListStack as a LinkedList:
    - All LinkedList features are inaccessible to clients
    - Can’t instantiate a ListStack to a *LinkedList
    - Can’t pass a ListStack to a function expecting a LinkedList

But this is exactly what you would want!
Solving CF

- private inheritance is a bit of a conceptual abuse.
  - You’re breaking the spirit of information-hiding and encapsulation, albeit in a small, contained area.
    - You do still have all of those parent features floating around inside the child class.
  - Although it’s basically a hack, it’s really not too terrible as the effects are fairly well contained.
    - There does exist protected inheritance too.
  - Usual advice: use private inheritance over 1st two choices:
    - only to mode the is-implemented-by relationship, and
    - only when you need direct access to the parent’s protected features too.
Solving CF

4. **Best choice: don’t inherit, instantiate instead!**
   
   Often, `has-a` (instantiates) is the appropriate abstract relationship, but `is-a` (inherits) is used out of laziness; e.g., a `ListStack` `is-a` `Stack` that `has-a` `LinkedList` to help implement the stack abstraction.

```cpp
class ListStack : public Stack {
public:
    void push(EltType e) {s.AddAtNthPlace (e, 1);}
    EltType pop () {s.RemoveNthElt (e, 1);}
    ...
private:
    LinkedList s;
};
```
MI and Java Interfaces

• One of the main design goals of Java was to provide most of the functionality of C++ while removing the features that tend to make code complex, buggy, and hard to maintain:
  – Operator overloading
  – User-managed storage (i.e., delete)
  – *Multiple inheritance*

• The usual correct use of MI is to model *orthogonal properties* of an object (i.e. mutually independent)
  – It’s too common a situation to ignore in any reasonable language.
  – In Java, you use *interfaces* to achieve it.
Java interfaces

- Each class can extend exactly one parent (java.lang.Object by default)
  - There’s only one class you can inherit method implementations from.
  - Consequently there should be no confusion about “which implementation”

- Each class can implement multiple interfaces
  - All methods are abstract, all variables must be final
  - No method implementations ⇒ no confusion
Java interfaces

- Why bother?
  - Increased polymorphism
    - You can treat a D instance as if it were an A, B, or C.
    - You can pass a D instance as a parameter to any method expecting an A, B, or C.
  - To implement a kind of genericity
    - Where C++ would use a template, Java often instead requires that a class implement some explicit, named interface (e.g., Cloneable).
  - To model *separation of concerns*
    - This allows distinct, abstract properties to be factored out of application code.
    - Operations are defined such that they “understand” all of the objects which have these appropriate abstract properties.
An example using interfaces

• Suppose we have an efficient sorting algorithm that only requires that a class support an abstract idea of “less than”.
  – This is similar how STL uses the `less` function object or the overloaded operator: `operator<`.
  – Recall that Java does not allow operator overloading.

Solution:

1. Define an interface `Sortable` that has a `lessThan()` method.

2. Define a `sort` method that sorts an array of `Sortables`.

3. Any class that wants to use this method must implement `Sortable` and provide an appropriate definition of `lessThan()`.
```java
public interface Sortable {
    abstract boolean lessThan (Sortable s);
}

public class ShellSort { // Don’t sweat the algorithm details

    public static void sort (Sortable [] A) {
        int n = A.length, incr = n/2, i;
        while (incr >= 1) {
            for (i=incr; i<n; i++) {
                Sortable temp = A[i];
                int j=i;
                while (j>=incr && temp.lessThan(A[j-incr])) {
                    A[j] = A[j-incr];
                    j = j - incr;
                }
                A[j] = temp;
            }
            incr = incr / 2;
        }
    }

}
public class Employee implements Sortable {

    private int empNum;
    private String name;

    public boolean lessThan (Sortable s) {
        // Cast will throw an exception if
        // s is not an Employee
        Employee otherEmployee = (Employee) s;
        return this.empNum < otherEmployee.empNum;
    }
}

public class EmployeeDB {
    private Employee[] db;

    // ... other stuff

    public void sortDB () {
        ShellSort.sort(db);
    }
}
Just a minute here …

• Why not just make Sortable an ABC and then have Employee, Figure, etc. extend it?
  – Well we could, but …
    • The element type we want to sort may already extend another class and Java doesn’t allow MI.
    • Most interfaces express an abstract property that may be shared across many otherwise-unrelated classes.
      – Usually, an interface models only one aspect, property, or possible use of a class, often orthogonal to its main use.
      – Thus, some (predefined) Java interface names end in able
        » Cloneable, Serializable, Scrollable, Comparable, etc...
Interfaces and orthogonal properties

• Think of an interface as being

\[ \text{Generic object} \ + \ \text{interesting property} \]

– *i.e.*, `Sortable` represents a generic object that can be sorted.

• The common, modern idiomatic C++ approach is to simply use templated definitions.
  
  – *i.e.*, can use `ShellSort` if the class happens to support `operator<`

