The Object Model

Introduction

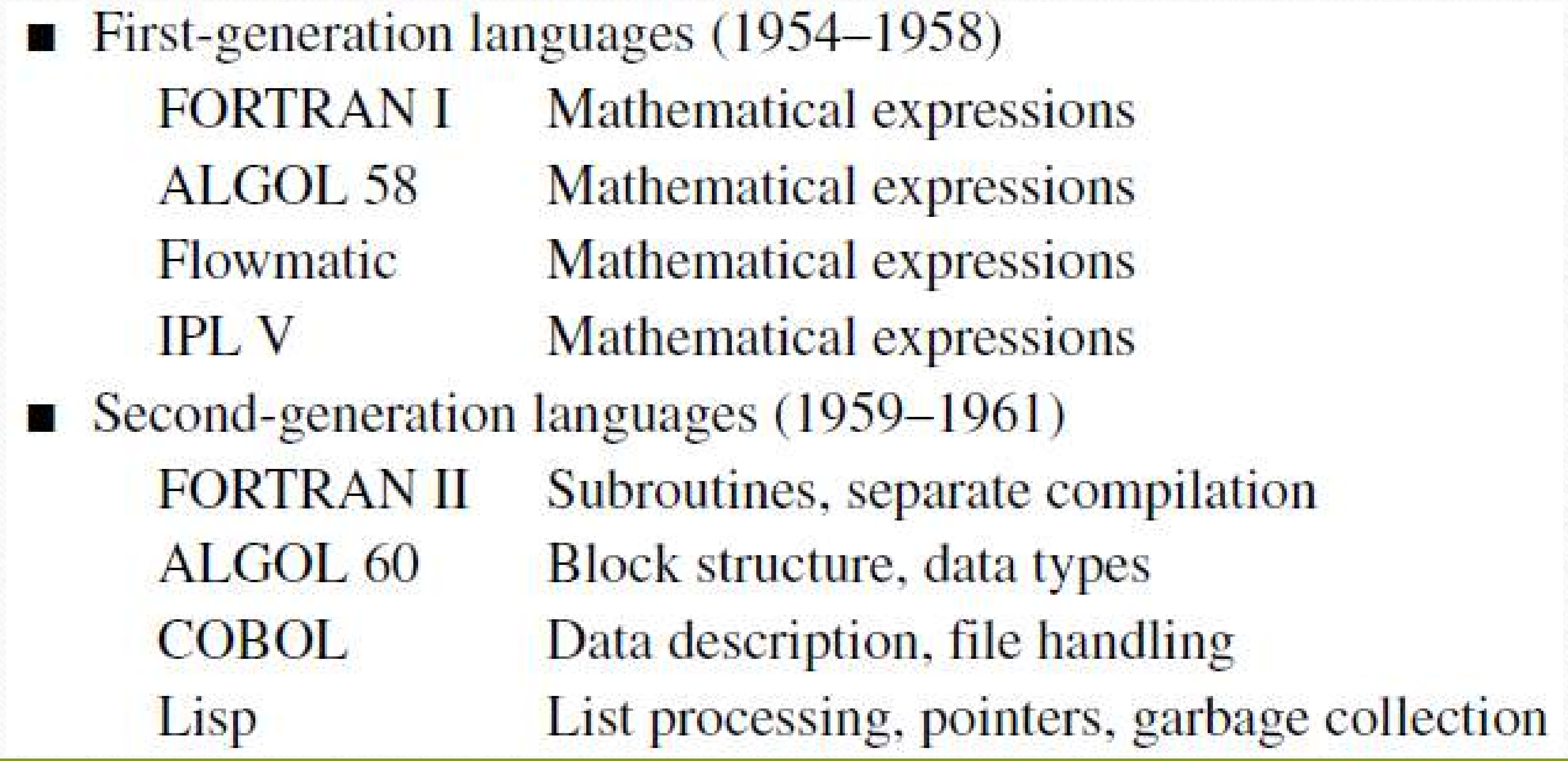
- Object-oriented technology is built on a sound engineering foundation, whose elements we collectively call as the object model of development or simply the object model.
- The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.
- By themselves, none of these principles are new.
- What is important about the object model is that these elements are brought together in a synergistic way.

The Evolution of the Object Model

- Object-oriented development is founded in the best ideas from prior technologies.
- Two sweeping trends in the history of software engineering :

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

The Generations of Programming Languages



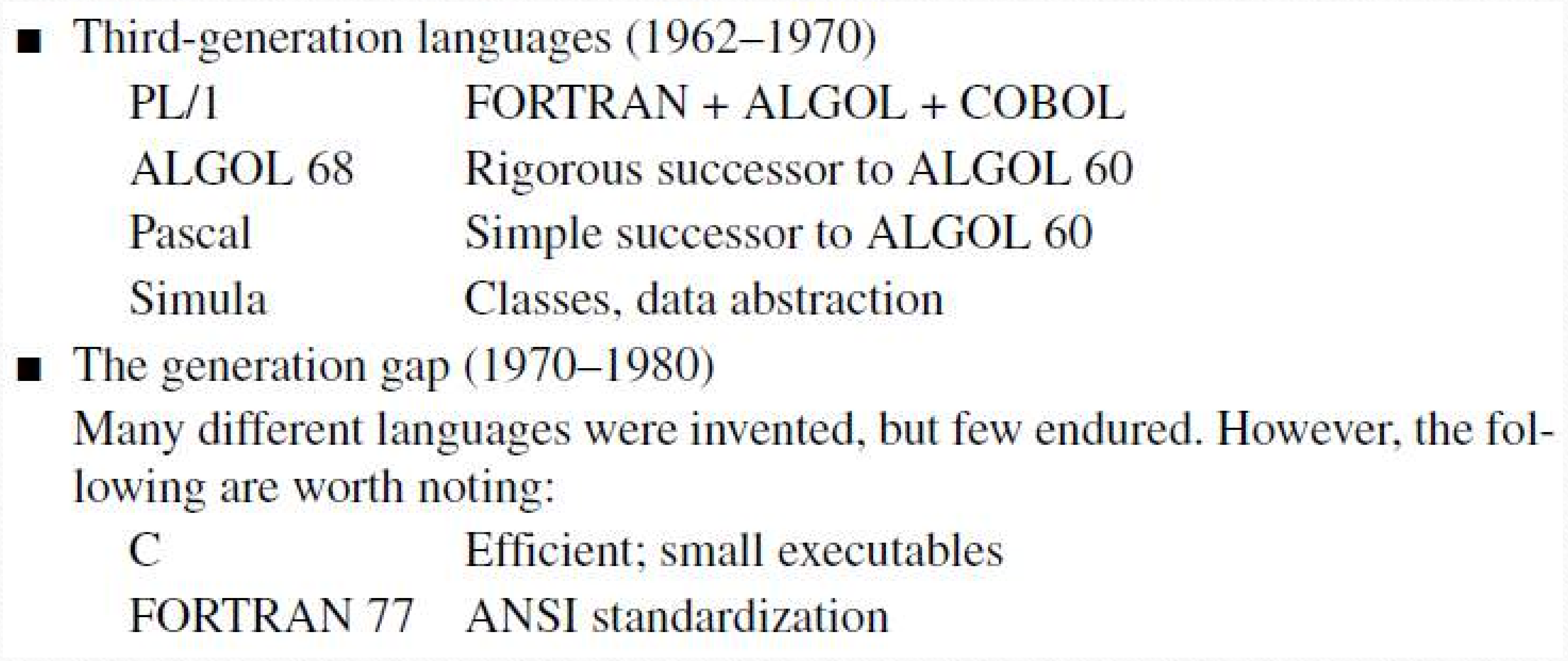
- First-generation languages (1954–1958)

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

The Generations of Programming Languages



- Third-generation languages (1962–1970)

