Object Thinking

Francesco Nidito

Programmazione Avanzata AA 2007/08
Outline

1. Introduction
2. Philosophy
3. Terms
4. Techniques
5. Conclusions

Reference: David West, “Object Thinking”, Chapters 1-5, 9
Real life example

Michael Hilzkit tells this story about the Apple’s famous visit to the Xerox PARC:

*Given this rare psychic encouragement, The Learning Research Group warmed to their subject. They even indulged in some of their favorite legerdemain. At one point Jobs, watching some text scroll up the screen line by line in the its normal fashion, remarked, “It would be nice if it moved smoothly, pixel by pixel, like paper”.*

With Ingalls at the keyboard, that was like asking a New Orleans jazz band to play “Limehouse Blues”. He clicked the mouse on the window displaying several lines of SmallTalk code, made minor edit, and returned to the text, *Presto! The scrolling was now continuous.*

*The Apple’s engineer’s eyes bulged in astonishment*
Why does OO matter?

- Object Orientation is a *natural* way to express concepts
- Easy construction of a domain *abstraction*
Why does OO matter?

- Object Orientation is a *natural* way to express concepts
- Easy construction of a domain *abstraction*
- Object Oriented languages *increment* productivity
Why does OO matter?

- Object Orientation is a *natural* way to express concepts
- Easy construction of a domain *abstraction*
- Object Oriented languages *increment* productivity
- It is important to learn to *think like objects*
Caveat

- Object Oriented programming is not only programming with objects
- We can write non-OO programs with Java, C++, C#
- In Object Oriented programming we must think the domain as:
  - A group of objects
  - Relations between objects
  - Objects using other objects
Destroy the cathedral...

Cathedral = Old way to think objects

Customer

ID
DateOfBirth
Gender
FirstName
...

GetID()
SetID()
...

SetID
GetID
SetGender
GetGender
ID

FirstName
DateOfBirth
Gender
...let’s build a bazaar!

Bazaar = New way to think objects

Customer

ID self
Present self
Indicate desires
Make decisions
Object thinking = Think like an object

- Traditional programmers think like *computers*
- OO programmers *must* learn to think like objects
- Thinking like an object is:
  - The object space is a community of *virtual persons*
  - We must concentrate on the *problem spaced* rather than the *solution space*
Virtual persons

- Objects know *their* resources
- Objects *ask* to other objects when something is needed
  - Objects do *not* know the internals of other objects
  - Objects *collaborate*, they do *not* use each other
Problem = Solution

- We must decompose a *problem* into a set of *objects*
- The *solution* is in the *interaction* of objects
- If the objects act as in the problem space *this* is the solution
Problem = Solution

- We must decompose a **problem** into a set of **objects**
- The **solution** is in the **interaction** of objects
- If the objects act as in the problem space **this** is the solution
- The objects **simulate** the problem to solve it
Four golden rules

- Everything is an *object*
- *Simulation* of the problem domain drives to object discovery and definition
- Objects must be *composition* enabled
- *Distributed* cooperation and communication must replace hierarchical centralized control as an organization paradigm
Traditional application (Example)
Object Oriented application (Example)
Is it simple?

The process of being an object thinker is not easy. You must start to think like an object and continue to learn day by day. The code, and the style, will be better with time.
Is it simple?

NO!
Is it simple?

NO!

- The process of being an *object thinker* is not easy
- You *must* start to *think like an object* and continue to learn day by day
- The *code*, and the *style*, will be better with time
Terms to deal with

First of all we must define the basic *terms*
Terms to deal with

- First of all we must define the basic terms
  - Class
  - Object
  - Responsibility
  - Message and method
Class

- *Classes* are the fundamental units of understanding
- We define the world in terms of *objects* associated to some *class*
- *Classes* define *attributes* and *methods* of the *objects* of its kind
class Integer{
    int val;

    void SetValue(int x){ val := x; }
    int GetValue(){ return val; }
    Integer +(Integer o){
        Integer i := new Integer();
        i.SetValue(o.GetValue()+val);
        return i;
    }
}

An *object* is an instance of a class.

