***Student No.: 18250038 Name: Shreyas Satish Angchekar***

**Title: Shakespeare Monkey’s Problem solved Using Genetic Algorithm**

**Abstract:** In this article I would like to demonstrate the famous Shakespeare Monkey’s Problem also famous as infinite Monkeys problem solving it with Genetic algorithm. As we know these are broader concepts in itself so would like to explain them step-step. Also I have demonstrated Singleton Design Pattern, Factory Design Pattern and Abstract Factory Pattern in the form of Project which is a Hospital Scenario. In Hospital Project there are various packages and factories implemented. Furthermore I have tried to apply most of the concepts which are studied so far in the CS613 Module (Advanced Concepts in Object Oriented Programming).

**Keywords:** Genetic Algorithm, Mutation, Crossover, Selection, FitnessEvaluator, Infinite\_Monkeys, Singleton, Strategy, Factory Pattern, Abstract\_Factory\_Pattern.

**Introduction:**  To start it is very important to understand what Shakespeare’s Monkey’s problem in computation perspective is.

***Shakespeare’s Monkey’s Problem:***

The infinite monkey problem or Shakespeare’s Monkey’s Problem states that a monkey hitting keys at [random](https://en.wikipedia.org/wiki/Randomness) on a typewriter keyboard for an [infinite](https://en.wikipedia.org/wiki/Infinity) amount of time will [almost surely](https://en.wikipedia.org/wiki/Almost_surely) type a given text, such as the complete works of [William Shakespeare](https://en.wikipedia.org/wiki/William_Shakespeare) [1]. In fact, the monkey would almost surely type every possible finite text an infinite number of times. However, the [probability](https://en.wikipedia.org/wiki/Probability) that monkeys filling the [observable universe](https://en.wikipedia.org/wiki/Observable_universe) would type a complete work such as Shakespeare's [Hamlet](https://en.wikipedia.org/wiki/Hamlet) is so tiny that the chance of it occurring during a period of time hundreds of thousands of orders of magnitude longer than the [age of the universe](https://en.wikipedia.org/wiki/Age_of_the_universe) is *extremely* low (but technically not zero)[2].

In this context, "almost surely" is a mathematical term with a precise meaning, and the "monkey" is not an actual monkey, but a [metaphor](https://en.wikipedia.org/wiki/Metaphor) for an abstract device that produces an endless [random sequence](https://en.wikipedia.org/wiki/Random_sequence) of letters and symbols. One of the earliest instances of the use of the "monkey metaphor" is that of French mathematician [Émile Borel](https://en.wikipedia.org/wiki/%C3%89mile_Borel) in 1913,[[1]](https://en.wikipedia.org/wiki/Infinite_monkey_theorem#cite_note-:0-1) but the first instance may have been even earlier [3].

Variants of the problem include multiple and even infinitely many typists, and the target text varies between an entire library and a single sentence. [Jorge Luis Borges](https://en.wikipedia.org/wiki/Jorge_Luis_Borges) traced the history of this idea from [Aristotle](https://en.wikipedia.org/wiki/Aristotle)'s [*On Generation and Corruption*](https://en.wikipedia.org/wiki/On_Generation_and_Corruption) and [Cicero](https://en.wikipedia.org/wiki/Cicero)'s [*De natura deorum*](https://en.wikipedia.org/wiki/De_natura_deorum) (On the Nature of the Gods), through [Blaise Pascal](https://en.wikipedia.org/wiki/Blaise_Pascal) and [Jonathan Swift](https://en.wikipedia.org/wiki/Jonathan_Swift), up to modern statements with their iconic simians and typewriters. In the early 20th century, Borel and [Arthur Eddington](https://en.wikipedia.org/wiki/Arthur_Eddington) used the theorem to illustrate the timescales implicit in the foundations of [statistical mechanics](https://en.wikipedia.org/wiki/Statistical_mechanics).

So to conclude in simple words, Shakespeare’s Monkey is the problem type in which we start from a random characters from any huge character set and then by trying any random new characters to match with the target and thereby reach the target string [4].

Now as that we have understood the logic of Infinite Monkey’s that is Shakespeare’s Monkey’s Problem, let see the Genetic Algorithm approach to solve it. For this purpose we should first look at the concept of Genetic Algorithm.

***Genetic Algorithm:***

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, non-differentiable, stochastic, or highly nonlinear [4].

* ***Initial Population:*** The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome [5].

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

* ***Selection*** *rules* select the individuals, called *parents*, that contribute to the population at the next generation.
* ***Crossover*** *rules* combine two parents to form children for the next generation.
* ***Mutation*** *rules* apply random changes to individual parents to form children [6].

Beside the rules the following thing is most important,

* ***Fitness Function :***The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score [7].

Thus A **genetic algorithm** is a search heuristic that is inspired by Charles Darwin’s theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

**Project Scenario:** We are considering an Hospital Environment to demonstrate the Genetic Algorithm solution to solve Infinite Monkey’s Problem. In Hospital Project there are two factories namely, 1. Branch\_Factroy and 2. Staff Factory. These factories are contained in the abstract factory which is named as Hospital\_Factory . In each simple factory there are object instances which are being created as and when requested by the client side. For example each instance is of the sub-class defined as follows:

For Staff Factory there are 3 concrete implementations of Staff as

* 1. Doctor.java which prints the name and experience of the doctor.
  2. Nurse.java which prints her suggestion.
  3. SupportBoy.java which prints he is available or not.

