Interfaces, Composition, and Inheritance

CS342 - Fall 2016

Interface and Implementation

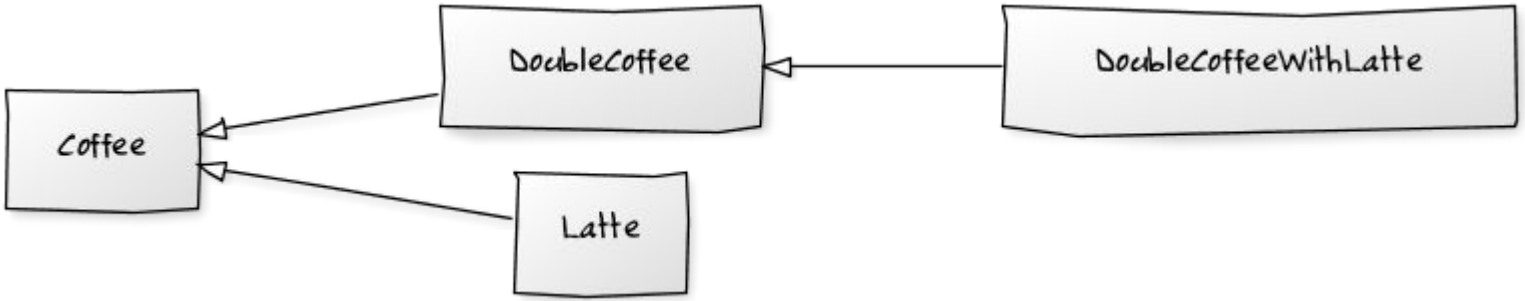
```
class VehicleInterface
     def start()
          raise NoMethodError
     end
end

class Car < VehicleInterface {
     #Required to implement start()
     def start()
          ...
     end
end
```

Ruby does not have interfaces in the Java sense, but we can fake it because it does have overriding methods

Interfaces are Final

- Programming to an interface, not an implementation allows:
  - flexibility
  - decoupling from the implementation
  - the implementation to vary or to be replaced



Interfaces Guarantee behaviors

- If a class is a subclass of an interface, the interfaces behaviours are guaranteed
- This allows parameter abstraction for methods
  - Assume the class superhero uses the interface power:
    - *class SuperHero*

      def aquirePower(new_power)

      @power = new_power

      end

      def usePower(new_power)

      #how to use power?

- class SuperHero
  def aquirePower(new_power)
  @power = new_power
  end
  def usePower(new_power)
  @power.use() #guaranteed method
  end
  end

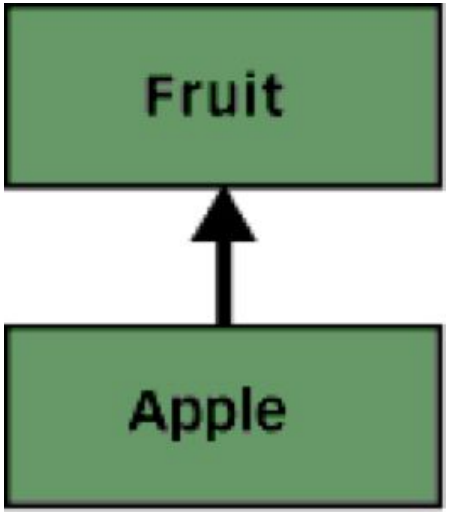
Composition vs Inheritance

```
class Fruit
end
class Apple < Fruit
end
```

- In this simple example, class Apple is related to class Fruit by inheritance, because Apple extends Fruit. Fruit is the superclass and Apple is the subclass.

Inheritance Relationship

(IS-A relationship)



Composition Relationship

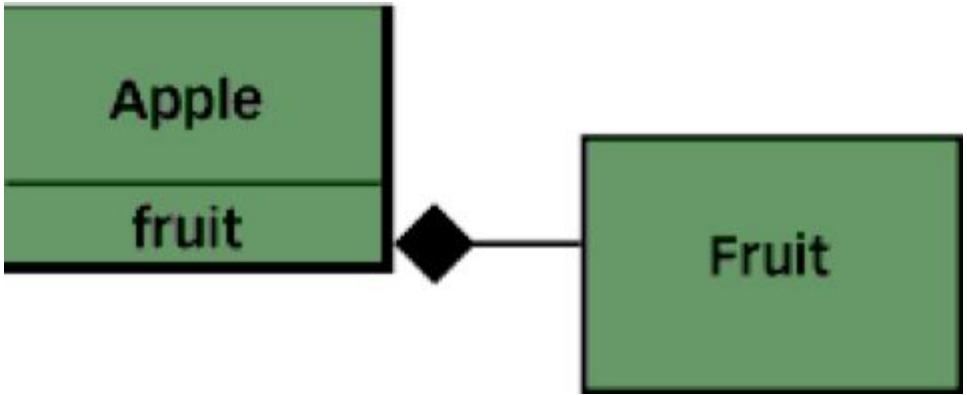
- Composition means using instance variables that are references to other objects. For example:

```
class Fruit
end
class Apple
      @fruit = new Fruit.new()
end
```

- Class Apple is related to class Fruit by composition, because Apple has an instance variable that holds a reference to a Fruit object.
  - Apple is the delegating class
  - Fruit is the back-end class
- In a composition relationship, the front-end class holds a reference in one of its instance variables to a back-end class.

