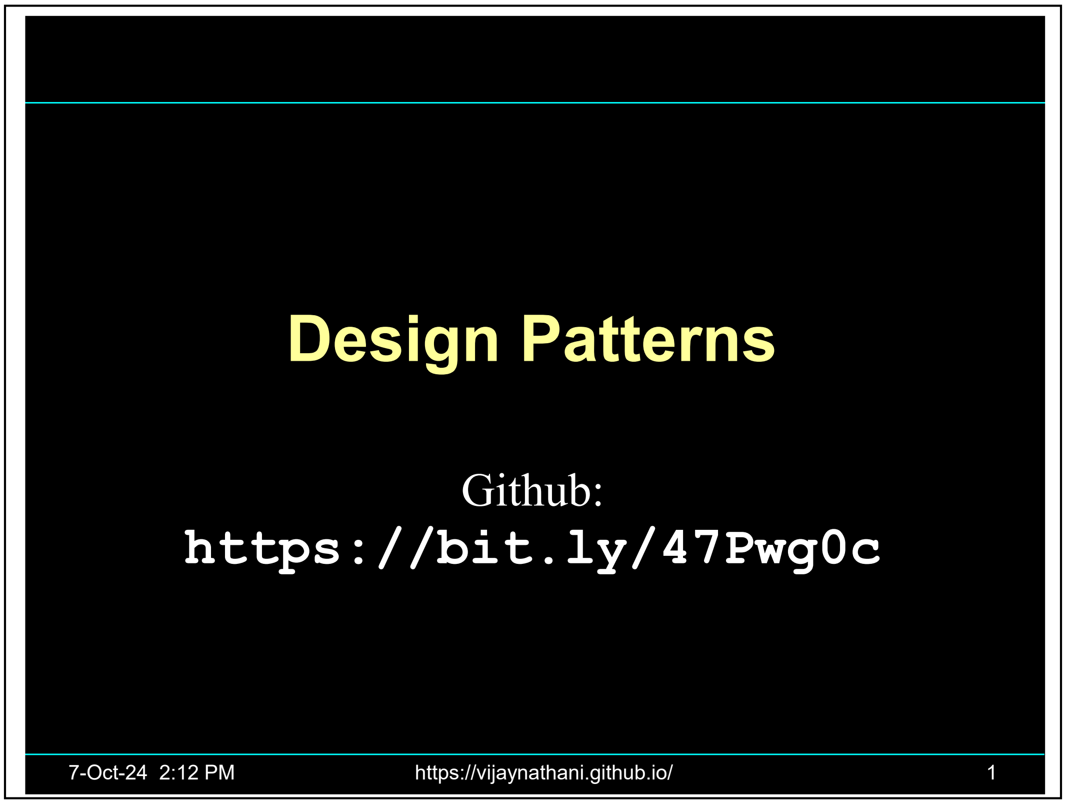


Design Patterns

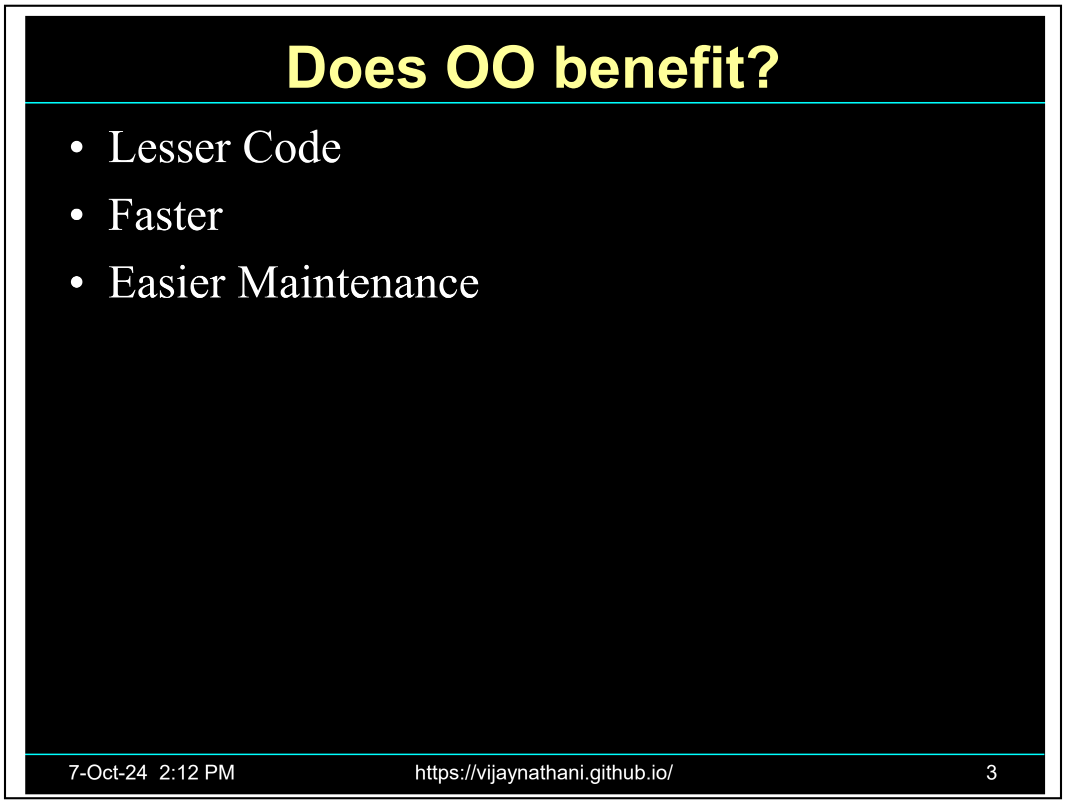
Github:
`https://bit.ly/47Pwg0c`

It’s different

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 2

Let there be no doubt that object-oriented design is fundamentally different than traditional structured design approaches. It requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of structured design culture. – Grady Booch

Many people tell the story of the CEO of a software company who claimed that his product would be object oriented because it was written in C++. Some tell the story without knowing that it is a joke.



Does OO benefit?

- Lesser Code
- Faster
- Easier Maintenance

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 3

The product of object thinking is software that manifests simplicity and composability, which lead, in turn, to adaptability, flexibility, and evolvability.

A 1 million-line program, written in with non-OO concepts can be duplicated in OO with 100K LOC or fewer.

Time is delivery is redacted by at least 50% and usually 70%.

Project that took 2 years can complete in 8 to 12 months.

Exceptions:

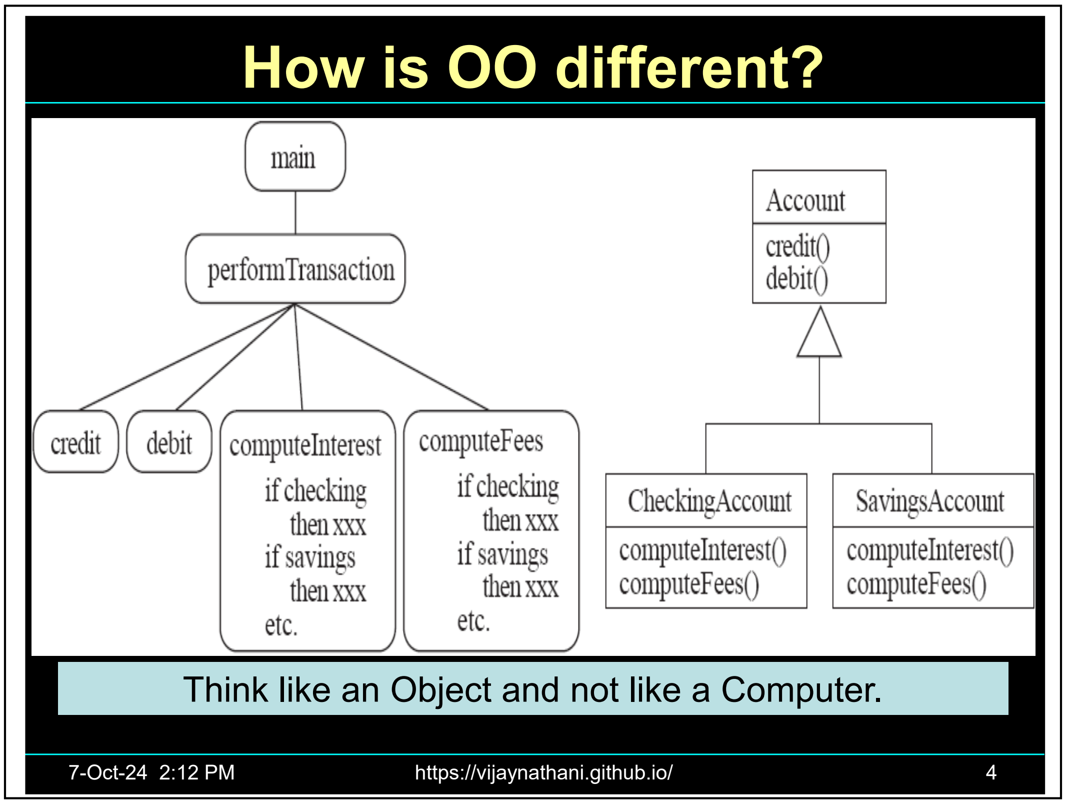
•Device driver written in 100 lines of assembly

•Internet search engine needs “database thinking”

