Object Orientation
Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the systems under design.
Classification [7]
Object-Based

implies the capability of data abstraction

---

Objects are unique instances, having only in common the need for memory.

- Similar to an abstract data type (ADT)
- Methods define the object's external interface
- Attributes represent instance variables
- Methods provide access to attributes

Object-Based
Class-Based

Depending on the programming language employed, a class may be an ADT.

- Indicating further data abstractions
- Describing a uniform management
- A meta-description specifies the interface
- Classes may be compared to types
- Properties are attributes of a class
- A class defines a set of common properties
- Objects are instances of classes

Class-Based
Inheritance allows for the specialization of the inherited class(es).

- The object will be type compatible to more than one class – according to the class hierarchy, a single object relates to several classes.

- Objects instantiated from classes composed in that way are polymorphous.

- A new class adds properties and/or redefines inherited properties.
- Properties of the existing class(es) are inherited to the new class.
- New classes are constructed by the reuse of (an) existing class(es).

Implies the composition of abstract data types on the basis of inheritance.
Object orientation is unattainable using imperative programming languages.

- It allows the modeling of objects as a polymorphic entity.
- It is called object-oriented only if class hierarchies can be built by inheritance.
- Such a language is referred to as an object-based programming language.
- It must enable the programmers to describe common properties of objects.

In an object-oriented programming language must provide linguistic support.

Object Orientation = Objects + Classes + Inheritance
Synonyms of the "Theory of Hierarchy"
Distinctness of Heredity

Inheritance appears in various shapes and is of different consequences:

- Single inheritance
- Multiple inheritance
- Multiple inclusion
- Sharing

Any of these kinds serves the construction of a class hierarchy.
Single Inheritance

- a class is derivable from only one base class
- the class hierarchy is narrow but may be deep
- method redefinition/overloading is simple
- only a single set of attributes to inherit
- class reuse implies composition, method redefinition, and kind of delegation
- the class hierarchy is narrow but may be deep 

- a derived class may serve as a base class
- brings about fairly efficient implementations at the expense of reusability
- only the very base class appears to be reusable without any add-to
- only a single set of attributes to inherit
- method redefinition/overloading is simple
- only the very base class appears to be reusable without any add-to
Multiple inheritance brings about better reusability at the expense of an efficient implementation. A concept for implementation unification, the class hierarchy may be deep and wide. Classes can be inherited several times method redefinition/overloading is crucial. Many sets of attributes to inherit a class is derivable from many base classes.

Multiple Inheritance
Multiple Inheritance

- The final object contains copies of object fragments of the same class.
- The attributes of classes inherited several times are included several times.

Multiple Inclusion

- Operating System Engineering — Object Orientation
the attributes of classes inherited several times are included once only

- the final object contains copies of pointers to the shared object fragment(s)
A method applied to an object is implicitly supplied with the object’s address.

- In C++, the pointer to the instance of a class (i.e., object) the invoked method is actually applied to.

  In C++, the pointer to the instance of a class (i.e., object) the invoked method is actually applied to.

  - A method applied to an object is implicitly supplied with the object’s address.

  - From base to derived class: subtracting some delta.

  - From derived to base class: adding some delta.

Multiple inheritance may entail pointer manipulation upon method invocation.

- Taking multiple inheritance branches into account, this becomes variable.

- Taking multiple inheritance branches into account, this is identical for all the classes within the single inheritance path.

- Operating-System Engineering — Object Orientation

  - p. 16

Object Layout
Polymorphism

from Foobar to Bar::Fop: this = sizeof(Foo)  
from Foobar to Foo::Fop: this remains unchanged

Object Layout

Operating-System Engineering — Object Orientation
Bar is a multiple inheritance branch: (cond.) adjustment

Foo is on the single inheritance path: this pass through

The adjustment

Object Layout
LateBinding

Late Binding

also called dynamic binding—but must not be mixed up with dynamic loading

1. the object’s methods are bound at the point in time of object instantiation — the object’s methods are bound at the point in time of object instantiation

2. the object’s methods are bound at the point in time of object instantiation — the object’s methods are bound at the point in time of object instantiation

3. and they must be redefined by the derived class(es)

1. they must have been defined in (the external interface of) a base class, three preconditions must hold:

