CS 332/780: Object-Oriented Databases Spring 2018
Course Notes #3
Keitaro Yukawa
Department of Computer Science
Queens College, CUNY

Purposes of Modeling

Modeling is an indispensable process for science and engineering projects that involve construction of complex systems. Database systems development is no exception. The model-driven database development has the following benefits.

- **Analysis and Synthesis**. Models facilitate analysis of the application domain by representing entities together with their structures and relationships at logical, conceptual levels. Models can be built in modular, evolutionary manner, and improved incrementally by continual evaluations.
- **Comprehension and Visualization**. Models capture logical and conceptual organizations of systems without low-level implementation details. Visualization provides fast, holistic, intuitive comprehension of systems.
- **Specification, Documentation, and Communication**. Models assist people that will implement, modify, maintain, and/or administer DB systems.
- **Model Compilation**. DB schemas can be automatically or semi-automatically generated from models by software tools.

Object-Oriented Database Modeling in UML

1 Introduction

Unified Modeling Language (UML) is one of the most widely used modeling and specification languages for software-intensive systems. (The official website is www.uml.org.) It is broad in scope and designed to model crucial aspects of software systems such as requirements, functional behaviors, and architectures. Although called a "language", it is actually a system of diagrams augmented with symbolic notations and annotations. In this and the next few installments of course notes, a relatively small sublanguage of UML suitable for object-oriented database modeling will be described: *class diagrams*, *object diagrams*, and *composite structure diagrams*. Class diagrams will be our main modeling tool. Although small, this sublanguage goes a long way for modeling essential object-oriented structures of databases at conceptual, logical levels. Another widely used database modeling method is *Entity-Relationship* diagrams described in many textbooks in database systems. Entity-Relationship diagrams have been used primarily for relational and object-relational databases. The sublanguage of UML we will study includes all the features of ER diagrams, casting them in object-oriented idioms, and also supports additional features such as multiple inheritance, aggregate/composite objects, and method functions. Today almost all database development projects are likely to be part of larger software development projects. The use of UML database models allows their seamless integration with the rest of software system models which themselves are constructed in UML.

In object-oriented databases (OODBs), data is stored in the form of *class*. A class definition serves as a data schema and its objects will constitute data items stored in the database. Classes in conventional object-oriented languages like C++ and Java are called *transient classes* because the objects created during a program execution cease to exist at termination of the execution. In contrast, classes used in OODBs are called *persistent classes* because the created objects will be automatically stored and remain in the database until they are explicitly deleted. Persistent classes may include method functions in the same manner that transient classes do. Functions may be used to retrieve data from persistent objects without changing database states in any way; such functions range from simple extraction of object attribute values to complex queries to lengthy algorithmic computations based on retrieved data. Other kinds of functions modify database states in some way, for example creation, deletion, or update of objects. In the following, "class" will be used in the sense of "persistent class".

The *extent* of a class is defined as the set of all objects currently existing in the class. As new objects are created and existing ones deleted, the extent of a class changes over time. Various relationships among class objects will be modeled as relations among the extents of classes. Class extent is distinct from class schema, which defines the format and type of a class by a list of attributes and functions.

It should be remarked that utility of object-oriented database modeling, in UML or other modeling languages, is not limited to OODBs. Object-oriented models can be implemented in object-relational databases by means of mapping from OO models to object-relational schemas. OO models can also be mapped to table schemas in pure relational databases, but the mapping is more complicated, especially the mapping of complex aggregate/composite objects, inheritance, and method functions.

2 Outline

In object-oriented database modeling in UML, the application domain is conceived of as being populated by various objects (entities). The objects are grouped into classes according to attributes and behavioral properties that characterize them. Classes may be organized by two kinds of hierarchies: *specialization* and *whole-part* hierarchies. Specialization captures categorization of a class into more specific subclasses that have additional, distinguishing attributes and behavioral properties. The extent of each subclass is a subset of the extent of the superclass. For example, a class of PCs may be categorized into subclasses of desktops, laptops, tablets, and smartphones. Each object of any subclass *is* an object of the superclass. Each desktop, laptop, tablet, or smartphone *is* a PC. For this reason, specialization is sometimes called the "is-a" relation between subclasses and their superclasses. When specialization is viewed in the reverse direction from subclasses to superclasses, it becomes *generalization*. The PC class is a generalization of the subclasses of desktops, laptops, tablets, and smartphones. Every subclass inherits the attributes and behavioral properties of its superclass(es) so specialization is often called *inheritance*. The subclass-superclass relation propagates transitively in both directions and defines the descendant and ancestor classes. Behavioral properties are often represented by method functions.

Whole-part relations capture aggregate or composite objects with component parts. For example, a PC, regarded as a whole, consists of many components like a processor, main memory, nonvolatile storage device, etc. This can be modeled by creating classes of processors, main memories, and nonvolatile storage devices, etc., and connecting them to the PC class by means of suitable whole-part relations. Note that each processor, main memory, or nonvolatile storage device *is a part of* a PC, but *is not* a PC. Hence the whole-part relation is sometimes called the "is-a-part-of" relation between component classes and their whole classes. One of the strengths of OO modeling and OO databases is the capability to model and implement specialization and whole-part relations in a direct, transparent manner.

