Programming Languages & Paradigms
PROP HT 2011

Lecture 6

Inheritance vs. delegation, method vs. message

Beatrice Åkerblom
beatrice@dsv.su.se

Abstraction & Modularity

Modularity

- Modular **Decomposability**
  - helps in *decomposing software problems* into a small number of less complex subproblems that are
    - connected by a simple structure
    - independent enough to let work proceed separately on each item
- Modular **Composability**
  - favours the production of *software elements which may be freely combined with each other* to produce new systems, possibly in a new environment

Modularity, cont'd.

- Modular **Understandability**
  - if it helps produce software in which a human reader can *understand each module without having to know the others*, or (at worst) by examining only a few others
- Modular **Continuity**
  - a small change in the problem specification will trigger a *change of just one module*, or a small number of modules

Modularity, cont'd.

- Modular **Protection**
  - the *effect of an error at run-time in a module will remain confined to that module*, or at worst will only propagate to a few neighbouring modules

Classes aren't Enough

- Classes provide a good modular decomposition technique.
  - They possess many of the qualities expected of reusable software components:
    - they are *homogenous*, coherent modules
    - their *interface* may be clearly separated from their implementation according to information hiding
    - they may be *precisely specified*
- But more is needed to fully achieve the goals of reusability and extendibility

Polymorphism

- Lets us wait with binding until runtime to achieve flexibility
  - Binding is not type checking
- Parametric polymorphism
  - Generics
- Subtype polymorphism
  - E.g. inheritance

Static Binding

- Function call in C:
  - Bound at compile-time
  - Allocate stack space
  - Push return address
  - Jump to function

Dynamic Binding

- Method invocatoin in Ruby:
  - Does the method exist?
  - Is it public?
  - Are the number of arguments OK?
  - Push it into local method cache
  - *Now*, start calling

Static vs. Dynamic Binding

- **Static** binding:
  - **Efficiency**—we know exactly what method to dispatch to at compile-time and can hard-code that into the object code (or whatever we compile to)
  - Changing binding requires recompilation, arguably against the “spirit of OO”
  - Very simple to implement (and easy to reason about)
- **Dynamic** binding:
  - **Flexibility**—supports program evolution through polymorphism
  - Harder to implement, especially in the presence of multiple inheritance and wrt. efficiency

Late Binding

- A form of dynamic binding found in e.g., C++, Java and C#
- Requires that types are known at compile-time and inclusion polymorphism (overriding)
- Example:
  - During type checking, we can determine that the type of `p` is some subclass of `Person`
  - We require that `setName( String )` is present in `Person` and can thus avoid errors of the type “`MessageNotUnderstood`”
  - Safer, and still much more flexible than static binding

Polymorphism

- Lets us wait with binding until runtime to achieve flexibility
  - Binding is not type checking

- Parametric polymorphism
  - Generics
- Subtype polymorphism
  - E.g. inheritance

Inheritance

Substitution Revisited

- B <: A -- B is a subtype of A
- Any expression of type A may also be given type B
  - in any situation
  - with no observable effect
- Nominal subtyping or structural subtyping?
- Almost always:
  - reflexive (meaning A<:A for any type A)
  - transitive (meaning that if A<:B and B<:C then A<:C)

Substitution in a STL

- The method below will only operate on arrays of instances of the BaseballPlayer class, (or instances of subclasses of BaseballPlayer)

```
public int sumOfWages( BaseballPlayer[] bs ) {
   int sum = 0;
   for ( int i=0; i < bs.length; ++i ) {
      sum += bs[ i ].wage( );
   }
   return sum;
}
```


Substitution in a DTL

- In a dynamically typed language, you can send any message to any object, and the language only cares that the object can accept the message — it doesn't require that the object be a particular type

```
def sumOfWages( aList ):
      sum = 0
      for item in aList:
            sum += item.wage( )
      return sum
```

Substitution in DPLs and STLs

- The importance of the principle of substitution differs between dynamically typed and statically typed languages
  - in statically typed languages objects are (typically) characterised by their class
  - in dynamically typed languages objects are (typically) characterised by their behaviour

Inheritance in DPLs and STLs

- The importance of inheritance differs between dynamically typed and statically typed languages
  - in statically typed languages subclasses inherit specifications (interfaces) and sometimes also behaviour (implementation)
  - in dynamically typed languages subclasses inherit behaviour (implementation)

What is Inheritance?

