More on OOP

by David G. Messerschmitt

Supplementary section for Understanding Networked Applications: A First Course, Morgan Kaufmann, 1999.

Object-oriented programming methodology is considerably more than just an object decomposition as described in Chapter 10. This section describes some more sophisticated and useful concepts in OOP, hoping to convey the richness and subtitles of this programming style.

Locating Objects

In order to interact with a server, a client object must be able to locate it through a name, address, or reference. In OOP, objects are normally located by reference. Any client must possess a reference to the server in order to invoke one of its methods, and must supply that reference as part of the invocation. Supplying the parameters and action to the object reference and returning the actions are roles for the infrastructure.

Complex Protocols in OOP

The core interaction among objects in OOP is the method invocation, which is a particular form of a request-response protocol. This does not rule out more sophisticated protocols, because more complex or simpler protocols can be constructed using the method invocation as a foundation. The send-receive protocol is the simplest, because it is a special case of the request-response protocol with no response. Thus, it can be implemented by simply specifying no returns for the method invocation.

A more complicated case can be illustrated by the publish-subscribe protocol. A server object that is willing to serve as a publisher can simply provide a `subscribe` method, as in:

```
subscribe: description, subscriber, subscription_ID →;
```

where `description` conveys the details of responses desired, `subscriber` is a reference to the subscriber—without which the publisher has no way to know where the subscription originated—and `subscribe_ID` is an identifier that is unique to `subscriber` for that specific subscription. It is the responsibility of the publisher to retain `subscribe_ID` and return it as a parameter of every response. The subscriber must also provide a method for the publisher to invoke whenever it has a response; for this purpose, it provides a `callback` method,

```
callback: subscribe_ID, information →;
```

The parameter `information` convey the desired information, and `subscribe_ID` allows the subscriber to uniquely identity the subscription associated with that information. This allows the subscriber to register subscriptions with multiple publishers, or multiple subscriptions with the same publisher, with all those subscriptions sharing the same `callback` method while keeping straight those responses.

Two Types of Relationships: "Is A" and "Part Of"

Object instances and classes illustrate that there are actually two complementary relationships among objects, called respectively the “part of” and “is a” relationship.

In Chapter 4, the architecture of a system was described in terms of its decomposition into modules. This decomposition is a “part of” relationship—each module is a “part of” the system. The primary purpose of the “part of” relationship, as described in Table 6.1 on page 135, is the separation of concerns. In OOP, an application is decomposed into objects (actually object instances).

The object class illustrates a complementary “is a” relationship among objects. The purpose of this “is a” relationship is an *economy of expression* [Boo94]. If a set of object instances behave identically given identical interactions, there is no sense implementing each one independently. A measure of software reuse can be obtained by implementing the class just once—or, in practical terms, this means specifying and documenting its interface and writing the programs describing it just once.

Example: Cosmology illustrates well the role of “part of” and “is a” relationships. Matter in the universe is decomposed into an unimaginatively large number of particles, most matter is composed of just four types of elementary particles: protons, neutrons, electrons, and neutrinos. There are only four distinct forces that bind or repel them: gravity, electromagnetism, and the strong and weak nuclear forces. Each proton, for example, is thought to have identical properties to all other protons—an example of economy of expression.

The cosmos also illustrates a hierarchical “part of” relation. For reasons not entirely understood, the matter in the universe is decomposed into galaxies, which are decomposed into stars, which are decomposed into molecules, which are decomposed into atoms, which are decomposed into “elementary” particles (which have no further decomposition) [Smo97].

In summary, the goal of architecture design can be stated more completely than in Chapter 6: The decomposition should be chosen to either separate concerns (make implementation independent) or to make concerns identical (only one implementation required).

Inheritance

The “part of” relationship is often hierarchical—modules decomposed into modules. Similarly, “is a” relationships can also be hierarchical.