This Printing of Output is done and demonstrated with the help of Genetic Algorithm solving approach to the Shakespeare’s Monkey’s Problem.

Similarly for Branch\_Factory there are 3 concrete implementations of Branches as

1. Maynooth.java which prints address of Maynooth hospital branch.
2. Dublin.java which prints address of Dublin hospital branch.
3. Galway.java which prints address of Galway hospital branch.

Again this Printing of Output is done and demonstrated with the help of Genetic Algorithm solving approach to the Shakespeare’s Monkey’s Problem.

Demonstration of Important Concepts:

1. **Singleton Design Pattern** : Definition is “The singleton pattern is a design pattern that restricts the instantiation of a class to one object [8].” Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers [9].

**Code Snippet**:

In Doctor.java

**public** **void** suggest() {

// **TODO** Auto-generated method stub

Thread t2 = **new** Thread(**new** Runnable () {

**public** **void** run () {

String phrase = "Shreyas Angchekar";

System.***out***.println("Enter the population variety: ");

**int** p = 0;

**try** {

p = Integer.*parseInt*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Enter the rate of mutation: ");

**float** m = 0;

**try** {

m = Float.*parseFloat*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

Shakespearean\_monkey SM2 = Shakespearean\_monkey.*getInstance*();

System.***out***.println(SM2);

SM2.shakespearean\_Monkey\_Runner(phrase, p, m);

SM2.process();

}

});

Thread t1 = **new** Thread(**new** Runnable () {

**public** **void** run () {

String phrase = "Shreyas Angchekar";

System.***out***.println("Enter the population variety: ");

**int** p = 0;

**try** {

p = Integer.*parseInt*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Enter the rate of mutation: ");

**float** m = 0;

**try** {

m = Float.*parseFloat*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

Shakespearean\_monkey SM2 = Shakespearean\_monkey.*getInstance*();

System.***out***.println(SM2);

SM2.shakespearean\_Monkey\_Runner(phrase, p, m);

SM2.process();

}

});

t1.start();

**try** {

Thread.*sleep*(40000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

t2.start();

}

In Shakespearean\_monkey.java

**public** **static** Shakespearean\_monkey *obj*;

**private** Shakespearean\_monkey (){

System.***out***.println("Instance Created for the first time");

}

**public** **static** Shakespearean\_monkey getInstance() {

**if** (*obj* == **null**)

{

**synchronized** (Shakespearean\_monkey.**class**) {

**if** (*obj* == **null**)

*obj* = **new** Shakespearean\_monkey();

}

}

**return** *obj*;

}

In Factory\_Test.java (main)

Hospital\_Factory staff\_factory = Factory\_Producer.*getFactory*("staff");

Hospital\_Staff doctor = (Hospital\_Staff) staff\_factory.createStaff("emergency", "Doctor", "Shreyas Angchekar", 12);

doctor.suggest();

***Code Explanation***: In the above code I am trying to demonstrate Singleton Design Pattern, so I have created 2 threads t1 and t2 to prove that logic follows the Singleton Design Pattern. We have used the keywords Synchronized to achieve Singleton on the Shakespearean\_Monkey.java class on Shakespearean\_Monkey\_Runner.java class. The output is demonstrated with help of print statement which prints the object instance for both the threads. To ensure successful demonstration of output we have used thread.sleep() method after starting one thread.

OUTPUTs:

Doctor Name: Shreyas Angchekar With experience of 12 Years

Enter the population variety:

1000

Enter the rate of mutation:

0.01

Instance Created for the first time

MonkeyLogic.Shakespearean\_monkey@70c7ffd4

Mutation Rate : 1.0

Population = : 1000

Generation: 1 ??zt\_?,=m3X%|eO>F

Generation: 2 m?\*M?\_-.AQ@??'>n2

Generation: 3 Oh3@fw?,A&I?|??ap

Generation: 4 3h0??0rgL?=h?Nx?

Generation: 5 ??n>+]~B/,g?h$k|l

Generation: 6 G?H-wQ?j~MHS)YBaT

Generation: 7 gh?5s?~jA6IFh?C}o

Generation: 8 W4(1?^gttn9c5rkfn

Generation: 9 ?h?'C??cASg=rMjcr

Generation: 10 &gFT?|?jA~zJheA??

Generation: 11 S?S????^v/?csezav

Generation: 12 Sh3td?1???g=?Mjcr

Generation: 13 ~hrcyB3jA?g=heza?

Generation: 14 S^<(iQtA?gc?r,a9

Generation: 15 dh)=qa????"=heN|r

Generation: 16 [h?cqa?jA9?R?ekar

Generation: 17 ShllOvfjA\_?b~eA<?

Generation: 18 S?S?y1?484[=~e.-r

Generation: 19 Oh3gf0~ ?xg=heAav

Generation: 20 1hUexws ?rUchekfr

Generation: 21 th\*@ya^xANg`he'ar

Generation: 22 d^/Jyah-8ng?h?Bar

Generation: 23 Sh?eym=#A,gy?ekar

Generation: 24 Shl?s?^ tngcheAa9

Generation: 25 ShE|yS? A\_?cheAar

Generation: 26 rhrAya\*DAvgc !Nar

Generation: 27 Shr?ya?jA?gcheNar

Generation: 28 yh`?y?U AG?csekar

Generation: 29 dhreya[ A?gE?eBa?

