OOPS Concepts

Core OOPS concepts are:

Abstraction

Encapsulation

Polymorphism

Inheritance

Association

Aggregation

Composition

[1. Abstraction](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#1-abstraction)

Abstraction is the concept of hiding the internal details and describing things in simple terms. For example, a method that adds two integers. The internal processing of the method is hidden from the outer world. There are many ways to achieve abstraction in object-oriented programmings, such as encapsulation and inheritance. A Java program is also a great example of abstraction. Here java takes care of converting simple statements to machine language and hides the inner implementation details from the outer world.

Further Reading: [What is Abstraction in OOPS?](https://www.digitalocean.com/community/tutorials/what-is-abstraction-in-oops)

[2. Encapsulation](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#2-encapsulation)

Encapsulation is the technique used to implement abstraction in object-oriented programming. Encapsulation is used for access restriction to class members and methods. [Access modifier](https://www.digitalocean.com/community/tutorials/java-access-modifiers) keywords are used for encapsulation in object oriented programming. For example, encapsulation in java is achieved using private, protected and public keywords.

[3. Polymorphism](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#3-polymorphism)

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism - compile time polymorphism and runtime polymorphism. Compile-time polymorphism is achieved by [method overloading](https://www.digitalocean.com/community/tutorials/method-overloading-in-java). For example, we can have a class as below.

public class Circle {

public void draw(){

System.out.println("Drwaing circle with default color Black and diameter 1 cm.");

}

public void draw(int diameter){

System.out.println("Drwaing circle with default color Black and diameter"+diameter+" cm.");

}

public void draw(int diameter, String color){

System.out.println("Drwaing circle with color"+color+" and diameter"+diameter+" cm.");

}

}

Here we have multiple draw methods but they have different behavior. This is a case of method overloading because all the methods name is same and arguments are different. Here compiler will be able to identify the method to invoke at compile-time, hence it’s called compile-time polymorphism. Runtime polymorphism is implemented when we have an “IS-A” relationship between objects. This is also called a method overriding because the subclass has to override the superclass method for runtime polymorphism. If we are working in terms of the superclass, the actual implementation class is decided at runtime. The compiler is not able to decide which class method will be invoked. This decision is done at runtime, hence the name as runtime polymorphism or dynamic method dispatch.

package com.journaldev.test;

public interface Shape {

public void draw();

}

package com.journaldev.test;

public class Circle implements Shape{

@Override

public void draw(){

System.out.println("Drwaing circle");

}

}

package com.journaldev.test;

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Drawing Square");

}

}

Shape is the superclass and there are two subclasses Circle and Square. Below is an example of runtime polymorphism.

Shape sh = new Circle();

sh.draw();

Shape sh1 = getShape(); //some third party logic to determine shape

sh1.draw();

In the above examples, java compiler doesn’t know the actual implementation class of Shape that will be used at runtime, hence runtime polymorphism.

[4. Inheritance](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#4-inheritance)

[Inheritance](https://www.digitalocean.com/community/tutorials/inheritance-java-example) is the object-oriented programming concept where an object is based on another object. Inheritance is the mechanism of code reuse. The object that is getting inherited is called the superclass and the object that inherits the superclass is called a subclass. We use extends keyword in java to implement inheritance. Below is a simple example of inheritance in java.

package com.journaldev.java.examples1;

class SuperClassA {

public void foo(){

System.out.println("SuperClassA");

}

}

class SubClassB extends SuperClassA{

public void bar(){

System.out.println("SubClassB");

}

}

public class Test {

public static void main(String args[]){

SubClassB a = new SubClassB();

a.foo();

a.bar();

}

}

[5. Association](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#5-association)

Association is the OOPS concept to define the relationship between objects. The association defines the multiplicity between objects. For example Teacher and Student objects. There is a one-to-many relationship between a teacher and students. Similarly, a student can have a one-to-many relationship with teacher objects. However, both student and teacher objects are independent of each other.

[Aggregation](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#aggregation)

Aggregation is a special type of association. In aggregation, objects have their own life cycle but there is ownership. Whenever we have “HAS-A” relationship between objects and ownership then it’s a case of aggregation.

[6. Composition](https://www.digitalocean.com/community/tutorials/oops-concepts-java-example#6-composition)

The composition is a special case of aggregation. The composition is a more restrictive form of aggregation. When the contained object in “HAS-A” relationship can’t exist on its own, then it’s a case of composition. For example, House has-a Room. Here the room can’t exist without the house. Composition is said to be better than inheritance, read more at [Composition vs Inheritance](https://www.digitalocean.com/community/tutorials/composition-vs-inheritance).

Further Reading: [Composition in Java](https://www.digitalocean.com/community/tutorials/composition-in-java-example)

That’s all for a quick round-up on OOPS concepts.

Tasks ================================================================

1. I have a pizza shop, which offers a Standard or Soft base. It also offers a selection of toppings from Tomato, corn, mushrooms. Customers can choose one base and many toppings. The cost of the pizza is the cost of the base plus the cost of the selected toppings. Write code to represent this model in an Object-Oriented approach and calculate the cost of a given pizza.

import java.util.ArrayList;

import java.util.List;

// Abstract class for Pizza Base

abstract class PizzaBase {

String name;

double cost;

public PizzaBase(String name, double cost) {

this.name = name;

this.cost = cost;

}

public double getCost() {

return cost;

}

public String getName() {

return name;

}

}

// Concrete classes for specific bases

class StandardBase extends PizzaBase {

public StandardBase() {

super("Standard", 100.0); // cost of Standard base is 100

}

}

class SoftBase extends PizzaBase {

public SoftBase() {

super("Soft", 120.0); // cost of Soft base is 120

}

}

// Topping class

class Topping {

String name;

double cost;

public Topping(String name, double cost) {

this.name = name;

this.cost = cost;

}

public double getCost() {

return cost;

}

public String getName() {

return name;

}

}

// Pizza class to represent the customer's pizza order

class Pizza {

private PizzaBase base;

private List<Topping> toppings = new ArrayList<>();

public Pizza(PizzaBase base) {

this.base = base;

}

public void addTopping(Topping topping) {

toppings.add(topping);

}

public double calculateCost() {

double totalCost = base.getCost();

for (Topping topping : toppings) {

totalCost += topping.getCost();

}

return totalCost;

}

public void displayOrder() {

System.out.println("Pizza Order:");

System.out.println("Base: " + base.getName() + " - ₹" + base.getCost());

System.out.println("Toppings:");

for (Topping topping : toppings) {

System.out.println(topping.getName() + " - ₹" + topping.getCost());

}

System.out.println("Total Cost: ₹" + calculateCost());

}

}

// Main class to test the pizza ordering

public class PizzaShop {

public static void main(String[] args) {

// Create pizza base

PizzaBase base = new SoftBase(); // Customer selects Soft base

// Create a pizza with the selected base

Pizza pizza = new Pizza(base);

// Add toppings

pizza.addTopping(new Topping("Tomato", 20.0)); // Customer selects Tomato topping

pizza.addTopping(new Topping("Corn", 30.0)); // Customer selects Corn topping

pizza.addTopping(new Topping("Mushroom", 25.0)); // Customer selects Mushroom topping

// Display the order and total cost

pizza.displayOrder();

}

}

Output:  
Pizza Order:

Base: Soft - ₹120.0

Toppings:

Tomato - ₹20.0

Corn - ₹30.0

Mushroom - ₹25.0

Total Cost: ₹195.0

2. A bouquet consists of different types of flowers, Roses, Jasmines, Lilys etc. Consider a rose costs 1$, jasmine costs 2$, Lily costs 3$. The cost of the bouquet is the sum of the cost of the flowers in it. Use the object-oriented approach to design the above situation and write a utility method to calculate the cost of the given bouquet.

import java.util.ArrayList;

import java.util.List;

// Abstract class for Flower

abstract class Flower {

String name;

double cost;

public Flower(String name, double cost) {

this.name = name;

this.cost = cost;

}

public double getCost() {

return cost;

}

public String getName() {

return name;

}

}

// Concrete classes for specific flowers

class Rose extends Flower {

public Rose() {

super("Rose", 1.0); // Rose costs $1

}

}

class Jasmine extends Flower {

public Jasmine() {

super("Jasmine", 2.0); // Jasmine costs $2

}

}

class Lily extends Flower {

public Lily() {

super("Lily", 3.0); // Lily costs $3

}

}

// Bouquet class to represent a bouquet of flowers

class Bouquet {

private List<Flower> flowers = new ArrayList<>();

// Add flowers to the bouquet

public void addFlower(Flower flower) {

flowers.add(flower);

}

// Method to calculate the total cost of the bouquet

public double calculateTotalCost() {

double totalCost = 0;

for (Flower flower : flowers) {

totalCost += flower.getCost();

}

return totalCost;

}

// Method to display the bouquet details

public void displayBouquet() {

System.out.println("Bouquet Details:");

for (Flower flower : flowers) {

System.out.println(flower.getName() + " - $" + flower.getCost());

}

System.out.println("Total Cost: $" + calculateTotalCost());

}

}

// Main class to test the bouquet creation and cost calculation

public class FlowerShop {

public static void main(String[] args) {

// Create a bouquet

Bouquet bouquet = new Bouquet();

// Add flowers to the bouquet

bouquet.addFlower(new Rose()); // Add Rose

bouquet.addFlower(new Jasmine()); // Add Jasmine

bouquet.addFlower(new Lily()); // Add Lily

bouquet.addFlower(new Rose()); // Add another Rose

// Display the bouquet and total cost

bouquet.displayBouquet();

}

}

Bouquet Details:

Rose - $1.0

Jasmine - $2.0

Lily - $3.0

Rose - $1.0

Total Cost: $7.0

3. As part of assessing your programming skills, can you design a Java program to manage a New Year's gift with various chocolates and sweets? The program should establish a hierarchy for these confectioneries,  create objects representing different sweets,calculate the total weight of a children's gift, implement a sorting mechanism for the chocolates based on a user-selected option, and identify candies within the gift that fall within a specified range of options. Please provide the code for each step and explain your design choices.

Confectionaries Base Class   
import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

// Abstract class for Confectionery

abstract class Confectionery {

String name;

double weight; // in grams

double sugarContent; // in grams

public Confectionery(String name, double weight, double sugarContent) {

this.name = name;

this.weight = weight;

this.sugarContent = sugarContent;

}

public double getWeight() {

return weight;

}

public double getSugarContent() {

return sugarContent;

}

public String getName() {

return name;

}

}

// Chocolate class extending Confectionery

class Chocolate extends Confectionery {

public Chocolate(String name, double weight, double sugarContent) {

super(name, weight, sugarContent);

}

}

// Sweet class extending Confectionery

class Sweet extends Confectionery {

public Sweet(String name, double weight, double sugarContent) {

super(name, weight, sugarContent);

}

}

Creating objects for Diff sweets and Confectionaries   
// Gift class to hold the chocolates and sweets

class Gift {

private List<Confectionery> items = new ArrayList<>();

// Add confectionery item to the gift

public void addItem(Confectionery item) {

items.add(item);

}

// Calculate the total weight of the gift

public double calculateTotalWeight() {

double totalWeight = 0;

for (Confectionery item : items) {

totalWeight += item.getWeight();

}

return totalWeight;

}

// Get the list of confectionery items

public List<Confectionery> getItems() {

return items;

}

// Sort chocolates based on user-selected option (weight or sugar content)

public void sortChocolates(String sortBy) {

Collections.sort(items, new Comparator<Confectionery>() {

@Override

public int compare(Confectionery c1, Confectionery c2) {

if (sortBy.equalsIgnoreCase("weight")) {

return Double.compare(c1.getWeight(), c2.getWeight());

} else if (sortBy.equalsIgnoreCase("sugarContent")) {

return Double.compare(c1.getSugarContent(), c2.getSugarContent());

}

return 0;

}

});

}

// Find candies (chocolates) within a specified range of sugar content

public List<Confectionery> findCandiesInRange(double minSugar, double maxSugar) {

List<Confectionery> filteredCandies = new ArrayList<>();

for (Confectionery item : items) {

if (item instanceof Chocolate && item.getSugarContent() >= minSugar && item.getSugarContent() <= maxSugar) {

filteredCandies.add(item);

}

}

return filteredCandies;

}

// Display the gift items

public void displayGift() {

for (Confectionery item : items) {

System.out.println(item.getName() + " - Weight: " + item.getWeight() + "g, Sugar Content: " + item.getSugarContent() + "g");

}

}

}  
  
Main Program:  
  
// Gift class to hold the chocolates and sweets

class Gift {

private List<Confectionery> items = new ArrayList<>();

// Add confectionery item to the gift

public void addItem(Confectionery item) {

items.add(item);

}

// Calculate the total weight of the gift

public double calculateTotalWeight() {

double totalWeight = 0;

for (Confectionery item : items) {

totalWeight += item.getWeight();

}

return totalWeight;

}

// Get the list of confectionery items

public List<Confectionery> getItems() {

return items;

}

// Sort chocolates based on user-selected option (weight or sugar content)

public void sortChocolates(String sortBy) {

Collections.sort(items, new Comparator<Confectionery>() {

@Override

public int compare(Confectionery c1, Confectionery c2) {

if (sortBy.equalsIgnoreCase("weight")) {

return Double.compare(c1.getWeight(), c2.getWeight());

} else if (sortBy.equalsIgnoreCase("sugarContent")) {

return Double.compare(c1.getSugarContent(), c2.getSugarContent());

}

return 0;

}

});

}

// Find candies (chocolates) within a specified range of sugar content

public List<Confectionery> findCandiesInRange(double minSugar, double maxSugar) {

List<Confectionery> filteredCandies = new ArrayList<>();

for (Confectionery item : items) {

if (item instanceof Chocolate && item.getSugarContent() >= minSugar && item.getSugarContent() <= maxSugar) {

filteredCandies.add(item);

}

}

return filteredCandies;

}

// Display the gift items

public void displayGift() {

for (Confectionery item : items) {

System.out.println(item.getName() + " - Weight: " + item.getWeight() + "g, Sugar Content: " + item.getSugarContent() + "g");

}

}

}

Output:   
Gift Contents:

Dairy Milk - Weight: 50.0g, Sugar Content: 30.0g

KitKat - Weight: 45.0g, Sugar Content: 25.0g

Snickers - Weight: 60.0g, Sugar Content: 35.0g

Gulab Jamun - Weight: 100.0g, Sugar Content: 50.0g

Rasgulla - Weight: 120.0g, Sugar Content: 55.0g

Ladoo - Weight: 80.0g, Sugar Content: 40.0g

Total weight of the gift: 455.0 grams

Sorting chocolates by weight:

KitKat - Weight: 45.0g, Sugar Content: 25.0g

Dairy Milk - Weight: 50.0g, Sugar Content: 30.0g

Snickers - Weight: 60.0g, Sugar Content: 35.0g

Gulab Jamun - Weight: 100.0g, Sugar Content: 50.0g

Rasgulla - Weight: 120.0g, Sugar Content: 55.0g

Ladoo - Weight: 80.0g, Sugar Content: 40.0g

Sorting chocolates by sugar content:

KitKat - Weight: 45.0g, Sugar Content: 25.0g

Dairy Milk - Weight: 50.0g, Sugar Content: 30.0g

Snickers - Weight: 60.0g, Sugar Content: 35.0g

Gulab Jamun - Weight: 100.0g, Sugar Content: 50.0g

Rasgulla - Weight: 120.0g, Sugar Content: 55.0g

Ladoo - Weight: 80.0g, Sugar Content: 40.0g

Finding chocolates with sugar content between 25g and 35g:

Dairy Milk - Sugar Content: 30.0g

KitKat - Sugar Content: 25.0g

Snickers - Sugar Content: 35.0g

4. In a home, there are multiple appliances. Each appliance consumes a certain amount of power when â€œSwitch onâ€. eg: Home can have FAN, Light, TV, Laptop, etc. when "Switch on" they consume FAN-1 unit, Light - 2 units, TV-3 units etc power. Create a method to calculate the amount of power consumption at any given point (current load). Sort the devices in an apartment by one of (any) the parameters.

Model Appliances and Power consumption  
import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

// Appliance class to represent an appliance in the home

class Appliance {

private String name;

private int powerConsumption; // Power consumption in units

private boolean isOn; // Status of the appliance (on or off)

public Appliance(String name, int powerConsumption) {

this.name = name;

this.powerConsumption = powerConsumption;

this.isOn = false; // By default, the appliance is off

}

public String getName() {

return name;

}

public int getPowerConsumption() {

return powerConsumption;

}

public boolean isOn() {

return isOn;

}

// Switch on the appliance

public void switchOn() {

isOn = true;

}

// Switch off the appliance

public void switchOff() {

isOn = false;

}

// Override toString() to easily print appliance details

@Override

public String toString() {

return name + " - Power: " + powerConsumption + " units - Status: " + (isOn ? "On" : "Off");

}

}

Creating a Home Class to manage appliances   
import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

// Appliance class to represent an appliance in the home

class Appliance {

private String name;

private int powerConsumption; // Power consumption in units

private boolean isOn; // Status of the appliance (on or off)

public Appliance(String name, int powerConsumption) {

this.name = name;

this.powerConsumption = powerConsumption;

this.isOn = false; // By default, the appliance is off

}

public String getName() {

return name;

}

public int getPowerConsumption() {

return powerConsumption;

}

public boolean isOn() {

return isOn;

}

// Switch on the appliance

public void switchOn() {

isOn = true;

}

// Switch off the appliance

public void switchOff() {

isOn = false;

}

// Override toString() to easily print appliance details

@Override

public String toString() {

return name + " - Power: " + powerConsumption + " units - Status: " + (isOn ? "On" : "Off");

}

}

Main Program  
  
public class HomeAppliances {

public static void main(String[] args) {

// Create a new home

Home home = new Home();

// Add appliances to the home

Appliance fan = new Appliance("Fan", 1);

Appliance light = new Appliance("Light", 2);

Appliance tv = new Appliance("TV", 3);

Appliance laptop = new Appliance("Laptop", 4);

home.addAppliance(fan);

home.addAppliance(light);

home.addAppliance(tv);

home.addAppliance(laptop);

// Switch on some appliances

fan.switchOn();

tv.switchOn();

// Display the current status of appliances

System.out.println("Appliances in the home:");

home.displayAppliances();

// Calculate the total power consumption

System.out.println("\nTotal power consumption: " + home.calculateTotalPowerConsumption() + " units");

// Sort appliances by power consumption

System.out.println("\nAppliances sorted by power consumption:");

home.sortAppliancesByPowerConsumption();

home.displayAppliances();

}

}

Output :  
Appliances in the home:

Fan - Power: 1 units - Status: On

Light - Power: 2 units - Status: Off

TV - Power: 3 units - Status: On

Laptop - Power: 4 units - Status: Off

Total power consumption: 4 units

Appliances sorted by power consumption:

Fan - Power: 1 units - Status: On

Light - Power: 2 units - Status: Off

TV - Power: 3 units - Status: On

Laptop - Power: 4 units - Status: Off

CORE JAVA

Task 1 and 2  
1. Daniel has 3 apples; Amber has 2 apples. How many apples do Daniel and Amber have together?

2. Daniel has 3.5 apples and Amber has 2.5 apples. How many apples do Daniel and Amber have together?  
  
public class AppleCount {

// Method to calculate the total number of apples

public static double calculateTotalApples(double danielApples, double amberApples) {

return danielApples + amberApples;

}

public static void main(String[] args) {

// First scenario (Integer apples)

int danielApples1 = 3;

int amberApples1 = 2;

int totalApples1 = (int) calculateTotalApples(danielApples1, amberApples1);

System.out.println("First scenario: Daniel and Amber together have " + totalApples1 + " apples.");

// Second scenario (Decimal apples)

double danielApples2 = 3.5;

double amberApples2 = 2.5;

double totalApples2 = calculateTotalApples(danielApples2, amberApples2);

System.out.println("Second scenario: Daniel and Amber together have " + totalApples2 + " apples.");

}

}

Output:  
First scenario: Daniel and Amber together have 5 apples.

Second scenario: Daniel and Amber together have 6.0 apples.  
  
Task 3:  
  
Create code which displays in console “Hello, NAME”, where NAME is taken from Command line (1 mark)

Approach1:  
public class HelloName {

public static void main(String[] args) {

if (args.length > 0) {

// Fetch the name from command line argument

String name = args[0];

System.out.println("Hello, " + name);

} else {

System.out.println("No name provided!");

}

}

}  
  
using Scanner approach:  
import java.util.Scanner;

public class HelloName {

public static void main(String[] args) {

// Create a scanner object to read input

Scanner scanner = new Scanner(System.in);

// Prompt user to enter their name

System.out.print("Please enter your name: ");

String name = scanner.nextLine(); // Read user input

// Display the greeting

System.out.println("Hello, " + name);

// Close the scanner

scanner.close();

}

}  
  
output   
Please enter your name: Daniel

Hello, Daniel  
  
Calculator Program  
  
public class Calculator {

public static void main(String[] args) {

if (args.length < 3) {

System.out.println("Usage: <number1> <operation> <number2>");

return;

}

// Parse input arguments

double number1 = Double.parseDouble(args[0]);

String operation = args[1];

double number2 = Double.parseDouble(args[2]);

double result = 0;

// Perform operation based on user input

switch (operation) {

case "+":

result = number1 + number2;

break;

case "-":

result = number1 - number2;

break;

case "\*":

result = number1 \* number2;

break;

case "/":

if (number2 != 0) {

result = number1 / number2;

} else {

System.out.println("Error: Division by zero.");

return;

}

break;

case "%":

result = number1 % number2;

break;

case "percentage":

result = (number1 \* number2) / 100;

break;

default:

System.out.println("Invalid operation! Use +, -, \*, /, %, or percentage.");

return;

}

System.out.println("Result: " + result);

}

}

Pyramid Task  
Generate Pyramid For a Given Number Example

--> Generate the Pyramid with while loop

--> Generate the Pyramid with do.. while loop

--> Generate the Pyramid with for loop

--> Ensure code is clean

=========================

Pyramid

Enter Number:

5

Enter X:

1

0

1 2

3 4 5

6 7 8 9

10 11 12 13 14

----------------------------------------------

Enter Number:

5

Enter X:

2

0

2 4

6 8 10

12 14 16 18

20 22 24 26 28

----------------------------------------------

Enter Number:

5

Enter X:

3

0

3 6

9 12 15

18 21 24 27

30 33 36 39 42

Using while loop  
import java.util.Scanner;

public class PyramidWhile {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter Number: ");

int number = scanner.nextInt();

System.out.print("Enter X: ");

int x = scanner.nextInt();

int count = 0;

int row = 0;

// Outer loop controls rows

while (row < number) {

int col = 0;

// Inner loop controls columns (number of elements in each row)

while (col <= row) {

System.out.print(count + "\t");

count += x;

col++;

}

System.out.println(); // Move to the next line

row++;

}

scanner.close();

}

}  
  
using do while   
import java.util.Scanner;

public class PyramidDoWhile {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter Number: ");

int number = scanner.nextInt();

System.out.print("Enter X: ");

int x = scanner.nextInt();

int count = 0;

int row = 0;

// Outer loop controls rows

do {

int col = 0;

// Inner loop controls columns (number of elements in each row)

do {

System.out.print(count + "\t");

count += x;

col++;

} while (col <= row);

System.out.println(); // Move to the next line

row++;

} while (row < number);

scanner.close();

}

}  
  
using for loop  
import java.util.Scanner;

public class PyramidFor {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter Number: ");

int number = scanner.nextInt();

System.out.print("Enter X: ");

int x = scanner.nextInt();

int count = 0;

// Outer loop controls rows

for (int row = 0; row < number; row++) {

// Inner loop controls columns (number of elements in each row)

for (int col = 0; col <= row; col++) {

System.out.print(count + "\t");

count += x;

}

System.out.println(); // Move to the next line

}

scanner.close();

}

}  
  
ARRAYS and Strings Programming   
  
Arrays: 1. Write a program to find the missing number in integer array of 1 to 100

public class MissingNumberInArray {

public static void main(String[] args) {

// Example array with one missing number (let's say 56 is missing)

int[] arr = new int[99];

int missingNumber = 56;

int index = 0;

for (int i = 1; i <= 100; i++) {

if (i != missingNumber) {

arr[index++] = i;

}

}

// Calculate the sum of numbers from 1 to 100

int expectedSum = 100 \* (100 + 1) / 2;

// Calculate the sum of the given array

int actualSum = 0;

for (int num : arr) {

actualSum += num;

}

// The missing number is the difference between the expected and actual sum

System.out.println("Missing number is: " + (expectedSum - actualSum));

}

}

2. Write a program to find biggest and smallest number in a array.

public class FindMinMaxInArray {

public static void main(String[] args) {

int[] arr = {45, 7, 89, 23, 56, 12, 99, 1, 67};

int min = arr[0];

int max = arr[0];

// Loop through the array to find min and max

for (int num : arr) {

if (num > max) {

max = num;

} else if (num < min) {

min = num;

}

}

System.out.println("Smallest number is: " + min);

System.out.println("Biggest number is: " + max);

}

}

Strings: 1. write a program to Print duplicate characters from String

public class FindMinMaxInArray {

public static void main(String[] args) {

int[] arr = {45, 7, 89, 23, 56, 12, 99, 1, 67};

int min = arr[0];

int max = arr[0];

// Loop through the array to find min and max

for (int num : arr) {

if (num > max) {

max = num;

} else if (num < min) {

min = num;

}

}

System.out.println("Smallest number is: " + min);

System.out.println("Biggest number is: " + max);

}

}

2. write a program to check if two strings are rotations of each other

public class StringRotationCheck {

public static void main(String[] args) {

String str1 = "ABCD";

String str2 = "DABC";

if (areRotations(str1, str2)) {

System.out.println("Strings are rotations of each other.");

} else {

System.out.println("Strings are not rotations of each other.");

}

}

public static boolean areRotations(String str1, String str2) {

// Check if lengths are equal and if str2 is a substring of str1 concatenated with itself

return str1.length() == str2.length() && (str1 + str1).contains(str2);

}

}  
  
Some Constrcutors Tasks   
1. Driving Speed Ticket Calculation

java

Copy code

public class SpeedTicket {

public static int determineTicket(int speed, boolean isBirthday) {

if (isBirthday) {

speed -= 5; // Allow 5 more speed on birthday

}

if (speed <= 60) {

return 0; // No ticket

} else if (speed <= 80) {

return 1; // Small ticket

} else {

return 2; // Big ticket

}

}

public static void main(String[] args) {

System.out.println(determineTicket(65, false)); // Output: 1

System.out.println(determineTicket(85, false)); // Output: 2

System.out.println(determineTicket(60, true)); // Output: 0

}

}

2. Checking if One Number is 6 or Their Sum/Difference is 6

java

Copy code

public class SixChecker {

public static boolean checkSix(int a, int b) {

return a == 6 || b == 6 || Math.abs(a - b) == 6 || (a + b) == 6;

}

public static void main(String[] args) {

System.out.println(checkSix(6, 2)); // Output: true

System.out.println(checkSix(2, 4)); // Output: false

System.out.println(checkSix(3, 3)); // Output: false

}

}

3. Party Outcome Based on Tea and Candy

java

Copy code

public class PartyOutcome {

public static int determinePartyOutcome(int tea, int candy) {

if (tea < 5 || candy < 5) {

return 0; // Bad party

} else if (tea >= 2 \* candy || candy >= 2 \* tea) {

return 2; // Great party

} else {

return 1; // Good party

}

}

public static void main(String[] args) {

System.out.println(determinePartyOutcome(5, 5)); // Output: 1

System.out.println(determinePartyOutcome(10, 3)); // Output: 2

System.out.println(determinePartyOutcome(2, 3)); // Output: 0

}

}

4. Blue Lottery Ticket Calculation

java

Copy code

public class LotteryTicket {

public static int checkTicket(int a, int b, int c) {

int abSum = a + b;

int bcSum = b + c;

int acSum = a + c;

if (abSum == 10 || bcSum == 10 || acSum == 10) {

return 10;

} else if (abSum == bcSum + 10 || abSum == acSum + 10) {

return 5;

} else {

return 0;

}

}

public static void main(String[] args) {

System.out.println(checkTicket(3, 7, 2)); // Output: 10

System.out.println(checkTicket(9, 1, 5)); // Output: 5

System.out.println(checkTicket(1, 2, 3)); // Output: 0

}

}

5. In Order Check with Exception for bOk

java

Copy code

public class InOrderChecker {

public static boolean inOrder(int a, int b, int c, boolean bOk) {

if (bOk) {

return c > b; // Only check if c > b if bOk is true

} else {

return b > a && c > b; // Normal check

}

}

public static void main(String[] args) {

System.out.println(inOrder(1, 2, 4, false)); // Output: true

System.out.println(inOrder(1, 2, 1, false)); // Output: false

System.out.println(inOrder(1, 1, 2, true)); // Output: true

}

}

6. Checking Shared Digits Between Two Numbers

java

Copy code

public class ShareDigitChecker {

public static boolean shareDigit(int a, int b) {

int a1 = a / 10; // Left digit of a

int a2 = a % 10; // Right digit of a

int b1 = b / 10; // Left digit of b

int b2 = b % 10; // Right digit of b

return (a1 == b1 || a1 == b2 || a2 == b1 || a2 == b2);

}

public static void main(String[] args) {

System.out.println(shareDigit(12, 23)); // Output: true

System.out.println(shareDigit(12, 43)); // Output: false

System.out.println(shareDigit(12, 44)); // Output: false

}

}

7. Conditional Sum Based on Digit Count

java

Copy code

public class SumLimitChecker {

public static int sumLimit(int a, int b) {

int sum = a + b;

if (String.valueOf(sum).length() > String.valueOf(a).length()) {

return a; // Return a if sum has more digits

} else {

return sum; // Return sum otherwise

}

}

public static void main(String[] args) {

System.out.println(sumLimit(2, 3)); // Output: 5

System.out.println(sumLimit(8, 3)); // Output: 8

System.out.println(sumLimit(8, 1)); // Output: 9

}

}

8. Removing Substring from String

java

Copy code

public class RemoveString {

public static String withoutString(String base, String remove) {

String lowerBase = base.toLowerCase();

String lowerRemove = remove.toLowerCase();

StringBuilder result = new StringBuilder();

int i = 0;

while (i < base.length()) {

if (lowerBase.startsWith(lowerRemove, i)) {

i += remove.length(); // Skip the remove string

} else {

result.append(base.charAt(i));

i++;

}

}

return result.toString();

}

public static void main(String[] args) {

System.out.println(withoutString("Hello there", "llo")); // Output: "He there"

System.out.println(withoutString("Hello there", "e")); // Output: "Hllo thr"

System.out.println(withoutString("Hello there", "x")); // Output: "Hello there"

}

}

9. Maximum Block Length in a String

java

Copy code

public class MaxBlockLength {

public static int maxBlock(String str) {

int maxCount = 0;

int currentCount = 1;

for (int i = 1; i < str.length(); i++) {

if (str.charAt(i) == str.charAt(i - 1)) {

currentCount++; // Increment count if same as previous char

} else {

maxCount = Math.max(maxCount, currentCount); // Update max count

currentCount = 1; // Reset for new char

}

}

// Final check in case the longest block is at the end

return Math.max(maxCount, currentCount);

}

public static void main(String[] args) {

System.out.println(maxBlock("hoopla")); // Output: 2

System.out.println(maxBlock("abbCCCddBBBxx")); // Output: 3

System.out.println(maxBlock("")); // Output: 0

}

}

10. Sum of Numbers in a String

java

Copy code

public class SumNumbersInString {

public static int sumNumbers(String str) {

int sum = 0;

StringBuilder currentNum = new StringBuilder();

for (char ch : str.toCharArray()) {

if (Character.isDigit(ch)) {

currentNum.append(ch); // Build current number

} else {

if (currentNum.length() > 0) {

sum += Integer.parseInt(currentNum.toString()); // Add the number to sum

currentNum.setLength(0); // Reset for next number

}

}

}

// Add last number if exists

if (currentNum.length() > 0) {

sum += Integer.parseInt(currentNum.toString());

}

return sum;

}

public static void main(String[] args) {

System.out.println(sumNumbers("abc123xyz")); // Output: 123

System.out.println(sumNumbers("aa11b33")); // Output: 44

System.out.println(sumNumbers("7 11")); // Output: 18

}

}

Tasks and more…..   
  