Example: In the physical world, an atom shares properties with all other atoms—a decomposition into electrons, protons, neutrons, and electrons, the ability to participate in chemical reactions, etc. However, to say an entity “is an” atom does not specify *all* relevant properties, like its number of protons, neutrons, and electrons, or exactly how it will behave in a chemical reaction. On the other hand, to say an entity is a gold atom or a lead atom specializes it in a way that does specify completely its properties. A gold atom “is an” atom illustrates a hierarchical “is a” relationship. This relationship is one of elaboration and specialization.

Observe in this example that there are actually two distinct “is a” relationships. One is specialization, and the other is instantiation. Both of these relationships has economy of expression as its goal.

In OOP, the specialization form of “is a” relationship is called *class inheritance*. A more specialized *subclass* is said to inherit a *superclass* (and the subclass is said to be *derived* from the superclass). The terminology follows the observation that the subclass inherits the properties of the more general superclass.

Analogy: The health professions have an “is a” hierarchy as illustrated in Figure 1.. Each profession is a more specialized version of the one above it. The actual number of levels of hierarchy

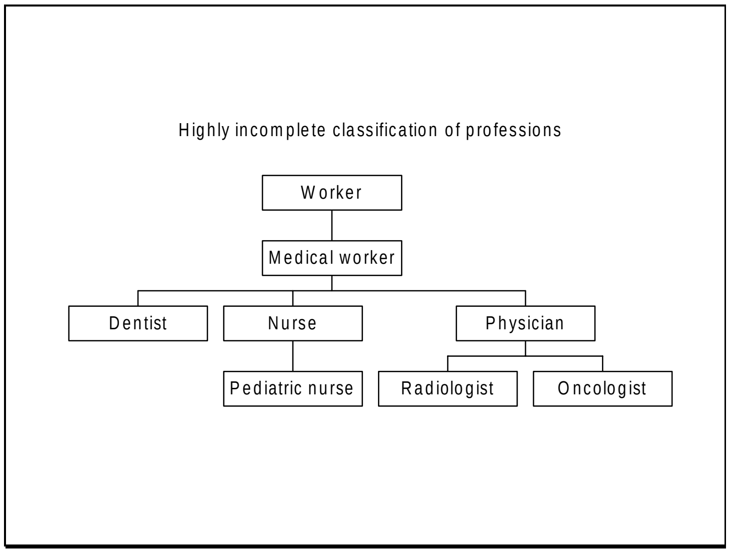

Figure 1. Hierarchy of "is a" relationships

is larger. For example, a pediatric oncologist "is a" pediatrician who specializes in treating tumors in children, while the pediatrician is a physician who specializes in the diseases of children. Each specialization requires the *addition* of increasingly specialized knowledge. To become a pediatric oncologist, you first become a physician, then add specialized knowledge to become a pediatrician, and then add further knowledge to become a pediatric oncologist.

In practical terms, a subclass incorporates the methods of its superclass and adds additional methods. Thus, specialization means "addition" of new data and methods, as illustrated by the last analogy. A subclass may also *modify* methods it inherits from its superclass.

Example: A simple example of inheritance is shown in Figure 2.. An `Auto`, `Bicycle`, and `Boat` are all subclass of class `Vehicle`, with the common property of moving people or goods. A `Vehicle` models a vehicle behavior (see "Relationship of Software and Real-World Entities" (Section on page 264)), and has two representative attributes—location and velocity, which can be ascertained by the methods shown. Class `Vehicle` captures everything in common among all vehicles. Particular `Vehicle`'s also have specialized characteristics. For example, the `Boat` may need a `bail_water` method, which makes no sense in the context of the `Auto` or Bicycle.

The class `Boat` has three methods shown in Figure 2.: `whatis_location`, `whatis_velocity`, and `bail_water`. The first two are inherited from `Vehicle` and the third is added in the subclass. It would also be possible for class `Boat` to redefine one of the methods defined in its superclass `Vehicle`. For example, the `whatis_velocity` method for class `Boat` must take into account water currents—not an issue with land vehicles—and thus might have to model its velocity differently.

It is important to note that "is a" and "part of" relationships *coexist* in a system. Every module has both relationships to other modules. A large number of such relationships may coexist.

Example: Take the example of a hospital:

- St. Mary's "is a" hospital, which "is a" corporation and "is a part of" the health care system.
- George "is a" nurse (which "is a" health care worker) and "a part of" St. Mary's.