Generation: 30 Sh$Jya?xA:gnhekvr

Generation: 31 ShreyT^ An/=hekar

Generation: 32 hhheyaU Aogch|kor

Generation: 33 dhr?yaa A/gche1a9

Generation: 34 Sh?Tyas A?gAeAar

Generation: 35 S^tey=?-?,'=he:6r

Generation: 36 Shr?ya^AA?gc? zfr

Generation: 37 Shrep.s Angch?kar

Generation: 38 ShEeFaX A?g?he:a!

Generation: 39 Sh?||a^#A,gcBek?r

Generation: 40 Sh`?yaQDA;gnh?kar

Generation: 41 .???yaE ArgcheNar

Generation: 42 Shrey.\*D?ngche?ar

Generation: 43 .hre|,s Angc?ekr?

Generation: 44 Shrdya^ Angchhka?

Generation: 45 ?hr+ya? Anvchekar

Generation: 46 Sh??ya? A,gchhkar

Generation: 47 VhreyaQqAngcheoar

Generation: 48 Sh`|?as A;gcheoz?

Generation: 49 2hrey!? Angchekcr

Generation: 50 Shreyav An7c?ekar

Generation: 51 ShEey?? Anichekar

Generation: 52 Shr?ya?#A?g?hekar

Generation: 53 ~/Eeyy? (ngch 2ar

Generation: 54 Shreya? |ngch?kor

Generation: 55 ~hrey3? Angch$kar

Generation: 56 Shreya?VAn8cKe:ar

Generation: 57 Sh?eya^ Anichek6r

Generation: 58 Sh)e|?? A?gcaekar

Generation: 59 Dhre?at Angc~ekar

Generation: 60 Shreha^ g$gc~ekar

Generation: 61 Sh|?ya} AngvheTar

Generation: 62 Sh2eyas Ajgchekar

Generation: 63 Shreya? Ang?hekar

Generation: 64 Shreyac Angchekar

Generation: 65 Sh?e&at\_Angche\*ar

Generation: 66 uhrehT? Angc?ekar

Generation: 67 Sh\*eya? A,gcheoar

Generation: 68 Sh?eyas Angchek"r

Generation: 69 Shreyas AnSchekar

Generation: 70 Shreya'?Angch0kar

Generation: 71 ShreyS} An4c#Xkar

Generation: 72 Shreya?jAngchekar

Generation: 73 Shreya? ACgchekar

Generation: 74 ShURyas Au)chekar

Generation: 75 Shreyas AEgShekar

Generation: 76 Sh\*eyasj0ngche?ar

Generation: 77 Sh$eyL? An?chekar

Generation: 78 -h?ema? Angca]kar

Generation: 79 Sh>eyas Angch:kao

Generation: 80 Shreya? Angchekar

Generation: 81 Shrepas Angc5ekPr

Generation: 82 Shreyas Angchekar

Generation of Solution: 82

Shreyas Angchekar

Enter the population variety:

1000

Enter the rate of mutation:

0.02

MonkeyLogic.Shakespearean\_monkey@70c7ffd4

Mutation Rate : 2.0

Population = : 1000

Generation: 1 D?oPaZa?1%?By

Generation: 2 IH.H??IaM?{w?1

Generation: 3 R5?n52<a!x:c!?O

Generation: 4 Qu?16@kmsh0???b

Generation: 5 ?u?+' Lshi{O/v

Generation: 6 IH.H(e5a?a>672}

Generation: 7 ?pRpi{kifhio Ya

Generation: 8 |K??o\_Ls?iw,?a

Generation: 9 s5?h| 1a=?{w#??

Generation: 10 R)A?' L?Mhu?DC?

Generation: 11 =uc0^B??fhh<`?a

Generation: 12 T,ch? JxMhioMC#

Generation: 13 ?sLh?,?%fhiQ?Ca

Generation: 14 RRl?} `sa?weBa

Generation: 128 R2cti Ta0EYwala

Generation: 129 Ruchi kashiuala

Generation: 130 Ru?hi H?wkiwala

Generation: 131 Ruch? kashQ:a?a

Generation: 132 Ruzhi?kashiwal(

Generation: 133 Ruc?i kashiwaca

Generation: 134 Ruuhi kas3iwala

Generation: 135 Ruc?ixka2qvwala

Generation: 136 Ruchi kashiwala

Generation of Solution: 136

Ruchi kashiwala

**NOTE:** The outputs of actual program are shortened to reduce the size of this document, I have skipped the display of every generation in Output.

***Observations***:

1. As we have used Singleton Design pattern only one instance is allowed so no matter how many treads are using the same function, they are using the same instance, to prove this we have highlighted the instance which was printed in the output.
2. As we increase the population the time or iterations to be more specific taken by Genetic Algorithm (GA) to solve and get to the target text is increased.
3. As we increase the mutation rate even by very little value the GA takes more and more iterations to calculate and reach the target String.

***Conclusion*:** Both Population and Mutation rate are very prime parameters which decide the performance of the GA, so it is very important to balance them to reach the solution as quickly as possible.

1. **Strategy Pattern:**

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), the strategy pattern (also known as the policy pattern) is a [behavioral](https://en.wikipedia.org/wiki/Behavioral_design_pattern) [software design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that enables selecting an [algorithm](https://en.wikipedia.org/wiki/Algorithm) at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use [10].

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern [11]. Thus in simple words we are not changing the local variable inside the class instead we are changing the Strategy or Composition of the class according to the client’s need.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object [12].