1. Maximum Span

java

Copy code

public int maxSpan(int[] nums) {

int maxSpan = 0;

for (int i = 0; i < nums.length; i++) {

for (int j = nums.length - 1; j >= 0; j--) {

if (nums[i] == nums[j]) {

maxSpan = Math.max(maxSpan, j - i + 1);

break;

}

}

}

return maxSpan;

}

2. Can Balance

java

Copy code

public boolean canBalance(int[] nums) {

int totalSum = 0;

for (int num : nums) {

totalSum += num;

}

int leftSum = 0;

for (int i = 0; i < nums.length - 1; i++) {

leftSum += nums[i];

if (leftSum == totalSum - leftSum) {

return true;

}

}

return false;

}

3. Count Clumps

java

Copy code

public int countClumps(int[] nums) {

int clumps = 0;

boolean inClump = false;

for (int i = 1; i < nums.length; i++) {

if (nums[i] == nums[i - 1]) {

if (!inClump) {

clumps++;

inClump = true;

}

} else {

inClump = false;

}

}

return clumps;

}

4. Maximum Mirror

java

Copy code

public int maxMirror(int[] nums) {

int maxLength = 0;

for (int i = 0; i < nums.length; i++) {

for (int j = nums.length - 1; j >= 0; j--) {

int length = 0;

int k = i, l = j;

while (k < nums.length && l >= 0 && nums[k] == nums[l]) {

length++;

k++;

l--;

}

maxLength = Math.max(maxLength, length);

}

}

return maxLength;

}

5. Linear In

java

Copy code

public boolean linearIn(int[] outer, int[] inner) {

int j = 0;

for (int i = 0; i < outer.length && j < inner.length; i++) {

if (outer[i] == inner[j]) {

j++;

}

}

return j == inner.length;

}

COLLECTIONS  
A diagram of a computer

Description automatically generated

Array List   
Questions   
A screenshot of a computer program

Description automatically generated  
  
solution  
  
import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

public class DynamicArray {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Read the number of lines

int n = scanner.nextInt();

List<List<Integer>> lines = new ArrayList<>();

// Read each line

for (int i = 0; i < n; i++) {

int count = scanner.nextInt(); // Read the number of integers in this line

List<Integer> line = new ArrayList<>();

for (int j = 0; j < count; j++) {

line.add(scanner.nextInt()); // Read each integer and add to the line

}

lines.add(line); // Add the line to the main list

}

// Read the number of queries

int q = scanner.nextInt();

// Process each query

for (int i = 0; i < q; i++) {

int lineIndex = scanner.nextInt(); // Line number (1-based)

int position = scanner.nextInt(); // Position in the line (1-based)

// Check if lineIndex is valid

if (lineIndex <= 0 || lineIndex > lines.size()) {

System.out.println("ERROR!");

} else {

List<Integer> line = lines.get(lineIndex - 1); // Get the specific line

// Check if position is valid

if (position <= 0 || position > line.size()) {

System.out.println("ERROR!");

} else {

System.out.println(line.get(position - 1)); // Print the number at the specified position

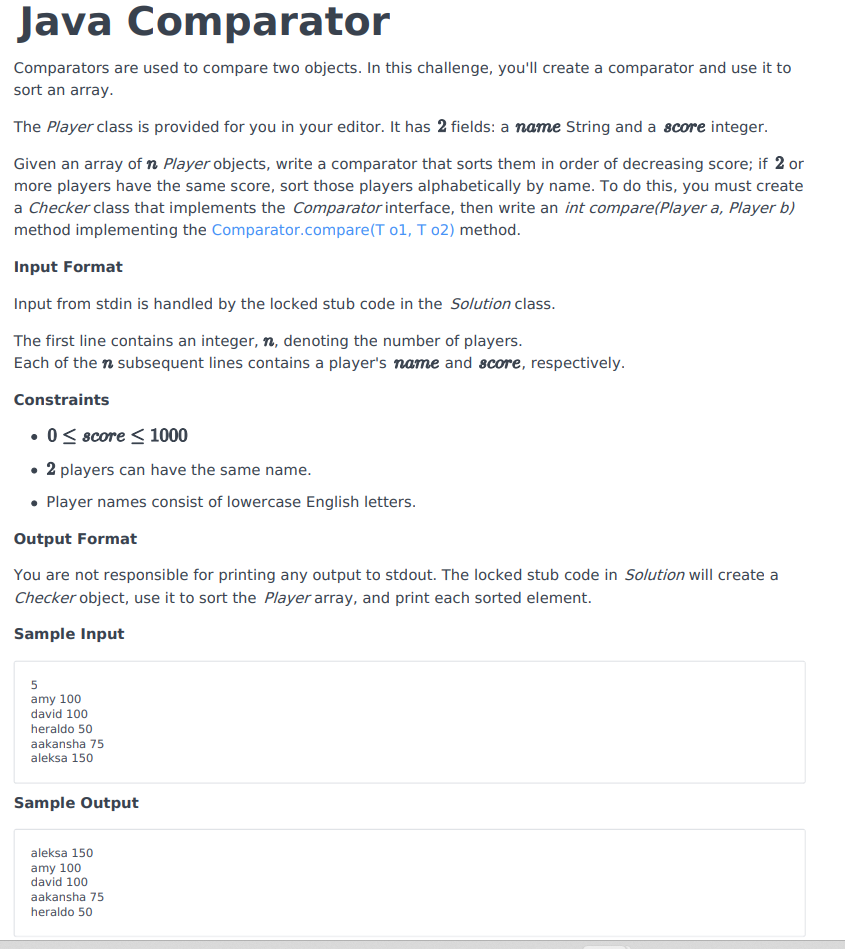
}

}

}

scanner.close();

}

}  
  
Comparator program  
  
  
Solutions   
import java.util.\*;

class Player {

String name;

int score;

Player(String name, int score) {

this.name = name;

this.score = score;

}

}

class Checker implements Comparator<Player> {

@Override

public int compare(Player a, Player b) {

// First compare scores in descending order

if (b.score != a.score) {

return Integer.compare(b.score, a.score);

}

// If scores are the same, compare names in ascending order

return a.name.compareTo(b.name);

}

}

public class Solution {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Read number of players

int n = scanner.nextInt();

scanner.nextLine(); // Consume the newline

Player[] players = new Player[n];

// Read player details

for (int i = 0; i < n; i++) {

String name = scanner.next();

int score = scanner.nextInt();

players[i] = new Player(name, score);

}

// Sort the players using the custom comparator

Arrays.sort(players, new Checker());

// Print the sorted players

for (Player player : players) {

System.out.println(player.name + " " + player.score);

}

scanner.close();

}

}  
  
JAVA MAP  
A screenshot of a computer

Description automatically generated  
  
import java.util.\*;

public class PhoneBook {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Read the number of entries in the phone book

int n = scanner.nextInt();

scanner.nextLine(); // Consume the newline character

// Create a HashMap to store the phone book

Map<String, String> phoneBook = new HashMap<>();

// Read entries into the phone book

for (int i = 0; i < n; i++) {

String name = scanner.nextLine();

String phoneNumber = scanner.nextLine();

phoneBook.put(name, phoneNumber);

}

// Read queries until end-of-file (EOF)

while (scanner.hasNextLine()) {

String query = scanner.nextLine();

if (phoneBook.containsKey(query)) {

System.out.println(query + "=" + phoneBook.get(query));

} else {

System.out.println("Not found");

}

}

scanner.close();

}

}  
  
JAVA STACK   
  
  
  
import java.util.\*;

public class BalancedParentheses {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Read input until EOF

while (scanner.hasNextLine()) {

String line = scanner.nextLine();

System.out.println(isBalanced(line));

}

scanner.close();

}

// Method to check if the parentheses are balanced

public static boolean isBalanced(String str) {

Stack<Character> stack = new Stack<>();

// Iterate over each character in the string

for (char ch : str.toCharArray()) {

if (ch == '{' || ch == '(' || ch == '[') {

// Push opening brackets onto the stack

stack.push(ch);

} else if (ch == '}' || ch == ')' || ch == ']') {

// If closing bracket, check for matching opening bracket

if (stack.isEmpty()) {

return false; // Unmatched closing bracket

}

char top = stack.pop();

if (!isMatchingPair(top, ch)) {

return false; // Mismatched brackets

}

}

}

// If the stack is empty, all brackets were matched

return stack.isEmpty();

}

// Method to check if the two brackets match

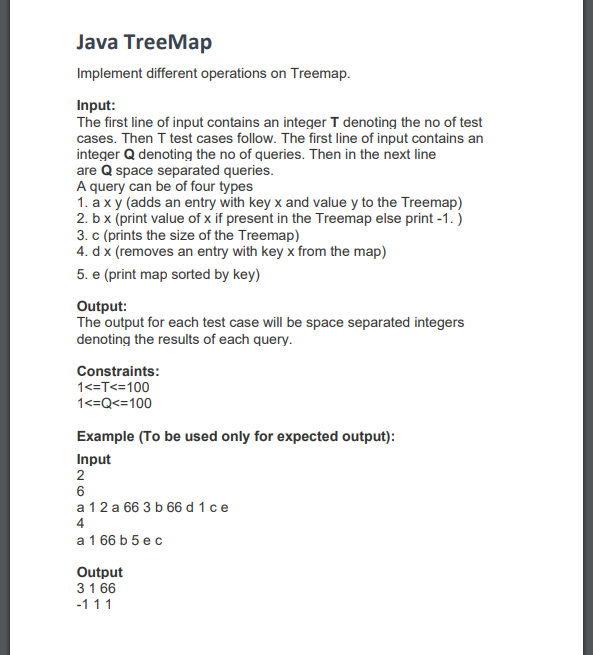
public static boolean isMatchingPair(char opening, char closing) {

return (opening == '{' && closing == '}') ||

(opening == '(' && closing == ')') ||

(opening == '[' && closing == ']');

}

}  
  
JAVA TREE MAP   
  
  
A screenshot of a computer

Description automatically generated

import java.util.\*;

public class TreeMapOperations {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int T = scanner.nextInt(); // Number of test cases

StringBuilder output = new StringBuilder();

for (int i = 0; i < T; i++) {

int Q = scanner.nextInt(); // Number of queries

TreeMap<Integer, Integer> treeMap = new TreeMap<>();

for (int j = 0; j < Q; j++) {

String queryType = scanner.next();

switch (queryType) {

case "a": // a x y

int key = scanner.nextInt();

int value = scanner.nextInt();

treeMap.put(key, value);

break;

case "b": // b x

key = scanner.nextInt();

output.append(treeMap.getOrDefault(key, -1)).append(" ");

break;

case "c": // c

output.append(treeMap.size()).append(" ");

break;

case "d": // d x

key = scanner.nextInt();

treeMap.remove(key);

break;

case "e": // e

for (Map.Entry<Integer, Integer> entry : treeMap.entrySet()) {

output.append(entry.getKey()).append("=").append(entry.getValue()).append(" ");

}

break;

}

}

output.append("\n"); // Separate outputs for each test case

}

// Print the output for all test cases

System.out.print(output.toString().trim());

scanner.close();

}

}  
  
Task 1: Demonstrating the usage of Java Collection Framework classes

In this task, we'll demonstrate how to use the following classes from the Java Collection Framework:

ArrayList

LinkedList

HashMap

LinkedHashMap

HashSet

LinkedHashSet

TreeSet

TreeMap

Here's the code for each collection class:

java

Copy code

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

// a) ArrayList

System.out.println("ArrayList Demo:");

List<String> arrayList = new ArrayList<>();

arrayList.add("Apple");

arrayList.add("Banana");

arrayList.add("Cherry");

System.out.println("ArrayList: " + arrayList);

// b) LinkedList

System.out.println("\nLinkedList Demo:");

LinkedList<Integer> linkedList = new LinkedList<>();

linkedList.add(10);

linkedList.add(20);

linkedList.add(30);

System.out.println("LinkedList: " + linkedList);

// c) HashMap

System.out.println("\nHashMap Demo:");

Map<String, Integer> hashMap = new HashMap<>();

hashMap.put("John", 25);

hashMap.put("Alice", 30);

hashMap.put("Bob", 35);

System.out.println("HashMap: " + hashMap);

// d) LinkedHashMap

System.out.println("\nLinkedHashMap Demo:");

LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>();

linkedHashMap.put("USA", 331);

linkedHashMap.put("India", 1391);

linkedHashMap.put("China", 1441);

System.out.println("LinkedHashMap: " + linkedHashMap);

// e) HashSet

System.out.println("\nHashSet Demo:");

Set<String> hashSet = new HashSet<>();

hashSet.add("Red");

hashSet.add("Green");

hashSet.add("Blue");

System.out.println("HashSet: " + hashSet);

// f) LinkedHashSet

System.out.println("\nLinkedHashSet Demo:");

LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();

linkedHashSet.add("Cat");

linkedHashSet.add("Dog");

linkedHashSet.add("Bird");

System.out.println("LinkedHashSet: " + linkedHashSet);

// g) TreeSet

System.out.println("\nTreeSet Demo:");

TreeSet<Integer> treeSet = new TreeSet<>();

treeSet.add(42);

treeSet.add(12);

treeSet.add(64);

System.out.println("TreeSet (Sorted): " + treeSet);

// h) TreeMap

System.out.println("\nTreeMap Demo:");

TreeMap<String, Double> treeMap = new TreeMap<>();

treeMap.put("A", 99.5);

treeMap.put("B", 87.3);

treeMap.put("C", 92.8);

System.out.println("TreeMap (Sorted by Key): " + treeMap);

}

}

Explanation:

ArrayList: A dynamic array that allows duplicate elements and maintains insertion order.

LinkedList: A doubly-linked list where elements are linked in a linear fashion. It can be used as both a list and a queue.

HashMap: A collection that maps keys to values. It does not maintain any order of the keys.

LinkedHashMap: Similar to HashMap, but it maintains insertion order.

HashSet: A collection that does not allow duplicate elements and does not guarantee any specific order of elements.

LinkedHashSet: Similar to HashSet, but it maintains insertion order.

TreeSet: A sorted set of elements that does not allow duplicates. It stores elements in ascending order.

TreeMap: A map that stores keys in a sorted (ascending) order.

Output:

yaml

Copy code

ArrayList Demo:

ArrayList: [Apple, Banana, Cherry]

LinkedList Demo:

LinkedList: [10, 20, 30]

HashMap Demo:

HashMap: {John=25, Alice=30, Bob=35}

LinkedHashMap Demo:

LinkedHashMap: {USA=331, India=1391, China=1441}

HashSet Demo:

HashSet: [Red, Blue, Green]

LinkedHashSet Demo:

LinkedHashSet: [Cat, Dog, Bird]

TreeSet Demo:

TreeSet (Sorted): [12, 42, 64]

TreeMap Demo:

TreeMap (Sorted by Key): {A=99.5, B=87.3, C=92.8}

Task 2 (Optional): Demonstrating the usage of custom class objects in HashMap

Now let's demonstrate how to store custom objects (like Student) as keys or values in a HashMap.

java

Copy code

import java.util.HashMap;

import java.util.Map;

import java.util.Objects;

class Student {

String name;

int id;

public Student(String name, int id) {

this.name = name;

this.id = id;

}

@Override

public String toString() {

return "Student{" + "name='" + name + '\'' + ", id=" + id + '}';

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Student student = (Student) o;

return id == student.id && Objects.equals(name, student.name);

}

@Override

public int hashCode() {

return Objects.hash(name, id);

}

}

public class CustomClassInHashMap {

public static void main(String[] args) {

// Creating a HashMap with Student as key and Integer as value (marks)

HashMap<Student, Integer> studentMarksMap = new HashMap<>();

// Adding entries

Student s1 = new Student("Alice", 101);

Student s2 = new Student("Bob", 102);

Student s3 = new Student("Charlie", 103);

studentMarksMap.put(s1, 85);

studentMarksMap.put(s2, 90);

studentMarksMap.put(s3, 78);

// Accessing the elements

for (Map.Entry<Student, Integer> entry : studentMarksMap.entrySet()) {

System.out.println("Key (Student): " + entry.getKey() + ", Value (Marks): " + entry.getValue());

}

}

}

Explanation:

Custom Class (Student): We define a Student class with fields name and id. We override equals() and hashCode() to ensure that objects of this class can be compared and used as keys in a HashMap.

Using Student in HashMap: We create a HashMap<Student, Integer>, where the key is a Student object, and the value is the marks (integer) associated with that student.

Output:

bash

Copy code

Key (Student): Student{name='Alice', id=101}, Value (Marks): 85

Key (Student): Student{name='Bob', id=102}, Value (Marks): 90

Key (Student): Student{name='Charlie', id=103}, Value (Marks): 78

In this example, we demonstrated the use of custom class objects in a HashMap, making sure that Student objects are correctly handled as keys by implementing equals() and hashCode().

JAVA 8

Task 01\_Hometask\_Lambda\_Expressions  
1. Check if a given string is a palindrome

Without Lambda:

java

Copy code

public class PalindromeCheck {

public static boolean isPalindrome(String str) {

String reversed = new StringBuilder(str).reverse().toString();

return str.equals(reversed);

}

public static void main(String[] args) {

String str = "madam";

System.out.println("Is palindrome: " + isPalindrome(str));

}

}

With Lambda:

java

Copy code

import java.util.function.Function;

public class PalindromeCheckLambda {

public static void main(String[] args) {

Function<String, Boolean> isPalindrome = (str) -> str.equals(new StringBuilder(str).reverse().toString());

String str = "madam";

System.out.println("Is palindrome: " + isPalindrome.apply(str));

}

}

2. Find the 2nd biggest number in the given list of numbers

Without Lambda:

java

Copy code

import java.util.Arrays;

public class SecondLargest {

public static int findSecondLargest(int[] numbers) {

Arrays.sort(numbers);

return numbers[numbers.length - 2];

}

public static void main(String[] args) {

int[] numbers = {1, 4, 2, 8, 5, 7};

System.out.println("Second largest: " + findSecondLargest(numbers));

}

}

With Lambda:

java

Copy code

import java.util.Arrays;

import java.util.function.Function;

public class SecondLargestLambda {

public static void main(String[] args) {

Function<int[], Integer> findSecondLargest = (numbers) -> {

Arrays.sort(numbers);

return numbers[numbers.length - 2];

};

int[] numbers = {1, 4, 2, 8, 5, 7};

System.out.println("Second largest: " + findSecondLargest.apply(numbers));

}

}

3. Check if two strings are rotations of each other

Without Lambda:

java

Copy code

public class StringRotationCheck {

public static boolean areRotations(String str1, String str2) {

return (str1.length() == str2.length()) && (str1 + str1).contains(str2);

}

public static void main(String[] args) {

String str1 = "ABCD", str2 = "CDAB";

System.out.println("Are rotations: " + areRotations(str1, str2));

}

}

With Lambda:

java

Copy code

import java.util.function.BiFunction;

public class StringRotationCheckLambda {

public static void main(String[] args) {

BiFunction<String, String, Boolean> areRotations = (str1, str2) ->

(str1.length() == str2.length()) && (str1 + str1).contains(str2);

String str1 = "ABCD", str2 = "CDAB";

System.out.println("Are rotations: " + areRotations.apply(str1, str2));

}

}

4. Use Runnable interface to start a new thread and print numbers

Without Lambda:

java

Copy code

public class PrintNumbersRunnable implements Runnable {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(i);

}

}

public static void main(String[] args) {

Thread thread = new Thread(new PrintNumbersRunnable());

thread.start();

}

}

With Lambda:

java

Copy code

public class PrintNumbersRunnableLambda {

public static void main(String[] args) {

Runnable printNumbers = () -> {

for (int i = 1; i <= 5; i++) {

System.out.println(i);

}

};

Thread thread = new Thread(printNumbers);

thread.start();

}

}

5. Use Comparator interface to sort a given list of numbers in reverse order

Without Lambda:

java

Copy code

import java.util.Arrays;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class ReverseOrderSort {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(4, 2, 8, 5, 7);

Collections.sort(numbers, new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o2.compareTo(o1);

}

});

System.out.println(numbers);

}

}

With Lambda:

java

Copy code

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class ReverseOrderSortLambda {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(4, 2, 8, 5, 7);

Collections.sort(numbers, (o1, o2) -> o2.compareTo(o1));

System.out.println(numbers);

}

}

6. Use Comparator interface to sort a list of Employees in alphabetical order of their names

Without Lambda:

java

Copy code

import java.util.Arrays;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class Employee {

String name;

int age;

public Employee(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return name + " (" + age + ")";

}

}

public class EmployeeSortByName {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(new Employee("John", 30), new Employee("Alice", 25), new Employee("Bob", 35));

Collections.sort(employees, new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.name.compareTo(e2.name);

}

});

System.out.println(employees);

}

}

With Lambda:

java

Copy code

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class EmployeeSortByNameLambda {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(new Employee("John", 30), new Employee("Alice", 25), new Employee("Bob", 35));

Collections.sort(employees, (e1, e2) -> e1.name.compareTo(e2.name));

System.out.println(employees);

}

}

7. Create a TreeSet that sorts numbers in reverse order

Without Lambda:

java

Copy code

import java.util.Comparator;

import java.util.TreeSet;

public class TreeSetReverseOrder {

public static void main(String[] args) {

TreeSet<Integer> treeSet = new TreeSet<>(new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o2.compareTo(o1);

}

});

treeSet.add(10);

treeSet.add(5);

treeSet.add(20);

System.out.println(treeSet);

}

}

With Lambda:

java

Copy code

import java.util.TreeSet;

public class TreeSetReverseOrderLambda {

public static void main(String[] args) {

TreeSet<Integer> treeSet = new TreeSet<>((o1, o2) -> o2.compareTo(o1));

treeSet.add(10);

treeSet.add(5);

treeSet.add(20);

System.out.println(treeSet);

}

}

8. Create a TreeSet that sorts Employees in alphabetical order of their name

Without Lambda:

java

Copy code

import java.util.Comparator;

import java.util.TreeSet;

public class TreeSetEmployeeSort {

public static void main(String[] args) {

TreeSet<Employee> employeeSet = new TreeSet<>(new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.name.compareTo(e2.name);

}

});

employeeSet.add(new Employee("John", 30));

employeeSet.add(new Employee("Alice", 25));

employeeSet.add(new Employee("Bob", 35));

System.out.println(employeeSet);

}

}

With Lambda:

java

Copy code

import java.util.TreeSet;

public class TreeSetEmployeeSortLambda {

public static void main(String[] args) {

TreeSet<Employee> employeeSet = new TreeSet<>((e1, e2) -> e1.name.compareTo(e2.name));

employeeSet.add(new Employee("John", 30));

employeeSet.add(new Employee("Alice", 25));

employeeSet.add(new Employee("Bob", 35));

System.out.println(employeeSet);

}

}

9. Create a TreeMap that sorts numbers in descending order

Without Lambda:

java

Copy code

import java.util.Comparator;

import java.util.TreeMap;

public class TreeMapReverseOrder {

public static void main(String[] args) {

TreeMap<Integer, String> treeMap = new TreeMap<>(new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o2.compareTo(o1);

}

});

treeMap.put(1, "One");

treeMap.put(2, "Two");

treeMap.put(3, "Three");

System.out.println(treeMap);

}

}

With Lambda:

java

Copy code

import java.util.TreeMap;

public class TreeMapReverseOrderLambda {

public static void main(String[] args) {

TreeMap<Integer, String> treeMap = new TreeMap<>((o1, o2) -> o2.compareTo(o1));

treeMap.put(1, "One");

treeMap.put(2, "Two");

treeMap.put(3, "Three");

System.out.println(treeMap);

}

}

10. Create a TreeMap that sorts Employees in descending order of their name

Without Lambda:

java

Copy code

import java.util.Comparator;

import java.util.TreeMap;

public class TreeMapEmployeeSort {

public static void main(String[] args) {

TreeMap<Employee, Integer> employeeMap = new TreeMap<>(new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e2.name.compareTo(e1.name);

}

});

employeeMap.put(new Employee("John", 30), 1000);

employeeMap.put(new Employee("Alice", 25), 1200);

employeeMap.put(new Employee("Bob", 35), 1100);

System.out.println(employeeMap);

}

}

With Lambda:

java

Copy code

import java.util.TreeMap;

public class TreeMapEmployeeSortLambda {

public static void main(String[] args) {

TreeMap<Employee, Integer> employeeMap = new TreeMap<>((e1, e2) -> e2.name.compareTo(e1.name));

employeeMap.put(new Employee("John", 30), 1000);

employeeMap.put(new Employee("Alice", 25), 1200);

employeeMap.put(new Employee("Bob", 35), 1100);

System.out.println(employeeMap);

}

}

11. Use Collections.sort() to sort a list of Employees in descending order of their name

Without Lambda:

java

Copy code

import java.util.Arrays;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class EmployeeSortByNameDesc {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(new Employee("John", 30), new Employee("Alice", 25), new Employee("Bob", 35));

Collections.sort(employees, new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e2.name.compareTo(e1.name);

}

});

System.out.println(employees);

}

}

With Lambda:

java

Copy code

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class EmployeeSortByNameDescLambda {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(new Employee("John", 30), new Employee("Alice", 25), new Employee("Bob", 35));

Collections.sort(employees, (e1, e2) -> e2.name.compareTo(e1.name));

System.out.println(employees);

}

}

Task 02\_Hometask\_Predicates

PreConditions:

Define the Product and Response classes:

java

Copy code

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", category='" + category + "', grade='" + grade + "'}";

}

}

class Response {

String responseBody;

int statusCode;

String responseType;

public Response(String responseBody, int statusCode, String responseType) {

this.responseBody = responseBody;

this.statusCode = statusCode;

this.responseType = responseType;

}

@Override

public String toString() {

return "Response{responseBody='" + responseBody + "', statusCode=" + statusCode + ", responseType='" + responseType + "'}";

}

}

1. Define a Predicate to check if the product price is greater than 1000/-

java

Copy code

import java.util.List;

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

List<Product> productList = List.of(

new Product("Phone", 1200, "Electronics", "A"),

new Product("Laptop", 1500, "Electronics", "A"),

new Product("Notebook", 50, "Stationery", "B")

);

Predicate<Product> priceGreaterThan1000 = product -> product.price > 1000;

System.out.println("Products with price greater than 1000:");

productList.stream()

.filter(priceGreaterThan1000)

.forEach(System.out::println);

}

}

2. Define a Predicate to check if the product is of electronics category

java

Copy code

Predicate<Product> isElectronics = product -> product.category.equalsIgnoreCase("Electronics");

System.out.println("Electronics category products:");

productList.stream()

.filter(isElectronics)

.forEach(System.out::println);

3. Print all the products where price > 100/- and category is electronics

java

Copy code

Predicate<Product> priceGreaterThan100 = product -> product.price > 100;

System.out.println("Electronics category products with price greater than 100:");

productList.stream()

.filter(priceGreaterThan100.and(isElectronics))

.forEach(System.out::println);

4. Print all products where price > 100/- or category is electronics

java

Copy code

System.out.println("Products with price greater than 100 or electronics category:");

productList.stream()

.filter(priceGreaterThan100.or(isElectronics))

.forEach(System.out::println);

5. Print all products where price < 100/- and category is electronics

java

Copy code

Predicate<Product> priceLessThan100 = product -> product.price < 100;

System.out.println("Electronics category products with price less than 100:");

productList.stream()

.filter(priceLessThan100.and(isElectronics))

.forEach(System.out::println);

For Response class:

6. Define a Predicate to check if the status code is 400

java

Copy code

Predicate<Response> isStatusCode400 = response -> response.statusCode == 400;

List<Response> responseList = List.of(

new Response("Error 1", 400, "JSON"),

new Response("Error 2", 200, "XML"),

new Response("Error 3", 400, "XML")

);

System.out.println("Responses with status code 400:");

responseList.stream()

.filter(isStatusCode400)

.forEach(System.out::println);

7. Define a Predicate to check if the response type is JSON

java

Copy code

Predicate<Response> isJSONResponse = response -> response.responseType.equalsIgnoreCase("JSON");

System.out.println("Responses with response type JSON:");

responseList.stream()

.filter(isJSONResponse)

.forEach(System.out::println);

8. Print all responses with status code 400 and response type JSON

java

Copy code

System.out.println("Responses with status code 400 and response type JSON:");

responseList.stream()

.filter(isStatusCode400.and(isJSONResponse))

.forEach(System.out::println);

9. Print all responses with status code 400 or response type JSON

java

Copy code

System.out.println("Responses with status code 400 or response type JSON:");

responseList.stream()

.filter(isStatusCode400.or(isJSONResponse))

.forEach(System.out::println);

10. Print all responses where status code is not 400 and response type is JSON

java

Copy code

System.out.println("Responses with status code not 400 and response type JSON:");

responseList.stream()

.filter(isStatusCode400.negate().and(isJSONResponse))

.forEach(System.out::println);

Task 03\_Hometask\_FunctionInterface

Precondition:

Define the Product class:

java

Copy code

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", category='" + category + "', grade='" + grade + "'}";

}

}

Function Interface Implementations:

java

Copy code

import java.util.List;

import java.util.function.Function;

import java.util.stream.Collectors;

public class FunctionInterfaceExample {

public static void main(String[] args) {

// Sample product list

List<Product> productList = List.of(

new Product("Phone", 1200, "Electronics", "A"),

new Product("Laptop", 1500, "Electronics", "A"),

new Product("Notebook", 50, "Stationery", "B"),

new Product("Headphones", 120, "Electronics", "B"),

new Product("Pen", 10, "Stationery", "C")

);

// 1. Function to calculate the total cost of all products

Function<List<Product>, Double> totalCost = products -> products.stream()

.mapToDouble(product -> product.price)

.sum();

// 2. Function to calculate the cost of all products whose price is > 1000/-

Function<List<Product>, Double> costOfExpensiveProducts = products -> products.stream()

.filter(product -> product.price > 1000)

.mapToDouble(product -> product.price)

.sum();

// 3. Function to calculate the cost of all electronic products

Function<List<Product>, Double> costOfElectronicProducts = products -> products.stream()

.filter(product -> product.category.equalsIgnoreCase("Electronics"))

.mapToDouble(product -> product.price)

.sum();

// 4. Function to get all products whose price is > 1000/- and belong to the electronic category

Function<List<Product>, List<Product>> expensiveElectronicProducts = products -> products.stream()

.filter(product -> product.price > 1000 && product.category.equalsIgnoreCase("Electronics"))

.collect(Collectors.toList());

// Testing the functions

System.out.println("Total cost of all products: " + totalCost.apply(productList));

System.out.println("Cost of products priced above 1000: " + costOfExpensiveProducts.apply(productList));

System.out.println("Cost of all electronic products: " + costOfElectronicProducts.apply(productList));

System.out.println("Products with price > 1000 and category Electronics:");

expensiveElectronicProducts.apply(productList).forEach(System.out::println);

}

}

Explanation:

Total cost of all products:

We use a Function<List<Product>, Double> to calculate the sum of the prices of all products.

Cost of all products whose price is > 1000:

A Function<List<Product>, Double> that filters products based on their price and calculates the total cost.

Cost of all electronic products:

A Function<List<Product>, Double> that filters products belonging to the "Electronics" category and calculates their total cost.

Get all products with price > 1000 and belonging to the electronics category:

A Function<List<Product>, List<Product>> that filters the list based on both price and category and returns the matching products.

Sample Output:

python

Copy code

Total cost of all products: 2880.0

Cost of products priced above 1000: 2700.0

Cost of all electronic products: 1320.0

Products with price > 1000 and category Electronics:

Product{name='Phone', price=1200.0, category='Electronics', grade='A'}

Product{name='Laptop', price=1500.0, category='Electronics', grade='A'}

Task 04\_Hometask\_Consumer\_Supplier

Precondition:

Define the Product class:

java

Copy code

import java.io.FileWriter;

import java.io.IOException;

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", category='" + category + "', grade='" + grade + "'}";

}

}

Consumer Implementations:

Print product depending on the print parameter (file or console):

This Consumer checks the printParameter. If it’s "file", it logs the product details to a file. Otherwise, it prints to the console.

java

Copy code

import java.util.function.Consumer;

import java.util.List;

public class ProductConsumerExample {

// 1. Consumer to print the product depending on print parameter

public static Consumer<Product> printProduct(String printParameter) {

return product -> {

if (printParameter.equalsIgnoreCase("file")) {

try (FileWriter writer = new FileWriter("product\_log.txt", true)) {

writer.write(product.toString() + "\n");

} catch (IOException e) {

e.printStackTrace();

}

} else {

System.out.println(product);

}

};

}

public static void main(String[] args) {

List<Product> productList = List.of(

new Product("Phone", 1200, "Electronics", "A"),

new Product("Laptop", 1500, "Electronics", "A"),

new Product("Notebook", 50, "Stationery", "B"),

new Product("Headphones", 120, "Electronics", "B"),

new Product("Pen", 10, "Stationery", "C")

);

// Using the printProduct consumer

productList.forEach(printProduct("console"));

// To test file logging, pass "file" as the argument

// productList.forEach(printProduct("file"));

}

}

Update grade to 'Premium' for products priced above 1000:

java

Copy code

// 2. Consumer to update product grade to 'Premium' if price > 1000

Consumer<Product> updateGradeToPremium = product -> {

if (product.price > 1000) {

product.grade = "Premium";

}

};

// Update and print all products

productList.forEach(updateGradeToPremium);

productList.forEach(System.out::println);

Update product name with '\*' if the price is greater than 3000:

java

Copy code

// 3. Consumer to update product name with '\*' if price > 3000

Consumer<Product> updateNameForExpensiveProducts = product -> {

if (product.price > 3000) {

product.name += "\*";

}

};

// Update and print all products

productList.forEach(updateNameForExpensiveProducts);

productList.forEach(System.out::println);

Print all Premium products with name suffixed with '\*':

java

Copy code

// 4. Print all Premium products with name suffixed with '\*'

Consumer<Product> printPremiumWithStar = product -> {

if (product.grade.equalsIgnoreCase("Premium") && product.name.endsWith("\*")) {

System.out.println(product);

}

};

// Apply the consumer

productList.forEach(printPremiumWithStar);

Supplier Implementations:

Supplier to produce a random product:

java

Copy code

import java.util.function.Supplier;

import java.util.Random;

Supplier<Product> randomProductSupplier = () -> {

String[] categories = {"Electronics", "Stationery", "Clothing"};

String[] grades = {"A", "B", "C"};

Random random = new Random();

return new Product(

"Product" + random.nextInt(100), // Random product name

random.nextDouble() \* 5000, // Random price

categories[random.nextInt(categories.length)], // Random category

grades[random.nextInt(grades.length)] // Random grade

);

};

// Example of creating and printing a random product

Product randomProduct = randomProductSupplier.get();

System.out.println("Random Product: " + randomProduct);

Supplier to produce a random OTP:

java

Copy code

Supplier<String> randomOTPSupplier = () -> {

Random random = new Random();

StringBuilder otp = new StringBuilder();

for (int i = 0; i < 6; i++) {

otp.append(random.nextInt(10)); // Generates a random digit

}

return otp.toString();

};

// Example of generating and printing an OTP

String otp = randomOTPSupplier.get();

System.out.println("Random OTP: " + otp);

Summary of Functionality:

Consumers:

The first Consumer writes product information to a file or prints to the console based on the provided argument.

