SOFTWARE ENGINEERING
(Week-15)

USAMA MUSHARAF

MS-CS (Software Engineering)

LECTURER (Department of Computer Science)

FAST-NUCES PESHAWAR

AGENDA OF WEEK # 15

- Software Design Patterns (Cont…)
- Software Testing
  - Black Box
  - White Box

FACTORY DESIGN PATTERN
(CREATIONAL PATTERN)

FACTORY DESIGN PATTERN

Factory Pattern defines an interface for creating the object but let the subclass decide which class to instantiate. Factory pattern let the class defer instantiation to the sub class.

FACTORY DESIGN PATTERN

Problem Statement:

If there exist class hierarchies i-e super / sub classes then client object usually know which class /sub class to instantiate but at times client object know that it needs to instantiate the object but of which class it does not know

it may be due to many factors

- The state of the running application
- Application configuration settings
- Expansion of requirements or enhancements

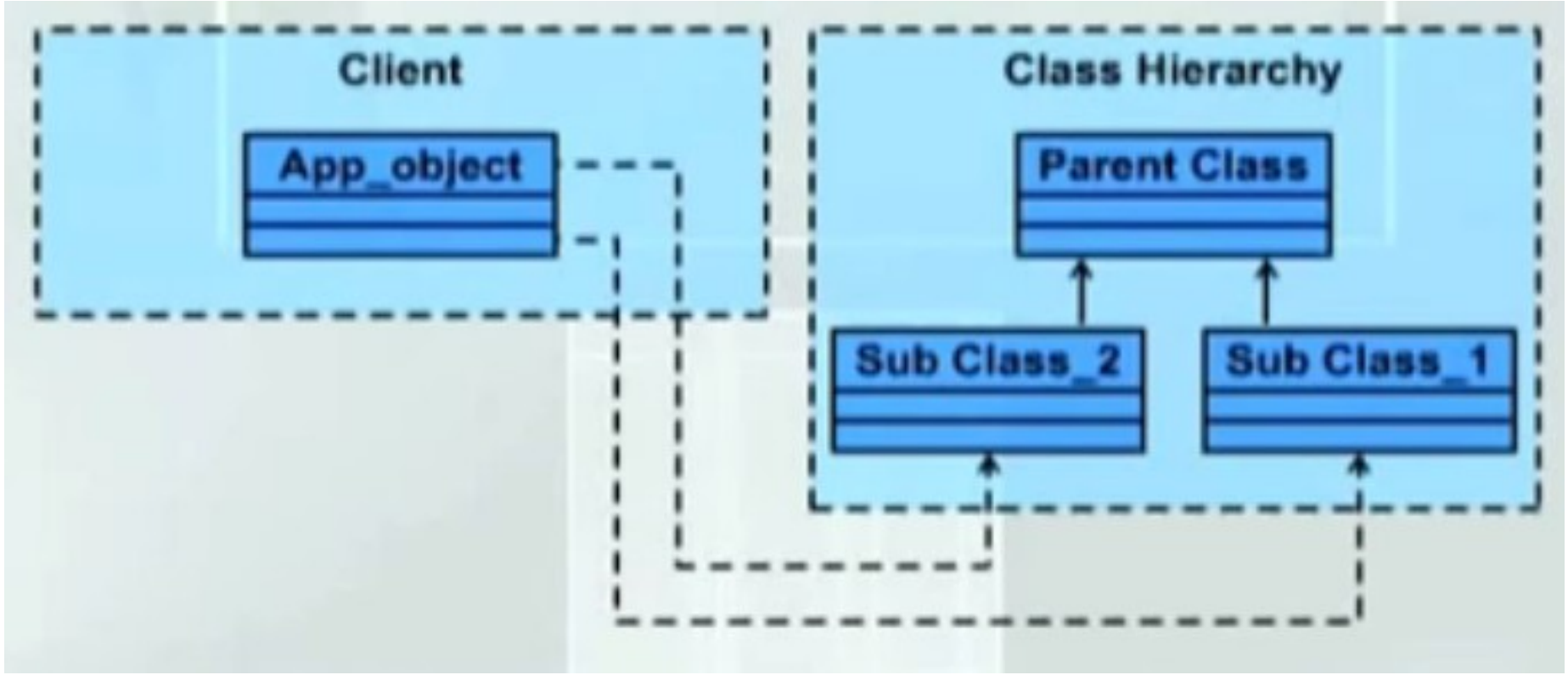
FACTORY DESIGN PATTERN

In such cases, an application object needs to implement the class selection criteria to instantiate an appropriate class from the hierarchy to access its services and that selection criteria will be considered as a part of the client code to access the concrete class from hierarchies of classes.