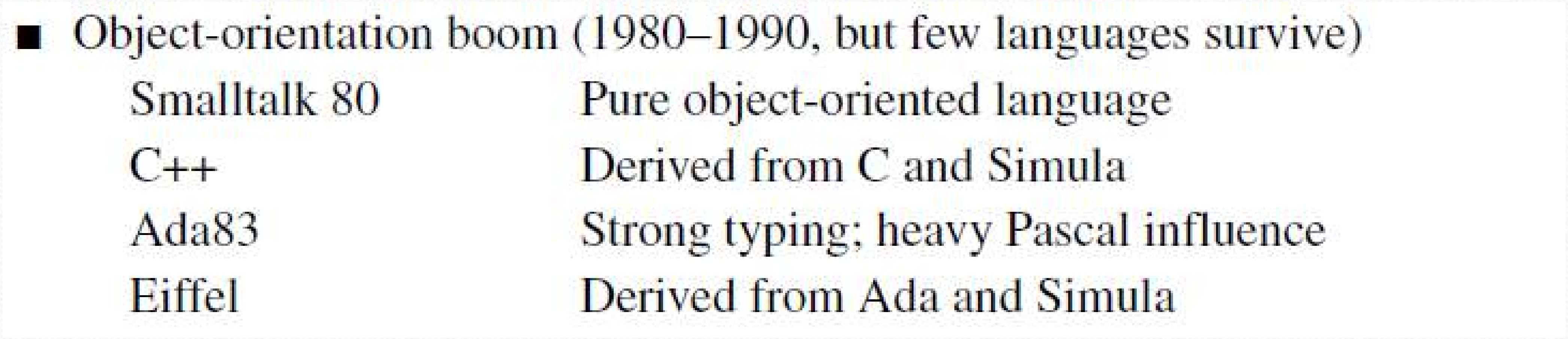
PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	Rigorous successor to ALGOL 60
Pascal	Simple successor to ALGOL 60
Simula	Classes, data abstraction

- The generation gap (1970–1980)

  Many different languages were invented, but few endured. However, the following are worth noting:

C	Efficient; small executables
FORTRAN 77	ANSI standardization

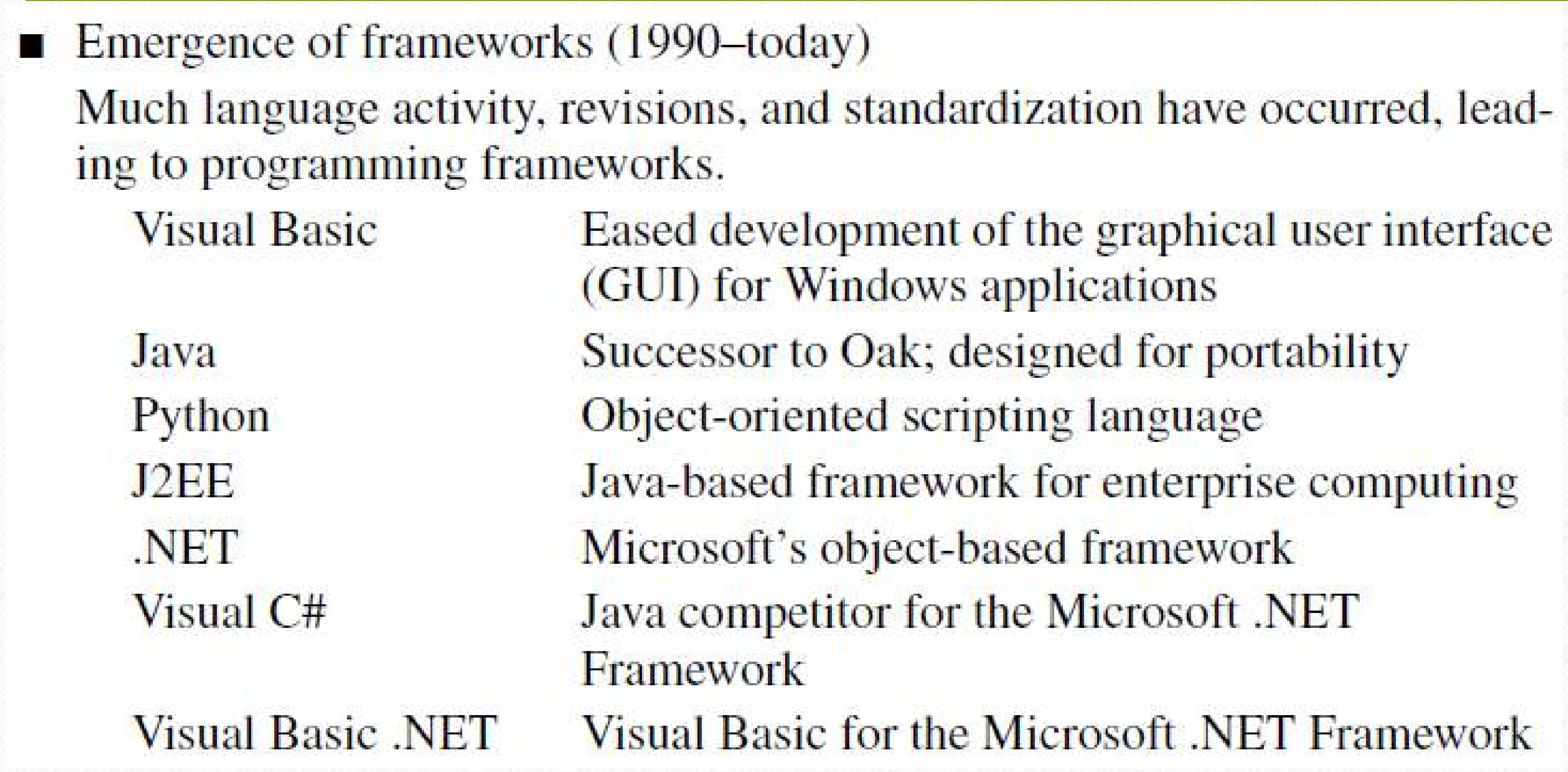
The Generations of Programming Languages



■ 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

The Generations of Programming Languages

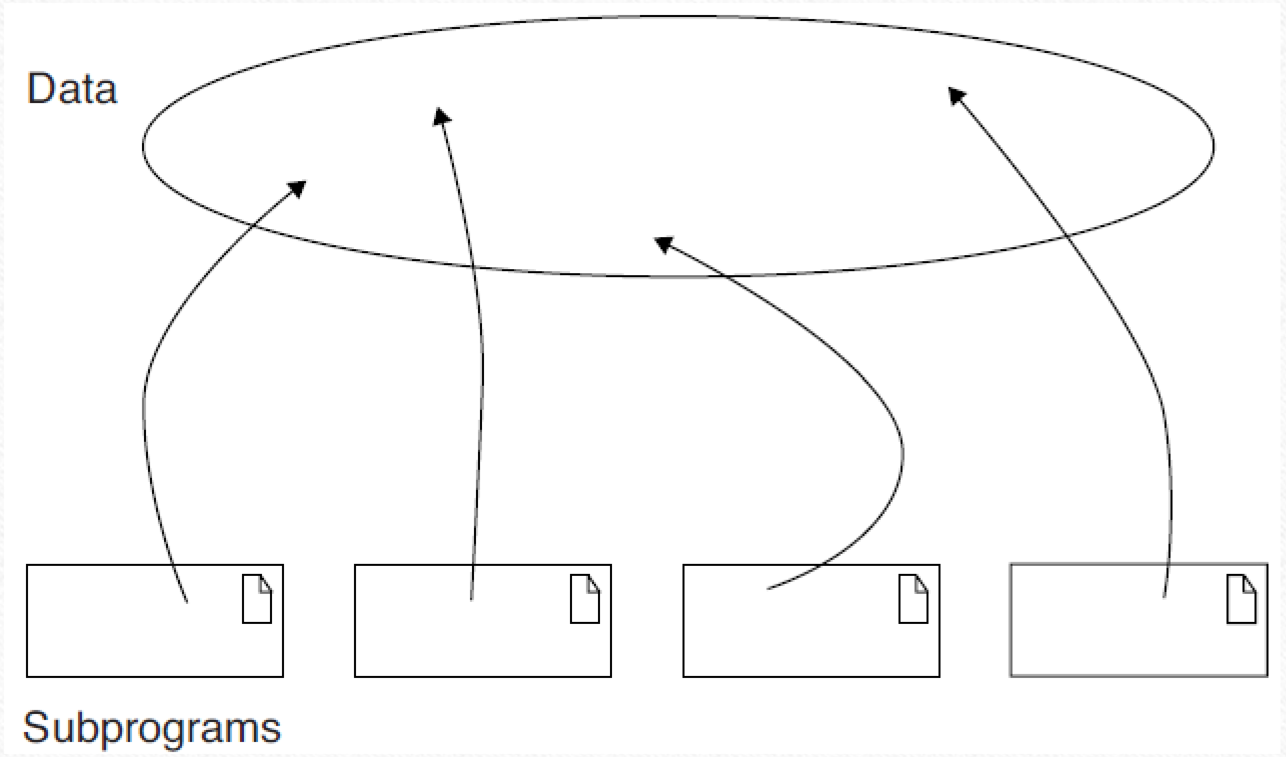


- Emergence of frameworks (1990–today)