**Code Snippet**:

In Strategy.java

**package** strategy;

**public** **class** Strategy {

**public** **static** Doctor\_Strategy getStrategy(){

Doctor\_Strategy s;

**if** (Math.*random*()>0.5){

s = **new** SubStrategy\_A();

}

**else** s = **new** SubStrategy\_B();

**return** s;

}

**public** **static** Doctor\_Strategy setStrategy\_A(){

**return** **new** SubStrategy\_A();

}

**public** **static** Doctor\_Strategy setStrategy\_B(){

**return** **new** SubStrategy\_B();

}

}

In Doctor Strategy.java

package strategy;

import java.io.BufferedReader;

import factories.Staff\_Factory;

import java.io.InputStreamReader;

import hospital.Details;

public class Doctor\_Strategy extends Details {

BufferedReader ob=new BufferedReader(new InputStreamReader(System.in));

public void run(){

}

}

In SubStrategy\_A.java

**package** strategy;

**import** java.io.IOException;

**import** monkey\_logic.Shakespearean\_monkey;

**public** **class** SubStrategy\_A **extends** Doctor\_Strategy {

**public** **void** run() {

System.***out***.println("This is Strategy A");

String phrase = "Ruchi Kashiwala";

System.***out***.println("Enter the population variety: ");

**int** p = 0;

**try** {

p = Integer.*parseInt*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Enter the rate of mutation: ");

**float** m = 0;

**try** {

m = Float.*parseFloat*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

Shakespearean\_monkey SM2 = Shakespearean\_monkey.*getInstance*();

System.***out***.println(SM2);

SM2.shakespearean\_Monkey\_Runner(phrase, p, m);

SM2.process();

}

}

In SubStrategy\_B.java

**package** strategy;

**import** java.io.IOException;

**import** monkey\_logic.Shakespearean\_monkey;

**public** **class** SubStrategy\_B **extends** Doctor\_Strategy{

**public** **void** run() {

System.***out***.println("This is Strategy B");

String phrase = "Shreyas Angchekar";

System.***out***.println("Enter the population variety: ");

**int** p = 0;

**try** {

p = Integer.*parseInt*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Enter the rate of mutation: ");

**float** m = 0;

**try** {

m = Float.*parseFloat*(ob.readLine());

} **catch** (NumberFormatException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

Shakespearean\_monkey SM2 = Shakespearean\_monkey.*getInstance*();

System.***out***.println(SM2);

SM2.shakespearean\_Monkey\_Runner(phrase, p, m);

SM2.process();

}

}

In Driver.java

**package** driver\_runner;

**import** monkey\_logic.Shakespearean\_monkey;

**import** strategy.\*;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** abstract\_factory.Factory\_Producer;

**import** monkey\_logic.\*;

**import** factories.\*;

**public** **class** Driver {

**static** Doctor\_Strategy *s*;

**static** Doctor\_Strategy randomStrategy(){

*s* = Strategy.*getStrategy*();

**return** *s*;

}

**static** Doctor\_Strategy userStrategy(String user){

**if** ("Ruchi".equalsIgnoreCase(user))

*s*= Strategy.*setStrategy\_A*();

**if** ("Shreyas".equalsIgnoreCase(user))

*s* = Strategy.*setStrategy\_B*();

**return** *s*;

}

**public** **static** **void** main (String [] args) {

// User can use random Strategy: This is actual Hospital Scenario where Doctor is allocated as he/she is free

Doctor\_Strategy s = Driver.*randomStrategy*();

s.run()

// In this Strategy User demands doctor name "Shreyas" and program executes accordingly.

Doctor\_Strategy s21 = Driver.*userStrategy*("Shreyas");

S21.run();

// In this Strategy User demands doctor name "Ruchi" and program executes accordingly

Doctor\_Strategy s1 = Driver.*userStrategy*("Ruchi");

S1.run();

}

}

OUTPUTS;

1. For randomStrategy:

This is Strategy B

Enter the population variety:

2000

Enter the rate of mutation:

0.02

Instance Created for the first time

monkey\_logic.Shakespearean\_monkey@6d06d69c

Mutation Rate : 2.0

Population = : 2000

Generation: 1 fk3]??!l/?6\_wOia%

Generation: 2 \*?4??E6??C/8sK8a?

Generation: 113 iZrN?fs An8c?e6ar

Generation: 114 Shrey?s An=che?ar

Generation: 115 ShMyyas Angcheka"

Generation: 116 ghreyasJAng?oekar

Generation: 126 S?re?as Ang?h?kar

Generation: 127 ?hreyar h?gc?ear

Generation: 128 Shr?fis AnLc7ekar

Generation: 129 Shreyas Angchekar

Generation of Solution: 129

Shreyas Angchekar

1. When User Demands doctor by name for example Ruchi (use userStrategy(“Ruchi”)):

This is Strategy A

Enter the population variety:

1500

Enter the rate of mutation:

0.01

Instance Created for the first time

monkey\_logic.Shakespearean\_monkey@15db9742

Mutation Rate : 1.0

Population = : 1500

Generation: 1 R??n???{U\?O#2b

Generation: 2 \:u?Z +CVd??a$t

Generation: 3 RNJ0LGbchk4:?l|

Generation: 4 R:N?q?N|F,0T6l?

Generation: 5 RNq???$aQiwwlr

Generation: 6 ??Ah? a?y&wpv"

Generation: 7 hu^(? `CV?iwWl?