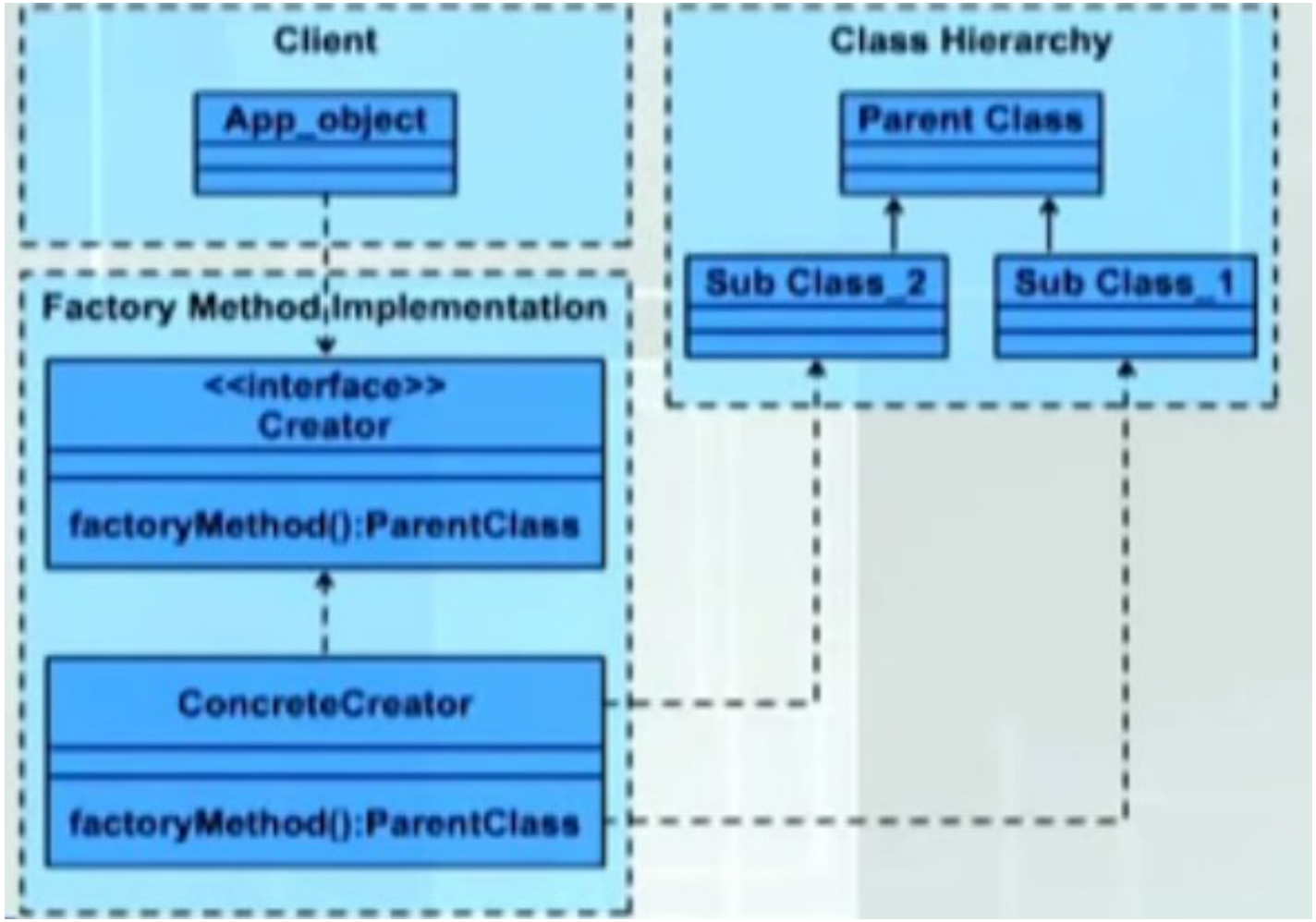
Disadvantage of this approach:

It results in high coupling.

HIGH DEGREE OF COUPLING



PROPOSED SOLUTION



PROPOSED SOLUTION

The solution has the build violation of principle of software design i-e “Loose coupling”; as opposite to the principle above solution is having high degree of coupling between client and classes in hierarchies.