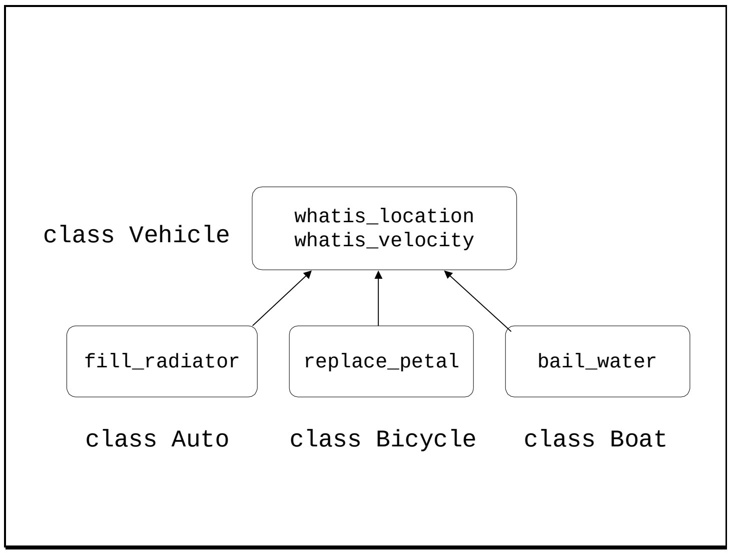

Figure 2. An example of inheritance in modeling vehicles. The arrows point from subclass to superclass, and each class is labeled with representative methods.

- Gwen Sickly "is a" person, who it happens "is a" patient and "is a part" of St. Mary's. Gwen is also "a part of" the Sickly family, which "is a" human family.

One goal of inheritance is economy of expression in software reuse (see "Software Reuse" (Section 10.1.2 on page 256)). All the effort applied to implementing a superclass need not be repeated for subclasses.

State

The term *state* is used in a number of contexts, including OOP. In pragmatic terms, the state of a module is the entirety of the data that it stores. Functionally, the state has an important role in influencing the behavior of a module, which is impacted not only by external events but also by state. Every object contains internal data, together with methods that change that data. Data visible at the interface are attributes, but there may also be encapsulated data that is not made visible. The sum total of the data in the object (visible and encapsulated) constitutes the *state*.

Analogy: The state of a battery-operated wallclock would include the displayed time (also an attribute) and the energy stored in its battery (which is encapsulated, but necessary to determine when the clock stops).

When a method is invoked on a server, the client state may be affected by the return values, and the server state may be directly affected. The method interaction typically affects the state of *both* objects. The state has several complementary interpretations:

- Two instances of the same class have the same interface and implementation. They differ only in their states, which may be different.
- The state reflects all the external interactions of the object since it was created. In fact, the state constitutes all the data the object keeps around as a record of its past interactions.
- The state affects an object's future behavior when it interacts with other objects.

Analogy: An example from the physical world would be a chess game. The current position of

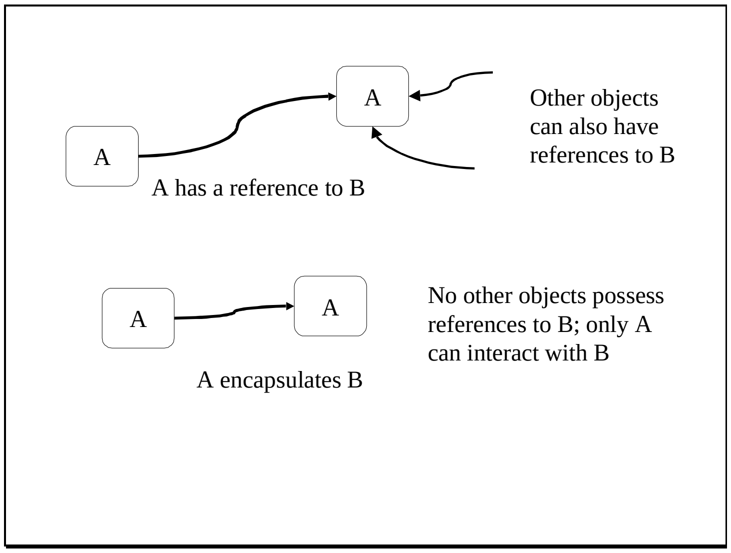
A
A
A has a reference to B
Other objects can also have references to B
A
A
A encapsulates B
No other objects possess references to B; only A can interact with B