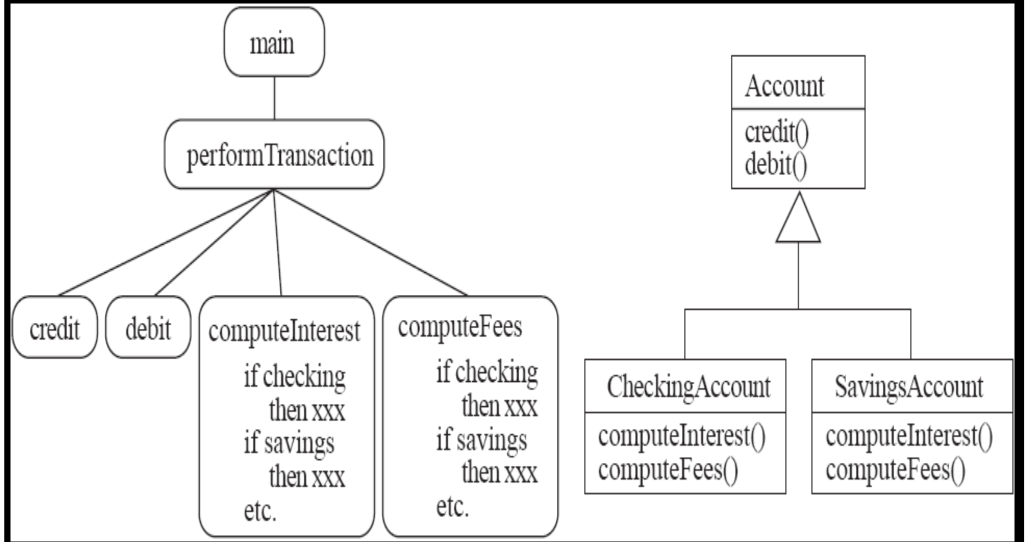
• Network Router (I/O performance)

•Mobile phone (memory footprint)

•Embedded Sensor (power consumption)



How is OO different?



Think like an Object and not like a Computer.

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 4

How is object thinking different from thinking like a computer?

- Object thinking involves a very different means of solution – a cooperating community of virtual persons
- Object thinking focuses our attention on the problem space rather than the solution space

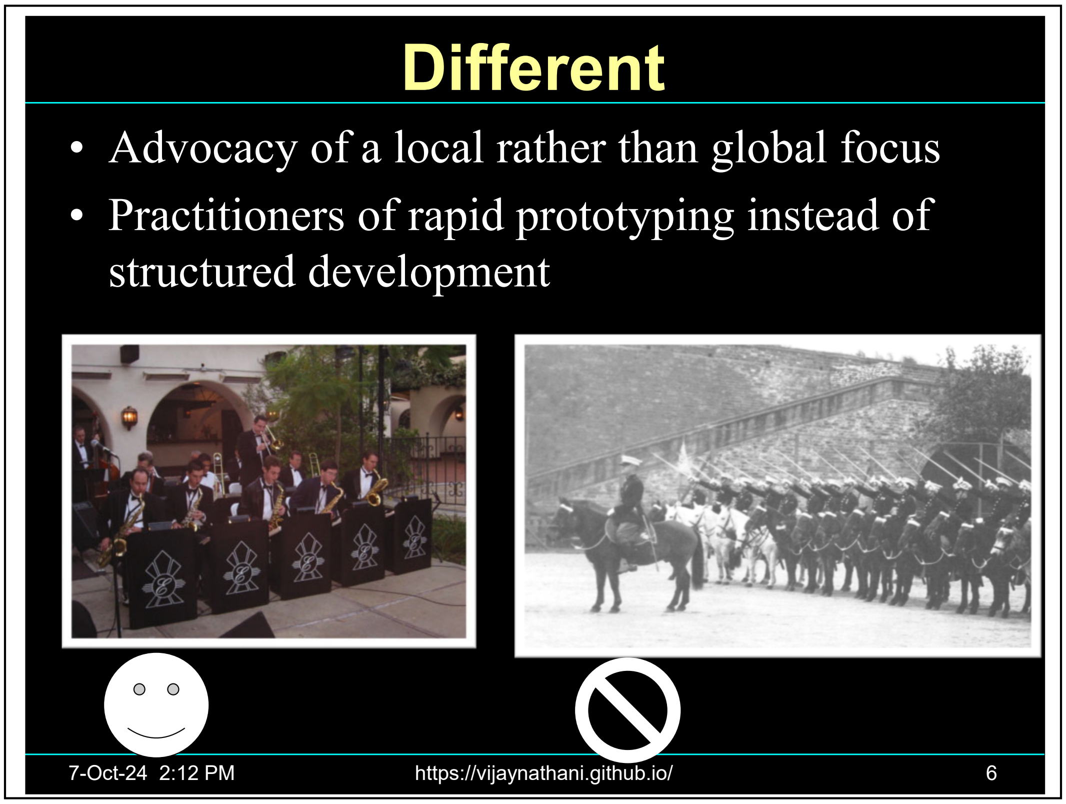
OO Design is an Art

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 5

OOAD books and classes can only enhance the innate talents of individuals and make them the best OO engineers they can be.

The education of artists is not focused on technique, process or tools.

> The majority of an art education combines ideas, history, appreciation, experience and constructive criticism.



Different

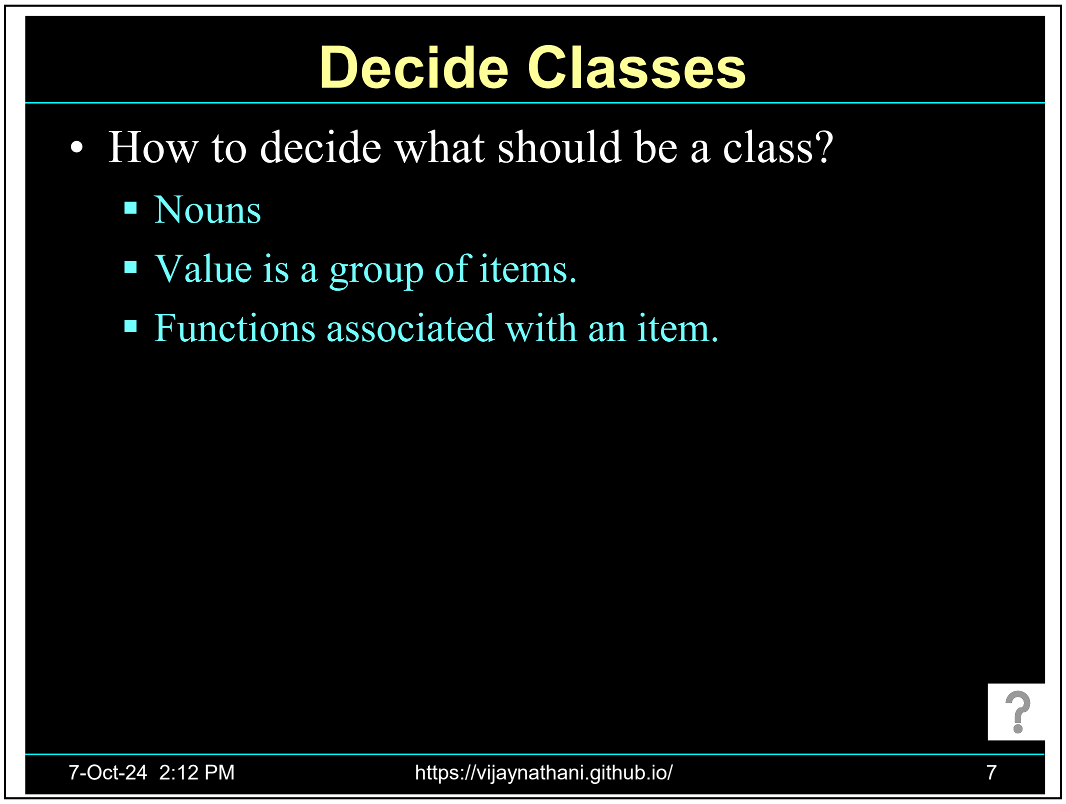
- Advocacy of a local rather than global focus
- Practitioners of rapid prototyping instead of structured development

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 6

Collaborative rather than imperial management style

OOD -

- Commitment to design based on coordination and cooperation rather than control
- Driven by internal capabilities instead of conforming to external procedures.



Decide Classes

- How to decide what should be a class?
  - Nouns
  - Value is a group of items.
  - Functions associated with an item.

?

