今天的内容

- Classes and objects – part III
- Foundations of the object model
- Elements of object model
  - Abstraction
  - Encapsulation
  - Modularization
  - Hierarchy

Class & Objects
Part III

齐琦

Abstract Classes

- Like an ordinary class except some methods or fields incomplete
- Abstract keyword for class having methods without definitions or fields without initialization
  - Requires type info for an abstract var or val

```
// AbstractKeyword.scala
abstract class WithValVar {
  val x:Int
  var y:Int
}

abstract class WithMethod {
  def f():Int
  def g(n:Double)
}
```

- Why uses abstract methods?
  - Template method --- captures common behavior in base class, relegates details that vary to derived classes

```
abstract class Animal {
  def templateMethod =
    s"The $animal goes $sound"
  // Abstract methods (no method body):
  def animal:String
  def sound:String
}

// Error -- abstract class
// cannot be instantiated:
// val a = new Animal

class Duck extends Animal {
  def animal = "Duck"
  // "override" is optional here:
  override def sound = "Quack"
}

```

```
class Cow extends Animal {
  def animal = "Cow"
  def sound = "Moo"
}

(new Duck).templateMethod is
"The Duck goes Quack"
(new Cow).templateMethod is
"The Cow goes Moo"
```

+

- Legal for templateMethod to call the animal and sound methods, even they haven't been defined yet

- Scala not allow make an instance of an abstract class

- Override is optional for definition in child class for abstract method from parent class; generally leave it out.

- Abstract classes can have arguments
  - Class inherits Adder can perform base-class initialization by calling Adder constructor

```
// AbstractAdder.scala
import com.atomicscala.AtomicTest._

abstract class Adder(x:Int) {
  def add(y:Int):Int
}
```

Traits

Traits

- Traits are basic piece of functionality that allow you to easily "mix in" ideas to create a class ------ often called *mixin types*
- Ideally, a trait represents a single concept
- Trait keyword; extends keyword; with keyword --- add additional traits

```
trait Color
trait Texture
trait Hardness

class Fabric

class Cloth extends Fabric with Color
  with Texture with Hardness

class Paint extends Color with Texture
  with Hardness
```

+

- Fields and methods in traits can be left abstract

```scala
trait AllAbstract {
  def f(n:Int):Int
  val d:Double
}

trait PartialAbstract {
  def f(n:Int):Int
  val d:Double
  def g(s:String) = s"($s)"
  val j = 42
}

trait Concrete {
  def f(n:Int) = n * 11
  val d = 1.61803
}
```

```scala
/* None of these are legal -- traits
cannot be instantiated:
new AllAbstract
new PartialAbstract
new Concrete
*/

// Scala requires 'abstract' keyword:
abstract class Klass1 extends AllAbstract
  with PartialAbstract
```

+

```scala
/* Can't do this -- d and f are undefined:
new Klass1
*/

// Class can provide definitions:
class Klass2 extends AllAbstract {
  def f(n:Int) = n * 12
  val d = 3.14159
}

new Klass2

// Concrete's definitions satisfy d & f:
class Klass3 extends AllAbstract
  with Concrete

new Klass3

class Klass4 extends PartialAbstract
  with Concrete

new Klass4
```

```scala
class Klass5 extends AllAbstract
  with PartialAbstract with Concrete

new Klass5

trait FromAbstract extends Klass1
trait fromConcrete extends Klass2

trait Construction {
  println("Constructor body")
}

class Constructable extends Construction
new Constructable
```

```scala
// Create unnamed class on-the-fly:
val x = new AllAbstract with
  PartialAbstract with Concrete
```

- Trait cannot be instantiated
- Definitions can be provided by class (as in Klass2), or through other traits (as Concrete does in Klass3, Klass4, Klass5)
- Traits can inherit from abstract or concrete classes
- Traits cannot have constructor arguments, but can have constructor bodies
- Create instance of a class that you assemble at site of creation. (Lines 70-71)
  - Type of the resulting object has no name

- Traits can inherit from other traits

```scala
trait Base {
  def f = "f"
}

trait Derived1 extends Base {
  def g = "17"
}

trait Derived2 extends Derived1 {
  def h = "1.11"
}

class Derived3 extends Derived2

val d = new Derived3

d.f
d.g
d.h
```

+

- If method or field signatures collide, can resolve by hand

```
trait A {
  def f = 1.1
  def g = "A.g"
  val n = 7
}

trait B {
  def f = 7.7
  def g = "B.g"
  val n = 17
}

object C extends A with B {
  override def f = 9.9
  override val n = 27
  override def g = super[A].g + super[B].g
}

C.f is 9.9
C.g is "A.gB.g"
C.n is 27
```

- Trait fields and methods can be used in calculation, even they haven’t been defined

```scala
trait Framework {
  val part1:Int
  def part2:Double
  // Even without definitions:
  def templateMethod = part1 + part2
}

def operation(impl:Framework) =
  impl.templateMethod

class Implementation extends Framework {
  val part1 = 42
  val part2 = 2.71828
}
```

```
operation(new Implementation) is 44.71828
```

Defining an operation in a base type that relies on pieces that will be defined by a derived type is called Template Method pattern and is foundation for many frameworks.

Some object-oriented languages support multiple inheritance to combine multiple classes.
Traits are usually considered a superior solution. If you have a choice between classes and traits, prefer traits.