Figure 3. Objects can interact with any other objects for which it has a reference. If only one reference exists, then that object is encapsulated by the object with that reference.

the pieces on the board summarizes the impact of all past moves, and constitutes the state of the game. Although the precise sequence of past moves cannot be inferred from that state, given it and a specified sequence of future moves the resultant state can be predicted.

The concept of state applies to circumstances other than objects as well. Whenever an entity keeps around data which may affect its future behavior, that is state.

Example: The Web server that keeps track of information on a particular user, such as how much money he has spent in the past, is maintaining state for that user. This terminology applies whether or not the Web server is an object.

Object Encapsulation

Decomposition of a system into subsystems is often hierarchical, so the system can be viewed at different granularities. In OOP, this hierarchy is achieved by allowing one object to encapsulate or “own” other objects, which are not visible or accessible from the outside.

As shown in Figure 3., encapsulation is determined by the number of existing references to an object. If exactly one reference exists, then whoever holds that reference and does not share it with anybody else encapsulates the referenced object—no other clients know about it in order to interact with it. On the other hand, if two (or more) clients possess a reference to a server, then they can each interact with it, and it is not encapsulated.

Example: An object modeling the department in an organization would probably have a list of employees in that department. Each employee’s information such as name, age, salary, etc. could be represented by an `Employee` object. If the `Department` object, and only the `Department` object, maintains references to each of those `Employee` objects, then they are encapsulated. To find out about a particular `Employee`, a client has to pose that question to the `Employee`’s `Department`, not directly to the `Employee`.

Polymorphism

It is common for objects with a hierarchical “is a” relationship to have common methods, but

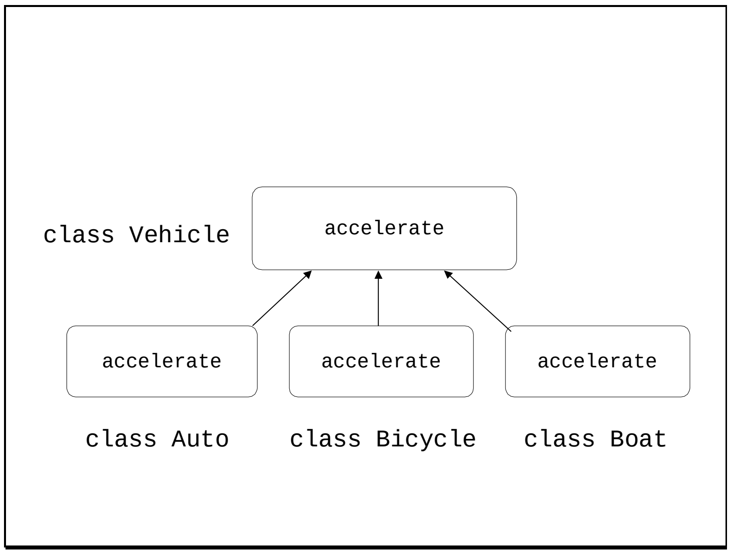

Figure 4. Polymorphism is illustrated by several types of vehicles. A vehicle could be modeled without knowing whether the vehicle was an auto, bicycle, or boat.

which behave differently. This is a rough definition of *polymorphism*. Polymorphism is one of the most subtle—but also most powerful—features of OOP.

Analogy: Automobile manufacturers distinguish their cars in various ways—including performance, features, esthetics, etc.—but they all have a very similar interface to the driver (steering wheel, brake and accelerator petals, etc). Any driver can immediately drive any auto. Yet, the different autos behave differently when that interface is exercised. For example, one may accelerate more quickly than another, even for the same depression of the accelerator petal.