An *object* can be uniquely identified by its *name*.

An *object* defines a *state* which is represented by the values of its attributes at a particular *time*.
An object is an instance of a class.

An object can be uniquely identified by its name.

An object defines a state which is represented by the values of its attributes at a particular time.

The only way to create an application must be to compose objects.
Object (Example)

class Integer{
    int val;

    void SetValue(int x){ ... }
    int GetValue(){ ... }
    Integer +(Integer o){ ... }
}

Integer i := new Integer(); //Object
Integer j := new Integer(); //Object

i.SetValue(2);
j.SetValue(3);

Integer k := i + j; // i.+(j)
Responsibility

- A *responsibility* is a service that an object can provide.
- If we define the world in terms of objects then...
Responsibility

- A *responsibility* is a service that an object can provide
- If we define the world in terms of objects then
  - An object is everything capable to provide a *limited* set of services
Responsibility

- A *responsibility* is a service that an object can provide.
- If we define the world in terms of objects then:
  - An object is everything capable to provide a *limited* set of services.
  - The only way to create an application is to *compose* objects.
A responsibility is a service that an object can provide.

If we define the world in terms of objects then

- An object is everything capable to provide a limited set of services.
- The only way to create an application is to compose objects.

The responsibility of an object is known also as the interface that the object implements.
An *Integer* object does **only** what it is intended to do:
- We can set or get its value
- We can perform some math operations on it
A message is a *request* to an object to *invoke* one of its methods. A message therefore contains:
A *message* is a *request* to an object to *invoke* one of its methods. A message therefore contains:
- The name of the method and
A message is a request to an object to invoke one of its methods. A message therefore contains:

- The name of the method and
- The arguments of the method.
Messages and methods

A *message* is a *request* to an object to *invoke* one of its methods. A message therefore contains:
- The name of the method and
- The arguments of the method.

A *method* is associated with a class. An object invokes one of its class methods as a reaction to the message.
class Integer{
    int val;

    void SetValue(int x){ val := x; }
    ...
}

Integer i := new Integer();
i.SetValue(42);

- The last instruction must be interpreted as:
  - We send a message to \( i \), the message says: “please set your value to 42”
  - When the object \( i \) receives the message it performs the operation(s) in the body of method \( \text{SetValue} \) to change its status
After terms we need *techniques*
Object Oriented programming: how to do it

- After terms we need *techniques*
  - Relation between classes
  - Polymorphism
Object Oriented programming: how to do it

- After terms we need *techniques*
  - Relation between classes
  - Polymorphism
- Followings are general techniques, you must adapt them to tools that you use
We have different ways to relate classes:
We have different ways to relate classes:

- *is-a-kind-of* and *is-a*
We have different ways to relate classes:

- *is-a-kind-of* and *is-a*
- *is-part-of* and *has-a*
We have different ways to relate classes:

- *is-a-kind-of* and *is-a*
- *is-part-of* and *has-a*
- *uses-a* and *is-used-by*
Is-a-kind-of and is-a

- We say that *Foo* class is-a-kind-of *Bar* class if *Foo* has the same responsibilities of *Bar*
- We say that an object *f* of *Foo* is-a *b* of class *Bar* if *Foo* is-a-kind-of *Bar*
Is-a-kind-of and is-a

- We say that *Foo* class is-a-kind-of *Bar* class if *Foo* has the same responsibilities of *Bar*

- We say that an object *f* of *Foo* is-a *b* of class *Bar* if *Foo* is-a-kind-of *Bar*

- *is-a-kind-of* is a relation between classes

- *is-a* is a relation between objects
Is-a-kind-of and is-a

- We say that *Foo* class is-a-kind-of *Bar* class if *Foo* has the same responsibilities of *Bar*.
- We say that an object *f* of *Foo* is-a *b* of class *Bar* if *Foo* is-a-kind-of *Bar*.
- *is-a-kind-of* is a relation between classes.
- *is-a* is a relation between objects.
- *Inheritance* is the way in which a *is-a-kind-of* relation (and *is-a* relation too) is established.
Is-a-kind-of (Example)

class Integer{
    void SetValue(int x){ ... } 
}

class UnsignedInteger: Integer{
    void SetValue(int x){
        if(x >= 0){
            Integer::SetValue(x);
        }else{
            RaiseException();
        }
    }
}

