今天的内容

- Classes and objects – part V (the last part)

- Elements of object model (concepts)
  - Abstraction
  - Encapsulation
  - Modularization
  - Hierarchy
  - Typing
  - Concurrency
  - Persistence

Type Parameter Constraints

- Scenario: you like your enumeration to be a subtype of a trait?
  - If a normal class, just add the trait to list of base types during inheritance
  - But with enumerations, must create a new Value type by inheriting from Val
- Following example shows:
  - parameterized types and traits,
  - and type constraints, allow to impose conditions on type parameters

```
4  trait Resilience
5
6  object Bounciness extends Enumeration
7    case class _Val() extends Val
8      with Resilience
9    type Bounciness = _Val
10   val level1, level2, level3 = _Val()
11 }
12 import Bounciness._
13
14 object Flexibility extends Enumeratio
15   case class _Val() extends Val
16     with Resilience
17   type Flexibility = _Val
18   val type1, type2, type3 = _Val()
19 }
20 import Flexibility._
```

```
22 trait Spring[R <: Resilience] {
23   val res:R
24 }
25
26 case class BouncingBall(res:Bounciness)
27   extends Spring[Bounciness]
28
29 BouncingBall(level2) is
30 "BouncingBall(level2)"
31
32 case class FlexingWall(res:Flexibility)
33   extends Spring[Flexibility]
34
35 FlexingWall(type3) is "FlexingWall(type3)"
```

- If only trait Spring[R]{}, then container types can be quite flexible; however, if want to let R call a method, then must determine that it has the method to be called, you must constrain R using bounds
- <: , it says "R must be of type Resilience, or something inherited from Resilience"
  - Equivalently, Resilience is the upper bound for R
- BouncingBall and FlexingWall each extend Spring with their own subtype of Resilience.
  - Their res field is satisfied by the res argument of the case class, but that field is a different subtype in each case.

Another example

```
1  // Constraint.scala
2  import com.atomicscala.AtomicTest._
3
4  class WithF {
5    def f(n:Int) = n * 11
6  }
7
8  class CallF[T <: WithF](t:T) {
9    def g(n:Int) = t.f(n)
10 }
11
12 new CallF(new WithF).g(2) is 22
13
14 new CallF(new WithF {
15   override def f(n:Int) = n * 7
16 }).g(2) is 14
```

- only reason it's possible to call f inside CallF is that the type is constrained to be of class WithF or a subclass of WithF
- On line 12 we pass an instance of WithF
- on lines 14-16 to make an unnamed subclass of WithF inline
  - Line 15 overrides f to give it a new meaning

Building Systems with Traits

- Traits are so independent and low-impact, you can break your problem down into as many small pieces as necessary
- Example: a model of various ingredients that make up different kinds of ice-cream treats
  - Using Enumeration subtype technique, type constraints

```scala
// SodaFountain.scala
package sodafountain

object Quantity extends Enumeration
  type Quantity = Value
  val None, Small, Regular,
    Extra, Super = Value
}
import Quantity._

object Holder extends Enumeration {
  type Holder = Value
  val Bowl, Cup, Cone, WaffleCone =
}
import Holder._

trait Flavor

object Syrup extends Enumeration {
  case class _Val() extends Val
    with Flavor
  type Syrup = _Val
  val Chocolate, HotFudge,
    Butterscotch, Caramel = _Val()
}
import Syrup._

object IceCream extends Enumeration {
  case class _Val() extends Val
    with Flavor
  type IceCream = _Val
  val Chocolate, Vanilla, Strawberry,
    Coffee, MochaFudge, RumRaisin,
    ButterPecan = _Val()
}
import IceCream._

object Sprinkle extends Enumeration {
  case class _Val() extends Val
    with Flavor
  type Sprinkle = _Val
  val None, Chocolate, Rainbow = _Val(
}
import Sprinkle._

trait Amount {
  val quant:Quantity
}
```

```
50 trait Taste[F <: Flavor] extends Amount {
51   val flavor:F
52 }
53
54 case class
55 Scoop(quant:Quantity, flavor:IceCream)
56 extends Taste[IceCream]
57
58 trait Topping
59
60 case class
61 Sprinkles(quant:Quantity, flavor:Sprinkle)
62 extends Taste[Sprinkle] with Topping
63
64 case class
65 Sauce(quant:Quantity, flavor:Syrup)
66 extends Taste[Syrup] with Topping
67
68 case class WhippedCream(quant:Quantity)
69 extends Amount with Topping
70
71 case class Nuts(quant:Quantity)
72 extends Amount with Topping
73
74 class Cherry extends Topping
```

