Polymorphism

```
AShape s = new Circle(2.7); // OK.
AShape t = new Rectangle(3, 4); // OK.

t = s; // OK, the old Rectangle is gone.

Circle u = new Rectangle(5, 6); // NO!
```

- A variable of class `AShape` can be assigned any instance of subclasses of `AShape` at any time in a program. `AShape` is said to be *polymorphic*.
  - In general, a variable of a superclass can be assigned an instance of any of its subclasses, but not the other way around. Polymorphism means a class can represent any of its subclasses.

Object-oriented Programming Principles

1. Objects are the only things that can perform computations.

2. Encapsulate that which varies (a variant) into a class, and make all related variants into concrete subclasses of an "abstract class".

   - E.g., `Rectangle` and `Circle` extend `Shape`.

3. Program to the interface (or abstract class).

   - E.g., `_shape.dArea();` where `_shape` is `AShape`, not `Rectangle` or `Circle`.

The Union Pattern

- Suppose I face the problem of computing the areas of geometrical shapes such as rectangles and circles.

- OOPP #0 suggests that I build objects that are capable of computing these areas.

- The variants for this problem are the infinitely many shapes: rectangles, circles, etc.
  - OOPP #1 drives me to define concrete classes such as Rectangle and Circle, and make them subclasses of an abstract class, called AShape, which has the abstract capability of computing its area.
    * This is an example of the simplest yet most fundamental OO design pattern called the *Union Pattern*. It is the result of applying OOPP #0 and OOPP #1.

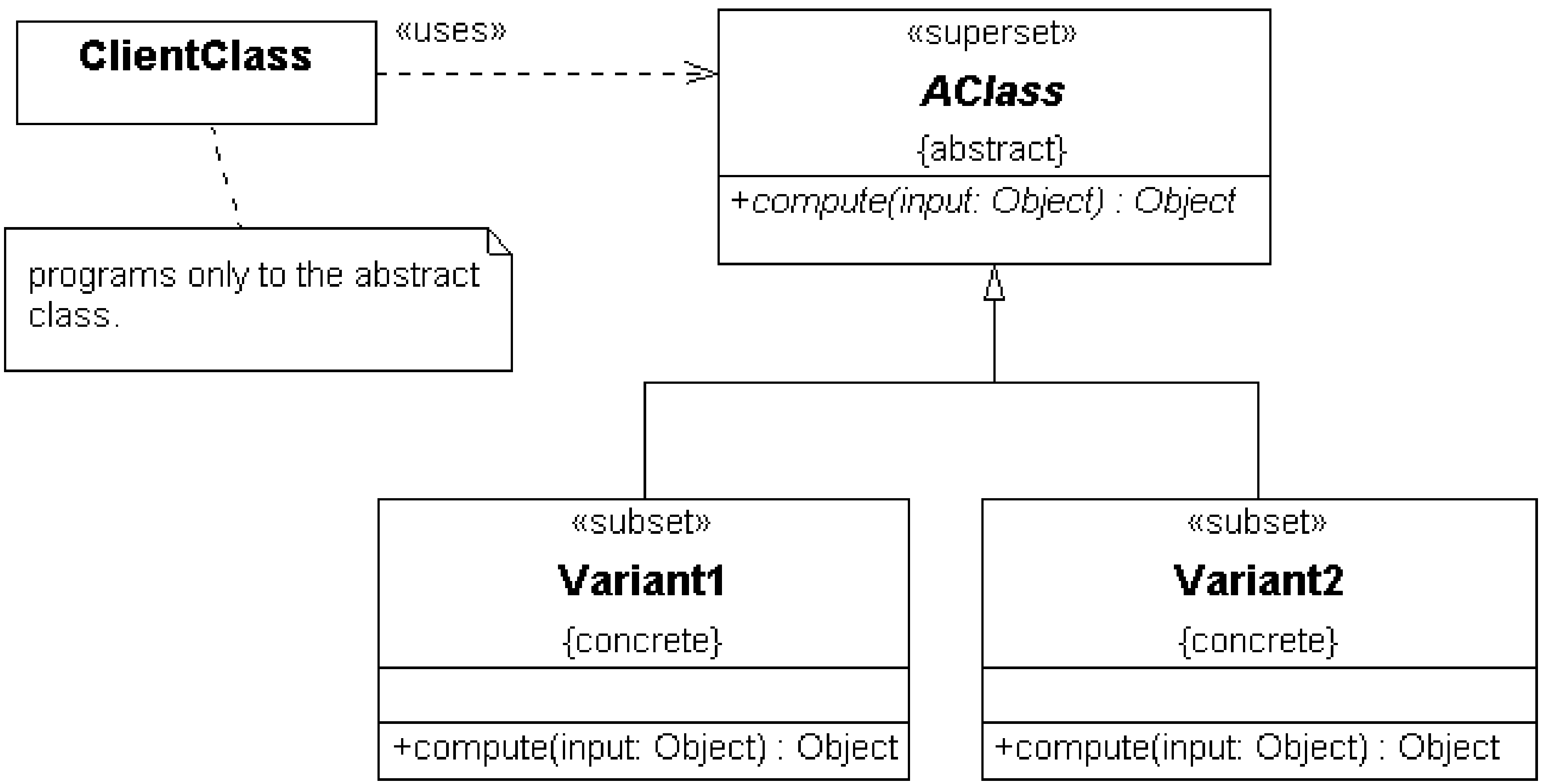
The Union Pattern (cont.)