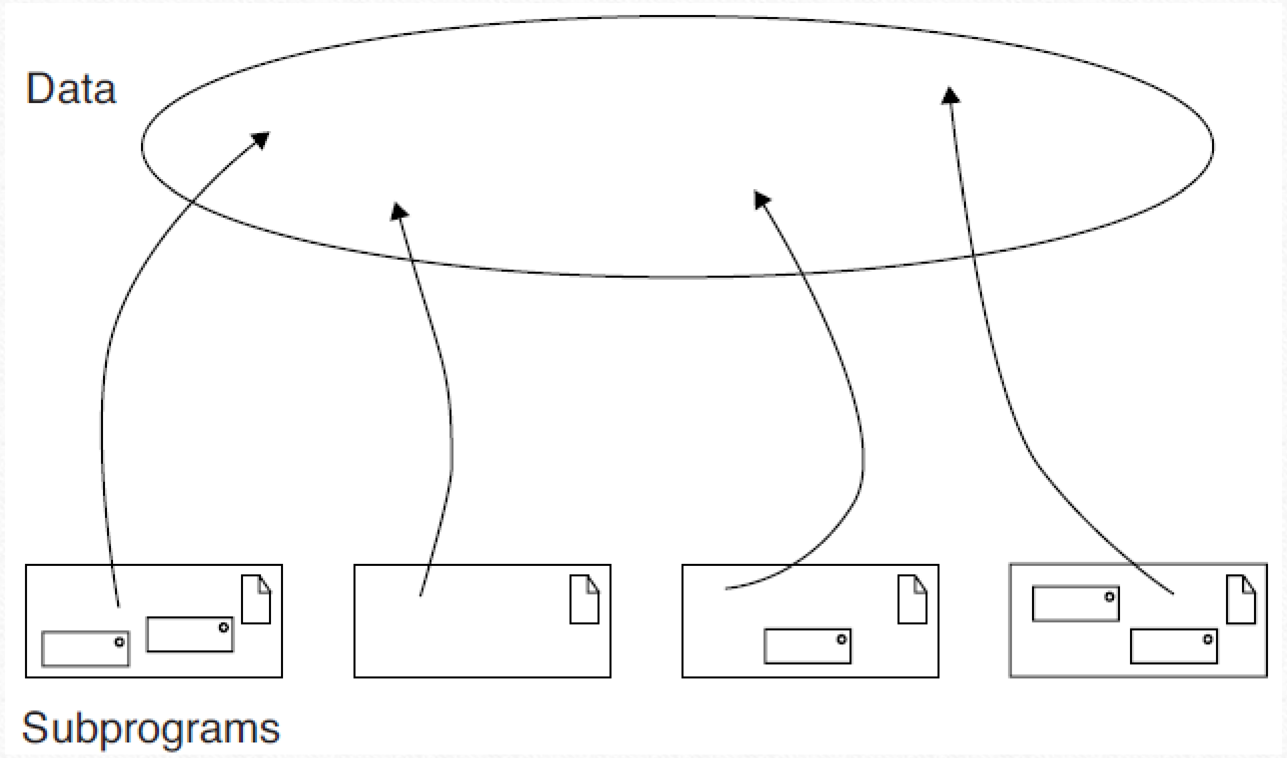
  Much language activity, revisions, and standardization have occurred, leading to programming frameworks.

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

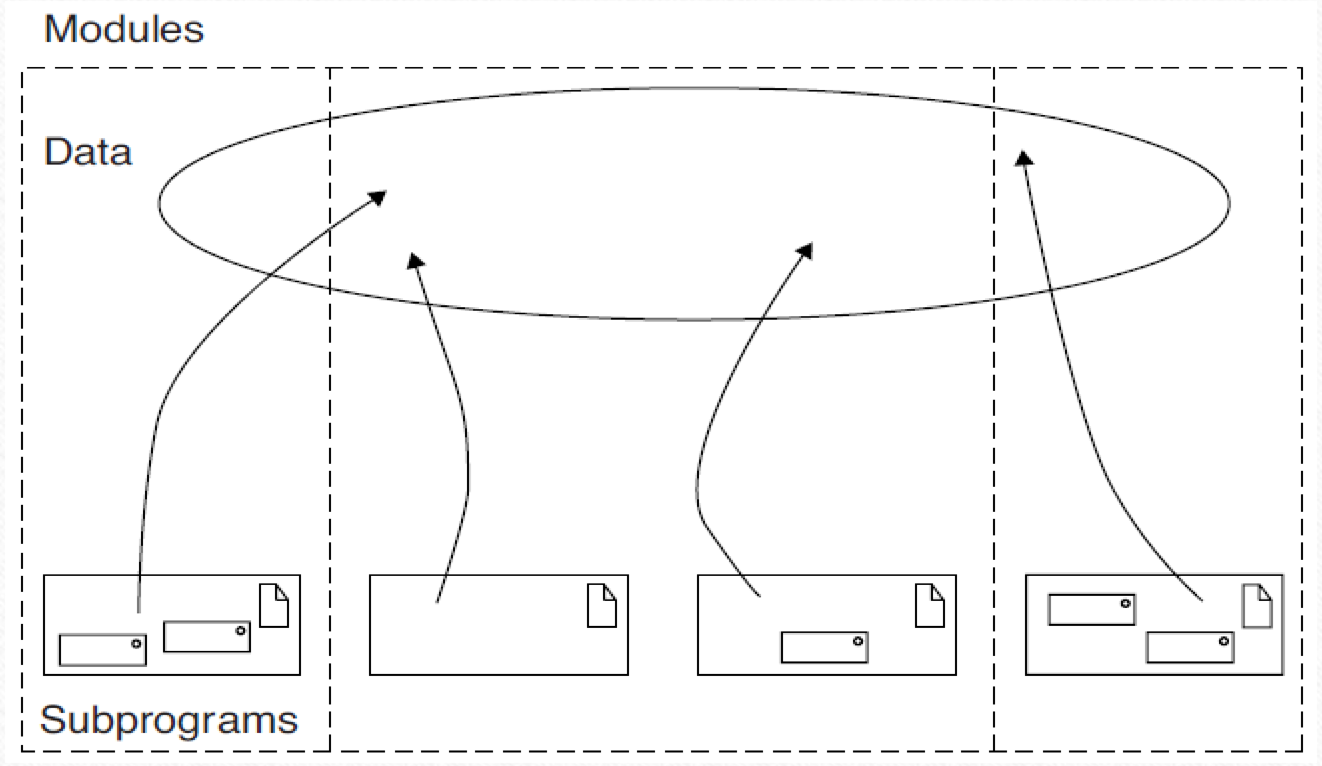
The Topology of First and Early Second Generation Programming Languages