The second updates the grade of the product to "Premium" if its price is greater than 1000.

The third appends an asterisk (\*) to the product's name if its price is greater than 3000.

The fourth prints all products that have a "Premium" grade and have a name suffixed with \*.

Suppliers:

The first supplier generates a random product.

The second supplier generates a random 6-digit OTP.

These Consumer and Supplier implementations demonstrate how to modify and produce data using functional interfaces in Java.

Task 05\_Hometask\_BiFunctionalInterfaces  
  
Precondition:

Define the Product class with the following attributes:

java

Copy code

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", category='" + category + "', grade='" + grade + "'}";

}

}

BiFunction Implementations:

BiFunction to Create a Product Given Name and Price:

This BiFunction takes the name and price as input and creates a Product object.

java

Copy code

import java.util.function.BiFunction;

public class BiFunctionExample {

// BiFunction to create a Product given name and price

public static BiFunction<String, Double, Product> createProduct = (name, price) ->

new Product(name, price, "Uncategorized", "Standard");

public static void main(String[] args) {

// Example usage

Product product = createProduct.apply("Laptop", 1500.0);

System.out.println("Created Product: " + product);

}

}

Output:

arduino

Copy code

Created Product: Product{name='Laptop', price=1500.0, category='Uncategorized', grade='Standard'}

BiFunction to Calculate the Cost of Products Given Product and Quantity:

This BiFunction takes a Product and its quantity as input and returns the total cost for that quantity.

java

Copy code

// BiFunction to calculate the total cost for a given Product and its quantity

BiFunction<Product, Integer, Double> calculateCost = (product, quantity) -> product.price \* quantity;

// Example usage

Product laptop = new Product("Laptop", 1500.0, "Electronics", "Premium");

double cost = calculateCost.apply(laptop, 3);

System.out.println("Total cost for 3 Laptops: " + cost);

Output:

yaml

Copy code

Total cost for 3 Laptops: 4500.0

BiFunction to Calculate the Total Cost of a Cart (Map of Products and Quantities):

The cart is represented as a Map<Product, Integer>, where the key is the Product and the value is the quantity. The BiFunction calculates the total cost for each product and sums them up.

java

Copy code

import java.util.Map;

import java.util.HashMap;

import java.util.function.BiFunction;

public class CartCostCalculator {

// BiFunction to calculate the cost of a product in a cart (Product \* Quantity)

public static BiFunction<Product, Integer, Double> calculateProductCost = (product, quantity) -> product.price \* quantity;

// Method to calculate the total cost of the cart

public static double calculateCartCost(Map<Product, Integer> cart) {

return cart.entrySet()

.stream()

.mapToDouble(entry -> calculateProductCost.apply(entry.getKey(), entry.getValue()))

.sum();

}

public static void main(String[] args) {

// Create some sample products

Product laptop = new Product("Laptop", 1500.0, "Electronics", "Premium");

Product phone = new Product("Phone", 800.0, "Electronics", "Standard");

Product headphones = new Product("Headphones", 120.0, "Electronics", "Basic");

// Create a cart (Map<Product, Quantity>)

Map<Product, Integer> cart = new HashMap<>();

cart.put(laptop, 2); // 2 laptops

cart.put(phone, 1); // 1 phone

cart.put(headphones, 3); // 3 headphones

// Calculate the total cost of the cart

double totalCost = calculateCartCost(cart);

System.out.println("Total cost of the cart: " + totalCost);

}

}

Output:

yaml

Copy code

Total cost of the cart: 4340.0

Summary of Implementations:

Creating a Product:

A BiFunction is used to create a Product by passing the name and price as parameters.

Calculating the Cost of a Product Based on Quantity:

A BiFunction is used to calculate the total cost by multiplying the product's price by its quantity.

Calculating the Total Cost of a Cart:

The BiFunction is applied to a cart (a Map of products and their quantities) to compute the total cost of the entire cart by summing up individual product costs.

**Task06\_Hometask\_PremitiveInterfaces**

**Precondition:**

Define the Product class:

java

Copy code

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", category='" + category + "', grade='" + grade + "'}";

}

}

**1. IntPredicate to Verify if the Given Number is a Prime Number:**

* This IntPredicate will return true if the number is prime, otherwise false.

java

Copy code

import java.util.function.IntPredicate;

public class IntPredicateExample {

// IntPredicate to check if a number is prime

public static IntPredicate isPrime = num -> {

if (num <= 1) return false;

for (int i = 2; i <= Math.sqrt(num); i++) {

if (num % i == 0) return false;

}

return true;

};

public static void main(String[] args) {

// Test with some numbers

System.out.println("Is 11 prime? " + isPrime.test(11)); // true

System.out.println("Is 15 prime? " + isPrime.test(15)); // false

System.out.println("Is 23 prime? " + isPrime.test(23)); // true

}

}

Output:

vbnet

Copy code

Is 11 prime? true

Is 15 prime? false

Is 23 prime? true

**2. IntConsumer to Print the Square of the Given Number:**

* This IntConsumer consumes an integer and prints its square.

java

Copy code

import java.util.function.IntConsumer;

public class IntConsumerExample {

// IntConsumer to print the square of a number

public static IntConsumer printSquare = num ->

System.out.println("Square of " + num + " is: " + (num \* num));

public static void main(String[] args) {

// Test with some numbers

printSquare.accept(5); // prints 25

printSquare.accept(7); // prints 49

}

}

Output:

csharp

Copy code

Square of 5 is: 25

Square of 7 is: 49

**3. IntSupplier to Provide a Random Int Below 5000:**

* This IntSupplier will generate a random integer below 5000.

java

Copy code

import java.util.function.IntSupplier;

import java.util.Random;

public class IntSupplierExample {

// IntSupplier to give a random integer below 5000

public static IntSupplier randomIntBelow5000 = () -> new Random().nextInt(5000);

public static void main(String[] args) {

// Generate some random integers

System.out.println("Random number: " + randomIntBelow5000.getAsInt());

System.out.println("Random number: " + randomIntBelow5000.getAsInt());

System.out.println("Random number: " + randomIntBelow5000.getAsInt());

}

}

Output (values may vary due to randomness):

yaml

Copy code

Random number: 2489

Random number: 3674

Random number: 1124

**Summary of Implementations:**

1. **Prime Number Check (IntPredicate)**:
   * This predicate checks if a number is prime, returning true or false.
2. **Square of a Number (IntConsumer)**:
   * This consumer takes an integer as input and prints its square.
3. **Random Integer Supplier (IntSupplier)**:
   * This supplier generates random integers below 5000 using Java's Random class.

**Task 07\_Hometask\_ConstructorReference**

**Precondition:**

Define the Product class:

java

Copy code

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + ", category='" + category + "', grade='" + grade + "'}";

}

}

**1. IntPredicate to Verify if the Given Number is a Prime Number:**

* This IntPredicate will return true if the number is prime, otherwise false.

java

Copy code

import java.util.function.IntPredicate;

public class IntPredicateExample {

// IntPredicate to check if a number is prime

public static IntPredicate isPrime = num -> {

if (num <= 1) return false;

for (int i = 2; i <= Math.sqrt(num); i++) {

if (num % i == 0) return false;

}

return true;

};

public static void main(String[] args) {

// Test with some numbers

System.out.println("Is 11 prime? " + isPrime.test(11)); // true

System.out.println("Is 15 prime? " + isPrime.test(15)); // false

System.out.println("Is 23 prime? " + isPrime.test(23)); // true

}

}

Output:

vbnet

Copy code

Is 11 prime? true

Is 15 prime? false

Is 23 prime? true

**2. IntConsumer to Print the Square of the Given Number:**

* This IntConsumer consumes an integer and prints its square.

java

Copy code

import java.util.function.IntConsumer;

public class IntConsumerExample {

// IntConsumer to print the square of a number

public static IntConsumer printSquare = num ->

System.out.println("Square of " + num + " is: " + (num \* num));

public static void main(String[] args) {

// Test with some numbers

printSquare.accept(5); // prints 25

printSquare.accept(7); // prints 49

}

}

Output:

csharp

Copy code

Square of 5 is: 25

Square of 7 is: 49

**3. IntSupplier to Provide a Random Int Below 5000:**

* This IntSupplier will generate a random integer below 5000.

java

Copy code

import java.util.function.IntSupplier;

import java.util.Random;

public class IntSupplierExample {

// IntSupplier to give a random integer below 5000

public static IntSupplier randomIntBelow5000 = () -> new Random().nextInt(5000);

public static void main(String[] args) {

// Generate some random integers

System.out.println("Random number: " + randomIntBelow5000.getAsInt());

System.out.println("Random number: " + randomIntBelow5000.getAsInt());

System.out.println("Random number: " + randomIntBelow5000.getAsInt());

}

}

Output (values may vary due to randomness):

yaml

Copy code

Random number: 2489

Random number: 3674

Random number: 1124

**Summary of Implementations:**

1. **Prime Number Check (IntPredicate)**:
   * This predicate checks if a number is prime, returning true or false.
2. **Square of a Number (IntConsumer)**:
   * This consumer takes an integer as input and prints its square.
3. **Random Integer Supplier (IntSupplier)**:
   * This supplier generates random integers below 5000 using Java's Random class.

4o

**You said:**

Constructor Reference 1. Write a program get the details of Employee Object , such as name,account,salary , where Employee class has parameterized constructor using constructor reference

**ChatGPT said:**

ChatGPT

To demonstrate the use of constructor reference, we'll create an Employee class with a parameterized constructor and then use a constructor reference to instantiate an Employee object.

**Steps:**

1. Define the Employee class with a parameterized constructor.
2. Use Function from java.util.function package to map input values to the Employee object using constructor reference.

**Example:**

java

Copy code

import java.util.function.BiFunction;

class Employee {

private String name;

private String account;

private double salary;

// Parameterized constructor

public Employee(String name, String account, double salary) {

this.name = name;

this.account = account;

this.salary = salary;

}

// Getters

public String getName() {

return name;

}

public String getAccount() {

return account;

}

public double getSalary() {

return salary;

}

@Override

public String toString() {

return "Employee{name='" + name + "', account='" + account + "', salary=" + salary + "}";

}

}

public class ConstructorReferenceExample {

public static void main(String[] args) {

// Using constructor reference to create an Employee object

TriFunction<String, String, Double, Employee> employeeCreator = Employee::new;

// Create an Employee object

Employee emp = employeeCreator.apply("John Doe", "12345", 75000);

// Print employee details

System.out.println("Employee Details:");

System.out.println("Name: " + emp.getName());

System.out.println("Account: " + emp.getAccount());

System.out.println("Salary: " + emp.getSalary());

}

}

// TriFunction interface (since Java doesn't have a built-in one)

@FunctionalInterface

interface TriFunction<T, U, V, R> {

R apply(T t, U u, V v);

}

**Explanation:**

1. **Employee Class**:
   * A simple class with a parameterized constructor that accepts name, account, and salary.
   * It also includes getters for these attributes and a toString method for easy printing.
2. **Constructor Reference**:
   * Employee::new is a reference to the constructor of the Employee class. We use this reference with a custom functional interface TriFunction because we need three parameters (name, account, and salary).
3. **Functional Interface TriFunction**:
   * Java does not provide a built-in functional interface for functions taking three arguments, so we create a custom one.
4. **Main Method**:
   * The employeeCreator.apply("John Doe", "12345", 75000) creates a new Employee object using the constructor reference.
   * We then print the employee details using getters.

**Output:**

yaml

Copy code

Employee Details:

Name: John Doe

Account: 12345

Salary: 75000.0

This demonstrates how to use constructor references to create objects in Java. The key part is using Employee::new to refer to the constructor of the Employee class and passing it to a functional interface.

**Task 08\_Hometask\_Streams**

**Precondition: Product class definition in Java**

java

Copy code

import java.util.List;

import java.util.stream.Collectors;

class Product {

String name;

double price;

String category;

String grade;

public Product(String name, double price, String category, String grade) {

this.name = name;

this.price = price;

this.category = category;

this.grade = grade;

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

public String getCategory() {

return category;

}

@Override

public String toString() {

return "Product{" +

"name='" + name + '\'' +

", price=" + price +

", category='" + category + '\'' +

", grade='" + grade + '\'' +

'}';

}

}

1. Get all the products with price > 1000 using Java Streams

java

Copy code

public static List<Product> getProductsWithPriceGreaterThan1000(List<Product> products) {

return products.stream()

.filter(product -> product.getPrice() > 1000)

.collect(Collectors.toList());

}

2. Get all the products from electronics category using Java Streams

java

Copy code

public static List<Product> getElectronicsProducts(List<Product> products) {

return products.stream()

.filter(product -> product.getCategory().equalsIgnoreCase("electronics"))

.collect(Collectors.toList());

}

3. Get all products with price > 1000 and from electronics category, and change the names to uppercase before printing

java

Copy code

public static List<Product> getElectronicsProductsWithPriceGreaterThan1000(List<Product> products) {

return products.stream()

.filter(product -> product.getPrice() > 1000 && product.getCategory().equalsIgnoreCase("electronics"))

.map(product -> {

product.name = product.getName().toUpperCase(); // Change names to uppercase

return product;

})

.collect(Collectors.toList());

}

4. Calculate the count of all electronic products using Java Streams

java

Copy code

public static long countElectronicsProducts(List<Product> products) {

return products.stream()

.filter(product -> product.getCategory().equalsIgnoreCase("electronics"))

.count();

}

Example of how to use these methods:

java

Copy code

import java.util.Arrays;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Product> products = Arrays.asList(

new Product("Laptop", 1500, "electronics", "A"),

new Product("Phone", 900, "electronics", "B"),

new Product("TV", 1200, "electronics", "B"),

new Product("Book", 200, "stationery", "C"),

new Product("Shoes", 800, "clothing", "B")

);

// 1. Products with price > 1000

System.out.println("Products with price > 1000:");

getProductsWithPriceGreaterThan1000(products).forEach(System.out::println);

// 2. Electronics products

System.out.println("\nElectronics Products:");

getElectronicsProducts(products).forEach(System.out::println);

// 3. Electronics products with price > 1000 and uppercase names

System.out.println("\nElectronics Products with price > 1000 and uppercase names:");

getElectronicsProductsWithPriceGreaterThan1000(products).forEach(System.out::println);

// 4. Count of electronic products

long count = countElectronicsProducts(products);

System.out.println("\nCount of electronics products: " + count);

}

}

Expected Output:

csharp

Copy code

Products with price > 1000:

Product{name='Laptop', price=1500.0, category='electronics', grade='A'}

Product{name='TV', price=1200.0, category='electronics', grade='B'}

Electronics Products:

Product{name='Laptop', price=1500.0, category='electronics', grade='A'}

Product{name='Phone', price=900.0, category='electronics', grade='B'}

Product{name='TV', price=1200.0, category='electronics', grade='B'}

Electronics Products with price > 1000 and uppercase names:

Product{name='LAPTOP', price=1500.0, category='electronics', grade='A'}

Product{name='TV', price=1200.0, category='electronics', grade='B'}

Count of electronics products: 3

This solution demonstrates how to use Java Streams for filtering and manipulating collections based on specific criteria, such as filtering products by price, category, and transforming product names to uppercase.

**Task 09\_Hometask\_DateTime**

**1. Predicate to check if the given date is yesterday’s date**

java

Copy code

import java.time.LocalDate;

import java.util.function.Predicate;

public class DatePredicates {

public static Predicate<LocalDate> isYesterday = date -> {

LocalDate yesterday = LocalDate.now().minusDays(1);

return date.equals(yesterday);

};

}

**Usage:**

java

Copy code

LocalDate someDate = LocalDate.of(2024, 10, 13); // Example date

System.out.println(DatePredicates.isYesterday.test(someDate)); // true if yesterday, false otherwise

**2. Supplier to get Date for next Thursday**

java

Copy code

import java.time.DayOfWeek;

import java.time.LocalDate;

import java.util.function.Supplier;

public class DateSuppliers {

public static Supplier<LocalDate> nextThursday = () -> {

LocalDate today = LocalDate.now();

return today.with(DayOfWeek.THURSDAY).isBefore(today) ?

today.with(DayOfWeek.THURSDAY).plusWeeks(1) : today.with(DayOfWeek.THURSDAY);

};

}

**Usage:**

java

Copy code

System.out.println("Next Thursday: " + DateSuppliers.nextThursday.get());

**3. Supplier to get Current Time in EST timezone**

java

Copy code

import java.time.ZoneId;

import java.time.ZonedDateTime;

import java.util.function.Supplier;

public class TimeSuppliers {

public static Supplier<ZonedDateTime> currentTimeInEST = () -> {

return ZonedDateTime.now(ZoneId.of("America/New\_York"));

};

}

**Usage:**

java

Copy code

System.out.println("Current time in EST: " + TimeSuppliers.currentTimeInEST.get());

**4. Function to calculate the age of a person given date of birth**

java

Copy code

import java.time.LocalDate;

import java.time.Period;

import java.util.function.Function;

public class AgeFunctions {

public static Function<LocalDate, Integer> calculateAge = (dob) -> {

return Period.between(dob, LocalDate.now()).getYears();

};

}

**Usage:**

java

Copy code

LocalDate dob = LocalDate.of(1990, 6, 15); // Example date of birth

System.out.println("Age: " + AgeFunctions.calculateAge.apply(dob));

**Example of all usage together:**

java

Copy code

import java.time.LocalDate;

import java.time.ZonedDateTime;

public class Main {

public static void main(String[] args) {

// 1. Check if a date is yesterday's date

LocalDate date = LocalDate.of(2024, 10, 13);

System.out.println("Is Yesterday: " + DatePredicates.isYesterday.test(date));

// 2. Get next Thursday

System.out.println("Next Thursday: " + DateSuppliers.nextThursday.get());

// 3. Get current time in EST timezone

ZonedDateTime currentTimeEST = TimeSuppliers.currentTimeInEST.get();

System.out.println("Current time in EST: " + currentTimeEST);

// 4. Calculate age given date of birth

LocalDate dob = LocalDate.of(1990, 6, 15);

System.out.println("Age: " + AgeFunctions.calculateAge.apply(dob));

}

}

**Expected Output:**

yaml

Copy code

Is Yesterday: true

Next Thursday: 2024-10-17

Current time in EST: 2024-10-14T12:34:56.789-04:00[America/New\_York]

Age: 34

This code provides concise implementations using Java's functional programming capabilities such as **Predicate**, **Supplier**, and **Function** interfaces.

API TESTING

**Task01\_HomeTask\_Webservices\_Functional**

Top of Form

Bottom of Form

To perform web service testing on the provided WSDL web service (http://www.dneonline.com/calculator.asmx), you can use various tools like **Postman**, **SoapUI**, or even **Rest Assured** in Java. The four methods exposed by the service are as follows:

* **Add (int a, int b)**: Adds two integers.
* **Subtract (int a, int b)**: Subtracts the second integer from the first.
* **Multiply (int a, int b)**: Multiplies two integers.
* **Divide (int a, int b)**: Divides the first integer by the second.

**Here's a step-by-step guide to perform this testing:**

**1. Using Postman for SOAP Requests**

Postman can be used to test SOAP services with the following steps:

1. **Open Postman** and set the request type to POST.
2. **Paste the URL**: http://www.dneonline.com/calculator.asmx?WSDL.
3. **Enter SOAP Body**: In the Body tab, select raw and choose XML (application/xml) as the format.

**Example for Add Operation**:

xml

Copy code

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<Add xmlns="http://tempuri.org/">

<intA>5</intA>

<intB>3</intB>

</Add>

</soap:Body>

</soap:Envelope>

1. **Send the Request** and observe the response for validation.

**2. SOAP Request Examples for All Methods**

Here are XML SOAP requests for each of the methods:

**a. Add Method**

xml

Copy code

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<Add xmlns="http://tempuri.org/">

<intA>5</intA>

<intB>7</intB>

</Add>

</soap:Body>

</soap:Envelope>

**b. Subtract Method**

xml

Copy code

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<Subtract xmlns="http://tempuri.org/">

<intA>15</intA>

<intB>7</intB>

</Subtract>

</soap:Body>

</soap:Envelope>

**c. Multiply Method**

xml

Copy code

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<Multiply xmlns="http://tempuri.org/">

<intA>6</intA>

<intB>3</intB>

</Multiply>

</soap:Body>

</soap:Envelope>

**d. Divide Method**

xml

Copy code

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<Divide xmlns="http://tempuri.org/">

<intA>10</intA>

<intB>2</intB>

</Divide>

</soap:Body>

</soap:Envelope>

**3. Send the Requests and Check Responses**

You will get an XML response for each request. Validate that the response data is correct and matches the expected outputs.

**4. Finding Defects**

Try testing various scenarios like:

1. **Edge cases** such as:
   * Dividing by zero.
   * Subtraction of larger number from smaller one.
   * Large integer values to check for overflow issues.
2. **Invalid inputs**:
   * Send string values instead of integers in the requests.
   * Missing tags in the request body.
3. **Negative testing**:
   * Invalid or malformed XML structure.

**5. Capturing Screenshots**

For each of the tested methods, take screenshots of the request sent and the response received. You can share these screenshots with your mentor for evaluation.

**6. Example Defects You Might Find**

* **Division by zero**: The response might not handle this case properly.
* **Large number handling**: The service might fail or return incorrect values when very large integers are used.

Would you like a guide on using **SoapUI** for this testing? Or would you prefer help writing Java code to test these SOAP requests using **Rest Assured**?

**Task 02\_HomeTask\_REST\_Functional**

Base URL -> https://jsonplaceholder.typicode.com

Resources available -->

/posts

/comments

/albums

/photos

/todos

/users

Task

1. Verify the number of resources at each resource end point (6 marks)

2. For each type of resource

a. Get a specific resource data (6 marks)

i. verify the response code

ii. Verify the response body

b. Modify a specific resource data (6 marks)

i. verify the response code

ii. Verify the response body

c. Delete a specific resource (6 marks)

i. verify the response code

ii. Verify the response body

d. Create your own resource (6 marks)

i. verify the response code

ii. Verify the response body

To complete this task, we'll be testing each of the REST API endpoints provided by **JSONPlaceholder** (https://jsonplaceholder.typicode.com) using tools such as **Postman** or **Rest Assured** in Java. Below are the detailed steps and structure to address each task. You can implement them using Postman or Rest Assured as per your choice.

**Task Breakdown**

**1. Verify the number of resources at each resource endpoint (6 marks)**

You need to send GET requests to each endpoint and validate the number of resources returned.

* **Request**:

arduino

Copy code

GET https://jsonplaceholder.typicode.com/posts

GET https://jsonplaceholder.typicode.com/comments

GET https://jsonplaceholder.typicode.com/albums

GET https://jsonplaceholder.typicode.com/photos

GET https://jsonplaceholder.typicode.com/todos

GET https://jsonplaceholder.typicode.com/users

* **Validation**:
  + Verify the HTTP status code (200 OK).
  + Validate the length of the JSON array returned at each endpoint.

**2. For each type of resource**

We'll perform four operations on each type of resource (posts, comments, albums, photos, todos, users).

**a. Get a specific resource data (6 marks)**

* **Request**:

ruby

Copy code

GET https://jsonplaceholder.typicode.com/posts/1

GET https://jsonplaceholder.typicode.com/comments/1

GET https://jsonplaceholder.typicode.com/albums/1

GET https://jsonplaceholder.typicode.com/photos/1

GET https://jsonplaceholder.typicode.com/todos/1

GET https://jsonplaceholder.typicode.com/users/1

* **Validation**:
  + **Response code**: Verify if it’s 200 OK.
  + **Response body**: Validate the JSON structure and specific fields like id, title, body, name, email, etc.

**b. Modify a specific resource data (6 marks)**

We'll use PUT or PATCH requests to modify the data for a specific resource. In this example, we'll use PUT to update an existing post.

* **Request**:

arduino

Copy code

PUT https://jsonplaceholder.typicode.com/posts/1

* + **Request Body**:

json

Copy code

{

"id": 1,

"title": "Modified Title",

"body": "Modified Body",

"userId": 1

}

* **Validation**:
  + **Response code**: Ensure it's 200 OK.
  + **Response body**: Verify the response reflects the modified data (i.e., the title and body are updated).

**c. Delete a specific resource (6 marks)**

We'll send a DELETE request to remove a specific resource.

* **Request**:

arduino

Copy code

DELETE https://jsonplaceholder.typicode.com/posts/1

* **Validation**:
  + **Response code**: Verify that the status code is 200 OK or 204 No Content.
  + **Response body**: Check that the response is either empty or reflects successful deletion.

**d. Create your own resource (6 marks)**

Use a POST request to create a new resource.

* **Request**:

arduino

Copy code

POST https://jsonplaceholder.typicode.com/posts

* + **Request Body**:

json

Copy code

{

"title": "New Title",

"body": "This is a new post",

"userId": 1

}

* **Validation**:
  + **Response code**: Ensure that the response code is 201 Created.
  + **Response body**: Verify that the newly created resource is returned in the response with a new id.

**Example Implementation in Postman**

1. **Getting Resource Data**:
   * **Method**: GET
   * **URL**: https://jsonplaceholder.typicode.com/posts/1
   * **Assertions**:
     + Check for status 200 OK.
     + Validate the structure of the response.
2. **Modifying Resource Data**:
   * **Method**: PUT
   * **URL**: https://jsonplaceholder.typicode.com/posts/1
   * **Body**:

json

Copy code

{

"id": 1,

"title": "Modified Title",

"body": "Modified Body",

"userId": 1

}

* + **Assertions**:
    - Check for status 200 OK.
    - Verify that the title and body are updated.

1. **Deleting a Resource**:
   * **Method**: DELETE
   * **URL**: https://jsonplaceholder.typicode.com/posts/1
   * **Assertions**:
     + Check for status 200 OK or 204 No Content.
2. **Creating a New Resource**:
   * **Method**: POST
   * **URL**: https://jsonplaceholder.typicode.com/posts
   * **Body**:

json

Copy code

{

"title": "New Post",

"body": "This is a new post.",

"userId": 1

}

* + **Assertions**:
    - Check for status 201 Created.
    - Verify that the new resource is returned with a new id.

**Example Implementation in Rest Assured (Java)**

java

Copy code

import io.restassured.RestAssured;

import io.restassured.response.Response;

import static io.restassured.RestAssured.given;

import static org.hamcrest.Matchers.equalTo;

public class JsonPlaceholderTests {

public static void main(String[] args) {

// Base URI

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

// Get a specific resource

Response getResponse = given()

.when()

.get("/posts/1")

.then()

.statusCode(200)

.extract()

.response();

System.out.println(getResponse.asPrettyString());

// Modify a specific resource

String putBody = "{ \"id\": 1, \"title\": \"Modified Title\", \"body\": \"Modified Body\", \"userId\": 1 }";

given()

.header("Content-Type", "application/json")

.body(putBody)

.when()

.put("/posts/1")

.then()

.statusCode(200)

.body("title", equalTo("Modified Title"))

.body("body", equalTo("Modified Body"));

// Delete a specific resource

given()

.when()

.delete("/posts/1")

.then()

.statusCode(200);

// Create a new resource

String postBody = "{ \"title\": \"New Post\", \"body\": \"This is a new post.\", \"userId\": 1 }";

Response postResponse = given()

.header("Content-Type", "application/json")

.body(postBody)

.when()

.post("/posts")

.then()

.statusCode(201)

.extract()

.response();

System.out.println(postResponse.asPrettyString());

}

}

**Conclusion**

For each operation (Get, Modify, Delete, and Create), you should verify the HTTP status code, response body, and any changes in the data structure as per the task requirements. Each operation covers key parts of the API testing process, helping identify potential issues with the API.

**Task 03\_HomeTask\_REST\_Automation**

**Task 1: Petstore API Testing**

1. **Create a PET using the POST call** from http://petstore.swagger.io/#/

**Request Body:**

json

Copy code

{

"id": 12345,

"category": {

"id": 1,

"name": "dog"

},

"name": "snoopie",

"photoUrls": [

"string"

],

"tags": [

{

"id": 0,

"name": "string"

}

],

"status": "pending"

}

1. **Test using GET call to verify the following**:
   * Status code is 200.
   * Content type is application/json.
   * Pet is a "dog".
   * Pet's name is "snoopie".
   * Status is "pending".

Here is the **Rest Assured** code for Task 1:

java

Copy code

import io.restassured.RestAssured;

import io.restassured.http.ContentType;

import io.restassured.response.Response;

import static io.restassured.RestAssured.given;

import static org.hamcrest.Matchers.equalTo;

public class PetstoreAPITest {

public static void main(String[] args) {

// Base URI for petstore

RestAssured.baseURI = "https://petstore.swagger.io/v2";

// 1. POST Call: Create a pet

String requestBody = "{\n" +

" \"id\": 12345,\n" +

" \"category\": {\n" +

" \"id\": 1,\n" +

" \"name\": \"dog\"\n" +

" },\n" +

" \"name\": \"snoopie\",\n" +

" \"photoUrls\": [\n" +

" \"string\"\n" +

" ],\n" +

" \"tags\": [\n" +

" {\n" +

" \"id\": 0,\n" +

" \"name\": \"string\"\n" +

" }\n" +

" ],\n" +

" \"status\": \"pending\"\n" +

"}";

Response postResponse = given()

.header("Content-Type", "application/json")

.body(requestBody)

.when()

.post("/pet")

.then()

.statusCode(200)

.extract()

.response();

System.out.println("Pet created: " + postResponse.asPrettyString());

// 2. GET Call: Verify the pet details

given()

.when()

.get("/pet/12345")

.then()

.statusCode(200) // Verify status code is 200

.contentType(ContentType.JSON) // Verify content type is JSON

.body("category.name", equalTo("dog")) // Verify pet is a 'dog'

.body("name", equalTo("snoopie")) // Verify pet name is 'snoopie'

.body("status", equalTo("pending")); // Verify pet status is 'pending'

}

}

**Explanation:**

* **POST** request creates a pet with the ID 12345 and verifies that the pet has been created.
* **GET** request checks if:
  + The status code is 200.
  + The content type is application/json.
  + The pet's category is dog.
  + The pet's name is snoopie.
  + The pet's status is pending.

**Task 2: JSONPlaceholder API Testing**

We will be sending a GET request to https://jsonplaceholder.typicode.com/users and verify:

1. **GET Call** to https://jsonplaceholder.typicode.com/users
   * Status code is 200.
   * There are more than 3 users in the list.
   * One of the users has the name "Ervin Howell".

Here is the **Rest Assured** code for Task 2:

java

Copy code

import io.restassured.RestAssured;

import io.restassured.http.ContentType;

import static io.restassured.RestAssured.given;

import static org.hamcrest.Matchers.\*;

public class JsonPlaceholderAPITest {

public static void main(String[] args) {

// Base URI for JSONPlaceholder

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

// 1. GET Call: Verify the list of users

given()

.when()

.get("/users")

.then()

.statusCode(200) // Verify status code is 200

.contentType(ContentType.JSON) // Verify content type is JSON

.body("size()", greaterThan(3)) // Verify there are more than 3 users

.body("find { it.name == 'Ervin Howell' }", notNullValue()); // Verify user with name "Ervin Howell" exists

}

}

**Explanation:**

* **GET** request checks:
  + The status code is 200.
  + There are more than 3 users in the response.
  + One of the users has the name "Ervin Howell".

**Steps to run the test:**

1. Import Rest Assured into your Java project (using Maven or Gradle).
2. Copy and paste the code provided for Task 1 and Task 2.
3. Run the tests and validate the results.

By executing these tests, you'll cover both the **Petstore** and **JSONPlaceholder** APIs, verifying key aspects such as status codes, content types, and specific response fields.

