Variations on Inheritance

Object-Oriented Programming

236703

Winter 2015-6
Utopia and Reality

• The OOP manifest talks about objects, classes and hierarchies
  • Clean, but limited model
• In real world, programmers need more tools
  • Some concepts are abstract
  • Some operations are common (serialize, undo, log), yet modeling them as objects makes no sense
    • Class = noun, Operation = verb
• Different languages offer different variations
Today's Agenda

1. Abstract classes
2. Interfaces vs. abstract classes
3. Structural interfaces
4. Final classes and methods
5. MultiMethods
6. Mixins and traits
Circle and Ellipse Problem

Is a Circle a kind-of an Ellipse?

**Problem**: Ellipse has a `scale(x, y)` method, which is meaningless for Circle.

**Poor Options:**

- `Circle::scale(x, y)` is an error (throw an exception)
  - **Pitfalls**: surprising the clients
- `Circle::scale(x, y)` is a no-op / scales both dimensions by average
  - **Pitfalls**: very difficult to document
Circle and Ellipse Problem

Is an Ellipse a Kind of Circle?

**Problem**: Circle should have center and radius methods

- **How should** `getCenter` and `getRadius` **be defined** in Ellipse?
- **We cannot delete** methods in inheritance!
  - **We can sometimes** *hide* methods or make them less visible, but that’s awkward and not very useful
Oval to the Rescue

• Both ellipse and circle are oval shapes – why not have both derive an **Oval** class?
  • Move shared methods to Oval, leave unique methods to deriving classes

• Conceptual problem:
  • What is the real-world equivalent of a new **Oval()**?
  • If an **Oval** can be created, an **Oval** will be created

• How can we avoid the problem?
Abstract Classes

• Abstract Class: A class that cannot be instantiated
  • Does this mean it cannot have a constructor?

• An abstraction mechanism that helps model reality
  • No, not “Big Brother” kind of reality – the real world!
  • Some concepts are important and well defined, but only exist as ideas.

• Commonly used as the base of an inheritance hierarchy, which uses subtyping
The Abstract Superclass Rule

- *All superclasses must be abstract* [Hürsch ’94]
  - Arguably models the real world
    - Entities of concrete type and concrete sub-type rarely coexist
  - Control inheritance
    - On the next slide
  - Control polymorphism
    - Variable of abstract type is polymorphic, variable of concrete type is not
  - Easier to maintain code
    - A change to abstract type propagates, a change to concrete type does not
Back To Circle and Ellipse Problem

• Define an *abstract* Oval class
• Have:
  • Ellipse and Circle derive from Oval
  • A scale(x, y) method of Ellipse
  • A scale(factor) method of Circle

• Now some concrete shapes/types can be treated as ovals, but no concrete oval object can be created
Abstract Methods

• A concept *complementing* overriding:
  • Overriding is a *re-implementation* of a method in an inheriting class
  • Abstract methods are *deferred* – their implementation is postponed to an inheriting class

• Usage:
  • Describe a protocol
    • Allowing partial internal implementation using other methods (template method pattern)
    • Different derived classes will provide different implementations
  • Make classes abstract
    • Does this *always* make sense?
Pure Virtual Functions in C++

class Shape {
    virtual void rotate(int angle) = 0;
    virtual void hide() const = 0;
};

• Pure Virtual Functions (PVF): virtual member functions which require an implementation in a derived class
  • A C++ terminology
• C++ PVFs vs. Java/C# abstract methods:
  • PVFs may, but usually don’t, have an implementation
  • Abstract methods cannot have an implementation
• Abstract Class in C++: A class with PVFs
• Concrete Class in C++: A class which is not abstract
  • Only concrete classes may have instances
Pure Virtual Functions in C++

```cpp
class Shape {
  virtual void rotate(int angle) = 0;
  virtual void hide() const = 0;
};
```

• Syntax is not very expressive
  • “= 0” is just an arbitrary sequence of characters
• Having a PVF should make the class abstract, but must an abstract class have a PVF?
  • In Java and C#, methods and classes can be defined as *abstract*
  • Can you make a C++ class abstract without making any “regular” member function pure virtual?

Pure Virtual Destructor

• Pure virtual destructor **must** be defined
  • Destructors cannot be overridden

• It must be defined outside the class declarations
  • Just as all defined pure virtual functions

• A class defining such a destructor is abstract, but this definition does not make any derived class abstract
  • No need to (explicitly) define a destructor in any derived class to make it concrete

```cpp
class Abstract {
public:
    virtual ~Abstract() = 0;
};

Abstract::~Abstract() { /* ... */}
```
Quiz: Spot The Errors

```cpp
struct Vehicle {
    virtual const char* media() const = 0;
    virtual unsigned speed() const = 0;
} V, *PV;
struct LandVehicle : public Vehicle {
    const char* media() const override { return "Land";}
} L, *PL;
struct Train : public LandVehicle {
    virtual unsigned speed() const override { return 130; }
} T, *PT;

PV = &V; PV = &L; PV = &T;
PL = &V; PL = &L; PL = &T;
PT = &V; PT = &L; PT = &T;

Vehicle &RV1, &RV2 = L, &RV3 = T;
LandVehicle &RL1, &RL2 = L, &RL3 = T;
Train &RT1, &RT2 = L, &RT3 = T;
```
Java: Interface vs. Abstract Class