Uniform Access & Setters

```
4   trait Base {
5     def f1:Int
6     def f2:Int
7     val d1:Int
8     val d2:Int
9     var d3:Int
10    var n = 1
11  }
12
13  class Derived extends Base {
14    def f1 = 1
15    val f2 = 1 // Was def, now val
16    val d1 = 1
17    // Can't do this; must be a val:
18    // def d2 = 1
19    val d2 = 1
20    def d3 = n
21    def d3_=(newVal:Int) = n = newVal
22  }
23
24  val d = new Derived
25  d.d3 is 1 // Calls getter (line 20)
26  d.d3 = 42 // Calls setter (line 21)
27  d.d3 is 42
```

- Line 15 implements def on line 6 using a val
  - In Scala, methods without arguments treated identically to vals with same type
  - Uniform Access Principle
    - From client view, can't tell how sth been implemented
- If have a val in base type, can't implement it using a def
  - Scala says "method d2 needs to be a stable, immutable value."
  - Val --- things can't change
  - Def --- execute code, produce result
- If a var (line 9), no promise it always same, can implement with def
  - Also needs a setter in addition getter

Reaching into Java

```
scala> import java.util.Date
import java.util.Date

scala> val d = new Date
```

```
// LinearRegression.scala
import com.atomicscala.AtomicTest._
import org.apache.commons.math._
import stat.regression.SimpleRegression

val r = new SimpleRegression
r.addData(1, 1)
r.addData(2, 1.1)
r.addData(3, 0.9)
r.addData(4, 1.2)

r.getN is 4
r.predict(6) is 1.19
```

- Import Java classes
  - Entire Java standard library available using import like this
- Also can download third-party Java libs and use in Scala
  - E.g. apache common math lib
  - Linear regression usage example
- Rice Java libs, a huge benefit to Scala

Applications

```scala
object WhenAmI extends App {
  hi
  println(new java.util.Date())
  def hi = println("Hello! It's:")
}
```

```
scalac Compiled.scala
```

```
scala WhenAmI
```

```scala
// CompiledWithArgs.scala

object EchoArgs extends App {
  for(arg <- args)
    println(arg)
}
```

- Object extends App
  - Constructor statements execute in order
- It doesn't matter what you call the file; the name of the resulting program depends on the name of the object.

- another form that follows a pattern used in older programming languages: you define a method called main, and the method arguments contain the command-line arguments.

- all the arguments come in as Strings. There's no particular reason to use a main other than that it might make the code more familiar to programmers from other languages (Java, in particular).

```
object EchoArgs2 {
  def main(args:Array[String]) =
    for(arg <- args)
      println(arg)
}
```

A little Reflection

- Reflection means taking an object and holding it up to a mirror, so it can discover things about itself.
- Example: take an object, find out its class name
  - Create a trait to add a toString to any class, to display the class name

```
package com.atomicscala
import reflect.runtime.currentMirror

object Name {
  def className(o:Any) =
    currentMirror.reflect(o).symbol.
    toString.replace('$', ' ').
    split(' ').last
}

trait Name {
  override def toString =
    Name.className(this)
}
```

```
import com.atomicscala.Name

class Solid extends Name
val s = new Solid
s is "Solid"

class Solid2(val size:Int) extends Name {
  override def toString =
    s"${super.toString}($size)"
}
val s2 = new Solid2(47)
s2 is "Solid2(47)"
```

Scala's reflection API is much more powerful and complex than we've shown here.

Polymorphism

Polymorphism

- Greek term, "many forms"
- In programming, it means we can perform the same operation on different types
- If we create a class using another class A along with traits B and C, we can choose to treat that class as if it were only an A or only a B or only a C
  - E.g. , animals, vehicles, mobile trait (ref. to code demo)

Example: design a game

- Each element in the game will draw itself on screen based on its location
  - When two elements in proximity, they'll interact
  - Sketch a draft making use of polymorphism

```scala
// Polymorphism.scala
import com.atomicscala.AtomicTest._
import com.atomicscala.Name

class Element extends Name {
  def interact(other:Element) =
    s"$this interact $other"
}

class Inert extends Element
class Wall extends Inert

trait Material {
  def resilience:String
}
trait Wood extends Material {
  def resilience = "Breakable"
}
trait Rock extends Material {
  def resilience = "Hard"
}
class RockWall extends Wall with Rock
class WoodWall extends Wall with Wood

trait Skill
trait Fighting extends Skill {
  def fight = "Fight!"
}
trait Digging extends Skill {
  def dig = "Dig!"
}
trait Magic extends Skill {
  def castSpell = "Spell!"
}
trait Flight extends Skill {
  def fly = "Fly!"
}
```

```
class Character(var player:String="None")
  extends Element
class Fairy extends Character with Magic
class Viking extends Character
  with Fighting
class Dwarf extends Character with Digging
  with Fighting
class Wizard extends Character with Magic
class Dragon extends Character with Magic
  with Flight

val d = new Dragon
d.player = "Puff"
```

```
d.interact(new Wall) is
"Dragon interact Wall"

def battle(fighter:Fighting) =
  s"$fighter, ${fighter.fight}"
battle(new Viking) is "Viking, Fight!"
battle(new Dwarf) is "Dwarf, Fight!"
battle(new Fairy with Fighting) is
"1, Fight!" // Name: $anon$1

def fly(flyer:Element with Flight,
  opponent:Element) =
    s"$flyer, ${flyer.fly}, " +
    s"${opponent.interact(flyer)}"

fly(d, new Fairy) is
"Dragon, Fly!, Fairy interact Dragon"
```