**Task 04\_HomeTask\_REST\_Automation  
  
Find the event names which are in English(En) language from below.**

**Verify the list of names returned with expected list of values**

**End Point : https://events.epam.com/api/v2/events**

**Hint:**

**- Use http://jsonviewer.stack.hu/ to remove/format the JSON response**

**- Use https://groovy-playground.appspot.com/ to verify the Gpath expression you write**

**import groovy.json.JsonSlurper**

**def jsonSlurper = new JsonSlurper()**

**def object = jsonSlurper.parseText(**

**'{Your JSON response here}'**

**)**

**println(object)**

**===============================================================**

**Task - 2 (20 marks):**

**openweathermap.org exposes APIs for reading the weather data from various cities. The documentation is at https://openweathermap.org/guide#how**

**1. Use the below weather end-point to get the current weather details of Hyderabad**

**http://api.openweathermap.org/data/2.5/weather?q=hyderabad&appid={your\_key}**

**2. Use the coordinates (longitude and latitude) of of the above response to the end-point**

**http://api.openweathermap.org/data/2.5/weather?lat={latitude}&lon={longitude}&appid={your\_key} and verify the below in response**

**- name --> Hyderabad**

**- sys.country --> IN**

**- main.temp\_min --> greater than 0**

**- main.temp --> greater than 0**

**Hint:**

**- You need to register in the application and get your API key as mentioned here in your API requests**

**Task 1: Fetching Event Names in English**

1. **API Endpoint: https://events.epam.com/api/v2/events**
2. **Objective: Find event names in English (En) and verify them against an expected list.**

**Step-by-Step Implementation**

1. **Make a GET request to fetch the events.**
2. **Parse the JSON response to filter event names with the language "En".**
3. **Verify the list of event names against an expected list.**

**Here's how you can do this using Java with Rest Assured:**

**java**

**Copy code**

**import io.restassured.RestAssured;**

**import io.restassured.response.Response;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**import static io.restassured.RestAssured.given;**

**public class EventNamesFetcher {**

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

**// Base URI for events API**

**RestAssured.baseURI = "https://events.epam.com/api/v2";**

**// Make GET request to fetch events**

**Response response = given()**

**.when()**

**.get("/events")**

**.then()**

**.statusCode(200) // Verify the response status**

**.extract().response();**

**// Parse the response to get event names in English**

**List<String> eventNamesInEnglish = response.jsonPath()**

**.getList("findAll { it.language == 'En' }.name");**

**// Expected list of event names**

**List<String> expectedEventNames = List.of("Event 1", "Event 2", "Event 3"); // Replace with actual expected values**

**// Verify the event names**

**System.out.println("Event Names in English: " + eventNamesInEnglish);**

**System.out.println("Expected Event Names: " + expectedEventNames);**

**if (eventNamesInEnglish.equals(expectedEventNames)) {**

**System.out.println("The event names match the expected list.");**

**} else {**

**System.out.println("The event names do not match the expected list.");**

**}**

**}**

**}**

**Explanation:**

* **We send a GET request to the events API and expect a successful response (status code 200).**
* **We use the JSONPath to filter out event names where the language is "En".**
* **Finally, we compare the returned list with an expected list and print the results.**

**Task 2: Weather Data from OpenWeatherMap API**

1. **Get current weather details of Hyderabad.**
2. **Use coordinates from the response to verify specific details.**

**Step-by-Step Implementation**

1. **Make a GET request to fetch weather details for Hyderabad.**
2. **Extract latitude and longitude from the response.**
3. **Use those coordinates to fetch weather details again and verify specific fields.**

**Here's how to implement this:**

**java**

**Copy code**

**import io.restassured.RestAssured;**

**import io.restassured.response.Response;**

**import static io.restassured.RestAssured.given;**

**public class WeatherDataFetcher {**

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

**// Replace with your OpenWeatherMap API key**

**String apiKey = "{your\_key}";**

**String city = "Hyderabad";**

**// Base URI for OpenWeatherMap API**

**RestAssured.baseURI = "http://api.openweathermap.org/data/2.5";**

**// Step 1: Get current weather details for Hyderabad**

**Response response = given()**

**.queryParam("q", city)**

**.queryParam("appid", apiKey)**

**.when()**

**.get("/weather")**

**.then()**

**.statusCode(200) // Verify the response status**

**.extract().response();**

**// Extract latitude and longitude**

**double latitude = response.jsonPath().getDouble("coord.lat");**

**double longitude = response.jsonPath().getDouble("coord.lon");**

**// Step 2: Get weather details using coordinates**

**Response coordResponse = given()**

**.queryParam("lat", latitude)**

**.queryParam("lon", longitude)**

**.queryParam("appid", apiKey)**

**.when()**

**.get("/weather")**

**.then()**

**.statusCode(200) // Verify the response status**

**.extract().response();**

**// Verify the required fields**

**String cityName = coordResponse.jsonPath().getString("name");**

**String countryCode = coordResponse.jsonPath().getString("sys.country");**

**double tempMin = coordResponse.jsonPath().getDouble("main.temp\_min");**

**double temp = coordResponse.jsonPath().getDouble("main.temp");**

**// Assertions**

**System.out.println("City: " + cityName);**

**System.out.println("Country Code: " + countryCode);**

**System.out.println("Minimum Temperature: " + tempMin);**

**System.out.println("Current Temperature: " + temp);**

**// Verify the conditions**

**if ("Hyderabad".equals(cityName) && "IN".equals(countryCode) && tempMin > 0 && temp > 0) {**

**System.out.println("All conditions met.");**

**} else {**

**System.out.println("Some conditions are not met.");**

**}**

**}**

**}**

**Explanation:**

* **We send a GET request to fetch the current weather for Hyderabad and extract the latitude and longitude.**
* **We then use those coordinates to make another request to get the weather details and verify the specified fields.**
* **The output prints the results, and the conditions are verified against the expected values.**

**Running the Code:**

1. **Make sure you have the necessary dependencies for Rest Assured in your project.**
2. **Replace {your\_key} with your actual OpenWeatherMap API key.**
3. **Compile and run the Java files, and the output will show the results.**

**These implementations provide a robust way to interact with both the events API and the weather API, verifying responses and extracting the necessary information.**

CUCUMBER

**BDD 101: Writing Good Gherkin**

So, [you and your team have decided](https://automationpanda.com/2017/06/22/who-should-lead-bdd/) to make [test automation](https://automationpanda.com/2017/01/16/10-things-you-lose-without-automation/) a priority. You plan to use [behavior-driven development](https://automationpanda.com/bdd/) to shift left with testing. You read the [BDD 101 Series](https://automationpanda.com/2017/01/25/bdd-101-introducing-bdd/) up through [the previous post](https://automationpanda.com/2017/01/27/bdd-101-gherkin-by-example/). You picked a [good language](https://automationpanda.com/2017/01/21/the-best-programming-language-for-test-automation/) for test automation. You even peeked at [Cucumber-JVM](https://automationpanda.com/2017/10/24/cucumber-jvm-for-java/) or [another BDD framework](https://automationpanda.com/2017/02/04/bdd-101-frameworks/) on your own. That’s great! Big steps! And now, you are ready to write your first Gherkin feature file.  You fire open [Atom with a Gherkin plugin](https://automationpanda.com/2017/10/12/gherkin-syntax-highlighting-in-atom/) or [Notepad++ with a Gherkin UDL](https://automationpanda.com/2017/01/18/gherkin-syntax-highlighting-in-notepad/), you type “Given” on the first line, and…

Writer’s block.  *How am I supposed to write my Gherkin steps?*

Good Gherkin feature files are not easy to write at first. Writing is definitely an art. With some basic pointers, and a bit of practice, Gherkin becomes easier. This post will cover how to write top-notch feature files. (Check the [Automation Panda BDD page](https://automationpanda.com/bdd/) for the full table of contents.)

*The Golden Gherkin Rule: Treat other readers as you would want to be treated. Write Gherkin so that people who don’t know the feature will understand it.*

**Proper Behavior**

The biggest mistake BDD beginners make is writing Gherkin without a behavior-driven mindset. They often write feature files as if they are writing “traditional” *procedure-driven* functional tests: step-by-step instructions with actions and expected results. HP ALM, [qTest](https://www.qasymphony.com/software-testing-tools/qtest-manager/test-case-management/), [AccelaTest](https://accelatest.com/), and many other test repository tools store tests in this format. These procedure-driven tests are often *imperative* and trace a path through the system that covers multiple behaviors. As a result, they may be unnecessarily long, which can delay failure investigation, increase maintenance costs, and create confusion.

For example, let’s consider a test that searches for images of pandas on Google. Below would be a reasonable test procedure:

1. Open a web browser.
   1. Web browser opens successfully.
2. Navigate to https://www.google.com/.
   1. The web page loads successfully and the Google image is visible.
3. Enter “panda” in the search bar.
   1. Links related to “panda” are shown on the results page.
4. Click on the “Images” link at the top of the results page.
   1. Images related to “panda” are shown on the results page.

I’ve seen many newbies translate a test like this into Gherkin like the following:

*# BAD EXAMPLE! Do not copy.*

**Feature**: Google Searching

**Scenario**: Google Image search shows pictures

**Given** the user opens a web browser

**And** the user navigates to "https://www.google.com/"

**When** the user enters "panda" into the search bar

**Then** links related to "panda" are shown on the results page

**When** the user clicks on the "Images" link at the top of the results page

**Then** images related to "panda" are shown on the results page

This scenario is terribly wrong. All that happened was that the author put BDD buzzwords in front of each step of the traditional test. This is *not* behavior-driven, it is still procedure-driven.

The first two steps are purely setup: they just go to Google, and they are strongly imperative. Since they don’t focus on the desired behavior, they can be reduced to one declarative step: “Given a web browser is at the Google home page.” This new step is friendlier to read.

After the Given step, there are two When-Then pairs. This is syntactically incorrect: **Given-When-Then steps must appear in order and cannot repeat**. A Given may not follow a When or Then, and a When may not follow a Then. The reason is simple: **any single When-Then pair denotes an individual behavior**. This makes it easy to see how, in the test above, there are actually two behaviors covered: (1) searching from the search bar, and (2) performing an image search. In Gherkin, **one scenario covers one behavior**. Thus, there should be two scenarios instead of one. Any time you want to write more than one When-Then pair, write separate scenarios instead. (Note: Some BDD frameworks may allow disordered steps, but it would nevertheless be anti-behavioral.)

This splitting technique also reveals unnecessary behavior coverage. For instance, the first behavior to search from the search bar may be covered in another feature file. I once saw a scenario with about 30 When-Then pairs, and many were duplicate behaviors.

Do not be tempted to arbitrarily reassign step types to make scenarios follow strict Given-When-Then ordering. **Respect the integrity of the step types**: Givens set up initial state, Whens perform an action, and Thens verify outcomes. In the example above, the first Then step *could* have been turned into a When step, but that would be incorrect because it makes an assertion. Step types are meant to be guide rails for writing good behavior scenarios.

The correct feature file would look something like this:

**Feature**: Google Searching

**Scenario**: Search from the search bar

**Given** a web browser is at the Google home page

**When** the user enters "panda" into the search bar

**Then** links related to "panda" are shown on the results page

**Scenario**: Image search

**Given** Google search results for "panda" are shown

**When** the user clicks on the "Images" link at the top of the results page

**Then** images related to "panda" are shown on the results page

The second behavior arguably needs the first behavior to run first because the second needs to start at the search result page. However, since that is merely setup for the behavior of image searching and is not part of it, the Given step in the second scenario can basically declare (*declaratively*) that the “panda” search must already be done. Of course, this means that the “panda” search would be run redundantly at test time, but the separation of scenarios guarantees behavior-level independence.

*The Cardinal Rule of BDD: One Scenario, One Behavior!*

Remember, behavior scenarios are more than tests – they also represent requirements and acceptance criteria. **Good Gherkin comes from good behavior**.

(For deeper information about the Cardinal Rule of BDD and multiple When-Then pairs per scenario, please refer to my article, [Are Gherkin Scenarios with Multiple When-Then Pairs Okay?](https://automationpanda.com/2018/02/03/are-gherkin-scenarios-with-multiple-when-then-pairs-okay/))

**Phrasing Steps**

How you write a step matters. If you write a step poorly, it cannot easily be reused. Thankfully, some basic rules maintain consistent phrasing and maximum reusability.

**Write all steps in third-person point of view**. If first-person and third-person steps mix, scenarios become confusing. I even dedicated a whole blog post entirely to this point: [Should Gherkin Steps Use First-Person or Third-Person?](https://automationpanda.com/2017/01/18/should-gherkin-steps-use-first-person-or-third-person/) TL;DR: just use third-person at all times.

**Write steps as a subject-predicate action phrase**. It may tempting to leave parts of speech out of a step line for brevity, especially when using Ands and Buts, but partial phrases make steps ambiguous and more likely to be reused improperly. For example, consider the following example:

*# BAD EXAMPLE! Do not copy.*

**Feature**: Google Searching

**Scenario**: Google search result page elements

**Given** the user navigates to the Google home page

**When** the user entered "panda" at the search bar

**Then** the results page shows links related to "panda"

**And** image links for "panda"

**And** video links for "panda"

The final two And steps lack the subject-predicate phrase format. Are the links meant to be subjects, meaning that they perform some action? Or, are they meant to be direct objects, meaning that they receive some action? Are they meant to be on the results page or not? What if someone else wrote a scenario for a different page that also had image and video links – could they reuse these steps? Writing steps without a clear subject and predicate is not only poor English but poor communication.

Also, use appropriate tense and phrasing for each type of step. For simplicity, **use present tense for all step types**. Rather than take a time warp back to middle school English class, let’s illustrate tense with a bad example:

*# BAD EXAMPLE! Do not copy.*

**Feature**: Google Searching

**Scenario**: Simple Google search

**Given** the user navigates to the Google home page

**When** the user entered "panda" at the search bar

**Then** links related to "panda" will be shown on the results page

The Given step above uses present tense, but its subject is misleading. It indicates an action when it says, “Given the user *navigates*.” Actions imply the exercise of behavior. However, Given steps are meant to establish an initial state, *not* exercise a behavior. This may seem like a trivial nuance, but it can confuse feature file authors who may not be able to tell if a step is a Given or When. A better phrasing would be, “Given the Google home page is displayed.” It establishes a starting point for the scenario. Use present tense with an appropriate subject to indicate a state rather than an action.

The When step above uses past tense when it says, “The user *entered*.” This indicates that an action has already happened. However, When steps should indicate that an action is presently happening. Plus, past tense here conflicts with the tenses used in the other steps.

The Then step above uses future tense when it says, “The results *will be shown*.” Future tense seems practical for Then steps because it indicates what the result should be after the current action is taken. However, future tense reinforces a procedure-driven approach because it treats the scenario as a time sequence. A behavior, on the other hand, is a present-tense aspect of the product or feature. Thus, it is better to write Then steps in the present tense.

The corrected example looks like this:

**Feature**: Google Searching

**Scenario**: Simple Google search

**Given** the Google home page is displayed

**When** the user enters "panda" into the search bar

**Then** links related to "panda" are shown on the results page

And note, all steps are written in third-person. Read [Should Gherkin Steps use Past, Present, or Future Tense?](https://automationpanda.com/2021/05/11/should-gherkin-steps-use-past-present-or-future-tense/) to learn more.

**Good Titles**

Good titles are just as important as good steps. **The title is like the face of a scenario** – it’s the first thing people read. It must communicate in one concise line what the behavior is. Titles are often logged by the [automation framework](https://automationpanda.com/2017/02/03/bdd-101-automation/) as well. Specific pointers for writing good scenario titles are given in my article, [Good Gherkin Scenario Titles](https://automationpanda.com/2018/01/31/good-gherkin-scenario-titles/).

**Choices, Choices**

Another common misconception for beginners is thinking that Gherkin has an “Or” step for conditional or combinatorial logic. People may presume that Gherkin has “Or” because it has “And”, or perhaps programmers want to treat Gherkin like a [structured language](https://en.wikipedia.org/wiki/Structured_programming). **However, Gherkin does not have an “Or” step**. When automated, every step is executed sequentially.

Below is a bad example based on [a classic Super Mario video game](https://en.wikipedia.org/wiki/Super_Mario_All-Stars), showing how people might want to use “Or”:

*# BAD EXAMPLE! Do not copy.*

**Feature**: SNES Mario Controls

**Scenario**: Mario jumps

**Given** a level is started

**When** the player pushes the "A" button

**Or** the player pushes the "B" button

**Then** Mario jumps straight up

Clearly, the author’s intent is to say that Mario should jump when the player pushes either of two buttons. The author wants to cover multiple variations of the same behavior. In order to do this the right way, **use Scenario Outline sections to cover multiple variations of the same behavior**, as shown below:

**Feature**: SNES Mario Controls

**Scenario Outline**: Mario jumps

**Given** a level is started

**When** the player pushes the "<letter>" button

**Then** Mario jumps straight up

**Examples**: Buttons

| letter |

| A |

| B |

**The Known Unknowns**

Test data can be difficult to handle. Sometimes, it may be possible to seed data in the system and write tests to reference it, but other times, it may not. Google search is the prime example: the result list will change over time as both Google and the Internet change. To handle the [known unknowns](https://en.wikipedia.org/wiki/There_are_known_knowns), **write scenarios defensively so that changes in the underlying data do not cause test runs to fail**. Furthermore, to be truly behavior-driven, **think about data not as test data but as examples of behavior**.

Consider the following example from the previous post:

**Feature**: Google Searching

**Scenario**: Simple Google search

**Given** a web browser is on the Google page

**When** the search phrase "panda" is entered

**Then** results for "panda" are shown

**And** the following related results are shown

| related |

| Panda Express |

| giant panda |

| panda videos |

This scenario uses a step table to explicitly name results that should appear for a search. The step with the table would be implemented to iterate over the table entries and verify each appeared in the result list. However, what if Panda Express were to go out of business and thus no longer be ranked as high in the results? (Let’s hope not.) The test run would then fail, not because the search feature is broken, but because a hard-coded variation became invalid. It would be better to write a step that more intelligently verified that each returned result somehow related to the search phrase, like this: “And links related to ‘panda’ are shown on the results page.” The step definition implementation could use regular expression parsing to verify the presence of “panda” in each result link.

Another nice feature of Gherkin is that **step definitions can hide data in the automation** when it doesn’t need to be exposed. Step definitions may also pass data to future steps in the automation. For example, consider another Google search scenario:

**Feature**: Google Searching

**Scenario**: Search result linking

**Given** Google search results for "panda" are shown

**When** the user clicks the first result link

**Then** the page for the chosen result link is displayed

Notice how the When step does not explicitly name the value of the result link – it simply says to click the first one. The value of the first link may change over time, but there will always be a first link. The Then step must know something about the chosen link in order to successfully verify the outcome, but it can simply reference it as “the chosen result link”. Behind the scenes, in the step definitions, the When step can store the value of the chosen link in a variable and pass the variable forward to the Then step.

**Handling Test Data**

Some types of test data should be handled directly within the Gherkin, but other types should not. Remember that BDD is *specification by example* – scenarios should be descriptive of the behaviors they cover, and any data written into the Gherkin should support that descriptive nature. Read [Handling Test Data in BDD](https://automationpanda.com/2017/08/05/handling-test-data-in-bdd/) for comprehensive information on handling test data.

**Less is More**

**Scenarios should be short and sweet**. I typically recommend that scenarios should have a single-digit step count (<10). Long scenarios are hard to understand, and they are often indicative of poor practices. One such problem is writing *imperative* steps instead of *declarative* steps. I have touched on this topic before, but I want to thoroughly explain it here.

Imperative steps state the mechanics of *how* an action should happen. They are very procedure-driven. For example, consider the following When steps for entering a Google search:

1. When the user scrolls the mouse to the search bar
2. And the user clicks the search bar
3. And the user types the letter “p”
4. And the user types the letter “a”
5. And the user types the letter “n”
6. And the user types the letter “d”
7. And the user types the letter “a”
8. And the user types the ENTER key

Now, the granularity of actions may seem like overkill, but it illustrates the point that imperative steps focus very much on how actions are taken. Thus, they often need many steps to fully accomplish the intended behavior. Furthermore, the intended behavior is not always as self-documented as with declarative steps.

Declarative steps state *what* action should happen without providing all of the information for how it will happen. They are behavior-driven because they express action at a higher level. All of the imperative steps in the example above could be written in one line: “When the user enters ‘panda’ at the search bar.” The scrolling and keystroking is implied, and it will ultimately be handled by the automation in the step definition. **When trying to reduce step count, ask yourself if your steps can be written more declaratively**.

Another reason for lengthy scenarios is scenario outline abuse. Scenario outlines make it all too easy to add unnecessary rows and columns to their Examples tables. Unnecessary rows waste test execution time. Extra columns indicate complexity. Both should be avoided. Below are questions to ask yourself when facing an oversized scenario outline:

* Does each row represent an equivalence class of variations?
  + For example, searching for “elephant” in addition to “panda” does not add much test value.
* Does every combination of inputs need to be covered?
  + *N* columns with *M* inputs each generates *MN* possible combinations.
  + Consider making each input appear only once, regardless of combination.
* Do any columns represent separate behaviors?
  + This may be true if columns are never referenced together in the same step.
  + If so, consider splitting apart the scenario outline by column.
* Does the feature file reader need to explicitly know all of the data?
  + Consider hiding some of the data in step definitions.
  + Some data may be derivable from other data.

These questions are meant to be sanity checks, not hard-and-fast rules. The main point is that **scenario outlines should focus on one behavior and use only the necessary variations**.

**Style and Structure**

While style often takes a backseat during [code review](https://automationpanda.com/2017/05/08/10-gotchas-for-automation-code-reviews/), it is a factor that differentiates good feature files from great feature files. In a truly behavior-driven team, non-technical stakeholders will rely upon feature files just as much as the engineers. **Good writing style improves communication**, and good communication skills are more than just resume fluff.

Below are a number of tidbits for good style and structure:

1. [Focus a feature on customer needs](https://automationpanda.com/2017/10/19/in-bdd-what-should-be-a-feature/).
2. Limit one feature per feature file. This makes it easy to find features.
3. Limit the number of scenarios per feature. Nobody wants a thousand-line feature file. A good measure is a dozen scenarios per feature.
4. Limit the number of steps per scenario to less than ten.
5. Limit the character length of each step. Common limits are 80-120 characters.
6. Use proper spelling.
7. Use proper grammar.
8. Capitalize Gherkin keywords.
9. Capitalize the first word in titles.
10. Do not capitalize words in the step phrases unless they are proper nouns.
11. Do not use punctuation (specifically periods and commas) at the end of step phrases.
12. Use single spaces between words.
13. Indent the content beneath every section header.
14. Separate features and scenarios by two blank lines.
15. Separate examples tables by 1 blank line.
16. Do not separate steps within a scenario by blank lines.
17. Space table delimiter pipes (“|”) evenly.
18. Adopt a standard set of tag names. Avoid duplicates.
19. Write all tag names in lowercase, and use hyphens (“-“) to separate words.
20. Limit the length of tag names.

Without these rules, you might end up with something like this:

*# BAD EXAMPLE! Do not copy.*

**Feature**: Google Searching

***@AUTOMATE @Automated @automation @Sprint32GoogleSearchFeature***

**Scenario outline**: GOOGLE STUFF

**Given** a Web Browser is on the Google page,

**when** The seach phrase "<phrase>" Enter,

**Then** "<phrase>" shown.

**and** The relatedd results include "<related>".

**Examples**: animals

| phrase | related |

| panda | Panda Express |

| elephant | elephant Man |

Don’t do this. It looks horrible. Please, take pride in your profession. While the automation code may look hairy in parts, **Gherkin files should look elegant**.

**Gherkinize Those Behaviors!**

With these best practices, you can write Gherkin feature files like a pro. Don’t be afraid to try: nobody does things perfectly the first time. As a beginner, I broke many of the guidelines I put in this post, but I learned as I went. Don’t give up if you get stuck. Always remember the **Golden Gherkin Rule** and the **Cardinal Rule of BDD**!

This is the last of three posts in the series focused exclusively on Gherkin. The next post will address how to adopt behavior-driven practices into the Agile software development process.

**Share this:**

DESIGN PATTERN

**Task 01\_Hometask\_Singleton\_pattern**

To implement the Singleton pattern in the CandyMaker class, we will ensure that the class can be instantiated only once and provide a global access point to that instance. Additionally, we'll make the class thread-safe to prevent concurrent access issues.

Here's how you can do it:

**Singleton Implementation with Thread Safety**

1. **Private Static Instance**: We'll add a private static instance of the CandyMaker class.
2. **Private Constructor**: The constructor will remain private to prevent direct instantiation.
3. **Thread-Safe Access**: We'll use synchronized access to ensure that only one thread can create the instance at a time.

**Updated CandyMaker Class**

java

Copy code

public class CandyMaker {

private static CandyMaker instance; // Step 1: Private static instance

private boolean empty;

private boolean boiled;

// Private constructor to prevent instantiation

private CandyMaker() {

empty = true;

boiled = false;

}

// Thread-safe method to get the single instance of CandyMaker

public static synchronized CandyMaker getInstance() {

if (instance == null) { // Lazy initialization

instance = new CandyMaker();

}

return instance;

}

// This code is only started when the boiler is empty

public void fill() {

if (isEmpty()) {

empty = false;

boiled = false;

// fill the candyMaker with milk and chocolate mix

System.out.println("Filling the candy maker with milk and chocolate mix.");

} else {

System.out.println("Cannot fill: the boiler is not empty.");

}

}

// To drain the boiler, it must be full (non-empty) and also boiled.

// Once it is drained, we set the empty back to true

public void drain() {

if (!isEmpty() && isBoiled()) {

// drain the boiled milk and chocolate

empty = true;

System.out.println("Draining the boiled milk and chocolate.");

} else {

System.out.println("Cannot drain: either the boiler is empty or not boiled.");

}

}

// To boil the mixture the boiler has to be full and not already boiled.

// Once it's boiled we set the boiled flag to true

public void boil() {

if (!isEmpty() && !isBoiled()) {

// bring the content to boil

boiled = true;

System.out.println("Boiling the milk and chocolate mixture.");

} else {

System.out.println("Cannot boil: either the boiler is empty or already boiled.");

}

}

public boolean isEmpty() {

return empty;

}

public boolean isBoiled() {

return boiled;

}

}

// Test class to demonstrate Singleton behavior and CandyMaker functionality

public class CandyMakerTest {

public static void main(String[] args) {

// Get the single instance of CandyMaker

CandyMaker candyMaker = CandyMaker.getInstance();

// Test the functionality

candyMaker.fill();

candyMaker.boil();

candyMaker.drain();

// Attempt to create another instance

CandyMaker anotherCandyMaker = CandyMaker.getInstance();

System.out.println("CandyMaker Instance: " + candyMaker);

System.out.println("Another CandyMaker Instance: " + anotherCandyMaker);

System.out.println("Are both instances the same? " + (candyMaker == anotherCandyMaker));

}

}

**Explanation**

1. **Singleton Instance**: The static variable instance is used to hold the only instance of CandyMaker.
2. **Private Constructor**: The constructor is private, preventing any external instantiation of the class.
3. **Thread-Safe Access**: The getInstance() method is synchronized, ensuring that only one thread can access it at a time. If multiple threads attempt to call this method simultaneously, only one will create the instance, while others will wait.
4. **CandyMaker Functionality**: The methods fill(), boil(), and drain() control the state of the candy-making process based on the conditions defined in the original code.
5. **Test Class**: The CandyMakerTest class demonstrates how to get the singleton instance and invoke its methods. It also shows that multiple calls to getInstance() return the same instance.

**Conclusion**

With this implementation, you now have a thread-safe Singleton class that manages the state of a candy-making process. You can further extend this class with additional functionality as needed.

**Task 02\_Hometask\_Factory\_pattern**

Factory pattern Task:

SubTask-1

Using factory pattern, write code to initialise it to your custom chrome, firefox or IE browser implementations.

Note: do not just say new ChromeDriver. Have a class something like myChromeDriver and

just do initialize the actual chrome driver in this calss.

similary for other browser classes .

SubTask-2

Now make it Fit to the GOF Factory Method pattern. As per the Gof Factory Method Pattern, the subclasses should decide which class to instantiate.

So have another set of implementations for chrome, firefox, IE browsers.

This customization is different from previous by having some custom caabilities set in these classes.

For example: Have another class from chrome driver and this time call it "MyChromeDriverWithCapabilities". Similar for other browsers

Declare factory classes, one which return normal chormdriver(MyChromeDriver) and other which return driver with pre-set capabilities("MyChromeDriverWithCapabilities")

To implement the Factory Pattern and the Factory Method Pattern as per your requirements, we will break it down into two main parts:

**SubTask-1: Basic Factory Pattern Implementation**

1. **Create Custom Driver Classes**: We'll create custom implementations for Chrome, Firefox, and IE browsers.
2. **Create a Factory Class**: This class will instantiate the desired browser driver based on the input.

**SubTask-2: GOF Factory Method Pattern**

1. **Create Driver Classes with Capabilities**: We will have different driver classes that set custom capabilities.
2. **Implement Factory Method Pattern**: We will create factories that will return either the basic drivers or the drivers with custom capabilities.

**Implementation**

**Step 1: Custom Driver Classes**

java

Copy code

// Base interface for browser drivers

public interface BrowserDriver {

void launch();

}

// Custom Chrome Driver

public class MyChromeDriver implements BrowserDriver {

@Override

public void launch() {

System.out.println("Launching My Chrome Driver...");

// Initialize actual ChromeDriver here

}

}

// Custom Firefox Driver

public class MyFirefoxDriver implements BrowserDriver {

@Override

public void launch() {

System.out.println("Launching My Firefox Driver...");

// Initialize actual FirefoxDriver here

}

}

// Custom IE Driver

public class MyIEDriver implements BrowserDriver {

@Override

public void launch() {

System.out.println("Launching My IE Driver...");

// Initialize actual IEDriver here

}

}

**Step 2: Basic Factory Class**

java

Copy code

// Factory class to create browser drivers

public class BrowserDriverFactory {

public static BrowserDriver getBrowserDriver(String browserType) {

if (browserType.equalsIgnoreCase("chrome")) {

return new MyChromeDriver();

} else if (browserType.equalsIgnoreCase("firefox")) {

return new MyFirefoxDriver();

} else if (browserType.equalsIgnoreCase("ie")) {

return new MyIEDriver();

}

return null;

}

}

**Step 3: Custom Driver Classes with Capabilities**

java

Copy code

// Custom Chrome Driver with capabilities

public class MyChromeDriverWithCapabilities implements BrowserDriver {

@Override

public void launch() {

System.out.println("Launching My Chrome Driver with custom capabilities...");

// Initialize actual ChromeDriver with custom capabilities here

}

}

// Custom Firefox Driver with capabilities

public class MyFirefoxDriverWithCapabilities implements BrowserDriver {

@Override

public void launch() {

System.out.println("Launching My Firefox Driver with custom capabilities...");

// Initialize actual FirefoxDriver with custom capabilities here

}

}

// Custom IE Driver with capabilities

public class MyIEDriverWithCapabilities implements BrowserDriver {

@Override

public void launch() {

System.out.println("Launching My IE Driver with custom capabilities...");

// Initialize actual IEDriver with custom capabilities here

}

}

**Step 4: Factory Method Classes**

java

Copy code

// Factory for basic drivers

public class BasicBrowserDriverFactory {

public static BrowserDriver getDriver(String browserType) {

return BrowserDriverFactory.getBrowserDriver(browserType);

}

}

// Factory for drivers with capabilities

public class CapabilityBrowserDriverFactory {

public static BrowserDriver getDriverWithCapabilities(String browserType) {

if (browserType.equalsIgnoreCase("chrome")) {

return new MyChromeDriverWithCapabilities();

} else if (browserType.equalsIgnoreCase("firefox")) {

return new MyFirefoxDriverWithCapabilities();

} else if (browserType.equalsIgnoreCase("ie")) {

return new MyIEDriverWithCapabilities();

}

return null;

}

}

**Step 5: Testing the Implementation**

java

Copy code

public class Main {

public static void main(String[] args) {

// SubTask-1: Using the Basic Factory Pattern

BrowserDriver chromeDriver = BasicBrowserDriverFactory.getDriver("chrome");

chromeDriver.launch();

BrowserDriver firefoxDriver = BasicBrowserDriverFactory.getDriver("firefox");

firefoxDriver.launch();

// SubTask-2: Using the Factory Method Pattern

BrowserDriver chromeDriverWithCapabilities = CapabilityBrowserDriverFactory.getDriverWithCapabilities("chrome");

chromeDriverWithCapabilities.launch();

BrowserDriver firefoxDriverWithCapabilities = CapabilityBrowserDriverFactory.getDriverWithCapabilities("firefox");

firefoxDriverWithCapabilities.launch();

}

}

**Summary**

* **SubTask-1**: We created custom driver classes (MyChromeDriver, MyFirefoxDriver, MyIEDriver) and a factory class to return instances based on the browser type.
* **SubTask-2**: We extended the implementation by creating additional driver classes with custom capabilities and a separate factory class to handle the instantiation of these drivers.

This structure adheres to the Factory Pattern and Factory Method Pattern principles, allowing for easy extension and maintenance of driver implementations.

**Task 03\_Hometask\_Strategy\_pattern**

To design the PhoneCameraApp in a way that is resistant to changes, particularly in the sharing functionality, we can apply the **Strategy Pattern**. This pattern allows us to encapsulate the sharing algorithm into separate classes, making it easy to add new sharing methods without modifying the existing code.

**Design Overview**

1. **Define a Sharing Strategy Interface**: Create an interface for sharing strategies that defines the method for sharing photos.
2. **Implement Concrete Sharing Strategies**: Create classes that implement the sharing strategy interface for each sharing method (e.g., Text, Email, Social Media).
3. **Create an Abstract Base Class for Camera Apps**: Define a base class for PhoneCameraApp that contains shared functionality.
4. **Implement Specific Camera Applications**: Extend the base class to create BasicCameraApp and CameraPlusApp.
5. **Allow Dynamic Sharing Strategy**: Include a mechanism to set or change the sharing strategy at runtime, enabling the addition of new sharing methods without altering existing code.