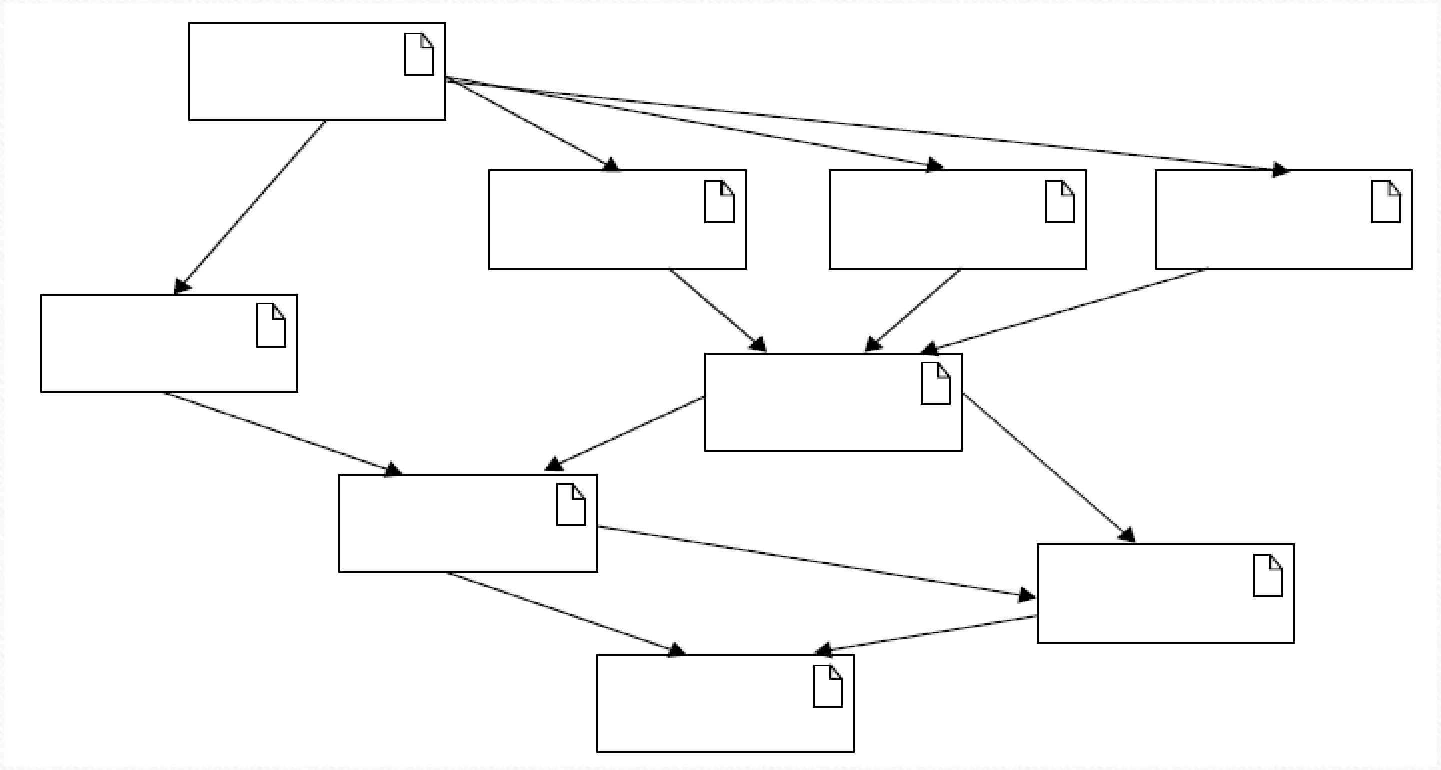
The Topology of Late Second and Early Third Generation Programming Languages



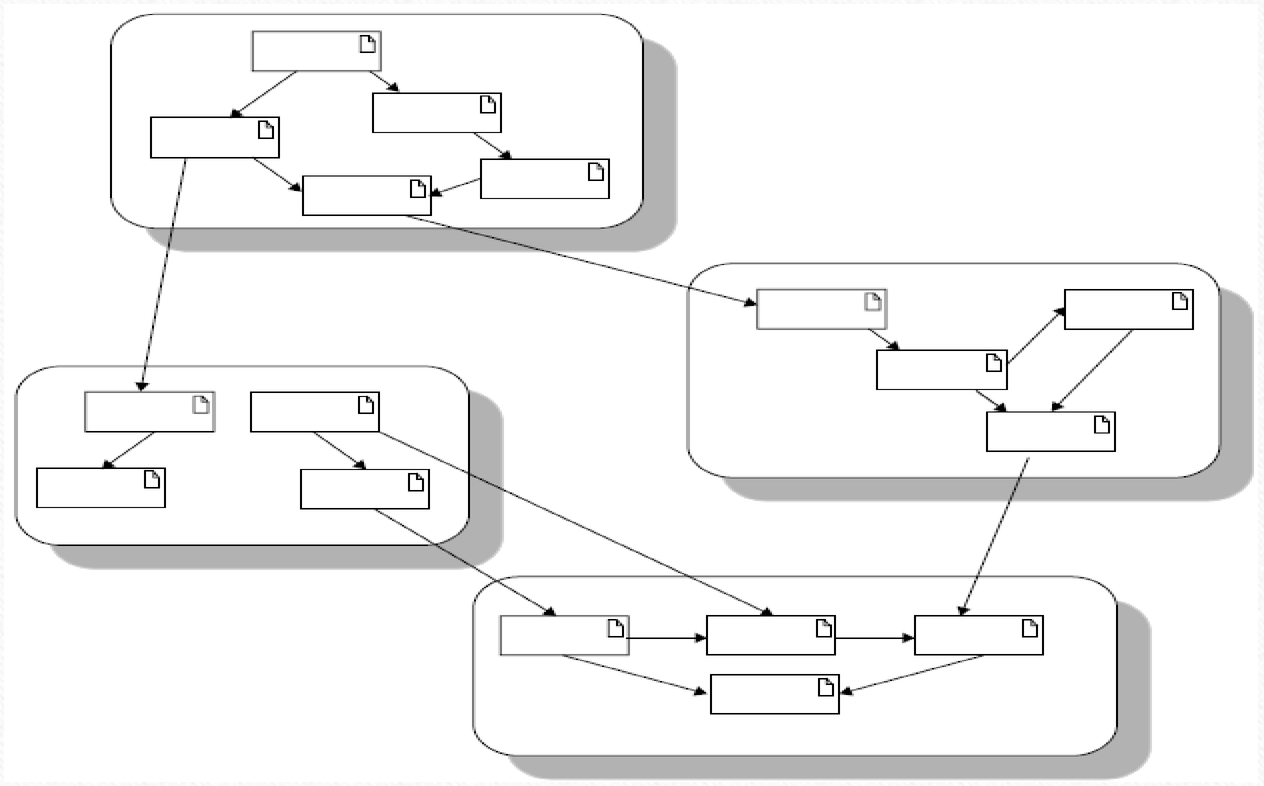
The Topology of Late Third-Generation Programming Languages



The Topology of Object-Based and Object-Oriented Programming Languages



The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages



Foundations of the Object Model

- Structured design methods evolved to guide developers in building complex systems using **algorithms as their fundamental building blocks**.
- Object-oriented design methods have evolved to help developers exploit the expressive power of object-based and object-oriented programming languages, using the **class and object as basic building blocks**.

Foundations of the Object Model

- The Object Model has proven to be a unifying concept in computer science, applicable not just to programming languages but also to the design of user interfaces, databases, and even computer architectures.
- The reason for this widespread appeal is simply that an object orientation helps us to cope with the complexity inherent in many different kinds of systems.
- Object-oriented analysis and design represents an evolutionary development, as it does not break with advances from the past but builds on proven ones.

Foundations of the Object Model

- Following events have contributed to the evolution of object-oriented concepts:

1. Advances in computer architecture, including capability systems and hardware support for operating systems concepts.
2. Advances in programming languages, as demonstrated in Simula, Smalltalk, CLU, and Ada.
3. Advances in programming methodology, including modularization and information hiding.