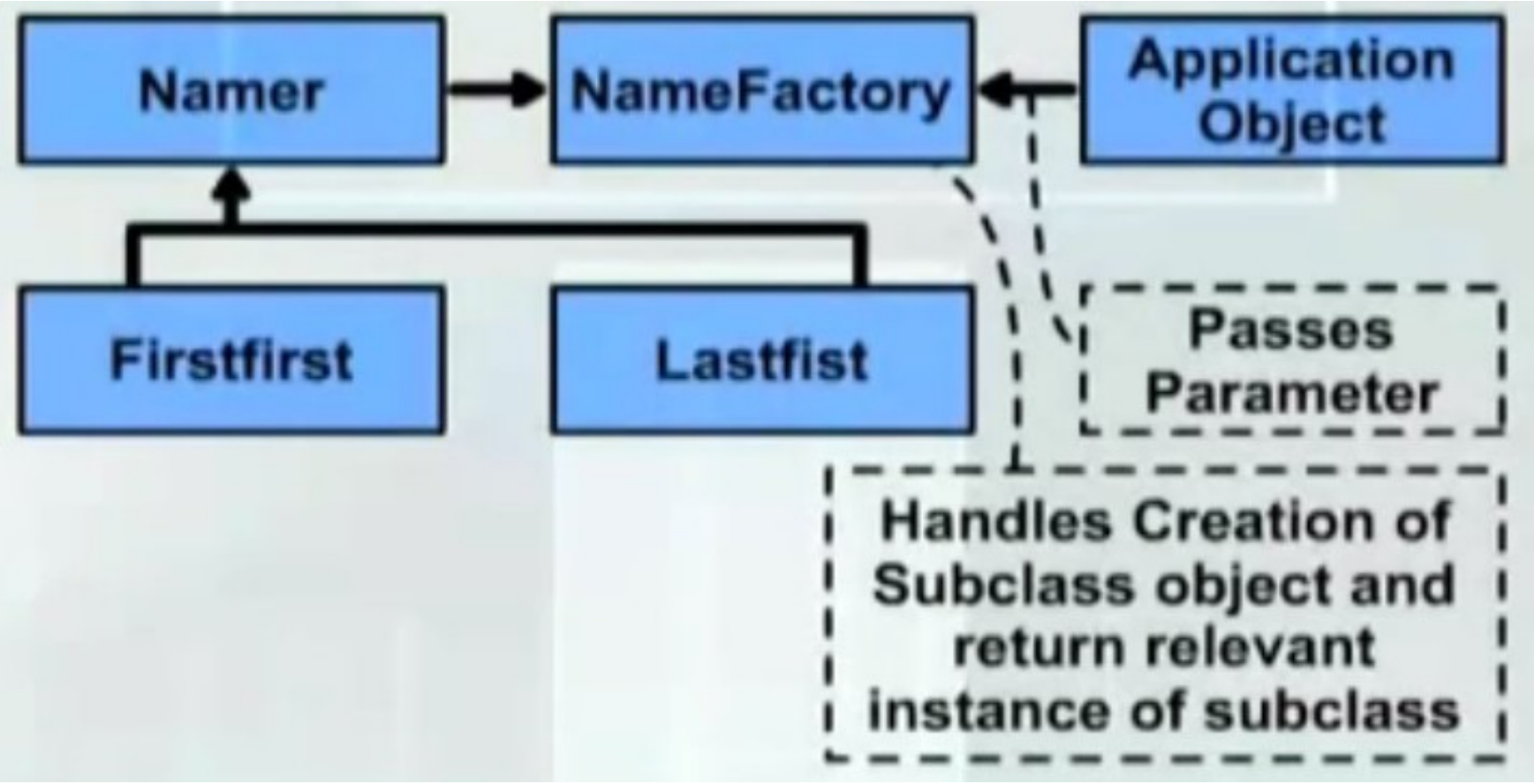
PROBLEM STATEMENT

We want the user to enter the name in either “first name last name or last name, first name” format.

We have made the assumption that there will always be a comma between last name and first name and space between first name last names.

The client does not need to be worried about which class is to access when it is entering the name in either of the format. Independent of the format of the data to be entered, system will display first name and last name.