- Interact method on line 6
  - One element interacts with another
- Create different types of elements, and traits to mix in to achieve different effects
  - Traits can inherit from each other
- Skill trait on line 25, classify different traits; can add common fields or methods in Skill
- Characters; constructor argument player on line 39, has a default argument
- Create a Dragon on line 50, change player the name
- Line 52, first example of polymorphism
  - Interact method takes an Element or anything derived from Element. --- a polymorphism

- That’s powerful, because now method can also applies to anything that inherits from that type(method’s argument type)
  - Transparent and safe, because Scala guarantees that a derived class “is a ” base class, by ensuring that derived class has all methods of the base class
- Viking and Dwarf include Fighting trait, so they can be passed to battle --- demonstrates polymorphism; otherwise you must write specific methods
- Polymorphism is a tool allows you to write less code and make it more reusable

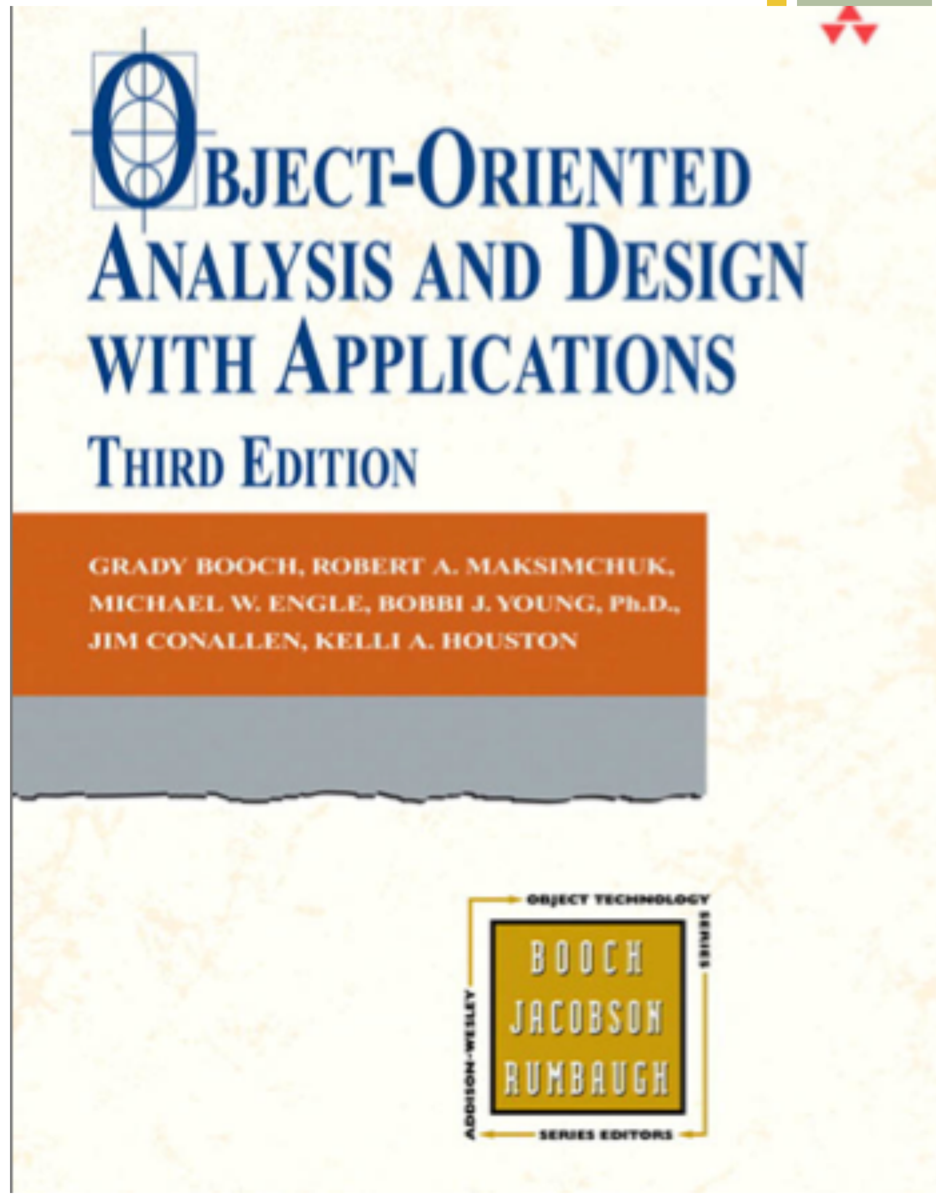
- Line 59, the argument passed to battle:
  - New Fairy with Fighting
  - Creates a new class, and immediately make an instance of that class; not give the class a name
- Line 62, argument flyer's type as "Element with Flight"
  - Arguments passed includes both Element and Flight, so fly can call everything it needs to
- "How did you know to do it this way?"
  - The Design challenge
  - Once decide what you want to build, many different ways to assemble it

- Create a base class and add new methods during inheritance, or mix in functionality using traits
- Design decisions
  - Using a combination of experience and observing the way your system is used
- Design process
  - Decide what makes sense based on the requirements of your system
- The pragmatic approach is not to assume that you can get it all right the first time. Instead, write something, get it working, then see how it looks. As you learn, "refactor" your code until the design feels right (don't settle for the first thing that works).