Foundations of the Object Model

- We would add to this list three more contributions to the foundation of the object model:

1. Advances in database models
2. Research in artificial intelligence
3. Advances in philosophy and cognitive science

Object-Oriented Programming

"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."

Object-Oriented Programming

- There are three important parts to this definition

1. Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks.
2. Each object is an instance of some class.
3. Classes may be related to one another via inheritance relationships.

Object-Oriented Programming

- A program may appear to be object-oriented, but if any of these elements is missing, it is not an object-oriented program.
- Specifically, programming without inheritance is distinctly not object oriented; that would merely be programming with abstract data types.

Object-Oriented Programming

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

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

Object-Oriented Programming

- For a language to support inheritance means that it is possible to express "is a" relationships among types.
- Example: A red rose is a kind of flower, and a flower is a kind of plant.
- If a language does not provide direct support for inheritance, then it is not object-oriented.
- Such languages are called **object-based** rather than **object-oriented**.

Object-Oriented Programming

- The important features of object–oriented programming are:

1. Bottom–up approach in program design
2. Programs organized around objects, grouped in classes
3. Focus on data with methods to operate upon object's data
4. Interaction between objects through functions
5. Reusability of design through creation of new classes by adding features to existing classes.

Object-Oriented Design

"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

- There are two important parts to this definition: object-oriented design
- (1) leads to an object-oriented decomposition.
- (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system.

Object-Oriented Design

- The support for object-oriented decomposition is what makes **object-oriented design** quite different from **structured design**.
- The former uses class and object abstractions to logically structure systems, and the latter uses algorithmic abstractions.
- The term object-oriented design is used to refer to any method that leads to an object-oriented decomposition.

Object-Oriented Design

- The implementation details generally include:

1. Restructuring the class data (if necessary)
2. Implementation of methods, i.e., internal data structures and algorithms
3. Implementation of control
4. Implementation of associations.

Object-Oriented Analysis

- The object model has influenced even earlier phases of the software development lifecycle.
- Traditional structured analysis techniques focus on the flow of data within a system.
- Object-oriented analysis (OOA) emphasizes the building of real-world models, using an object-oriented view of the world.

Object-Oriented Analysis

"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"

Object-Oriented Analysis

- The products of object-oriented analysis serve as the models from which we may start an object-oriented design.
- The products of object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods.

Object-Oriented Analysis

- The primary tasks in object-oriented analysis (OOA) are:

1. Identifying objects
2. Organizing the objects by creating object model diagram
3. Defining the internals of the objects, or object attributes
4. Defining the behavior of the objects, i.e., object actions
5. Describing how the objects interact.

Elements of the Object Model

Elements of the Object Model

- There are five main kinds of programming styles, listed here with the kinds of abstractions they employ:
- Procedure-oriented → Algorithms
- Object-oriented → Classes and objects
- Logic-oriented → Goals, often expressed in a predicate calculus
- Rule-oriented → If–then rules
- Constraint-oriented → Invariant relationships

Elements of the Object Model

- There are four major elements in the Object Model:
  1. **Abstraction**
  2. **Encapsulation**
  3. **Modularity**
  4. **Hierarchy**
- By major, we mean that a model without any one of these elements is not object oriented.

Elements of the Object Model

- There are three minor elements of the Object Model:
  1. **Typing**
  2. **Concurrency**
  3. **Persistence**
- By minor, we mean that each of these elements is a useful, but not essential, part of the object model.

Abstraction

- Abstraction is one of the fundamental ways that we as humans cope with complexity.
- Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences.
- **Definition 1: "A Simplified Description, or Specification, of a System that Emphasizes some of the System’s Details or Properties while Suppressing Others”.**

Abstraction

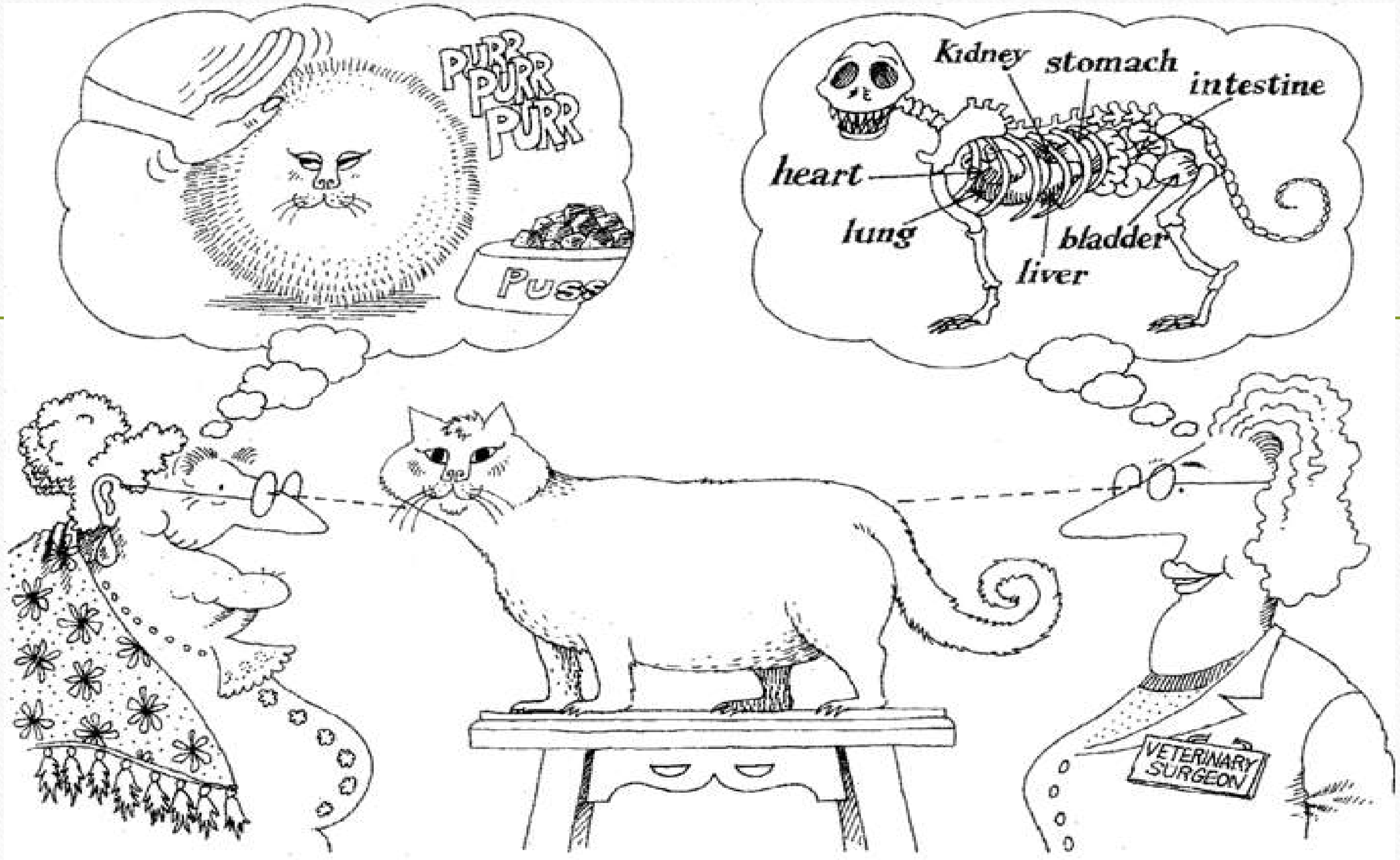
- A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial.
- **Definition 2: "A Concept Qualifies as an Abstraction only if it can be Described, Understood, and Analyzed Independently of the Mechanism that will Eventually be Used to Realize It".**

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.

Abstraction

- An abstraction focuses on the outside view of an object and so serves to separate an object's essential behavior from its implementation.
- Additional principle that we call the principle of least astonishment, through which an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction.



