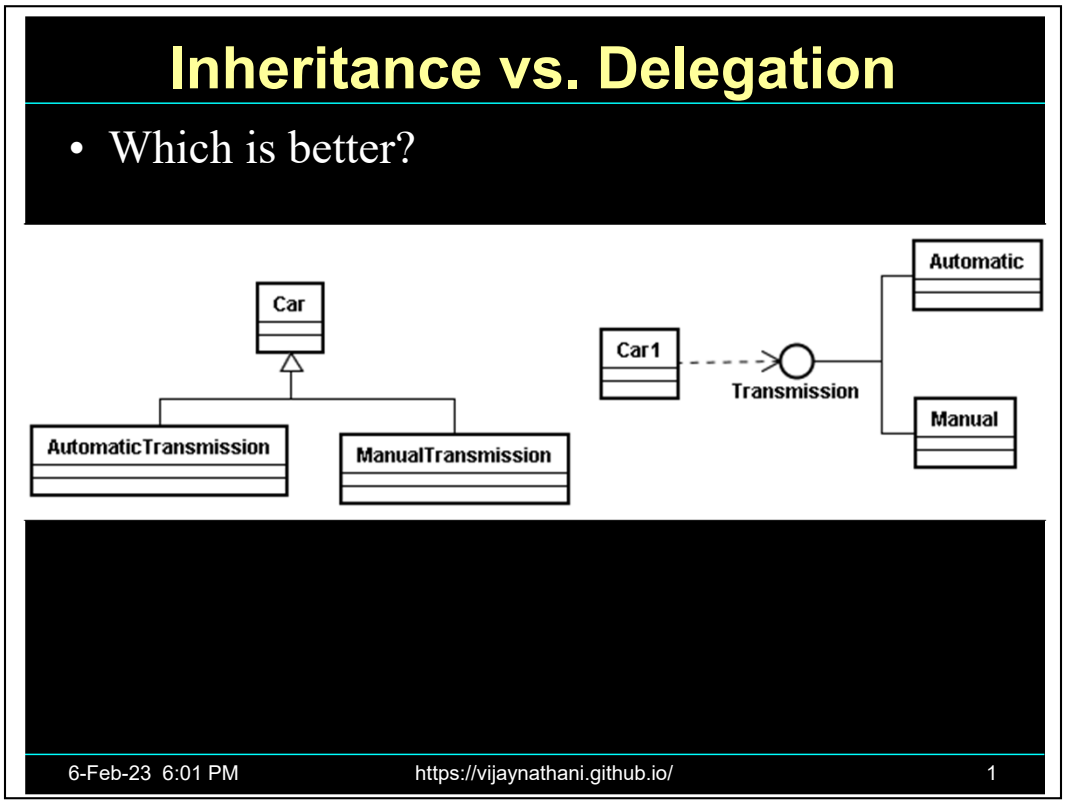


Composition Advantages

- Contained objects are accessed by the containing class solely through their interfaces
- "Black-box" reuse, since internal details of contained objects are not visible
- Good encapsulation
- Fewer implementation dependencies
- Each class is focused on just one task
- The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type

Composition Disadvantages

- Resulting systems tend to have more objects
- Interfaces must be carefully defined in order to use many different objects as composition.

Prefer Composition/Interfaces to Inheritance.

- Composition implies has-a or uses-a relationship.
- Inheritance implies is-like-a relationship.

While using inheritance, the Liskov's Substitution Principle must not be violated.

- Also, Avoid deep inheritance trees

- As far as possible, it is preferable to inherit from an abstract class.

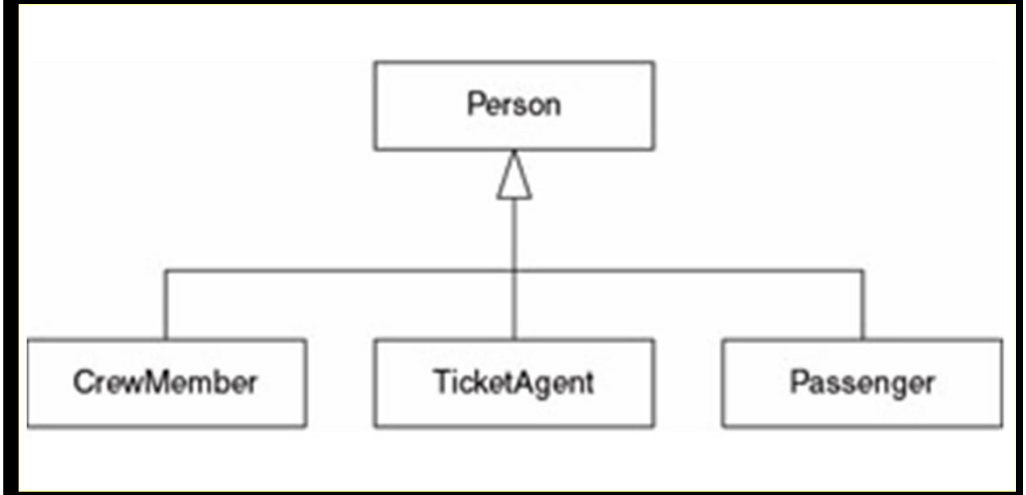
Generalization Advantages

- New implementation is easy, since most of it is inherited
- Easy to modify or extend the implementation being reused

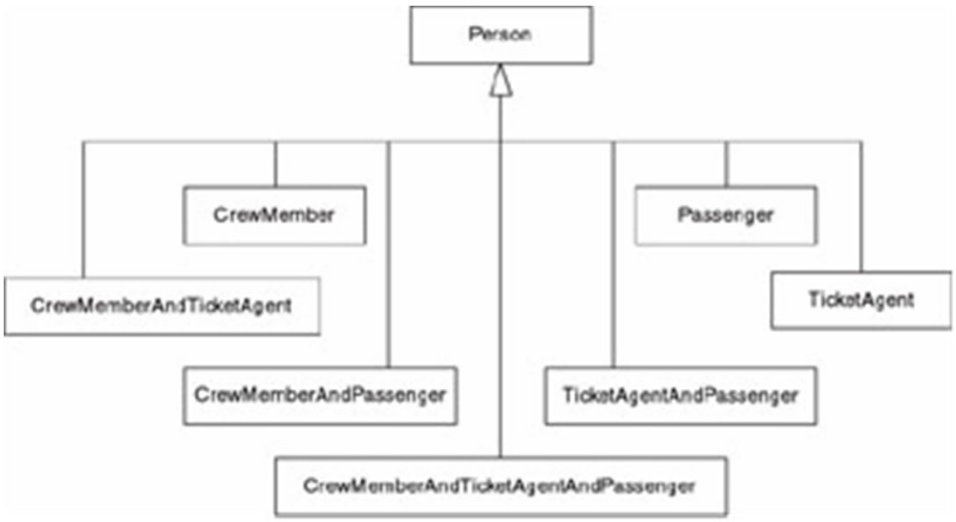
Generalization Disadvantages

- Breaks encapsulation, since it exposes a subclass to implementation details of its super class
- "White-box" reuse, since internal details of super classes are often visible to subclasses
- Subclasses may have to be changed if the implementation of the super class changes
- Implementations inherited from super classes can not be changed at runtime

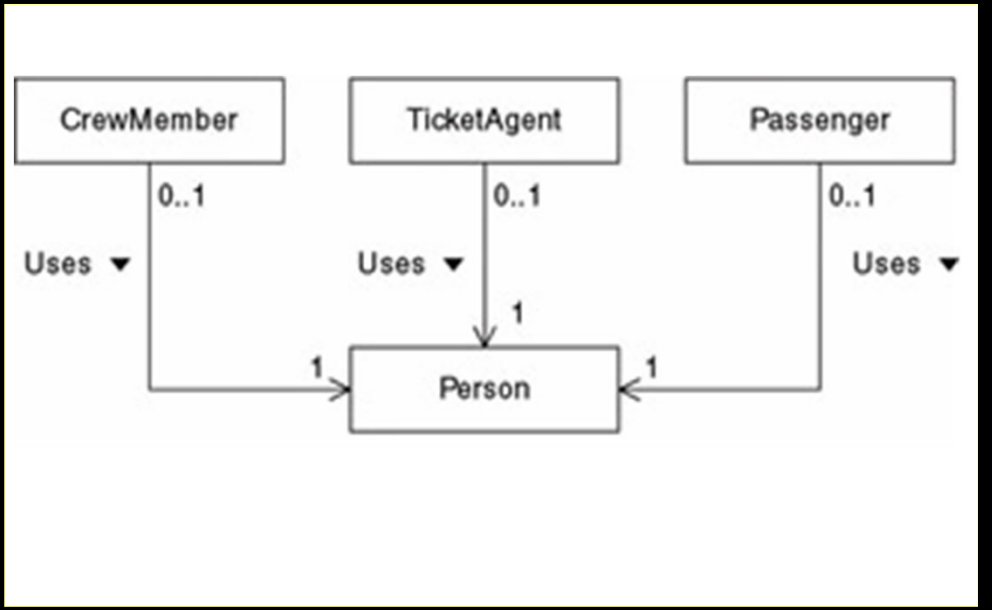
Airline Reservation System



Same Person – Multiple roles



A person can change roles Now



Stack is not ArrayList

```
class Stack extends ArrayList {
    private int topOfStack = 0;
    public void push( Object article ) {
        add( topOfStack++, article ); }
    public Object pop() {
        return remove( --topOfStack );}
    public void pushMany( Object[] articles ) {
        for( int i = 0; i < articles.length; ++i )
            push( articles[i] ); }
}
Stack aStack = new Stack();
aStack.push("1");
aStack.push("2");
aStack.clear(); //Error
```


Even a class as simple as this one has problems. Consider what happens when a user leverages inheritance and uses the ArrayList 's clear() method to pop everything off the stack.

The code compiles just fine, but since the base class doesn't know anything about the index of the item at the top of the stack (topOfStack), the Stack object is now in an undefined state. The next call to push() puts the new item at index 2 (the current value of the topOfStack), so the stack effectively has three elements on it, the bottom two of which are garbage.

One (hideously bad) solution to the inheriting-undesirable-methods problem is for Stack to override all the methods of ArrayList that can modify the state of the array to manipulate the stack pointer. This is a lot of work, though, and doesn't handle problems such as adding a method like clear() to the base class after you've written the derived class.

To solve the problem, use delegation:

```
class Stack {
    private int topOfStack = 0;
    private ArrayList theData = new ArrayList();
    public void push( Object article ) {
          theData.add( topOfStack++, article );
    }
    public Object pop() {
          return theData.remove( --topOfStack );
    }
    public void pushMany( Object[] articles ) {
         for( int i = 0; i < articles.length; ++i )
             push( articles[i] );
    }
    public int size() // current stack size.
    {  return theData.size();
    }
}
```

My Stack in C++

```
class Stack : public vector<void*> {
      int topOfStack = 0;
public:
      void push(void* article)
            { push_back(article); }
      void* pop() {
            void * r = back();
            pop_back();
            return nullptr;          }
};
void StackDemo() {
      Stack aStack;        int a = 1, b = 2;
      aStack.push(&a);    aStack.push(&b);
      aStack.erase(aStack.erase(aStack.begin()));
}
```


Bad code

What's the output?

```
public class ci1<T> extends HashSet<T> {
  private int addCount = 0;
  public ci1() {}
  public ci1 (Collection<T> c) {super(c);}
  public ci1 (int initCap, float loadFactor) {
      super(initCap, loadFactor); }
  @Override public boolean add(T o) {
      addCount++; return super.add(o); }
  @Override public boolean addAll(
            Collection<? extends T> c) {
      addCount += c.size(); return super.addAll(c);
  }
  public int getAddCount() {return addCount; }
  public static void main(String[] args) {
      ci1<String> s = new ci1<String>();
      s.addAll(Arrays.asList(new String[]
                  {"Snap","Crackle","Pop"}));
      System.out.println(s.getAddCount()); } }
```

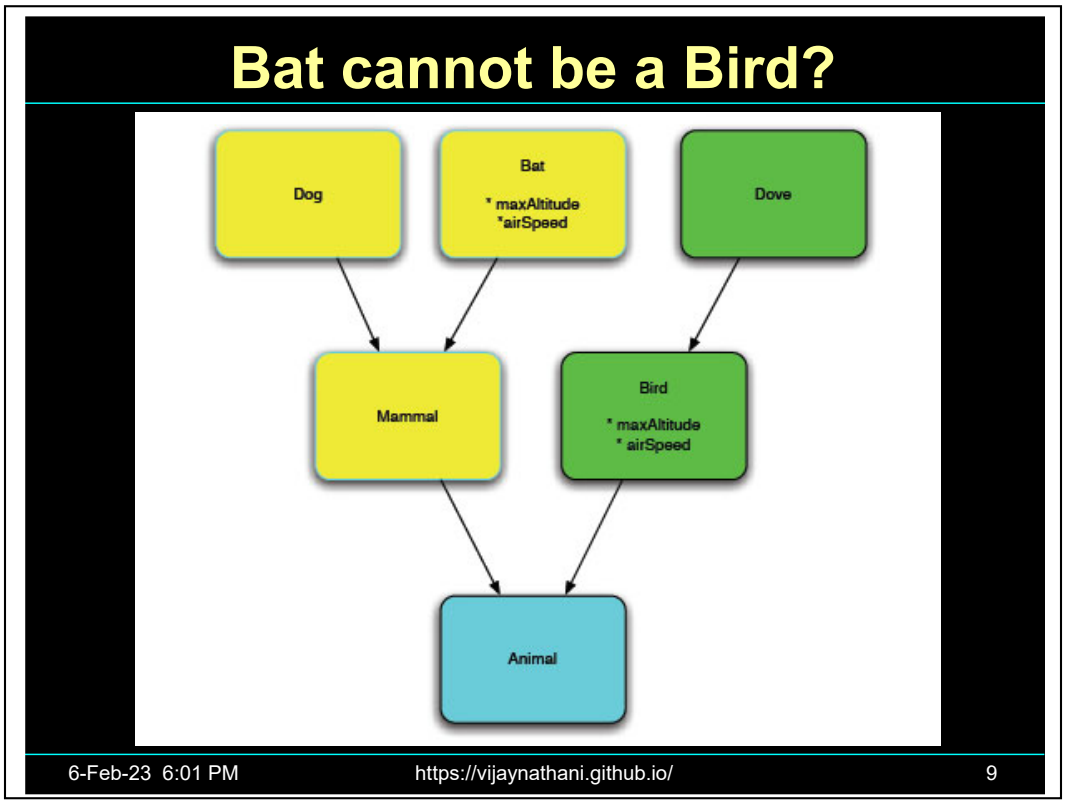

The output comes as 6 instead of 3.

The class should implement the interface Set and use HashSet internally.

The **fragile base class problem** is a fundamental architectural problem of object-oriented programming systems where base classes (super classes) are considered "fragile" because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction. The programmer cannot determine whether a base class change is safe simply by examining in isolation the methods of the base class.

C++ version

```
class MySet : public set<int> {
public: virtual void addNumber(int n) { insert(n); }
        virtual void addAllNumbers(vector<int> &numbers) {
            for (auto it = numbers.begin(); it != numbers.end(); it++)
                addNumber(*it); }
};
class ci1 : public MySet {
public: int addCount = 0;
        virtual void addNumber(int n) override {
                addCount++; MySet::addNumber(n); }
        virtual void addAllNumbers(vector<int> &numbers) override {
                addCount += numbers.size();
                MySet::addAllNumbers(numbers);   }
};
void MySetMain() {       vector<int> numbers = { 1, 2, 3 };  ci1 s;
        s.addAllNumbers(numbers);
        cout << s.addCount << endl;
}
```



Bat has a flying capability

Dove has a flying capability

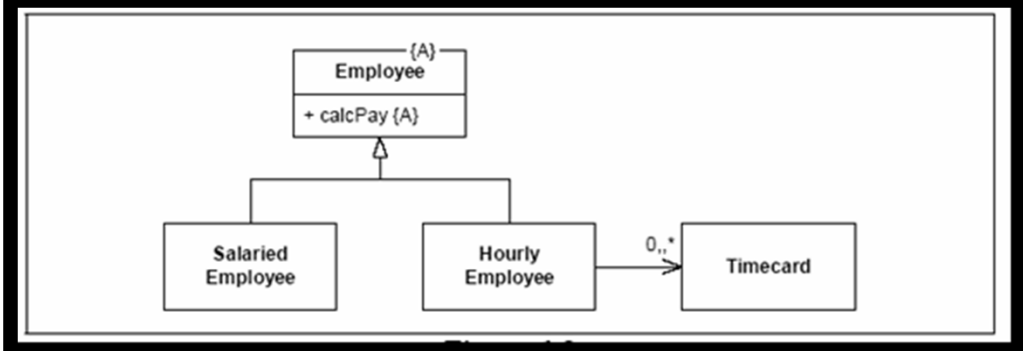
is better than

Bat is a flying creature

Dove is a flying creature

Liskov Substitution Principle

- All derived classes must be substitutable for their base class.



6-Feb-23 6:01 PM https://vijaynathani.github.io/ 10

Now we have a Category of Employee – “Volunteer”, who does not receive salary

E.g. Ellipse and Circle. If some class sets attributes and prints major and minor axis hard coded then it is a problem.

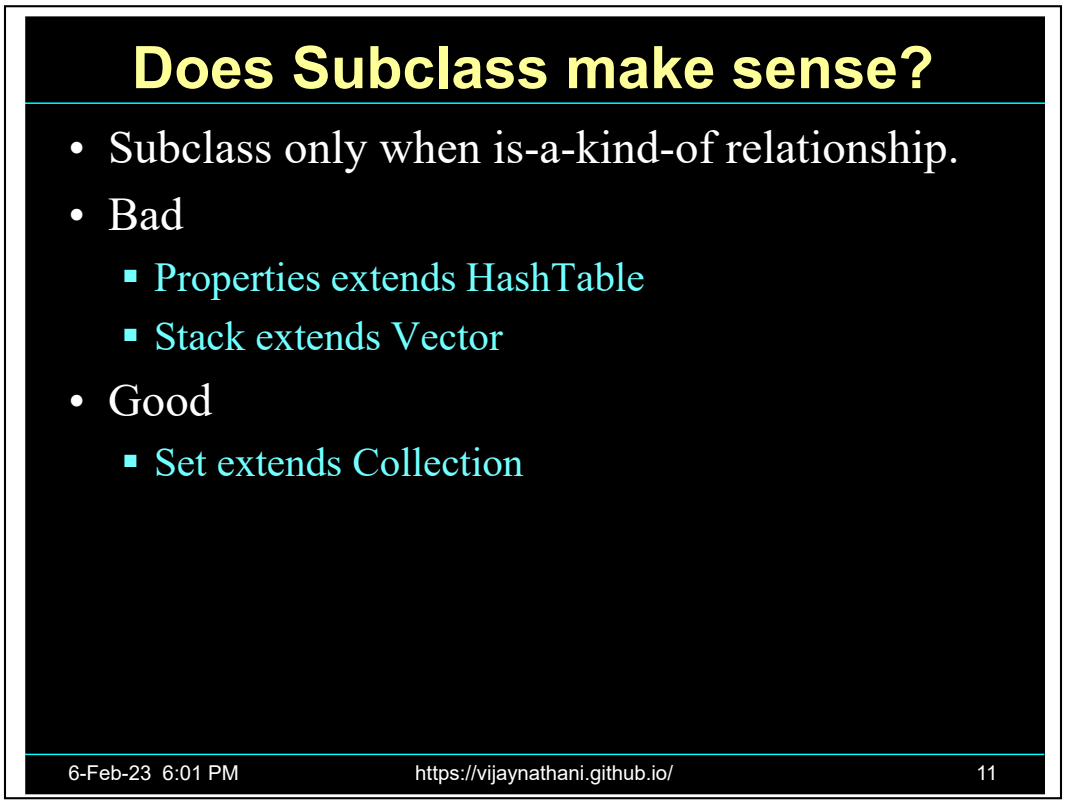
The Liskov Substitution Principle (LSP) makes it clear that the ISA relationship is all about behavior.

Violation of this law leads to usage of instanceof operator or throwing of exceptions for certain functions in a class.

E.G Deriving Square from Rectangle violates this principle because Rectangle has two functions: setWidth and setHeight.

E.g. CarOwner being derived from Car and Person.

Inheritance should preferably be done from abstract classes with minimal code.



==========

Assume that you have an inheritance hierarchy with Person and Student. Wherever you can use Person, you should also be able to use a Student, because Student is a subclass of Person. At first this might sound like that's always the case automatically, but when you start thinking about reflection (reflection is a technique for being able to programmatically inspect the type of an instance and read and set its properties and fields and call its methods, without knowing about the type beforehand), for example, it's not so obvious anymore. A method that uses reflection for dealing with Person might not expect Student.

The reflection problem is a syntactical one. Martin uses a more semantically example of Square that is a Rectangle. But when you use SetWidth() for the Square, that doesn't make sense, or at least you have to internally call SetHeight() as well. A pretty different behavior from what Rectangle needs.

==========

Design & Document for inheritance

Otherwise prohibit inheritance

Conservative Policy:

All concrete classes are final

Never override a concrete function.

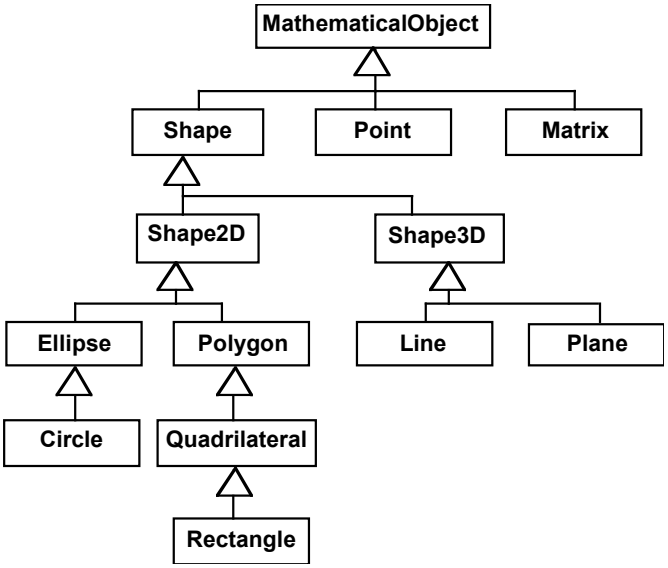
Bad

Many concrete classes in Java are not final

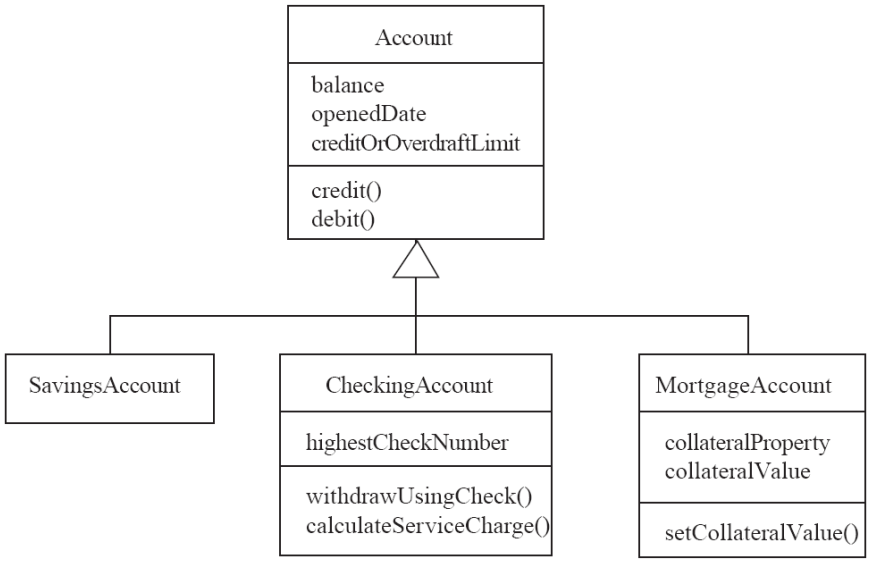
Good

AbstractSet, AbstractMap

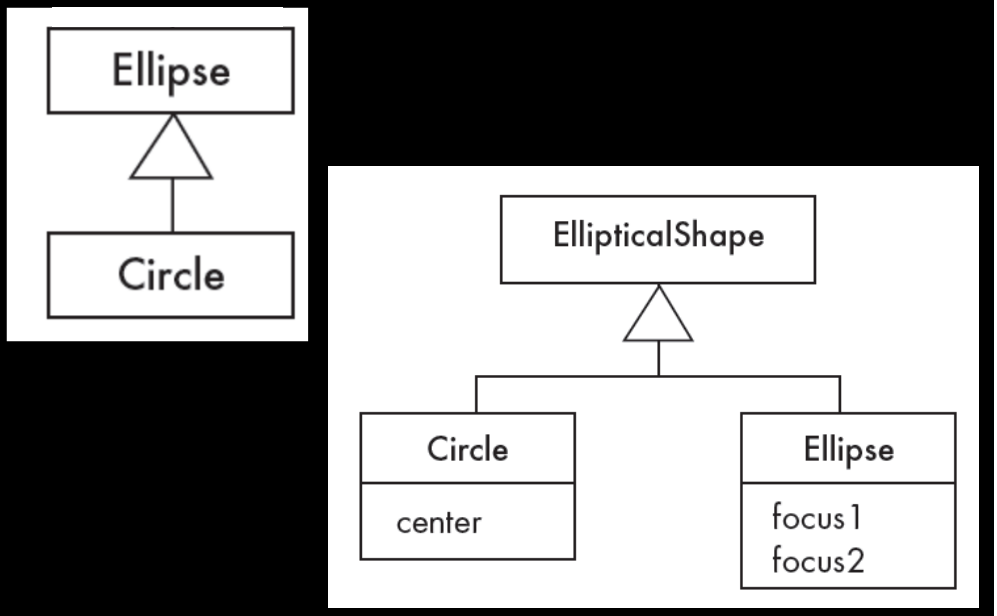
Avoid Deep Inheritance



All Inherited Features should make sense in Subclasses



Reorganize


Ellipse
Circle
EllipticalShape
Circle
center
Ellipse
focus1
focus2

Inheritance rules

- Prefer Delegation over Inheritance
- All instance variables and functionality of base class should be applicable to derived class
- Prefer interfaces to Abstract base class
- Liskov Substitution principle
- Avoid deep inheritance hierarchy
- Prefer to extend abstract classes, if inheritance has to be used.

Q72Inheri: Employee, LinkList

Q31 - UserAccount

Q71inheri – CourseCatalog Q52 – NormalPayment Q54 – Account

Q74 – BitmapButton Q75 – PropertyFileWriter

Avoid deep Inheritance Hierarchy: Q76