Generation: 52 RucYi Ga?iwala

Generation: 53 RuNh? Kas?Vwila

Generation: 54 Ruc3C KashiwaS~

Generation: 55 Ruch1 Kashiwa9a

Generation: 56 Ruchi Kash?wala

Generation: 57 Ru#hi KasGiIala

Generation: 58 Zuchi Mas?iwala

Generation: 59 Muchi4.asiwa'|

Generation: 60 Ruchi Kashiwala

Generation of Solution: 60

Ruchi Kashiwala

1. In Similar way Doctor “Shreyas” can be passed in userStrategy(“”) to get Strategy pattern Demonstrated.

**NOTE:** The outputs of actual program are shortened to reduce the size of this document, I have skipped the display of every generation in Output.

***Conclusion & Observations:*** Thus we can see that I have implemented 2 Strategies in code first is randomStrategy which changes composition ‘Strategy’ of Driver.java class in random fashion and secondly userStrategy(“ ”) which takes in String as argument and sets the composition ‘Strategy’ of Driver.java class accordingly. Thus I feel Strategy pattern is demonstrated successfully.

***Note:*** Just to keep Data abstraction in mind and not using “new” keyword in the main class (Driver.java), I have used static keyword for Context class Strategy which invokes Doctor\_Strategy accordingly.

1. **Abstract Factory**

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to [Factory Pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories [13].

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type [14].

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object [15].

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern [16].

**Code Snippet:**

In Hospital\_Factory.java

**package** Factories;

**import** Hospital.\*;

**public** **interface** Hospital\_Factory {

Details createStaff (String keyword,String type, String aName, **int** exp);

BranchDetails createBranch(String city, **int** nof, **int** earnings);

}

In Factory\_Producer.java

**package** AbstractFactory;

**import** Factories.\*;

**public** **class** Factory\_Producer {

**public** **static** Hospital\_Factory getFactory(String factoryType) {

**if** ("staff".equalsIgnoreCase(factoryType))

{

**return** **new** Staff\_Factory();

}

**else**

**if** ("branch".equalsIgnoreCase(factoryType))

{

**return** **new** Branch\_Factory();

}

**return** **null**;

}

}

In Driver.java (main)

Hospital\_Factory branch\_factory = Factory\_Producer.*getFactory*("branch");

Hospital\_Branches dublin = (Hospital\_Branches) branch\_factory.createBranch("Dublin", 500, 5210);

dublin.displayDetailedAddress();

In Dublin.java

package hospital;

/\*

\* This is one of the Concrete Implementations for Branch\_Factory Demonstration which is also used to demonstrate Abstract\_Factory.

\*/

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import factories.Hospital\_Branches;

import monkey\_logic.Shakespearean\_monkey;

public class Dublin extends BranchDetails implements Hospital\_Branches {

@Override

public void setDetails(String aAddress, int nos, int aEarnings) {

// TODO Auto-generated method stub

Address = aAddress;

numberOfStaff = nos;

earnings = aEarnings;

}

@Override

public String getAddress() {

// TODO Auto-generated method stub

return Address;

}

@Override

public int getNumberOfStaff() {

// TODO Auto-generated method stub

return numberOfStaff;

}

@Override

public int getEarnings() {

// TODO Auto-generated method stub

return earnings;

}

@Override

public void displayDetailedAddress() {

// TODO Auto-generated method stub

deatilAddress = "Colon Street Dublin";

BufferedReader ob=new BufferedReader(new InputStreamReader(System.in));

System.out.println("Searching Address for you");

System.out.println("Enter the population variety: ");

int p = 0;

try {

p = Integer.parseInt(ob.readLine());

} catch (NumberFormatException | IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("Enter the rate of mutation: ");

float m = 0;

try {

m = Float.parseFloat(ob.readLine());

} catch (NumberFormatException | IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

Shakespearean\_monkey SM3 = Shakespearean\_monkey.getInstance();

System.out.println(SM3);

SM3.shakespearean\_Monkey\_Runner(deatilAddress, p, m);

SM3.process();

System.out.println("The detailed Address is: "+deatilAddress);

}

}

OUTPUTs

This is Dublin Branch and having 500 staff and earning 5210

Searching Address for you

Enter the population variety:

1500

Enter the rate of mutation:

0.01

Instance Created for the first time

MonkeyLogic.Shakespearean\_Monkey@15db9742

Mutation Rate : 1.0

Population = : 1500

Generation: 1 JL??+zX\_r??mqX'gl?(

Generation: 2 z"t}x?S#F?$"+?1?a?w

Generation: 3 ?o'-nsl\r?1?5V3s-0a

Generation: 4 ='?@`:S!^P-doD?:??

Generation: 5 CBYTn`,nL\*6L?CA??\*^

Generation: 6 -olQ1QSD`?P8ub?b@i4

Generation: 7 ev8o?B9[Qe:l4?Pb9i?

Generation: 8 ?5uon?StGas6UnE]{?X

Generation: 122 Colon Strlet Dub?in

Generation: 123 Colon /treet Oublin

Generation: 124 Col?n Street Du?lin

Generation: 125 CoGon StreetTDublin

Generation: 126 Colon StN1Ig Dublin

Generation: 127 ColXn StPEet Du?lin

Generation: 128 Colon St>eet Dyb/jn

Generation: 129 Colon Etreet Dub\*in

Generation: 130 Colon Str`et Dublin

Generation: 131 ?olon Strert Dublin

Generation: 132 Colon St#eet DuGlon

Generation: 133 Colon St/eet Dubl?n

Generation: 134 Colon StreWt Dob-in

Generation: 135 nolon?St?eet Dublin

Generation: 136 o/lop Street D>bl?n

Generation: 137 ColpnjStreBt Dublin

Generation: 138 Co?on?St?eet D\blin

Generation: 139 Colon StreeA Dublin

Generation: 140 Colon j(reAt Dubzin

Generation: 141 Colon Stree< D'bl7?