Composition

(HAS-A)



Difference in Method Invocation

Inheritance

- A subclass automatically inherits the implementation of any non-private superclass method that it does not override.

Composition

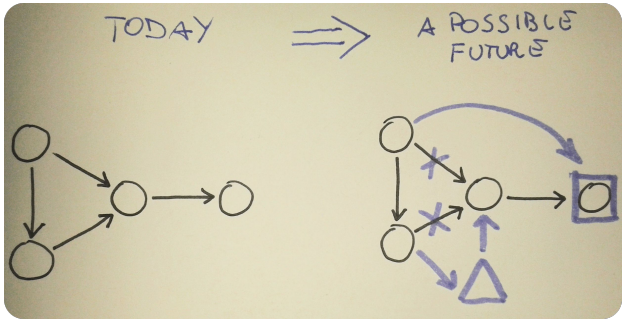
- The front-end class must explicitly invoke a corresponding method in the back-end class from its own implementation of the method.
- This explicit call is called ***delegating*** the method invocation.
    - if apple.peel() method immediately calls fruit.peel(), then the apple method is delegating behavior

Composition > Inheritance

- Composition provides stronger encapsulation than inheritance
  - why?
    - *Because you only inherit the functionality you need*
- Changes to a back-end classes won't break any front end code
  - why?
    - *because the delegating class can intercept and 'translate' any changes*

Inheritance is weak encapsulation

- If the subclass doesn't override the superclass method, it will reuse the implementation of the superclass
- Changes in the superclass interface may break dependent code
  - Therefore, it is called weak encapsulation
    - *For example, you use class B from a class library, but what you may not know is that the methods you are using are inherited from class A. Changes in class A alters the return value from a boolean to an integer,*



Composition vs Inheritance: Example

```
#inheritance
class Fruit
     def peel()
          return 1
     end
end
class Apple < Fruit
end

mac = Apple.new()
num_peels = mac.peel()
```

```
#composition
class Fruit
      def peel()
            return 1
      end
end
class Apple
      @fruit = Fruit.new()
      def peel(){
            return fruit.peel()
      end
end
mac = Apple.new()
num_peels = mac.peel();
```

Changes to Base class

```
#inheritance
class Fruit
     def peel()
          return self
     end
end
class Apple < Fruit
end

mac = Apple.new()
num_peels = mac.peel() #breaks
```

```
#composition
class Fruit
      def peel()
            return self
      end
end
class Apple
      @fruit = Fruit.new()
      def peel(){
            return 1
      end
end

mac = Apple.new()
num_peels = mac.peel(); #no changes necessary
```

Driver code does not have to change if Composition is used though the code in the Apple class has to be modified.

Composition over Inheritance

- This new code for Fruit and peel() does not break the compilation of Apple.
- When Apple used inheritance, the user code would break on changes to the base class
- If using composition, just a change in Apple would be required.
    - Driver code would not break

When Composition? When Inheritance?

- Changes stop at the back-end subclasses with composition. The front-end code does NOT change.
  - The goal is to eliminate the need to rippling changes
    - *B changes so C must change so D must change....*
  - There is no 100% bulletproof method to eliminate rippling changes
- Adding new subclasses is easier with inheritance
  - less code, less chance for error
- Composition is more expensive (delegation cost) as opposed to a single (virtual) call in inheritance.

Classwork 3 - Part 1

- Solutions
  - Automobile IS_A Vehicle
  - Groceries HAS_A Food
  - Project IS_A Assignment
  - Employee HAS_A Person
  - Child HAS_A Parent
  - Book HAS_A Author --or-- Author HAS_A Book

IS-A Test

- Inheritance represents IS-A relationship. The subclass IS-A superclass relationship should last the lifetime of the application:
  - Apple IS-A Fruit
    - *Good candidate for inheritance*
  - Superhero IS-A Person
    - *What if the Person loses powers*
    - *At that instance, this relationship will not hold true*
    - *Good candidate for composition*

Composition vs Inheritance: general guidelines

- ***Design Principle:* Favor Composition over Inheritance**
- Composition gives more flexibility in terms of being easier to change code.
- Lets you change behavior at run-time
  - Ruby only has Single Inheritance
  - Composition allows for multiple inheritance
- Composition is used in many design patterns