2. the method’s base class(es) must be inherited by some derived class(es),

3. the method must be redefined by the derived class(es)

form methods to be capable of late binding, three preconditions must hold:
class Foo

public:
  virtual int foo() = 0;

class Bar

public:
  virtual int bar() = 0;

class Foobar: public Foo, public Bar

int foo()
{
  return 4711;
}

int bar()
{
  return 42;
}

text_foobar (Foo* fp, Bar* bp)
{
  return fp->foo() + bp->bar();
}

Foobar fb;
Late Binding

C++ ← x86

Operating-System Engineering — Object Orientation

GCC -O6 -S -fno-rtti -fno-exceptions -fomit-frame-pointer

The compiler implements the thunk model

- gives priority to the single inheritance path
- handicaps multiple inheritance branches
- treats all virtual methods identical
- the compiler implements the common model
that a method is subjected to late binding is transparent to the method itself.

the generated code is the same for the common model and the thunk model.

Method Redefinition

Late Binding
every class containing a virtual function has a virtual function table of "call descriptors" – a triple of offset, ? and function pointer

- no matter which path/branch will be taken – the caller adjusts the object pointer (this)

Common Model (1)
Late Binding Common Model (2)

overhead at the caller's site:

```c
int foobar (Foo* fp, Bar* bp)
{...
  return fp->foo() + bp->bar();
}
```

---

- Avoidable overhead
- An adjustment by 0
- Single inheritance path:

```assembly
; \text{void} \text{foobar}(\text{Foo}^*, \text{Bar}^*)$
  \text{ret}$
  \text{addl} \#20, \%esp$
  \text{pop} \%esi$
  \text{pop} \%ebx$
  \text{addl} \%esi, \%ebx$
  \text{addl} \%edi, \%edi$
  \text{mov} \%ebx, \%edi$
  \text{call} *\%edi$
  \text{mov} \%eax, \%esi$
  \text{addl} \#-12, \%esp$
  \text{mov} \%ecx, \%ecx$
  \text{movswl} 8(\%ecx), \%eax$
  \text{addl} \%eax, \%eax$
  \text{pushl} \%eax$
  \text{movl} 12(\%ecx), \%eax$
  \text{call} *\%eax$
  \text{addl} \%esi, \%eax$
  \text{addl} \#32, \%esp$
  \text{popl} \%ebx$
  \text{popl} \%esi$
  \text{addl} \#20, \%esp$
  \text{ret}$
```

---

Indirect function call

- Adjustment of this
- Overhead at the caller's site:

Common Model (2)
Late Binding (1)

- Every class containing a virtual function has a virtual-function table
- Thunks relate to multiple inheritance branches
- After that, it jumps to the redefined method
- The thunk adjusts the object pointer (this)
- Every virtual method in it has such a thunk

```
class FooBar : public Foo, public Bar
{
    int foo();
    int bar();
};

.jmp -4,4(%esp)
.addl $-4,4(%esp)

-thunk-4-bar--FooBar:
    .long 0
    .long -4
    .long __thunk_4_bar__6Foobar
    .byte 4

-thunk-4-bar__6Foobar:
    addl $-4,4(%esp)
    jmp bar__6FooBar

-thunk-6-FooBar:
    .long 0
    .long 0
    .long __vt_FooBar
```

```
---

-thunk-6-FooBar:
    .long 0
    .long 0
    .long __vt_FooBar
```
Thunk Model (2)

Late Binding

indirect function call

adjustment of this where really required

- location of the redefined method
- location of the virtual-function table pointer
- overhead at the caller's site:

```c
{ fooBar (Foo* fp, Bar* bp) {
    int foobar (Foo* fp, Bar* bp) {
        return fp->foo() + bp->bar();
    }
    .............
    int foobar (Foo* fp, Bar* bp) {
        return fp->foo() + bp->bar();
    }
    .............
```
Inheritance Ambiguity

- redefinition of a multiple inherited method
- unproblematic using single inheritance
- problematic using multiple inheritance
- the subclass needs to rename the methods
- i.e., associate them with unique names
- linguistic support would be nice to have
- * as provides Eiffel, but not C++
- the conflict can't be resolved automatically
Inheritance differs from Subtyping