Generation: 142 C?lon Street D2blin

Generation: 143 Colon Street gu?lin

Generation: 144 C?lon Str1et Dublin

Generation: 145 Colon Street Dublin

Generation of Solution: 145

Colon Street Dublin

The detailed Address is: Colon Street Dublin

**NOTE:** The outputs of actual program are shortened to reduce the size of this document, I have skipped the display of every generation in Output.

***Note:*** Similarly with the help of factory (Branch\_Factory.java) reference we can collect the instances of concrete implementations like Maynooth.java and Galway.java and with the help of Staff\_Factory.java reference we can collect the reference of Staffs like Doctor.java, Nurse.java and SupportBoy.java. Thus with the reference of abstract\_factory (Hospital\_Factory.java) we can collect the references of Branch\_Factory.java or Staff\_Factory.java.

***Code Explanation and Observation:***

In the above code I have tried to implement abstract factory Hospital\_Factory.java in which there are 2 simple factories namely, Branch\_Factory and Staff\_Factory. I have used the super-class as the return type of all the methods in them.

In main we are just calling the Factory\_Producer.java ‘s method getFactory() which brings us the specified simple-factory that is branch or staff factory, furthermore we use specific factory super-class type to get the concrete type for example Hospital\_Staff is used to get object instance of Doctor.java and similarly Hospital\_Branches is used to get the object instances of Dublin.java etc.

Thus Abstract\_Factory is demonstrated:

**DEMONSTRATION OF OTHER CONEPTS AS SPECIFIED IN MARKING SCHEME**

1. **DATA Abstraction:**

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send [17].

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces [18].

For Example in our Hospital Project The user only knows that Branch\_Factory.java object only brings the objects of Dublin.java, Maynooth.java and Galway.java, He does not know that concept of Factory\_Pattern or exactly how does the object is called and collected etc.

Similarly user does not know how the randomStartegy works. User just know what is the output of randomStrategy which is it shows the doctor name based on the data picked by the randomStrategy method.

**Code Snippet:**

In Strategy.java

**package** strategy;

/\*

\* This is basic Strategy class which passes the Doctor\_Strategy according to the user needs.

\* Note: I have implemented 2 strategies as stated in the main class.

\* 1. // User can use random Strategy: This is actual Hospital Scenario where Doctor is allocated as he/she is free

\* 2. // In this Strategy User demands doctor name "Shreyas" and program executes accordingly.

\* or //In this Strategy User demands doctor name "Ruchi" and program executes accordingly

\*/

**public** **class** Strategy {

**public** **static** Doctor\_Strategy getStrategy(){

Doctor\_Strategy s;

**if** (Math.*random*()>0.5){

s = **new** SubStrategy\_A();

}

**else** s = **new** SubStrategy\_B();

**return** s;

}

**public** **static** Doctor\_Strategy setStrategy\_A(){

**return** **new** SubStrategy\_A();

}

**public** **static** Doctor\_Strategy setStrategy\_B(){

**return** **new** SubStrategy\_B();

}

}

In Driver.java

// User can use random Strategy: This is actual Hospital Scenario where Doctor is allocated as he/she is free

Doctor\_Strategy s = Driver.*randomStrategy*();

s.run();

// In this Strategy User demands doctor name "Shreyas" and program executes accordingly.

Doctor\_Strategy s21 = Driver.*userStrategy*("Shreyas");

s.run();

// In this Strategy User demands doctor name "Ruchi" and program executes accordingly

Doctor\_Strategy s1 = Driver.*userStrategy*("Ruchi");

s1.run();

OUTPUTS:

This is Strategy B

Enter the population variety:

2000

Enter the rate of mutation:

0.02

Instance Created for the first time

monkey\_logic.Shakespearean\_monkey@6d06d69c

Mutation Rate : 2.0

Population = : 2000

Generation: 1 fk3]??!l/?6\_wOia%

Generation: 2 \*?4??E6??C/8sK8a?

Generation: 113 iZrN?fs An8c?e6ar

Generation: 114 Shrey?s An=che?ar

Generation: 115 ShMyyas Angcheka"

Generation: 116 ghreyasJAng?oekar

Generation: 126 S?re?as Ang?h?kar

Generation: 127 ?hreyar h?gc?ear

Generation: 128 Shr?fis AnLc7ekar

Generation: 129 Shreyas Angchekar

Generation of Solution: 129

Shreyas Angchekar

**NOTE:** The outputs of actual program are shortened to reduce the size of this document, I have skipped the display of every generation in Output.

***Code Explanation and Observation:*** Thus in the above output it is demonstrated that user does not know how the randomStrategy() method works, it just serves the user need that at current time which doctor is available and prints the name of the doctor which is free at that time with the help of logic implemented in the randomStrategy which is kept abtract from the user.

1. **Information Hiding:**

Information hiding data is not the full extent of information hiding. David Parnas first introduced the concept of information hiding around 1972. He argued that the primary criteria for system modularization should concern the hiding of critical design decisions. He stressed hiding "difficult design decisions or design decisions which are likely to change." Hiding information in that manner isolates clients from requiring intimate knowledge of the design to use a module, and from the effects of changing those decisions [19].