- Flavor and Topping are both tagging traits
- We are trying to :
  - Create a set of types that fit together in a sensible way
  - Configure it so that compiler catches misuse of types
  - Eliminate duplicate code
- “Eliminate code duplication”

Now we can make some ice cream

```
1  // MaltShoppe.scala
2  import com.atomicscala.AtomicTest._
3  import sodafountain._
4  import Quantity._
5  import Holder._
6  import Syrup._
7  import IceCream._
8  import Sprinkle._
9
10 case class
11 Scoops(holder:Holder, scoops:Scoop*)
12
13 val iceCreamCone = Scoops(
14   WaffleCone,
15   Scoop(Extra, MochaFudge),
16   Scoop(Extra, ButterPecan),
17   Scoop(Extra, IceCream.Chocolate))
18
19 iceCreamCone is "Scoops(WaffleCone," +
20 "WrappedArray(Scoop(Extra,MochaFudge)
21 "Scoop(Extra,ButterPecan), " +
22 "Scoop(Extra,Chocolate)))"
```

```
24 case class MadeToOrder(
25   holder:Holder,
26   scoops:Seq[Scoop],
27   toppings:Seq[Topping])
28
29 val iceCreamDish = MadeToOrder(
30   Bowl,
31   Seq(
32     Scoop(Regular, Strawberry),
33     Scoop(Regular, ButterPecan)),
34   Seq[Topping]())
35
36 iceCreamDish is "MadeToOrder(Bowl," +
37 "List(Scoop(Regular,Strawberry), " +
38 "Scoop(Regular,ButterPecan)),List())"
39
40 case class Sundae(
41   sauce:Sauce,
42   sprinkles:Sprinkles,
43   whipped:WhippedCream,
44   nuts:Nuts,
```

```
45    scoops:Scoop*) {
46    val holder:Holder = Bowl
47  }
48
49  val hotFudgeSundae = Sundae(
50    Sauce(Regular, HotFudge),
51    Sprinkles(Regular, Sprinkle.Chocolate),
52    WhippedCream(Regular), Nuts(Regular),
53    Scoop(Regular, Coffee),
54    Scoop(Regular, RumRaisin))
55
56  hotFudgeSundae is "Sundae(" +
57    "Sauce(Regular,HotFudge)," +
58    "Sprinkles(Regular,Chocolate)," +
59    "WhippedCream(Regular),Nuts(Regular)," +
60    "WrappedArray(Scoop(Regular,Coffee), " +
61    "Scoop(Regular,RumRaisin)))"
```

- Scoops, a basic implementation allowing to create a cone or dish of ice cream
- MadeToOrder, adds more possibilities but still generic
- Sundae, quite specific

+

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 concepts

- An object contains encapsulated data and procedures grouped together to represent an entity .
- An object-oriented program is described by the interaction of these objects.
- Information hiding: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*
- Inheritance: The ability for a *class* to extend or override functionality of another *class*. The so-called *subclass* has a whole section that is derived (inherited) from the *superclass* and then it has its own set of functions and data

Object concepts, contd.