**Implementation**

Here's a possible implementation in Java:

**Step 1: Define the Sharing Strategy Interface**

java

Copy code

// Strategy interface

public interface SharingStrategy {

void sharePhoto(String photo);

}

**Step 2: Implement Concrete Sharing Strategies**

java

Copy code

// Concrete strategy for sharing via Text

public class TextSharingStrategy implements SharingStrategy {

@Override

public void sharePhoto(String photo) {

System.out.println("Sharing photo via Text: " + photo);

}

}

// Concrete strategy for sharing via Email

public class EmailSharingStrategy implements SharingStrategy {

@Override

public void sharePhoto(String photo) {

System.out.println("Sharing photo via Email: " + photo);

}

}

// Concrete strategy for sharing via Social Media

public class SocialMediaSharingStrategy implements SharingStrategy {

@Override

public void sharePhoto(String photo) {

System.out.println("Sharing photo on Social Media: " + photo);

}

}

**Step 3: Create an Abstract Base Class for Camera Apps**

java

Copy code

// Base class for Camera Apps

public abstract class PhoneCameraApp {

protected SharingStrategy sharingStrategy;

public void setSharingStrategy(SharingStrategy sharingStrategy) {

this.sharingStrategy = sharingStrategy;

}

public void takePhoto() {

System.out.println("Taking photo...");

}

public void editPhoto() {

System.out.println("Editing photo...");

}

public void savePhoto() {

System.out.println("Saving photo...");

}

public void sharePhoto(String photo) {

if (sharingStrategy != null) {

sharingStrategy.sharePhoto(photo);

} else {

System.out.println("No sharing strategy set.");

}

}

}

**Step 4: Implement Specific Camera Applications**

java

Copy code

// Basic Camera Application

public class BasicCameraApp extends PhoneCameraApp {

@Override

public void editPhoto() {

System.out.println("Basic Camera App - Editing photo...");

// Specialized editing logic

}

}

// Camera Plus Application

public class CameraPlusApp extends PhoneCameraApp {

@Override

public void editPhoto() {

System.out.println("Camera Plus App - Editing photo with advanced features...");

// Specialized editing logic

}

}

**Step 5: Using the Application**

java

Copy code

public class Main {

public static void main(String[] args) {

// Create a BasicCameraApp

PhoneCameraApp basicCameraApp = new BasicCameraApp();

// Set a sharing strategy

basicCameraApp.setSharingStrategy(new TextSharingStrategy());

basicCameraApp.takePhoto();

basicCameraApp.editPhoto();

basicCameraApp.savePhoto();

basicCameraApp.sharePhoto("MyPhoto.jpg");

// Change the sharing strategy to Email

basicCameraApp.setSharingStrategy(new EmailSharingStrategy());

basicCameraApp.sharePhoto("MyPhoto.jpg");

// Create a CameraPlusApp

PhoneCameraApp cameraPlusApp = new CameraPlusApp();

// Set the sharing strategy to Social Media

cameraPlusApp.setSharingStrategy(new SocialMediaSharingStrategy());

cameraPlusApp.takePhoto();

cameraPlusApp.editPhoto();

cameraPlusApp.savePhoto();

cameraPlusApp.sharePhoto("MyPhoto.jpg");

// Dynamically changing the strategy

cameraPlusApp.setSharingStrategy(new TextSharingStrategy());

cameraPlusApp.sharePhoto("MyPhoto.jpg");

}

}

**Summary**

* **Encapsulation of Sharing Logic**: By using the Strategy Pattern, we encapsulate the sharing logic into separate classes, making it easy to add new sharing methods without modifying the PhoneCameraApp.
* **Flexibility**: Users can change the sharing strategy at runtime, allowing them to define "how to share" without needing to update the app.
* **Code Reuse**: Shared functionalities like taking, editing, and saving photos are maintained in the base class, reducing code duplication between different camera applications.

This design will help keep the PhoneCameraApp flexible and maintainable, allowing for easy adaptation to new trends in sharing photos.

**Task 04\_Hometask\_Decorator\_pattern**

To represent the model for a webpage that can have different formats and widgets, we can use a combination of classes to define the Page, Widget, and their associated properties, including rankings.

**Code Implementation**

1. **Widget Class**: Represents an individual widget with a rank.
2. **Page Class**: Represents the webpage with different formats (like mobile and desktop) and manages the widgets added to it, along with the functionality to calculate the total rank of the page.

**Step-by-Step Code Implementation**

**Step 1: Create the Widget Class**

java

Copy code

// Widget class representing individual widgets

public class Widget {

private String name;

private int rank;

public Widget(String name, int rank) {

this.name = name;

this.rank = rank;

}

public String getName() {

return name;

}

public int getRank() {

return rank;

}

}

**Step 2: Create the Page Class**

java

Copy code

import java.util.ArrayList;

import java.util.List;

// Page class representing the webpage

public class Page {

private String format; // e.g., "Mobile" or "Desktop"

private List<Widget> widgets;

public Page(String format) {

this.format = format;

this.widgets = new ArrayList<>();

}

public void addWidget(Widget widget) {

widgets.add(widget);

}

public int calculateTotalRank() {

int totalRank = 0;

// Add the rank of each widget to the total rank

for (Widget widget : widgets) {

totalRank += widget.getRank();

}

return totalRank;

}

public String getFormat() {

return format;

}

public List<Widget> getWidgets() {

return widgets;

}

}

**Step 3: Using the Page and Widget Classes**

java

Copy code

public class WebPageRanking {

public static void main(String[] args) {

// Create some widgets

Widget widget1 = new Widget("Image Slider", 5);

Widget widget2 = new Widget("Contact Form", 3);

Widget widget3 = new Widget("Newsletter Signup", 4);

// Create a mobile page

Page mobilePage = new Page("Mobile");

mobilePage.addWidget(widget1);

mobilePage.addWidget(widget2);

// Create a desktop page

Page desktopPage = new Page("Desktop");

desktopPage.addWidget(widget1);

desktopPage.addWidget(widget3);

// Calculate and print the total rank for both pages

System.out.println("Total rank of " + mobilePage.getFormat() + " page: " + mobilePage.calculateTotalRank());

System.out.println("Total rank of " + desktopPage.getFormat() + " page: " + desktopPage.calculateTotalRank());

}

}

**Output**

mathematica

Copy code

Total rank of Mobile page: 8

Total rank of Desktop page: 9

**Summary**

* **Widgets**: Each widget has a name and a rank associated with it.
* **Page**: The Page class maintains a list of widgets and calculates the total rank by summing the ranks of all the widgets added to it.
* **Scalability**: You can easily add more widgets or page formats without changing the existing code structure, making it flexible and extensible.

This implementation allows you to represent a webpage and its associated functionalities, making it easy to manage and calculate the page ranking based on the widgets included.

**Task 05\_Hometask\_Builder\_pattern**

To represent a bicycle manufacturing model where customers can choose optional features like gears, stands, seats, and carriers, we can use a builder pattern. The builder pattern is a design pattern that allows you to construct complex objects step by step. In this case, we'll create a Bicycle class that encapsulates the properties of the bicycle, and a BicycleBuilder that provides a fluent interface for building the bicycle.

**Code Implementation**

1. **Bicycle Class**: Represents the bicycle with optional features.
2. **BicycleBuilder Class**: Provides methods to select optional features while ensuring that once a feature is selected, it cannot be changed.

**Step-by-Step Code Implementation**

**Step 1: Create the Bicycle Class**

java

Copy code

// Bicycle class representing the final bicycle product

public class Bicycle {

private final boolean hasGears;

private final boolean hasDoubleStands;

private final boolean hasDoubleSeats;

private final boolean hasCarrier;

private Bicycle(BicycleBuilder builder) {

this.hasGears = builder.hasGears;

this.hasDoubleStands = builder.hasDoubleStands;

this.hasDoubleSeats = builder.hasDoubleSeats;

this.hasCarrier = builder.hasCarrier;

}

@Override

public String toString() {

return "Bicycle{" +

"hasGears=" + hasGears +

", hasDoubleStands=" + hasDoubleStands +

", hasDoubleSeats=" + hasDoubleSeats +

", hasCarrier=" + hasCarrier +

'}';

}

// BicycleBuilder nested class

public static class BicycleBuilder {

private boolean hasGears;

private boolean hasDoubleStands;

private boolean hasDoubleSeats;

private boolean hasCarrier;

public BicycleBuilder() {

// Default values

this.hasGears = false;

this.hasDoubleStands = false;

this.hasDoubleSeats = false;

this.hasCarrier = false;

}

public BicycleBuilder addGears() {

this.hasGears = true;

return this;

}

public BicycleBuilder addDoubleStands() {

this.hasDoubleStands = true;

return this;

}

public BicycleBuilder addDoubleSeats() {

this.hasDoubleSeats = true;

return this;

}

public BicycleBuilder addCarrier() {

this.hasCarrier = true;

return this;

}

public Bicycle build() {

return new Bicycle(this);

}

}

}

**Step 2: Using the Bicycle Class and Builder**

java

Copy code

public class BicycleBuilderDemo {

public static void main(String[] args) {

// Building a bicycle with selected features

Bicycle bicycle1 = new Bicycle.BicycleBuilder()

.addGears()

.addDoubleSeats()

.build();

Bicycle bicycle2 = new Bicycle.BicycleBuilder()

.addDoubleStands()

.addCarrier()

.build();

// Printing the details of the bicycles

System.out.println(bicycle1);

System.out.println(bicycle2);

}

}

**Output**

arduino

Copy code

Bicycle{hasGears=true, hasDoubleStands=false, hasDoubleSeats=true, hasCarrier=false}

Bicycle{hasGears=false, hasDoubleStands=true, hasDoubleSeats=false, hasCarrier=true}

**Summary**

* **Bicycle Class**: Represents the final product with properties for each optional feature. The properties are final and can only be set through the builder, ensuring immutability.
* **BicycleBuilder Class**: Allows customers to add features to the bicycle. Once the build() method is called, the bicycle is created, and the state of the builder cannot be changed.
* **Flexibility**: This implementation provides flexibility for customers to select optional features while ensuring that the choices cannot be changed after the bicycle is built.

This pattern allows for a clean and clear way to construct complex objects like a bicycle with various optional features.

Identifying automation Candidate

**How to Choose Which Test to Automate?**

[**testing**](https://devqa.io/testing)

**→**[**#automation**](https://devqa.io/tag/automation)[**#tips**](https://devqa.io/tag/tips)

How do you choose which tests to automate and which tests to leave for manual testing?

Before you start automating a test, you need to see what benefits you get by automating the test after you factor in the time, effort and resource invested in test automation.

Below are some factors to consider to help identify which manual tests should or should not be automated. As the old saying goes, just because you can automate something doesn’t necessarily mean that you should.

Here are some guidelines to help identify good candidates for test automation:

**Tests that should be automated:**

* Business critical paths - the features or user flows that if they fail, cause a considerable damage to the business.
* Tests that need to be run against every build/release of the application, such as smoke test, sanity test and regression test.
* Tests that need to run against multiple configurations — different OS & Browser combinations.
* Tests that execute the same workflow but use different data for its inputs for each test run e.g. data-driven.
* Tests that involve inputting large volumes of data, such as filling up very long forms.
* Tests that can be used for performance testing, like stress and load tests.
* Tests that take a long time to perform and may need to be run during breaks or overnight.
* Tests during which images must be captured to prove that the application behaved as expected, or to check that a multitude of web pages looks the same on multiple browsers.

*Generally speaking, the more repetitive the test run, the better it is for automation.*

*Also remember that tests are not the only candidates for automation.****Tasks****such as setting up or creating test data for manual exploratory testing are also great candidates for automation.*

**Tests that should not be automated:**

* Tests that you will only run only once. The only exception to this rule is that if you want to execute a test with a very large set of data, even if it’s only once, then it makes sense to automate it.
* User experience tests for usability (tests that require a user to respond as to how easy the app is to use).
* Tests that need to be run ASAP. Usually, a new feature which is developed requires a quick feedback so testing it manually at first
* Tests that require ad hoc/random testing based on domain knowledge/expertise - Exploratory Testing.
* Intermittent tests. Tests without predictable results cause more noise that value. To get the best value out of automation the tests must produce predictable and reliable results in order to produce pass and fail conditions.
* Tests that require visual confirmation, however, we can capture page images during automated testing and then have a manual check of the images.
* Test that cannot be 100% automated should not be automated at all, unless doing so will save a considerable amount of time.

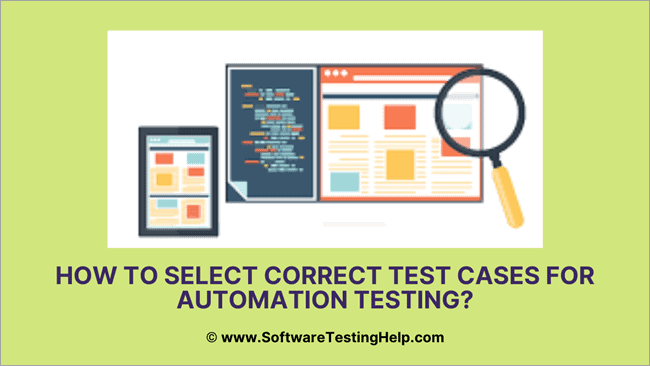
**How to Select Correct Test Cases for Automation Testing (and Ultimately Achieve a Positive Automation ROI)**

By [**Vijay**](https://www.softwaretestinghelp.com/author/vijay/)  Updated March 3, 2024

**Now is the era of Test Automation**. Most of the testing projects are trying to translate their manual test cases to automated ones to improve productivity and coverage.

One of the key steps to commence Automation Testing is – selecting the appropriate test cases and determining the ROI (Return On Investment).

**What to expect from this article?**  
In this article, I have tried to cite down few important points based on my experience to help you select the correct candidate forautomation and determine various other factors that will yield better test results and benefits.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2014/03/How-to-Select-Correct-Test-Cases-for-Automation-Testing.png)

**Why Automated Testing?**

Automation does not overpower or replaces Manual Testing but it compliments it. Like Manual, Automation needs a strategy with proper planning, monitoring & control. Automation, when implemented correctly, can become an asset to the team, project and ultimately to the organization.

**There are many advantages of Automation; here are few important to mention:**

* Useful to execute routine tasks like [Smoke tests](https://www.softwaretestinghelp.com/smoke-testing-and-sanity-testing-difference/) and [Regression tests](https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/).
* Useful in preparing the [Test data](https://www.softwaretestinghelp.com/tips-to-design-test-data-before-executing-your-test-cases/).
* Helps to execute the [Test cases which involve complex business logic](https://www.softwaretestinghelp.com/decision-table-test-case-design-technique/).
* Good to execute the cross-platform test cases (like different OS, browsers, etc.)
* Great to execute the test cases which are a bit difficult to execute manually.
* When the number of iterations of the test case executions is not known.

Many a time stakeholders feel that test automation acts as a support tool for Manual Testing, so it’s vital to understand that automation is the best way to increase the effectiveness, efficiency, and coverage of testing. It not only saves time but also improves accuracy as repetitive tasks via the manual approach can prone to human errors and can be time-consuming.

**Candidates For Automation**

**Basic Mistake to Avoid:**  
One of the most basic mistakes which testers make is NOT Selecting the correct test cases for automation.

Don’t just select any test suite. Analyze the test cases thoroughly and select the candidates for automation considering the most important factor i.e. ROI. First, we should understand and find out the ways for getting a higher and positive ROI.

*(****ROI – Return On Investment****– It is a calculation of benefits in terms of cost-saving, increased efficiency, and quality)*

There is no standard procedure for determining the correct test cases for automation. It all depends on the application you are testing.

***Based on my experience, I have tried to chalk down few steps which may provide some insight for selecting the test cases and ultimately progress to achieve a positive ROI for automation.***

**See Also = >**[**How to Translate Manual Test Cases into Automation Scripts?**](https://www.softwaretestinghelp.com/how-to-translate-manual-test-cases-into-automation-scripts/)

**Table of Contents:**[[Show](https://www.softwaretestinghelp.com/manual-to-automation-testing-process-challenges/)]

**How To Select Correct Test Cases For Automation Testing**

**Step 1:**

Identify the parameters on which you will base your test case as a candidate for automation.

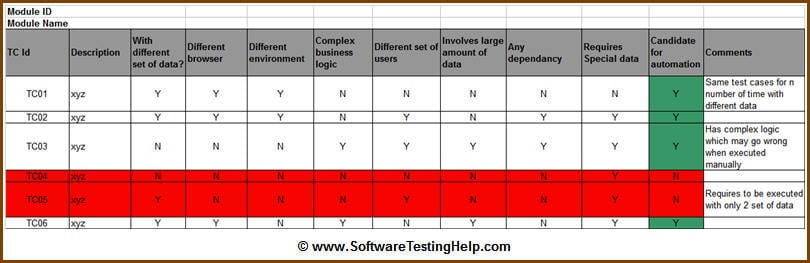
As of now, I am identifying the below parameters, you can have your own parameters depending on your application.

* Test case executed with different sets of data.
* Test case executed with different browsers.
* Test case executed with different environments.
* Test case executed with complex business logic
* Test case executed with a different set of users
* Test case involves a large amount of data
* Test case has any dependency
* Test case requires Special data

**Step 2:**

Break each application into modules. For each module, analyze and try to identify the test cases which should be automated based on the parameters. This list will vary for projects to projects and can also be enhanced to suit your needs:

**Fig 1.0**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2014/03/Manual-to-automation-testing.jpg)

Y – Yes  
N – No

In a similar way, for all modules, this list can be used to identify the automation candidate test cases.

**Step 3**:

Consolidate and group the number of test cases for each module shown below

**Fig 2.0**

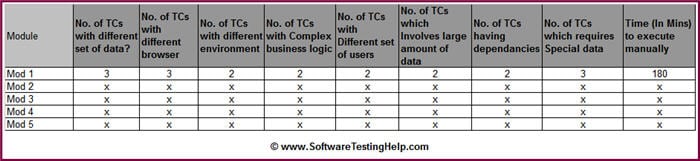
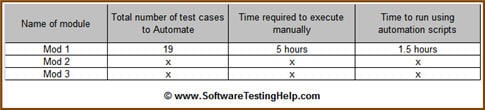
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2014/03/Manual-to-automation-testing-1.jpg)

Figure 2.0 is quite straightforward and self-explanatory. Here I am trying to quantify the details and giving an estimate to finish the testing manually.

**Step 4:**

Once you have identified all the granular level details, you can present them in the below way. We are now progressing to calculate the ROI.

**Fig 3.0:**

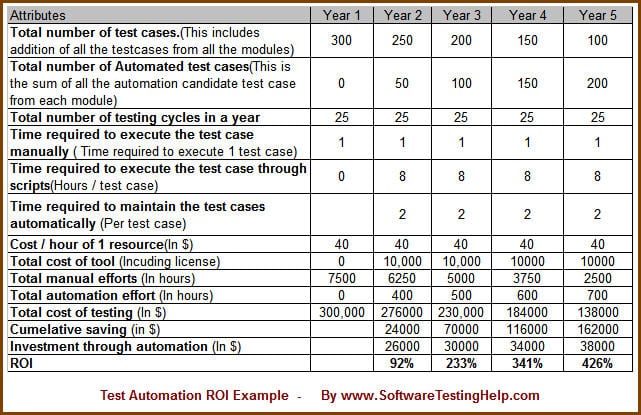
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2014/03/Manual-to-automation-testing-2.jpg)

We should also take into account the below **attributes which form the basis of deterring the ROI**:

* Purchasing and licensing cost of the tool
* Time to develop the scripts
* Time to maintain the scripts.
* Time to analyze the results manually and automatically
* Time and cost to train the resources.
* Management overheads

**Test Automation ROI Calculation Example**

In most of the cases, ROI is calculated for 5 years, but it’s not mandatory. Based on the above factors, let me try to elaborate on the calculation of ROI for 5 years. As usual, you can always tailor and enhance it.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2014/03/Manual-to-automation-testing-3.jpg)

**\*ROI = (Cumulative saving / Investment through automation)\*100**

**Manual To Automation Testing – What Are The Process Challenges?**

**I have tried to cite the points, which I feel, are a big challenge, when we try to automate a test suite.**

**#1) Automation Need:** Every test team is unique and has an exclusive need for automation. We cannot develop a fixed standard, but we can tailor a standard which will suit our need.  Because of this reason, automation does require good support from the management as well as from the development team.

**#2) Automating the complete application:**Automating a 100% application is a big task. Not that it is impossible, but it requires proper planning and monitoring and of course; some time. There are lots of permutations and combinations of data, n number of environments with n number of authentication and authorization attributes that need to be validated and hence require a strategy to automate.

**#3) Manual Vs Automation mentality:** “**We normally automate which is important and repetitive, but we prefer to test the important functionality manually**”. Confused? Even I am!! But this is a fact. We should have criteria which will decide which are the **important** test cases. These criteria can be based on multiple factors like complex business logic, areas that are of more interest to clients, risk-prone areas, etc.

**#4) Deciding on the framework:** [Designing the framework](https://www.softwaretestinghelp.com/qtp-tutorial-17-test-automation-frameworks/) is the most important facet of automation. I believe we should devote relatively more time to develop the framework than to script. Whenever we develop the automation plan, framework designing should be the main focus.

PLAN to design the framework. Identify and make a checklist of the items which will form the framework. If the framework is rock solid, scripting and maintaining become easy.

**#5) Knowledge of the team:** Whenever we think of automation, we immediately jump to learn the programming language or scripting language. Learning this language will definitely help but more emphasis should be on building and developing logic.

Automation should not be the responsibility of some handful of resources, rather an entire team should contribute towards it. This will help not only to enhance the skills of the resources but also [keep them motivated](https://www.softwaretestinghelp.com/how-to-keep-motivation-alive-in-software-testers/).

**#6) Reporting:** Every tool has a standard to report the test results. To customize it; is a challenging task. Reporting the test results also requires coordination and maintenance which adds to the cost.

**#7) Trust:** We should trust our automation stuff. We invest man-hours to build an automation suite but still, we don’t believe in the test results. Efforts should be given to maintaining the scripts. Also, we should see that the team who is doing the manual testing of the application, should be involved to automate it as they know their application.

Most of the time, a third-team does the automation so the actual testing team is not aware of the scripts and finally end up with running the tests manually because they feel follow up on the scripts,  add up to their tasks

ROI   
  
To address your questions regarding the implementation of automation testing in your current project (SUT), let’s break it down step by step.

**1. Should we implement automation on the project? Why?**

Yes, we should implement automation for the following reasons:

* **Efficiency**: Automation significantly speeds up the testing process compared to manual testing. Tests can be executed much faster, especially for regression testing.
* **Consistency**: Automated tests can be run consistently and repeatedly without human error, ensuring that the tests are executed in the same manner each time.
* **Cost-Effectiveness**: While there is an upfront investment in creating automated tests, over time, it can be more cost-effective compared to extensive manual testing, especially in large projects with frequent releases.
* **Coverage**: Automation allows for more extensive test coverage. Tests can be run on multiple configurations and environments that may not be feasible with manual testing.
* **Immediate Feedback**: Automated tests can provide immediate feedback to developers, which is essential for maintaining quality in continuous integration and delivery environments.

**2. What can we automate? Why?**

We can automate the following:

* **Regression Tests**: These are prime candidates for automation since they need to be executed frequently. Automated regression tests ensure that new changes do not break existing functionality.
* **Smoke Tests**: Automating smoke tests can quickly verify if the application is stable enough for further testing.
* **API Testing**: APIs can be tested more efficiently through automation, enabling us to quickly validate functionality and performance.
* **Performance Testing**: Tools can simulate multiple users and monitor performance metrics, which is difficult to achieve manually.
* **Data-Driven Tests**: If the application allows for multiple input sets, these tests can be automated to verify different scenarios without requiring manual intervention.

**3. Calculate ROI (Return on Investment) based on manual and automated testing price**

Assuming you have some data, let's say:

* **Manual Testing Cost**:
  + 20 man-hours per week \* 5 years \* 52 weeks = 5200 man-hours
* **Automated Tests Development Cost**:
  + Framework setup: 80 man-hours
  + Scenario development: 40 man-hours per week \* 0.5 years \* 52 weeks = 1040 man-hours
  + Total automated tests development cost = 80 + 1040 = 1120 man-hours
* **Automated Test Execution & Result Analysis**:
  + 8 man-hours per week \* 5 years \* 52 weeks = 2080 man-hours

**ROI Calculation Steps**

1. **Calculate Costs**:
   * Cost of Manual Testing = 5200 man-hours
   * Cost of Automation = 1120 man-hours (development) + 2080 man-hours (execution) = 3200 man-hours
2. **Calculate Net Profit**:

Net Profit=Cost of Manual Testing−Cost of Automation\text{Net Profit} = \text{Cost of Manual Testing} - \text{Cost of Automation}Net Profit=Cost of Manual Testing−Cost of Automation Net Profit=5200−3200=2000 man-hours\text{Net Profit} = 5200 - 3200 = 2000 \text{ man-hours}Net Profit=5200−3200=2000 man-hours

1. **Calculate ROI**:

ROI=Net ProfitCost of Automation=20003200=0.625 or 62.5%ROI = \frac{\text{Net Profit}}{\text{Cost of Automation}} = \frac{2000}{3200} = 0.625 \text{ or } 62.5\%ROI=Cost of AutomationNet Profit​=32002000​=0.625 or 62.5%

**Summary**

* **Automation Implementation**: Recommended due to increased efficiency, consistency, cost-effectiveness, coverage, and immediate feedback.
* **Automation Targets**: Focus on regression, smoke tests, API testing, performance testing, and data-driven tests.
* **ROI Calculation**: The calculated ROI based on the provided costs shows a positive return on investment, indicating that automation is beneficial in the long term.

This analysis provides a clear rationale for implementing automation in your project, along with a quantifiable metric to evaluate its effectiveness.

4o mini

GIT

To complete the tasks outlined for using Git, follow these steps. You’ll need to have Git installed and a Git account on your organization’s Git server (in this case, git.epam.com or github.com). Here’s a step-by-step guide:

**Step 1: Create a Git Repository**

1. **Log in** to your Git server (git.epam.com/github.com).
2. **Create a new repository**:
   * Navigate to the appropriate section to create a new repository (usually found in your profile or dashboard).
   * Name your repository (e.g., MyTestRepo).
   * Set the visibility (public/private as required).
   * Initialize with a README if needed.

**Step 2: Clone Using SSH Keys**

1. **Ensure your SSH keys are set up**:
   * Check for existing SSH keys by running ls -al ~/.ssh in your terminal.
   * If you don't have SSH keys, generate them with:

bash

Copy code

ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

* + Add the public key (~/.ssh/id\_rsa.pub) to your Git server account under SSH keys.

1. **Clone the repository**:

bash

Copy code

git clone git@<git.epam.com/github.com>:<your\_username>/MyTestRepo.git

cd MyTestRepo

**Step 3: Push Few Commits in Master Branch**

1. **Make changes to files** (e.g., update the README).
2. **Stage and commit the changes**:

bash

Copy code

git add README.md # or any modified files

git commit -m "Initial commit with README"

1. **Push the commits to the master branch**:

bash

Copy code

git push origin master

**Step 4: Create New Test Branch and Push 2 Commits Inside**

1. **Create and switch to a new branch**:

bash

Copy code

git checkout -b test

1. **Make changes and commit twice**:

bash

Copy code

echo "First change in test branch" >> testfile.txt

git add testfile.txt

git commit -m "First commit in test branch"

echo "Second change in test branch" >> testfile.txt

git add testfile.txt

git commit -m "Second commit in test branch"

1. **Push the test branch**:

bash

Copy code

git push origin test

**Step 5: Push One New Commit in Master Branch**

1. **Switch back to master**:

bash

Copy code

git checkout master

1. **Make changes and commit**:

bash

Copy code

echo "New commit in master" >> masterfile.txt

git add masterfile.txt

git commit -m "New commit in master"

1. **Push to master**:

bash

Copy code

git push origin master

**Step 6: Merge Test to Master**

1. **Merge the test branch into master**:

bash

Copy code

git checkout master

git merge test

1. **Push the merged changes**:

bash

Copy code

git push origin master

**Step 7: Add 2 Additional Commits to Test, 1 Commit in Master. Rebase Test**

1. **Switch to test branch**:

bash

Copy code

git checkout test

1. **Add two new commits**:

bash

Copy code

echo "Third change in test branch" >> testfile.txt

git add testfile.txt

git commit -m "Third commit in test branch"

echo "Fourth change in test branch" >> testfile.txt

git add testfile.txt

git commit -m "Fourth commit in test branch"

1. **Switch back to master and add a new commit**:

bash

Copy code

git checkout master

echo "Another commit in master" >> masterfile.txt

git add masterfile.txt

git commit -m "Another commit in master"

1. **Rebase test branch with master**:

bash

Copy code

git checkout test

git rebase master

1. **Resolve any conflicts if they occur and continue the rebase**:

bash

Copy code

git rebase --continue

**Step 8: Add 2 Additional Commits in Test. Cherry-Pick First Commit to Master. Revert the Second Commit**

1. **Add two additional commits in test**:

bash

Copy code

echo "Fifth change in test branch" >> testfile.txt

git add testfile.txt

git commit -m "Fifth commit in test branch"

echo "Sixth change in test branch" >> testfile.txt

git add testfile.txt

git commit -m "Sixth commit in test branch"

1. **Cherry-pick the first commit from test to master**:

bash

Copy code

git checkout master

git cherry-pick <commit\_id\_of\_first\_commit\_in\_test> # Get commit ID using git log

1. **Revert the second commit in test branch**:

bash

Copy code

git checkout test

git revert <commit\_id\_of\_second\_commit\_in\_test>

**Final Step: Share the Link to the Repo**

1. After completing all steps and pushing your branches, you can share the repository link with your mentor.
   * The link will typically be: git.epam.com/github.com/<your\_username>/MyTestRepo

**Summary**

You have now completed a series of Git tasks, including branching, committing, merging, rebasing, cherry-picking, and reverting. Make sure to check your Git history (git log) at each stage to understand the changes made. If any issues arise, don't hesitate to consult Git documentation or seek help.

Jenkins

**Practice**

1. **Install Jenkins:**
   * Download the Jenkins installer from the official Jenkins website.
   * Follow the installation instructions for your operating system.
   * Start Jenkins and access it via http://localhost:8080.
2. **Open a GitHub Account and Create a Repository:**
   * Go to [GitHub](https://github.com/) and sign up for an account if you don’t have one.
   * Create a new repository by clicking on the "+" icon in the upper right corner and selecting "New repository."
   * Upload a shell script (for Unix/Linux) or a batch script (for Windows) to your repository.
3. **Configure a New Jenkins Job:**
   * In Jenkins, click on "New Item."
   * Choose "Freestyle project" and give it a name.
   * In the configuration, under the "Source Code Management" section, select "Git" and enter the URL of your GitHub repository.
   * Under the "Build" section, add a build step to execute your shell or batch script.
   * Save and build the job to ensure it pulls the code and runs the script.
4. **Implement a CI Workflow with a Jenkinsfile (Scripted Pipeline):**
   * Create a Jenkinsfile in your repository with a scripted pipeline definition. Example:

groovy

Copy code

node {

stage('Clone') {

git 'https://github.com/your\_username/your\_repository.git'

}

stage('Build') {

sh 'chmod +x your\_script.sh'

sh './your\_script.sh'

}

stage('Test') {

// Add your testing commands here

}

stage('Notify') {

// Add notification commands here

}

}

* + Create a new pipeline job in Jenkins pointing to your Jenkinsfile.

1. **Implement a CI Workflow with a Jenkinsfile (Declarative Pipeline):**
   * Create a Jenkinsfile with a declarative pipeline. Example:

groovy

Copy code

pipeline {

agent any

stages {

stage('Clone') {

steps {

git 'https://github.com/your\_username/your\_repository.git'

}

}

stage('Build') {

steps {

sh 'chmod +x your\_script.sh'

sh './your\_script.sh'

}

}

stage('Test') {

steps {

// Add your testing commands here

}

}

stage('Notify') {

steps {

// Add notification commands here

}

}

}

}

* + Create a new pipeline job in Jenkins pointing to your Jenkinsfile.

1. **Install Blue Ocean:**
   * In Jenkins, go to "Manage Jenkins" > "Manage Plugins" > "Available" tab.
   * Search for "Blue Ocean" and install it.
   * After installation, you can access Blue Ocean by clicking on the "Open Blue Ocean" link in the Jenkins dashboard.
   * Create a CI workflow for different branches using Blue Ocean.
2. **Implement a CD Workflow:**
   * In your Jenkinsfile, add a deployment stage after the testing stage. Example:

groovy

Copy code

stage('Deploy') {

steps {

// Commands to deploy your application (e.g., to a server)

}

}

* + Ensure that your CI/CD pipeline includes deployment steps.

1. **Create a Pipeline with Visual Editor:**
   * Open Blue Ocean, create a new pipeline, and use the visual editor to add stages.
   * Instead of dummy messages, configure the stages to build a web application or deploy it on localhost.

**Theory**

1. **Describe Jenkins:** Jenkins is an open-source automation server used to automate parts of the software development process related to building, testing, and deploying applications. It supports continuous integration (CI) and continuous delivery (CD) practices.
2. **Prerequisites for Installing Jenkins:**
   * Java Development Kit (JDK) installed (Jenkins requires Java to run).
   * Appropriate permissions to install software on your machine (admin rights).
3. **Why Install Jenkins LTS:** Jenkins LTS (Long Term Support) is favored because it provides a stable release that is tested extensively, ensuring fewer bugs and compatibility issues. The latest Jenkins release may have new features but can also introduce instability. LTS is more appropriate for production environments where stability is crucial.
4. **Difference Between a Job/Project and a Build:** A job/project in Jenkins is a defined process (like a pipeline) that specifies how to run a task, while a build is the actual execution of that job/project, producing results (like binaries or reports).
5. **What is a Workspace?** A workspace is a directory on the Jenkins server where a job's files are stored. Each job has its own workspace, which includes the checked-out source code and any artifacts generated during the build process.
6. **What is a Build Trigger?** A build trigger is a condition that initiates the execution of a job. For example, triggers can include polling a source code repository for changes, receiving webhooks from a version control system, or scheduled times.
7. **What is a Parameterized Job?** A parameterized job allows users to pass parameters (inputs) when triggering the job. This is useful for customizing builds based on different configurations or environments.
8. **Pipeline as Code and Infrastructure as Code:** Pipeline as Code refers to defining your CI/CD pipelines in code (using Jenkinsfile), allowing version control and better collaboration. Infrastructure as Code (IaC) is managing and provisioning computing infrastructure through code, enabling automation and consistency in environments.
9. **Benefits of Using Pipelines Over Normal Build Jobs:** Pipelines provide a more flexible and powerful way to define the build process, allowing complex workflows, easy version control, and better visualization of stages and progress.
10. **What is a Jenkinsfile?** A Jenkinsfile is a text file that contains the definition of a Jenkins pipeline, written in either declarative or scripted syntax, enabling version control and easy management of build processes.
11. **Difference Between Declarative and Scripted Pipeline:**

* **Declarative Pipeline**: More structured and easier to read; uses predefined syntax and is designed for simplicity and best practices.
* **Scripted Pipeline**: More flexible and allows complex logic, but requires a deeper understanding of Groovy syntax.

