

OOAD

"There is no silver bullet."
—Fred P. Brooks, Jr.

It's different



27-Jul-21 12:17 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

27-Jul-21 12:17 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.

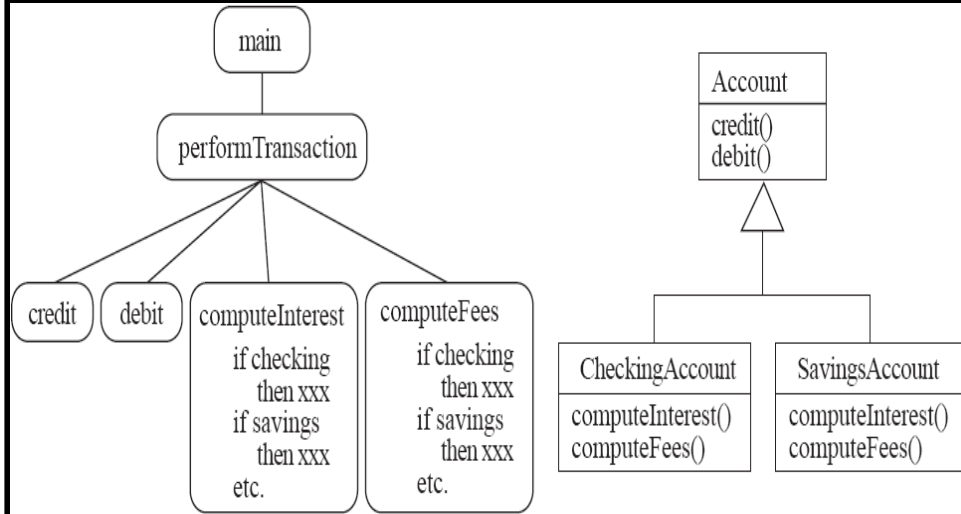
Time to delivery is reduced by at least 50% and usually 70%.

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.

27-Jul-21 12:17 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



27-Jul-21 12:17 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



27-Jul-21 12:17 PM



<https://vijaynathani.github.io/>

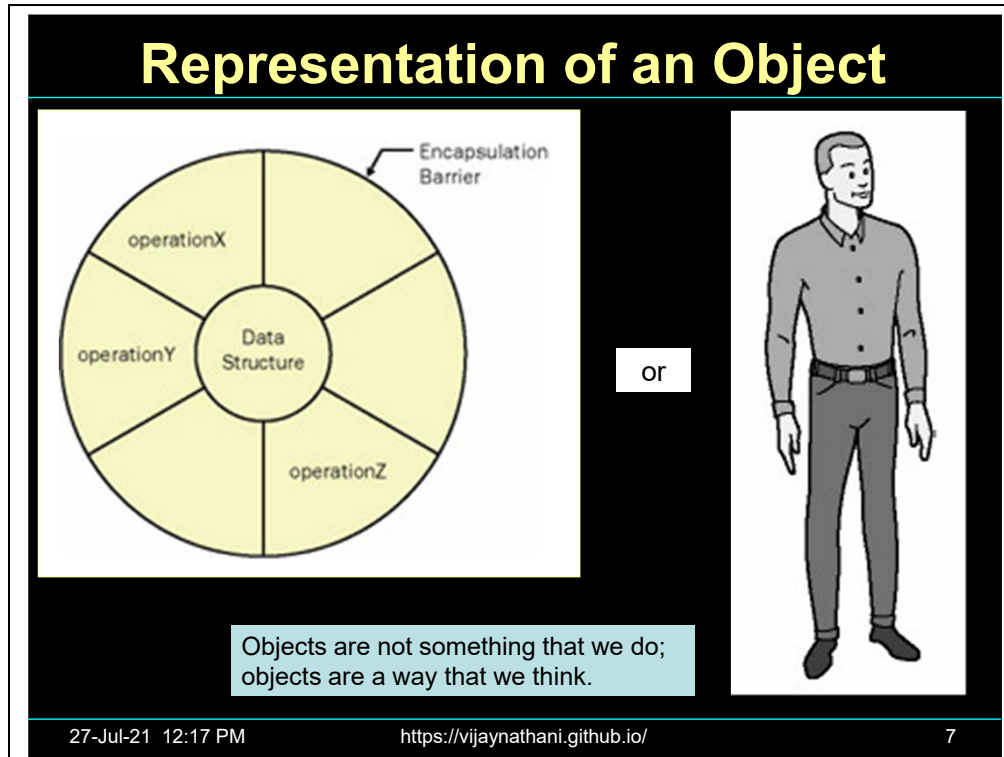
6

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.



In a data-driven approach, the attributes of an object are discovered first and then responsibilities are meted out as a function of which object holds which data.