- Interface (object-oriented programming): The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them
- Polymorphism: The ability to replace an *object* with its *subobjects*. The ability of an *object-variable* to contain, not only that *object*, but also all of its *subobjects*

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 design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.
- The input for object-oriented design is provided by the output of object-oriented analysis
- Analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process

Input for OOD

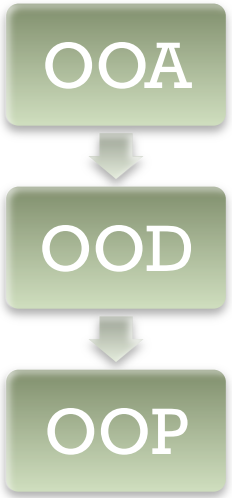
- Conceptual model: The result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details
- Use case: A description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function
- User interface documentations (if applicable): Document that shows and describes the look and feel of the end product's user interface
- Relational data model (if applicable): A data model is an abstract model that describes how data is represented and used.

Output of OOD

- Class diagram: A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes
- Sequence diagram: Extend the system sequence diagram to add specific objects that handle the system events.
  - A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur

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

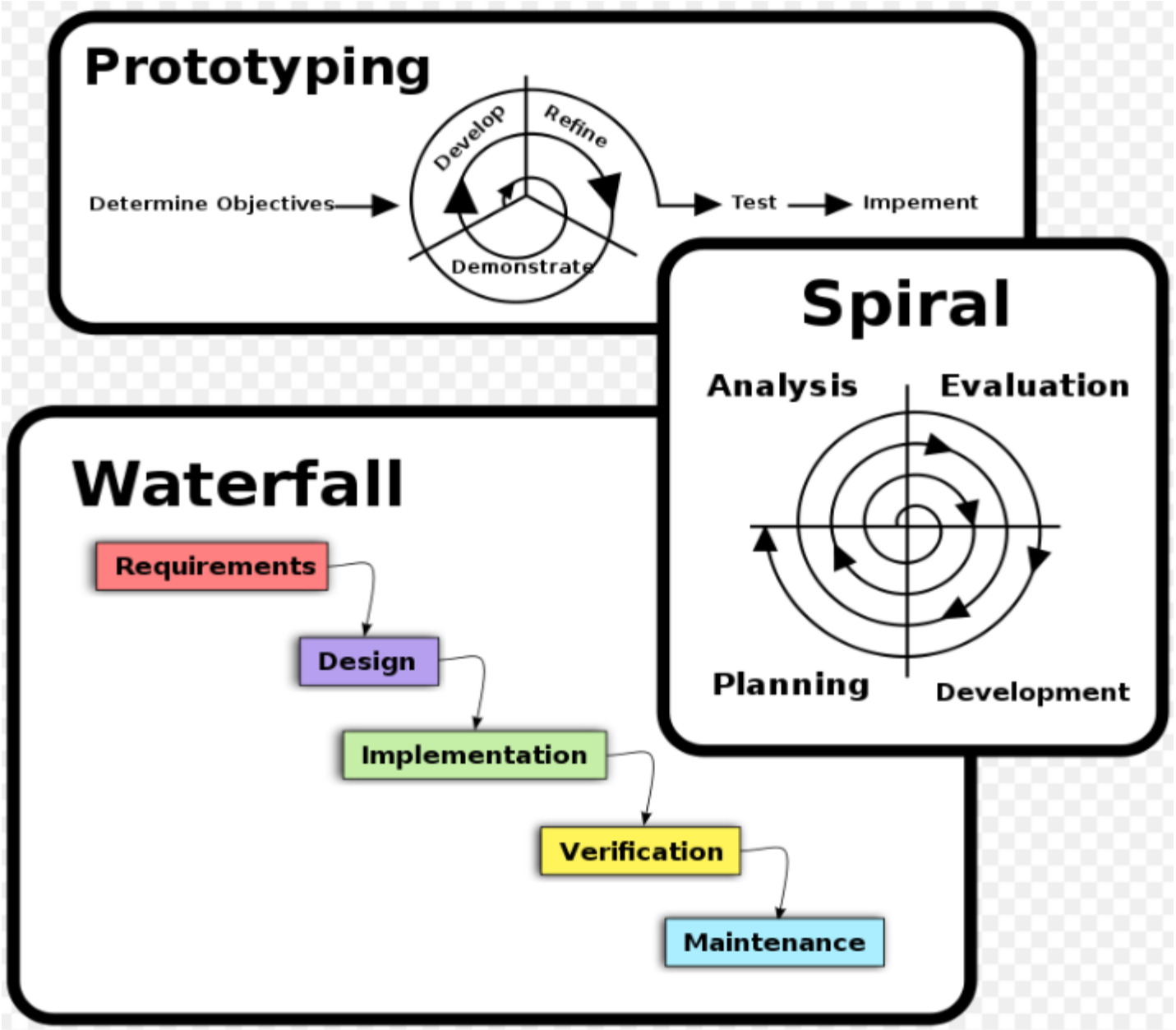
- The purpose of any analysis activity in the software life-cycle is to create a model of the system's functional requirements that is independent of implementation constraints