1. **Advantages of Using Blue Ocean:** Blue Ocean provides a modern, user-friendly interface for Jenkins, simplifying the creation and management of pipelines, and improving visualization of CI/CD workflows and progress.
2. **What is a Multibranch Pipeline?** A multibranch pipeline automatically creates and manages pipelines for each branch of a repository. It can detect branches in version control and configure a separate pipeline for each.
3. **What is a Pull Request?** A pull request is a request to merge code changes from one branch into another (often from a feature branch into the main branch) in a version control system. It is often accompanied by discussions, code reviews, and automated checks before merging.

By completing these tasks and answering the questions, you'll gain a solid understanding of Jenkins and its role in CI/CD practices.

TEST Pyramid

[Test Pyramid](https://martinfowler.com/bliki/TestPyramid.html)

**1 May 2012**

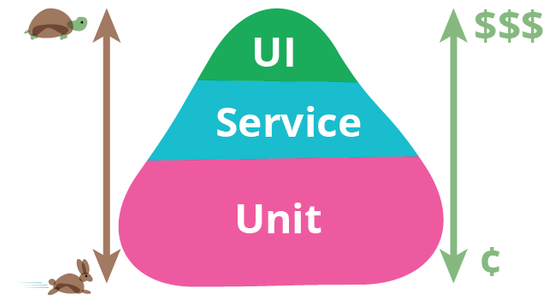
[A person wearing a hat

Description automatically generated](https://martinfowler.com/)

[**Martin Fowler**](https://martinfowler.com/)

[testing](https://martinfowler.com/tags/testing.html)

The test pyramid is a way of thinking about how different kinds of automated tests should be used to create a balanced portfolio. Its essential point is that you should have many more low-level [UnitTests](https://martinfowler.com/bliki/UnitTest.html) than high level [BroadStackTests](https://martinfowler.com/bliki/BroadStackTest.html) running through a GUI.



For much of my career test automation meant tests that drove an application through its user-interface. Such tools would often provide the facility to record an interaction with the application and then allow you to play back that interaction, checking that the application returned the same results. Such an approach works well initially. It's easy to record tests, and the tests can be recorded by people with no knowledge of programming.

But this kind of approach quickly runs into trouble, becoming an [ice-cream cone](https://watirmelon.blog/testing-pyramids). Testing through the UI like this is slow, increasing build times. Often it requires installed licences for the test automation software, which means it can only be done on particular machines. Usually these cannot easily be run in a “headless” mode, monitored by scripts to put in a proper deployment pipeline.

Most importantly such tests are very brittle. An enhancement to the system can easily end up breaking lots of such tests, which then have to be re-recorded. You can reduce this problem by abandoning record-playback tools, but that makes the tests harder to write. 1 Even with good practices on writing them, end-to-end tests are more prone to [non-determinism problems](https://martinfowler.com/articles/nonDeterminism.html), which can undermine trust in them. In short, tests that run end-to-end through the UI are: brittle, expensive to write, and time consuming to run. So the pyramid argues that you should do much more automated testing through unit tests than you should through traditional GUI based testing. 2

The pyramid also argues for an intermediate layer of tests that act through a service layer of an application, what I refer to as [SubcutaneousTests](https://martinfowler.com/bliki/SubcutaneousTest.html). These can provide many of the advantages of end-to-end tests but avoid many of the complexities of dealing with UI frameworks. In web applications this would correspond to testing through an API layer while the top UI part of the pyramid would correspond to tests using something like [Selenium](http://seleniumhq.org/) or Sahi.

The test pyramid comes up a lot in Agile testing circles and while its core message is sound, there is much more to say about building a well-balanced test portfolio. A common problem is that teams conflate the concepts of end-to-end tests, UI tests, and customer facing tests. These are all orthogonal characteristics. For example a rich javascript UI should have most of its UI behavior tested with javascript unit tests using something like [Jasmine](http://jasmine.github.io/). A complex set of business rules could have tests captured in a customer-facing form, but run just on the relevant module much as unit tests are.

I always argue that high-level tests are there as a second line of test defense. If you get a failure in a high level test, not just do you have a bug in your functional code, you also have a missing or incorrect unit test. Thus I advise that before fixing a bug exposed by a high level test, you should replicate the bug with a unit test. Then the unit test ensures the bug stays dead.

The Practical Test Pyramid

*The “Test Pyramid” is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups. Although the concept of the Test Pyramid has been around for a while, teams still struggle to put it into practice properly. This article revisits the original concept of the Test Pyramid and shows how you can put this into practice. It shows which kinds of tests you should be looking for in the different levels of the pyramid and gives practical examples on how these can be implemented.*

**26 February 2018**

[](http://www.hamvocke.com/)

[**Ham Vocke**](http://www.hamvocke.com/)

Ham is a software developer and consultant at Thoughtworks in Germany. Being tired of deploying software manually at 3 a.m., he added continuous delivery and diligent automation to his toolbox and set out to help teams deliver high-quality software reliably and efficiently. He makes up for the time gained by annoying people with his antics.

[testing](https://martinfowler.com/tags/testing.html)

**Translations:**[Brazilian Portuguese](https://github.com/paulojunior-ufla/piramide-de-teste-pratica) · [Chinese](https://insights.thoughtworks.cn/practical-test-pyramid/)

**Contents**

* [**The Importance of (Test) Automation**](https://martinfowler.com/articles/practical-test-pyramid.html#TheImportanceOftestAutomation)
* [**The Test Pyramid**](https://martinfowler.com/articles/practical-test-pyramid.html#TheTestPyramid)
* [**Tools and Libraries We'll Look at**](https://martinfowler.com/articles/practical-test-pyramid.html#ToolsAndLibrariesWellLookAt)
* [**The Sample Application**](https://martinfowler.com/articles/practical-test-pyramid.html#TheSampleApplication)
  + [**Functionality**](https://martinfowler.com/articles/practical-test-pyramid.html#Functionality)
  + [**High-level Structure**](https://martinfowler.com/articles/practical-test-pyramid.html#High-levelStructure)
  + [**Internal Architecture**](https://martinfowler.com/articles/practical-test-pyramid.html#InternalArchitecture)
* [**Unit tests**](https://martinfowler.com/articles/practical-test-pyramid.html#UnitTests)
  + [**What's a Unit?**](https://martinfowler.com/articles/practical-test-pyramid.html#WhatsAUnit)
  + [**Sociable and Solitary**](https://martinfowler.com/articles/practical-test-pyramid.html#SociableAndSolitary)
  + [**Mocking and Stubbing**](https://martinfowler.com/articles/practical-test-pyramid.html#MockingAndStubbing)
  + [**What to Test?**](https://martinfowler.com/articles/practical-test-pyramid.html#WhatToTest)
  + [**Test Structure**](https://martinfowler.com/articles/practical-test-pyramid.html#TestStructure)
  + [**Implementing a Unit Test**](https://martinfowler.com/articles/practical-test-pyramid.html#ImplementingAUnitTest)
* [**Integration Tests**](https://martinfowler.com/articles/practical-test-pyramid.html#IntegrationTests)
  + [**Database Integration**](https://martinfowler.com/articles/practical-test-pyramid.html#DatabaseIntegration)
  + [**Integration With Separate Services**](https://martinfowler.com/articles/practical-test-pyramid.html#IntegrationWithSeparateServices)
* [**Contract Tests**](https://martinfowler.com/articles/practical-test-pyramid.html#ContractTests)
  + [**Consumer Test (our team)**](https://martinfowler.com/articles/practical-test-pyramid.html#ConsumerTestourTeam)
  + [**Provider Test (the other team)**](https://martinfowler.com/articles/practical-test-pyramid.html#ProviderTesttheOtherTeam)
  + [**Provider Test (our team)**](https://martinfowler.com/articles/practical-test-pyramid.html#ProviderTestourTeam)
* [**UI Tests**](https://martinfowler.com/articles/practical-test-pyramid.html#UiTests)
* [**End-to-End Tests**](https://martinfowler.com/articles/practical-test-pyramid.html#End-to-endTests)
  + [**User Interface End-to-End Test**](https://martinfowler.com/articles/practical-test-pyramid.html#UserInterfaceEnd-to-endTest)
  + [**REST API End-to-End Test**](https://martinfowler.com/articles/practical-test-pyramid.html#RestApiEnd-to-endTest)
* [**Acceptance Tests — Do Your Features Work Correctly?**](https://martinfowler.com/articles/practical-test-pyramid.html#acceptance)
* [**Exploratory Testing**](https://martinfowler.com/articles/practical-test-pyramid.html#ExploratoryTesting)
* [**The Confusion About Testing Terminology**](https://martinfowler.com/articles/practical-test-pyramid.html#TheConfusionAboutTestingTerminology)
* [**Putting Tests Into Your Deployment Pipeline**](https://martinfowler.com/articles/practical-test-pyramid.html#PuttingTestsIntoYourDeploymentPipeline)
* [**Avoid Test Duplication**](https://martinfowler.com/articles/practical-test-pyramid.html#AvoidTestDuplication)
* [**Writing Clean Test Code**](https://martinfowler.com/articles/practical-test-pyramid.html#WritingCleanTestCode)
* [**Conclusion**](https://martinfowler.com/articles/practical-test-pyramid.html#Conclusion)

**Sidebars**

* [**But I Really Need to Test This Private Method**](https://martinfowler.com/articles/practical-test-pyramid.html#private-methods-sidebar)
* [**Specialised Test Helpers**](https://martinfowler.com/articles/practical-test-pyramid.html#SpecialisedTestHelpers)

A magnifying glass over a triangle

Description automatically generated

Production-ready software requires testing before it goes into production. As the discipline of software development matured, software testing approaches have matured too. Instead of having myriads of manual software testers, development teams have moved towards automating the biggest portion of their testing efforts. Automating their tests allows teams to know whether their software is broken in a matter of seconds and minutes instead of days and weeks.

The drastically shortened feedback loop fuelled by automated tests goes hand in hand with agile development practices, continuous delivery and DevOps culture. Having an effective software testing approach allows teams to move fast and with confidence.

This article explores what a well-rounded test portfolio should look like to be responsive, reliable and maintainable - regardless of whether you're building a microservices architecture, mobile apps or IoT ecosystems. We'll also get into the details of building effective and readable automated tests.

**The Importance of (Test) Automation**

Software has become an essential part of the world we live in. It has outgrown its early sole purpose of making businesses more efficient. Today companies try to find ways to become first-class digital companies. As users everyone of us interacts with an ever-increasing amount of software every day. The wheels of innovation are turning faster.

If you want to keep pace you'll have to look into ways to deliver your software faster without sacrificing its quality. **Continuous delivery**, a practice where you automatically ensure that your software can be released into production any time, can help you with that. With continuous delivery you use a **build pipeline** to automatically test your software and deploy it to your testing and production environments.

Building, testing and deploying an ever-increasing amount of software manually soon becomes impossible — unless you want to spend all your time with manual, repetitive work instead of delivering working software. Automating everything — from build to tests, deployment and infrastructure — is your only way forward.

A white cube with black background

Description automatically generated

*Figure 1: Use build pipelines to automatically and reliably get your software into production*

Traditionally software testing was overly manual work done by deploying your application to a test environment and then performing some black-box style testing e.g. by clicking through your user interface to see if anything's broken. Often these tests would be specified by test scripts to ensure the testers would do consistent checking.

It's obvious that testing all changes manually is time-consuming, repetitive and tedious. Repetitive is boring, boring leads to mistakes and makes you look for a different job by the end of the week.

Luckily there's a remedy for repetitive tasks: *automation*.

Automating your repetitive tests can be a big game changer in your life as a software developer. Automate these tests and you no longer have to mindlessly follow click protocols in order to check if your software still works correctly. Automate your tests and you can change your codebase without batting an eye. If you've ever tried doing a large-scale refactoring without a proper test suite I bet you know what a terrifying experience this can be. How would you know if you accidentally broke stuff along the way? Well, you click through all your manual test cases, that's how. But let's be honest: do you really enjoy that? How about making even large-scale changes and knowing whether you broke stuff within seconds while taking a nice sip of coffee? Sounds more enjoyable if you ask me.

**The Test Pyramid**

If you want to get serious about automated tests for your software there is one key concept you should know about: the **test pyramid**. Mike Cohn came up with this concept in his book *Succeeding with Agile*. It's a great visual metaphor telling you to think about different layers of testing. It also tells you how much testing to do on each layer.

A close-up of a triangle

Description automatically generated

*Figure 2: The Test Pyramid*

Mike Cohn's original test pyramid consists of three layers that your test suite should consist of (bottom to top):

1. Unit Tests
2. Service Tests
3. User Interface Tests

Unfortunately the concept of the test pyramid falls a little short if you take a closer look. Some argue that either the naming or some conceptual aspects of Mike Cohn's test pyramid are not ideal, and I have to agree. From a modern point of view the test pyramid seems overly simplistic and can therefore be misleading.

Still, due to its simplicity the essence of the test pyramid serves as a good rule of thumb when it comes to establishing your own test suite. Your best bet is to remember two things from Cohn's original test pyramid:

1. Write tests with different granularity
2. The more high-level you get the fewer tests you should have

Stick to the pyramid shape to come up with a healthy, fast and maintainable test suite: Write *lots* of small and fast *unit tests*. Write *some* more coarse-grained tests and *very few* high-level tests that test your application from end to end. Watch out that you don't end up with a [test ice-cream cone](https://watirmelon.blog/testing-pyramids/) that will be a nightmare to maintain and takes way too long to run.

Don't become too attached to the names of the individual layers in Cohn's test pyramid. In fact they can be quite misleading: *service test* is a term that is hard to grasp (Cohn himself talks about the observation that [a lot of developers completely ignore this layer](https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid)). In the days of single page application frameworks like react, angular, ember.js and others it becomes apparent that *UI tests* don't have to be on the highest level of your pyramid - you're perfectly able to unit test your UI in all of these frameworks.

Given the shortcomings of the original names it's totally okay to come up with other names for your test layers, as long as you keep it consistent within your codebase and your team's discussions.

**Tools and Libraries We'll Look at**

* [JUnit](http://junit.org/): our test runner
* [Mockito](http://site.mockito.org/): for mocking dependencies
* [Wiremock](http://wiremock.org/): for stubbing out external services
* [Pact](https://docs.pact.io/): for writing CDC tests
* [Selenium](http://docs.seleniumhq.org/): for writing UI-driven end-to-end tests
* [REST-assured](https://github.com/rest-assured/rest-assured): for writing REST API-driven end-to-end tests

**The Sample Application**

I've written a [simple microservice](https://github.com/hamvocke/spring-testing) including a test suite with tests for the different layers of the test pyramid.

The sample application shows traits of a typical microservice. It provides a REST interface, talks to a database and fetches information from a third-party REST service. It's implemented in [Spring Boot](https://projects.spring.io/spring-boot/)and should be understandable even if you've never worked with Spring Boot before.

Make sure to check out [the code on Github](https://github.com/hamvocke/spring-testing). The readme contains instructions you need to run the application and its automated tests on your machine.

**Functionality**

The application's functionality is simple. It provides a REST interface with three endpoints:

|  |  |
| --- | --- |
| GET /hello | Returns *“Hello World”*. Always. |
| GET /hello/{lastname} | Looks up the person with the provided last name. If the person is known, returns *“Hello {Firstname} {Lastname}”*. |
| GET /weather | Returns the current weather conditions for *Hamburg, Germany*. |

**High-level Structure**

On a high-level the system has the following structure:

A black and white rectangular sign

Description automatically generated

*Figure 3: the high level structure of our microservice system*

Our microservice provides a REST interface that can be called via HTTP. For some endpoints the service will fetch information from a database. In other cases the service will call an external [weather API](https://darksky.net/) via HTTP to fetch and display current weather conditions.

**Internal Architecture**

Internally, the Spring Service has a Spring-typical architecture:

A diagram of a weather api

Description automatically generated

*Figure 4: the internal structure of our microservice*

* Controller classes provide *REST* endpoints and deal with *HTTP* requests and responses
* Repository classes interface with the *database* and take care of writing and reading data to/from persistent storage
* Client classes talk to other APIs, in our case it fetches *JSON* via *HTTPS* from the darksky.net weather API
* Domain classes capture our [domain model](https://en.wikipedia.org/wiki/Domain_model) including the domain logic (which, to be fair, is quite trivial in our case).

Experienced Spring developers might notice that a frequently used layer is missing here: Inspired by [Domain-Driven Design](https://en.wikipedia.org/wiki/Domain-driven_design) a lot of developers build a *service layer* consisting of *service* classes. I decided not to include a service layer in this application. One reason is that our application is simple enough, a service layer would have been an unnecessary level of indirection. The other one is that I think people overdo it with service layers. I often encounter codebases where the entire business logic is captured within service classes. The domain model becomes merely a layer for data, not for behaviour (an [Anemic Domain Model](https://en.wikipedia.org/wiki/Anemic_domain_model)). For every non-trivial application this wastes a lot of potential to keep your code well-structured and testable and does not fully utilise the power of object orientation.

Our repositories are straightforward and provide simple CRUD functionality. To keep the code simple I used [Spring Data](http://projects.spring.io/spring-data/). Spring Data gives us a simple and generic CRUD repository implementation that we can use instead of rolling our own. It also takes care of spinning up an in-memory database for our tests instead of using a real PostgreSQL database as it would in production.

Take a look at the codebase and make yourself familiar with the internal structure. It will be useful for our next step: Testing the application!

**Unit tests**

The foundation of your test suite will be made up of unit tests. Your unit tests make sure that a certain unit (your *subject under test*) of your codebase works as intended. Unit tests have the narrowest scope of all the tests in your test suite. The number of unit tests in your test suite will largely outnumber any other type of test.

A white cube with a black square and a yellow square with a black background

Description automatically generated with medium confidence

*Figure 5: A unit test typically replaces external collaborators with test doubles*

**What's a Unit?**

If you ask three different people what *“unit”* means in the context of unit tests, you'll probably receive four different, slightly nuanced answers. To a certain extent it's a matter of your own definition and it's okay to have no canonical answer.

If you're working in a functional language a *unit* will most likely be a single function. Your unit tests will call a function with different parameters and ensure that it returns the expected values. In an object-oriented language a unit can range from a single method to an entire class.

**Sociable and Solitary**

Some argue that all collaborators (e.g. other classes that are called by your class under test) of your subject under test should be substituted with *mocks* or *stubs* to come up with perfect isolation and to avoid side-effects and a complicated test setup. Others argue that only collaborators that are slow or have bigger side effects (e.g. classes that access databases or make network calls) should be stubbed or mocked.

[Occasionally](https://martinfowler.com/bliki/UnitTest.html) people label these two sorts of tests as **solitary unit tests** for tests that stub all collaborators and **sociable unit tests** for tests that allow talking to real collaborators (Jay Fields' [Working Effectively with Unit Tests](https://leanpub.com/wewut) coined these terms). If you have some spare time you can go down the rabbit hole and [read more about the pros and cons](https://martinfowler.com/articles/mocksArentStubs.html) of the different schools of thought.

At the end of the day it's not important to decide if you go for solitary or sociable unit tests. Writing automated tests is what's important. Personally, I find myself using both approaches all the time. If it becomes awkward to use real collaborators I will use mocks and stubs generously. If I feel like involving the real collaborator gives me more confidence in a test I'll only stub the outermost parts of my service.

**Mocking and Stubbing**

Mocks and Stubs are two different kinds of [Test Doubles](https://martinfowler.com/bliki/TestDouble.html) (there are more than these two). A lot of people use the terms Mock and Stub interchangeably. I think it's good to be precise and keep their specific properties in mind. You can use test doubles to replace objects you'd use in production with an implementation that helps you with testing.

In plain words it means that you replace a real thing (e.g. a class, module or function) with a fake version of that thing. The fake version looks and acts like the real thing (answers to the same method calls) but answers with canned responses that you define yourself at the beginning of your unit test.

Using test doubles is not specific to unit testing. More elaborate test doubles can be used to simulate entire parts of your system in a controlled way. However, in unit testing you're most likely to encounter a lot of mocks and stubs (depending of whether you're the sociable or solitary kind of developer), simply because lots of modern languages and libraries make it easy and comfortable to set up mocks and stubs.

Regardless of your technology choice, there's a good chance that either your language's standard library or some popular third-party library will provide you with elegant ways to set up mocks. And even writing your own mocks from scratch is only a matter of writing a fake class/module/function with the same signature as the real one and setting up the fake in your test.

Your unit tests will run very fast. On a decent machine you can expect to run thousands of unit tests within a few minutes. Test small pieces of your codebase in isolation and avoid hitting databases, the filesystem or firing HTTP queries (by using mocks and stubs for these parts) to keep your tests fast.

Once you got a hang of writing unit tests you will become more and more fluent in writing them. Stub out external collaborators, set up some input data, call your subject under test and check that the returned value is what you expected. Look into [Test-Driven Development](https://en.wikipedia.org/wiki/Test-driven_development) and let your unit tests guide your development; if applied correctly it can help you get into a great flow and come up with a good and maintainable design while automatically producing a comprehensive and fully automated test suite. Still, it's no silver bullet. Go ahead, give it a real chance and see if it feels right for you.

**But I *Really* Need to Test This Private Method**

If you ever find yourself in a situation where you *really really* need to test a private method you should take a step back and ask yourself why.

I'm pretty sure this is more of a design problem than a scoping problem. Most likely you feel the need to test a private method because it's complex and testing this method through the public interface of the class requires a lot of awkward setup.

Whenever I find myself in this situation I usually come to the conclusion that the class I'm testing is already too complex. It's doing too much and violates the *single responsibility* principle - the *S* of the five [SOLID](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) principles.

The solution that often works for me is to split the original class into two classes. It often only takes one or two minutes of thinking to find a good way to cut the one big class into two smaller classes with individual responsibility. I move the private method (that I urgently want to test) to the new class and let the old class call the new method. Voilà, my awkward-to-test private method is now public and can be tested easily. On top of that I have improved the structure of my code by adhering to the single responsibility principle.

**What to Test?**

The good thing about unit tests is that you can write them for all your production code classes, regardless of their functionality or which layer in your internal structure they belong to. You can unit tests controllers just like you can unit test repositories, domain classes or file readers. Simply stick to the **one test class per production class** rule of thumb and you're off to a good start.

A unit test class should at least **test the *public* interface of the class**. Private methods can't be tested anyways since you simply can't call them from a different test class. *Protected* or *package-private* are accessible from a test class (given the package structure of your test class is the same as with the production class) but testing these methods could already go too far.

There's a fine line when it comes to writing unit tests: They should ensure that all your non-trivial code paths are tested (including happy path and edge cases). At the same time they shouldn't be tied to your implementation too closely.

Why's that?

Tests that are too close to the production code quickly become annoying. As soon as you refactor your production code (quick recap: refactoring means changing the internal structure of your code without changing the externally visible behaviour) your unit tests will break.

This way you lose one big benefit of unit tests: acting as a safety net for code changes. You rather become fed up with those stupid tests failing every time you refactor, causing more work than being helpful; and whose idea was this stupid testing stuff anyways?

What do you do instead? Don't reflect your internal code structure within your unit tests. Test for observable behaviour instead. Think about

*if I enter values x and y, will the result be z?*

instead of

*if I enter x and y, will the method call class A first, then call class B and then return the result of class A plus the result of class B?*

Private methods should generally be considered an implementation detail. That's why you shouldn't even have the urge to test them.

I often hear opponents of unit testing (or TDD ) arguing that writing unit tests becomes pointless work where you have to test all your methods in order to come up with a high test coverage. They often cite scenarios where an overly eager team lead forced them to write unit tests for getters and setters and all other sorts of trivial code in order to come up with 100% test coverage.

There's so much wrong with that.

Yes, you should *test the public interface*. More importantly, however, you **don't test trivial code**. Don't worry, [Kent Beck said it's ok](https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/). You won't gain anything from testing simple *getters* or *setters* or other trivial implementations (e.g. without any conditional logic). Save the time, that's one more meeting you can attend, hooray!

**Test Structure**

A good structure for all your tests (this is not limited to unit tests) is this one:

1. Set up the test data
2. Call your method under test
3. Assert that the expected results are returned

There's a nice mnemonic to remember this structure: [*“Arrange, Act, Assert”*](https://xp123.com/articles/3a-arrange-act-assert/). Another one that you can use takes inspiration from BDD . It's the [*“given”*, *“when”*, *“then”*](https://martinfowler.com/bliki/GivenWhenThen.html) triad, where *given* reflects the setup, *when* the method call and *then* the assertion part.

This pattern can be applied to other, more high-level tests as well. In every case they ensure that your tests remain easy and consistent to read. On top of that tests written with this structure in mind tend to be shorter and more expressive.

**Specialised Test Helpers**

It's a thing of beauty that you can write unit tests for your entire codebase, regardless of what layer of your application's architecture you're on. The example shows a simple unit test for a controller. Unfortunately, when it comes to Spring's controllers there's a downside to this approach: Spring MVC's controller make heavy use of annotations to declare which paths they're listening on, which HTTP verbs to use, which parameters they parse from the URL path or query params and so on. Simply invoking a controller's method within your unit tests won't test all of these crucial things. Luckily, the Spring folks came up with a nice test helper you can use to write better controller tests. Make sure to check out [MockMVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-server). It gives you a nice DSL you can use to fire fake requests against your controller and check that everything's cool. I've included an [example](https://github.com/hamvocke/spring-testing/blob/master/src/test/java/example/ExampleControllerAPITest.java) in the sample codebase. A lot of frameworks offer test helpers to make testing specific aspects of your codebase more pleasant. Check out the documentation of your framework of choice and see if it offers any useful helpers for your automated tests.

**Implementing a Unit Test**

Now that we know what to test and how to structure our unit tests we can finally see a real example.

Let's take a simplified version of the ExampleController class:

@RestController

public class ExampleController {

private final PersonRepository personRepo;

@Autowired

public ExampleController(final PersonRepository personRepo) {

this.personRepo = personRepo;

}

@GetMapping("/hello/{lastName}")

public String hello(@PathVariable final String lastName) {

Optional<Person> foundPerson = personRepo.findByLastName(lastName);

return foundPerson

.map(person -> String.format("Hello %s %s!",

person.getFirstName(),

person.getLastName()))

.orElse(String.format("Who is this '%s' you're talking about?",

lastName));

}

}

A unit test for the hello(lastname) method could look like this:

public class ExampleControllerTest {

private ExampleController subject;

@Mock

private PersonRepository personRepo;

@Before

public void setUp() throws Exception {

initMocks(this);

subject = new ExampleController(personRepo);

}

@Test

public void shouldReturnFullNameOfAPerson() throws Exception {

Person peter = new Person("Peter", "Pan");

given(personRepo.findByLastName("Pan"))

.willReturn(Optional.of(peter));

String greeting = subject.hello("Pan");

assertThat(greeting, is("Hello Peter Pan!"));

}

@Test

public void shouldTellIfPersonIsUnknown() throws Exception {

given(personRepo.findByLastName(anyString()))

.willReturn(Optional.empty());

String greeting = subject.hello("Pan");

assertThat(greeting, is("Who is this 'Pan' you're talking about?"));

}

}

We're writing the unit tests using [JUnit](http://junit.org/), the de-facto standard testing framework for Java. We use [Mockito](http://site.mockito.org/) to replace the real PersonRepository class with a stub for our test. This stub allows us to define canned responses the stubbed method should return in this test. Stubbing makes our test more simple, predictable and allows us to easily setup test data.

Following the *arrange, act, assert* structure, we write two unit tests - a positive case and a case where the searched person cannot be found. The first, positive test case creates a new person object and tells the mocked repository to return this object when it's called with *“Pan”* as the value for the lastName parameter. The test then goes on to call the method that should be tested. Finally it asserts that the response is equal to the expected response.

The second test works similarly but tests the scenario where the tested method does not find a person for the given parameter.

**Integration Tests**

All non-trivial applications will integrate with some other parts (databases, filesystems, network calls to other applications). When writing unit tests these are usually the parts you leave out in order to come up with better isolation and faster tests. Still, your application will interact with other parts and this needs to be tested. [**Integration Tests**](https://martinfowler.com/bliki/IntegrationTest.html) are there to help. They test the integration of your application with all the parts that live outside of your application.

For your automated tests this means you don't just need to run your own application but also the component you're integrating with. If you're testing the integration with a database you need to run a database when running your tests. For testing that you can read files from a disk you need to save a file to your disk and load it in your integration test.

I mentioned before that “unit tests” is a vague term, this is even more true for “integration tests”. For some people integration testing means to test through the entire stack of your application connected to other applications within your system. I like to treat integration testing more narrowly and test one integration point at a time by replacing separate services and databases with test doubles. Together with contract testing and running contract tests against test doubles as well as the real implementations you can come up with integration tests that are faster, more independent and usually easier to reason about.

Narrow integration tests live at the boundary of your service. Conceptually they're always about triggering an action that leads to integrating with the outside part (filesystem, database, separate service). A database integration test would look like this:

A black and white logo

Description automatically generated

*Figure 6: A database integration test integrates your code with a real database*

1. start a database
2. connect your application to the database
3. trigger a function within your code that writes data to the database
4. check that the expected data has been written to the database by reading the data from the database

Another example, testing that your service integrates with a separate service via a REST API could look like this:

A black and white logo

Description automatically generated

*Figure 7: This kind of integration test checks that your application can communicate with a separate service correctly*

1. start your application
2. start an instance of the separate service (or a test double with the same interface)
3. trigger a function within your code that reads from the separate service's API
4. check that your application can parse the response correctly

Your integration tests - like unit tests - can be fairly whitebox. Some frameworks allow you to start your application while still being able to mock some other parts of your application so that you can check that the correct interactions have happened.

Write integration tests for all pieces of code where you either *serialize* or *deserialize* data. This happens more often than you might think. Think about:

* Calls to your services' REST API
* Reading from and writing to databases
* Calling other application's APIs
* Reading from and writing to queues
* Writing to the filesystem

Writing integration tests around these boundaries ensures that writing data to and reading data from these external collaborators works fine.

When writing *narrow integration tests* you should aim to run your external dependencies locally: spin up a local MySQL database, test against a local ext4 filesystem. If you're integrating with a separate service either run an instance of that service locally or build and run a fake version that mimics the behaviour of the real service.

If there's no way to run a third-party service locally you should opt for running a dedicated test instance and point at this test instance when running your integration tests. Avoid integrating with the real production system in your automated tests. Blasting thousands of test requests against a production system is a surefire way to get people angry because you're cluttering their logs (in the best case) or even DoS 'ing their service (in the worst case). Integrating with a service over the network is a typical characteristic of a *broad integration test* and makes your tests slower and usually harder to write.

With regards to the test pyramid, integration tests are on a higher level than your unit tests. Integrating slow parts like filesystems and databases tends to be much slower than running unit tests with these parts stubbed out. They can also be harder to write than small and isolated unit tests, after all you have to take care of spinning up an external part as part of your tests. Still, they have the advantage of giving you the confidence that your application can correctly work with all the external parts it needs to talk to. Unit tests can't help you with that.

**Database Integration**

The PersonRepository is the only repository class in the codebase. It relies on *Spring Data* and has no actual implementation. It just extends the CrudRepository interface and provides a single method header. The rest is Spring magic.

public interface PersonRepository extends CrudRepository<Person, String> {

Optional<Person> findByLastName(String lastName);

}

With the CrudRepository interface Spring Boot offers a fully functional CRUD repository with findOne, findAll, save, update and delete methods. Our custom method definition (findByLastName()) extends this basic functionality and gives us a way to fetch Persons by their last name. Spring Data analyses the return type of the method and its method name and checks the method name against a naming convention to figure out what it should do.

Although Spring Data does the heavy lifting of implementing database repositories I still wrote a database integration test. You might argue that this is *testing the framework* and something that I should avoid as it's not our code that we're testing. Still, I believe having at least one integration test here is crucial. First it tests that our custom findByLastName method actually behaves as expected. Secondly it proves that our repository used Spring's wiring correctly and can connect to the database.

To make it easier for you to run the tests on your machine (without having to install a PostgreSQL database) our test connects to an in-memory *H2* database.

I've defined H2 as a test dependency in the build.gradle file. The application.properties in the test directory doesn't define any spring.datasource properties. This tells Spring Data to use an in-memory database. As it finds H2 on the classpath it simply uses H2 when running our tests.

When running the real application with the int profile (e.g. by setting SPRING\_PROFILES\_ACTIVE=int as environment variable) it connects to a PostgreSQL database as defined in the application-int.properties.