In addition to class objects, various association relationships among them also need to be identified and established. They are modeled by n-ary relations, $n \geq 2$, among class objects. For example, a binary relation *manufactures* from the company class to the PC class can be established to indicate which company manufactures which PC.

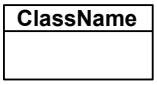
To summarize: Class objects in object-oriented database models are characterized by the following four aspects:

- attributes
- component parts
- relations
- functions

In the following, the main elements of our UML sublanguage – class diagrams – will be explained in terms of their graphical notation, semantics, and usage.

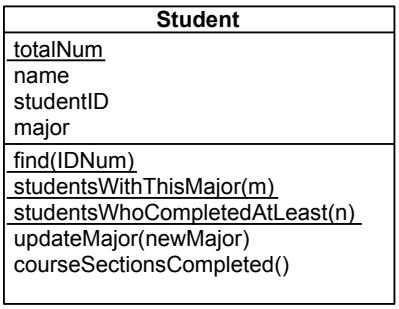
3 Classes

A class is represented by a rectangle labeled with the class name.



A class's attributes and functions may be listed inside its rectangle, but doing so tends to clutter a diagram with a large number of classes. UML tools usually provide options to display or hide attributes and functions in diagrams. Attributes and functions of a class will be collectively called *features* of the class. Class features divide into the following two kinds.

- **Instance Features**. Instance attributes are attributes of individual class objects. Each object has its own values of the instance attributes. Instance functions are applied to individual objects as their targets; the target object serves as a special parameter for the function.
- **Static Features**. Static attributes are attributes of classes as wholes. For each static attribute *a* of a class C, there will be just one value of *a* shared by all objects of C (the value of *a*, however, can be changed). Static functions are applied to class names as their targets. For example, the class of students in a college may have a static attribute totalNum to hold the total number of student objects currently existing in the class, i.e., the size of its extent. Alternatively, it may have a static function totalNum() that will dynamically count the total number of students. It may also have a static function studentsWithThisMajor(m) to return the collection of students that have majored or are majoring in the given parameter major m. Static features are underlined inside class rectangles.

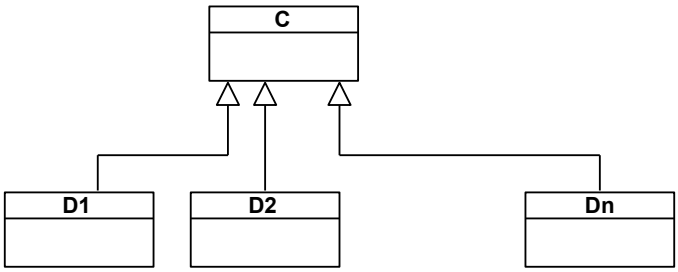


(Note: In UML versions 1.x, static features are called classifier-level features.)

4 Inheritance (Specialization)

Normally, a specialized subclass has additional features not present in the superclass, constraining its objects and reducing the extent of the superclass. Whenever D is a subclass of C, each D-object *is* a C-object, hence the extent of D is always a subset of the extent of C. This relation generalizes by transitivity: Whenever E is a descendant class of C in an inheritance hierarchy, each E-object *is* a C-object, hence the extent of E is a subset of the extent of C at all times. A descendant (specialized) class has more features, but fewer objects, than an ancestor (general) class.

4.1 Single Inheritance When a subclass is created from a single superclass, it is an instance of single inheritance.



The subclasses D_i inherit the superclass C's instance features. The static features *f* of C, if any, are features of class C as a whole and do not in general make semantic sense for D_i and hence are not inherited – they cannot be applied to D_i as in $D_i.f$. However, they are accessible in the context of the D_i by C.*f* (unless their scope is explicitly limited by visibility indicators like *private* described below). Inheritance polymorphism of functions is permitted; the code of any inherited function may be redefined in the subclasses. An inheritance hierarchy consisting only of single inheritance forms a rooted directed tree.

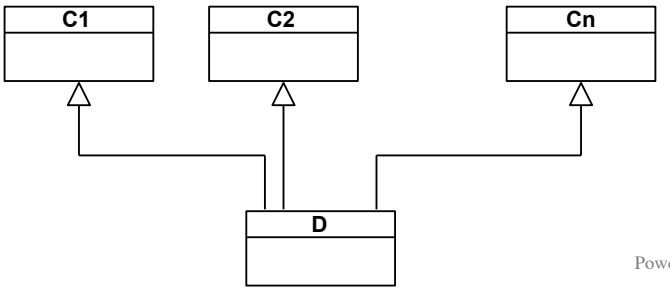
UML supports four visibility indicators (scope indicators) for class features:

- *private*: visible only in the defining class
- *protected*: visible within any descendant classes
- *package*: visible within the package
- *public*: visible anywhere

The private instance features of the superclass are not visible in the descendant classes. Yet they are inherited to the descendant objects and can be accessed within the superclass's context. In the above generic diagram, for example, any private attribute or function *r* of the superclass C is inherited to each subclass D_i but is not accessible within the context of D_i. However, C may have a function that takes a C-object as parameter, and this function can receive any D_i-object as parameter value and access its *r* feature inherited from C. The same visibility rules also apply to multiple inheritance described below.

