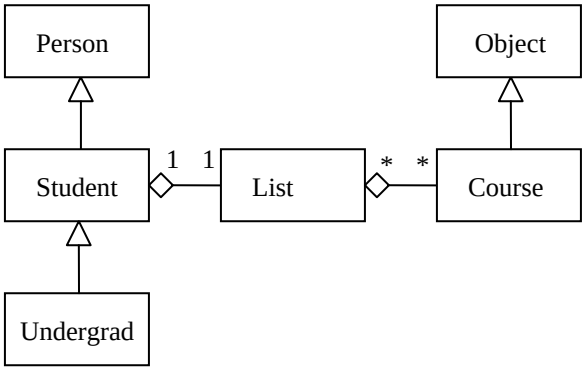
Object-oriented design

Professors Clark F. Olson and Carol Zander

There are some good articles at:
http://www.objectmentor.com/resources/publishedArticles.html (click on Topic: Object Oriented Design)
See, in particular, the articles on "The Open Closed Principle" and "The Liskov Substitution Principle." The "Principles and Patterns" article summarizes several of principles. For design strategies, see the Payroll Case Study, including "Analysis by Noun Lists and Use Cases" and "The Payroll Case Study: Finding the Underlying Abstractions."

UML notation

To represent object-oriented designs, we will use UML (Unified Modeling Language) notation to show class relationships. The relationships that we need to express are inheritance and composition. Inheritance is, of course, what we've just been studying. In UML, an inheritance relationship between two classes is denoted using an arrow from the derived class to the base class (typically, the head of the arrows is unshaded). The other relationship is composition. This is where one class has another class in the private data. Inheritance is an *is-a* relationship. Composition is a *has-a* relationship. The following diagram indicates that a Student is a Person and an Undergrad is a Student. In addition, each Student has a List (in their private data) representing the list of courses that the student has taken. The List is composed of Courses. Note that each Student has a single List, but a List can consist of many Courses. In fact, each Course object can be in many Lists (through pointers presumably). Composition relationships can be one-to-one, one-to-many, or many-to-many. Inheritance relationships are always one-to-one, so we don't need to specify the multiplicity on them. Note that Undergrads inherit the composition relationship with List, so that all Undergrads also have a List of Courses.



See all the List of Fruit examples for example implementations of this kind of relationship.

Object-oriented design principles

We've already seen some important object-oriented design principles. Some examples:

- Data should be hidden from other classes to prevent them from becoming dependent on it.
- Classes should be loosely coupled. That is, when possible, they shouldn't depend upon members in other class (except that overloaded operators can usually be assumed to exist).

Here are some that we haven't talked about before.

The Open Closed Principle

A class should be open for extension, but closed for modification. We should be able to extend classes, without requiring the class to be modified. Inheritance and polymorphism are important ways to accomplish this. Let's say that we have a class representing a store that carries different types of items for sale. When we sell an item, we may need to decrement inventory, determine whether to order more, and other functions that can be different depending on the type of the item. We should *not* do something like this:

```
if (item->type() == "fruit")
      SoldFruit(item);
if (item->type() == "clothes")
      SoldClothes(item);
if (item->type() = "software")
      SoldSoftware(item);
```

What if we added another type (or another ten types)? A better way to handle this is for all of the items to inherit from a general class, which has a single "sold" method that operates differently depending on the type of the item using polymorphism:

```
item->sold();
```

This abstracts away from how to deal with objects that are sold. In fact, the function doesn't need to know. Only the object itself needs to know what to do when it is sold.

One of the key ideas behind this principle is that it is better not to change working code, since you might break it. You will need to change some code, but it will be in code on the outskirts of the design instead of central. A related principle is that, whenever there is a set of alternatives, then, ideally, only one class should need to know the entire set of alternatives.

The restriction on public data and global variables is related to this principle. If other functions or classes are dependent on these things, then they can't be closed, since they must be changed if the data they are dependent on is changed.

This principle is the key to good object-oriented design. It is not always possible to achieve, but you should try to!

The Liskov Substitution Principle

Subclasses should be substitutable for their base classes.

The key here is that derived classes should inherit (and not disable) all of the behaviors of the base class. It is a strong is-a relationship. Does a square inherit all of the behaviors of a rectangle? Unfortunately, no. We can change the height of a rectangle to h and the width of a rectangle to w and get an area that is w*h. Squares do not inherit this behavior. This implies that we should not use inheritance to derive a square from a rectangle. Even though every square is a rectangle, behaviorally a square is not a rectangle. Software is really about behaviors, not properties.

Another way of looking at this is in terms of preconditions and postconditions:

- The derived class should have preconditions that are no stronger than the base class method.
- The derived class should have postconditions that are no weaker than the base class method.

The reason for this is that a base class reference may refer to a derived class. If the derived class has greater preconditions, we may try to perform operations on it that we shouldn't. If the postconditions are weaker, then we may expect some operations to occur that don't happen. (Squares have higher preconditions than the shapes that they inherit from.)

Example

As an example, let's consider a design for a university. One useful way to start is to consider the nouns that are associated with the problem (for example, in the user requirements). This won't always yield a complete set (and it may have many extra objects), but it can point you in the right direction.

Some possibilities: student, professor, course, program, grade, section, campus, building, room, etc.
Subclasses: undergraduate, graduate student, assistant professor, associate professor, professor, etc.

A further technique that helps identify members and relationships is "use-case analysis." Here we ask: What can a user do? How does the system respond? A full use-case analysis would examine each of these scenarios in detail.

Less intuitive inheritance

Less intuitive is when verbs become classes. Typically, we think of verbs as functions. Consider a financial institution with Accounts. Transactions include Deposit, Withdraw, etc. Suppose you want to add another transaction such as transferring funds. Changing the switch that would accomplish this code violates the open-closed principle. Instead if an inheritance hierarchy is created with a function *perform* to perform the transaction using polymorphism, then adding another class (open for modification) does not violate the principle: