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.
Object-Based

- implies the capability of *data abstraction*
  - methods provide access to attributes
  - attributes represent instance variables
- methods define the object’s external interface
  - similar to an abstract data type (ADT)
- objects are unique instances, having only in common the need for memory
Class-Based

- objects are instances of classes
  - a class defines a set of common properties
  - properties are attributes of a class
  - classes may be compared to types

- a meta-description specifies the interface
  - describing a uniform management
  - indicating further data abstractions

- depending on the programming language employed, a class may be an ADT

Object-Oriented

- implies the composition of abstract data types on the basis of inheritance
  - new classes are constructed by the reuse of (an) existing class(es)
  - properties of the existing class(es) are inherited to the new class
  - a new class adds properties and/or redefine inherited properties

- objects instantiated from classes composed in that way are polymorphous
  - according to the class hierarchy, a single object relates to several classes
  - the object will be type compatible to more than one class

- inheritance allows for the specialization of the inherited class(es)
Object Orientation = Objects + Classes + Inheritance

- an object-oriented programming language must provide linguistic support
  - it must enable the programmers to describe common properties of objects
    * such a language is referred to as an object-based programming language
  - it is called object-oriented only if class hierarchies can be built by inheritance
    * so that the objects can be instances of classes that have been inherited
  - it allows the modeling of an object as a polymorphous entity

- object orientation is unattainable using imperative programming languages

\[1\text{Notwithstanding, from time to time there are comments stating the development of object-oriented systems using e.g. the C programming language. This is in contradiction to the classical definition of object orientation [7].}\]

Synonyms of the “Theory of Heredity”

| base class | ............ | derived class |
| superclass | ............ | subclass |
| parent class | ............ | child class |
| inheriting class | ............ | inherited class |
| common class | ............ | specialized class |
| “upper class” | ............ | “lower class” |
Distinctness of Heredity

• inheritance appears in various shapes and is of different consequences:
  – single inheritance ..................................................... 9
  – multiple inheritance .................................................. 10
    * multiple inclusion .................................................. 11
    * sharing ..................................................................... 12

• any of these kinds serves the construction of a class hierarchy

Single Inheritance

• a class is derivable from only one base class
  – only a single set of attributes to inherit
  – method redefinition/overloading is simple

• 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
  – class reuse implies composition, method redefinition, and kind of delegation
  – only the very base class appears to be reusable without any add-to
Multiple Inheritance

- a class is derivable from many base classes
  - many sets of attributes to inherit
  - method redefinition/overloading is crucial
  - classes can be inherited several times

- the class hierarchy may be deep and wide
  - a concept for implementation unification

- brings about better reusability at the expense of an efficient implementation
  - class reuse is still limited to the very base class, but there are many of which

Multiple Inheritance

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

- the attributes of classes inherited several times are included once only
- the final object contains copies of pointers to the shared object fragment(s)

Object Layout

- a method applied to an object is implicitly supplied with the object’s address
  - within the single inheritance path, \texttt{this}\(^2\) is identical for all the classes
  - taking multiple inheritance branches into account, \texttt{this} becomes variable

- multiple inheritance may entail pointer manipulation upon method invocation
  - from derived to base class: adding some delta
  - from base to derived class: subtracting some delta

\(^2\)In C++, the pointer to the instance of a class (i.e., object) the invoked method is actually applied to.
Object Layout

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

Operating-System Engineering — Object Orientation

Object Layout

• Foo is on the single inheritance path: this pass through

```cpp
class Foobar : public Foo, public Bar {
    int foo () {
        return Foo::foo() + 4711;
    }

    int bar () {
        return Bar::bar() + 42;
    }
};
```

• Bar is a multiple inheritance branch: (cond.) adjustment

```
foo__6Foobar:
pushl 4(%esp)
call foo__3Foo
addl $4711,%eax
addl $4,%esp
ret
```

```
bar__6Foobar:
movl 4(%esp),%eax
testl %eax,%eax
jne .L4
xorl %eax,%eax
jmp .L5
.p2align 4,,7
.L4:
addl $4,%eax
.L5:pu
pushl %eax
call bar__3Bar
addl $42,%eax
addl $4,%esp
ret
```
Late Binding

- the association at runtime of which method is going to be applied to an object
  - the object's methods are bound at the point in time of object instantiation