Although visibility is an important issue in information hiding, it is best determined and specified at later stages of database modeling. It is not a pertinent issue at early to middle stages of modeling where structures and relations are identified and modeled. For this reason we will set aside the visibility issue in this course.

4.2 Multiple Inheritance When a subclass is created from two or more superclasses, it is an instance of multiple inheritance.

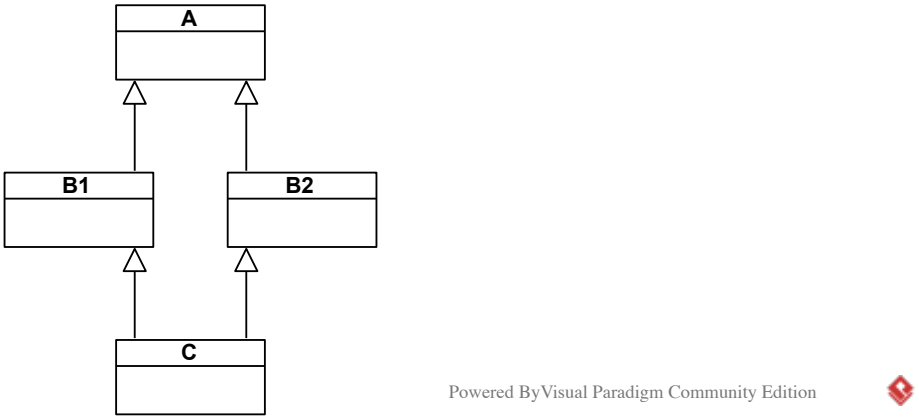


The subclass D inherits the instance features of all the superclasses C_i. The static features of C_i, if any, are not inherited but are accessible in the context of D (unless their scope is explicitly limited by visibility indicators like *private*). The code of any inherited function may be redefined in D. An inheritance hierarchy with multiple inheritance is no longer a tree but takes the more general form of a directed acyclic graph.

Multiple inheritance may introduce name-clash problems.

- Two or more C_i could inherit features of a common ancestor class higher in the inheritance hierarchy. How are these inherited in D?
- Two or more C_i could inherit a function of a common ancestor class, and some of them could redefine its code. How is this function name, now bound to different codes, inherited in D?
- Two or more C_i could define features with the same name (which are not inherited from a higher class). How are these inherited in D?

UML itself does not mandate any particular name-clash resolution rules, although it does provide some renaming methods so that users can implement the rules they adopt. Developers should adopt suitable name-clash resolution rules in consideration of the rules used in the implementation database system. For example, one rule might stipulate that all clashing names must be renamed to different names in the inheriting subclass. A more sweeping rule might enforce that all clashing features must be redefined to a unique, single feature in the inheriting subclass. In this course we shall adhere to one particular resolution rule for the first type of problem only. Suppose that two or more C_i inherit a feature of a common ancestor class higher in the hierarchy and they do not redefine it. Then we assume that, by default, only one copy of this feature is inherited in D. Consider for example this hierarchy:



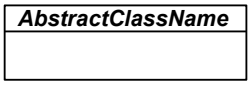
All the features of A are inherited by B_1 and B_2, which both use the same names as defined in A. If these features are not redefined, C inherits one copy of each of these features by default.

Later we will analyze in more detail the power, utilization, and complexity problems of multiple inheritance.