- Inheritance gives us the possibility to create something that is partly or totally the same as something else
  - Child classes as **extension** of an already existing class definition
  - Child class as a **specialisation** of an already existing class definition
  - Enables subtypes to produced using an already existing supertype

Javascript

Forms of Inheritance

- Inheritance for
  - **specialisation** (subtyping) -- the new class is a specialised form if the parent class
  - **specification** -- to guarantee that classes maintain a certain interface
  - **extension** -- adding totally new abilities to the child class
  - **limitation** -- the behaviour of the child class is more limited than the behaviour of the parent class (violates the principle of substitution)
  - **variance** -- when two or more classes have similar implementations, but no relationships between the abstract concepts exist
  - **combination** -- multiple inheritance

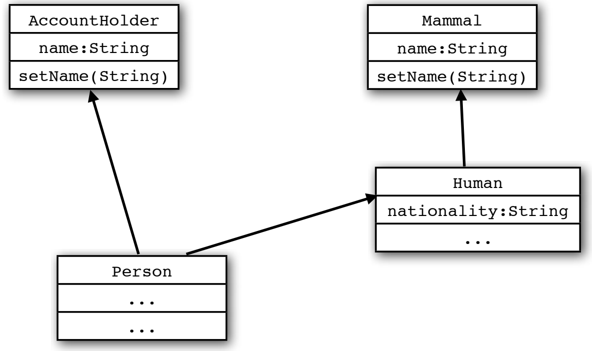
Single and Multiple Inheritance

- Multiple inheritance allows a class to inherit from one or more classes
- Sometimes convenient, natural and valuable
- Increases language and implementation complexity (partly because of name collisions)
- Potentially inefficient - dynamic binding costs (even) more with multiple inheritance (but not that much)

Multiple inheritance

- Multiple interface inheritance (Java, C#)
  - Can inherit from more than one protocol specification, but cannot inherit implementation details from more than one source
- Multiple implementation inheritance (C++, Python)
  - What is generally meant by “multiple inheritance”
  - Protocol and implementation is inherited
- Problems
  - Ambiguous lookup
  - Memory layout
  - Clashes

Ambiguous lookup



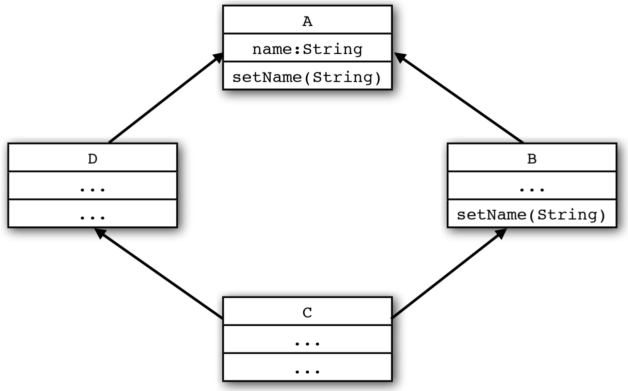
Stockholm University

Solutions

- Multiple dispatch (still need to consider order)
- Require renaming or use of qualified names or reject programs with conflicts
- Employ a specific strategy
  - Graph inheritance
  - Tree inheritance
  - Linearisation
- Use of different strategies in different PLs (or impls. of the same PL) affects a program's portability
- Opportunity for subtle bugs due to lookup complexity