A behavioral approach mandates the assignment of responsibilities first. Only when you are satisfied with the distribution of responsibilities among your objects are you ready to make a decision about what they need to know to fulfill those responsibilities and which parts of that knowledge they need to keep as part of their structure—in instance variables or attributes.

This is the single biggest difference in definition between data-driven and behavior-driven or responsibility-driven approaches to objects.

Responsibilities are not functions, although there is a superficial resemblance. The former reflects expectations in the domain—the problem space—while the latter reflects an implementation detail in the solution space—the computer program.

Our goal during discovery is object definition, not object specification. Definition means we want to capture how the object is defined by those using it, what they think of that object, what they expect of it when they use it, and the extent to which it is similar to and different from other objects in the domain. Specification will come later (maybe 30 seconds later if you are doing XP and working on a single object), when we allow ourselves to consider how this object might be implemented (simulated) in software.

It's a major mistake, from an object thinking perspective, to define objects in terms of an application instead of a domain.

Decide Classes

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



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

8

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.

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

9

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

How to find classes?

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



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

10

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

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

11

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



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

12

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/Component/Library should be as shy as possible



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

13

It is about managing complexity

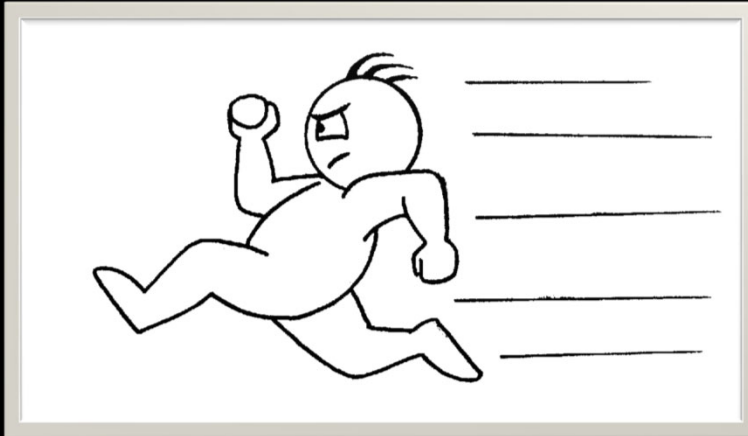
Put state information in the class that works on it. Collaborating classes are unaware of the internal state of this object.

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



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

14

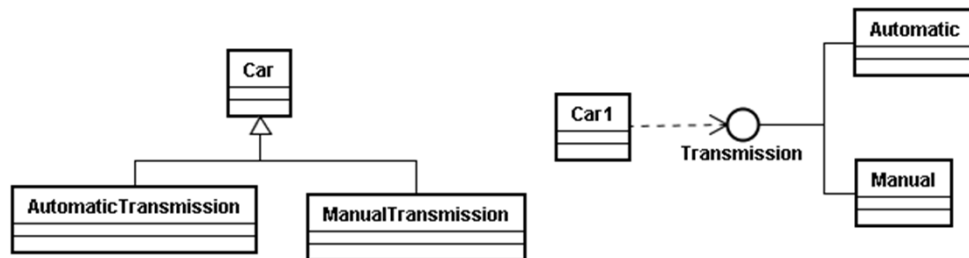
Q30 – Shape; Q40Smell: Courses weekly, range, list; Q43smell: home address / work address

Q36 – USD, RMB;

Q41 – Session (Part of Composite Design Pattern)

Inheritance vs. Delegation

- Which is better?



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

15

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

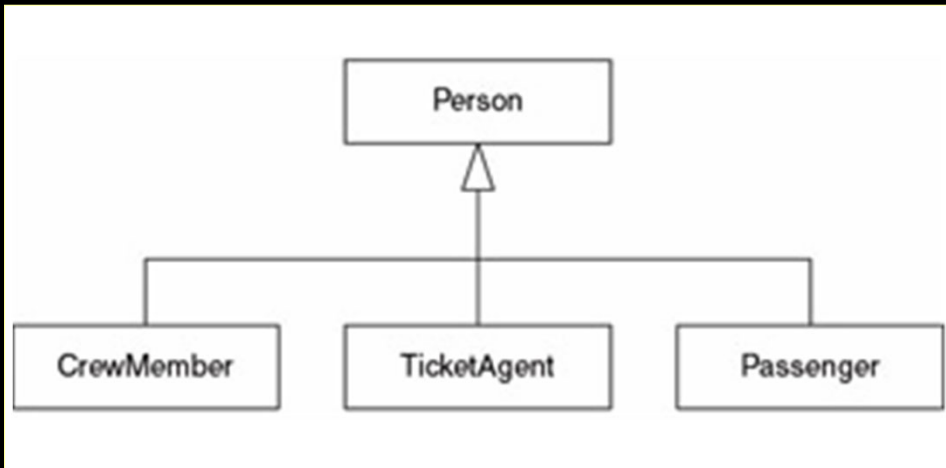
Breaks encapsulation, since it exposes a subclass to implementation details of its super class