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 7

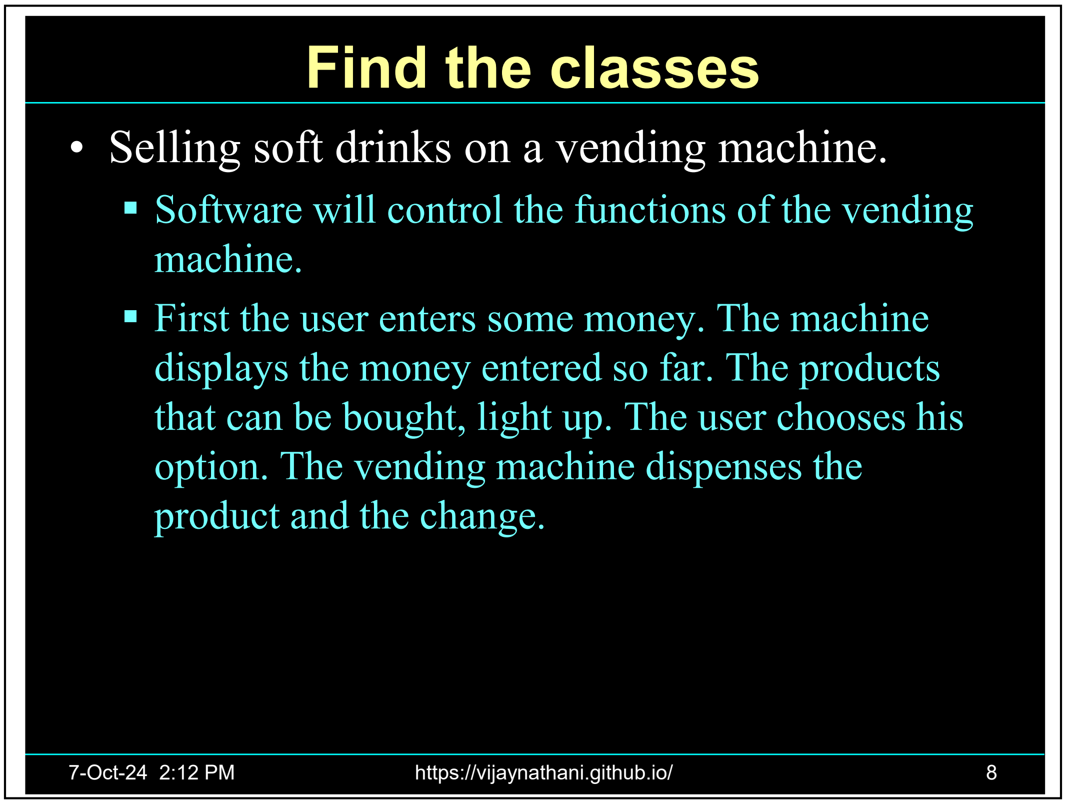
Q022: telLocalNumber

Q01: IsSameString

Class should be highly cohesive

A Class should have a single well focused purpose.

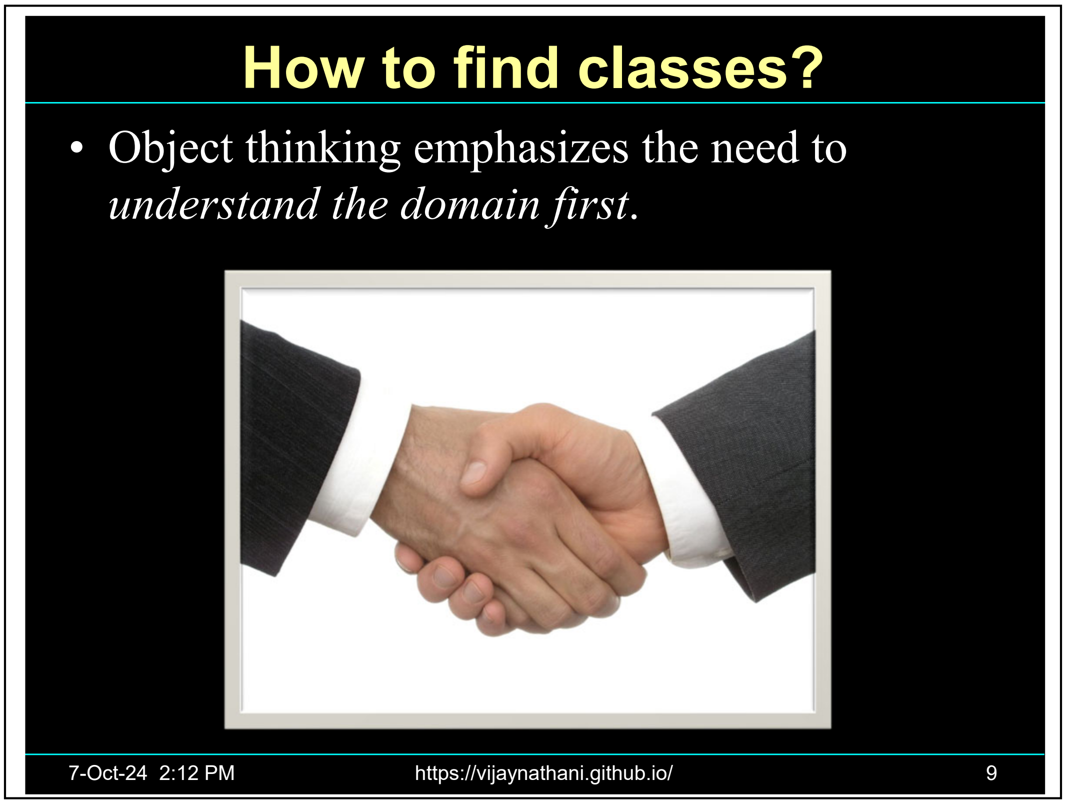
A Class should do one thing and do it well.



Find the classes

- Selling soft drinks on a vending machine.
  - Software will control the functions of the vending machine.
  - First the user enters some money. The machine displays the money entered so far. The products that can be bought, light up. The user chooses his option. The vending machine dispenses the product and the change.

Answer: VendingMachine, MoneyBox, Screen, PriceList, SoftDrink, SoftDrinkList, SoftDrinkDispenser, Safe



How to find classes?

- Object thinking emphasizes the need to *understand the domain first*.

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 9

Simulation of a problem domain drives object discovery and definition

A truism of software development is that the most costly mistakes are made the first day. Why?

Simply for a lack of knowledge.

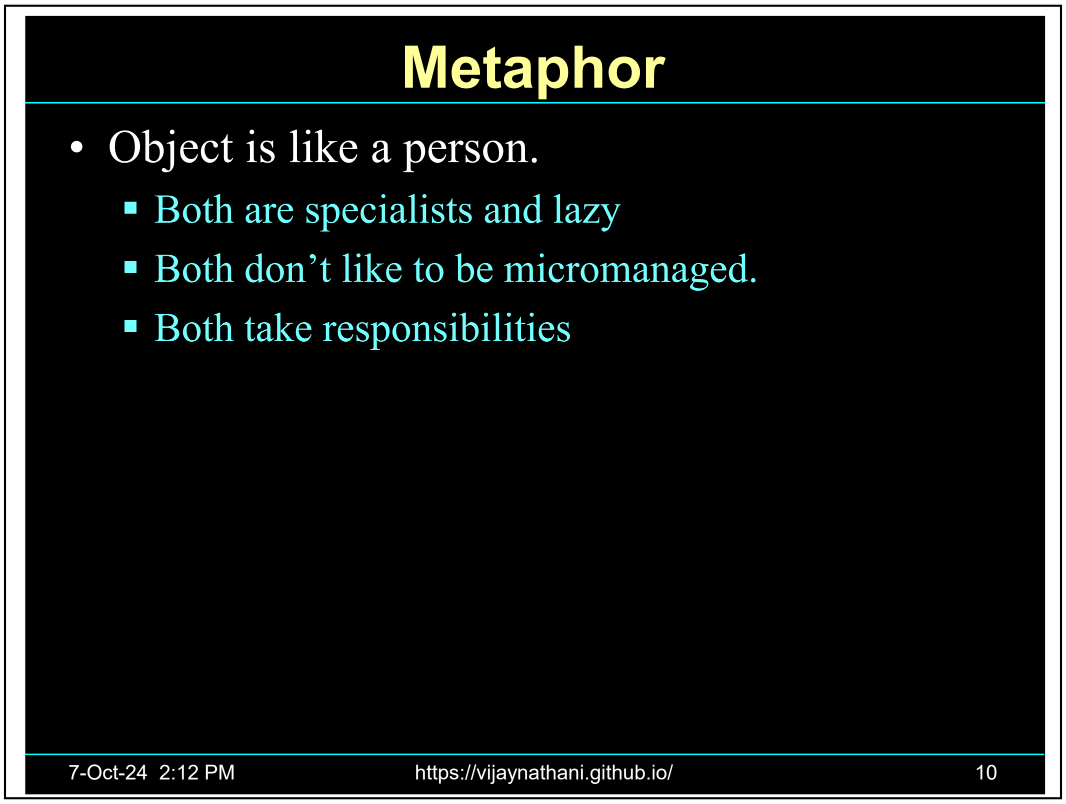
Developers anticipate how the computer is going to implement your software before trying to understand how the software should simulate some part of the domain in which it is going to be used.

It’s never appropriate to tell yourself, “This is what the code will look like, so I need an object to hold these parts of the code, and another to hold these parts, and another to make sure these two do what they are told to do when they are told to do it,” which is precisely what structured development tempts you to do.

Perhaps the greatest benefit of object thinking is that of helping you start off in the right direction. Object thinking does this by emphasizing the need to *understand the domain first*.

Software expertise does not trump domain expertise. The longer a software developer works in a domain, the more effective her software work will be.