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Abstraction

- Different Kinds of Abstractions:
  1. **Entity Abstraction:** An object that represents a useful model of a problem domain or solution domain entity.
  2. **Action Abstraction:** An object that provides a generalized set of operations, all of which perform the same kind of function.

Abstraction

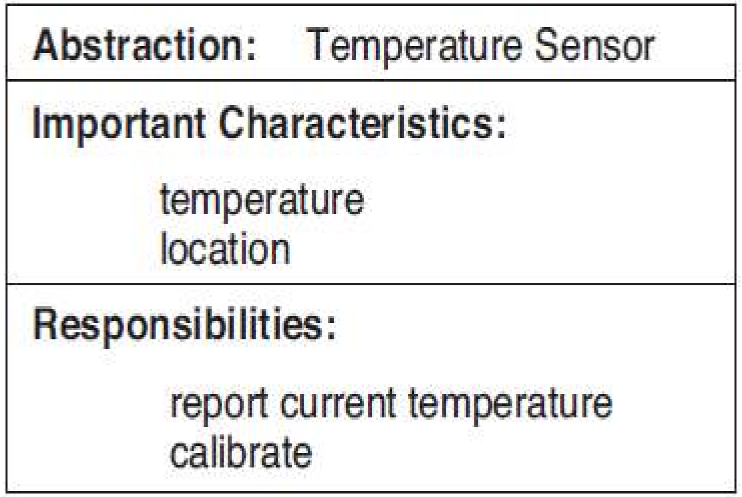
- Different Kinds of Abstractions:

3. **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.
4. **Coincidental Abstraction:** An object that packages a set of operations that have no relation to each other.

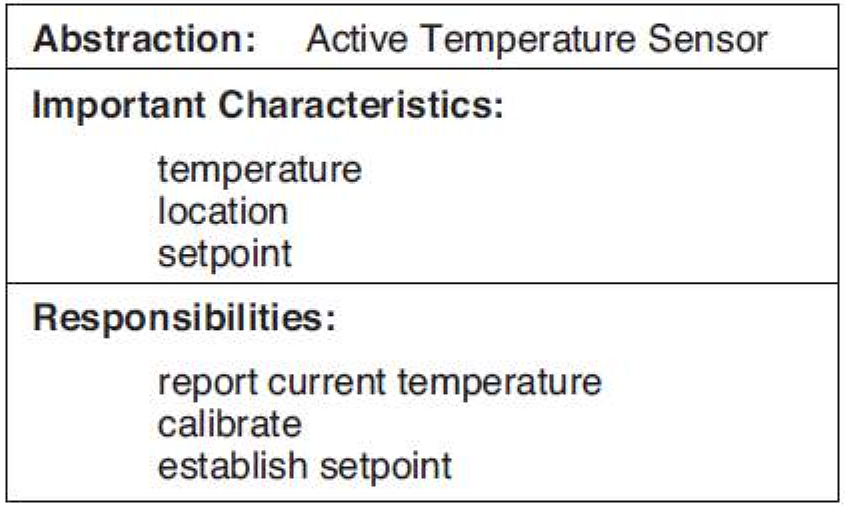
Example of Abstraction: Hydroponics farm

- Plants are grown in a nutrient solution, without sand, gravel, or other soils.
- Maintaining the proper greenhouse environment is a delicate job and depends on the kind of plant being grown and its age.
- One must control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations.
- Key Abstractions in this problem is that of a **sensor**.

Example of Abstraction: Hydroponics farm



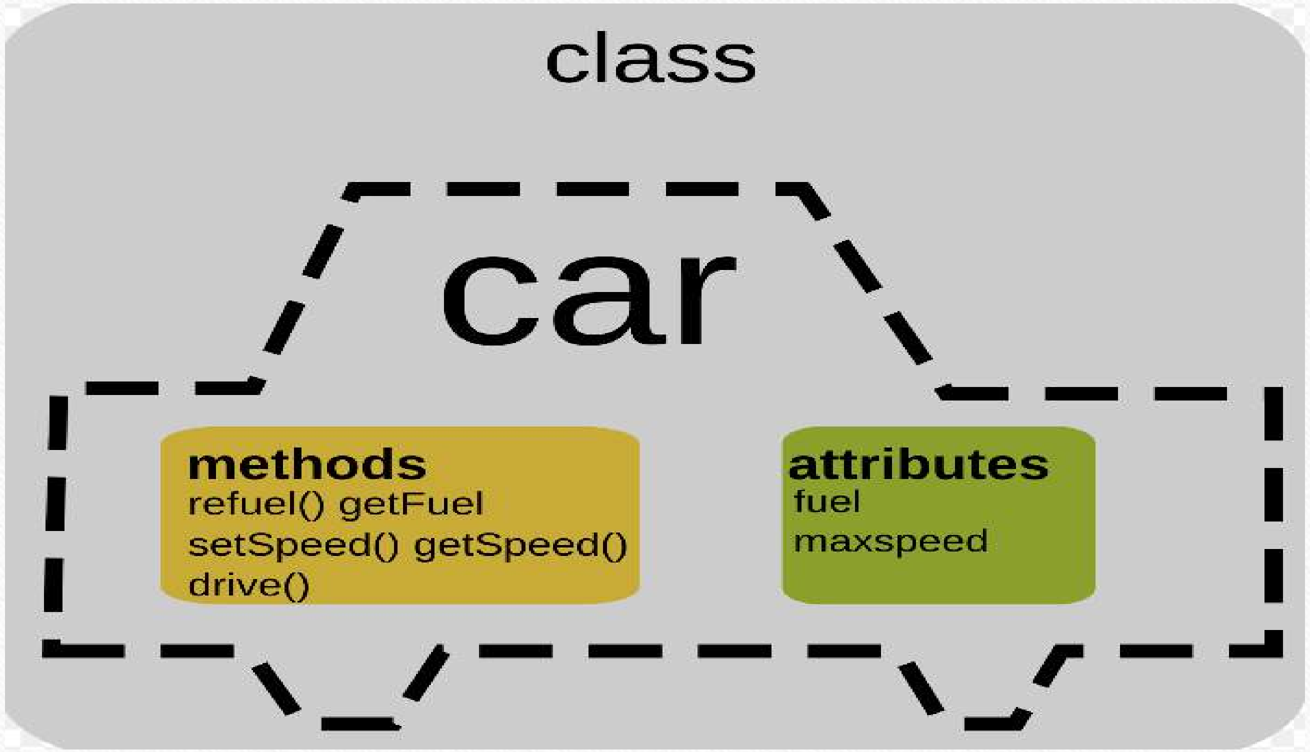
Abstraction: Temperature Sensor
Important Characteristics: temperature, location
Responsibilities: report current temperature, calibrate



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

- **Passive;** some client object must operate on an air Temperature Sensor, object to determine its current temperature to calibrate it.
- **Active:** A client of this abstraction may invoke an operation to establish a critical range of temperatures. It is then the responsibility of the sensor to report whenever the temperature at its location drops below or rises above the given setpoint.

Abstraction



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, which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics.
- Typically, the structure of an object is hidden, as well as the implementation of its methods.

Encapsulation

- For abstraction to work, implementations must be encapsulated.
- Each class must have two parts: an interface and an implementation.
- The interface of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class.
- The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior.
- The interface of a class is the one place where we assert all of the assumptions that a client may make about any instances of the class.
- The implementation encapsulates details about which no client may make assumptions.