"White-box" reuse, since internal details of super classes are often visible to subclasses

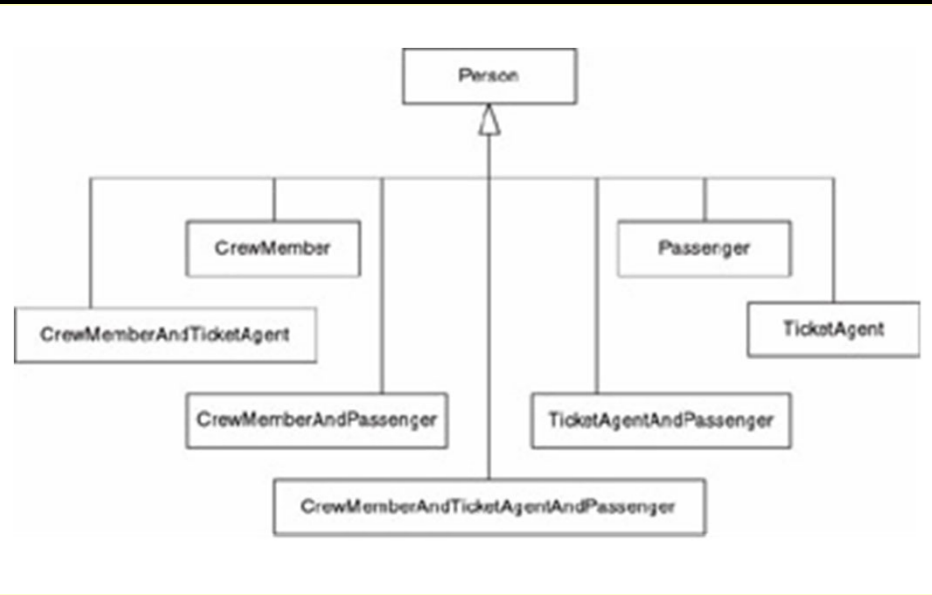
Subclasses may have to be changed if the implementation of the super class changes

Implementations inherited from super classes can not be changed at runtime

Airline Reservation System



Same Person – Multiple roles

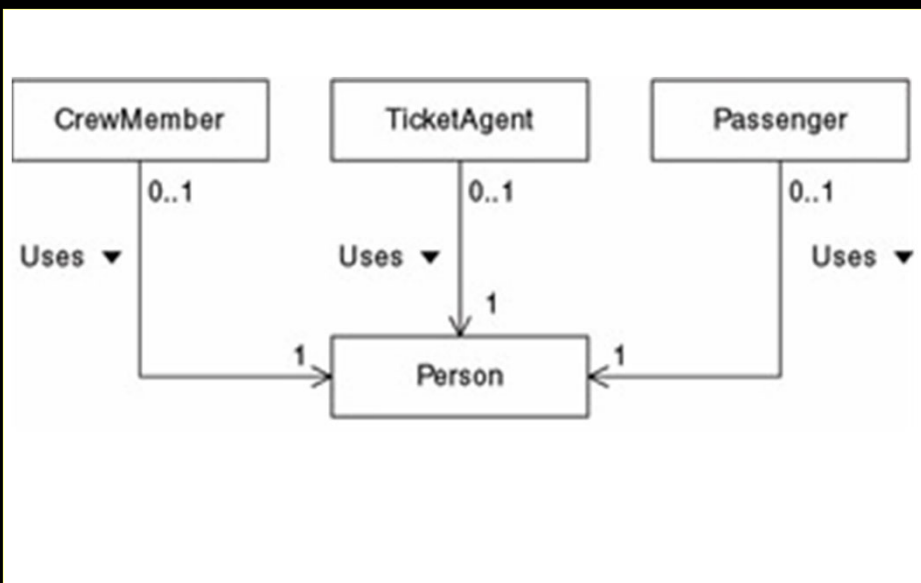


27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

17

A person can change roles Now



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

18

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
```

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

19

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 cil<T> extends HashSet<T> {
    private int addCount = 0;
    public cil() {}
    public cil (Collection<T> c) {super(c);}
    public cil (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) {
        cil<String> s = new cil<String>();
        s.addAll(Arrays.asList(new String[]
            {"Snap", "Crackle", "Pop"}));
        System.out.println(s.getAddCount()); } }
```