...
Is-a (Example)

class VectorOfInteger{
    Integer v[N];

    void Add(Integer i){ ... }
}

VectorOfInteger v := new VectorOfInteger();

Integer i := new Integer();
i.SetValue(-3);
v.Add(i);

UnsignedInteger u := new UnsignedInteger();
u.SetValue(42);
v.Add(u); // an UnsignedInteger is-a Integer
Is-part-of and has-a

- We say that *Foo* class is-part-of *Bar* class if *Bar* has one, or more, attributes of type *Foo*
- Has-a is exactly the opposite of is-part-of relation
Is-part-of and has-a

- We say that *Foo* class is-part-of *Bar* class if *Bar* has one, or more, attributes of type *Foo*
- *Has-a* is exactly the opposite of *is-part-of* relation
- *Composition* is the way in which a *is-part-of* relation (and *has-a* relation too) is established.
Is-part-of and has-a (Example)
Uses-a and is-used-by

- We say that *Foo* class uses-a *Bar* class if *Foo* knows how to use an object of type *Bar* but *without* *Bar* is-part-of *Foo*.
- *Is-used-by* is exactly the opposite of *uses-a* relation.
Uses-a (Example)

class Output{
    ...
    void Print(Integer i){...
    ...
}

Output o = new Output();
Integer i = new Integer();
i.SetValue(42);
o.Print(i);

The Output class knows how to manage an Integer object but after the Print method execution there is no more trace of object i in object o.
When we send a message to an object, the object can *interpret* the message in *various ways*.
Polymorphism

- When we send a message to an object, the object can *interpret* the message in *various ways*
- As a consequence of this multiple classes can expose the same *interface*
Polymorphism

- When we send a message to an object, the object can interpret the message in various ways.
- As a consequence of this multiple classes can expose the same interface.
- The same message can yield many, different, responses.
Polymorphism

- When we send a message to an object, the object can interpret the message in various ways.
- As a consequence of this, multiple classes can expose the same interface.
- The same message can yield many, different, responses.
- The sender's interest moves from how a class performs some task to what a class can perform.
Polymorphism classification

- Universal
- Parametric
- Sub-type
- Overloading
- Ad-hoc
- Coercion
Sub-type polymorphism is given by inheritance and method overriding.
Sub-type polymorphism

- *Sub-type polymorphism* is given by inheritance and *method overriding*
- With *method overriding* we redefine in a subclass methods of the super class(es)
Sub-type polymorphism

- *Sub-type polymorphism* is given by inheritance and *method overriding*
- With *method overriding* we redefine in a subclass methods of the super class(es)
- With the use of *late binding* we are able to dispatch the message to the right receiver
Sub-type polymorphism

- *Sub-type polymorphism* is given by inheritance and *method overriding*
- With *method overriding* we redefine in a subclass methods of the super class(es)
- With the use of *late binding* we are able to dispatch the message to the right receiver
- We can use, as base classes, *abstract* classes
Abstract classes

- *Abstract classes* define a set of responsibilities
Abstract classes

- *Abstract classes* define a set of responsibilities
- *Abstract classes* implement only some of their responsibilities
  - Other responsibilities implementation is left to sub classes
Abstract classes