• Both specify a type

• Interface:
  • Pure protocol specification
  • Only (abstract) method specifications and constant definitions

• Abstract class:
  • Method specification and optionally:
    • Partial or full default method implementation, instance variables, constructors
  • Any class with abstract methods must be an abstract class
  • Abstract classes do not necessarily need to have abstract methods

• What about inheritance?
  • Remember: Java allows a single class but multiple interfaces
Structural Interfaces

-  Java/C# interfaces:
  - Represent a protocol, which is a contract
  - Implemented via inheritance
  - Prevent dependencies by hiding implementation
-  If the contract requires certain methods, do we really care what type (interface or class) provides them?
  - Often, yes. It's part of the abstraction
    - Employee.fire() is different than Gun.fire()
  - Sometimes, no. So why not simplify our code?
Go Interfaces

type Shaper interface {
    Area() int
}

func (r Rectangle) Area() int { // no relation to Shaper
    return r.length * r.width
}

func (sq Square) Area() int { // no relation to Shaper
    return sq.side * sq.side
}

func Print(s Shaper) { // take any matching type
    fmt.Println("Area: ", s.Area())
}

credit: http://golangtutorials.blogspot.co.il/2011/06/interfaces-in-go.html
Go Interfaces

• Semantically, Go interfaces describe *capabilities*, not *types*
  • Different kinds of abstractions
  • Acceptable in other languages interfaces: Comparable
• A type matches an interface if they are structurally compatible
  • Subtyping without inheritance!
• Similar to function pointers in C/C++
  • But one interface can have multiple methods
Are Go Interfaces Dynamic?

• If we can throw any unrelated type at an interface, does it mean Go has dynamic typing?
  • No!
    • Compatibility is checked at compile time
    • Use of the interface is guaranteed to succeed
  • So:
    • Structural interfaces make code simpler
    • Stronger guarantees than dynamic variables
    • Vulnerable to semantic mismatches
Final Classes and Methods

- Final Method: cannot be overridden
  - Call can safely be statically bound and even inlined
- Final Class: cannot be further derived
  - Implies that all methods are finalized
  - No dynamic binding mechanism is required
- Food for thought:
  - Can a final class have abstract methods?
  - Can a final class be abstract (no abstract methods)?
  - Can an abstract class have final methods?
  - What’s easier: turn final or non-final?
MultiMethods

- *Overriding* allows multiple implementations of the same method; selection is done at run time
  - Base::foo(int), Derived::foo(int)
- *Overloading* allows multiple methods with the same name; selection is done at compile time
  - bar(Base*), bar(Derived*)
- Can we choose among overloads by dynamic types?
  - Base* b = GetBaseOrDerived();
    bar(b); // overload determined at run time
MultiMethods

- **Multiple Dispatch**: the executed method is selected by the dynamic type of one (or more) argument(s)
  - Overloads possibly not even in the same class – Base defines f(A*), Derive adds f(B*)
- **MultiMethod**: a method that takes part in multiple dispatch
  - I.e., can be chosen at run time based on –
    1. Dynamic type of receiver (a-la *overriding*)
    2. Dynamic types of arguments (a-la *overloading*)
Example: Binary Methods

```
public class Shape {
    public void intersect(Shape s) {
        // generic code for two shapes
    }
}

public class Rectangle extends Shape {
    public void intersect(Rectangle s) {
        // more efficient code for two rectangles
    }
}
```

Shape s = new Shape();
s.intersect(new Shape());
// no problem: Shape.intersect() is invoked

s = new Rectangle();
s.intersect(new Rectangle());
// two rectangles but Shape.intersect() is invoked again -
// overloading resolution happens at compile time!
public class Shape {
    public void intersect(Shape s) {
        // generic code for two shapes
    }
}

public class Rectangle extends Shape {
    public void intersect(Shape@Rectangle s) {
        // more efficient code
    }
}

Shape s = new Shape();
s.intersect(new Shape());
    // no problem: Shape.intersect() is invoked
s = new Rectangle();
s.intersect(new Rectangle());
    // no problem: Rectangle.intersect() is invoked
    // new method is both an overload and an override

Static type - override
Dynamic type - overload
Drawbacks of MultiMethods

• Expensive dispatch process
  – Must examine dynamic types of all arguments and make yet another method call

• Possible run time ambiguity
  – Given $A::f(A, A)$, $B::f(B, A)$ and $B::f(A, B)$
  – What happens if you pass two $B$’s to $A::f$?