I know, that's an awful lot of Spring specifics to know and understand. To get there, you'll have to sift through [a lot of documentation](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html#boot-features-embedded-database-support). The resulting code is easy on the eye but hard to understand if you don't know the fine details of Spring.

On top of that going with an in-memory database is risky business. After all, our integration tests run against a different type of database than they would in production. Go ahead and decide for yourself if you prefer Spring magic and simple code over an explicit yet more verbose implementation.

Enough explanation already, here's a simple integration test that saves a Person to the database and finds it by its last name:

@RunWith(SpringRunner.class)

@DataJpaTest

public class PersonRepositoryIntegrationTest {

@Autowired

private PersonRepository subject;

@After

public void tearDown() throws Exception {

subject.deleteAll();

}

@Test

public void shouldSaveAndFetchPerson() throws Exception {

Person peter = new Person("Peter", "Pan");

subject.save(peter);

Optional<Person> maybePeter = subject.findByLastName("Pan");

assertThat(maybePeter, is(Optional.of(peter)));

}

}

You can see that our integration test follows the same *arrange, act, assert* structure as the unit tests. Told you that this was a universal concept!

**Integration With Separate Services**

Our microservice talks to [darksky.net](https://darksky.net/), a weather REST API. Of course we want to ensure that our service sends requests and parses the responses correctly.

We want to avoid hitting the real *darksky* servers when running automated tests. Quota limits of our free plan are only part of the reason. The real reason is *decoupling*. Our tests should run independently of whatever the lovely people at darksky.net are doing. Even when your machine can't access the *darksky* servers or the darksky servers are down for maintenance.

We can avoid hitting the real *darksky* servers by running our own, fake *darksky* server while running our integration tests. This might sound like a huge task. Thanks to tools like [Wiremock](http://wiremock.org/) it's easy peasy. Watch this:

@RunWith(SpringRunner.class)

@SpringBootTest

public class WeatherClientIntegrationTest {

@Autowired

private WeatherClient subject;

@Rule

public WireMockRule wireMockRule = new WireMockRule(8089);

@Test

public void shouldCallWeatherService() throws Exception {

wireMockRule.stubFor(get(urlPathEqualTo("/some-test-api-key/53.5511,9.9937"))

.willReturn(aResponse()

.withBody(FileLoader.read("classpath:weatherApiResponse.json"))

.withHeader(CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)

.withStatus(200)));

Optional<WeatherResponse> weatherResponse = subject.fetchWeather();

Optional<WeatherResponse> expectedResponse = Optional.of(new WeatherResponse("Rain"));

assertThat(weatherResponse, is(expectedResponse));

}

}

To use Wiremock we instantiate a WireMockRule on a fixed port (8089). Using the DSL we can set up the Wiremock server, define the endpoints it should listen on and set canned responses it should respond with.

Next we call the method we want to test, the one that calls the third-party service and check if the result is parsed correctly.

It's important to understand how the test knows that it should call the fake Wiremock server instead of the real *darksky* API. The secret is in our application.properties file contained in src/test/resources. This is the properties file Spring loads when running tests. In this file we override configuration like API keys and URLs with values that are suitable for our testing purposes, e.g. calling the fake Wiremock server instead of the real one:

weather.url = http://localhost:8089

Note that the port defined here has to be the same we define when instantiating the WireMockRule in our test. Replacing the real weather API's URL with a fake one in our tests is made possible by injecting the URL in our WeatherClient class' constructor:

@Autowired

public WeatherClient(final RestTemplate restTemplate,

@Value("${weather.url}") final String weatherServiceUrl,

@Value("${weather.api\_key}") final String weatherServiceApiKey) {

this.restTemplate = restTemplate;

this.weatherServiceUrl = weatherServiceUrl;

this.weatherServiceApiKey = weatherServiceApiKey;

}

This way we tell our WeatherClient to read the weatherUrl parameter's value from the weather.url property we define in our application properties.

Writing *narrow integration tests* for a separate service is quite easy with tools like Wiremock. Unfortunately there's a downside to this approach: How can we ensure that the fake server we set up behaves like the real server? With the current implementation, the separate service could change its API and our tests would still pass. Right now we're merely testing that our WeatherClient can parse the responses that the fake server sends. That's a start but it's very brittle. Using *end-to-end tests* and running the tests against a test instance of the real service instead of using a fake service would solve this problem but would make us reliant on the availability of the test service. Fortunately, there's a better solution to this dilemma: Running contract tests against the fake and the real server ensures that the fake we use in our integration tests is a faithful test double. Let's see how this works next.

**Contract Tests**

More modern software development organisations have found ways of scaling their development efforts by spreading the development of a system across different teams. Individual teams build individual, loosely coupled services without stepping on each others toes and integrate these services into a big, cohesive system. The more recent buzz around microservices focuses on exactly that.

Splitting your system into many small services often means that these services need to communicate with each other via certain (hopefully well-defined, sometimes accidentally grown) interfaces.

Interfaces between different applications can come in different shapes and technologies. Common ones are

* REST and JSON via HTTPS
* RPC using something like [gRPC](https://grpc.io/)
* building an event-driven architecture using queues

For each interface there are two parties involved: the provider and the consumer. The **provider** serves data to consumers. The **consumer** processes data obtained from a provider. In a REST world a provider builds a REST API with all required endpoints; a consumer makes calls to this REST API to fetch data or trigger changes in the other service. In an asynchronous, event-driven world, a provider (often rather called **publisher**) publishes data to a queue; a consumer (often called **subscriber**) subscribes to these queues and reads and processes data.

A white and black cubes with a white circle

Description automatically generated

*Figure 8: Each interface has a providing (or publishing) and a consuming (or subscribing) party. The specification of an interface can be considered a contract.*

As you often spread the consuming and providing services across different teams you find yourself in the situation where you have to clearly specify the interface between these services (the so called **contract**). Traditionally companies have approached this problem in the following way:

* Write a long and detailed interface specification (the *contract*)
* Implement the providing service according to the defined contract
* Throw the interface specification over the fence to the consuming team
* Wait until they implement their part of consuming the interface
* Run some large-scale manual system test to see if everything works
* Hope that both teams stick to the interface definition forever and don't screw up

More modern software development teams have replaced steps 5. and 6. with something more automated: Automated [contract tests](https://martinfowler.com/bliki/ContractTest.html) make sure that the implementations on the consumer and provider side still stick to the defined contract. They serve as a good regression test suite and make sure that deviations from the contract will be noticed early.

In a more agile organisation you should take the more efficient and less wasteful route. You build your applications within the same organisation. It really shouldn't be too hard to talk to the developers of the other services directly instead of throwing overly detailed documentation over the fence. After all they're your co-workers and not a third-party vendor that you could only talk to via customer support or legally bulletproof contracts.

**Consumer-Driven Contract tests** (CDC tests) let the [consumers drive the implementation of a contract](https://martinfowler.com/articles/consumerDrivenContracts.html). Using CDC, consumers of an interface write tests that check the interface for all data they need from that interface. The consuming team then publishes these tests so that the publishing team can fetch and execute these tests easily. The providing team can now develop their API by running the CDC tests. Once all tests pass they know they have implemented everything the consuming team needs.

A group of cubes with a white circle in the middle

Description automatically generated

*Figure 9: Contract tests ensure that the provider and all consumers of an interface stick to the defined interface contract. With CDC tests consumers of an interface publish their requirements in the form of automated tests; the providers fetch and execute these tests continuously*

This approach allows the providing team to implement only what's really necessary (keeping things simple, YAGNI and all that). The team providing the interface should fetch and run these CDC tests continuously (in their build pipeline) to spot any breaking changes immediately. If they break the interface their CDC tests will fail, preventing breaking changes to go live. As long as the tests stay green the team can make any changes they like without having to worry about other teams. The Consumer-Driven Contract approach would leave you with a process looking like this:

* The consuming team writes automated tests with all consumer expectations
* They publish the tests for the providing team
* The providing team runs the CDC tests continuously and keeps them green
* Both teams talk to each other once the CDC tests break

If your organisation adopts a microservices approach, having CDC tests is a big step towards establishing autonomous teams. CDC tests are an automated way to foster team communication. They ensure that interfaces between teams are working at any time. Failing CDC tests are a good indicator that you should walk over to the affected team, have a chat about any upcoming API changes and figure out how you want to move forward.

A naive implementation of CDC tests can be as simple as firing requests against an API and assert that the responses contain everything you need. You then package these tests as an executable (.gem, .jar, .sh) and upload it somewhere the other team can fetch it (e.g. an artifact repository like [Artifactory](https://www.jfrog.com/artifactory/)).

Over the last couple of years the CDC approach has become more and more popular and several tools been build to make writing and exchanging them easier.

[Pact](https://github.com/realestate-com-au/pact) is probably the most prominent one these days. It has a sophisticated approach of writing tests for the consumer and the provider side, gives you stubs for separate services out of the box and allows you to exchange CDC tests with other teams. Pact has been ported to a lot of platforms and can be used with JVM languages, Ruby, .NET, JavaScript and many more.

If you want to get started with CDCs and don't know how, Pact can be a sane choice. The [documentation](https://docs.pact.io/) can be overwhelming at first. Be patient and work through it. It helps to get a firm understanding for CDCs which in turn makes it easier for you to advocate for the use of CDCs when working with other teams.

Consumer-Driven Contract tests can be a real game changer to establish autonomous teams that can move fast and with confidence. Do yourself a favor, read up on that concept and give it a try. A solid suite of CDC tests is invaluable for being able to move fast without breaking other services and cause a lot of frustration with other teams.

**Consumer Test (our team)**

Our microservice consumes the weather API. So it's our responsibility to write a **consumer test** that defines our expectations for the contract (the API) between our microservice and the weather service.

First we include a library for writing pact consumer tests in our build.gradle:

testCompile('au.com.dius:pact-jvm-consumer-junit\_2.11:3.5.5')

Thanks to this library we can implement a consumer test and use pact's mock services:

@RunWith(SpringRunner.class)

@SpringBootTest

public class WeatherClientConsumerTest {

@Autowired

private WeatherClient weatherClient;

@Rule

public PactProviderRuleMk2 weatherProvider =

new PactProviderRuleMk2("weather\_provider", "localhost", 8089, this);

@Pact(consumer="test\_consumer")

public RequestResponsePact createPact(PactDslWithProvider builder) throws IOException {

return builder

.given("weather forecast data")

.uponReceiving("a request for a weather request for Hamburg")

.path("/some-test-api-key/53.5511,9.9937")

.method("GET")

.willRespondWith()

.status(200)

.body(FileLoader.read("classpath:weatherApiResponse.json"),

ContentType.APPLICATION\_JSON)

.toPact();

}

@Test

@PactVerification("weather\_provider")

public void shouldFetchWeatherInformation() throws Exception {

Optional<WeatherResponse> weatherResponse = weatherClient.fetchWeather();

assertThat(weatherResponse.isPresent(), is(true));

assertThat(weatherResponse.get().getSummary(), is("Rain"));

}

}

If you look closely, you'll see that the WeatherClientConsumerTest is very similar to the WeatherClientIntegrationTest. Instead of using Wiremock for the server stub we use Pact this time. In fact the consumer test works exactly as the integration test, we replace the real third-party server with a stub, define the expected response and check that our client can parse the response correctly. In this sense the WeatherClientConsumerTest is a narrow integration test itself. The advantage over the wiremock-based test is that this test generates a *pact file* (found in target/pacts/&pact-name>.json) each time it runs. This pact file describes our expectations for the contract in a special JSON format. This pact file can then be used to verify that our stub server behaves like the real server. We can take the pact file and hand it to the team providing the interface. They take this pact file and write a provider test using the expectations defined in there. This way they test if their API fulfils all our expectations.

You see that this is where the *consumer-driven* part of CDC comes from. The consumer drives the implementation of the interface by describing their expectations. The provider has to make sure that they fulfil all expectations and they're done. No gold-plating, no YAGNI and stuff.

Getting the pact file to the providing team can happen in multiple ways. A simple one is to check them into version control and tell the provider team to always fetch the latest version of the pact file. A more advances one is to use an artifact repository, a service like Amazon's S3 or the pact broker. Start simple and grow as you need.

In your real-world application you don't need both, an *integration test* and a *consumer test* for a client class. The sample codebase contains both to show you how to use either one. If you want to write CDC tests using pact I recommend sticking to the latter. The effort of writing the tests is the same. Using pact has the benefit that you automatically get a pact file with the expectations to the contract that other teams can use to easily implement their provider tests. Of course this only makes sense if you can convince the other team to use pact as well. If this doesn't work, using the *integration test* and Wiremock combination is a decent plan b.

**Provider Test (the other team)**

The provider test has to be implemented by the people providing the weather API. We're consuming a public API provided by darksky.net. In theory the darksky team would implement the provider test on their end to check that they're not breaking the contract between their application and our service.

Obviously they don't care about our meager sample application and won't implement a CDC test for us. That's the big difference between a public-facing API and an organisation adopting microservices. Public-facing APIs can't consider every single consumer out there or they'd become unable to move forward. Within your own organisation, you can — and should. Your app will most likely serve a handful, maybe a couple dozen of consumers max. You'll be fine writing provider tests for these interfaces in order to keep a stable system.

The providing team gets the pact file and runs it against their providing service. To do so they implement a provider test that reads the pact file, stubs out some test data and runs the expectations defined in the pact file against their service.

The pact folks have written several libraries for implementing provider tests. Their main [GitHub repo](https://github.com/DiUS/pact-jvm) gives you a nice overview which consumer and which provider libraries are available. Pick the one that best matches your tech stack.

For simplicity let's assume that the darksky API is implemented in Spring Boot as well. In this case they could use the [Spring pact provider](https://github.com/DiUS/pact-jvm/tree/master/pact-jvm-provider-spring) which hooks nicely into Spring's MockMVC mechanisms. A hypothetical provider test that the darksky.net team would implement could look like this:

@RunWith(RestPactRunner.class)

@Provider("weather\_provider") // same as the "provider\_name" in our clientConsumerTest

@PactFolder("target/pacts") // tells pact where to load the pact files from

public class WeatherProviderTest {

@InjectMocks

private ForecastController forecastController = new ForecastController();

@Mock

private ForecastService forecastService;

@TestTarget

public final MockMvcTarget target = new MockMvcTarget();

@Before

public void before() {

initMocks(this);

target.setControllers(forecastController);

}

@State("weather forecast data") // same as the "given()" in our clientConsumerTest

public void weatherForecastData() {

when(forecastService.fetchForecastFor(any(String.class), any(String.class)))

.thenReturn(weatherForecast("Rain"));

}

}

You see that all the provider test has to do is to load a pact file (e.g. by using the @PactFolder annotation to load previously downloaded pact files) and then define how test data for pre-defined states should be provided (e.g. using Mockito mocks). There's no custom test to be implemented. These are all derived from the pact file. It's important that the provider test has matching counterparts to the *provider name* and *state* declared in the consumer test.

**Provider Test (our team)**

We've seen how to test the contract between our service and the weather provider. With this interface our service acts as consumer, the weather service acts as provider. Thinking a little further we'll see that our service also acts as a provider for others: We provide a REST API that offers a couple of endpoints ready to be consumed by others.

As we've just learned that contract tests are all the rage, we of course write a contract test for this contract as well. Luckily we're using consumer-driven contracts so there's all the consuming teams sending us their Pacts that we can use to implement our provider tests for our REST API.

Let's first add the Pact provider library for Spring to our project:

testCompile('au.com.dius:pact-jvm-provider-spring\_2.12:3.5.5')

Implementing the provider test follows the same pattern as described before. For the sake of simplicity I simply checked the pact file from our [simple consumer](https://github.com/hamvocke/spring-testing-consumer) into our service's repository. This makes it easier for our purpose, in a real-life scenario you're probably going to use a more sophisticated mechanism to distribute your pact files.

@RunWith(RestPactRunner.class)

@Provider("person\_provider")// same as in the "provider\_name" part in our pact file

@PactFolder("target/pacts") // tells pact where to load the pact files from

public class ExampleProviderTest {

@Mock

private PersonRepository personRepository;

@Mock

private WeatherClient weatherClient;

private ExampleController exampleController;

@TestTarget

public final MockMvcTarget target = new MockMvcTarget();

@Before

public void before() {

initMocks(this);

exampleController = new ExampleController(personRepository, weatherClient);

target.setControllers(exampleController);

}

@State("person data") // same as the "given()" part in our consumer test

public void personData() {

Person peterPan = new Person("Peter", "Pan");

when(personRepository.findByLastName("Pan")).thenReturn(Optional.of

(peterPan));

}

}

The shown ExampleProviderTest needs to provide state according to the pact file we're given, that's it. Once we run the provider test, Pact will pick up the pact file and fire HTTP request against our service that then responds according to the state we've set up.

**UI Tests**

Most applications have some sort of user interface. Typically we're talking about a web interface in the context of web applications. People often forget that a REST API or a command line interface is as much of a user interface as a fancy web user interface.

*UI tests* test that the user interface of your application works correctly. User input should trigger the right actions, data should be presented to the user, the UI state should change as expected.

A black and white rectangular object with a green tick

Description automatically generated

UI Tests and end-to-end tests are sometimes (as in Mike Cohn's case) said to be the same thing. For me this conflates two things that are rather orthogonal concepts.

Yes, testing your application end-to-end often means driving your tests through the user interface. The inverse, however, is not true.

Testing your user interface doesn't have to be done in an end-to-end fashion. Depending on the technology you use, testing your user interface can be as simple as writing some unit tests for your frontend javascript code with your backend stubbed out.

With traditional web applications testing the user interface can be achieved with tools like [Selenium](http://docs.seleniumhq.org/). If you consider a REST API to be your user interface you should have everything you need by writing proper integration tests around your API.

With web interfaces there's multiple aspects that you probably want to test around your UI: behaviour, layout, usability or adherence to your corporate design are only a few.

Fortunately, testing the *behaviour* of your user interface is pretty simple. You click here, enter data there and want the state of the user interface to change accordingly. Modern single page application frameworks ([react](https://facebook.github.io/react/), [vue.js](https://vuejs.org/), [Angular](https://angular.io/) and the like) often come with their own tools and helpers that allow you to thoroughly test these interactions in a pretty low-level (unit test) fashion. Even if you roll your own frontend implementation using vanilla javascript you can use your regular testing tools like [Jasmine](https://jasmine.github.io/) or [Mocha](http://mochajs.org/). With a more traditional, server-side rendered application, Selenium-based tests will be your best choice.

Testing that your web application's *layout* remains intact is a little harder. Depending on your application and your users' needs you may want to make sure that code changes don't break the website's layout by accident.

The problem is that computers are notoriously bad at checking if something “looks good” (maybe some clever machine learning algorithm can change that in the future).

There are some tools to try if you want to automatically check your web application's design in your build pipeline. Most of these tools utilise Selenium to open your web application in different browsers and formats, take screenshots and compare these to previously taken screenshots. If the old and new screenshots differ in an unexpected way, the tool will let you know.

[Galen](http://galenframework.com/) is one of these tools. But even rolling your own solution isn't too hard if you have special requirements. Some teams I've worked with built [lineup](https://github.com/otto-de/lineup) and its Java-based cousin [jlineup](https://github.com/otto-de/jlineup) to achieve something similar. Both tools take the same Selenium-based approach I described before.

Once you want to test for *usability* and a “looks good” factor you leave the realms of automated testing. This is the area where you should rely on [exploratory testing](https://en.wikipedia.org/wiki/Exploratory_testing), usability testing (this can even be as simple as [hallway testing](https://en.wikipedia.org/wiki/Usability_testing#Hallway_testing)) and showcases with your users to see if they like using your product and can use all features without getting frustrated or annoyed.

**End-to-End Tests**

Testing your deployed application via its user interface is the most end-to-end way you could test your application. The previously described, webdriver driven UI tests are a good example of end-to-end tests.

A black and white cubes

Description automatically generated

*Figure 11: End-to-end tests test your entire, completely integrated system*

End-to-end tests (also called [Broad Stack Tests](https://martinfowler.com/bliki/BroadStackTest.html)) give you the biggest confidence when you need to decide if your software is working or not. [Selenium](http://docs.seleniumhq.org/) and the [WebDriver Protocol](https://www.w3.org/TR/webdriver/) allow you to automate your tests by automatically driving a (headless) browser against your deployed services, performing clicks, entering data and checking the state of your user interface. You can use Selenium directly or use tools that are build on top of it, [Nightwatch](http://nightwatchjs.org/) being one of them.

End-to-End tests come with their own kind of problems. They are notoriously flaky and often fail for unexpected and unforeseeable reasons. Quite often their failure is a false positive. The more sophisticated your user interface, the more flaky the tests tend to become. Browser quirks, timing issues, animations and unexpected popup dialogs are only some of the reasons that got me spending more of my time with debugging than I'd like to admit.

In a microservices world there's also the big question of who's in charge of writing these tests. Since they span multiple services (your entire system) there's no single team responsible for writing end-to-end tests.

If you have a centralised *quality assurance* team they look like a good fit. Then again having a centralised QA team is a big anti-pattern and shouldn't have a place in a DevOps world where your teams are meant to be truly cross-functional. There's no easy answer who should own end-to-end tests. Maybe your organisation has a community of practice or a *quality guild* that can take care of these. Finding the correct answer highly depends on your organisation.

Furthermore, end-to-end tests require a lot of maintenance and run pretty slowly. Thinking about a landscape with more than a couple of microservices in place you won't even be able to run your end-to-end tests locally — as this would require to start all your microservices locally as well. Good luck spinning up hundreds of applications on your development machine without frying your RAM.

Due to their high maintenance cost you should aim to reduce the number of end-to-end tests to a bare minimum.

Think about the high-value interactions users will have with your application. Try to come up with user journeys that define the core value of your product and translate the most important steps of these user journeys into automated end-to-end tests.

If you're building an e-commerce site your most valuable customer journey could be a user searching for a product, putting it in the shopping basket and doing a checkout. That's it. As long as this journey still works you shouldn't be in too much trouble. Maybe you'll find one or two more crucial user journeys that you can translate into end-to-end tests. Everything more than that will likely be more painful than helpful.

Remember: you have lots of lower levels in your test pyramid where you already tested all sorts of edge cases and integrations with other parts of the system. There's no need to repeat these tests on a higher level. High maintenance effort and lots of false positives will slow you down and cause you to lose trust in your tests, sooner rather than later.

**User Interface End-to-End Test**

For end-to-end tests [Selenium](http://docs.seleniumhq.org/) and the [WebDriver](https://www.w3.org/TR/webdriver/) protocol are the tool of choice for many developers. With Selenium you can pick a browser you like and let it automatically call your website, click here and there, enter data and check that stuff changes in the user interface.

Selenium needs a browser that it can start and use for running its tests. There are multiple so-called *'drivers'* for different browsers that you could use. [Pick one](https://www.mvnrepository.com/search?q=selenium+driver) (or multiple) and add it to your build.gradle. Whatever browser you choose, you need to make sure that all devs in your team and your CI server have installed the correct version of the browser locally. This can be pretty painful to keep in sync. For Java, there's a nice little library called [webdrivermanager](https://github.com/bonigarcia/webdrivermanager) that can automate downloading and setting up the correct version of the browser you want to use. Add these two dependencies to your build.gradle and you're good to go:

testCompile('org.seleniumhq.selenium:selenium-chrome-driver:2.53.1')

testCompile('io.github.bonigarcia:webdrivermanager:1.7.2')

Running a fully-fledged browser in your test suite can be a hassle. Especially when using continuous delivery the server running your pipeline might not be able to spin up a browser including a user interface (e.g. because there's no X-Server available). You can take a workaround for this problem by starting a virtual X-Server like [xvfb](https://en.wikipedia.org/wiki/Xvfb).

A more recent approach is to use a *headless* browser (i.e. a browser that doesn't have a user interface) to run your webdriver tests. Until recently [PhantomJS](http://phantomjs.org/) was the leading headless browser used for browser automation. Ever since both [Chromium](https://developers.google.com/web/updates/2017/04/headless-chrome) and [Firefox](https://developer.mozilla.org/en-US/Firefox/Headless_mode) announced that they've implemented a headless mode in their browsers PhantomJS all of a sudden became obsolete. After all it's better to test your website with a browser that your users actually use (like Firefox and Chrome) instead of using an artificial browser just because it's convenient for you as a developer.

Both, headless Firefox and Chrome, are brand new and yet to be widely adopted for implementing webdriver tests. We want to keep things simple. Instead of fiddling around to use the bleeding edge headless modes let's stick to the classic way using Selenium and a regular browser. A simple end-to-end test that fires up Chrome, navigates to our service and checks the content of the website looks like this:

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class HelloE2ESeleniumTest {

private WebDriver driver;

@LocalServerPort

private int port;

@BeforeClass

public static void setUpClass() throws Exception {

ChromeDriverManager.getInstance().setup();

}

@Before

public void setUp() throws Exception {

driver = new ChromeDriver();

}

@After

public void tearDown() {

driver.close();

}

@Test

public void helloPageHasTextHelloWorld() {

driver.get(String.format("http://127.0.0.1:%s/hello", port));

assertThat(driver.findElement(By.tagName("body")).getText(), containsString("Hello World!"));

}

}

Note that this test will only run on your system if you have Chrome installed on the system you run this test on (your local machine, your CI server).

The test is straightforward. It spins up the entire Spring application on a random port using @SpringBootTest. We then instantiate a new Chrome webdriver, tell it to go navigate to the /hello endpoint of our microservice and check that it prints “Hello World!” on the browser window. Cool stuff!

**REST API End-to-End Test**

Avoiding a graphical user interface when testing your application can be a good idea to come up with tests that are less flaky than full end-to-end tests while still covering a broad part of your application's stack. This can come in handy when testing through the web interface of your application is particularly hard. Maybe you don't even have a web UI but serve a REST API instead (because you have a single page application somewhere talking to that API, or simply because you despise everything that's nice and shiny). Either way, a [Subcutaneous Test](https://martinfowler.com/bliki/SubcutaneousTest.html) that tests just beneath the graphical user interface and can get you really far without compromising on confidence too much. Just the right thing if you're serving a REST API like we do in our example code:

@RestController

public class ExampleController {

private final PersonRepository personRepository;

// shortened for clarity

@GetMapping("/hello/{lastName}")

public String hello(@PathVariable final String lastName) {

Optional<Person> foundPerson = personRepository.findByLastName(lastName);

return foundPerson

.map(person -> String.format("Hello %s %s!",

person.getFirstName(),

person.getLastName()))

.orElse(String.format("Who is this '%s' you're talking about?",

lastName));

}

}

Let me show you one more library that comes in handy when testing a service that provides a REST API. [REST-assured](https://github.com/rest-assured/rest-assured) is a library that gives you a nice DSL for firing real HTTP requests against an API and evaluating the responses you receive.

First things first: Add the dependency to your build.gradle.

testCompile('io.rest-assured:rest-assured:3.0.3')

With this library at our hands we can implement an end-to-end test for our REST API:

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class HelloE2ERestTest {

@Autowired

private PersonRepository personRepository;

@LocalServerPort

private int port;

@After

public void tearDown() throws Exception {

personRepository.deleteAll();

}

@Test

public void shouldReturnGreeting() throws Exception {

Person peter = new Person("Peter", "Pan");

personRepository.save(peter);

when()

.get(String.format("http://localhost:%s/hello/Pan", port))

.then()

.statusCode(is(200))

.body(containsString("Hello Peter Pan!"));

}

}

Again, we start the entire Spring application using @SpringBootTest. In this case we @Autowire the PersonRepository so that we can write test data into our database easily. When we now ask the REST API to say “hello” to our friend “Mr Pan” we're being presented with a nice greeting. Amazing! And more than enough of an end-to-end test if you don't even sport a web interface.

**Acceptance Tests — Do Your Features Work Correctly?**

The higher you move up in your test pyramid the more likely you enter the realms of testing whether the features you're building work correctly from a user's perspective. You can treat your application as a black box and shift the focus in your tests from

when I enter the values x and y, the return value should be z

towards

*given* there's a logged in user

*and* there's an article “bicycle”

*when* the user navigates to the “bicycle” article's detail page

*and* clicks the “add to basket” button

*then* the article “bicycle” should be in their shopping basket

Sometimes you'll hear the terms [**functional test**](https://en.wikipedia.org/wiki/Functional_testing) or [**acceptance test**](https://en.wikipedia.org/wiki/Acceptance_testing#Acceptance_testing_in_extreme_programming) for these kinds of tests. Sometimes people will tell you that functional and acceptance tests are different things. Sometimes the terms are conflated. Sometimes people will argue endlessly about wording and definitions. Often this discussion is a pretty big source of confusion.

Here's the thing: At one point you should make sure to test that your software works correctly from a *user's* perspective, not just from a technical perspective. What you call these tests is really not that important. Having these tests, however, is. Pick a term, stick to it, and write those tests.

This is also the moment where people talk about BDD and tools that allow you to implement tests in a BDD fashion. BDD or a BDD-style way of writing tests can be a nice trick to shift your mindset from implementation details towards the users' needs. Go ahead and give it a try.

You don't even need to adopt full-blown BDD tools like [Cucumber](https://cucumber.io/) (though you can). Some assertion libraries (like [chai.js](http://chaijs.com/guide/styles/#should) allow you to write assertions with should-style keywords that can make your tests read more BDD-like. And even if you don't use a library that provides this notation, clever and well-factored code will allow you to write user behaviour focused tests. Some helper methods/functions can get you a very long way:

# a sample acceptance test in Python

def test\_add\_to\_basket():

# given

user = a\_user\_with\_empty\_basket()

user.login()

bicycle = article(name="bicycle", price=100)

# when

article\_page.add\_to\_.basket(bicycle)

# then

assert user.basket.contains(bicycle)

Acceptance tests can come in different levels of granularity. Most of the time they will be rather high-level and test your service through the user interface. However, it's good to understand that there's technically no need to write acceptance tests at the highest level of your test pyramid. If your application design and your scenario at hand permits that you write an acceptance test at a lower level, go for it. Having a low-level test is better than having a high-level test. The concept of acceptance tests - proving that your features work correctly for the user - is completely orthogonal to your test pyramid.

**Exploratory Testing**

Even the most diligent test automation efforts are not perfect. Sometimes you miss certain edge cases in your automated tests. Sometimes it's nearly impossible to detect a particular bug by writing a unit test. Certain quality issues don't even become apparent within your automated tests (think about design or usability). Despite your best intentions with regards to test automation, manual testing of some sorts is still a good idea.

A black and white picture of a person

Description automatically generated

*Figure 12: Use exploratory testing to spot all quality issues that your build pipeline didn't spot*

Include [Exploratory Testing](https://en.wikipedia.org/wiki/Exploratory_testing) in your testing portfolio. It is a manual testing approach that emphasises the tester's freedom and creativity to spot quality issues in a running system. Simply take some time on a regular schedule, roll up your sleeves and try to break your application. Use a destructive mindset and come up with ways to provoke issues and errors in your application. Document everything you find for later. Watch out for bugs, design issues, slow response times, missing or misleading error messages and everything else that would annoy you as a user of your software.

The good news is that you can happily automate most of your findings with automated tests. Writing automated tests for the bugs you spot makes sure there won't be any regressions of that bug in the future. Plus it helps you narrowing down the root cause of that issue during bugfixing.

During exploratory testing you will spot problems that slipped through your build pipeline unnoticed. Don't be frustrated. This is great feedback on the maturity of your build pipeline. As with any feedback, make sure to act on it: Think about what you can do to avoid these kinds of problems in the future. Maybe you're missing out on a certain set of automated tests. Maybe you have just been sloppy with your automated tests in this iteration and need to test more thoroughly in the future. Maybe there's a shiny new tool or approach that you could use in your pipeline to avoid these issues in the future. Make sure to act on it so your pipeline and your entire software delivery will grow more mature the longer you go.

**The Confusion About Testing Terminology**

Talking about different test classifications is always difficult. What I mean when I talk about *unit tests* can be slightly different from your understanding. With integration tests it's even worse. For some people integration testing is a very broad activity that tests through a lot of different parts of your entire system. For me it's a rather narrow thing, only testing the integration with one external part at a time. Some call them *integration tests*, some refer to them as *component tests*, some prefer the term *service test*. Even others will argue, that all of these three terms are totally different things. There's no right or wrong. The software development community simply hasn't managed to settle on well-defined terms around testing.

Don't get too hung up on sticking to ambiguous terms. It doesn't matter if you call it end-to-end or broad stack test or functional test. It doesn't matter if your integration tests mean something different to you than to the folks at another company. Yes, it would be really nice if our profession could settle on some well-defined terms and all stick to it. Unfortunately this hasn't happened yet. And since there are many nuances when it comes to writing tests it's really more of a spectrum than a bunch of discrete buckets anyways, which makes consistent naming even harder.

The important takeaway is that you should find terms that work for you and your team. Be clear about the different types of tests that you want to write. Agree on the naming in your team and find consensus on the scope of each type of test. If you get this consistent within your team (or maybe even within your organisation) that's really all you should care about. [Simon Stewart](https://testing.googleblog.com/2010/12/test-sizes.html) summed this up very nicely when he described the approach they use at Google. And I think it shows perfectly how getting too hung up on names and naming conventions just isn't worth the hassle.

**Putting Tests Into Your Deployment Pipeline**

If you're using Continuous Integration or Continuous Delivery, you'll have a [Deployment Pipeline](https://martinfowler.com/bliki/DeploymentPipeline.html) in place that will run automated tests every time you make a change to your software. Usually this pipeline is split into several stages that gradually give you more confidence that your software is ready to be deployed to production. Hearing about all these different kinds of tests you're probably wondering how you should place them within your deployment pipeline. To answer this you should just think about one of the very foundational values of Continuous Delivery (indeed one of the core [values of Extreme Programming](http://www.extremeprogramming.org/values.html) and agile software development): **Fast Feedback**.