Thus information hiding for programmers is executed to prevent system design change. If design decisions are hidden, certain program code cannot be modified or changed. Information hiding is usually done for internally changeable code, which is sometimes especially designed not to be exposed. Such stored and derived data is not expounded upon, most generally. Change resilience of classes and ease of use by client objects are two by-products of hidden data [20].

Information hiding serves as an effective criterion for dividing any piece of equipment, software or hardware, into modules of functionality. For instance, a car is a complex piece of equipment. In order to make the design, manufacturing, and maintenance of a car reasonable, the complex piece of equipment is divided into [modules](https://en.wikipedia.org/wiki/Module_(programming)) with particular interfaces hiding design decisions. By designing a car in this fashion, a car manufacturer can also offer various options while still having a vehicle which is economical to manufacture.

For instance, a car manufacturer may have a luxury version of the car as well as a standard version. The luxury version comes with a more powerful engine than the standard version. The [engineers](https://en.wikipedia.org/wiki/Engineer) designing the two different car engines, one for the luxury version and one for the standard version, provide the same interface for both engines. Both engines fit into the engine bay of the car which is the same between both versions. Both engines fit the same transmission, the same engine mounts, and the same controls. The differences in the engines are that the more powerful luxury version has a larger displacement with a fuel injection system that is programmed to provide the fuel air mixture that the larger displacement engine requires.

In addition to the more powerful engine, the luxury version may also offer other options such as a better radio with CD player, more comfortable seats, a better suspension system with wider tires, and different paint colors. With all of these changes, most of the car is the same between the standard version and the luxury version. The radio with CD player is a module which replaces the standard radio, also a module, in the luxury model. The more comfortable seats are installed into the same seat mounts as the standard types of seats. Whether the seats are leather or plastic, or offer lumbar support or not, doesn't matter.

The engineers design the car by dividing the task up into pieces of work which are assigned to teams. Each team then designs their [component](https://en.wikipedia.org/wiki/Software_componentry) to a particular standard or interface which allows the team flexibility in the design of the component while at the same time ensuring that all of the components will fit together.

Motor vehicle manufacturers frequently use the same core structure for several different models, in part as a cost-control measure. Such a "[platform](https://en.wikipedia.org/wiki/Automobile_platform)" also provides an example of information hiding, since the floorplan can be built without knowing whether it is to be used in a sedan or a hatchback.

As can be seen by this example, information hiding provides flexibility. This flexibility allows a programmer to modify the functionality of a computer program during normal evolution as the computer program is changed to better fit the needs of users. When a computer program is well designed decomposing the source code solution into modules using the principle of information hiding, evolutionary changes are much easier because the changes typically are local rather than global changes [21].

Cars provide another example of this in how they interface with drivers. They present a standard interface (pedals, wheel, shifter, signals, gauges, etc.) on which people are trained and licensed. Thus, people only have to learn to drive a car; they don't need to learn a completely different way of driving every time they drive a new model. (Granted, there are manual and automatic transmissions and other such differences, but on the whole, cars maintain a unified interface.)

***Code Explanation and Observation:*** In my hospital project I have also tried and followed the same information hiding principal as stated and explained above with the help of car example. Firstly I have tried and kept all the design decisions as separate namely the processing logic, factories, concrete classes in the Hospital, strategies etc. Also I have marked the important fields in the processing logic (Shakespearean\_Monkey.java) like,

**private** DNA population[];//The DNA of the population

**private** String target;//The target string that is to be achieved

**int** targetLength;

**private** ArrayList<DNA> matingPool;//The mating pool for each generation

**private** **float** mutationRate;

**private** **int** generation = 1;

So that user cannot excess them only that class can change it which is handled correctly with the help of parametrized methods .

Hence the user doesn’t have to study of program each time he want new branch or staff, thus information hiding principle is achieved successfully.

1. **Separation of behavior and implementation particularly with respect to the Strategy & Factory design patterns.**

**Code Snippet:**

/\*

\* The following is the First Part of the code which Demonstrates Abstract\_Factory Design Pattern

\*/

Hospital\_Factory branch\_factory = Factory\_Producer.*getFactory*("branch");

Hospital\_Branches galway = (Hospital\_Branches) branch\_factory.createBranch("Galway", 500, 5210);

galway.displayDetailedAddress();

/\*

\* The Following is the Second part of the code which demonstrates Singleton Design Pattern

\*/

Hospital\_Factory staff\_factory = Factory\_Producer.*getFactory*("staff");

Hospital\_Staff doctor = (Hospital\_Staff) staff\_factory.createStaff("emergency", "Doctor", "Shreyas Angchekar", 12);

doctor.suggest();

/\*

\* The following is the Third part of the code which Demonstrates the Strategy Design Pattern

\*/

// User can use random Strategy: This is actual Hospital Scenario where Doctor is allocated as he/she is free

Doctor\_Strategy s = Driver.*randomStrategy*();

s.run();

// In this Strategy User demands doctor name "Shreyas" and program executes accordingly.

Doctor\_Strategy s21 = Driver.*userStrategy*("Shreyas");

s.run();

// In this Strategy User demands doctor name "Ruchi" and program executes accordingly

Doctor\_Strategy s1 = Driver.*userStrategy*("Ruchi");

s1.run();

***Code Explanation and Observation:***  I have kept the separate package for Strategy Pattern and Abstract\_factory and the Factories. They are demonstrated separately as described above in this document. In Strategy the composition that is object of Doctor\_Strategy.java class has been changed in the Driver.java class according the need of the user, whereas in Abstract\_Factory demonstrated the user is given the privilege to first choose which factory that is branch or staff and then select the respective branch of staff according to the need.