- for methods to be capable of late binding, three preconditions must hold:
  1. they must have been defined in (the external interface of) a base class,
  2. the method's base class(es) must be inherited by some derived class(es),
  3. and they must be redefined by the derived class(es)

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

---

Late Binding C++ Case Study

class Foo {
public:
  virtual int foo () = 0;
};

class Bar {
public:
  virtual int bar () = 0;
};

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

class Foobar : public Foo, public Bar {
  int foo () {
    return 4711;
  }
  int bar () {
    return 42;
  }
};

 Foobar fb;
Late Binding

- the compiler implements the *common model*
  - treats all virtual methods identical

```c
Foobar fb;
```

- the compiler implements the *thunk model*
  - gives priority to the single inheritance path
  - handicaps multiple inheritance branches

```
gcc -O6 -S -fno-rtti -fno-exceptions -fomit-frame-pointer
```

Late Binding

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

```c
int Foobar::foo () {
    return 4711;
}
```

```c
foo__6Foobar:
    movl $4711,%eax
    ret
```

- that a method is subjected to late binding is transparent to the method itself
Late Binding Common Model (2)

- **overhead** at the caller’s site:
  - adjustment of `this`
- **indirect** function call

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

- single inheritance path:
  - an adjustment by 0
  - avoidable overhead

```
foobar__FP3FooP3Bar:
  subl $20,%esp
  pushl %esi
  pushl %ebx
  movl 32(%esp),%edx
  movl 36(%esp),%ebx
  addl $-12,%esp
  movl (%edx),%ecx
  movsl 8(%ecx),%eax
  addl %eax,%edx
  pushl %edx
  movl 12(%ecx),%eax
  call *%eax
  movl %eax,%esi
  addl $-12,%esp
  movl (%ebx),%edx
  movsl 8(%edx),%eax
  addl %eax,%ebx
  pushl %ebx
  movl 12(%edx),%eax
  call *%eax
  addl %esi,%eax
  addl $32,%esp
  popl %ebx
  popl %esi
  addl $20,%esp
  ret
```

- every class containing a virtual function has a virtual-function table of "call descriptors"
- a triple of offset, ?, and function pointer
- the caller adjusts the object pointer (this)
- no matter which path/branch will be taken
Late Binding

- every class containing a virtual function has a virtual-function table
  - of function pointers

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

- thunks relate to multiple inheritance branches
  - every virtual method in it has such a thunk
  - the thunk adjusts the object pointer (this)
  - after that, it jumps to the redefined method

```
__vt_6Foobar:
  .long 0
  .long 0
  .long foo__6Foobar

__vt_6Foobar.3Bar:
  .long -4
  .long 0
  .long __thunk_4_bar__6Foobar

__thunk_4_bar__6Foobar:
  addl $-4,4(%esp)
  jmp bar__6Foobar
```

Late Binding

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

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

- adjustment of this where really required
- indirect function call

```
foobar__FP3FooP3Bar:
  pushl %esi
  pushl %ebx
  movl 12(%esp),%eax
  movl 16(%esp),%ebx
  movl (%eax),%edx
  pushl %eax
  movl 8(%edx),%eax
  call *%eax
  movl %eax, %esi
  movl (%ebx),%eax
  pushl %ebx
  movl 8(%eax),%eax
  call *%eax
  addl %esi,%eax
  addl $8,%esp
  popl %ebx
  popl %esi
  ret
```

Thunk Model (1)

Thunk Model (2)
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

**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

**inheritance**
- at first sight, a {super,sub}-class is very similar to a {super,sub}-type
  - but superclass methods are not obliged to be redefined by the subclass
- superclass properties are “passed through” to subclasses and beyound
- this corresponds to *hierarchical structuring* of incremental machine design
Varieties of Inheritance

**implementation inheritance** i.e. **class inheritance**

- is also known as **subclassing** and considered promoting **software reuse**
- is subjected to the risk of making derived classes more **fragile** [6]
  - the implementation becomes more likely to depend on base class details
  - the class hierarchy corresponds to the **open/close principle** [4]
- can be made “stable” by designing base classes to be “semi-abstract”

**interface inheritance**

- corresponds to **subtyping** if a superclass was designed as **abstract base class**
  - methods need to be redefined in subclasses to enable object instantiation
- implies **late binding**, thus provides overhead-prone high flexibility/dynamics