CLASS DIAGRAM



CODE

```
class Namer
{
protected String last;     //store last name here
protected String first;    //store first name here
public String getFirst()
  {
   return first;      //return first name
  }
public String getLast() {
return last;         //return last name
        }
}
```

CODE

```
class FirstFirst extends Namer
{
public FirstFirst(String s)
{
int i = s.lastIndexOf(" ");    //find space
if (i > 0)
 {
   //left is first name
first = s.substring(0, i).trim();
   //right is last name
last =s.substring(i+1).trim();
}
```

```
else
 {
first = " ";      // put all in last name
last = s;         // if no space
   }
  }
}
```

CODE

```
class LastFirst extends Namer
 { public LastFirst(String s)
{
int i = s.indexOf(",");     //find comma
if (i > 0)
{
//left is last name
last = s.substring(0, i).trim();
//right is first name
first = s.substring(i + 1).trim();
}
```

```
else {
last = s;        // put all in last name
first = “ ";     // if no comma
    }
  }
}
```

CODE

```
class NameFactory
{
//returns an instance of LastFirst or FirstFirst
//depending on whether a comma is found
public Namer getNamer(String entry)
{
int i = entry.indexOf(",");
//comma determines name order
if (i>0)
return new LastFirst(entry);
//return one class
```

```
else
return new FirstFirst(entry);
//or the other
  }
}
```

CODE

```
public class testFactory
 {
   public static void main(String args[])
   {
   NameFactory nfactory = new NameFactory();
   String name="Ali khan";
   Namer namer = nfactory.getNamer(name);       // Delegation
       //compute the first and last names
       //using the returned class
   System.out.println(namer.getFirst());
   System.out.println(namer.getLast());
   }
}
```

STRATEGY DESIGN PATTERN
(BEHAVIORAL PATTERN)

STRATEGY DESIGN PATTERN

Defines a family of algorithms, encapsulates each one, and make them interchangeable,

Strategy lets the algorithm vary independently from clients that use it.

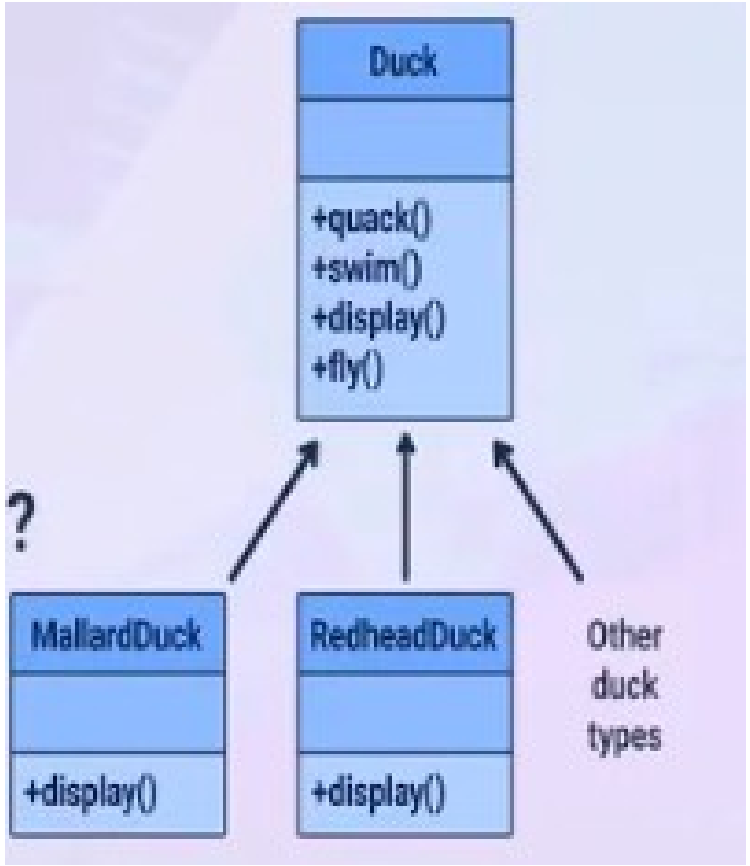
STRATEGY DESIGN PATTERN (EXAMPLE)

STRATEGY DESIGN PATTERN (EXAMPLE)

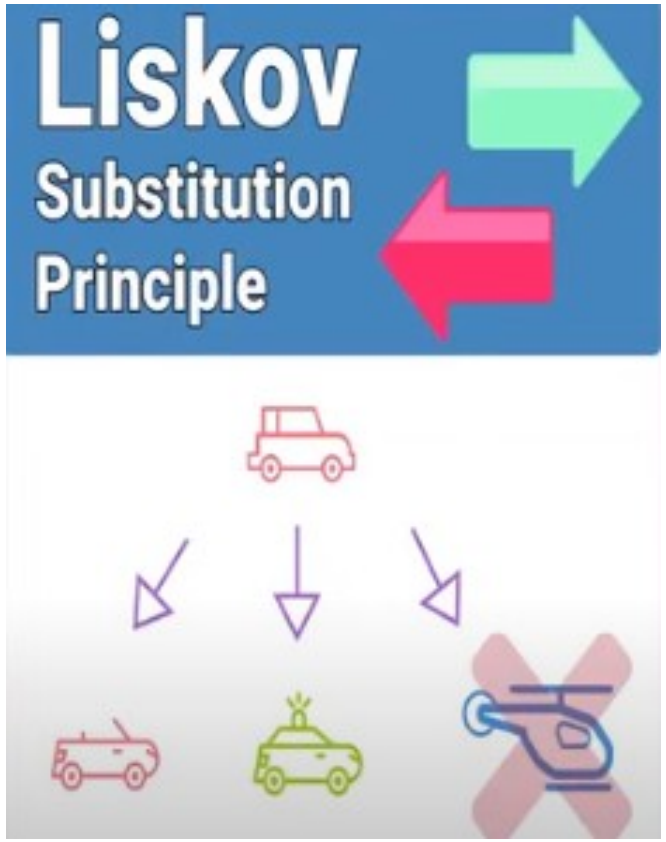
STRATEGY DESIGN PATTERN (EXAMPLE)