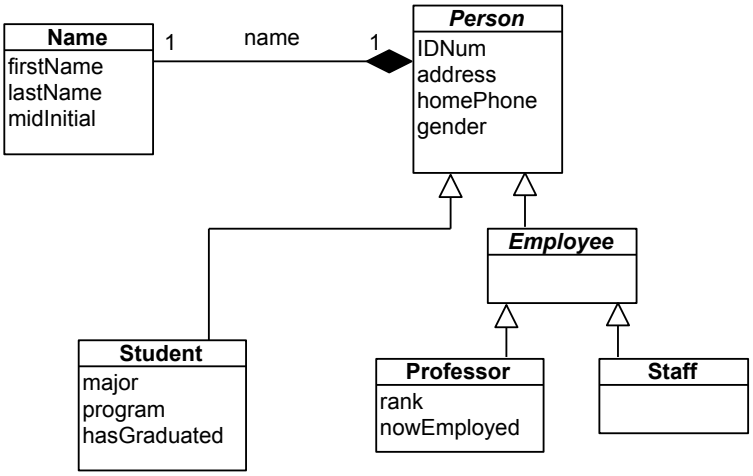
5 Abstract Classes

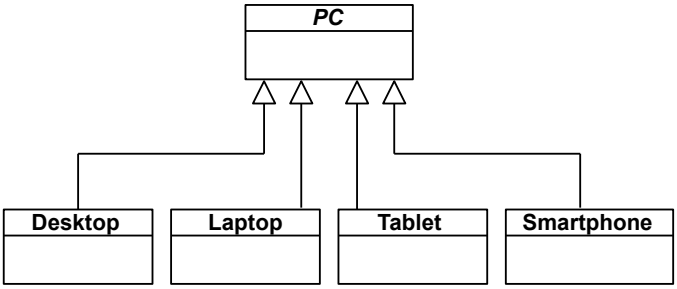
An object is called a *direct object* of a class C if it is an object of C but not of any of the descendants of C. An object is an *indirect object* of a class C if it is an object of C but is not a direct object of it. An *abstract class* is one that cannot have any direct objects of its own; it can only have indirect objects created in descendant classes that are not abstract. A *concrete class* is one that is not abstract and can have direct objects of its own. (This does not mean that a concrete class must always have direct objects - it may not have any.) Recall that the *extent* of a class C is the set of all the objects currently existing in C, including the objects in the descendants of C. We define the *proper extent* of C to be the set of all

direct objects currently existing in C, excluding the objects in the descendants of C. An abstract class is then characterized as one whose proper extent is required to be empty. An abstract class is represented by a rectangle with a class name in ***italics***.

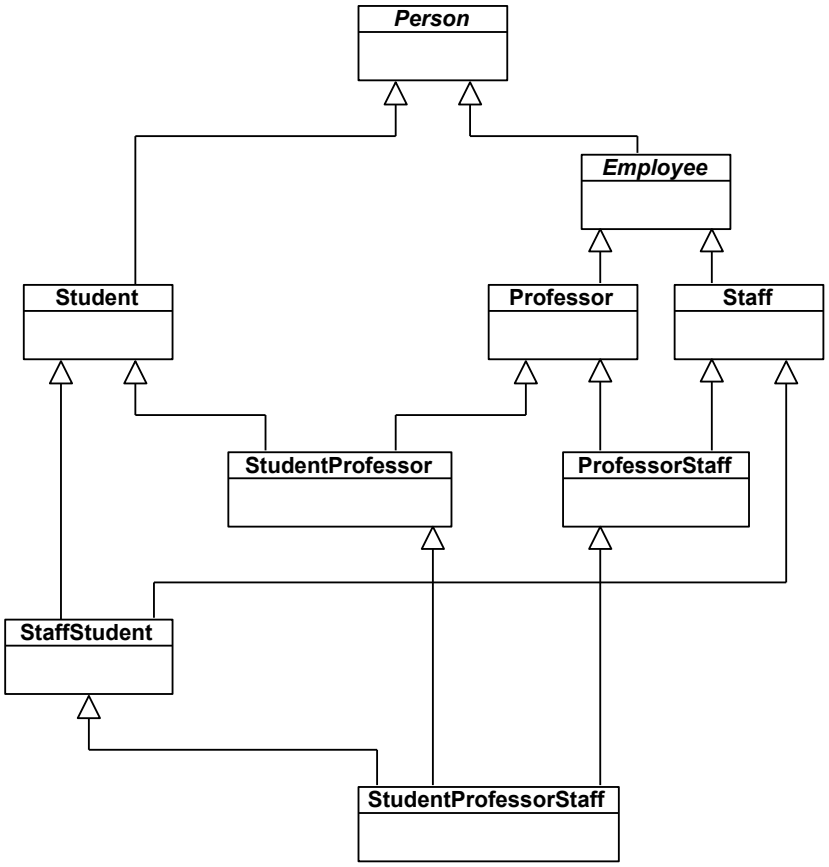


Abstract classes may, and usually do, have attributes, functions, and/or relations to other classes. Abstract classes are useful for modeling "generic" classes that capture common features of specialized descendant classes when the generic classes themselves need not have any direct objects. In many single inheritance hierarchies, only leaf classes (i.e. classes having no subclasses) need to be concrete. In the following diagrams, the classes *Person*, *Employee*, *PC*, and *Gadget* are modeled as abstract classes.