---

Open/Close Principle

- **inside** the hierarchy of heredity is all access to attributes unrestricted
  - e.g. instance variables may be read/written by any subclass
  - the typical case of **implementation inheritance**

- **from outside** is all attribute access restricted, it must be enabled explicitly
  - e.g. by specifically providing access functions to instance variables
  - the “closed classes” together can be considered an ADT

- base classes are open to the derived classes but closed to the clients
“Semi-Abstract Classes”

- class attributes are directly visible but not directly accessible
  - clients are aware of the internal (data) structure of a class
  - attribute access happens only through methods\(^4\) (i.e., access functions)
  - changes made at base-class level will not impair derived classes

- proximity to an abstract data type (ADT) is given
  - base classes have a stable interface, syntactically and semantically
  - a base-class implementation may be fragile without affecting derived classes

- attributes will be closed to any kind of public, except to “their” methods

\(^4\)Method call optimization is left to the compiler, e.g. by inlining of the method implementation. The interrelationship between base class and derived class(es) is defined by a functional hierarchy.

```
class Fop {
    int _fop;
public:
    Fop () { _fop = 0; }
    int fop () const { return _fop; }
};

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

class Bar : public Fop {
    int _bar;
public:
    Bar (int i) { _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() + bar(); }
};

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

fooobar__Fv:
  movl $-4669,%eax
  ret
```
Polymorphism

. . . at type level

- objects of derived classes are *type compatible* to the base class(es)
- that is to say, subclass objects are also superclass objects
  - the reverse is not true, i.e., superclass objects are no subclass objects
  - a superclass object is kind of a fragment of a subclass object

. . . at function level

- methods of base classes are *applicable to* objects of the derived classes
- subclass methods redefine superclass methods subjected to late binding\(^5\)
  - the final redefinition (i.e., specialization) becomes effective

\(^5\)Not every method must be necessarily subjected to late binding, as e.g. is the case of Eiffel. In contrast, in C++ late binding must be explicitly enabled by specifying a method to be *virtual.*
Inheritance is Inclined to Break Encapsulation [5]

• base class dependence may limit flexibility and, ultimately, reusability
  – base classes define at least part of their derived-classes’ “physics”
  – derived classes may become bound up with base class implementations
  – changes made in a base class may affect the derived class

• a way out of this dilemma is to consequently employ interface inheritance
  – that is to say, to inherit only from (semi-) abstract classes

• the alternative is to favor object composition over implementation inheritance

---

6 A further disadvantage of implementation inheritance may be that one can’t change the implementations inherited from bases classes at runtime, because inheritance is defined only at compile-time.

Object Composition

• object composition is defined dynamically at runtime through object interfaces
  – composition requires objects to respect each others’ interfaces
  – objects are accessed solely through their interfaces, ensuring encapsulation
  – any object can be replaced by another object of the same type

• an effect on system design is that classes and class hierarchies remain small
  – they will be less likely to grow into “unmanageable monsters”
  – at the expense of a larger number of objects

• the system behavior depends on object inter-relationships, not on classes
Delegation

- a way of making composition as powerful for reuse as inheritance
  - two objects are involved: a receiving object forwards requests to its delegate
  - analogous to derived classes deferring requests to base classes

- behaviors can be composed at runtime, just as to make changes
  - dynamic and highly parameterized software is the outcome
  - software that is not easy to understand and prone to runtime inefficiencies

- works best when used in highly stylized ways—i.e., in “standard patterns”

---

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

- description of an “important and recurring design in object-oriented systems” [3]
  
  **name** a *handle*, describes a design problem, its solutions, and consequences in a word or two.
  
  **problem** describes when to apply the pattern, explains the problem and its context.
  
  **solution** describes the elements that make up the design, their responsibilities, and collaborations.
  
  **consequences** are the results and trade-offs of applying the pattern.

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

Object-Oriented Design vs. Buildings and Towns

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 millions times over, without ever doing it the same way twice. [1]
Summary

- try to comply with the “principles of (reusable) object-oriented design” [3]:
  1. program to an interface, not an implementation
  2. favor object composition over class inheritance

- interface inheritance should not only be put on a level with abstract classes
  - late binding abstracts from implementation and from variance
  - variance at runtime is not always what needs to be provided
  - abstraction from implementation is what remains as a must

- object-oriented systems programming largely depends on compiler quality

Bibliography