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

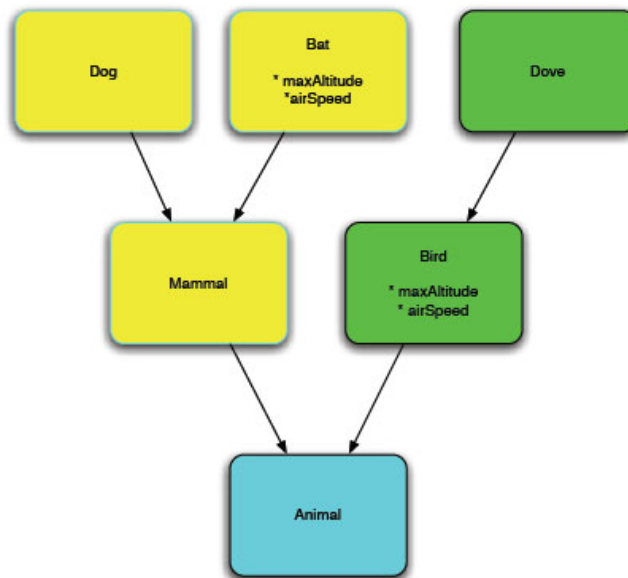
20

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?



27-Jul-21 12:17 PM

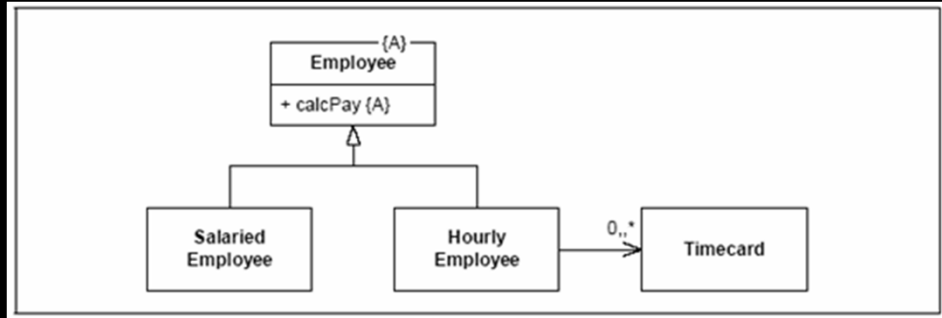
<https://vijaynathani.github.io/>

21

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.



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

22

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

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

23

=====

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:

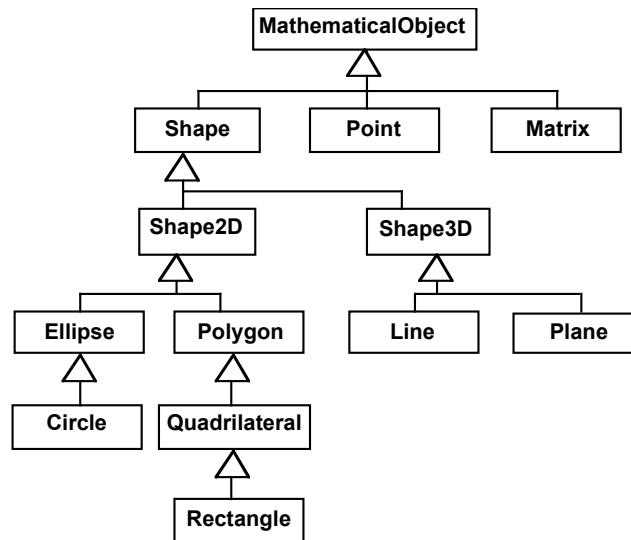
All concrete classes are final

Never override a concrete function.