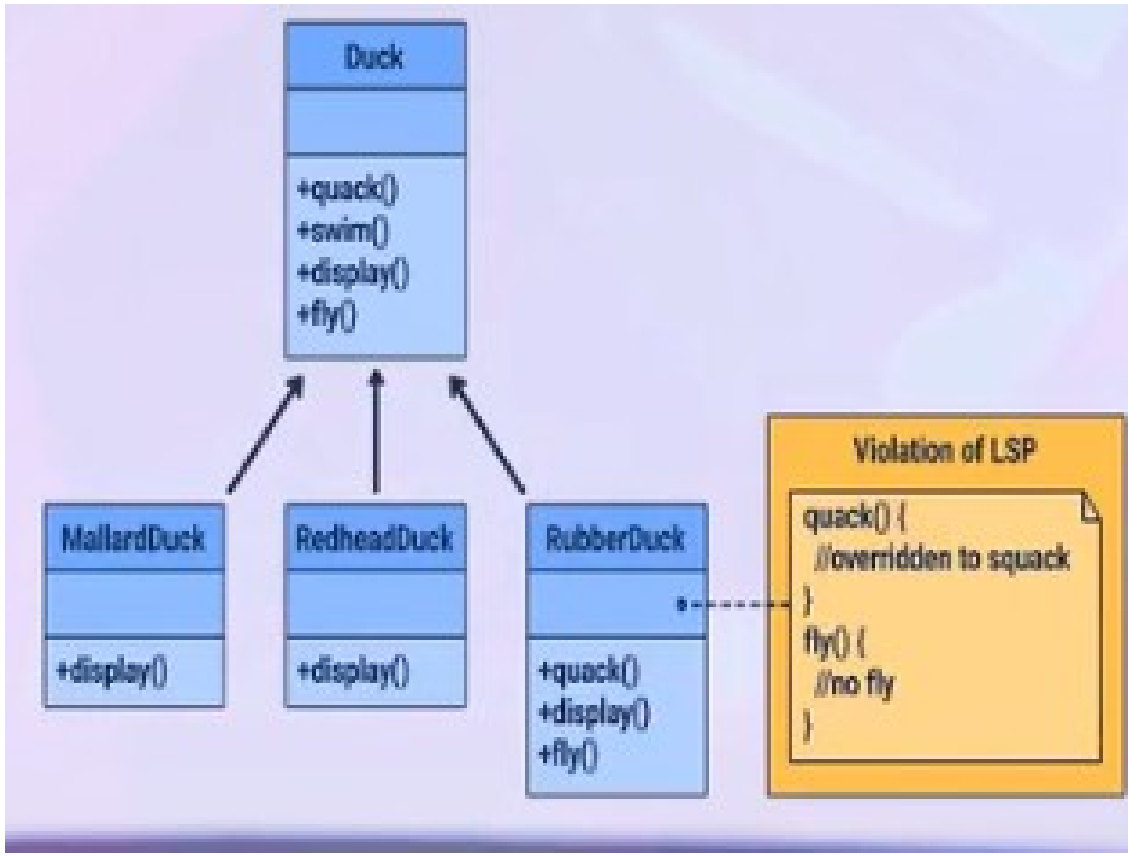
Add fly()

What about rubber duck or wooden duck?