- a supertype defines the fundamental properties of an entity
- a subtype serves the refinement of these properties
- supertype operations must be redefined by the subtype to be visible
- as a consequence, identical operations at both type levels lead to redundancy

at first sight, a subclass is very similar to a super type
- but superclass methods are not obliged to be redefined by the subclass
- superclass properties are "passed through" to subclasses and beyond
- this corresponds to hierarchical structuring of incremental machine design
Varieties of Inheritance

Implementation inheritance – i.e., class inheritance

- Impliecs late binding, thus provides overridable, invasive, high flexibility/dynamics
- Methods need to be redefined in subclasses to enable object instantiation
- The class hierarchy corresponds to the open/closed principle
- The implementation becomes more likely to depend on base class details
- Is subjected to the risk of making derived classes more fragile
- Is also known as subclassing and considered promoting software reuse

Interface inheritance

- Can be made "stable" by designing base classes to be "semi-abstract"
- The class hierarchy corresponds to the open/closed principle

Implementation Inheritance i.e. Class Inheritance

Interfaces of Inheritance
Open/Closed Principle

The base classes are open to the derived classes but closed to the clients.

- The “closed classes” together can be considered an ADT.

- The typical case of implementation inheritance.

- The inheritance of heredity is all access to attributes unrestricted.

- By specifically providing access functions to instance variables.

- E.g. Instance variables may be read/written by any subclass.

- From outside is all attribute access restricted, it must be enabled explicitly.

The closed classes” together can be considered an ADT.

- E.g. By specifically providing access functions to instance variables.

- From outside is all attribute access restricted, it must be enabled explicitly.

- The “closed classes” together can be considered an ADT.

- The typical case of implementation inheritance.

- The inheritance of heredity is all access to attributes unrestricted.

- By specifically providing access functions to instance variables.

- E.g. Instance variables may be read/written by any subclass.

- From outside is all attribute access restricted, it must be enabled explicitly.

- The “closed classes” together can be considered an ADT.

- The typical case of implementation inheritance.

- The inheritance of heredity is all access to attributes unrestricted.

- By specifically providing access functions to instance variables.

- E.g. Instance variables may be read/written by any subclass.

- From outside is all attribute access restricted, it must be enabled explicitly.
Semi-Abstract Classes

Attributes will be closed to any kind of public, except to „their“ methods.

- Changes made at base-class level will not impair derived classes.
- Attribute access happens only through access functions (i.e., access methods).
- Clients are aware of the internal (data) structure of a class.
- Class attributes are directly visible but not directly accessible.

Proximity to an abstract data type (ADT) is given.

Method call optimization is left to the compiler, e.g. by inlining the method implementation. The proximity of client programming to an abstract class is thereby left to the decision of the developer. Changes made at base-class level will not impair derived classes.

Interrelationships between base class and derived class(es) is defined by a functional hierarchy.
class Fop {
public:
    Fop() : _fop(0) {}
    int fop() const { return _fop; }
};

class Foo : public Fop {
public:
    Foo(int i) : Fop(), _foo(i) {}
    int foo() const { return fop() + _foo; }
};

class Bar : public Fop {
public:
    Bar(int i) : Fop(), _bar(i) {}
    int bar() const { return fop() - _bar; }
};

class Foobar : public Foo, public Bar {
public:
    Foobar(int f, int b) : Foo(f), Bar(b) {}
int foobar() {
    return Foo(4711, 42).foo() + Bar(4711, 42).bar();
}
};

int foobar() {
    return Foobar(4711, 42).foobar();
}
```
int foobar()
{
    return FooBar(4711, 42).foobar();
}
```

```
semi-abstract classes
```

```
// C++ 2.91

x86
```

```
Operating System Engineering — Object Orientation
```

```
ret
addl $16, %esp
addl %edx, %eax
addl $42, %eax
movl (%esp), %eax
movl 8(%esp), %eax
movl 4(%esp), %eax
movl (%eax, %edx, 0), %eax
movl (%eax, 4(%esp), 0), %eax
movl (%eax, 8(%esp), 0), %eax
movl 4(%eax, %edx, 0), %eax
movl (%eax, 4(%esp), 0), %eax
subl $16, %esp

```

Polymorphism

...at type level

Objects of derived classes are type compatible to the base class(es).