- The main difference between object-oriented analysis and other forms of analysis is that by the object-oriented approach we organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with. In other or traditional analysis methodologies, the two aspects: processes and data are considered separately. For example, data may be modeled by ER diagrams, and behaviors by flow charts or structure charts.

Primary tasks in OOA

- Find the objects
- Organize the objects
- Describe how the objects interact
- Define the behavior of the objects
- Define the internals of the objects
- Common models used in OOA are use cases and object models. Use cases describe scenarios for standard domain functions that the system must accomplish.
- Object models describe the names, class relations (e.g. Circle is a subclass of Shape), operations, and properties of the main objects

three basic approaches applied to software development methodology frameworks



Object-oriented Programming

- **Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods*
- A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated
- computer programs are designed by making them out of objects that interact with one another
- most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type

Criticism

- The OOP paradigm has been criticised for a number of reasons, including not meeting its stated goals of reusability and modularity,[36][37] and for overemphasizing one aspect of software design and modeling (data/objects) at the expense of other important aspects (computation/algorithms)

- OOP languages have "extremely poor modularity properties with respect to class extension and modification", and tend to be extremely complex.[36] The latter point is reiterated by Joe Armstrong, the principal inventor of Erlang, who is quoted as saying:[37]
  - The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

- In an article Lawrence Krubner claimed that compared to other languages (lisps, functional languages, etc) OOP languages have no unique strengths, and inflict a heavy burden of unneeded complexity

- Paul Graham has suggested that OOP's popularity within large companies is due to "large (and frequently changing) groups of mediocre programmers." According to Graham, the discipline imposed by OOP prevents any one programmer from "doing too much damage."[

- Steve Yegge noted that, as opposed to functional programming:[45]
  - Object Oriented Programming puts the Nouns first and foremost. Why would you go to such lengths to put one part of speech on a pedestal? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think. It's a strangely skewed perspective
- Rich Hickey, creator of Clojure, described object systems as overly simplistic models of the real world. He emphasized the inability of OOP to model time properly, which is getting increasingly problematic as software systems become more concurrent
- Eric S. Raymond, a Unix programmer and open-source software advocate, has been critical of claims that present object-oriented programming as the "One True Solution," and has written that object-oriented programming languages tend to encourage thickly-layered programs that destroy transparency.[46] Raymond contrasts this to the approach taken with Unix and the C programming language

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.