- Abstract classes define a set of responsibilities
- Abstract classes implement only some of their responsibilities
  - Other responsibilities implementation is left to sub classes
- In some languages (Java, C#...) classes that implement none of their responsibilities are called interfaces
Sub-type polymorphism (Example)
Without sub-type polymorphism (Example)

class Rule{
    type_t t;
    type_t GetType(){return t;}
    void SetType(type_t x){t := x;}
}

class Rule_1: Rule{
    Rule_1(){ Rule::SetType(RULE_1);}  
    bool MustDiscardWithRule1(Message m){ ... } 
}

class Rule_2: Rule{
    Rule_2(){ Rule::SetType(RULE_2);}  
    bool MustDiscardWithRule2(Message m){ ... } 
}
Without sub-type polymorphism (Example 2)

class FireWall{
    Rule rule[N];
    int rulesnumber;
    ...
    void AppendRule(Rule r){ ... }
    bool MustDiscard(Message m){
        bool res = false; int i = 0;
        while(i < rulesnumber){
            if(rule[i].GetType() = RULE_1){
                res = ((Rule_1)rule[i]).MustDiscardWithRule1(m);
                if(res = true){break;}
            }elsif(...){...}
            i++;
        }
        return res;
    }
}
With sub-type polymorphism (Example)

class Rule{
    abstract bool MustDiscard(Message m);
}

class Rule_1: Rule{
    bool MustDiscard(Message m){ ... }
}

class Rule_2: Rule{
    bool MustDiscard(Message m){ ... }
}
class FireWall{
    Rule rule[N];
    int rulesnumber;
    ...
    void AppendRule(Rule r){ ... }
    bool MustDiscard(Message m){
        bool res = false; int i = 0;
        while(i < rulesnumber){
            res = rule[i].MustDiscard(m);
            if(res = true){
                break;
            }
            i++;
        }
        return res;
    }
}
With *parametric polymorphism* we are able to write generic code.
Parametric polymorphism

- With *parametric polymorphism* we are able to write generic code
- *Generic coding* permits to write one piece of code for multiple types
Parametric polymorphism

- With *parametric polymorphism* we are able to write generic code.
- *Generic coding* permits to write one piece of code for multiple types.
- C++ templates are an example of parametric polymorphism.
class Buffer of T{
    T v[N];
    ...
    void SetAt(int idx, T val){
        if(idx < N && i > 0){
            v[idx] := val;
        }else{
            RaiseException();
        }
    }
}

Buffer<Integer> bi = new Buffer<Integer>()
Integer i = new Integer();
i.setValue(42);
bi.SetAt(5, i);
Parametric polymorphism (Example 2)

class Buffer of T{
    T v[N];
    ...
    T GetAt(int idx){
        if(idx < N && i > 0){
            return v[idx];
        }else{
            RaiseException();
        }
    }
}

Buffer<Integer> bi = new Buffer<Integer>();
...
Integer i = bi.GetAt(5);
With *overloading polymorphism* we are able to write multiple versions of the same method with different signature.
With *overloading polymorphism* we are able to write multiple versions of the same method with different signature.

The compiler dispatch the message to the right method using the type information.
class Output{
    void Print(int i){...}
    void Print(string s){...}
    void Print(real r){...}
}

...

Output o = new Output();
o.Print(42); // 42 is int
o.Print('foo'); // 'foo' is string
o.Print(5.0); // 5.0 is real
With *coercion polymorphism* we are able to perform automatic type conversions.
Coercion polymorphism

- With *coercion polymorphism* we are able to perform automatic type conversions.
- We are able to do a kind of *overloading polymorphism* in an implicit way.
Coercion polymorphism

- With *coercion polymorphism* we are able to perform automatic type conversions
- We are able to do a kind of *overloading polymorphism* in a implicit way
- In C++ we can use operator overloading with cast operators to perform automatic cast
Coercion polymorphism (Example)

class Real{
  real val;

  void SetValue(real r){val := r}
}

Real r = new Real();
r.SetValue(5.0);
r.SetValue(42);//42 -> 42.0

The effect is similar to the one in which we define
SetValue(int) in the Real class.
Conclusions

- This slides are not enough
- The only important thing is to begin to think like objects
- The only way to learn *object thinking* is to practice and to apply the concepts seen in this lecture
Conclusions

- This slides are not enough
- The only important thing is to begin to think like objects
- The only way to learn object thinking is to practice and to apply the concepts seen in this lecture
- Good work!