Bad

Many concrete classes in Java are not final
Good
AbstractSet, AbstractMap

Avoid Deep Inheritance

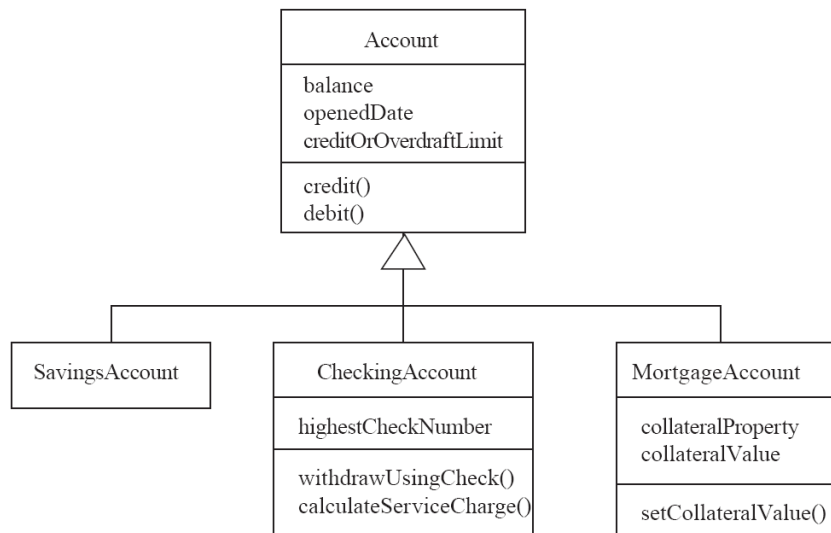


27-Jul-21 12:17 PM

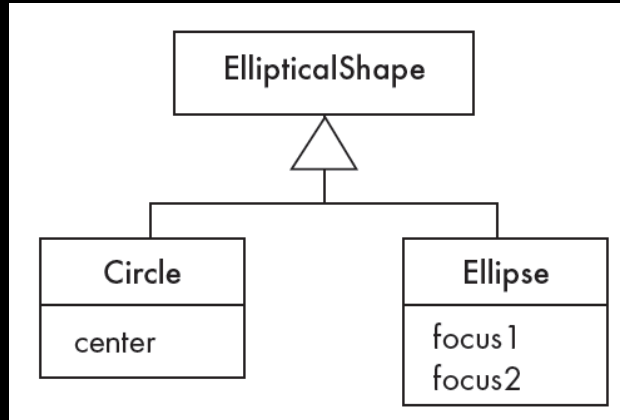
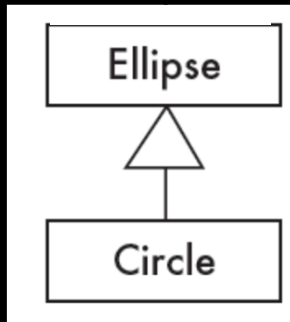
<https://vijaynathani.github.io/>

24

All Inherited Features should make sense in Subclasses



Reorganize



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.



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

27

Q72Inheri: Employee, LinkedList

Q31 - UserAccount

Q71inheri – CourseCatalog Q52 – NormalPayment

Q54 – Account

Q74 – BitmapButton Q75 – PropertyFileWriter

Avoid deep Inheritance Hierarchy: Q76

Guidelines

- DRY



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

28

Q021: BookRentals; Q11: BookRental; Q33 – SurveyData; Q37 – FOC, TT

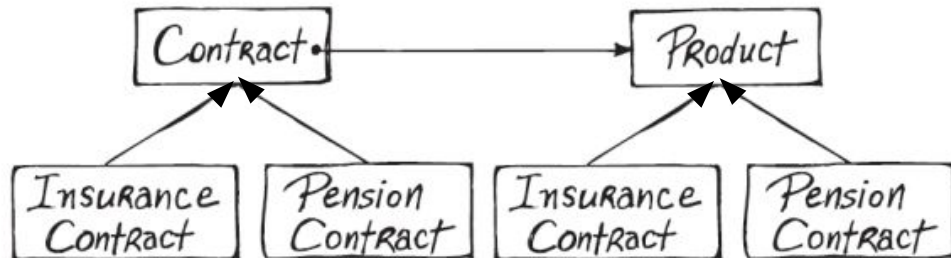
Q10 – JButton – Only for Java.

Q34 – replace; Q35 – BookRental

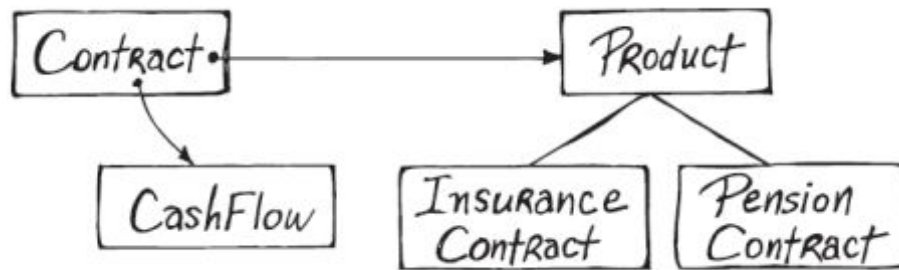
Q70 – Array Scan, Java only.

There are three numbers in software: 0, 1, and infinity. 0 represents the things we do not do in a system (we do those for free). 1 represents the things we do once and only once. But at the moment we do something twice, we should treat it as infinitely many and create cohesive services that allow it to be reused.

Improve



Eliminate Parallel Hierarchies



Reduce Coupling - Problem

```
Server s = new Server();
```

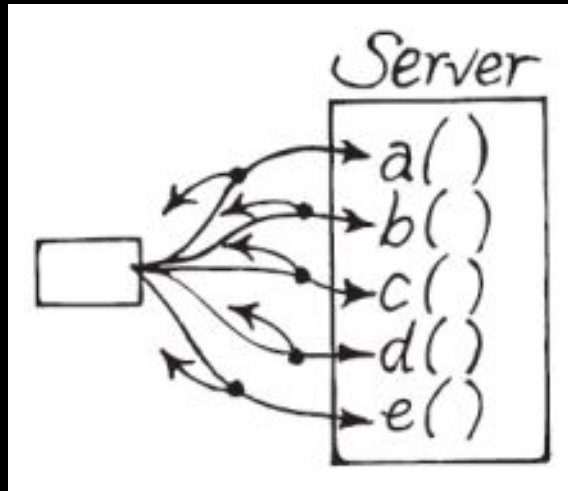
```
s.a(this);
```

```
s.b(this);
```

```
s.c(this);
```

```
s.d(this);
```

```
s.e(this);
```