Polymorphism is achieved in the context of inheritance. A superclass may have a method that is redefined (specialized) within different subclasses. Polymorphism means that we might "pretend" that an object is an instance of its superclass, but the actual behavior is determined by the subclass. Polymorphism is so useful because it allows programs to be constructed around abstract views of objects; later, when instances of objects drawn from subclasses are substituted, new behaviors result without any other program modification.

Example: A class `Vehicle` would probably include an `accelerate()` method that causes it to gain speed. As shown in Figure 4., subclasses `Auto`, `Bicycle`, and `Boat` would implement `accelerate()` differently. For example, the `Auto` and `Bicycle` might simply release the brake if the `Vehicle` is coasting downhill, whereas a `Boat` doesn't even have a brake. A object-oriented application might deal with `Vehicle`'s without even knowing about their subclasses. When a `Vehicle` is accelerated, the object instance will substitute its own specialized behavior depending on its subclass. New subclasses of `Vehicle`'s can be added without any other changes to the program.

Object Interfaces

The class of an object comprises both an interface and an implementation, and it is valuable to separate these two aspects. A client is concerned with a server's interface, but not its implementation. For this purpose, an *interface definition language (IDL)*—simpler than a system program-

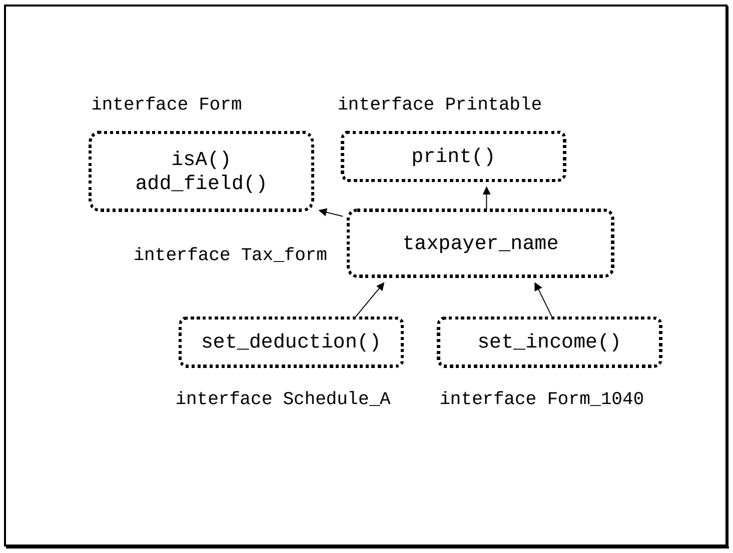

Figure 5. An illustrate of interfaces and multiple interface inheritance.

ming language—describes object interfaces but without introducing implementation issues. One advantage of an IDL is that it is simpler, because it deliberately ignores all implementation issues. (IDL's are also important in distributed object management, see Chapter 16.) One IDL is defined in the CORBA standard discussed in Chapter 16 (see the book homepage for a description of this IDL with examples).

Example: The interfaces of classes sharing "is a" relationships are illustrated in Figure 5.. Each interface is outlined with a dotted line (to distinguish it from a class) and labeled with some representative methods and attributes. Interface inheritance—denoted by arrows from a derived interface to its base interface—simply means that the derived interface has all the methods and attributes of the base, plus more methods and attributes that further specialize it. Also illustrated is *multiple* interface inheritance, since interface `Tax_form` inherits two base interfaces, `Form` and `Printable`, acquiring all the methods and attributes of both. `Schedule_A` and `Form_1040` are each more specialized `Tax_form`'s that inherit all its methods and attributes.

This example also illustrates polymorphism. Suppose there is a repository that stores `Form`'s and a print server that arranges to print all `Printable`'s. Then any `Tax_form`, including `Schedule_A` and `Form_1040` and any others defined in the future, can be stored or printed without modifying either the repository or print server.

Events

"Complex Protocols in OOP" (Section on page 1) illustrated how a more complicated publish-subscribe protocol can be implemented using request-response as a building block. Publish-subscribe protocols are so useful and widely used that component technology typically supports this in the infrastructure.

