Data Abstraction and OO

Textbook: Chapter 9 (incl. CD material)

- Modules or Objects/Classes
- Reduces conceptual load by modularizing code
- Contains faults to small parts of code
- Makes program components more independent

Why OO?

- Groups data with code
  - Data is isolated in private object fields
  - Methods operating on the data are bundled with the data in classes
  - Other code can't mess with the data
  - Classes can be easily added or removed from the code
Programming Style

- **C++**
  
  ```
  class C { ... public: virtual int foo() { ... }}
  class D : public C { virtual int foo() { ... }}
  class E : public C { virtual int foo() { ... }}
  ```

- **C**
  
  ```
  int foo(C * obj) {
    if (obj is a D)
      ... // code from D::foo()
    else if (obj is an E)
      ... // code from E::foo()
    else
      ... // code from C::foo()
  }
  ```

OO vs. ADT/Functional Style

- **Object-oriented programming**
  - Extending a data structure is easy
  - Adding new code to an existing data structure is hard

- **ADT/Functional/Imperative Style**
  - Adding new code is easy
  - Extending a data structure is hard

Object-Oriented Programming

- **User-defined types**
  - Classes

- **Encapsulation**
  - Private fields

- **Subtype polymorphism**
  - Inheritance
  - Structural subtyping

- **Code reuse**
  - Inheritance
What is a Type?

- **Built-in types**: int, char, etc.
- **Class types**
- **Interface types (in Java)**

- Think of a type as a set of values
  - int = \(-2^{31} \ldots 2^{31}-1\)
  - \(C\) = set of instances of class \(C\) or any subclass of \(C\)

Subtyping

- \(C\) is a subtype of \(B\) (\(C <: B\))
- or a \(C\) object is \(\text{a}\) \(B\) object
- or a \(C\) object can be used wherever a \(B\) object is expected
- or \(C\) is more specific than \(B\)

- The set of \(C\) instances is a subset of the set of \(B\) instances

Implementation of Subtyping

- The layout of a \(C\) object includes a \(B\) object
- If (virtual) methods are overridden, we need to select the appropriate method at run time
Selection of Virtual Methods

```java
class B {
    public int foo() { return 0; }
}

class C extends B {
    public int foo() { return 1; }
}

B obj = new C();
int i = obj.foo(); /* executes C.foo */
```

Method Selection Algorithm

- Method call
  ```
  C * p = new D();
p->foo(42, p)
  ```
- Compile-time overload resolution
  - Find the receiver’s (p’s) static type (C)
  - Inside C find an appropriate method for the static types of the arguments ((int,C))
- Run-time virtual method dispatch
  - Look up the code for the method signature foo(int,C) in the virtual function table

Implementation of Method Dispatch

- Conceptually: an object contains a list of pointers to its methods
- C++: the object contains a pointer to the virtual function table
- Java: an object pointer consists of a pointer to the object and a pointer to the dispatch table
Other Design Issues

- **Single or multiple inheritance**
  - C++: class C : public A, public B {};
  - Java: class C extends B {}

- **C++-style virtual inheritance**
  - class C : public virtual A {};
  - allows objects to share a common part from multiple base classes

Other Design Issues

- **Single dispatch or multimethods**
  - single dispatch: virtual functions
  - multiple dispatch: run-time overloading

- **Function call or message passing**
  - C++/Java: o.foo()
  - Smalltalk: o foo

- **Typed or untyped**

- **Interface types vs. abstract classes**

Multiple Inheritance

```cpp
class A { int x; };
class B {
    int y;
    public:
        virtual int foo(int); 
};
class C : public A, public B {
    int z;
    public:
        virtual int foo(int); 
        virtual void bar(int); 
};
```
Implementation of Multiple Inheritance

- Concatenate all the pieces in layout
- Multiple vtables per object (C++)

```
x
  y
  B_vptr
  z
  C_vptr
  C::foo
  C::bar
```

Implementation of Multiple Inheritance

- Adjust ‘this’ pointer
  ```
  B * p = new C;
  int i = p->foo(42);
  ```
- is translated to
  ```
  int i = ((p->B_vptr)[1].fn)
  (p+(p->B_vptr)[1].delta, 42)
  ```

Virtual Inheritance

- Sharing of common part
  ```
  class A;
  class B : public virtual A;
  class C : public virtual A;
  class D : public B, public C;
  ```
- Implementation: B and C parts contain pointers to common A part
Multimethods

- Instead of run-time method selection based on receiver (virtual in C++)
- Run-time method selection based on all arguments
- Like overloading but with method selection at run time
- Allows different program structure
- Languages: CLOS, Cecil, Brew

Multimethod Implementation

- Generic functions dispatch using an n-dimensional table lookup

  Example
  ```
  int foo(C);
  int foo(D);
  C p = new D();
  int i = foo(p);   // calls foo(D)
  ```

Function Call vs. Message Passing

- Function call syntax (C++)
  ```
  p.foo(5);
  p->foo(5);
  ```
- Message passing syntax (SmallTalk)
  ```
  p foo 5.
  6 * 7.
  someArray at: 1 put: 5.
  x = 0 ifTrue: [y <- 1]
  ifFalse: [y <- 2].
  ```
Typed vs. Untyped

- **Statically typed (C++, Java, etc.)**
  - no 'message not understood' errors at run time
  - more efficient dispatch since layout of dispatch table is known
- **Untyped (SmallTalk, CLOS, Cecil)**
  - more flexibility
  - try to infer types for optimizing dispatch

Abstract Classes

```cpp
class A {
public:
   virtual int foo() = 0;
};
class C : public A { /* ... */ }
```

- Defines type
- Implementation is supplied in subclass

Interface Types

```java
interface I {
   int foo ();
}
class C implements I { /* ... */ }
```

- Separation of interface and implementation
- Cleaner design of type hierarchies
Structural Subtyping or Retroactive Abstraction

```java
class C { public int foo() { ... } }
interface I {
    int foo();
}

● Structural Subtyping
    I p = new C();

● Retroactive Abstraction
    class C implements I;
    I p = new C();
```

Signatures in G++

```java
signature I {
    int foo();
};
class C { public: int foo(); }; I * p = new C;

● Implemented in G++ Versions 2.6-2.95
● Use `-fhandle-signatures' option
```