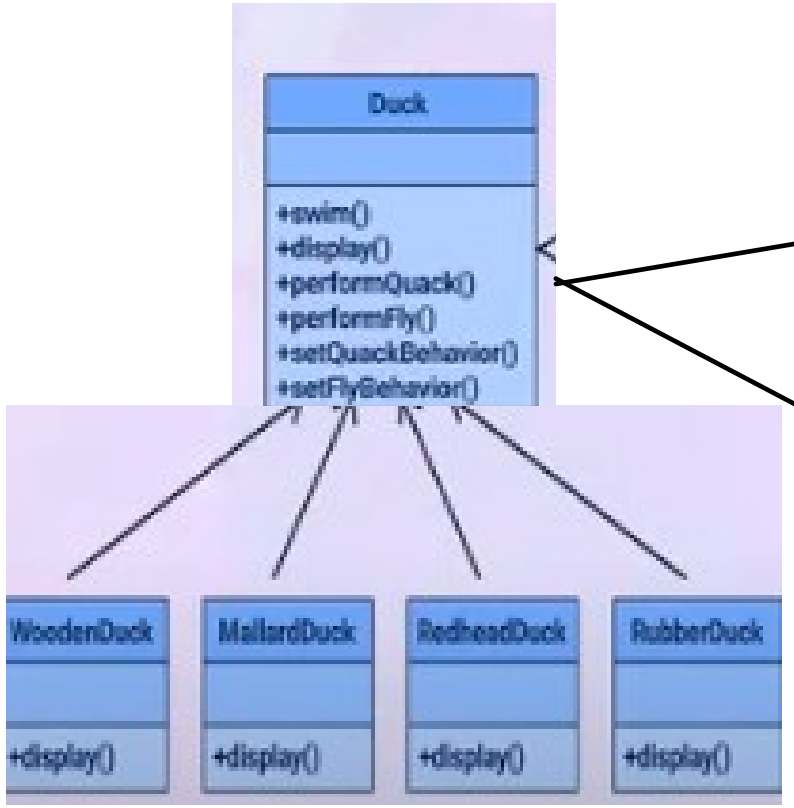
STRATEGY DESIGN PATTERN (EXAMPLE)





SOLUTION

Associate Algorithms:



Quack Behavior

Quack()

Fly Behavior

Fly()

SOLUTION

```
public abstract class Duck {
  FlyBehavior  flyBehavior;
  QuackBehavior  quackBehavior;
  public Duck() { }
  public abstract void display();
  public void performFly() {
    flyBehavior.fly();
  }
  public void performQuack() {
    quackBehavior.quack();
  }
  // other
```

```
public class MallardDuck extends Duck {
  public MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new fly();
  }
  // other
}
```

SOFTWARE TESTING

TESTING METHODOLOGIES & TYPES

Black box testing

White box testing

TESTING LEVELS

Unit testing

Integration testing

System testing

Acceptance testing

EXAMPLE

A	b	Expected result
“cat”	“dog”	False
“”	“”	True
“hen”	“hen”	True
“hen”	“heN”	False
“”	“”	False
“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ ”	True

EXAMPLE

```
bool isStringsEqual(char a[],
char b[]) {

      bool result;

      if (strlen(a) != strlen(b))
      {
      result = false;
      }
```

```
else {
      for ( int i =0; i < strlen(a); i++)
      {
       if ( a[i] == b[i] )
      {  result = true;      }
      else
       { result = false;  }
      }

        }

     return result;
}
```

BLACK BOX TESTING

BLACK BOX TESTING

No knowledge of internal design or code required.

Tests are based on requirements and functionality.

- Not based on any knowledge of internal design or code.
- Covers all combined parts of a system.
- Tests are data driven.

TYPES OF BLACK BOX TESTING

1. Functional testing
2. System testing
3. End-to-end testing
4. Sanity testing
5. Regression testing
6. Acceptance testing
7. Load testing
8. Stress testing
9. Install/uninstall testing
10. Recovery testing
11. Compatibility testing
12. Exploratory testing
13. Comparison testing
14. Alpha testing
15. Beta testing
16. Mutation testing

Functional testing

Black box type testing geared to functional requirements of an application.

System testing

Black box type testing that is based on overall requirements specifications; covering all combined parts of the system.

End-to-end testing

Similar to system testing; involves testing of a complete application environment in a situation that mimics real-world use.

Sanity testing

Initial effort to determine if a new software version is performing well enough to accept it for a major testing effort.

Regression testing

Re-testing after fixes or modifications of the software or its environment.

Acceptance testing

Final testing based on specifications of the end-user or customer.

Load testing

Testing an application under heavy loads.

Eg. Testing of a web site under a range of loads to determine, when the system response time degraded or fails.

Stress Testing

Testing under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database etc.

Term often used interchangeably with ‘load’ and ‘performance’ testing.

Performance testing

Testing how well an application complies to performance requirements

Install/uninstall testing

Testing of full, partial or upgrade install/uninstall process.