A component and its interface are illustrated in Figure 6.. At this level of detail (before enhancements like component metadata are considered) the component interface looks very similar to an object interface. The primary difference is that the component specifically publishes *events*. Events are used, albeit on an *ad hoc* basis, in object systems. An event is an action or occurrence that a component publishes for the benefit of other components. Other components, at their

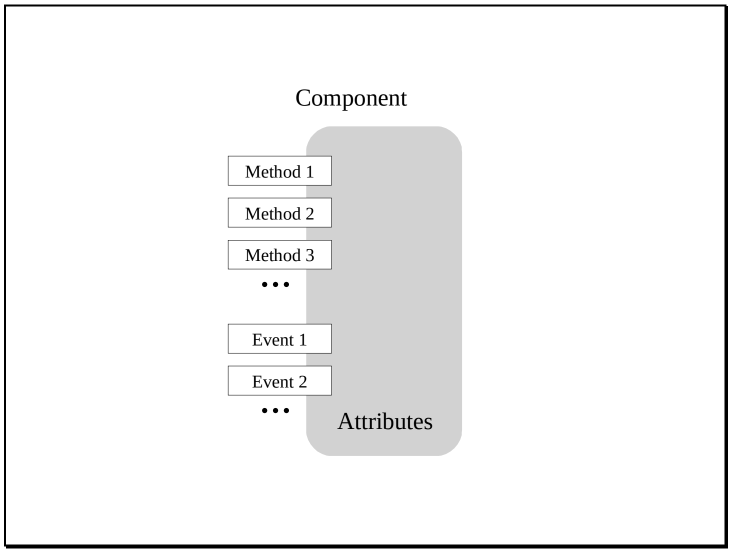

Figure 6. External view of a software component. There is no internal view.

option, subscribe to notifications of the event (similar to information management, see "User Awareness: Notifications" on page 41).

A typical form of specification would be an attribute crossing a threshold.

Example: Suppose a component models a bank account. Examples of event specifications might be “with the latest deposit, the balance in the account has just exceeded $1,000,000” or “with the latest withdrawal the balance would have been negative so it was rejected”.

When an event occurs, components that have subscribed are notified. This works as follows:

- The component metadata describes any available events, and the component interface provides a mechanism to subscribe to any event.
- A subscribing component provides a parameterization (the threshold on the account balance, for instance), but does not have to identify itself because the infrastructure takes care of that.
- During execution, a component notes all events for which there is one or more active subscription, and notifies all the subscribers whenever that event occurs.

With events, a client doesn’t have to repeatedly query a server, but need subscribe only once. Events are useful in many contexts, but particularly business applications as illustrated by the following example.

Example: A bank credit department may ask to be notified whenever a particular customer’s bank balance falls below zero, because that customer is deemed a collection problem. A marketing department may ask for notification whenever the weekly sales of a particular product falls below some threshold to trigger a reevaluation of marketing plans. A inventory department may desire notification is a supplier ‘s production is affected by some natural disaster, so that manufacturing forecasts can be adjusted. These and similar cases are easily handled by events, if the designers of relevant components have anticipated the need.

Discussion

D1 Discuss the use of “part of” and “is a” relationships in biology, the social sciences, the economy and commerce.

D2 Discuss the use of polymorphism in physical-world products and services. Is this a valuable way to think about the organization of products and services, or not?

Examples

The ides of inheritance and polymorphism can be illustrated by a couple examples.

Address Book

Suppose the goal is to implement an address book as an adjunct to an email application (see "Remote Conferencing with Shared Workspace" (Section 2.2.3 on page 23)). The address book manages a list of email recipients, keeping information like name, address, phone number, and a convenient nickname for each recipient. As shown in Figure 7., an object with interface `Entry` can be used to manage the information associated with each recipient. Since the name, address, and phone number are widely used in many applications, they are each represented by their own objects, whose interfaces are called `Name`, `Address`, and `Phone_number` respectively. Those three interfaces are inherited—with the nickname and email address added—to yield the `Entry` interface. This illustrates something that has not be discussed previously—*multiple inheritance*. `Entry` inherits three interfaces, which simply means that it gathers together the attributes and methods of all those interfaces, and then adds its own specialized attributes and methods.