Evolution of the Object Model

Another Textbook

- "Object-Oriented Analysis and Design with Applications"
  - 3rd edition
  - Brady Booch, etc.
  - Addison-Wesley



- Two sweeping trends

1. The shift in focus from programming-in-the-small to programming-in-the-large
2. The evolution of high-order programming languages

- Complexity in software system prompted applied research in software engineering
  - Decomposition
  - Abstraction
  - Hierarchy

- Needs more expressive programming languages

Generations of programming languages

- First-generation languages (1954–1958)

FORTRAN I	Mathematical expressions
ALGOL 58	Mathematical expressions
Flowmatic	Mathematical expressions
IPL V	Mathematical expressions

- Second-generation languages (1959–1961)

FORTRAN II	Subroutines, separate compilation
ALGOL 60	Block structure, data types
COBOL	Data description, file handling
Lisp	List processing, pointers, garbage collection

- Third-generation languages (1962–1970)

PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	Rigorous successor to ALGOL 60
Pascal	Simple successor to ALGOL 60
Simula	Classes, data abstraction

- The generation gap (1970–1980)

Many different languages were invented, but few endured. Howe lowing are worth noting:

C	Efficient; small executables
FORTRAN 77	ANSI standardization

- Object-orientation boom (1980–1990, but few languages survive)

Smalltalk 80	Pure object-oriented language
C++	Derived from C and Simula
Ada83	Strong typing; heavy Pascal influence
Eiffel	Derived from Ada and Simula

- Emergence of frameworks (1990–today)

Much language activity, revisions, and standardization have occurred, leading to programming frameworks.

Visual Basic	Eased development of the graphical user interface (GUI) for Windows applications
Java	Successor to Oak; designed for portability
Python	Object-oriented scripting language
J2EE	Java-based framework for enterprise computing
.NET	Microsoft's object-based framework
Visual C#	Java competitor for the Microsoft .NET Framework
Visual Basic .NET	Visual Basic for the Microsoft .NET Framework

- First-generation
  - Primarily for scientific and engineering apps, vocabulary entirely mathematics; write math formulas, freeing from assembly or machine code.
- Second-generation
  - Machine gets powerful, business application
  - Emphasis on algorithmic abstractions; tell machine what to do
- Third
  - Transistors advent; integrated circuit technology; hardware cost dropped
  - Demands of data manipulation; Support for data abstraction

- 70s
  - Thousand of different program languages;
  - Larger programs highlighted inadequacies of earlier languages
  - Few survived; but many concepts introduced adopted by successors
- Object-oriented (from 80s, 90s)
  - Object-oriented decomposition of software
  - Main streams: Java, C++, etc.
  - Emergence of frameworks (e.g. J2EE, .NET)

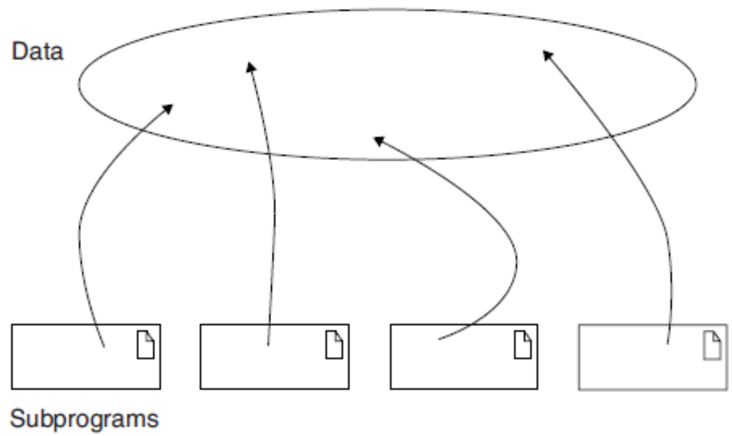


Figure 2–1 The Topology of First- and Early Second-Generation Programming Languages

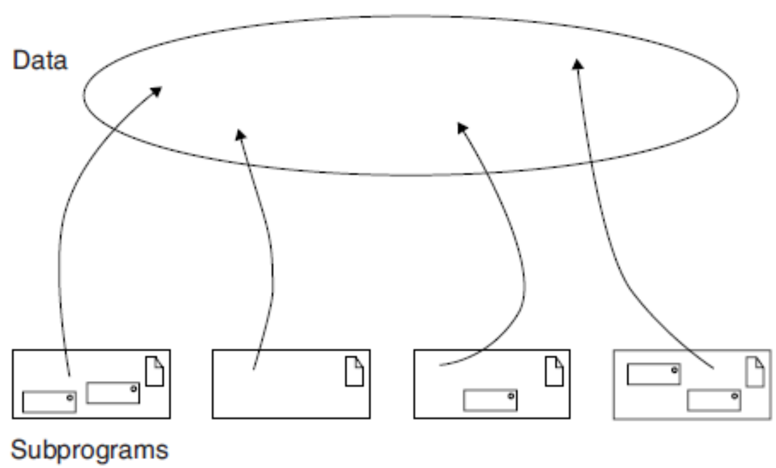


Figure 2–2 The Topology of Late Second- and Early Third-Generation Programming Languages