```cpp
template <typename T>
void ShellSort (T[] A){
  // ...
  if (A[i] < A[j-incr])
    // ...
}
```
C++ and orthogonal properties

• Templates allow the commonalities to be simply assumed …
  (e.g., the existence of an operator< or a lessThan() method)
  – … without the creation of a special entity that encapsulates the abstract
    property (i.e., that any element supporting operator< isSortable)

• Also, C++ encourages the use of function objects to create abstract
  strategies that can serve as parameters to other methods.
  – e.g., less() within the STL, which by default uses operator< of the
    element type, but you can provide your own idea of less too.
C++ and orthogonal properties

- Traditionally, orthogonal properties in C++ were implemented using “mix-in” classes.
  
  - These are small classes that define virtual (often pure virtual) methods of general utility.
  
  - Mix-ins are inherited as needed, usually via MI.
  
  - Unlike Java interfaces, mix-ins are often fully or partially defined inside the mix-in class itself.
Specification and implementation classes

- Another reason to use interfaces in Java
  - Want to specify the full interface of some ADS/ADT without providing any implementation details to clients.
    - This leaves great freedom to the implementor to make appropriate design decisions that will be (mostly, if not entirely) hidden from clients.
  - The “pure”-ness of the interfaces emphasizes to clients that the API is what is important here.
    - “Don’t make assumptions about how this will be implemented.”
  - Such a definition is often called a specification class.
    - It models a full ADT/ADS, not a narrow “orthogonal property”.
    - The implementor is called an implementation class.
  - Of course, you can do this in C++ too.
C++’s specification & implementation classes

• In C++, there is a tradition of defining ABCs with only pure virtual methods
  – The implementor class inherits `public`ly from the specification class.
    • The constructors are made `public`
    • All other methods are made `private` (or `protected`).

  – To use the implementor class:

    ```
    SpecClass *inst;
    inst = new ImplementorClass(...);
    ```

    *i.e.*, use the specification class as the static type of the pointer

• This guarantees very limited coupling is possible between the client and the implementor.
• Note that `inst` uses the access rights of the static class `SpecClass`, so it can get at all of its (expected) `public` methods.
MI and GUIs

- It’s well known that GUI code is very hard to write, understand, evolve, …
  - Use of event handling, control depends on message passing along non-obvious chains.
  - There are **LOTS** of small classes that are *almost* identical to other classes.
  - The structure of the GUI doesn’t resemble inheritance hierarchies
    - Lots of setup/takedown code
  - So, there are lots of cases to consider.
  - In many cases, there are GUI toolkit peculiarities, implementation dependencies, performance issues, etc…
- Thus, we often try to separate out the GUI layer (**“presentation”**) from the abstract functionality (**“application logic”**) of a system as much as possible to ease understandability and long-term maintenance.
- Generally, it’s a good idea to auto-generate GUI code as much as possible via special GUI builders.
MI and GUIs

- Historically, a very common and reasonable use of MI and mix-in classes is in implementing GUI toolkits.
  - Design a set of small functionality-based classes that can be combined (via MI) easily and reasonably as needed by clients.
  - Of course, some toolkits are better designed than others…

- Java’s lack of MI is in part responsible for the perceived awkwardness, slowness, etc. of its two GUI toolkits: AWT (Java 1.0, 1.1) and Swing (1.2).
  - This is because you must “load in” all of the functionality that you might need into a “monster library class”, rather than letting the client combine smaller library pieces as needed via MI.
  - Java does support JavaBeans, a neat mechanism for “wrapping” a piece of useful functionality and putting an interface around it for later reuse.
    - This is used to create subcomponents rather than designing a single class.
MI in summary

• MI is a good example of how OOPLs provide powerful techniques that, if carelessly used, can lead to very complicated code.
  – MI is complicated even when used correctly!
  – Abuse of MI has given it a bad reputation in industry.

• Sometimes using MI is the right and proper thing to do!
  – … however, Java and some other OOPLs don’t support the MI of method implementations
  – More often, the correct relationship is *has-a* rather than *is-a*.

• CS246 remark:
  – Don’t use MI or virtual inheritance; you won’t need to.