Most books and methods addressing how to do object development recommend that the object discovery process begin with underlining the nouns (names) in a domain or problem description. While it is true that many of those nouns will indeed turn out to be viable objects, it is unlikely that any written description will be sufficiently complete or accurate to meet the needs of domain



Metaphor

- Object is like a person.
  - Both are specialists and lazy
  - Both don’t like to be micromanaged.
  - Both take responsibilities

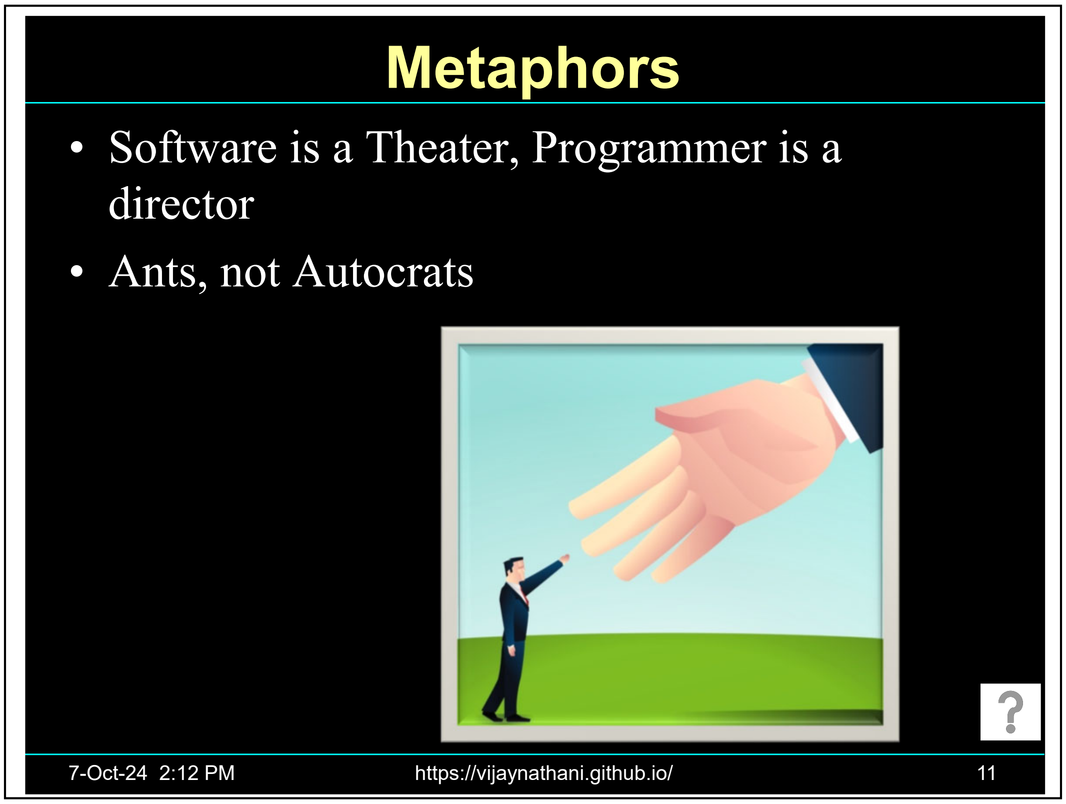
7-Oct-24 2:12 PM https://vijaynathani.github.io/ 10

Distributed cooperation and communication must replace hierarchical centralized control as an organizational paradigm. E.g. A traffic signal and cars.

Like people, software objects are specialists and lazy. A consequence of both these facts is the distribution of work across a group of objects. Take the job of adding a sentence to a page in a book. Granted, it might be quite proper to ask the book, “Please replace the sentence on page 58 with the following.” (The book object is kind of a spokesperson for all the objects that make up the book.) It would be quite improper, however, to expect the book itself to do the work assigned. If the book were to do that kind of work, it would have to know everything relevant about each page and page type that it might contain and how making a simple change might alter the appearance and the abilities of the page object. Plus the page might be offended if the book attempted to meddle with its internals.

The task is too hard (lazy object) and not the book’s job (specialist object), so it delegates—merely passes to the page object named #58 the requested change. It’s the page object’s responsibility to carry out the task. And it too might delegate any part of it that is hard—to a string object perhaps.

Objects, like the people we metaphorically equate them to, can work independently and concurrently on large-scale tasks, requiring only general coordination. When we ask an object collective to perform a task, it’s important that we avoid micromanagement by imposing explicit control structures on those objects. You don’t like to work for a boss who doesn’t trust you and allow you to do your job, so why should your software objects put up with similar abuse?



Metaphors

- Software is a Theater, Programmer is a director
- Ants, not Autocrats

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 11

Q57srp – Students Q58srp – Servers

Hierarchical and centralized control is the anathema in OO. Complex systems are characterized by simple elements, acting on local knowledge with local rules, give rise to complicated patterned behavior. Object can inherit like a person.

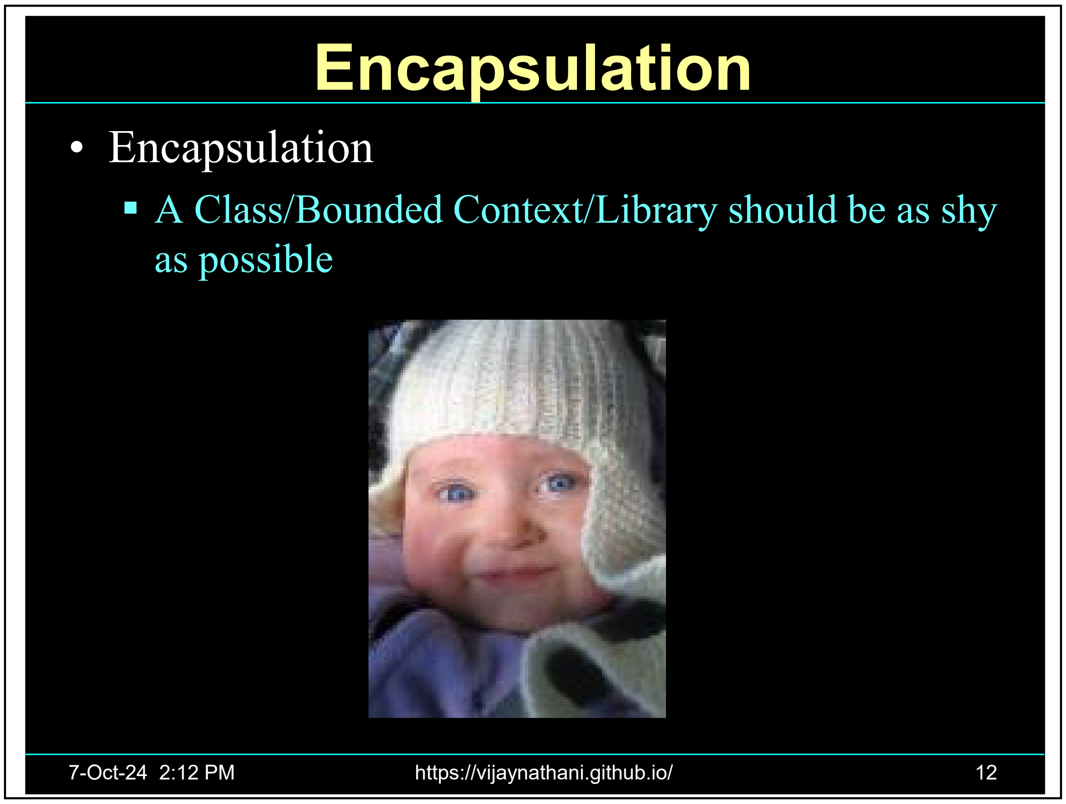
For example, suppose an airplane object has a responsibility to report its location. This is a hard task because the location is constantly changing; a location is a composite structure (latitude, longitude, altitude, direction, speed, and vector); the values of each part of that structure come from a different source; and someone has to remember who asked for the location and make sure it gets back to them in a timely fashion. If the task is broken up so that

- The airplane actually returns a location object to whoever asked for it after appending its ID to the location so that there is no confusion about who is where. (We cannot assume that our airplane is the only one reporting its location at any one time.)

- An instrument cluster keeps track of the instruments that must be asked for their current values and knows how to ask each one in turn for its value (a collection iterating across its contents).

- An instrument merely reports its current value.
- A location object collects and returns a set of label:value pairs (altitude:15,000 ft.).

None of the objects do anything particularly difficult, and yet collectively they solve a complicated problem that would be very hard for any one of them to accomplish individually.



Encapsulation

- Encapsulation
  - A Class/Bounded Context/Library should be as shy as possible

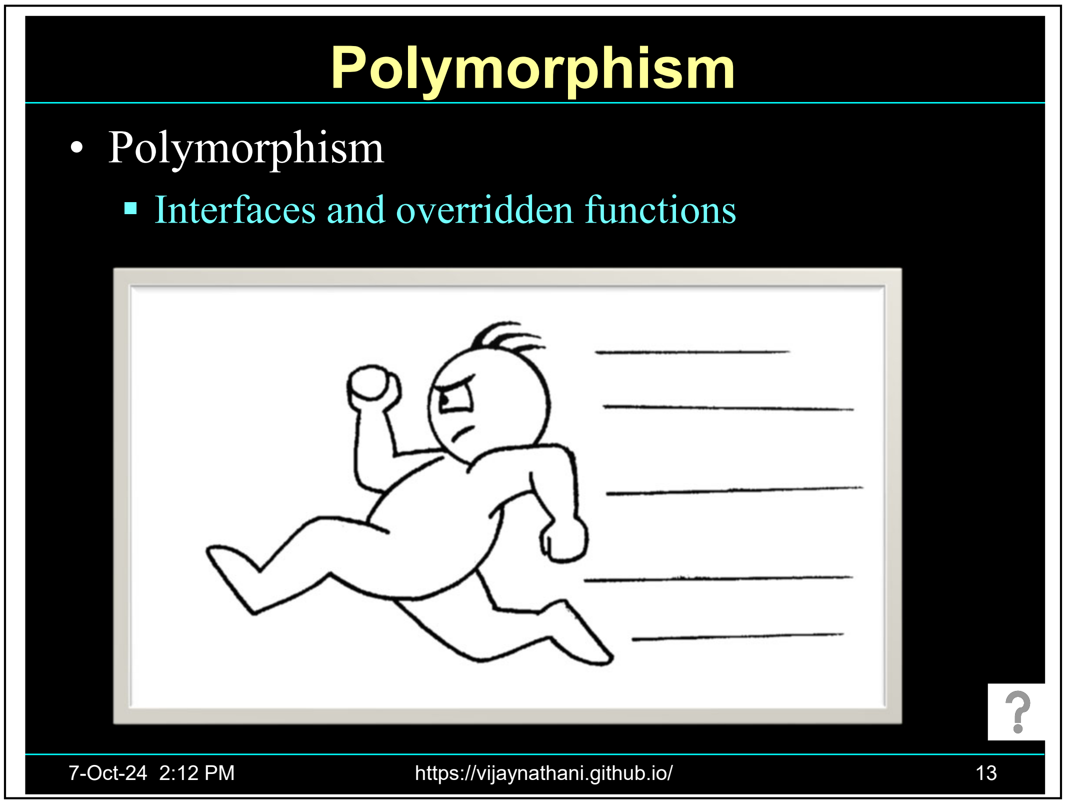
7-Oct-24 2:12 PM https://vijaynathani.github.io/ 12