Reduce Coupling - Solution

```
Server s = new Server(this);
```

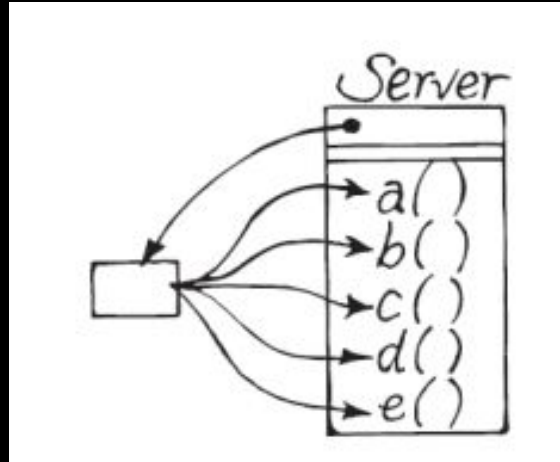
```
s.a();
```

```
s.b();
```

```
s.c();
```

```
s.d();
```

```
s.e();
```



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

32

Better still: use Observer design pattern

Classes in an Application

- Many simple classes means that each class
 - encapsulates less of overall system intelligence
 - is more reusable
 - is easier to implement
- A few complex classes means that each class
 - encapsulates a large portion of system intelligence
 - is less likely to be reusable
 - is more difficult to implement

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

33

Lots of little pieces

Classes are cohesive

Methods do only one thing.

Guidelines

- A class should have less than 50 lines
- Most functions should be less than or equal to 5 lines.
 - A function taking more than 3 arguments should be rare and justified specially.

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

34

Note: On a Home PC - 1 Million function calls take 8 milliseconds and 1 Million objects are created in 23 milliseconds

Some Real Examples

Tool	Files	Lines/file (avg)	LOC/file (avg)
JUnit	88	71	39
Hibernate	1063	90	72
Eclipse	14,620	153	106
DomainObjects for .NET	422	164	98
Compiere ERP &CRM	1191	163	114
Hsqldb	290	503	198

?

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

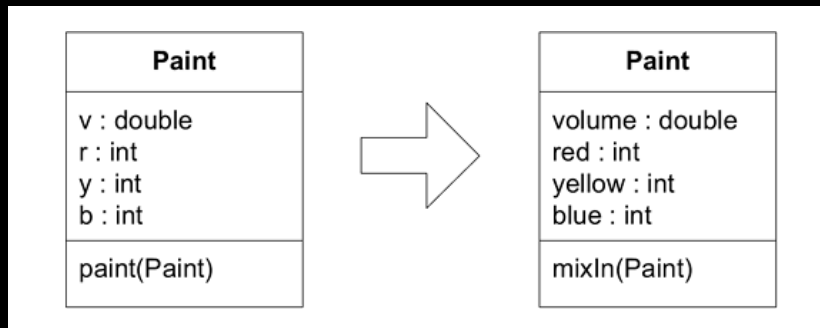
35

Q56srp – Restaurants

Q59srp - Customers

Rule: Don't abbreviate

- Code should be self documenting.
 - Tools: SonarQube, Codepro.



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

36

Public API's have to be documented i.e. every class, function, interface, exceptions.
Mutable objects that can / cannot be modified.

This is possible by choosing the right variable and function names.

Comments are secondary because they tend to lie

Java: Checkstyle, PMD

C#: SytleCop+, FxCop, Simian, Ncover, NDepend for cyclomatic complexity.

C++: Simian or PMD CPD for duplication, coverity for source code analysis

JavaScript: JSHint


```
public List<int[]> getThem() {  
    List<int[]> list1 =  
        new ArrayList<int[]>>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}  
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells =  
        new ArrayList<Cell>();  
    for (Cell cell : gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```



27-Jun-21 12:17 PM

<https://www.dynamichighlights.com>

37

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = "102";  
    /* ... */  
};
```



```
class Customer {  
    private Date generationTimestamp;  
    private Date modificationTimestamp;;  
    private static final String  
        RECORD_ID = "102";  
    /* ... */  
};
```

Guidelines

- Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser.
 - Avoid words like Manager, Processor, Data, or Info in the name of a class.
 - A class name should not be a verb.
- Methods should have verb or verb phrase names like postPayment, deletePage, or save.