- The Union Pattern is the result of *partitioning* the sets of objects in the problem domain into *disjoint* subsets and consists of
  - an abstract class (AClass) representing the superset of all the objects of interest,
  - several concrete subclasses (Variant1, Variant2) representing disjoint subsets of the above superset;
    - the union of these subsets equals the superset.

The Union Pattern (cont.)

- A client of the Union Pattern uses instances of the concrete subclasses (Variant1, Variant2), but should only see them as AClass objects.
  - The client class code should only concern itself with the public methods of AClass and should not need to check for the class type of the concrete instances it is working with.
    - Conditional statements to distinguish the various cases are gone, reducing code complexity and making the code easier to maintain.

The Union Pattern (cont.)



A Scheme-like List

- What is a list?
  - An ordered collection of zero or more objects.
- What are the (basic) operations supported on a list?
  - `getFirst()`
  - `getRest()`

A Scheme-like List (cont.)

```
public abstract class AList
{
  // Returns the first object in this AList, if any.
  public abstract Object getFirst();

  // Returns the tail ("rest") of this AList, if any.
  public abstract AList getRest();
  ...
```

A Scheme-like List (cont.)

```
public class NEList extends AList
{
  private Object _first;
  private AList _rest;
  ...
  // Returns the first object in this AList.
  public Object getFirst()
  {
    return _first;
  }

  // Returns the tail ("rest") of this AList.
  public AList getRest()
  {
    return _rest;
  }
  ...
```

A Scheme-like List (cont.)

```
public class EmptyList extends AList
{
  ...
  // Throws an IllegalArgumentException
  public Object getFirst()
  {
    throw new IllegalArgumentException("Empty List has no data.");
  }

  // Throws an IllegalArgumentException
  public AList getRest()
  {
    throw new IllegalArgumentException("Empty List has no tail.");
  }
  ...
```

A Scheme-like List (cont.)

```
public abstract class AList
{
  ...
  // Returns the number of elements in this AList.  Uses
  // the "helper" method helpGetLength().
  public abstract int getLength();

  // Returns the sum of ``acc'' and the number of elements
  // in the AList.
  public abstract int helpGetLength(int acc);
  ...
```

A Scheme-like List (cont.)

```
public class NEList extends AList
{
  ...
  // Asks the tail for help to compute the length,
  // passing it an accumulated length of 1.
  public int getLength()
  {
    return _rest.helpGetLength(1);
  }

  // Adds 1 to the accumulated length and pass it down to the
  // tail for help to compute the length.
  public int helpGetLength(int acc)
  {
    return _rest.helpGetLength(acc + 1);
  }
```

A Scheme-like List (cont.)

```
public class EmptyList extends AList
{
  ...
  // Returns 0.
  public int getLength()
  {
    return 0;
  }

  // Returns the accumulated length, since this is the end of the list.
  public int helpGetLength(int acc)
  {
    return acc;
  }
  ...
```