It is about managing complexity

Put state information in the class that works on it. Collaborating classes are unaware of the internal state of this object.

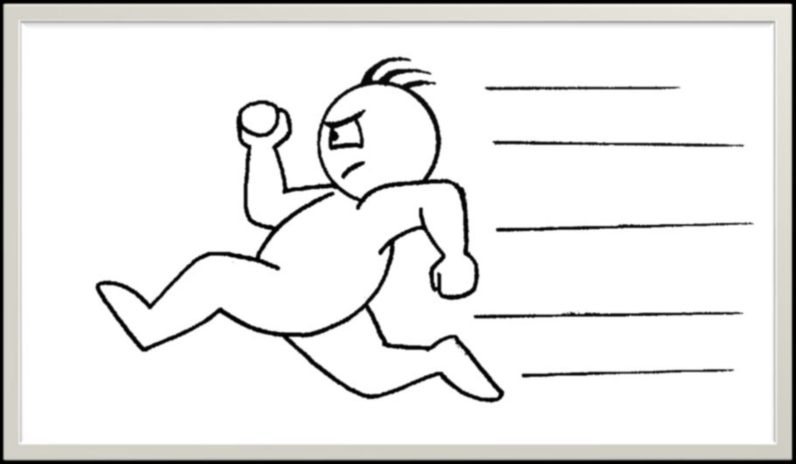
Keep a variable/function private, if possible. Use minimum visibility.

Every module should have a secret. If it does not have a secret, why does it exist?



Polymorphism

- Polymorphism
  - Interfaces and overridden functions

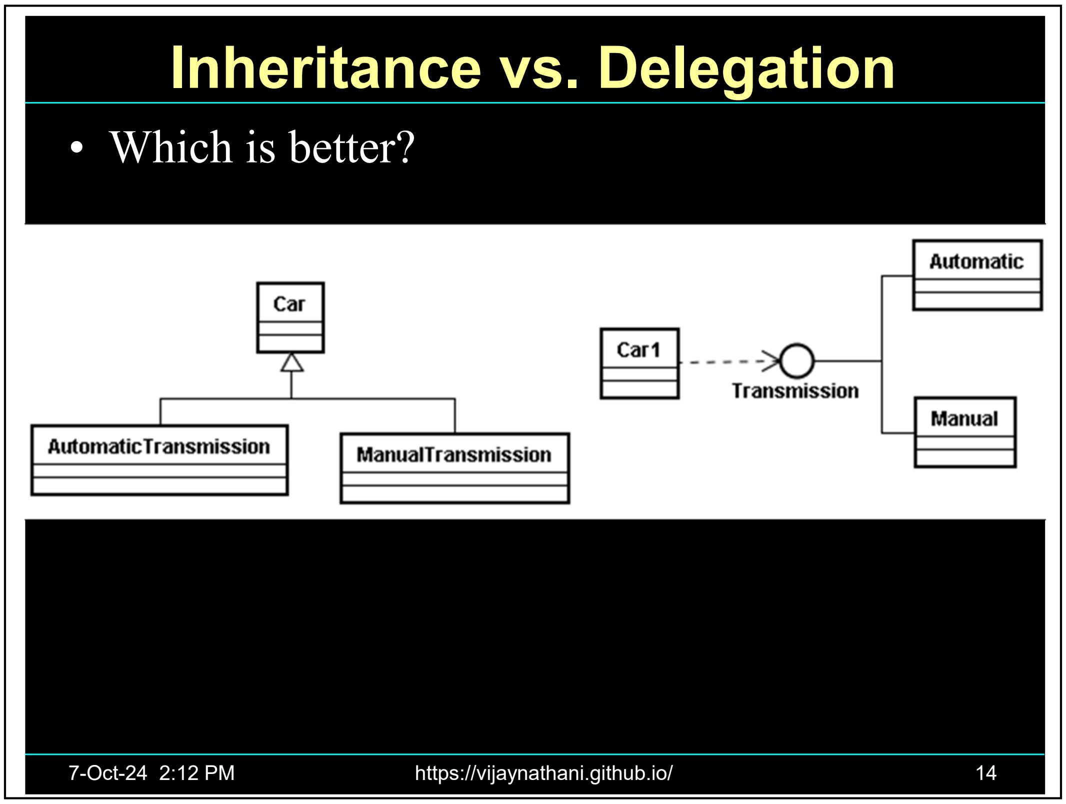


?

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 13

Q30 – Shape; Q40Smell: Courses weekly, range, list; Q43smell: home address / work address

Q36 – USD, RMB;



Inheritance vs. Delegation

- Which is better?

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.

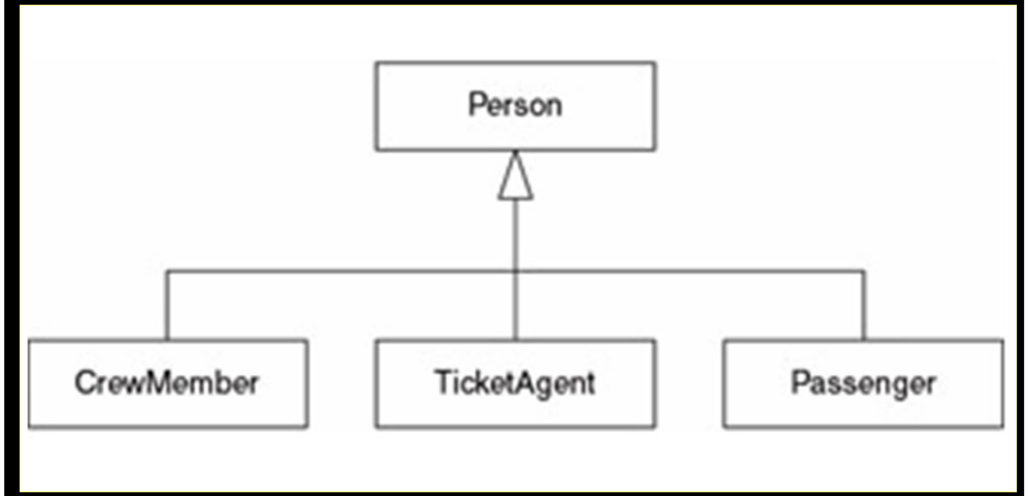
Generalization Advantages

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

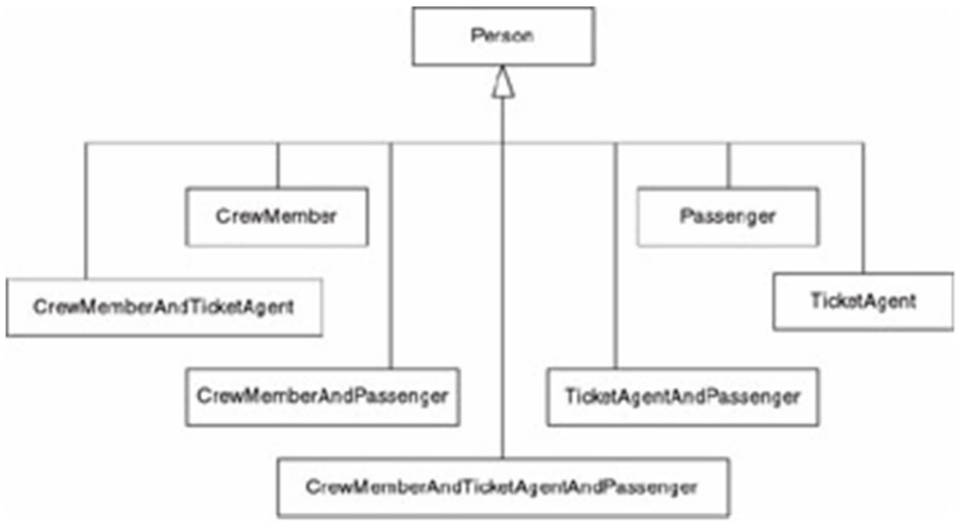
Generalization Disadvantages

- 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

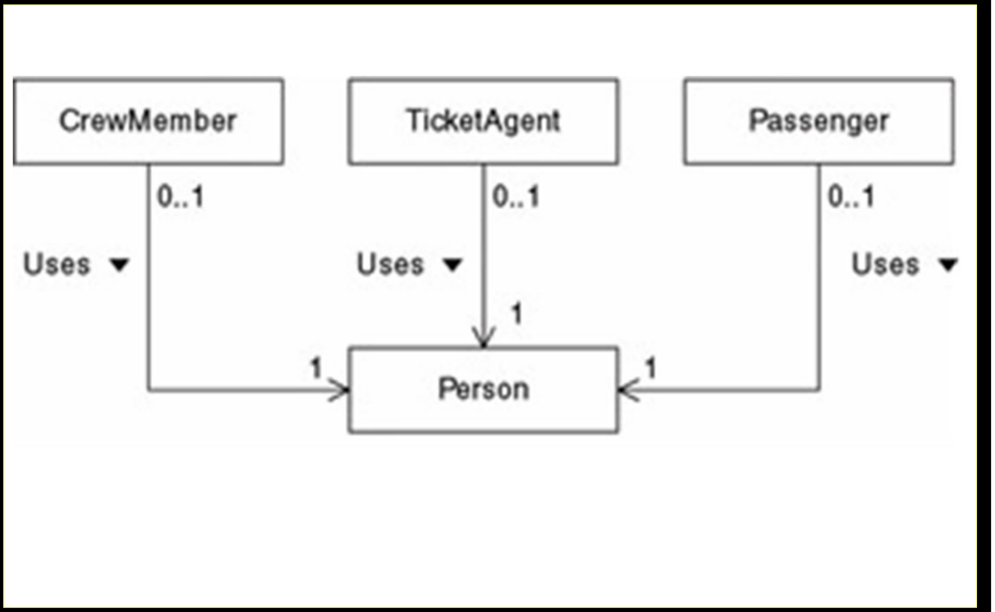
Airline Reservation System

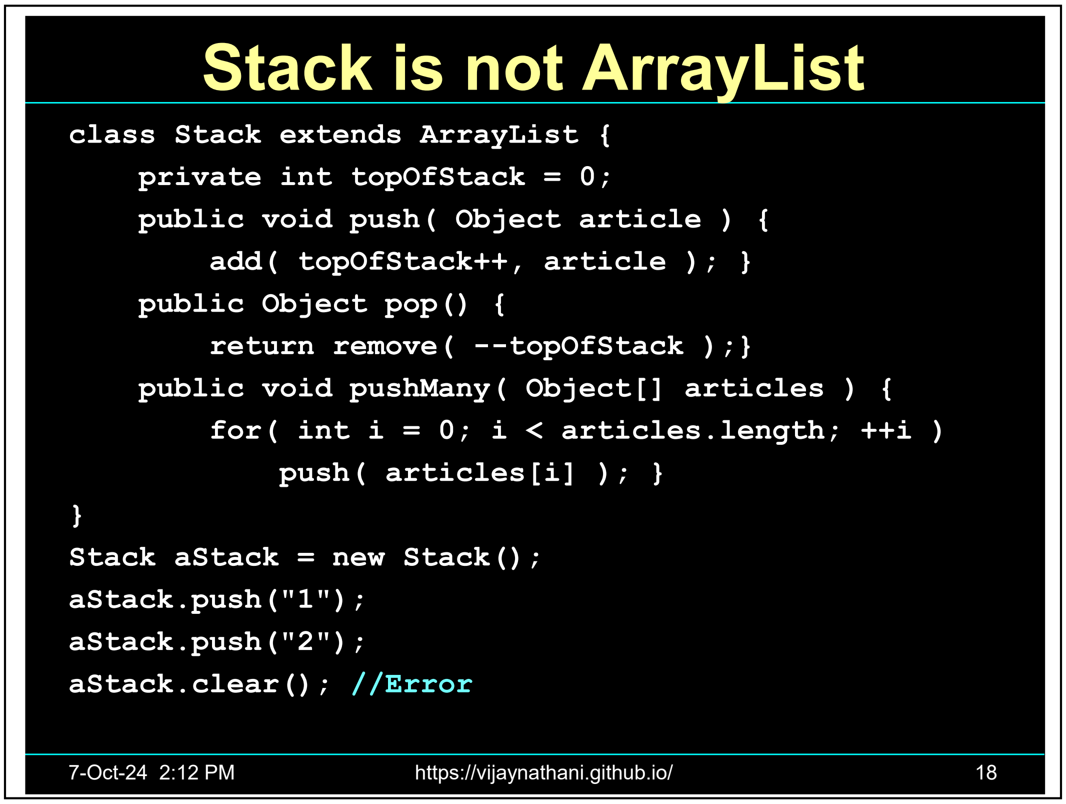


Same Person – Multiple roles



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
```

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 18

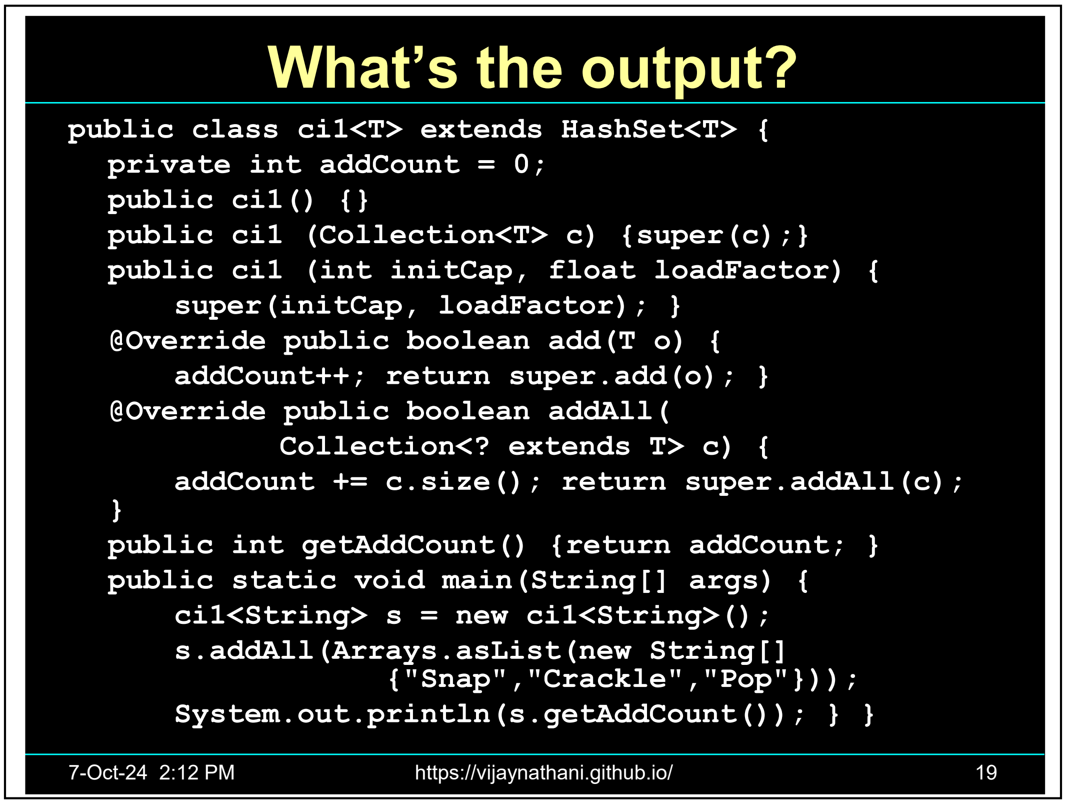
Even a class as simple as this one has problems. Consider what happens when a user leverages inheritance and uses the ArrayList 's clear() method to pop everything off the stack.

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();
    }
}
```



What's the output?

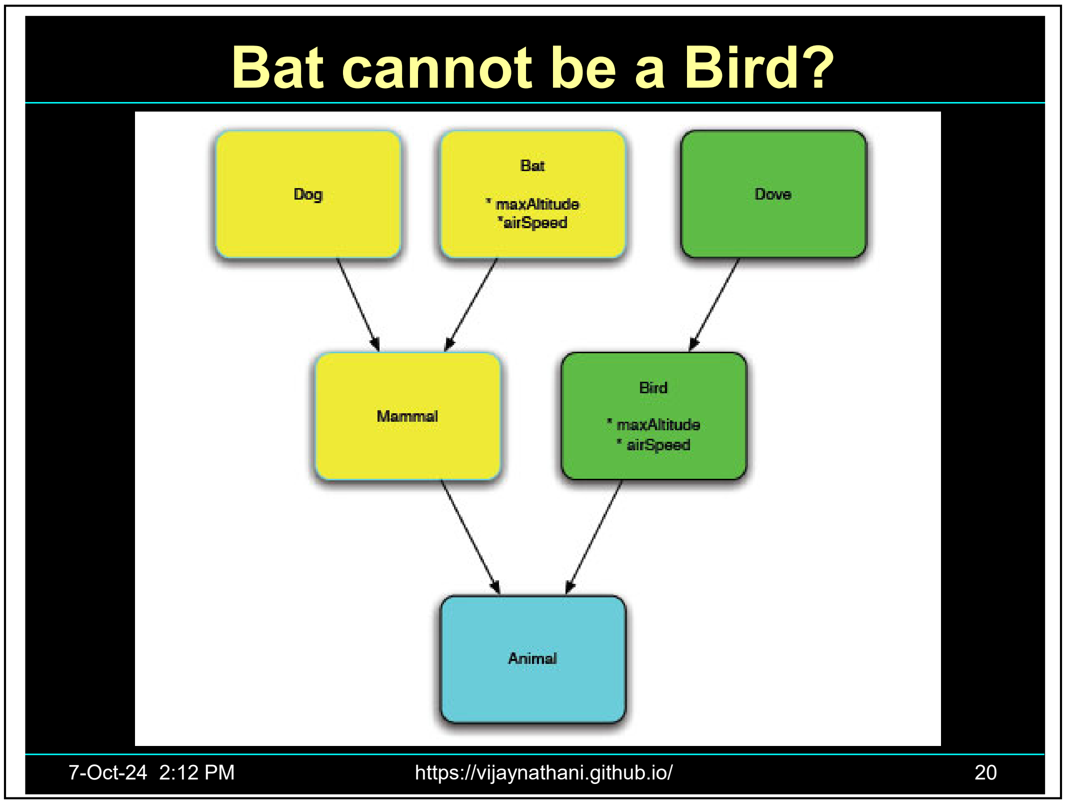
```
public class ci1<T> extends HashSet<T> {
  private int addCount = 0;
  public ci1() {}
  public ci1 (Collection<T> c) {super(c);}
  public ci1 (int initCap, float loadFactor) {
      super(initCap, loadFactor); }
  @Override public boolean add(T o) {
      addCount++; return super.add(o); }
  @Override 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()); } }
```

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 19

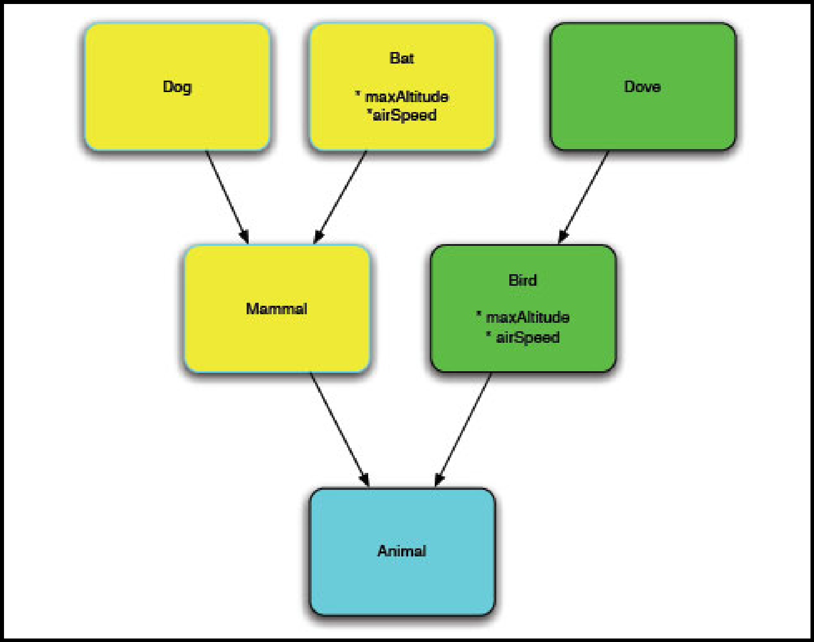
The output comes as 6 instead of 3.

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.



Bat cannot be a Bird?



7-Oct-24 2:12 PM https://vijaynathani.github.io/ 20

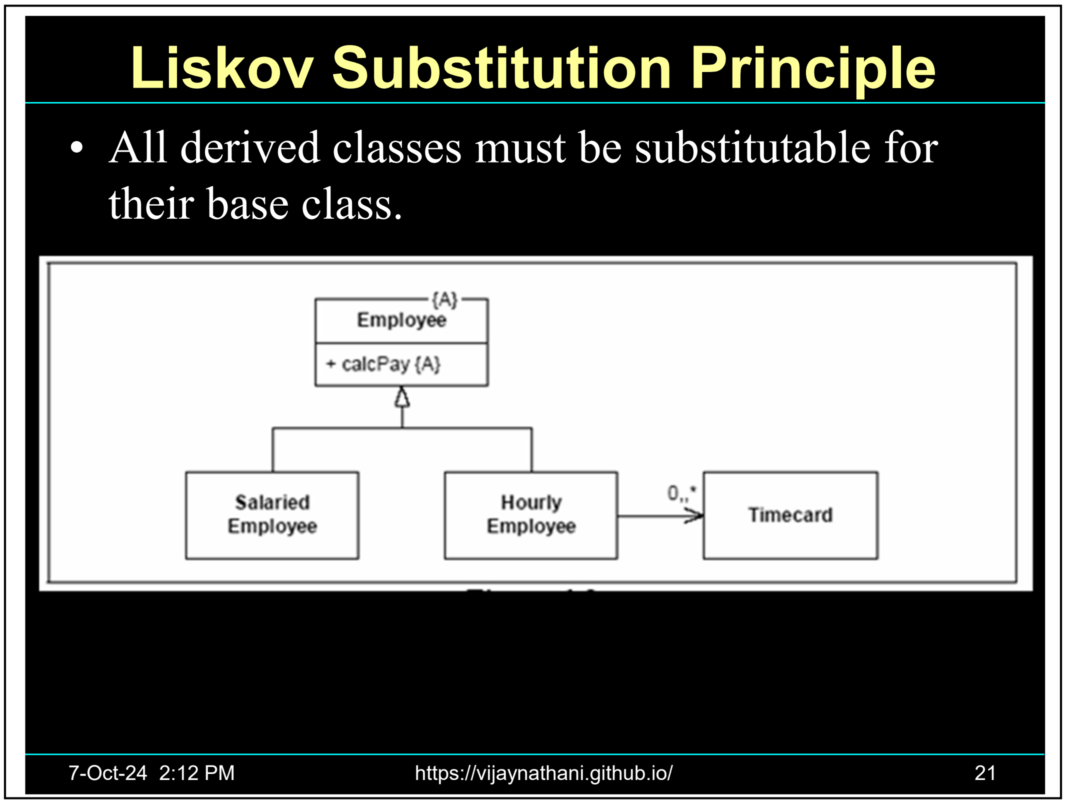
Bat has a flying capability

Dove has a flying capability

is better than

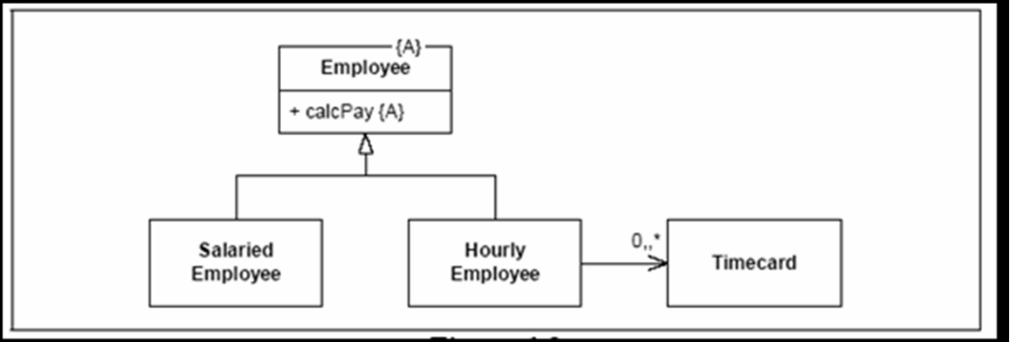
Bat is a flying creature

Dove is a flying creature



Liskov Substitution Principle

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



7-Oct-24 2:12 PM https://vijaynathani.github.io/ 21

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.

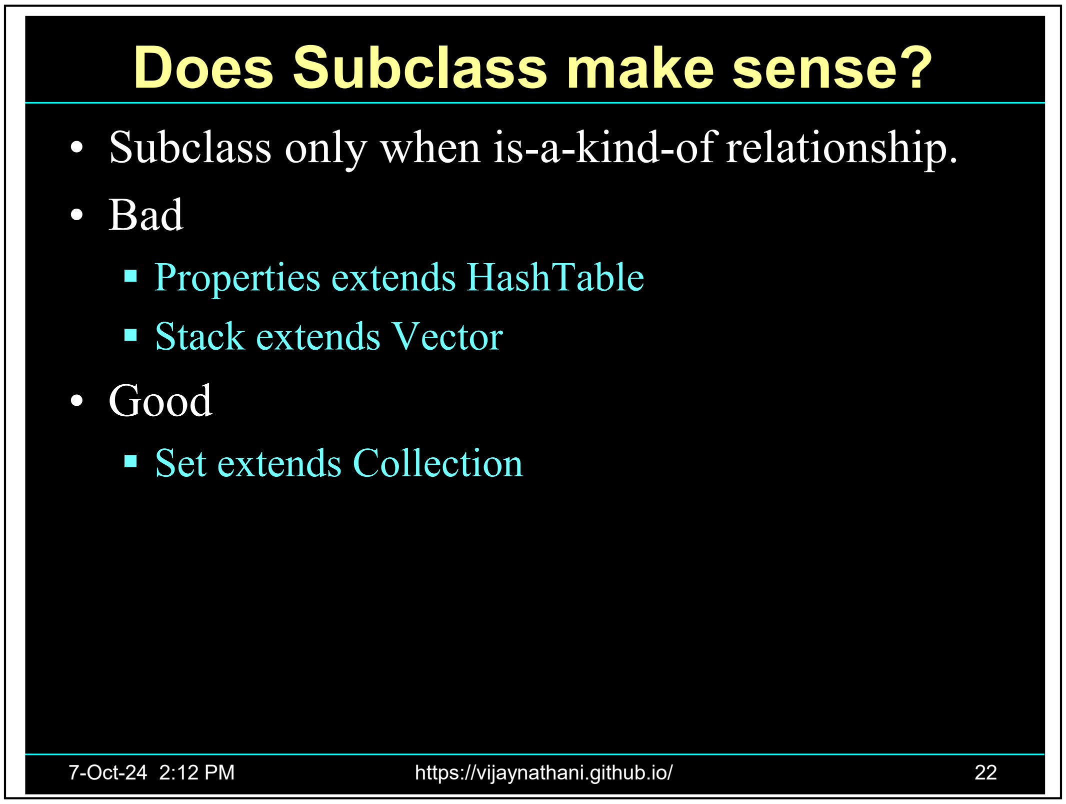
The Liskov Substitution Principle (LSP) makes it clear that the ISA relationship is all about behavior.

