Object Orientation

Object-Oriented Design [2]

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
  - 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 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 polymorphic entity
- object orientation is unattainable using imperative programming languages\(^1\)

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 redefines inherited properties
- objects instantiated from classes composed in that way are polymorphic
  - 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)

Synonyms of the “Theory of Heredity”

<table>
<thead>
<tr>
<th>base class</th>
<th>derived class</th>
</tr>
</thead>
<tbody>
<tr>
<td>subclass</td>
<td>superclass</td>
</tr>
<tr>
<td>parent class</td>
<td>inheriting class</td>
</tr>
<tr>
<td>common class</td>
<td>inherited class</td>
</tr>
<tr>
<td>“upper class”</td>
<td>“lower class”</td>
</tr>
</tbody>
</table>

\(^1\)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 contrast to the classical definition of object orientation.
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

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

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

- 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, this is identical for all the classes
  - taking multiple inheritance branches into account, 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

---

Object Layout

- from FooBar to Foo::Fop ⇒ this remains unchanged
- from FooBar to Bar::Fop ⇒ this += sizeof(Foo)

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

---

this-Polymorphism

- Bar* Foo*

this-Adjustment

- pushl 4(%ebp)
- call Foo::foo
- addl $4714, %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 foo (Foo* fp, Bar* bp) {
  return fp->foo() + bp->bar();
};
```

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

  int bar () {
    return 42;
  }
};
```

```
foo_foo ();
```

Late Binding

C++ → x86

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

```
foo_foo ();
```

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

```
f:
  .zero 8
  movl %rt6FooBar, fb+4
  movl %rt6FooBar, fb
```

```
buffer fb:
```

Late Binding

Method Redefinition

```
int Foo::foo () {
  return 4711;
}
```

```
buffer_6FooBar:
  movl $4711, %eax
  ret
```

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

```
buffer_6Bar:
  movl $42, %eax
  ret
```

- the generated code is the same for the common model and the thunk model
- that a method is subjected to late binding is transparent to the method itself
- 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

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

```
// rt$@FooBar:
   .value 0
   .long 0
   .value 0
   .long 0
   .value 0
   .long 0
   .long bar__FooBar
```

```
// rt$@Bar:
   .long 0
   .long foo__FooBar
```

```
// rt$@Bar$@Bar:
   .value -4
   .value 0
   .long 0
   .value 0
   .long 0
   .long __thunk4_bar__FooBar
```

```
// thunk4_bar__FooBar:
   subi 4,-4(Csp)
   jmp bar__FooBar
```

- 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
**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

**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

**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 beyond
- this corresponds to hierarchical structuring of incremental machine design

**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\(^6\) (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

\(^6\) Method\(s\) 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.

---

### C++ \(\rightarrow\) x86

```cpp
class Pop {
  int _fop;
public:
  Pop (int _fop) { _fop = _fop; }
  int fop (int i) const { return _fop; }
};

class Foo : public Pop {
  int _foo;
public:
  Foo (int i, int b) : _foo (i), _barr (i) { }
  int fop (int i) const { return _barr[i] + foo[i]; }
};

class Bar : public Pop {
  int _bar;
public:
  Bar (int i, int b) : _barr (i), _bar (i) { }
  int fop (int i) const { return _barr[i] + _bar; }
};

int fobar (int i) {
  return fobar(4711, 42).foober[i];
}
```

### 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 subject to late binding\(^5\)
  - the final redefinition (i.e., specialization) becomes effective

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

- base class dependency 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

---

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”

---

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

---

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 relationships, 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