Subprograms as an abstraction mechanism

+

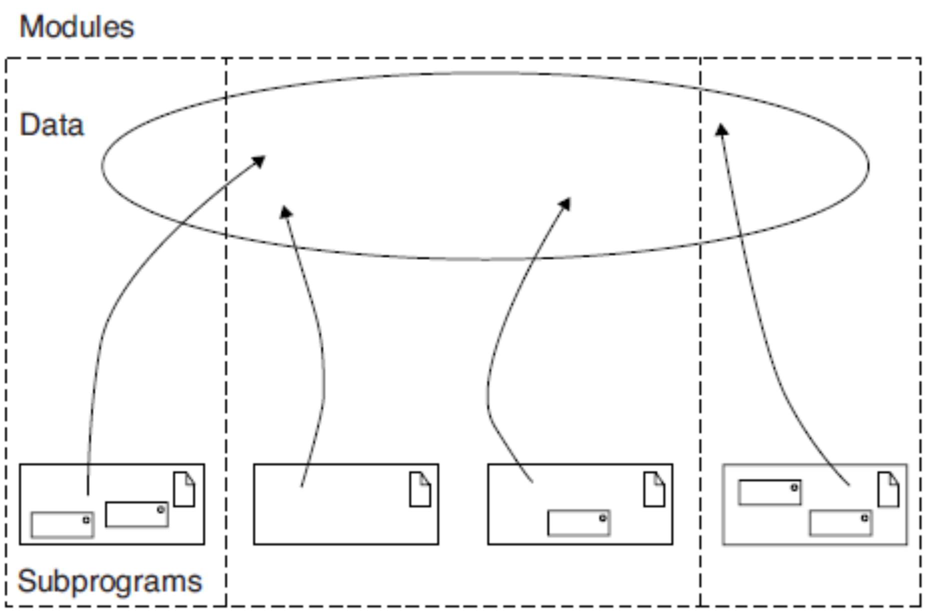


Figure 2–3 The Topology of Late Third-Generation Programming Languages

Modular structure

Most lacked support for data abstraction and strong typing, some errors can only detected during execution of program.

For object-oriented

- Data abstraction important to master complexity of problem.
- Physical building block is module(a logical collection of classes and objects instead of subprograms)

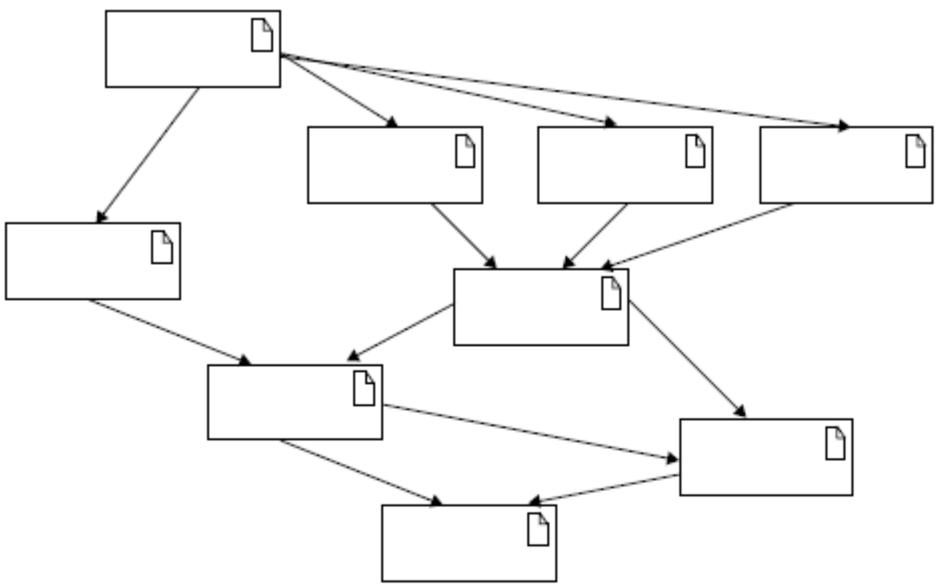


Figure 2–4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

For object-oriented

- Data and operations are united, that fundamental logical building blocks are no longer algorithms, but classes and objects
- Little or no global data

guages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" [6]. For this reason, the

Foundations of Object Model

Object-oriented programming(OOP)

- Object orientation cope with complexity inherent in many different systems
  - Not just to programming languages, user interface design, databases, computer architectures

- OOP

  > Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

  - Uses objects as building blocks
  - Each object is an instance of some class
  - Classes relates to one another via inheritance

What's object-oriented?

- Cardelli and Wegner say:

[A] language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types [classes] may inherit attributes from supertypes [superclasses]. [34]

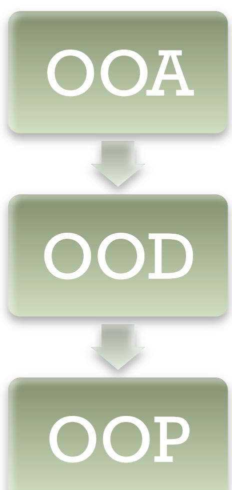
Object-oriented design(OOD)

- Leads to an object-oriented decomposition
- Uses different notations to express different models of logical (class and object structure), and physical (module and process architecture) design of a system

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 system under design.

Object-oriented analysis(OOA)

Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.