The address book has to manage a *list* of entries—another generic function that can be shared by many applications. Thus, as shown in Figure 8., the interface `Address_book` can inherit a `List`

How OOP Contains Complexity

It is useful to review how OOP assists in containing complexity:

- *Modularity*. OOP encourages the decomposition of the application into modules, namely objects. Hierarchical decomposition is supported (albeit rather crudely) by encapsulation of other objects by holding exclusive references.
- *Interface*. Each object has an interface that allows other objects to interact with it in well-defined and well-documented ways.
- *Abstraction*. The programmer can carefully choose what is displayed at an object interface, and choose to hide internal details.
- *Encapsulation*. Nothing internal to an object is accessible from the outside except what is explicitly embodied in the methods at the interface. Implementation details can be changed without affecting interaction with other objects.
- *Class*. An economy of expression—which can reduce implementation effort—is achieved by implementing a class once and instantiating it many times. Inheritance enhances economy of expression by implementing once what is common among a set of related classes.
- *Polymorphism*. Subclasses can be hidden from a client object, which sees only the superclass interface, allowing different subclass behaviors to be automatically substituted.

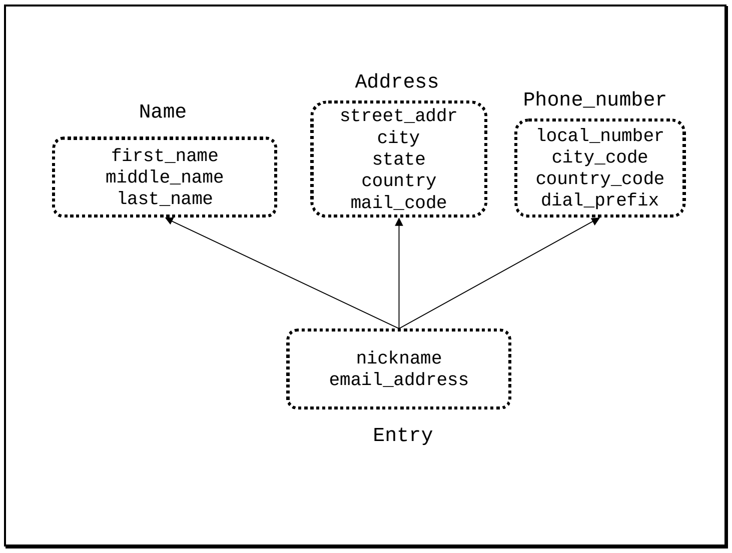

Figure 7. Interface inheritance for an address book entry object.

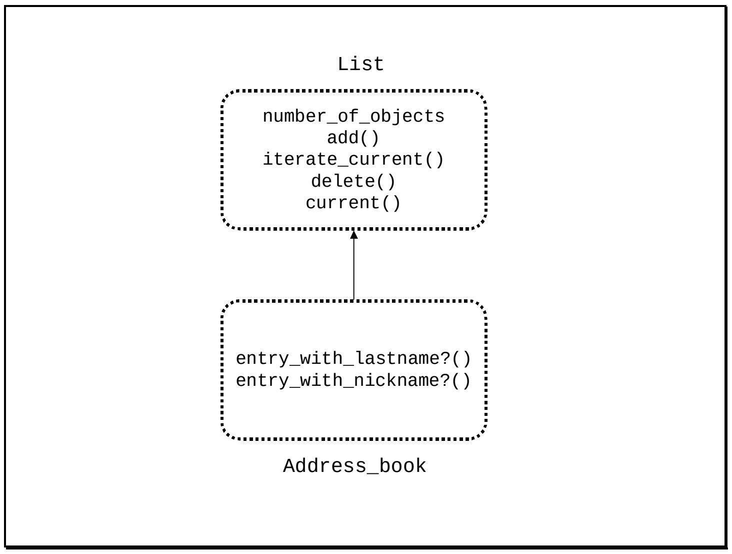

Figure 8. The address book object interface can be inherited from a generic list inter-

interface. A `List` exploits polymorphism to manage a list of many objects (with literally any interface). The `Address_book` adds methods specific to the email application, such as searches for specific last names and nicknames.