Code should be readable

- Any fool can write code that a computer can understand. Good programmers write code that humans can understand. – Martin Fowler

```
▪ Calendar c=Calendar.getInstance();  
  c.set(2005,Calendar.NOVEMBER, 20);  
  Date t = c.getTime(); OR  
▪ Date t = november(20, 2005);  
  public Date november(  
      int day, int year) { ... }
```

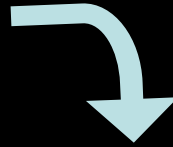
Compare

```
void process() {  
    input();  
    count++;  
    output();  
}
```

```
void process() {  
    input();  
    tally();  
    output();  
}  
private void tally() {  
    count++;  
}
```

Compose Method Pattern

```
public void add(Object element) {  
    if (!readOnly) {  
        int newSize = size + 1;  
        if (newSize > elements.length) {  
            Object[] newElements =  
                new Object[elements.length + 10];  
            for (int i = 0; i < size; i++)  
                newElements[i] = elements[i];  
            elements = newElements;  
        }  
        elements[size++] = element;  
    }  
}
```



```
public void add(Object element) {  
    if (readOnly)  
        return;  
    if (atCapacity())  
        grow();  
    addElement(element);  
}
```

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

42

Benefits and Liabilities

- +Efficiently communicates what a method does and how it does what it does.
- +Simplifies a method by breaking it up into well-named chunks of behavior at the same level of detail.
- Can lead to an overabundance of small methods.
- Can make debugging difficult because logic is spread out across many small methods.

Improve

```
flags |= LOADED_BIT;
```

- Solution: Extract to a message

```
void setLoadedFlag() {  
    flags |= LOADED_BIT;  
}
```

Improve Code

```
// Check to see if the employee
// is eligible for full benefits
if ((employee.flags & HOURLY_FLAG)
    && (employee.age > 65)) ...

if (employee.
    isEligibleForFullBenefits())
```


Improve

```
class Board {  
    ...  
    String board() {  
        StringBuffer buf = new StringBuffer();  
        for(int i = 0; i < 10; i++) {  
            for(int j = 0; j < 10; j++)  
                buf.append(data[i][j]);  
            buf.append("\n" );  
        }  
        return buf.toString();  
    }  
}
```

Only one level of indentation per method

```
Class Board {  
    ...  
    String board() {  
        StringBuffer buf = new StringBuffer();  
        collectRows(buf);  
        return buf.toString();  
    }  
    void collectRows(StringBuffer buf) {  
        for(int i = 0; i < 10; i++)  
            collectRow(buf, i);  
    }  
    void collectRow(StringBuffer buf, int row) {  
        for(int i = 0; i < 10; i++)  
            buf.append(data[row][i]);  
        buf.append("\n" );  
    }  
}
```



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

46

Self documenting code

Q20 – inch; Q06 – NO_GROUPING; Q07 – addHoliday; Q21 – full name in English;

Q22 – complexPassword; Q23 – TokenStream; Q25 - orderItems

Good Comments?

```
String text = ""'bold text'";
ParentWidget parent = new BoldWidget(
    new MockWidgetRoot(), ""'bold text'");
AtomicBoolean failFlag = new AtomicBoolean();
failFlag.set(false);
//This is our best attempt to get a race condition
//by creating large number of threads.
for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread = new
        WidgetBuilderThread(widgetBuilder, text,
            parent, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
}
assertEquals(false, failFlag.get());
```

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

47

Comment to

- explain WHY we are doing something?
- Also for external documentation. Javadocs public API
- To give warnings: e.g. Don't run unless you want to kill this program.
- Todo comments

Tell, Don't Ask

- Ask for help, not information



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

48

Never ask an object for information that you need to do something; rather, ask the object that has the information to do the work for you.

In other words: Don't use any getters/setters/properties.

Avoid getters and setters

- Wrong

```
Money a, b, c;  
//...  
a.setValue( a.getValue() +  
            b.getValue() );
```

- Right

```
Money a, b, c;  
//...  
a.increaseBy( b );
```

Improve

```
if (aCargo.getStatus() ==  
    HandlingStatus.MISDIRECTED)  
    ...  
  
if (aCargo.isMisdirected())  
    ...
```

Compare

```
Dog dog = new Dog();  
dog.setBall(  
    new Ball());  
Ball ball =  
    dog.getBall();
```

```
Dog dog = new Dog();  
Dog.setWeight("23Kg");
```

```
Dog dog = new Dog();  
dog.take(new Ball());  
Ball ball =  
    dog.give();
```

```
Dog dog =  
    new Dog("23Kg");
```

Example

Wrong:

```
MyThing[] things =  
    thingManager.getThingList();  
for (int i = 0; i < things.length; i++) {  
    MyThing thing = things[i];  
    if (thing.getName().equals(thingName))  
        return thingManager.delete(thing);  
}
```

Right:

```
return thingManager.deleteThingNamed  
    (thingName);
```


Open Closed Principle

- Software entities (Classes, modules, functions) should be open for extension but closed for modifications.

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

53

It should be possible to change the environment of a class without changing the class.

A class should be closed for modification, but open for extension. When you change a class, there is always a risk that you will break something. But if instead of modifying the class you extend it with a sub-class, that's a less risky change.