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.



Does Subclass make sense?

- Subclass only when is-a-kind-of relationship.
- Bad
  - Properties extends HashTable
  - Stack extends Vector
- Good
  - Set extends Collection

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 22

==

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.

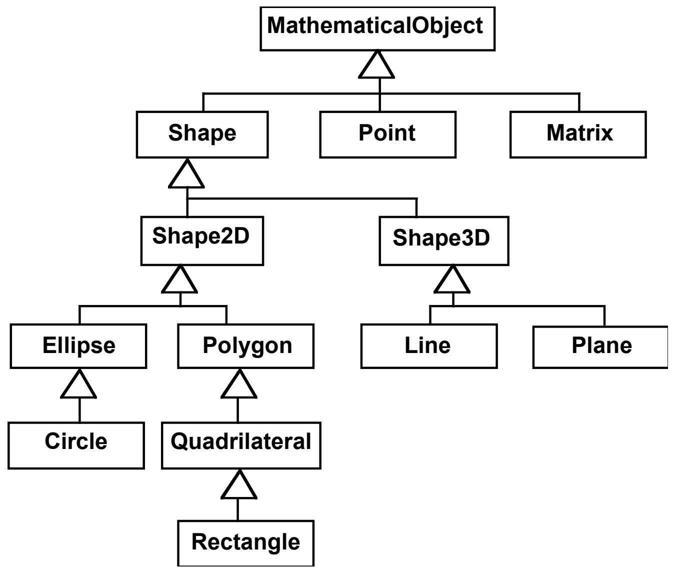
Bad

Many concrete classes in Java are not final

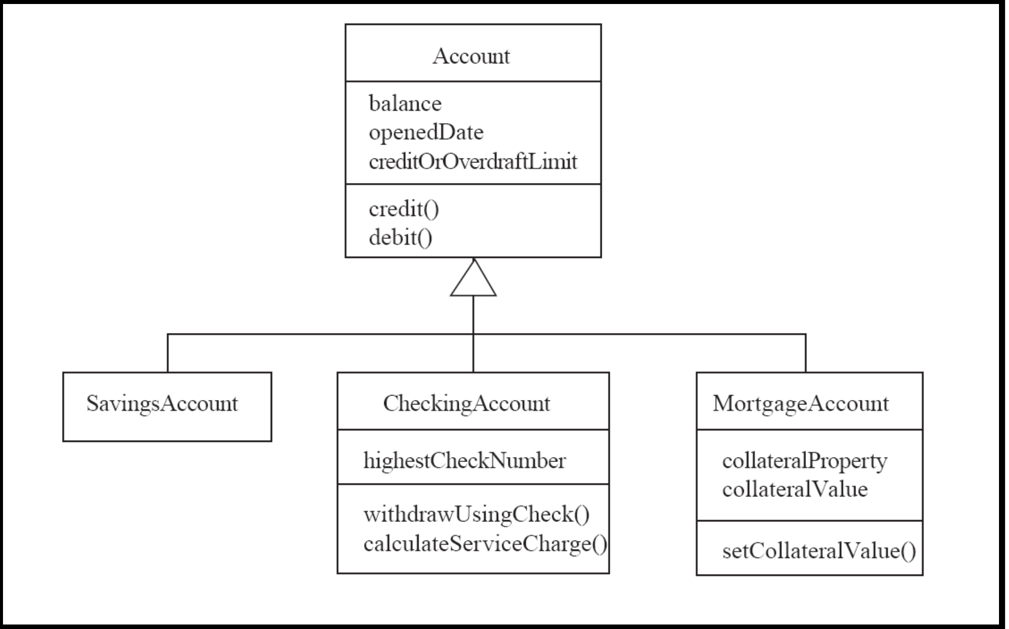
Good

AbstractSet, AbstractMap

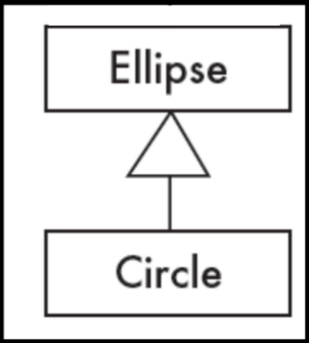
Avoid Deep Inheritance

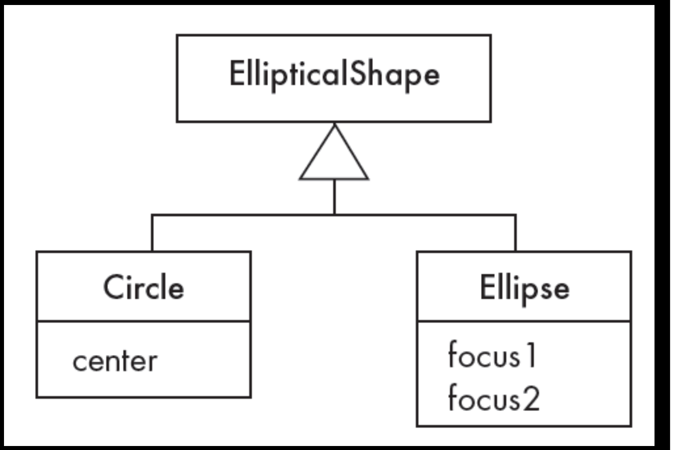


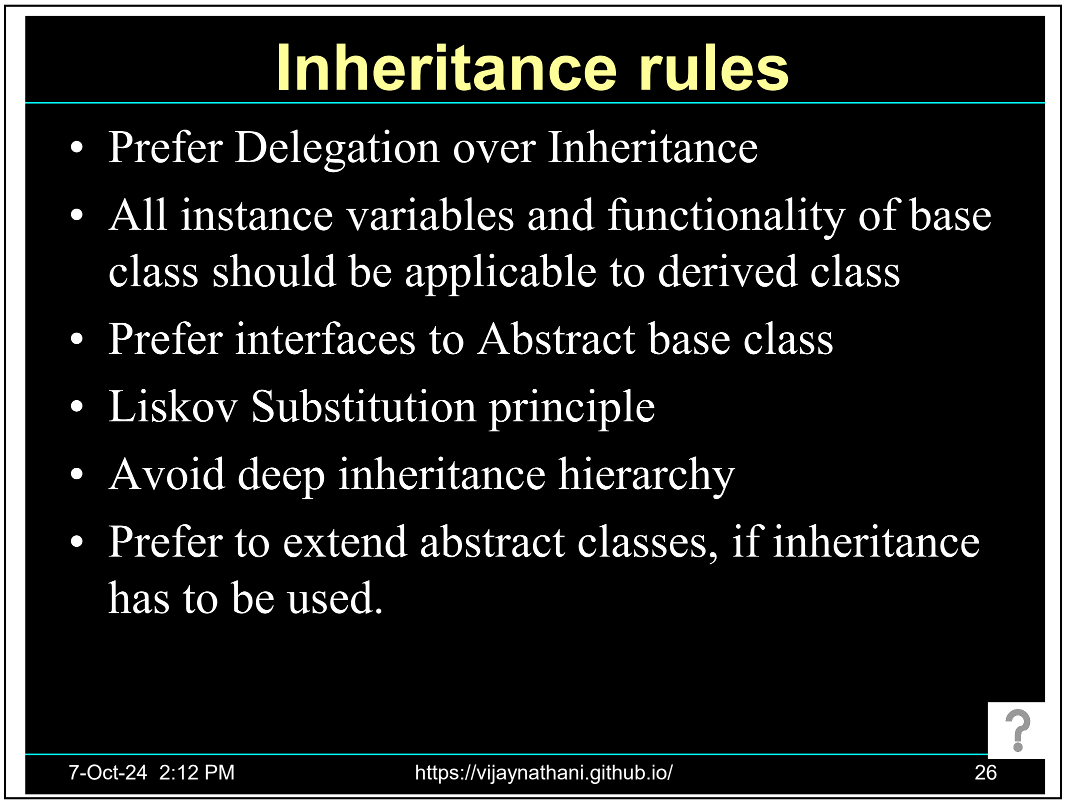
All Inherited Features should make sense in Subclasses



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.

7-Oct-24 2:12 PM https://vijaynathani.github.io/ 26

Q72Inheri: Employee, LinkList

Q31 - UserAccount

Q71inheri – CourseCatalog Q52 – NormalPayment Q54 – Account

Q74 – BitmapButton Q75 – PropertyFileWriter

Avoid deep Inheritance Hierarchy: Q76