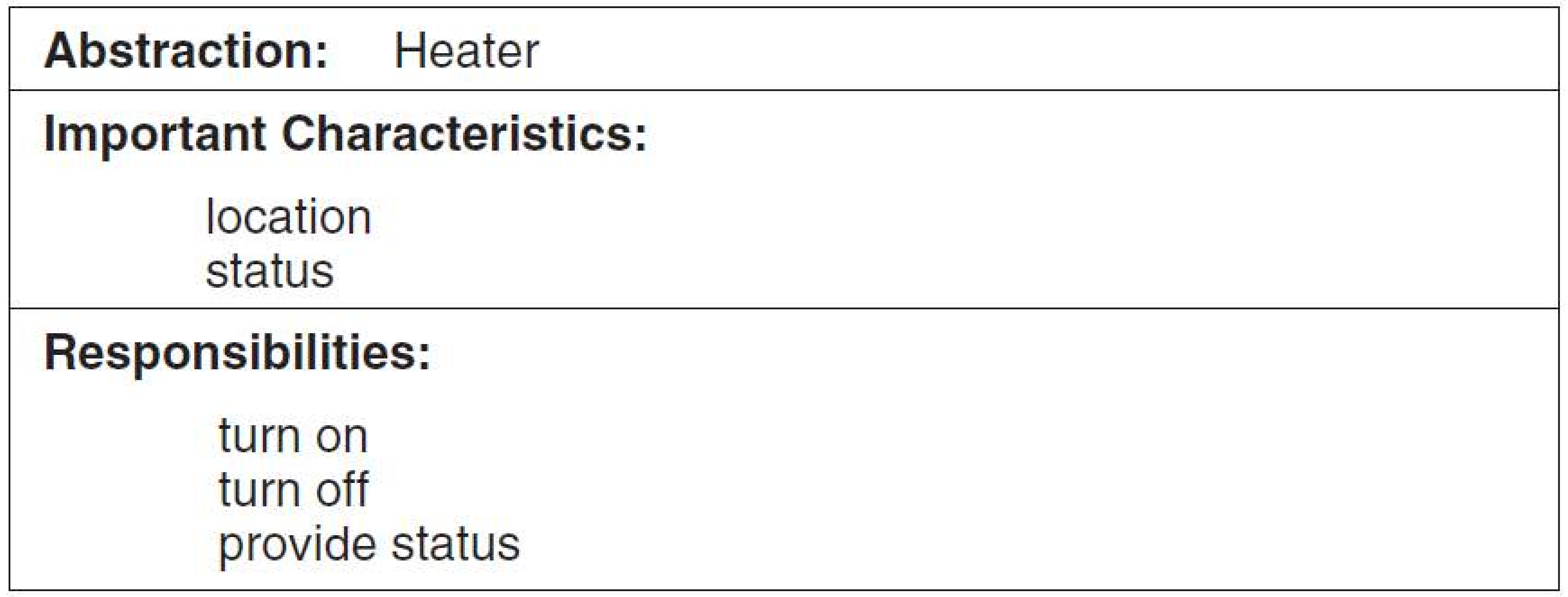
Encapsulation

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.

Examples of Encapsulation

- **Hydroponics Gardening System**
- Another key abstraction in this problem domain is that of a heater.
- A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform on this object: turn it on, turn it off, and find out if it is running.
- All a client needs to know about the class Heater is its available interface.

Examples of Encapsulation



Abstraction:	Heater
Important Characteristics:	location status
Responsibilities:	turn on turn off provide status

Related Candidate Abstractions: Heater Controller, Temperature Sensor

Examples of Encapsulation

- **Hydroponics Gardening System**
- Another key abstraction in this problem domain is that of a heater.
- A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform on this object: turn it on, turn it off, and find out if it is running.
- All a client needs to know about the class Heater is its available interface.

Examples of Encapsulation

- **Hydroponics Gardening System**
- We would not need to change the interface of the Heater, yet the implementation would be very different.
- The client would not see any change at all as the client sees only the Heater interface.
- This is the key point of encapsulation.
- The client should not care what the implementation is, as long as it receives the service it needs from the Heater.

Examples of Encapsulation

- Intelligent encapsulation localizes design decisions that are likely to change.
- As a system evolves, its developers might discover that, in actual use, certain operations take longer than is acceptable or that some objects consume more space than is available.
- In such situations, the representation of an object is often changed so that more efficient algorithms can be applied.
- This ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Modularity

- The act of partitioning a program into individual components can reduce its complexity to some degree.
- Partitioning a program is to create a number of well-defined, documented boundaries within the program.
- These boundaries, or interfaces, are invaluable in the comprehension of the program.
- Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.

Modularity

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Modularity

- 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.
- For tiny problems, the developer might decide to declare every class and object in the same package.
- For anything but the most trivial software, a better solution is to group logically related classes and objects in the same module and to expose only those elements that other modules absolutely must see.

ASSEMBLY INSTRUCTIONS
TICKER
STICK-ON STRIPES
DO-IT-YOURSELF CAT KIT
LEG-REAR OFFSIDE
PURRER
MIAOWER

Modularity packages abstractions into discrete units.

Modularity

- The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently.
- Each module's structure should be simple enough that it can be understood fully.
- It should be possible to change the implementation of other modules without knowledge of the implementation of other modules and without affecting the behavior of other modules.
- The ease of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed.

Modularity

- 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.
- System details that are likely to change independently should be the secrets of separate modules.
- The only assumptions that should appear between modules are those that are considered unlikely to change.

Modularity

- Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module.
- Any other program that requires information stored in a module's data structures must obtain it by calling module programs.
- Thus, the principles of abstraction, encapsulation, and modularity are synergistic.
- An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.

Examples of Modularity - Hydroponics Gardening System

- Suppose we decide to use a commercially available workstation where the user can control the system's operation.
- At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones.
- An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.
- Since one of our key abstractions here is that of a growing plan, we might therefore create a module whose purpose is to collect all of the classes associated with individual growing plans.

Examples of Modularity - Hydroponics Gardening System

- The implementations of these GrowingPlan classes would appear in the implementation of this module.
- We might also define a module whose purpose is to collect all of the code associated with all user interface functions.
- Our design will probably include many other modules.
- Ultimately, we must define some main program from which we can invoke this application.
- In object oriented design, defining this main program is often the least important decision, whereas in traditional structured design, the main program serves as the root, the keystone that holds everything else together.

Hierarchy

- Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time.
- 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.
- A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem.

Hierarchy

- **Hierarchy is a ranking or ordering of abstractions.**

- The two most important hierarchies in a complex system are:
  1. **Class Structure (“is a” hierarchy)**
  2. **Object Structure (“part of” hierarchy)**

Examples of Hierarchy: Single Inheritance

- Inheritance is the most important “is a” hierarchy.
- It is an essential element of object-oriented systems.
- Inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes.
- Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more super-classes.
- Typically, a subclass augments or redefines the existing structure and behavior of its super-classes.

Examples of Hierarchy: Single Inheritance

- Semantically, inheritance denotes an “is a” relationship.
- For example, a bear “is a” kind of mammal, a house “is a” kind of tangible asset.
- Inheritance thus implies a generalization/specialization hierarchy.
- Wherein a subclass specializes the more general structure or behavior of its super-classes.
- Indeed, this is the litmus test for inheritance: If B is not a kind of A, then B should not inherit from A.