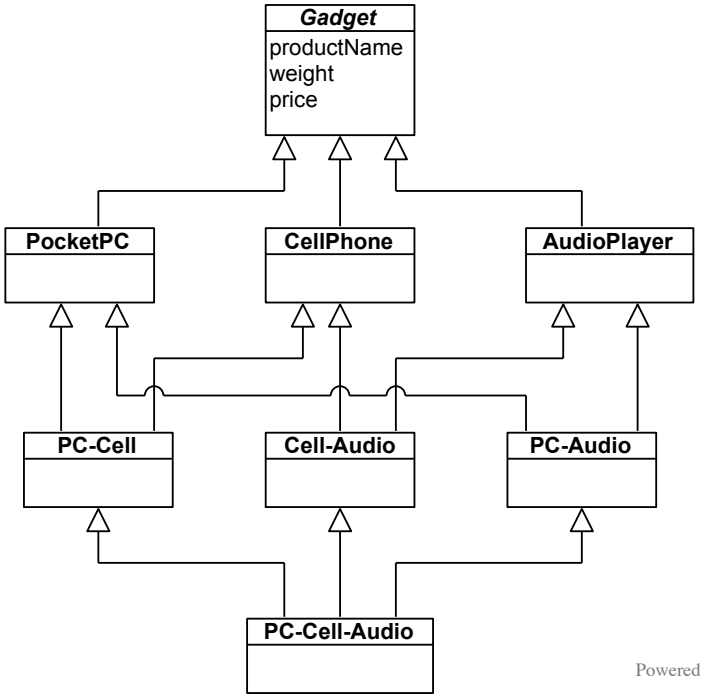




multiple inheritance hierarchy of Person in college



multiple inheritance hierarchy of Gadget



6 Criteria for Creating Subclasses

Generally speaking, creation of subclasses D_1, ..., D_n from the parent class C may be useful if the following criterion is met:

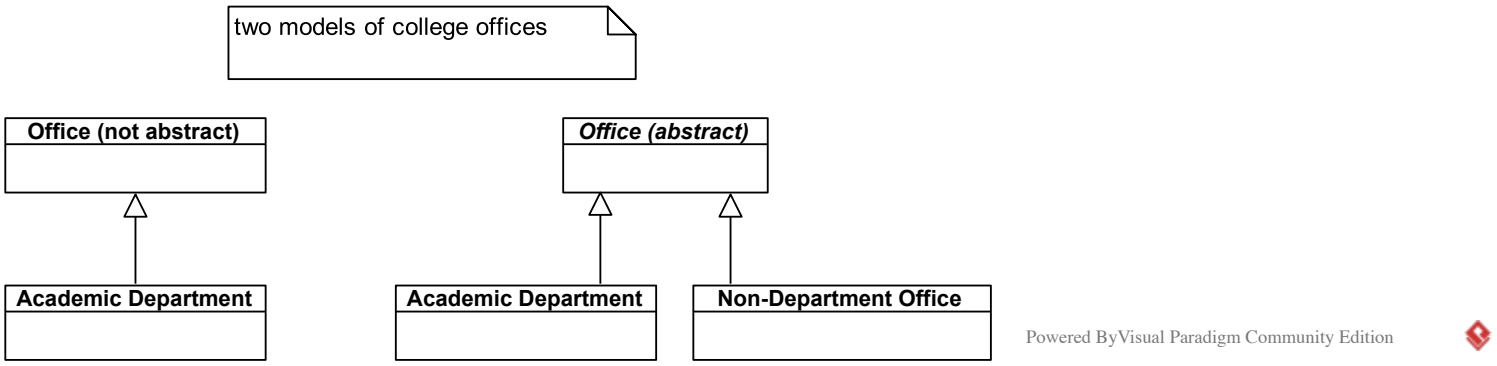
- Each D_i has at least one attribute, function, or relation which the other D_j do not have.

This condition is more of a necessary condition rather than a sufficient condition for creating subclasses. That is, creation of subclasses is generally justified only if this condition is met, but it may or may not be useful to create subclasses even if this condition is met. If fact, creating a subclass indiscriminately for every distinguishing attribute, function, or relation may well result in an unnecessary large number of subclasses. So a rule of thumb is: Create subclasses D_i when each of them has a significant number of attributes, functions, and/or relations which clearly distinguish its category from all other subclasses.

Consider the class of students in a college database maintaining the history of registered courses and transcripts. It would not be useful to create separate subclasses of male and female students since there are few, if any, gender-specific attributes/functions/relations that are relevant to such data; gender should simply be an attribute of the student class. Nor would it be useful to create subclasses of USA and international students since registered courses and transcripts, per se, are blind to the distinction. However, if the database also stores data like tuition fee and visa status along with functions to process them, it may well be useful to create the subclasses. Consider the class of patients in a hospital database. It would be useful to create subclasses of male and female patients since the patient database would need gender-specific attributes/functions/relations that are relevant to medical matters; for example, there are gender-specific body organs that require different medical examinations, procedures, and treatments. On the other hand, it would not be useful to create subclasses of male and female employees in a hospital payroll database.

It is clear from the above discussion that distinguishing features of subclasses *depend on application domains* and can only be identified by close analysis of the nature, relevance, and roles of class objects *in the context of application domains*.

There is an exception to the above criterion. Suppose we have created subclasses D_1, ..., D_n from the parent class C using the criterion. Sometimes we can identify objects of C that do not belong to any of the D_i. These objects may be put in C itself. An alternative is to create a subclass D_{n+1} to contain all such objects. The introduction of D_{n+1} may be justified even if it has no feature that distinguishes it from the other D_1, ..., D_n (in a sense its distinguishing feature is the absence of special features of the D_i). For example, consider the class of offices in a college. It is useful to create the subclass of academic departments because of its crucial distinguishing feature of offering courses. The other offices, like the registrar's office and the security office, may be all lumped together in the parent Office class itself, or may be put in the second subclass of non-academic departments.