1. **Polymorphism Usage:**

**Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding [22].

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object [23].

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed [24].

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

***Code Explanation and Observation:***  As I have demonstrated above all concept of Abstract\_Factory (Hospital\_factory.java) and Simple Factory(Branch\_Factory.java and Staff\_Factory.java) have exhaustively used the concept of Polymorphism to collect the object of respective child (Concrete) Class and execute the methods present in them using the reference of parent-type.

1. **OPEN-CLOSE Principle:**

Let’s begin with a short summary of what the Open/Closed Principle is. It’s a principle for object oriented design first described by Bertrand Meyer that says that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” [25].

At first thought that might sound quite academic and abstract. What it means though is that **we should strive to write code that doesn’t have to be changed every time the requirements change**. How we do that can differ a bit depending on the context, such as our programming language. When using Java, C# or some other statically typed language the solution often involves inheritance and polymorphism, which is what this example will illustrate.

As applications evolve, changes are required. Changes are required when a new functionality is added or an existing functionality is updated in the application. Often in both situations, you need to modify the existing code, and that carries the risk of breaking the application’s functionality. For good application design and the code writing part, you should avoid change in the existing code when requirements change. Instead, you should extend the existing functionality by adding new code to meet the new requirements. You can achieve this by following the Open Closed Principle [26].

**“Open for extension “**: This means that the behavior of a software module, say a class can be extended to make it behave in new and different ways. It is important to note here that the term “extended ” is not limited to inheritance using the Java extend keyword. As mentioned earlier, Java did not exist at that time. What it means here is that a module should provide extension points to alter its behavior. One way is to make use of [polymorphism](https://springframework.guru/polymorphism-java/) to invoke extended behaviors of an object at run time [27].

**“Closed for modification “:** This means that the source code of such a module remains unchanged [27].

***Code Explanation and Observation:***  As I have used interfaces and parent classes wherever possible my code follows Open-Closed Principle the best example could be addition of one more branch Cork which extends Branch\_Details.java class and implements Hospital\_Branches.java interface to override the methods accordingly and use the common variable and calling it with the help of parent class reference which is Branch\_Factory.java to maintain the design requirements. Similarly for Staffs which implements the Hospital\_Staff.java interface and extends Details.java class to override the methods accordingly and use the common variable and calling it with the help of parent class reference which is Staff\_Factory.java to maintain the design requirements.

**References:**

1. Isaac, Richard E. (1995). The Pleasures of Probability. Springer. pp. 48–50. [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*0-387-94415-X*](https://en.wikipedia.org/wiki/Special:BookSources/0-387-94415-X).*Isaac generalizes this argument immediately to variable text and alphabet size; the common main conclusion is on p.50.*
2. [*"No words to describe monkeys' play"*](http://news.bbc.co.uk/2/hi/3013959.stm). BBC News. 2003-05-09*. Retrieved 2009-07-25*.
3. *Émile Borel (1913). "Mécanique Statistique et Irréversibilité". J. Phys. (Paris). Series 5. 3: 189–196.*
4. <https://www.sciencedirect.com/science/article/pii/S0377042705000774>
5. <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>
6. <https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html>
7. ["The Singleton design pattern - Problem, Solution, and Applicability"](http://w3sdesign.com/?gr=c05&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-16
8. <https://en.wikipedia.org/wiki/Genetic_algorithm>
9. <https://www.geeksforgeeks.org/singleton-design-pattern/>
10. <https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm>
11. ["The Strategy design pattern - Problem, Solution, and Applicability"](http://w3sdesign.com/?gr=b09&ugr=proble). *w3sDesign.com*. Retrieved 2017-08-12.
12. <https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm>
13. <https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm>
14. Freeman, Eric; Robson, Elisabeth; Sierra, Kathy; Bates, Bert (2004). Hendrickson, Mike; Loukides, Mike, eds. ["Head First Design Patterns"](http://shop.oreilly.com/product/9780596007126.do) (paperback). **1**. O'REILLY: 156. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [978-0-596-00712-6](https://en.wikipedia.org/wiki/Special:BookSources/978-0-596-00712-6). Retrieved 2012-09-12
15. <https://www.geeksforgeeks.org/abstract-factory-pattern/>
16. ["Abstract Factory: Implementation"](http://www.oodesign.com/abstract-factory-pattern.html). OODesign.com. Retrieved 2012-05-16.
17. <https://www.tutorialspoint.com/java/java_abstraction.htm>
18. <https://www.guru99.com/java-data-abstraction.html>
19. <https://www.javaworld.com/article/2075271/core-java/encapsulation-is-not-information-hiding.html>
20. <https://www.techopedia.com/definition/3814/information-hiding>
21. <https://en.wikipedia.org/wiki/Information_hiding>
22. <https://beginnersbook.com/2013/03/polymorphism-in-java/>
23. <https://www.tutorialspoint.com/java/java_polymorphism.htm>
24. <https://www.javatpoint.com/runtime-polymorphism-in-java>
25. <http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/>
26. <https://dzone.com/articles/the-openclosed-principle>
27. <https://springframework.guru/principles-of-object-oriented-design/open-closed-principle/>