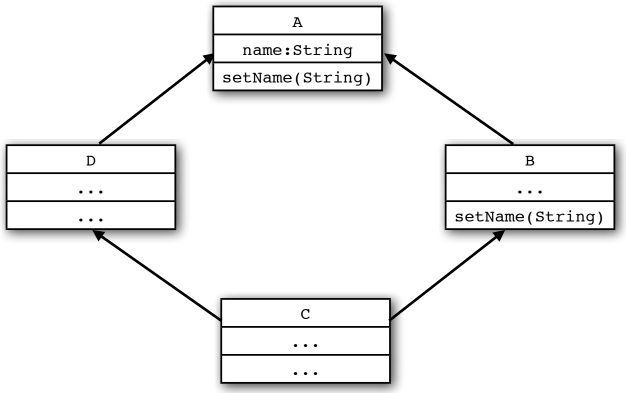
Stockholm University

Diamond problem



Graph Inheritance

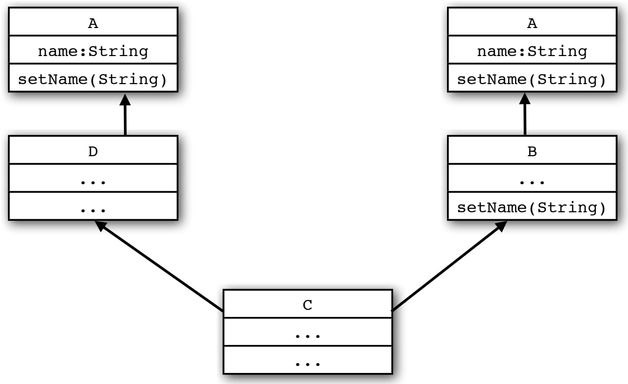
Stockholm University



- Possible multiple dispatch of same method
- Shared fields may break encapsulation
- Fragile inheritance situation
- Cannot deal with conflicting invariants on fields

Tree Inheritance

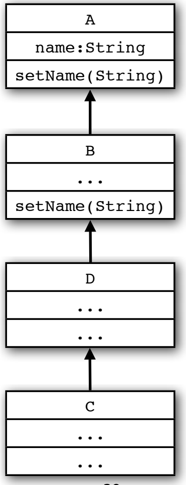
Stockholm University



- Separate copies of superclasses' implementation
- Does not work well when only one field f is sensible
- Does not work well if both A::m and C::m should be invoked as D::m — renaming

Linearisation

- Transform hierarchy into a single inheritance hierarchy without duplicates
- Transformation may or may not be under programmer control
- Order of linearisation effects the program!s semantics
- A::m is overridden by C::m in our example
- D is given B as a superclass, unknown to D's programmer — possibly changing the meaning of super in D



Languages with Multiple Inheritance

- C++ — graph or tree inheritance under programmer control (very subtle though)
- CLOS — linearisation
- Eiffel — tree inheritance or linearisation under programmer control
- Python
  - "New style" classes use linearisation
  - "Old style" classes go depth-first and then left to right
  - As objects are dynamically typed hash tables, field clashes are less of a problem

Avoiding Multiple Inheritance

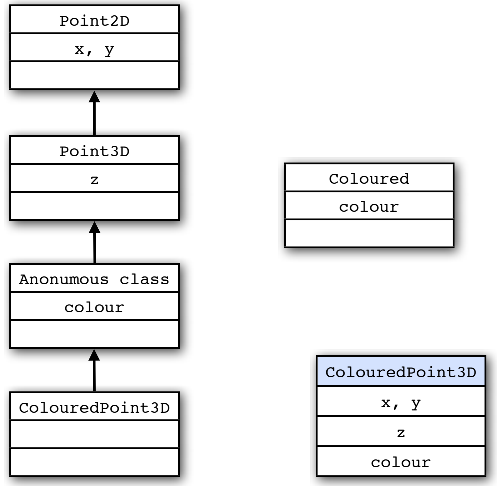
- When the question of whether to use multiple inheritance comes up, ask at least two questions:
  - Do you need to show the public interfaces of both these classes through you new type?
  - Do you need to upcast to both of the base classes?
  - If your answer is "no" for either question, you can avoid using multiple inheritance and should probably do so

Mixin Inheritance

- Creating an inheritance hieararchy by mixing modules or classes
- A mixin is an abstract subclass that may be used to specialise the behaviour of various superclasses
- A mixin is a freestanding record of extra fields, intended to be combined with any other object
- If `C` is a class and `M` is a mixin, we can create class `D` by saying `let D = M extends C`
- Reduces to a single inheritance structure with an explicit linearisation, controlled by mixin order
- Can be used to model both single and multiple inheritance

Mixins in Ruby

```
class Point2D
  attr_accessor :x, :y
end
class Point3D < Point2D
  attr_accessor :z
end
module Coloured
  attr_accessor :color
end
class ColouredPoint3D < Point3D
  include Coloured
end
```

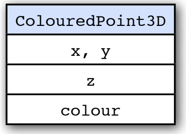
Different Takes on Overriding Methods

- In Java and Smalltalk descendent methods control whether ancestor methods are called
- C# does support method overriding, but only if explicitly requested with the keywords override and virtual
- In Simula, ancestor methods control whether descendent methods are called - and the ancestor methods will be called first anyway
- In Eiffel, descendents can cancel or rename ancestor features.