7 Summary of Key Points

- Generalization-specialization class hierarchy by means of inheritance is a powerful tool in OO modeling and OODBs.
- Classes' *features* are attributes and functions. Classes may have instance features and static features.
- Multiple inheritance lets us create a subclass inheriting the features of more than one superclass and build inheritance lattices of generalization-specialization. But it may introduce complexity and potential name-clash problems.
- An abstract class cannot have direct objects – its proper extent is required to be empty. Abstract classes are used for "generic" classes in higher parts of inheritance hierarchies.
- In many single inheritance trees, only leaf classes need to be concrete.
- A rule of thumb: Create subclasses when each of them has a significant number of attributes, functions, and/or relations which clearly distinguish its properties from all other subclasses. Distinguishing features of class objects depend on the nature of application domains.

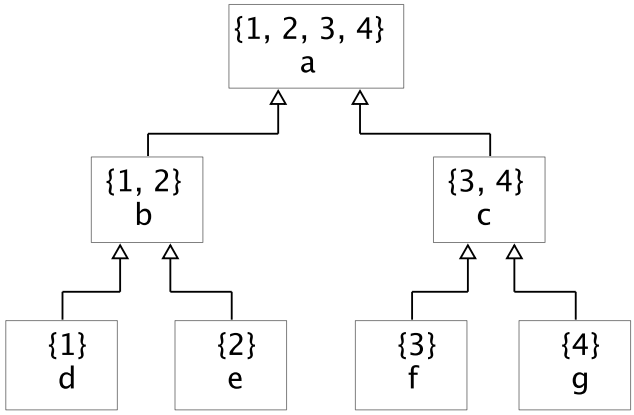
8 Algorithm to Build Inheritance Hierarchies

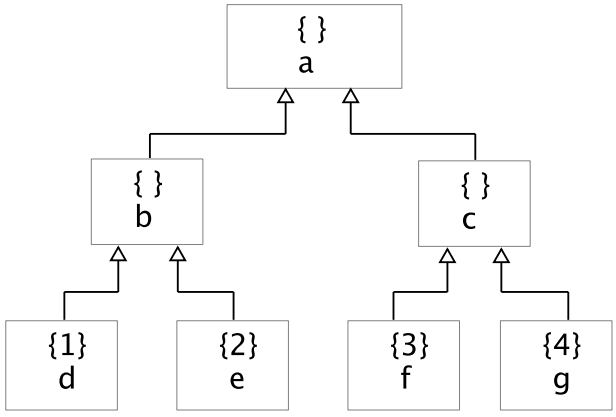
Input: Classes numbered 1, …, n and for each class i, $1 \le i \le n$, the set F_i of its features.

1. For each feature f in $F_1 \cup \cdots \cup F_n$, compute the set C_f of classes having feature f. It is possible that $C_{f_1} = C_{f_2}$ for distinct features f_1, f_2.
2. Create the hierarchy of the sets C_f's computed in step 1 with respect to the set containment relation $\subset$. The node for C_f introduces the feature f in this hierarchy. Each node now contains its class number and/or the class numbers of descendant classes.
3. From the set in each node, delete the class numbers contained in its descendant nodes. The nodes with empty sets are superclasses to be created and are candidates for abstract classes.
4. If any nodes have more than one class number, create a further lower hierarchy with respect to $\subset$ so that each node has at most one class number by repeating the process of deleting class numbers contained in descendant classes. This case occurs if an intended subclass does not introduce any new features. There may be multiple possible lower hierarchies, each of which is valid insofar as the hierarchical structure is concerned; however, some of them may be preferable to the others for semantic/pragmatic reasons.

Example 1 Input: Classes 1, …, 4 and the following F_i:

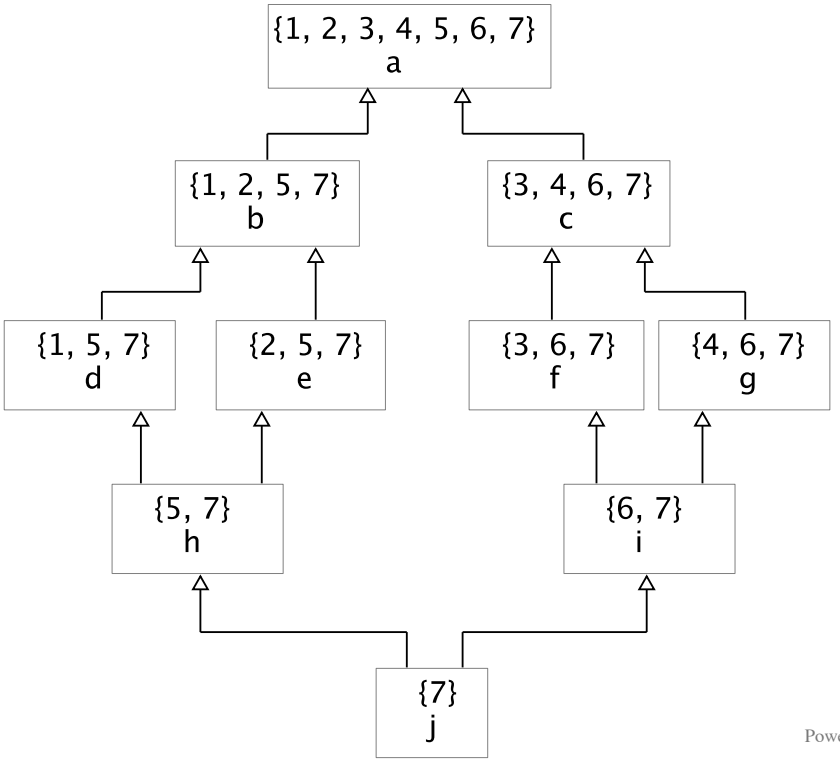
$F_1 = \{a, b, d\}$
$F_2 = \{a, b, e\}$
$F_3 = \{a, c, f\}$
$F_4 = \{a, c, g\}$

1. $C_a = \{1, 2, 3, 4\}$
   $C_b = \{1, 2\}$
   $C_c = \{3, 4\}$
   $C_d = \{1\}$
   $C_e = \{2\}$
   $C_f = \{3\}$
   $C_g = \{4\}$
2. 

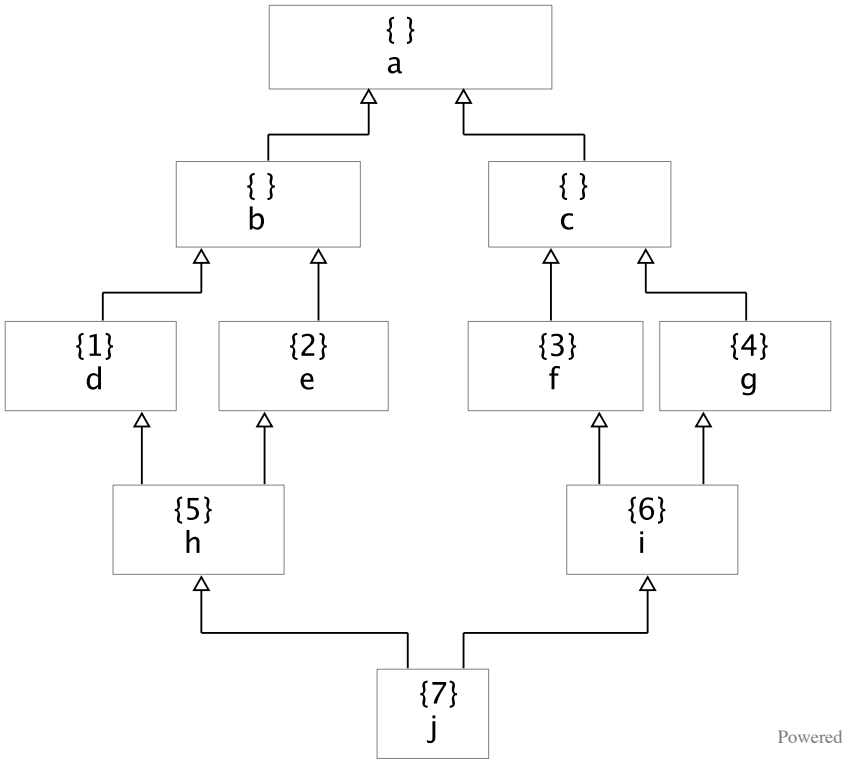

3. 

Example 2 Input: Classes 1, …, 7 and the following F_i:

F_1 = {a, b, d}
F_2 = {a, b, e}
F_3 = {a, c, f}
F_4 = {a, c, g}
F_5 = {a, b, d, e, h}
F_6 = {a, c, f, g, i}
F_7 = {a, b, c, d, e, f, g, h, i, j}

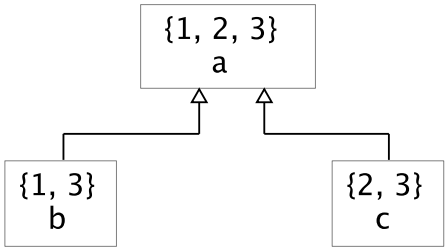
1. C_a = {1, 2, 3, 4, 5, 6, 7}
   C_b = {1, 2, 5, 7}
   C_c = {3, 4, 6, 7}
   C_d = {1, 5, 7}
   C_e = {2, 5, 7}
   C_f = {3, 6, 7}
   C_g = {4, 6, 7}
   C_h = {5, 7}
   C_i = {6, 7}
   C_j = {7}
2. 