Guidelines

- Keep things that vary separately from things that are common.



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

54

Q45 – LoanHandler

Q44 – FILE1, DATABASE1

Q39 - Scheduler

Q83 – Cooker

Q84 – ChooseFontDialog

Dependency Inversion Principle

- Program to an interface and not to an implementation
 - Any problem?



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

55

“HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES, BOTH SHOULD DEPEND UPON ABSTRACTIONS.

ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.”

No variable should hold a reference to a concrete class.

No class should derive from a concrete class.

No method should override an implemented method of any of its base classes.

It is OK to depend on stable classes like String, Integer, JPanel, etc.

Avoids designs that are rigid, Fragile and Immobile.

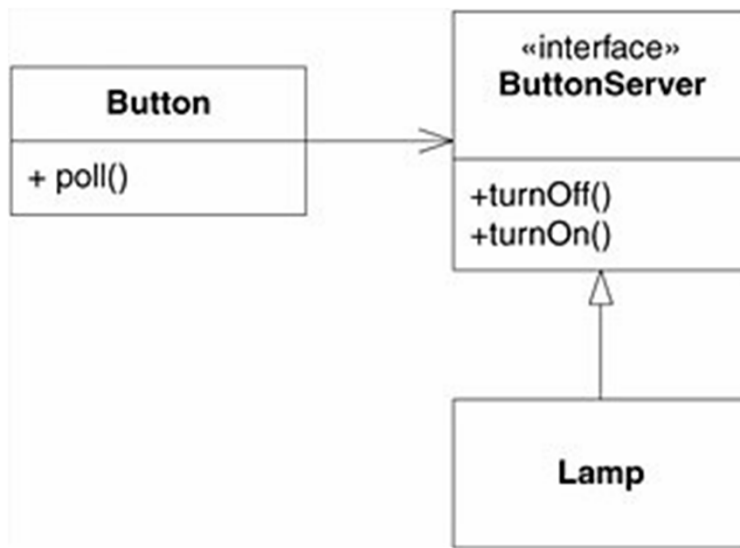
Example: We have a classes: Copy, Keyboard, Printer. The Copy reads from keyboard and prints to the printer. If we use DIP, we have an abstraction for Keyboard and Printer. So we can add input and output devices later on.

Example: Collections in Java.

Example: Customer is a class. Employee is a class. Employees are now allowed to purchase on credit. We need an interface Buyer.

For every variable use the maximum abstract type possible.

DIP implemented



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

56

Advantages

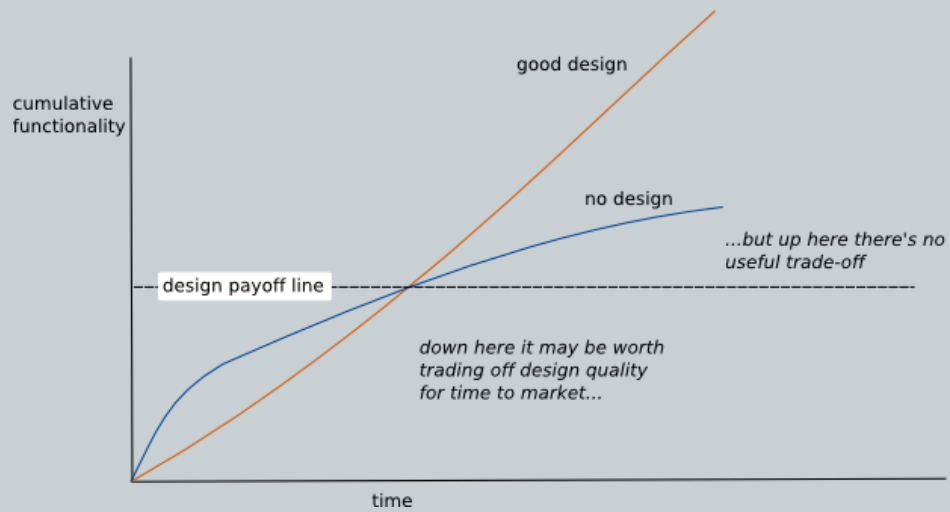
- Clients are unaware of the specific class of the object they are using
- One object can be easily replaced by another
- Object connections need not be hardwired to an object of a specific class, thereby increasing flexibility
- Loosens coupling
- Increases likelihood of reuse
- Improves opportunities for composition since contained objects can be of any class that implements a specific interface

Disadvantages

- Modest increase in design complexity

Q96, Q94, Q93, Q91 - DIP

Is Good Design worth it?



27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

57

Summary

- Keep it DRY and shy
- Prefer composition over inheritance
- Self-documenting code
- Small Classes / Functions

27-Jul-21 12:17 PM

<https://vijaynathani.github.io/>

58

No Golden Bullet.