• Can get the same functionality in standard Java
  – Requires “manual” dispatch
    – or double dispatch design pattern
Merging Functionality

Say we need an Integer class with Undo support:

• Inheritance not always appropriate
  • Single inheritance violates is a relation:
    ```
    class Integer : public Undo
    // is Integer really an Undo?
    // must all Integers be Undo-able?
    ```
  • Multiple inheritance needs additional code:
    ```
    class UndoableInteger : public Integer, Undo
    // what binds Integer and Undo functionality?
    ```
Merging Functionality

Say we need an Integer class with Undo support:

• Functions abstract operations, but aren’t bound to specific classes
  • Separate receiver from method

```cpp
void Undo(Integer& i)
// how does Integer user know about Undo?
// need all Integers be undoable?
// is this Undo reusable?
```
Merging Functionality

Say we need an Integer class with Undo support:

- Composition does not extend protocol and behavior
  - Require cumbersome wrapper methods

```cpp
class UndoableInteger : public Integer {
    Undoer undoer;

public:
    void Undo() { undoer.undo(this); }
};
```
Mixins and Traits

- Mixins and traits are language mechanisms that allow merging functionality of different classes, without relying on conventional inheritance.

- **Overloaded terms!**
  - Scala defines mixins using the keyword `trait`.
  - D mixins are like C macros.
  - Boost.mixin and Java’s CGLIB create mixins at run time.

- **Our goal: understand the need, learn possible solutions**
Mixins vs. Traits in a Nutshell
Mixin

• C++ Mixin: a subclass parametric in the superclass

• The problem:

```java
class UndoInt extends Int { Undo() {...} }
class UndoChar extends Char { Undo() {...} }
```

• Drawbacks:
  – Code duplication – UndoInt and UndoChar add the same
  – Inability to use both undos in a uniform way (the extensions are not a type)
struct Int {
  int n;
  virtual void setVal(int v) { n = v; }
  int getVal() const { return n; }
};

template<typename Base> // Define the mixin
struct UndoMixin : Base {
  int old;
  void setVal(int v) override { old = getVal(); Base::setVal(v); }
  void undo() { Base::setVal(old); }
};

typedef UndoMixin<Int> UndoableInt; // Define mixed-in type

int main() {
  UndoableInt u;
  u.setVal(1); u.setVal(2);
  u.undo();
  cout << u.getVal(); // output: 1
}
Jam: Java with Mixins
D. Ancona, G. Lagorio, and E. Zucca

• Drawback of the C++’s mixin idiom:
  • The mixin class is not a type
  • `Undo* p = new UndoableNumber(); // Compiler error!`

• Jam – An extension of Java with mixins

```java
mixin Undo {
  inherited void setValue(int v);
  inherited int getValue();

  int old;
  void setValue(int v) {
    old = getValue(); super.setValue(v); }
  void undo() { setValue(old); }
}
```
Jam – Mixin Instantiation

class Number {
    int n;
    void setValue(int v) { n = v; }
    int getValue() { return n; }
}

// mixin instantiation, creating a new class:
class UndoableNumber = Undo extends Number{};

• UndoableNumber “extends” Undo, which extends Number
• Mixins are now types:

void g(Undo u) {
    u.setValue(1);
    u.setValue(2);
    System.out.println(u.old);
    System.out.println(u.getValue());
}
Mixins: Drawbacks

- Impose total ordering:

```java
class A {
    public int f() { return 0; }
    public int g() { return 1; }
}
mixin B {
    public int f() { return 2; }
    public int g() { return 3; }
}
class C = B extends A {
    • Can C offer B.f() and A.g() ?!
}
```

- Fragile inheritance:
  - Adding a method to a mixin may silently override an inherited method
Traits: Flattening over Linearization

- Trait: A composable unit of behaviour
- Serves as a type
- No fields
- Provides some methods (with behaviour)
- Requires other methods (abstract)
- When composing traits, if a method has more than one implementation it becomes abstract
- Similar to interfaces that allow methods implementation
trait T1 {
  abstract void add(int v);
  void inc() { add(1); }
}

trait T2 {
  abstract int getValue();
  abstract void setValue(int v);
  void add(int v) { setValue(getValue() + v); }
}

class Int uses T1, T2 {
  int n;
  int getValue() { return n; }
  void setValue(int v) { n = v; }
}

T1 t1 = new Int(); // A trait is also type
  t1.add(3);
Conflict Resolution

trait T1 {
    void add(int v) { while(--v >= 0) inc(); }
    void inc() { add(1); }
}

trait T2 {
    abstract int getValue();
    abstract void setValue(int v);
    void add(int v) { setValue(getValue() + v); }
}

class Int uses T1, T2 { // Int can also extend a “normal” superclass
    int n;
    int getValue() { return n; }
    void setValue(int v) { n = v; }
    void add(int v) { T2.this.add(); } // Resolve the conflict
    // Otherwise, a compiler error
    // when compiling Int
}
Traits in Squeak

Trait named: #T1 uses: {} category: 'TraitExample'!
  inc
    self add: 1

  add: v | x |
  x := v.
  [x > 0] whileTrue: [ n := n + 1. x := x - 1. ]

Trait named: #T2 uses: {} category: 'TraitExample'!
  add: val
    self setValue: ((self getValue) + val)

Object subclass: #Int
  uses: T1 - {#add:} + T2
  instanceVariableNames: 'n'

  getValue
    ^n

  setValue: val
    n := val