Examples of Hierarchy: Single Inheritance

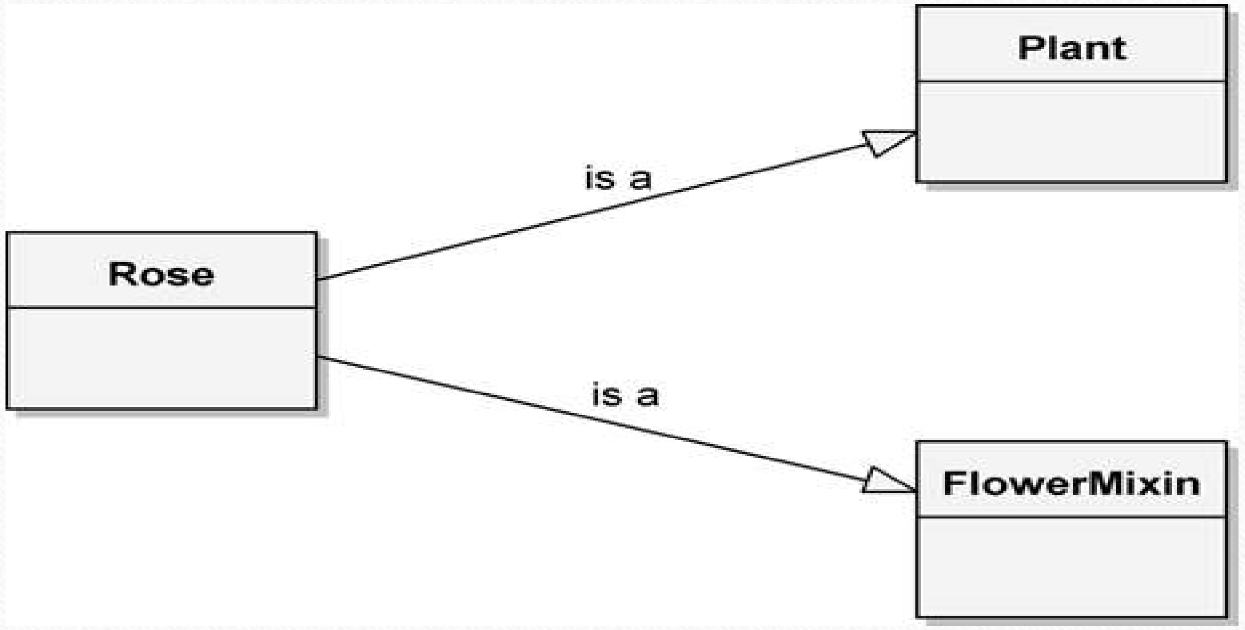
- **Hydroponics Gardening System**
- An earlier section described our abstraction of a very generalized growing plan.
- Different kinds of crops, however, demand specialized growing plans.
- For example, the growing plan for all fruits is generally the same but is quite different from the plan for all vegetables, or for all floral crops.
- Because of this clustering of abstractions, it is reasonable to define a standard fruit-growing plan that encapsulates the behavior common to all fruits, such as the knowledge of when to pollinate or when to harvest the fruit. We can assert that **FruitGrowingPlan** "is a" kind of **GrowingPlan**.

Examples of Hierarchy: Multiple Inheritance

- **Hydroponics Gardening System**
- The previous example illustrated the use of single inheritance: the subclass FruitGrowingPlan had exactly one superclass, the class GrowingPlan.
- For certain abstractions, it is useful to provide inheritance from multiple superclasses.

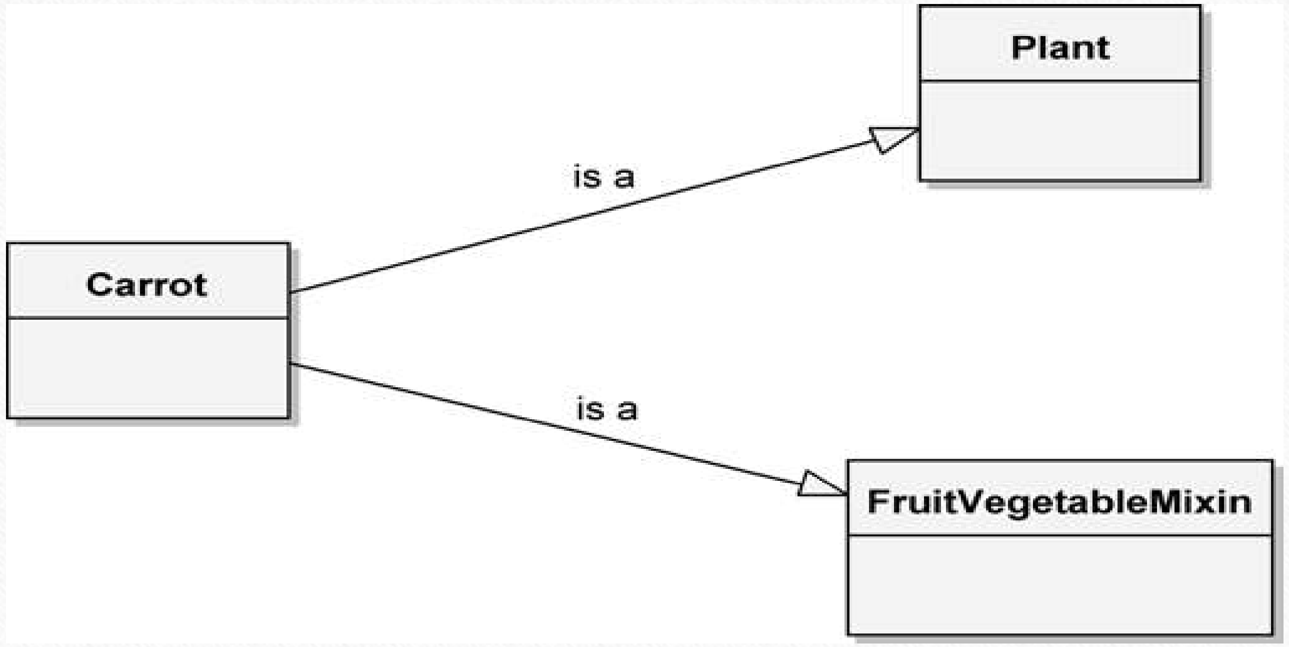
Examples of Hierarchy: Multiple Inheritance

- Hydroponics Gardening System



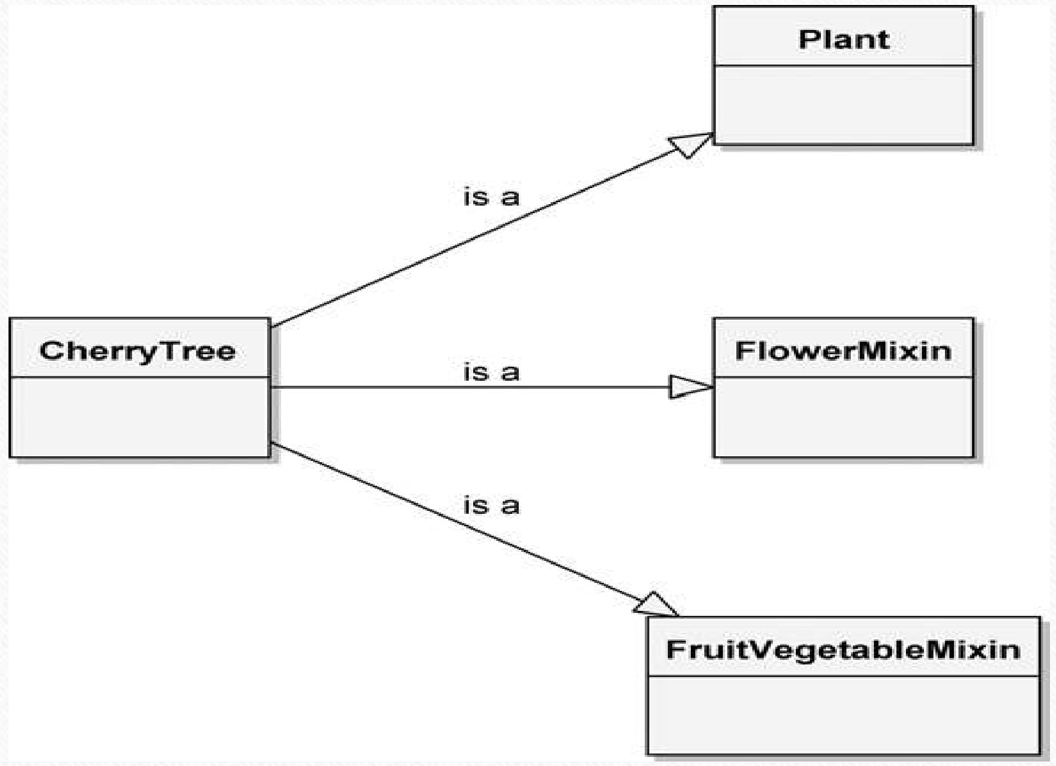
Examples of Hierarchy: Multiple Inheritance

- Hydroponics Gardening System



Examples of Hierarchy: Multiple Inheritance

- **Hydroponics Gardening System**


Plant
is a
CherryTree
is a
FlowerMixin
is a
FruitVegetableMixin