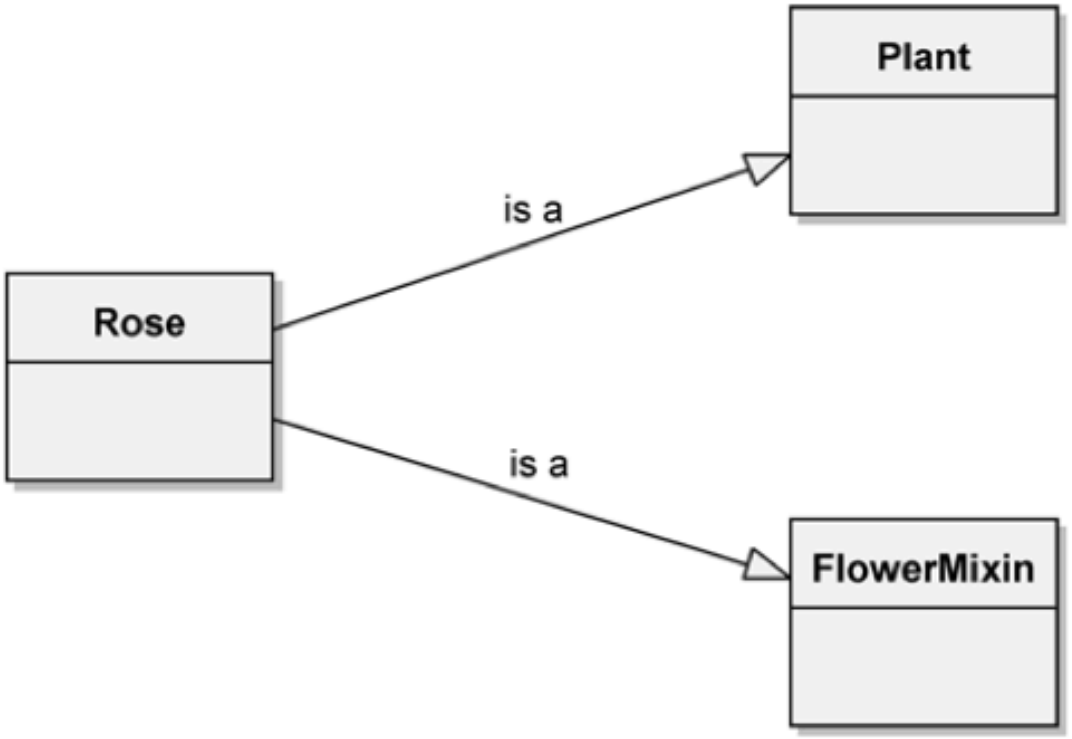