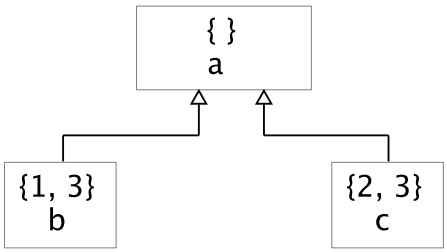

3.



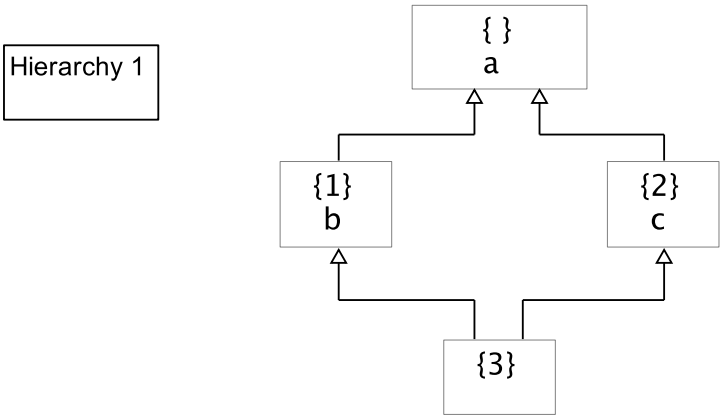
Example 3 Input: Classes 1, …, 3 and the following F_i:

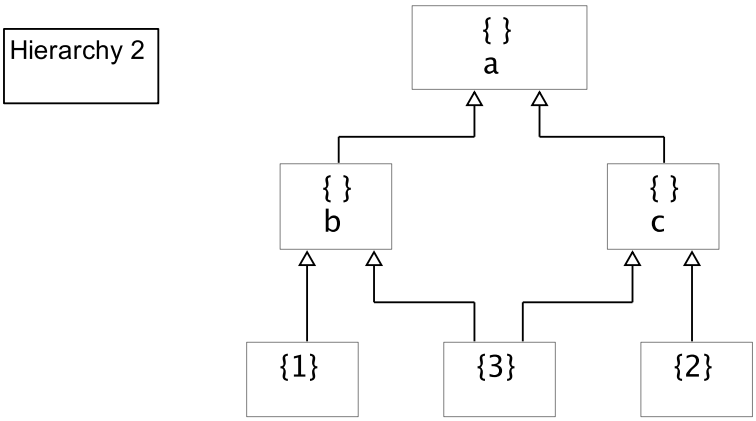
F_1 = {a, b}
F_2 = {a, c}
F_3 = {a, b, c}

1. C_a = {1, 2, 3}
   C_b = {1, 3}
   C_c = {2, 3}
2. 

3. 

4.





Example 4 Input: Classes 1 = Student, 2 = Employee, 3 = Professor, 4 = Staff (non-instructional), and the following F_i:

F_1 = {a, b, c, d, e, f, g, h}
F_2 = {a, b, c, d, e, i}
F_3 = {a, b, c, d, e, i, j, k}
F_4 = {a, b, c, d, e, i}

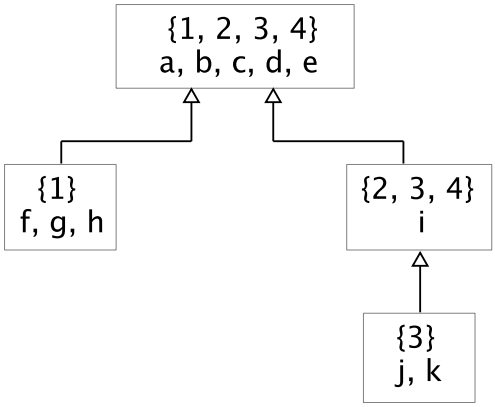
where

a = name
b = IDNum
c = address
d = homePhone
e = gender
f = major
g = program
h = hasGraduated
i = employment
j = rank
k = nowEmployed

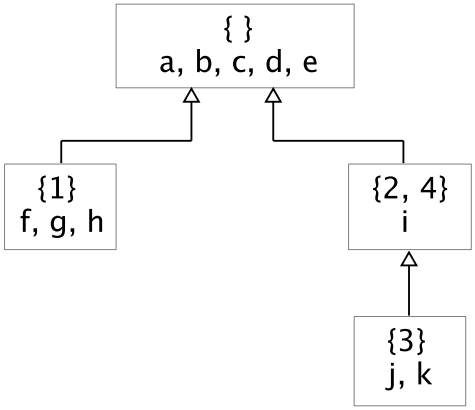
1. C_a = {1, 2, 3, 4}
   C_b = {1, 2, 3, 4}
   C_c = {1, 2, 3, 4}
   C_d = {1, 2, 3, 4}
   C_e = {1, 2, 3, 4}
   C_f = {1}
   C_g = {1}
   C_h = {1}
   C_i = {2, 3, 4}
   C_j = {3}

C_k = {3}

2.



3.



4.