Problems With Inheritance

Inheritance Breaks Encapsulation

- Inheritance exposes a subclass to details of its parent's implementation



Fragile Base Class

- Seemingly safe modifications to a base class may cause derived classes to break
- You can't tell whether a base class change is safe simply by examining the base class's methods in isolation, you must look at (and test) all derived classes as well.
  - you must check all code that uses both base-class and derived-class objects too, since this code might also be broken by the new behavior.
  - a simple change to a key base class can render an entire program inoperable

Fragile Base Class -- Example

```
Bag = class
    b : bag of char

    init ≙ b := ⟦ ⟧
    add(val x : char) ≙
        b := b ∪ ⟦x⟧
    addAll(val bs : bag of char) ≙
        while bs ≠ ⟦ ⟧ do
            begin var y | y ∈ bs·
                self.add(y);
                bs := bs − ⟦y⟧
            end
        od
    cardinality(res r : int) ≙
        r := |b|
end
```

```
CountingBag = class
inherits Bag
    n : int

    init ≙ n := 0; super.init
    add(val x : char) ≙
        n := n + 1; super.add(x)

    cardinality(res r : int) ≙
        r := n
end
```


Fragile Base Class -- Example

```
Bag = class
    b : bag of char

    init ≙ b := ⟦ ⟧
    add(val x : char) ≙
        b := b ∪ ⟦x⟧
    addAll(val bs : bag of char) ≙
        while bs ≠ ⟦ ⟧ do
            begin var y | y ∈ bs·
                self.add(y);
                bs := bs − ⟦y⟧
            end
        od
    cardinality(res r : int) ≙
        r := |b|
end
```

```
CountingBag = class
inherits Bag
    n : int

    init ≙ n := 0; super.init
    add(val x : char) ≙
        n := n + 1; super.add(x)

    cardinality(res r : int) ≙
        r := n
end
```

```
Bag′ = class
    b : bag of char
    init ≙ b := ⟦ ⟧
    add(val x : char) ≙ b := b ∪ ⟦x⟧
    addAll(val bs : bag of char) ≙ b := b ∪ bs
    cardinality(res r : int) ≙ r := |b|
end
```

Stockholm University

Examples of Errors

- Unanticipated Mutual Recursion
- Unjustified Assumptions in Revision Class
- Unjustified Assumptions in Modifier
- Direct Access to Base Class State
- Unjustified Assumptions of Binding Invariant in Modifier

Stockholm University

How can we prevent them?

- “No cycles” requirement
- “No revision self-calling assumptions” requirement
- “No base class down-calling assumptions” requirement
- “No direct access to the base class state” requirement

Is that enough?

Stockholm University

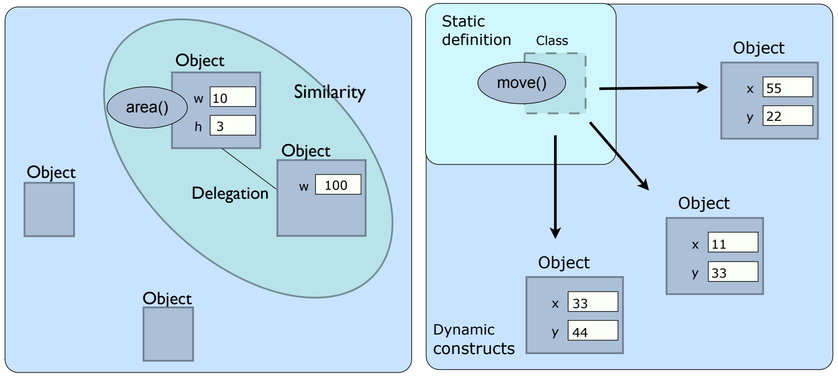
Do We Really Need Inheritance?