OOA serves OOD; OOD as blueprints for implementing system using OOP methods

Elements of the Object Model

Programming style

- No single one best for all kinds applications.
  - Knowledge base
  - Computation-intense operation
  - Broadest set of applications

1.	Procedure-oriented	Algorithms
2.	Object-oriented	Classes and objects
3.	Logic-oriented	Goals, often expressed in a predicate calculus
4.	Rule-oriented	If–then rules
5.	Constraint-oriented	Invariant relationships

Elements of object model

- Conceptual framework for object-oriented, is the object model

- Four major elements of this model(a model without any one of these is not object-oriented)
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy

- Three minor elements: (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

Meaning of Abstraction

- Define abstraction:

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

- Focus on outside view of an object, separate object's essential behavior from its implementation

- Decide right set of abstractions for a given domain, is central problem in OOD

Spectrum of abstraction

- From most to least useful:

Entity abstraction	An object that represents a useful model of a problem domain or solution domain entity
Action abstraction	An object that provides a generalized set of operations, all of which perform the same kind of function
Virtual machine abstraction	An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
Coincidental abstraction	An object that packages a set of operations that have no relation to each other

+

- A client is any object that uses resources of another object (known as server).
  - Characterize behavior of an object by considering services it provides to other objects
  - Force to concentrate on outside view of an object, which defines a contract on which other objects may depend, and which must be carried out by inside view

- Protocol: entire set of operations that contributes to the contract, with legal ordering of their invoking
  - Denotes ways that object may act and react, thus constitutes entire outside view of the abstraction.

- Terms: operation, method, member function virtually mean same thing.

Examples of Abstraction

- Farm, maintaining proper greenhouse environment
- A key abstraction is about a sensor
  - A temperature sensor: an object that measures temperature at a location
  - What are responsibilities of a temp sensor? Answers yield different design decisions

Abstraction: Temperature Sensor
Important Characteristics: temperature location
Responsibilities: report current temperature calibrate

Figure 2–6 Abstraction of a `Temperature Sensor`

Abstraction: Active Temperature Sensor
Important Characteristics: temperature location setpoint
Responsibilities: report current temperature calibrate establish setpoint

Figure 2–7 Abstraction of an `Active Temperature Sensor`

- No objects stands alone; every object collaborates with others to achieve some behavior.
- Design decisions about how they cooperate, define boundaries of each abstraction and the responsibilities and protocol of each object.

Meaning of Encapsulation

- Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.
- Encapsulation is most often achieved through information hiding (not just data hiding)
- Whereas abstraction "helps people to think about what they are doing," encapsulation "allows program changes to be reliably made with limited effort"
- Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns.
  - DB application, programs depend on a schema(data's logical view),not care physical data representation

- For abstraction to work, implementations must be encapsulated
  - each class must have two parts: an interface and an implementation
  - Interface – outside view, behavior abstraction
  - Implementation – achieve the behavior
- Define encapsulation
  - "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation. "

Encapsulation contd.

- Encapsulates implementation details; no client need know about the implementation decisions
  - Because it not affect observable behavior of class
- As system evolves, implementation often changed to use more efficient algorithms
- Ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Meaning of Modularity

- Partition a program into individual components
  - Reduce complexity
  - Creates well-defined, documented boundaries within program
  - Examples
    - Smalltalk – class
    - Java – packages containing classes
    - C++, Ada – module construct
- Classes and objects form logical structure a system; place them in modules to produce system's physical architecture
  - Hundreds classes, to help manage complexity

- Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.
- Deciding on the right set of modules for a given problem is almost as hard a problem as deciding on the right set of abstractions.
- Modules serve as the physical containers in which we declare the classes and objects of our logical design.

- Guidelines
  - overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently
  - In practice, the cost of recompiling the body of a module is relatively small: Only that unit need be recompiled and the application relinked
  - a module's interface should be as narrow as possible, yet still satisfy the needs of the other modules that use it.
    - cost of recompiling the interface of a module is relatively high
  - hide as much as we can in the implementation of a module

- The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions and the need to make certain abstractions visible to other modules.
  - strive to build modules that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among modules)
- Define modularity
  - Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- Additional technical issues may affect modularization decisions
  - Modules as units of a software can be reused across applications; package classes and objects into modules way that makes reuse convenient
  - many compilers generate object code in segments, one for each module; may be practical limits on the size of individual modules
- Modules also serve as the unit of documentation and configuration management. (more modules more docs)
- Identification of classes and objects is part of the logical design of the system, but identification of modules is part of the system's physical design.
  - These design decisions happen iteratively

Meaning of Hierarchy

- Encapsulation helps manage this complexity by hiding the inside view of our abstractions ; Modularity helps also, by giving us a way to cluster logically related abstractions.
- Define Hierarchy
  - Hierarchy is a ranking or ordering of abstractions.
- Two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

Examples of Hierarchy

- Single Inheritance
- Multiple Inheritance
- Aggregation

Single Inheritance

- "is a" hierarchy, relationship
  - A bear "is a" kind of mammal
- Inheritance defines a relationship among classes
  - One class shares structure or behavior defined in on class or more classes (single inheritance or multiple inheritance, respectively)
  - A subclass augments or redefines existing structure and behavior of its super-classes
- Imply a generalization/specialization hierarchy
  - Subclass specializes more general structure or behavior of its superclasses.
  - As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses

- Trade off support for encapsulation and inheritance
  - Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction
  - C++ and Java offer great flexibility
  - he interface of a class may have three parts:
    - private parts, which declare members that are accessible only to the class itself;
    - protected parts, which declare members that are accessible only to the class and its subclasses;
    - public parts, which are accessible to all clients

Examples of Hierarchy: Multiple Inheritance

- Inheritance from multiple super-classes
- Flowering plant, fruits and vegetables plant example
  - classes that independently capture the properties unique to flowering plants and to fruits and vegetables ;
  - They have no superclass; they stand alone. These are called ***mixin classes*** because they are meant to be mixed together with other classes to produce new subclasses.