Recovery testing

Testing how well a system recovers from crashes, HW failures or other problems.

Compatibility testing

Testing how well software performs in a particular HW/SW/OS/NW environment.

Exploratory testing / ad-hoc testing

Informal SW test that is not based on formal test plans or test cases; testers will be learning the SW in totality as they test it.

Comparison testing

Comparing SW strengths and weakness to competing products.

Alpha testing

Testing done when development is nearing completion; minor design changes may still be made as a result of such testing.

Beta-testing

Testing when development and testing are essentially completed and final bugs and problems need to be found before release.

Mutation testing

To determining if a set of test data or test cases is useful, by deliberately introducing various bugs.

Re-testing with the original test data/cases to determine if the bugs are detected.

WHITE BOX TESTING

White box testing / Structural testing

Based on knowledge of internal logic of an application's code

Based on coverage of code statements, branches, paths, conditions.

Tests are logic driven .

MCCABE'S COMPLEXITY METRIC

The cyclomatic complexity of the program is computed from its control flow graph (CFG) using the formula:

$$V(G) = Edges - Nodes + 2$$

or by counting the conditional statements and adding 1

This measure determines the basis set of linearly independent paths and tries to measure the complexity of a program.

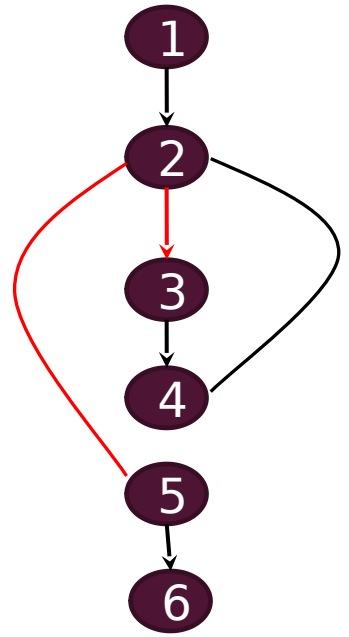
CYCLOMATIC COMPLEXITY

V(G) = Edges – Nodes + 2
*V(G) = 6 – 6 + 2 = **2***

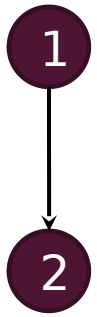
V(G) = conditional statements + 1
= 1 + 1 = **2**

Two linearly independent paths:
1, 2, 5, 6
1, 2, 3, 4, 2, 5, 6

Complexity = 2 < 10 => good quality



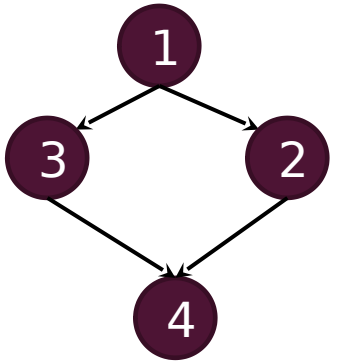
PROGRAM FLOW GRAPH
BASIC CONTROL FLOW GRAPHS



A sequence:

X = 1;

Y = X * 10;



If condition:

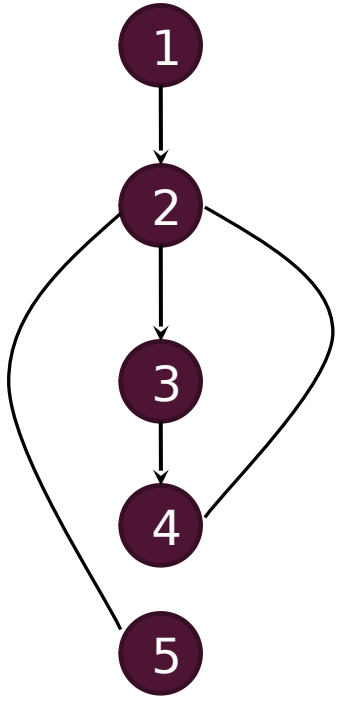
If … Then

…

Else

…

End if



While loop:

While … do

…

statements

…

End while

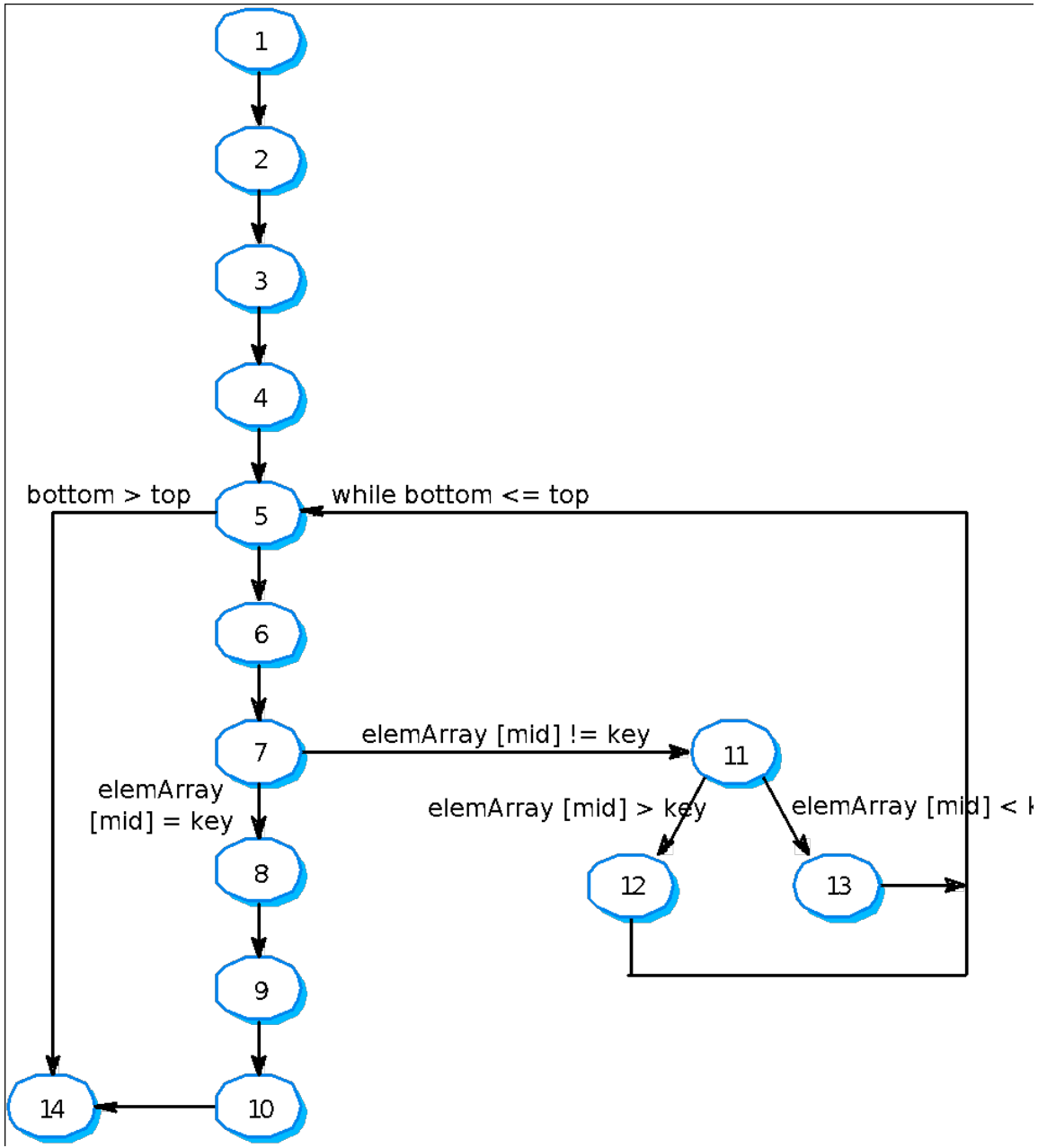
PROGRAM FLOW GRAPH
BASIC CONTROL FLOW GRAPHS

```
class IfStatement {
  public static void main(String[] args) {

    int number = 10;

      if (number > 0) {
       System.out.println("The number is positive.");
              }
      else {
      System.out.println("The number is negarive.");
                  }

    System.out.println("Statement outside if block");
     }
}
```



V(G) = Edges – Nodes + 2
*V(G) = 16 – 14 + 2 = **4***

V(G) = conditional statements + 1

= 3 + 1 = **4**

four linear independent paths

Complexity = 4 < 10 => good quality

V(G)	Risk
1 – 10	easy program, low risk
11 – 20	complex program, tolerable risk
21 – 50	complex program, high risk
>50	impossible to test, extremely high risk

Coverage:

Statement Coverage:

In this scheme, statements of the code are tested for a successful test that checks all the statements lying on the path of a successful scenario.

Branch Coverage:

In this scheme, all the possible branches of decision structures are tested. Therefore, sequences of statements following a decision are tested.

Path Coverage:

In path coverage, all possible paths of a program from input instruction to the output instruction are tested. An exhaustive list of test cases is generated and tested against the code.

HAVE A GOO DAY!