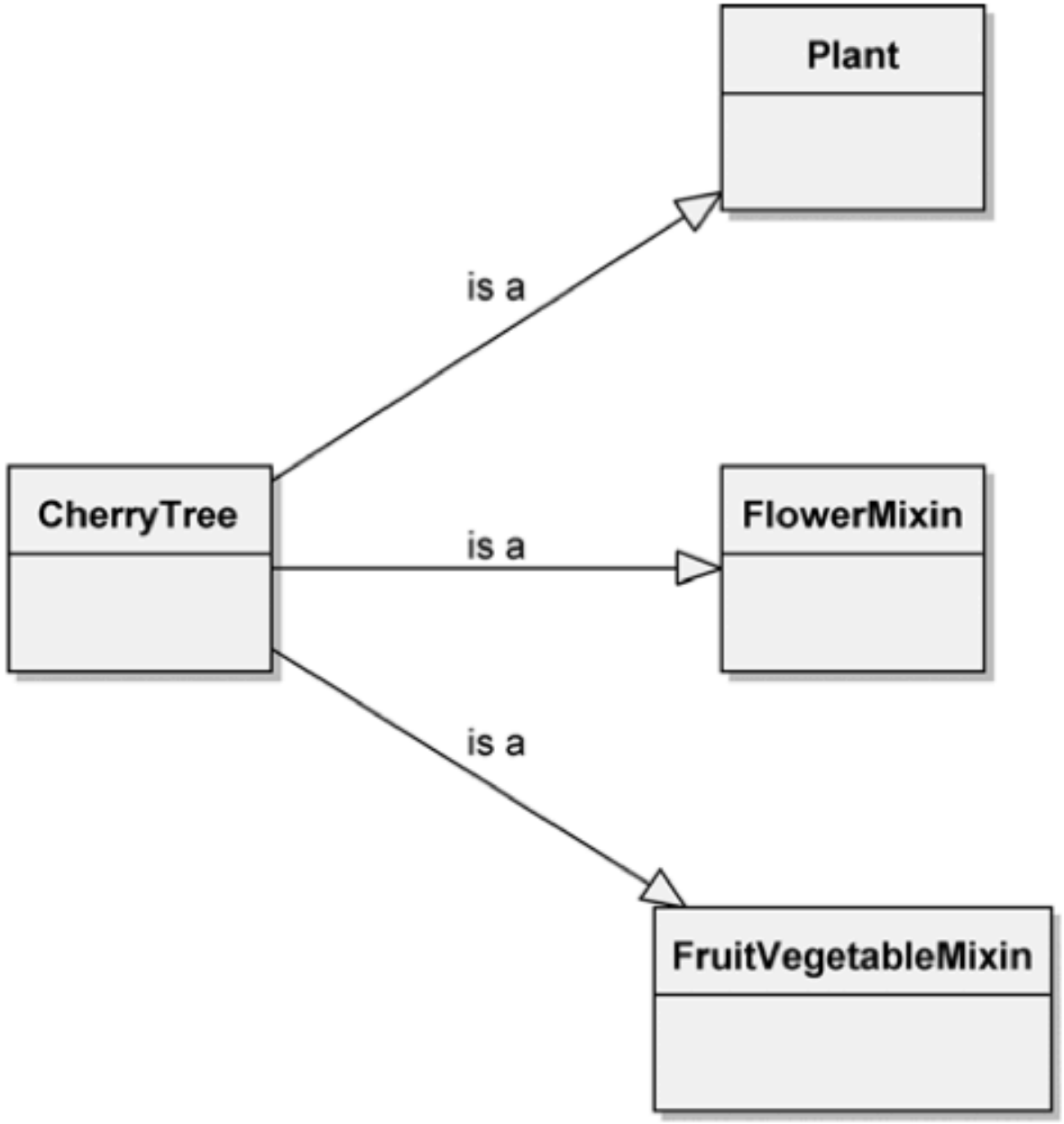