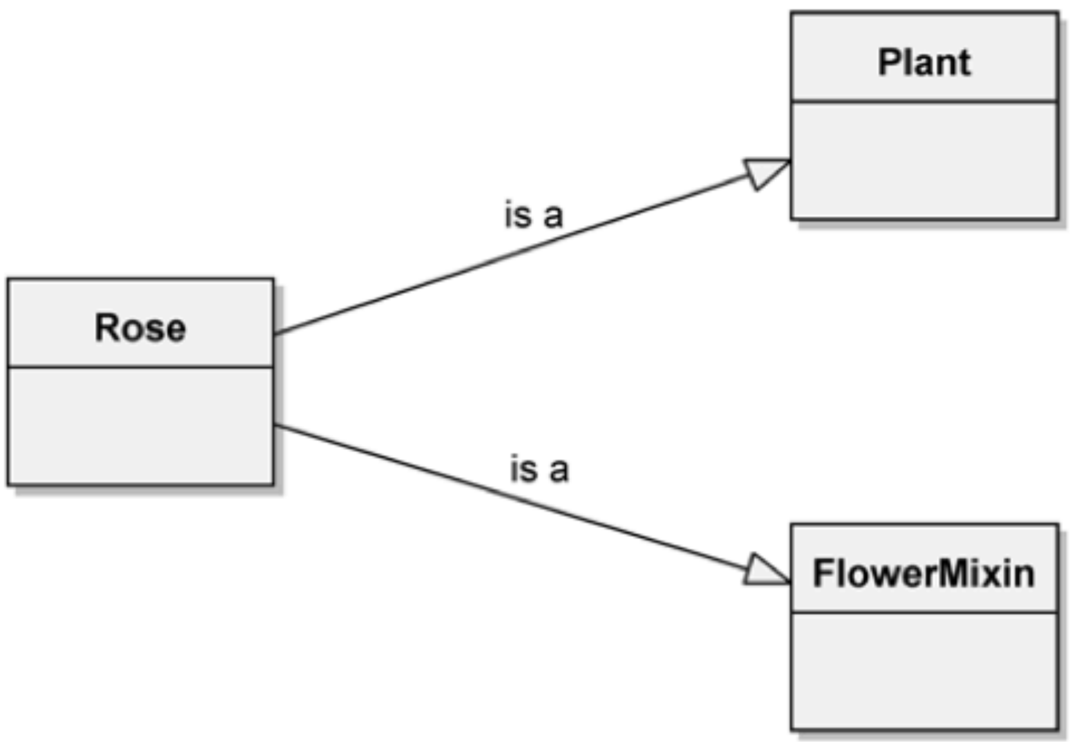
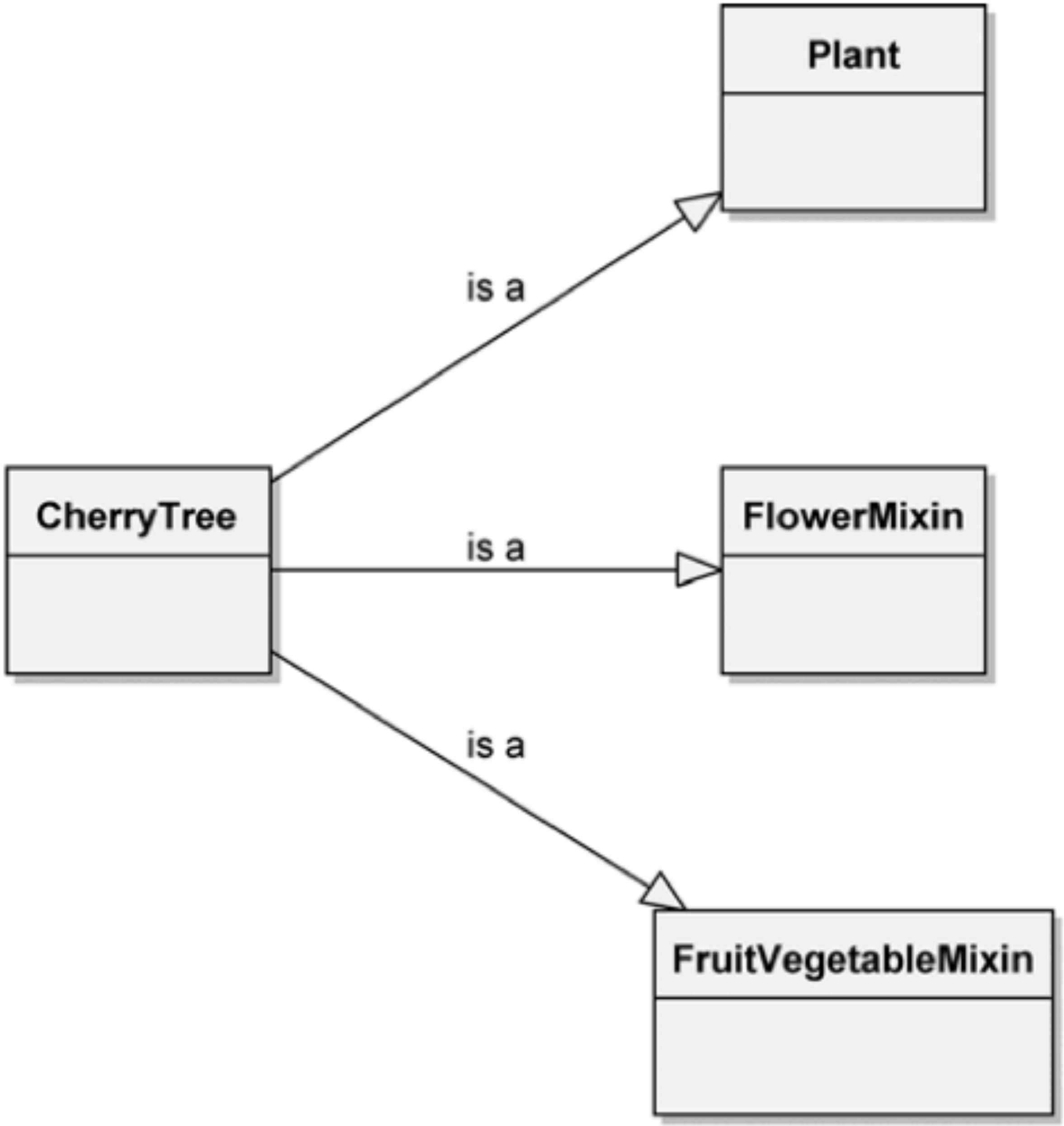