...at function level

A subclass object is kind of a fragment of a superclass object.

The reverse is not true, i.e., superclass objects are no subclass objects.

That is to say, subclass objects are also superclass objects.

Late binding must be explicitly enabled by specifying a method to be virtual.

Not every method must be necessarily subjected to late binding, as e.g. in C++.

The final redefinition (i.e., specialization) becomes effective – the final redefinition subject to late binding.

Subclasses methods redefine superclass methods subjected to late binding.

Methods of base classes are applicable to objects of the derived classes.
Inheritance is inclined to break encapsulation \([5]\) because inheritance is defined only at compile-time. A further disadvantage of implementation inheritance may be that one can’t change the implementations inherited from base classes at runtime, because implementation inheritance is defined only at compile-time.

A further disadvantage of implementation inheritance over implementation inheritance is to favor object composition over implementation inheritance. That is to say, to inherit only from (semi-)abstract classes – a way out of this dilemma is to consequently employ interface inheritance. Changes made in a base class may affect the derived class – derived classes may become bound up with base class implementations – base classes define at least part of their derived classes’ ‘physics’ base class dependency may limit flexibility and, ultimately, reusability.

Afurther disadvantage of implementation inheritance over implementation inheritance is to favor object composition over implementation inheritance. That is to say, to inherit only from (semi-)abstract classes – a way out of this dilemma is to consequently employ interface inheritance. Changes made in a base class may affect the derived class – derived classes may become bound up with base class implementations – base classes define at least part of their derived classes’ ‘physics’ base class dependency may limit flexibility and, ultimately, reusability.
Object Composition

Object Composition is defined dynamically at runtime through object interfaces. The system behavior depends on object inter-relationships, not on classes. Any object can be replaced by another object of the same type. Object Composition requires objects to respect each other's interfaces. Operating System Design is that classes and class hierarchies remain small. At the expense of a larger number of objects, they will be less likely to grow into “unmanageable monsters.” An effect on system design is that classes and class hierarchies remain small.

- Objects are accessed solely through their interfaces, ensuring encapsulation.
- Composition requires objects to respect each other's interfaces.
- Any object can be replaced by another object of the same type.
- Object Composition is defined dynamically at runtime through object interfaces.
- The system behavior depends on object inter-relationships, not on classes.
Delegation

Delegation is a way of making composition as powerful for reuse as inheritance. Two objects are involved: a receiving object forwards requests to its delegate; e.g., the this member variable. With delegation, the same effect is achieved by having the receiver in charge of passing itself to the delegate.

With inheritance, an inherited operation implicitly refers to the receiving object through e.g., the this member variable. With delegation, an operation explicitly refers to the receiver.

Behavior can be composed at runtime, just as to make changes.

Inheritance works best when used in highly stylized ways—i.e., in standard patterns.

Software that is not easy to understand and prone to runtime inefficiencies—dynamic and highly parameterized software is the outcome.

Analogous to derived classes defining requests to base classes; two objects are involved: a receiving object forwards requests to its delegate.
Reusing Functionality in Object-Oriented Systems

white-box reuse i.e., reuse by subclassing
- defines the implementation of one class in terms of (an)other class(es)
  - implementation or class inheritance
- refers to visibility, i.e. base class internals are visible to derived classes

black-box reuse i.e., reuse by composition
- new functionality is obtained by assembling objects to a more complex one
  - requires well-defined object interfaces
- no internal details of objects are visible to the outside
Design Patterns

patterns capture design experience in a form that people can use effectively.

consequences are the results and trade-offs of applying the pattern.

solution describes the elements that make up the design, their responsibilities, and collaborations.

text context.

problem describes when to apply the pattern, explains the problem and its context.

name a handle, describes a design problem, its solutions, and consequences in a word or two.

description of an „important and recurring design in object-oriented systems‟ [3]
Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [1]
1. Program to an interface, not an implementation
2. Favor object composition over class inheritance

Object-oriented systems programming largely depends on compiler quality

- abstraction from implementation is what remains as a must
- variance at runtime is not always what needs to be provided
- late binding abstracts from implementation and from variance

Interface inheritance should not only be put on a level with abstract classes

Summary