Figure 2–10 The `Rose` Class, Which Inherits from Multiple Superclasses



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

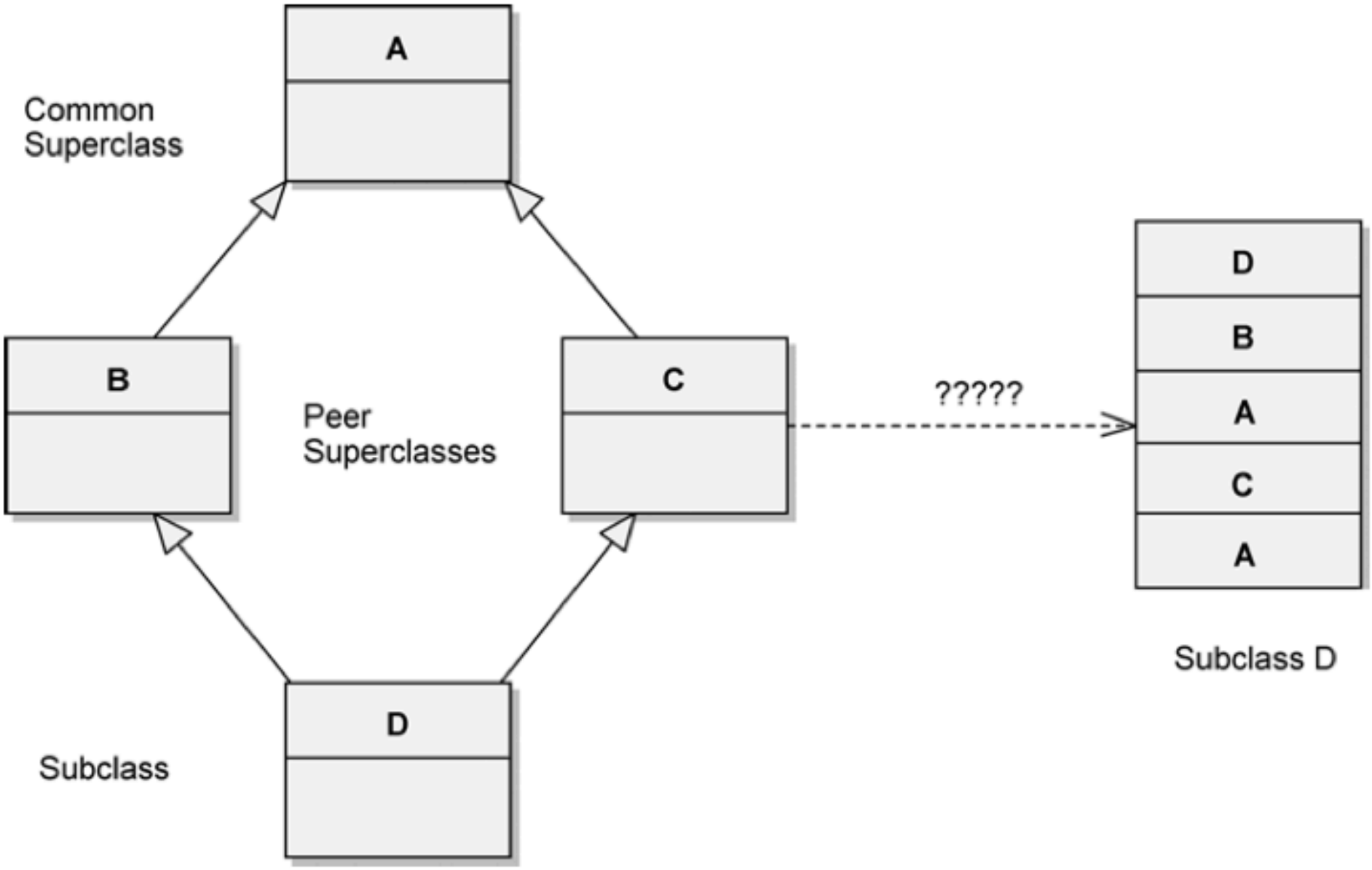


Figure 2–13 The Repeated Inheritance Problem

Examples of Hierarchy: Aggregation

- "is a" hierarchies denote generalization/specialization relationships,
- "part of" hierarchies describe aggregation relationships.
  - E.g. , a garden consists of a collection of plants with a growing plan
  - i.e. , plants are "part of " the garden, growing plan is "part of " garden
- combination of inheritance with aggregation is powerful: Aggregation permits the physical grouping of logically related structures, and inheritance allows these common groups to be easily reused among different abstractions.

Aggregation raises issue of ownership

- replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants
- lifetime of a garden and its plants are independent
- In contrast, GrowingPlan object is intrinsically associated with a Garden object and does not exist independently

Meaning of Typing

- concept of a type derives primarily from the theories of abstract data types
- Deutsch suggests, "A type is a precise characterization of structural or behavioral properties which a collection of entities all share"
- Though type and class similar, type places a different emphasis on meaning of abstraction
  - Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways

- Strongly typed, weakly typed, untyped
- Strong typing enforces certain design decisions; but introduces semantic dependencies such that even small changes in interface of a base class require recompilation of all subclasses
- Two general solutions
  - First, we could use a type-safe container class that manipulates only objects of a specific class. This approach addresses the first problem, wherein objects of different types are incorrectly mingled
  - Second, we could use some form of runtime type identification; this addresses the second problem of knowing what kind of object you happen to be examining at the moment.

Benefits using strongly typed languages

- Without type checking, a program in most languages can 'crash' in mysterious ways at runtime.
- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
- Type declarations help to document programs.
- Most compilers can generate more efficient object code if types are declared. [72]

Static and dynamic typing

- Strong and weak typing refers to *type consistency, whereas* static and dynamic typing refers to the *time when names are bound to types*
- Static typing (also known as *static binding or early binding) means that the types of all* variables and expressions are fixed at the time of compilation;
- dynamic typing (also known as *late binding) means that the types of all variables and expressions* are not known until runtime
- A language may be both strongly and statically typed (Ada), strongly typed yet supportive of dynamic typing (C++, Java), or untyped yet supportive of dynamic typing (Smalltalk).

- *Polymorphism is a condition that exists when the features of dynamic typing and* inheritance interact.
- Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass
- opposite of polymorphism is *monomorphism, which is found in all languages that* are both strongly and statically typed
- Polymorphism is the most powerful feature of object-oriented programming languages next to their support for abstraction
  - is what distinguishes object-oriented programming from more traditional programming with abstract data types.