Figure 2–10 The `Rose` Class, Which Inherits from Multiple Superclasses



gure 2–12 The CherryTree Class, Which Inherits from Multiple Superclasses

- Languages must address two issues: clashes among names from different superclasses and repeated inheritance.
  - Repeated inheritance occurs when two or more peer superclasses share a common superclass
  - question arises, does the leaf class (i.e., subclass) have one copy or multiple copies of the structure of the shared superclass?
  - Some languages prohibit repeated inheritance, some unilaterally choose one approach, and others, such as C++, permit the programmer to decide
  - In C++, virtual base classes are used to denote a sharing of repeated structures, whereas nonvirtual base classes result in duplicate copies appearing in the subclass

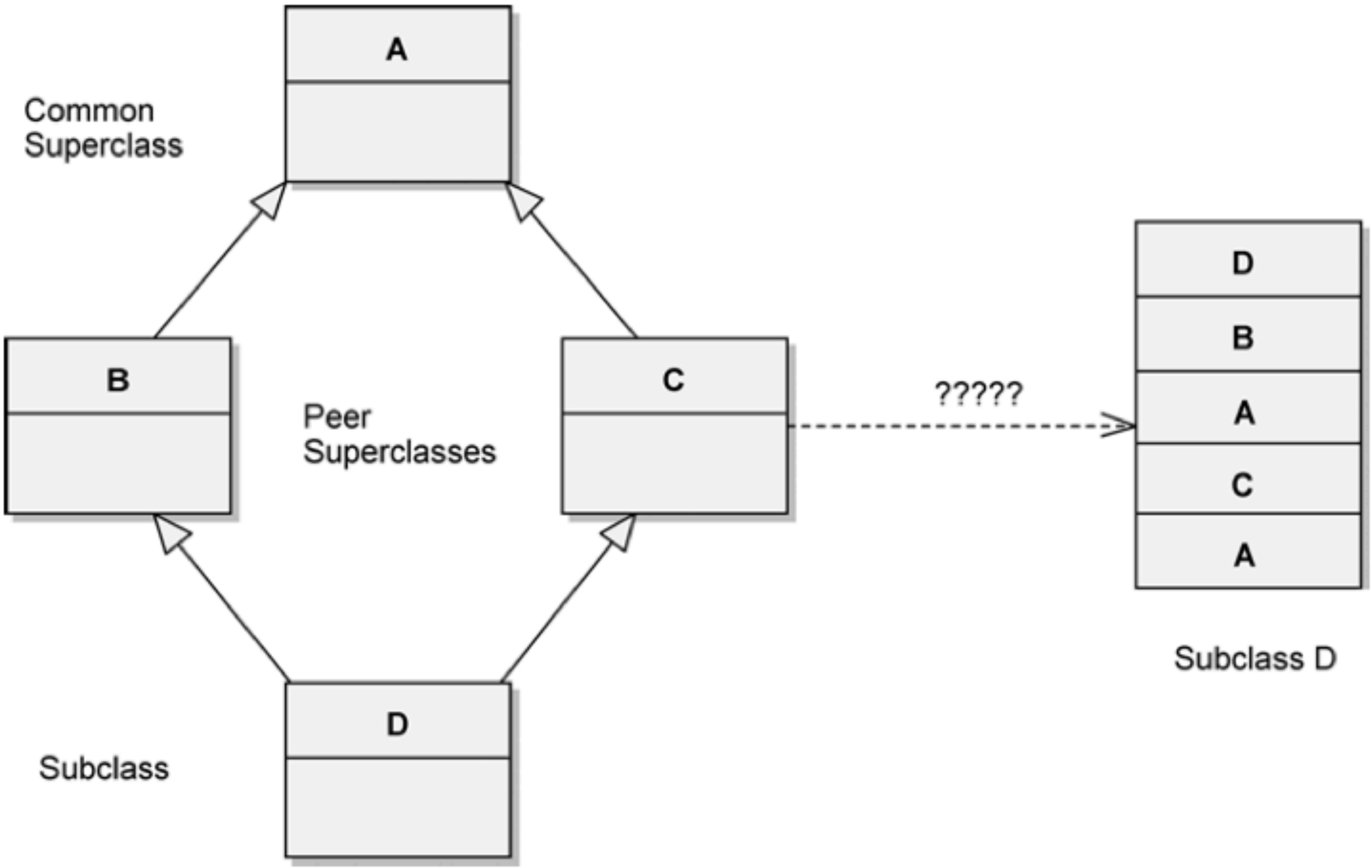


Figure 2–13 The Repeated Inheritance Problem

Examples of Hierarchy: Aggregation