- What do we really need to use inheritance to achieve?
- What do we really gain?
- Is it worth all the problems?
  - Inheritance breaks encapsulation
- Can we use other solutions instead?
  - Many proposals suggest that inheritance should be decomposed into the more basic mechanisms of object composition and message forwarding
  - Delegation?

Stockholm University

Delegation

Stockholm University



Stockholm University

Delegation Breaks Encapsulation Even More?

Stockholm University

Reintroducing Order: Traits

- Traits allows us to factor out common behaviour from several objects and collecting it in one place
  - A trait provides a set of methods that implement behaviour
  - A trait requires a set of methods that serve as parameters for the provided behaviour
  - Traits do not specify any state variables, and the methods provided by traits never access state variables directly
  - Classes and traits can be composed from other traits, but the composition order is irrelevant. Conflicting methods must be explicitly resolved by trait composer

Comparable Trait

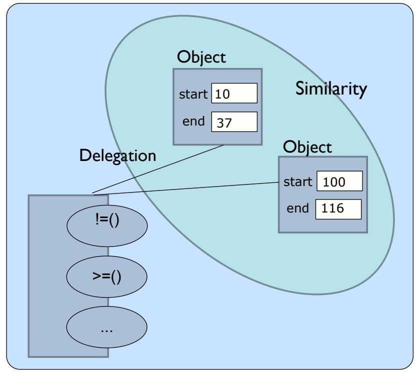
Stockholm University

```
var ComparableTrait = Trait({
  '<': Trait.required,
      // this['<'](other) -> boolean
 '==': Trait.required,
      // this['=='](other) -> boolean

 '<=': function(other) {
   return this['<'](other) || this['=='](other);
  },
  '>': function(other) {
   return other['<'](this);
  },
 '>=': function(other) {
   return other['<'](this) || this['=='](other);
  },
 '!=': function(other) {
   return !(this['=='](other));
  }
});
```

```
function makeInterval(min, max) {
  return Trait.create(Object.prototype,
    Trait.compose(
      EnumerableTrait,
      ComparableTrait,
      Trait({
        start: min,
        end: max,
        size: max - min - 1,
        toString: function() { return ''+min+'..!'+max; },
        '<': function(ival) { return max <= ival.start; },
        '==': function(ival) { return min == ival.start && max == ival.end; },
        contains: function(e) { return (min <= e) && (e < max); },
        forEach: function(consumer) {
          for (var i = min; i < max; i++) {
            consumer(i,i-min);
          }
        }
      })));
}

var i1 = makeInterval(0,5);
var i2 = makeInterval(7,12);
i1['=='](i2) // false
i1['<'](i2) // true
```

Method vs. Message vs. Function

Method Invocation

- Calling a subroutine

Message Passing

- Objects send and receive messages
- The response to a message is executing a method
- Which method to use is determined by the receiver at run-time.
- Messages can be passed synchronously or asynchronously
- Messages can be sent to an "unknown" object (you may not know its exact identity, type or location)
- Message not understood

The End

References

Stockholm University

- Sebesta, R. - "Concepts of Programming Languages"
- Budd, T. - "An Introduction to Object- Oriented Programming", 2nd edition. Addison-Wesley, 2000.
- Craig, I. - "The Interpretation of Object- Oriented Programming Languages", 2nd edition,SpringerVerlag, 2002.
- Joyner, I. - "Objects Unencapsulated", Prentice-Hall, 1999.
- Lieberman, H. - "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", 1986.
- Anton Eilëns, Principles of Object-Oriented Software Development, 2nd edition. Addison-Wesley, 2000.
- Bertrand Meyer, Object-oriented Software Construction, 2nd edition, Prentice-Hall 1997.
- Barbara H Liskov and Jeannette M Wing, A Behavioural Notion of Subtyping, 1994.

References, cont'd

Stockholm University

- Lynn Andrea Stein, Delegation is Inheritance.
- Mikhajlov & Sekerinski, A Study of the Fragile Base Class Problem
- Luca Cardelli, Semantics of multiple inheritance, 1984
- Weck & Szyperski, Do We Need Inheritance? (1996)
- Traits for Javascript, http://traitsjs.org/