Meaning of Concurrency

- an automated system may have to handle many different events simultaneously. Other problems may involve so much computation that they exceed the capacity of any single processor
- natural to consider using a distributed set of computers for the target implementation or to use multitasking
- System involving concurrency may have many threads of control, some are transitory, others last entire lifetime of the system's execution
- Systems across multiple CPUs allow for truly concurrent threads of control, whereas running on a single CPU only achieve illusion of concurrent threads of control, by means of time-slicing algorithm

- designing one that encompasses multiple threads of control is much harder because one must worry about such issues as deadlock, livelock, starvation, mutual exclusion, and race conditions.
- Black et al. therefore suggest that "an object model is appropriate for a distributed system because it implicitly defines (1) the units of distribution and movement and (2) the entities that communicate" [77]
- Object-oriented programming focuses on data abstraction, encapsulation, and inheritance, concurrency focuses on process abstraction and synchronization [78]

- Active Objects
  - object is a concept that unifies these two different viewpoints: Each object (drawn from an abstraction of the real world) may represent a separate thread of control (a process abstraction).
- In a system based on an object-oriented design, we can conceptualize the world as consisting of a set of cooperative objects, some of which are active and thus serve as centers of independent activity
- Define concurrency as follows:
  - Concurrency is the property that distinguishes an active object from one that is not active

Approaches to concurrency in OO-design

- First, concurrency is an intrinsic feature of certain programming languages, which provide mechanisms for concurrency and synchronization. So can use it to create an active object.
- Second, we may use a class library that implements some form of lightweight processes. Naturally, the implementation of this kind is highly platform-dependent, although the interface to the library may be relatively portable
- Third, we may use interrupts to give us the illusion of concurrency.
  - knowledge of certain low-level hardware details
  - might have a hardware timer that periodically interrupts the application

- you must consider how active objects synchronize their activities with one another as well as with objects that are purely sequential
  - E.g. , if two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of the object being acted on is not corrupted
  - This is the point where the ideas of abstraction, encapsulation, and concurrency interact
- In the presence of concurrency, it is not enough simply to define the methods of an object; we must also make certain that the semantics of these methods are preserved in the presence of multiple threads of control.

Meaning of Persistence

- An object in software takes up some amount of space and exists for a particular amount of time
- Spectrum of object persistence encompasses the following:

  - Transient results in expression evaluation
  - Local variables in procedure activations
  - Own variables [as in ALGOL 60], global variables, and heap items whose extent is different from their scope
  - Data that exists between executions of a program
  - Data that exists between various versions of a program
  - Data that outlives the program [79]

- "Data that outlives the program" is the case of Web applications where the application may not be connected to the data it is using through the entire transaction execution.
- introducing the concept of persistence to the object model gives rise to object-oriented databases
  - offer to the programmer the abstraction of an object-oriented interface, through which database queries and other operations are completed in terms of objects whose lifetimes transcend the lifetime of an individual program.
  - it allows us to apply the same design methods to the database and nondatabase segments of an application

- Some object-oriented programming languages provide direct support for persistence.
  - Java provides Enterprise Java Beans (EJBs) and Java Data Objects.
- However, streaming objects to flat files is a naive solution to persistence that does not scale well.
- typical approach to persistence is to provide an object-oriented skin over a relational database
- Customized object-relational mappings can be created by the individual developer, but challenging to do well
  - Frameworks available to ease this task, e.g., Hibernate

- for systems that execute on a distributed set of processors, we must sometimes be concerned with persistence across space
- Define persistence as follows:
  - Persistence is the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created).

Benefits of Object Model

- First, the use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages.
- Second, the use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks
- Third, the use of the object model produces systems that are built on stable intermediate forms, which are more resilient to change.
- Fourth, object model's guidance in designing an intelligent separation of concerns also reduces development risk and increases our confidence in the correctness of our design.