Theory of Programming Languages

Week 11

Support for Object Oriented Programming

Support for Object Oriented Programming

(I) Introduction

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., Ada 95 and C++)
  - Some support functional program (e.g., CLOS)
  - Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)
  - Some are pure OOP language (e.g., Smalltalk & Ruby)

(II) Object Oriented Programming

- Abstract data types
- Inheritance
  - Inheritance is the central theme in OOP and languages that support it
- Polymorphism

Inheritance

- Productivity increases can come from reuse
  - ADTs are difficult to reuse—always need changes
  - All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- Inheritance addresses both of the above concerns--reuse ADTs after minor changes and define classes in a hierarchy

Object Oriented Concepts

- ADTs are usually called *classes*
- Class instances are called *objects*
- A class that inherits is a *derived class* or a *subclass*
- The class from which another class inherits is a parent class or *superclass*
- Subprograms that define operations on objects are called *methods*

- Calls to methods are called *messages*
- The entire collection of methods of an object is called its *message protocol* or *message interface*
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent

- Inheritance can be complicated by access controls to encapsulated entities
  - A class can hide entities from its subclasses
  - A class can hide entities from its clients
  - A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one *overrides* the inherited one
  - The method in the parent is *overriden*

- There are two kinds of variables in a class:
  - *Class variables* - one/class
  - *Instance variables* - one/object
- There are two kinds of methods in a class:
  - *Class methods* – accept messages to the class
  - *Instance methods* – accept messages to objects
- Single vs. Multiple Inheritance
- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance

Dynamic Binding

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants
- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic
- Allows software systems to be more easily extended during both development and maintenance

Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- An *abstract class* is one that includes at least one virtual method
- An abstract class cannot be instantiated

(III) Design Issues for OOP languages

- The Exclusivity of Objects
- Are Subclasses Subtypes
- Type Checking and Polymorphism
- Single and Multiple Inheritance
- Object Allocation and DeAllocation
- Dynamic and Static Binding
- Nested Classes

Exclusivity of Objects

- Everything is an object
  - Advantage - elegance and purity
  - Disadvantage - slow operations on simple objects
- Add objects to a complete typing system
  - Advantage - fast operations on simple objects
  - Disadvantage - results in a confusing type system (two kinds of entities)
- Include an imperative-style typing system for primitives but make everything else objects
  - Advantage - fast operations on simple objects and a relatively small typing system
  - Disadvantage - still some confusion because of the two type systems

Are subclasses sub-types?

- Does an "is-a" relationship hold between a parent class object and an object of the subclass?
  - If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object
- A derived class is a subtype if it has an is-a relationship with its parent class
  - Subclass can only add variables and methods and override inherited methods in "compatible" ways

Type Checking for Polymorphism

- Polymorphism may require dynamic type checking of parameters and the return value
  - Dynamic type checking is costly and delays error detection
- If overriding methods are restricted to having the same parameter types and return type, the checking can be static

Single and Multiple Inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:
  - Language and implementation complexity (in part due to name collisions)
  - Potential inefficiency - dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - Sometimes it is quite convenient and valuable

Allocation and De-Allocation of Objects

- From where are objects allocated?
  - If they behave line the ADTs, they can be allocated from anywhere
    - Allocated from the run-time stack
    - Explicitly create on the heap (via new)
  - If they are all heap-dynamic, references can be uniform thru a pointer or reference variable
    - Simplifies assignment - dereferencing can be implicit
  - If objects are stack dynamic, there is a problem with regard to subtypes
- Is deallocation explicit or implicit?

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?
  - If none are, you lose the advantages of dynamic binding
  - If all are, it is inefficient
- Allow the user to specify

Nested Classes

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class
- Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa

(IV) Support for OOP in Smalltalk

- Smalltalk is a pure OOP language
  - Everything is an object
  - All objects have local memory
  - All computation is through objects sending messages to objects
  - None of the appearances of imperative languages
  - All objected are allocated from the heap
  - All deallocation is implicit

small, Smalltalk examples

```
OrderedCollection subclass: #Group.

Group >> eachRespondsTo: aSelector
    ^ self allSatisfy: [ :each | each respondsTo: aSelector ]

Group >> doesNotUnderstand: aMessage
    ^ (self eachRespondsTo: aMessage selector)
        ifTrue: [ self collect: [ :each | aMessage sendTo: each
        ifFalse: [ super doesNotUnderstand: aMessage ]
```

```
g := Group new.
g should not respondTo: #x.
[ g x ] should raise: MessageNotUnderstood.
g add: 2 @ 3.
g add: 3 @ 4.
g add: 1 @ 2.
g should respondTo: #x.
g x should beSameSequence: #(2 3 1).
g y should beSameSequence: #(3 4 2).
```

- Type Checking and Polymorphism
  - All binding of messages to methods is dynamic
    - The process is to search the object to which the message is sent for the method; if not found, search the superclass, etc. up to the system class which has no superclass
  - The only type checking in Smalltalk is dynamic and the only type error occurs when a message is sent to an object that has no matching method

- An object in Smalltalk is like a C "struct" or a Pascal record. The only difference between a record and an object is that the object contains a single special field that identifies which class the object belongs to.
- In C++ and Java, too, there is a single hidden field in every object. But whereas in Smalltalk, this field is called the "class pointer." In C++ and Java this hidden field is called the "V-table pointer" or "dispatch table pointer".

- Inheritance
  - A Smalltalk subclass inherits all of the instance variables, instance methods, and class methods of its superclass
  - All subclasses are subtypes (nothing can be hidden)
  - All inheritance is implementation inheritance
  - No multiple inheritance

- Evaluation of Smalltalk
  - The syntax of the language is simple and regular
  - Good example of power provided by a small language
  - Slow compared with conventional compiled imperative languages
  - Dynamic binding allows type errors to go undetected until run time
  - Introduced the graphical user interface
  - Greatest impact: advancement of OOP

(V) Support for OOP in C++

- General Characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities

- Inheritance
  - A class need not be the subclass of any class
  - Access controls for members are
    - Private (visible only in the class and friends) (disallows subclasses from being subtypes)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)

- In addition, the subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses
  - Private derivation - inherited public and protected members are private in the subclasses
  - Public derivation public and protected members are also public and protected in subclasses

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class { … };
//      In this one, b and y are protected and
//      c and z are public

class subclass_2 : private base_class { … };
//     In this one, b, y, c, and z are private,
//     and no derived class has access to any
//     member of base_class
```

- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (::), e.g.,

```
class subclass_3 : private base_class {
        base_class :: c;
        …
}
```

- One motivation for using private derivation
  - A class provides members that must be visible, so they are defined to be public members; a derived class adds some new members, but does not want its clients to see the members of the parent class, even though they had to be public in the parent class definition

- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator

- Dynamic Binding
  - A method can be defined to be `virtual`, which means that they can be called through polymorphic variables and dynamically bound to messages
  - A pure virtual function has no definition at all
  - A class that has at least one pure virtual function is an *abstract class*

- Evaluation
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance
  - In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound
    - Static binding is faster!
  - Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  - Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

Support for Object Oriented Programming (continued…)

- Introduction
- Object-Oriented Programming
- Design Issues for Object-Oriented Languages
- Support for Object-Oriented Programming in Smalltalk
- Support for Object-Oriented Programming in C++
- Support for Object-Oriented Programming in Objective-C
- Support for Object-Oriented Programming in Java
- Support for Object-Oriented Programming in C#
- Support for Object-Oriented Programming in Ada 95
- Support for Object-Oriented Programming in Ruby
- Implementation of Object-Oriented Constructs