Shopping Cart

In the shopping cart object architecture, the `Customer_info` and `Entry_info` instances are encapsulated in `Entry_list`. The latter object maintains references to those objects which it does not share with others. All inquires from `Customer_interface` are handled by appropriate methods of `Entry_list`, which may in turn consult its encapsulated objects.

`Entry_list` inherits the same `List` class as the `Address_book`, adding methods specific to the shopping cart like searching on authors or book titles.

Review

Although the method invocation directly supports a request-response protocol, simpler or more complex protocols can be realized using the message. Common examples include the send-receive (message) and publish-subscribe (event notification).

The instances of a class are an example of an "is a" relationship, which can be hierarchical through inheritance. Decomposition leads to a "part of" relationship.

State is the collective data stored in an object, reflecting all the information it keeps about its past history. Knowledge of state is needed to predict future behavior.

Polymorphism allows the substitution of a superclass for a subclass, allowing subclasses to substitute distinct behaviors.

Concepts

Object-oriented programming:

- Object: attribute, method, interface, state, and instance
- Class: instance, inheritance, and polymorphism
- "is a" relationships: specialization and instantiation
- "part of" relationships: hierarchical decomposition
- Events

Exercises

E1. For each of the following ideas, give two examples of physical-world objects that illustrate the idea, and tell how they illustrate the idea.

a. State
b. The instantiation type of "is a" relation
c. The specialization type of "is a" relation
d. "part of" relation
e. Polymorphism

E2. Even though the objects in each collection below undoubtedly don't belong to the same class, describe briefly what the objects have in common that could be captured in a superclass.

a. Tax form, driver's license, house
b. Tennis racket, baseball bat, shovel
c. Box of chocolates, floppy disk, automobile trunk
d. Physician, nurse, patient

E3. Define an inheritance hierarchy for any one of the following. Try to define classes that would be reusable.

a. Vending machines
b. Textbooks in a campus bookstore
c. Classes in a university
d. Food items in a cafeteria

E4. Suppose that you were writing software using objects to model or represent the following real-world objects. What would constitute their state?

a. Television set

b. Automobile

c. Bank account

d. Flashlight

E5. Develop a hierarchy of "is a" relationships for transportation vehicles. Pay particular attention to the number of levels of hierarchy you believe is appropriate to maximize the economy of expression. Include at least 8-10 different classes of vehicles in your hierarchy. Briefly justify the features of your design.

E6. Create an "is a" hierarchy for each of the following. Your hierarchy doesn't need to be complete, but should have representative examples.

a. Ways for two people to communicate

b. Methods of payment in commercial transactions

E7. The point of this problem is that one object can participate simultaneously in more than one "is a" and "part of" hierarchies. Illustrate this, by creating some partial hierarchies including this object, for any two of the following:

a. A house in San Francisco

b. A rock in the Berkeley hills

c. A painting in The Louvre art museum in Paris

E8. Give three examples not mentioned in the chapter of polymorphism in real-world objects.

E9. Assume you define software classes designed to serve as proxies for electronic pianos, trumpets, and drums in a "digital band" application. Describe briefly how polymorphism would be valuable in defining these classes and their superclass "musical instruments".

E10. State how polymorphism could be used to advantage for objects modeling each of the following combinations of classes.

a. Radio, television, and book

b. Warehouse, bookstore, automobile dealership

c. Piano, trumpet, drum

E11. For any one of the following examples, design a set of classes and an interface specification for each class, taking advantage of inheritance and polymorphism. There is no need to design a detailed interface, but rather it suffices to illustrate how inheritance and polymorphism are helpful.

a. An inventory of books maintained by the on-line book merchant books4u.com described in "On-line Book Selling" (Section 3.1.2 on page 55)

b. Different forms of monetary value used in "On-line Stock Trading" (Section 3.1.3 on page 57) to represent money

c. Different types of flowers in the catalog of the "Floral Delivery Service" (Section 3.1.4 on page 58)