A good build pipeline tells you that you messed up as quick as possible. You don't want to wait an hour just to find out that your latest change broke some simple unit tests. Chances are that you've probably gone home already if your pipeline takes that long to give you that feedback. You could get this information within a matter of seconds, maybe a few minutes by putting the fast running tests in the earlier stages of your pipeline. Conversely you put the longer running tests - usually the ones with a broader scope - in the later stages to not defer the feedback from the fast-running tests. You see that defining the stages of your deployment pipeline is not driven by the types of tests but rather by their speed and scope. With that in mind it can be a very reasonable decision to put some of the really narrowly-scoped and fast-running integration tests in the same stage as your unit tests - simply because they give you faster feedback and not because you want to draw the line along the formal type of your tests.

**Avoid Test Duplication**

Now that you know that you should write different types of tests there's one more pitfall to avoid: duplicating tests throughout the different layers of the pyramid. While your gut feeling might say that there's no such thing as too many tests let me assure you, there is. Every single test in your test suite is additional baggage and doesn't come for free. Writing and maintaining tests takes time. Reading and understanding other people's test takes time. And of course, running tests takes time.

As with production code you should strive for simplicity and avoid duplication. In the context of implementing your test pyramid you should keep two rules of thumb in mind:

1. If a higher-level test spots an error and there's no lower-level test failing, you need to write a lower-level test
2. Push your tests as far down the test pyramid as you can

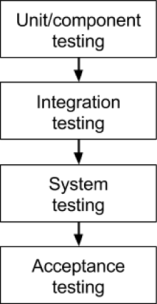
The first rule is important because lower-level tests allow you to better narrow down errors and replicate them in an isolated way. They'll run faster and will be less bloated when you're debugging the issue at hand. And they will serve as a good regression test for the future. The second rule is important to keep your test suite fast. If you have tested all conditions confidently on a lower-level test, there's no need to keep a higher-level test in your test suite. It just doesn't add more confidence that everything's working. Having redundant tests will become annoying in your daily work. Your test suite will be slower and you need to change more tests when you change the behaviour of your code.

Let's phrase this differently: If a higher-level test gives you more confidence that your application works correctly, you should have it. Writing a unit test for a Controller class helps to test the logic within the Controller itself. Still, this won't tell you whether the REST endpoint this Controller provides actually responds to HTTP requests. So you move up the test pyramid and add a test that checks for exactly that - but nothing more. You don't test all the conditional logic and edge cases that your lower-level tests already cover in the higher-level test again. Make sure that the higher-level test focuses on the part that the lower-level tests couldn't cover.

I'm rigorous when it comes to eliminating tests that don't provide any value. I delete high-level tests that are already covered on a lower level (given they don't provide extra value). I replace higher-level tests with lower-level tests if possible. Sometimes that's hard, especially if you know that coming up with a test was hard work. Beware of the sunk cost fallacy and hit the delete key. There's no reason to waste more precious time on a test that ceased to provide value.

Differences Between the Different Levels & Types of Testing

**What differences are there between the different levels of testing? The focus shifts from early component testing to late acceptance testing. It is important that everybody understands this.**



There are generally four recognized levels of testing: unit/component testing, integration testing, system testing, and acceptance testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Details of different Levels of Testing:

1. Unit/component testing

The most basic type of testing is unit, or component, testing.

Unit testing aims to verify each part of the software by isolating it and then perform tests to demonstrate that each individual component is correct in terms of fulfilling requirements and the desired functionality.

This type of testing is performed at the earliest stages of the development process, and in many cases it is executed by the developers themselves before handing the software over to the testing team.

The advantage of detecting any errors in the software early in the day is that by doing so the team minimises software development risks, as well as time and money wasted in having to go back and undo fundamental problems in the program once it is nearly completed.

(Must read: [5 best unit testing frameworks to automate unit tests](https://reqtest.com/testing-blog/unit-testing-frameworks/))

2. Integration testing

Integration testing aims to test different parts of the system in combination in order to assess if they work correctly together. By testing the units in groups, any faults in the way they interact together can be identified.

There are many ways to test how different components of the system function at their interface; testers can adopt either a bottom-up or a top-down integration method.

In bottom-up [integration testing](https://en.wikipedia.org/wiki/Integration_testing), testing builds on the results of unit testing by testing higher-level combination of units (called modules) in successively more complex scenarios.

It is recommended that testers start with this approach first, before applying the top-down approach which tests higher-level modules first and studies simpler ones later.

A close up of a sign

Description automatically generated

3. System testing

The next level of testing is system testing. As the name implies, all the components of the software are tested as a whole in order to ensure that the overall product meets the requirements specified.

[System testing](https://reqtest.com/testing-blog/software-system-testing/) is a very important step as the software is almost ready to ship and it can be tested in an environment which is very close to that which the user will experience once it is deployed.

System testing enables testers to ensure that the product meets business requirements, as well as determine that it runs smoothly within its operating environment. This type of testing is typically performed by a specialized testing team.

4. Acceptance testing

Finally, [acceptance testing](https://reqtest.com/testing-blog/a-guide-to-excellent-acceptance-testing/) is the level in the software testing process where a product is given the green light or not. The aim of this type of testing is to evaluate whether the system complies with the end-user requirements and if it is ready for deployment.

The testing team will utilise a variety of methods, such as pre-written scenarios and test cases to test the software and use the results obtained from these tools to find ways in which the system can be improved.

The scope of acceptance testing ranges from simply finding spelling mistakes and cosmetic errors, to uncovering bugs that could cause a major error in the application.

By performing acceptance tests, the testing team can find out how the product will perform when it is installed on the user’s system. There are also various legal and contractual reasons why acceptance testing has to be carried out.

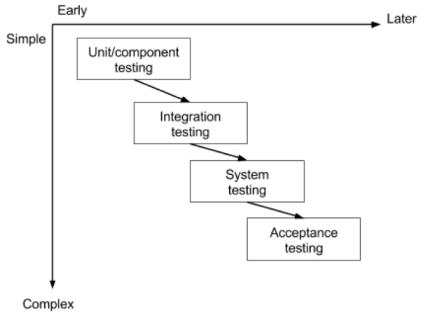
The testing sequence

These four types of testing cannot be applied haphazardly during development. There is a logical sequence that should be adhered to in order to minimise the risk of bugs cropping up just before the launch date.

Any testing team should know that testing is important at every phase of the development cycle.

By progressively testing the simpler components of the system and moving on the bigger, more complex groupings, the testers can rest assured they are thoroughly examining the software in the most efficient way possible.

The four  levels of testing shouldn’t only be seen as a hierarchy that extends from simple to complex, but also as a sequence that spans the whole development process from the early to the later stages. Note however that later does not imply that acceptance testing is done only after say 6 months of development work. In a more agile approach, acceptance testing can be carried out as often as every 2-3 weeks, as a part of the sprint demo. In an organization working more traditionally it is quite typical to have 3-4 releases per year, each following the cycle described here.



Test Design Techniques

**There are two main categories of testing:**  
  
1) Static Testing  
2) Dynamic Testing

**Difference between Static and Dynamic testing:**

|  |  |
| --- | --- |
| **Static Testing** | **Dynamic Testing** |
| Static testing is completed without executing the program. | Dynamic testing is completed with the execution of program. |
| This testing is executed in verification stage. | This testing is executed in validation stage. |
| Static testing is executed before the compilation. | Dynamic testing is executed after the compilation. |
| This testing prevents the defects. | This testing finds and fixes the defects. |
| The cost is less for finding the defects and fixes. | The cost is high for finding and fixing the defects. |
| It consists of Walkthrough, Inspection, reviews etc. | It consists of specification based, structure based, Experience based, unit testing, integration testing etc. |

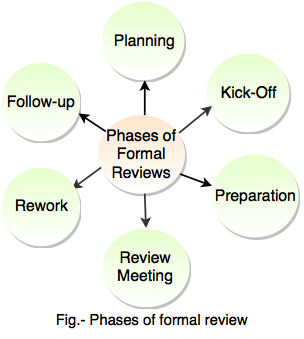
Static Testing Techniques

In this chapter we are going to discuss about the static testing technique i.e Informal review, Walkthrough, Inspection, Technical Reviews. Reviews are vary from informal to formal review.

**Informal Reviews**

* Informal reviews are applied in the early stages of the life cycle of the document.
* These reviews are conducted between two person team. In later stages more people are involved.
* The aim of informal reviews is to improve the quality of the document and help the authors.
* These reviews are not based on the procedure and not documented.

**Formal reviews**

Formal reviews follow the formal process i.e these reviews are well structured and managed.  
  
**Following are the phases of formal reviews:**  
  
i) Planning  
ii) Kick-Off  
iii) Preparation  
iv) Review meeting  
v) Rework  
vi) Follow-up  
  


i) Planning

* Review process starts with planning phase. In planning, the review process starts with a request for review by the authors to the inspection leader.
* In the formal review, the inspection leader executes the entry check and defines the exit criteria.
* Entry criteria verifies the document is ready to enter the formal review process.

ii) Kick-off

* Kick-off meeting is optional in review procedure.
* The aim of kick-off step is to explain the objectives of review and  distribute the documents in meeting etc.

iii) Preparation

* In preparation, reviewers review the document separately using related rules, procedures, documents.
* Every reviewer recognizes the defects, questions and comments as per their role and understanding of document.

iv) Review meeting

**Review meeting includes three phases:**  
  
**1. Logging Phase -** Defects and issues are identified in the preparation step that are logged page by page.  
**2. Discussion Phase -** This phase handles the issues that require discussion.

**3. Decision Phase -** Decision on the document reviews is constructed by reviewers or participants. Sometimes decision is based on formal **exit criteria** (Average number of major defects found per page).

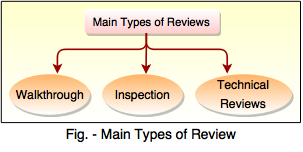
v) Rework

If the number of defects found per page is more than certain level then the document needs to be reworked.

vi) Follow-up

* In follow up, moderator ensures that author has taken an action on all known defects.
* The distribution of updated document and collection of feedback is completed in follow-up.
* In follow up, it is the responsibility of the moderator to ensure that the information is correct and it stored for future analysis.

Types of Reviews

**Main types of review are as follows:**  
  
1) Walkthrough  
2) Inspection  
3) Technical Review  
  
  
  
**These types of review comes under static testing technique.**

**1) Walkthrough**

* In walkthrough, author guides the review team via the document to fulfil the common understanding and collecting the feedback.
* Walkthrough is **not a formal process.**
* In walkthrough, a review team does not require to do detailed study before meeting while authors are already in the scope of preparation.
* Walkthrough is useful for higher-level documents i.e requirement specification and architectural documents.

**Goals of walkthrough**

* Make the document available for the stakeholders both outside and inside the software discipline for collecting the information about the topic under documentation.
* Describe and evaluate the content of the document.
* Study and discuss the validity of possible alternatives and proposed solutions.

**2) Inspection**

* The trained moderator guides the Inspection. **It is most formal type of review.**
* The reviewers are prepared and check the documents before the meeting.
* In Inspection, a separate preparation is achieved when the product is examined and defects are found. These defects are documented in issue log.
* In Inspection, moderator performs a formal follow-up by applying exit criteria.

**Goals of Inspection**

* Assist the author to improve the quality of the document under inspection.
* Efficiently and rapidly remove the defects.
* Generating the documents with higher level of quality and it helps to improve the product quality.
* It learns from the previous defects found and prohibits the occurrence of similar defects.
* Generate common understanding by interchanging information.

**3) Technical Review**

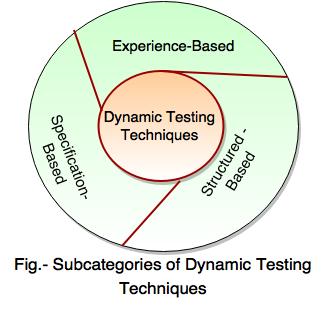
* Technical review is a discussion meeting that focuses on technical content of the document. **It is a less formal review.**
* It is guided by a trained moderator or a technical expert.

**Goals of technical review**

* The goal is to evaluate the value of technical concept in the project environment.
* Build the consistency in the use and representation of the technical concepts.
* In early stages it ensures that the technical concepts are used correctly.
* Notify the participants regarding the technical content of the document.

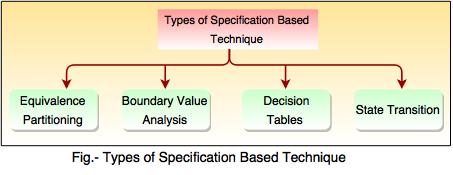
Dynamic testing   
  
Introduction

* Dynamic testing is a method to access the use of a software program by giving input and examining the output.
* This technique is used to detect defects and to determine quality attributes of the code.
* Dynamic testing is completed in validation process.

**Subcategories of Dynamic testing techniques**  
  


1) Specification-Based Testing Technique

* Specification-based testing technique is called as **Black Box** or **Input/Output driven** Testing Technique. It is because the testers view the software as black-box with inputs and output.
* Functional and non-functional testing are the types of specification based testing.
* Specification based technique is applicable at all the levels of testing.
* Specification based test design techniques use the specification of the program.
* Specification can be anything such as written document, collection of use cases, a set of modules or a prototype.

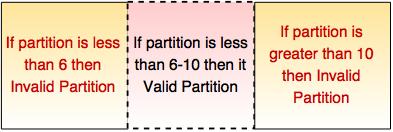
**Following are the types of specification Based testing techniques:**  
  
i) Equivalence Partitioning  
ii) Boundary Value Analysis  
iii) Decision Tables  
iv) State Transition  
  


**i) Equivalence Partitioning**

* In equivalence partitioning, software testing technique divides the input data of a software unit into the partitions of equivalent data and test cases are derived from the partitions of equivalent data.
* Equivalence Partitioning is applied at any level of testing.
* The software treats all the conditions in one partition as the same. Hence, equivalence partitioning needs to check only one condition from each partition.
* Since, all the condition in one partition are same, if one condition in partition works then we can assume that all the conditions in that partition work or otherwise.

**ii) Boundary Value Analysis**

* Boundary Value Analysis is the test case design technique to test boundary value between partitions.
* Boundary value is an input or value on the border of an equivalence partition.
* It consists of start-end, lower-upper, maximum-minimum on inside and outside boundaries.
* Boundary Value Analysis (BVA) checks the boundary values of Equivalence Class Partitioning(ECP), hence BVA comes after ECP.
* For example, A tool consists of user name and password field. It accesses minimum 6 and maximum 10 characters to work with this tool. Valid range is 6-10 characters, Invalid range is 5 or less than 5 characters and more than 10 characters.



**iii) Decision Tables**

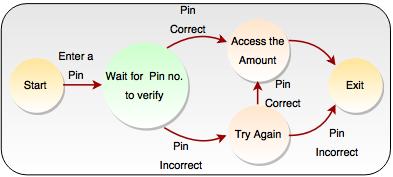
* Decision table testing is a combination of inputs that generates various outputs. It is also known as Cause-Effect table.
* It gives a standard way to test complicated business rules that are useful for testers and developers.
* The first task is to recognize a suitable function that behaves according to the combination of inputs.
* If the system contains many inputs then combination becomes unmanageable.

**iv) State Transition Testing**

* The system can be in a limited number of various states and the transitions from one state to another. It is determined by the rules of the “machine”.  This is the model based on the system and the tests.
* The model is more elaborated as we require the advantage of the state transition technique.

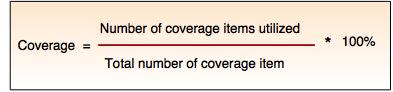
**Following are the basic parts of the state transition model:**

* The state of the software could be start, open, close, exit etc.
* The transition from one state to another is allowed but not compulsory.
* An event causes a transition like closing a file or withdrawing money etc.
* The action results from a transition i.e an error message or access the cash.
* **For example,**State transition diagram for ATM machine.



2) Structure-Based Testing Techniques

* Structure based technique is called as **White box** testing or **Glass box** testing.
* Using structured based technique, we produce additional test cases which are different from existing tests.
* Structure based testing assists two purposes i.e test coverage measurement and structural test case design.
* The test cases derived in specification based techniques are used to access coverage of testing in above purposes.
* These are used to design more tests with the goal of increasing the test coverage.
* Test coverage measures the amount of testing executed by a set of test.



* The hazard of coverage measure is that 100 % coverage does not mean 100 % tested.
* From the view of multi dimensional concept, coverage technique measures only one dimension.
* There is a possibility that two distinct test cases execute exactly same coverage. But, there is a possibility that input data of one detects an error while that of another does not find an error.

**Advantages of code coverage measurement**

* Code coverage measurement produces more test cases to increase coverage.
* It helps to find area of a program which is not utilized by a set of test cases.
* It helps in determining a significant measure of code coverage that indirectly measures the quality of an application or product.

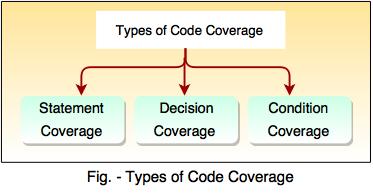
Coverage measure

The coverage measurement of code is completed by using tools. Many tools are available for this task.  
  
**These tools helps for following tasks:**

* It helps to improve the quality and productivity of testing.
* The quality improves to verify more structural aspects that are tested. This helps to find the defects on those structural path.
* It helps in testing the same structure with different data.

**Types of coverage**

* There are many types of test coverages. The Test coverage is used at all levels of testing.
* The coverage is measured at component level testing, integration level of testing or acceptance level of testing.
* On the basis of a number of various structural elements in the system or component the coverage can be measured.

**Following are the types of Code Coverage:**  
  
i) Statement coverage  
ii) Decision coverage  
iii) Condition coverage  
  


i) Statement coverage

* Statement coverage is called as line coverage or segment coverage.
* It covers only the true conditions.
* The executed statement can recognize during the process of statement coverage but the code is not executed because of blockage.

**Advantages of Statement Coverage**

* It ensures what the written code  is expected to do and not to do.
* It measures the quality of written code.
* It verifies the flow of various paths in the program and also verifies if those path are tested or not.

**Disadvantages of Statement Coverage**

* Statement coverage cannot test the false conditions.
* It does not report if the loop has reached at its termination or not.
* It does not recognize the logical operators.

ii) Decision coverage

* Decision coverage is called as branch coverage or all edges coverage.
* It tests both true and false conditions.
* A branch is the result of a decision. So, branch coverage measures which decision results were tested.
* It seems good as it takes a more in-depth view of source code than the simple statement coverage.
* A decision is an IF statement, a loop control statement i.e DO-WHILE or REPEAT-UNTIL etc. or CASE statement.
* In these statements two or more result are possible.
* With an IF statement, the exit can be TRUE or FLASE, depending on the value of the logical condition that comes after IF.

**Advantages of Decision Coverage**

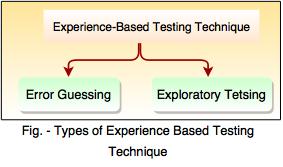
* It ensures that all the branches in the code are reached.
* It removes the problems that occur with statement coverage testing.

iii) Condition coverage

* Condition coverage is also called as Predicate Coverage.
* Condition coverage is for boolean expressions. It verifies whether all the boolean expression evaluate to both TRUE and FALSE.
* This technique needs the coverage of all conditions which determine the decision result.

3) Experience-Based Testing Technique

* Experience-Based testing technique requires testers to possess knowledge, skills and background. These are important to test conditions and test cases.
* The experience people in both technical and business are needed, because they bring various aspects to test, analysis and design process as well as they may have an idea about which is the best way for testing and what is going wrong.
* This technique is used with specification-based and structured based techniques and also used when there is no specialization or the specification is out of date.
* This technique is used for low-risk system.

**Types of Experience Based Testing Technique**  
  
i) Error guessing  
ii) Exploratory testing  
  


**i) Error guessing**

* In error guessing, experienced and good testers are required to recognize the defects in the component.
* The experienced testers are able to find weaknesses of a system. So, the error guessing approach is used after more formal techniques and it is effective.
* The assumption and guesses are constructed by the experienced testers. Hence it saves time in error guessing.
* The success of Error Guessing technique is absolutely dependent on the skills and experience of the tester.

**ii) Exploratory testing**

* Exploratory testing examines or explores the software. It finds out if software works or not.
* The testers continuously make decisions about what to test next and where to spend time. This approach is useful when there is limited time and poor specifications are available.
* In exploratory testing, testers are involved in minimum planning and maximum test execution.
* This testing is used to check formal test process i.e. planning, test design, test execution etc. which ensures that most serious defects that are found.

Deciding the test technique

Which testing technique is best for particular project is based on the number of Internal and external factors.

**Internal factors**

**Following internal factors help in deciding which technique should be used for particular project:**  
  
i) The model used in the development of system  
ii) Tester's knowledge and experience.  
iii) Experience with a similar previous system  
iv) Objectives of the test  
v) Initial documentation of the content and its style.  
vi) Use of life cycle model

**External factors**

**Following external factors decide which technique should be used for a particular project:**

* **Risk Assessment -** If the risk is high then it requires more detailed and formal testing. Commercial risk can be affected by quality issue hence the exploratory testing is suitable.
* The requirements of customer and commitments in the **contract.**
* **The type of system used -** The type of system i.e embedded, graphical, financial etc. will affect the choice of techniques.  
  **For example:**The financial application consists of a lot of calculations. In this case, boundary value analysis is beneficial.
* **Regulatory requirements -**Some of the industries have regulatory standards or guidelines that guide the testing techniques to be used.
* **Time and budget of the project -** The available time always affects  the choice of testing techniques. If more time is available then we can select more techniques and if limited time is available then find the most important defects only.

DEFECT Management

**Defect Management Process: How to Manage a Defect Effectively**

By [**Vijay**](https://www.softwaretestinghelp.com/author/vijay/)  Updated March 7, 2024

**Introduction to Defect Management Process:**

The more focused process and testing will allow less buggy software in the market.

**Defect Prevention** is much more efficient and effective in reducing the number of defects and also is very cost effective to fix the defects found during the early stage of the software process. Most of the organizations conduct **Defect Discovery**, **Defect Removal** and then **Process Improvement** which is collectively known as a**Defect Management Process**.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2018/05/Defect-Management-Process.jpg)

**Table of Contents:**[[Show](https://www.softwaretestinghelp.com/defect-management-process/)]

**Goals of Defect Management Process (DMP)**

**Given below are the various goals of this process:**

* Prevent the Defect
* Early Detection
* Minimize the impact
* Resolution of the Defect
* Process improvement

Before exploring the Defect Management Process, let us first understand**what actually a defect or bug is?**

**Defect Management Life Cycle**

When a system gives a different output other than the actual business requirement i.e. when there is a deviation from the actual or original business requirement then we can say that there is a defect in the system/software.

When the testing team executes the test cases, they come across a situation where the actual test result is different from the expected result. This variation is termed as a **Defect**.

Basically, a Software Defect is a condition which does not meet the software requirement. The defect is an error or a flaw which produces an unexpected or incorrect behavior in the system.

In order to handle the projects appropriately, you need to know how to deal with the development and release, but along with that you also need to know how to handle defects.

Just imagine, what will happen if the testing team reports the defects verbally and the development team also updates the status of defect verbally? The process will be more complicated as these defects include all defects like actually fixed and working as expected, fixed but still not working, rejected, postponed and the process goes on.

In the above case, as the numbers of defects get increased and communication is performed verbally, the situation will soon be very worst. **In order to control and handle the defect effectively, you need proper Defect Life Cycle.**

Defect Life Cycle ensures that the process is uniform and standardized. A defect attains different states in the life cycle. After a defect has been found, it goes through various stages during its lifetime and it is commonly known as **Defect Life Cycle**.

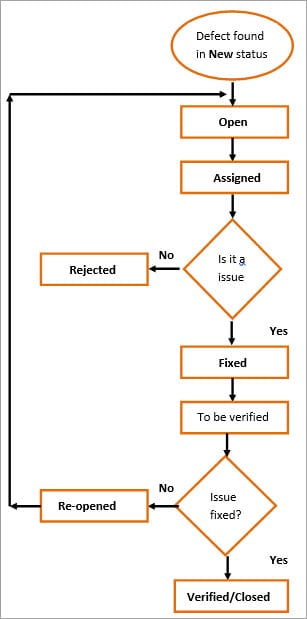
Generally, Defect life cycle starts from the stage when the defect is found or raised by the testing team and ends when a defect is closed either by ensuring that it’s not reproducible or rejected by a development team. The number of states that a defect goes through varies from project to project.

**Further reading:**

[*What is Defect/Bug Life Cycle in Software Testing? Defect Life Cycle Tutorial*](https://www.softwaretestinghelp.com/bug-life-cycle/)

**Note:** Below cycle slightly differs from organization to organization.

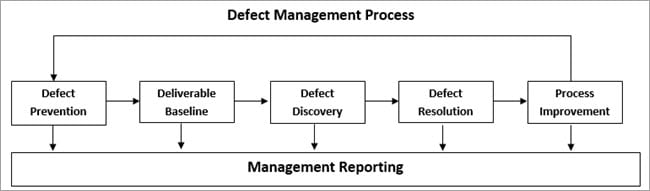
**The below defect life cycle covers all possible status:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2018/04/Defect-Life-Cycle-1.jpg)

* Whenever the testing team finds a defect in the application, they raise the defect with the status as “**NEW**”.
* When a new defect is reviewed by a QA lead and if the defect is valid, then the status of the defect would be “**Open**” and it is ready to be assigned to the development team.
* When a QA lead assigns the defect to the corresponding developer, the status of the defect would be marked as “**Assigned**”. A developer should start analyzing and fixing the defect at this stage.
* When the developer feels that the defect is not genuine or valid, then the developer rejects the defect. The status of the defect is marked as “**Rejected**” and assigned back to the testing team.
* If the defect logged is repeated twice or both the defects reported have similar results and steps to reproduce, then one defect status is changed to “**Duplicate**”.
* If there are some issues or hurdles in the current release for fixing a particular defect, then the defect would be taken in the upcoming releases instead of the current release and then it is marked as “**Deferred**” or “**Postponed**”.
* When a developer is not able to reproduce the defect by the steps mentioned in “Steps to Reproduce” by the testing team then the developer can mark the defect as “**Not Reproducible”**. In this stage, the testing team should provide detailed reproducing steps to a developer.
* If the developer is not clear about the steps to reproduce provided by a QA to reproduce the defect, then he/she can mark it as “**Need more information**”. In this case, the testing team needs to provide the required details to the development team.
* If a defect is already known and currently present in the production environment then the defect is marked as “**Known defect**”.
* When a developer makes the necessary changes, then the defect is marked as “**Fixed**”.
* The developer now passes the defect to the testing team to verify, so the developer changes the status as “**Ready for Retest**”.
* If the defect has no further issues and it is properly verified, then the tester marks the defect as “**Closed**”.
* While retesting the defect if the tester found that, the defect is still reproducible or partially fixed then the defect would be marked as “**Reopened**”. Now the developer has to look again into this defect.

A well planned and controlled Defect Life Cycle gives the total number of defects found in a release or in all releases. This standardized process gives a clear picture of how the code was written, how properly the testing has been carried out, how the defect or software has been released, etc. This will reduce the number of defects in production by finding the defects in the testing phase itself.

**Defect Management Process**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2018/04/Defect-Management-Process.jpg)

**Defect management process is explained below in detail.**

**#1) Defect Prevention:**

[Defect Prevention](https://www.softwaretestinghelp.com/defect-prevention-methods/) is the best method to eliminate the defects in the early stage of testing instead of finding the defects in the later stage and then fixing it. This method is also cost effective as the cost required for fixing the defects found in the early stages of testing is very low.

However, it is not possible to remove all the defects but at least you can minimize the impact of the defect and cost to fix the same.

**The major steps involved in Defect Prevention are as follow:**

* **Identify Critical Risk**: Identify the critical risks in the system which will impact more if occurred during testing or in the later stage.
* **Estimate Expected Impact**: For each critical risk, calculate how much would be the financial impact if the risk actually encountered.
* **Minimize expected impact**: Once you identify all critical risks, take the topmost risks which may be harmful to the system if encountered and try to minimize or eliminate the risk. For risks which cannot be eliminated, it reduces the probability of occurrence and its financial impact.

**#2) Deliverable Baseline:**

When a deliverable (system, product or document) reaches its pre-defined milestone then you can say a deliverable is a baseline. In this process, the product or the deliverable moves from one stage to another and as the deliverable moves from one stage to another, the existing defects in the system also gets carried forward to the next milestone or stage.

**For Example,** consider a scenario of coding, unit testing and then system testing. If a developer performs coding and unit testing then system testing is carried out by the testing team. Here coding and Unit Testing is one milestone and System Testing is another milestone.

So during unit testing, if the developer finds some issues then it is not called as a defect as these issues are identified before the meeting of the milestone deadline. Once the coding and unit testing have been completed, the developer hand-overs the code for system testing and then you can say that the code is **“baselined”** and ready for next milestone, here, in this case, it is “system testing”.

Now, if the issues are identified during testing then it is called as the defect as it is identified after the completion of the earlier milestone i.e. coding and unit testing.

Basically, the deliverables are baselined when the changes in the deliverables are finalized and all possible defects are identified and fixed. Then the same deliverable passes on to the next group who will work on it.

**#3) Defect Discovery:**

It is almost impossible to remove all the defects from the system and make a system as a defect-free one. But you can identify the defects early before they become costlier to the project. We can say that the defect discovered means it is formally brought to the attention of the development team and after analysis of that the defect development team also accepted it as a defect.

**Steps involved in Defect Discovery are as follows:**

* **Find a Defect**: Identify defects before they become a major problem to the system.
* **Report Defect**: As soon as the testing team finds a defect, their responsibility is to make the development team aware that there is an issue identified which needs to be analyzed and fixed.
* **Acknowledge Defect**: Once the testing team assigns the defect to the development team, its the development team’s responsibility to acknowledge the defect and continue further to fix it if it is a valid defect.

**#4) Defect Resolution:**

In the above process, the testing team has identified the defect and reported to the development team. Now here the development team needs to proceed for the resolution of the defect.

**The steps involved in the defect resolution are as follows:**

* **Prioritize the risk**: Development team analyzes the defect and prioritizes the fixing of the defect. If a defect has more impact on the system then they make the fixing of the defect on a high priority.
* **Fix the defect**: Based on the priority, the development team fixes the defect, higher priority defects are resolved first and lower priority defects are fixed at the end.
* **Report the Resolution**: Its the development team’s responsibility to ensure that the testing team is aware when the defects are going for a fix and how the defect has been fixed i.e. by changing one of the configuration files or making some code changes. This will be helpful for the testing team to understand the cause of the defect.

**#5) Process Improvement:**

Though in the defect resolution process the defects are prioritized and fixed, from a process perspective, it does not mean that lower priority defects are not important and are not impacting much to the system. From process improvement point of view, all defects identified are same as a critical defect.

Even these minor defects give an opportunity to learn how to improve the process and prevent the occurrences of any defect which may impact system failure in the future. Identification of a defect having a lower impact on the system may not be a big deal but the occurrences of such defect in the system itself is a big deal.

For process improvement, everyone in the project needs to look back and check from where the defect was originated. Based on that you can make changes in the validation process, base-lining document, review process which may catch the defects early in the process which are less expensive.

RCA of DEFECT   
  
**How to handle root cause analysis of software defects**

**Root cause analysis plays a major role in how software teams can fix their defects. Here's how to employ root cause analysis and how teams can get the most out of the process.**

****

By

* [**Stephen J. Bigelow,**](https://www.techtarget.com/contributor/Stephen-J-Bigelow)Senior Technology Editor

Published: **26 Jan 2021**

Enterprise software involves a complex interplay of instructions, data, associated services and dependencies. When software defects occur -- as they inevitably do -- developers must identify and understand the underlying reasons for those glitches.

[Root cause analysis](https://www.techtarget.com/searchitoperations/definition/root-cause-analysis) (RCA) for software defects is an approach developers use to better understand why a fault occurred and to take steps to drive improvements. The process is akin to how a medical team wants to diagnose and cure a patient's illness rather than simply treat the symptoms.

In a broad sense, root cause analysis is a process to identify underlying causes -- the *whys* -- of defects or failure events. Once the underlying cause is clear, a team can remediate the problem at its source. When software professionals perform the process properly, the team can use RCA results to improve product design, testing and overall quality. Let's take a closer look at some organizational benefits of root cause analysis for software defects.

**What are defects?**

From a software development perspective, a defect isn't just an error message or system crash because of a coding mistake. Defects are any deviation between an actual and expected result, such as when software works perfectly, but doesn't do what the user expects.

**This article is part of**

[**What is APM? Application performance monitoring guide**](https://www.techtarget.com/searchenterprisedesktop/definition/Application-monitoring-app-monitoring)

* Which also includes:
* [**5 benefits of APM for businesses**](https://www.techtarget.com/searchapparchitecture/feature/Learn-the-benefits-of-APM-software-in-the-enterprise)
* [**APM vs. observability: Key differences explained**](https://www.techtarget.com/searchapparchitecture/tip/APM-vs-observability-Key-differences-explained)
* [**Explore the 2022 application performance monitoring market**](https://www.techtarget.com/searchitoperations/feature/Explore-the-2022-application-performance-monitoring-market)

A defect represents a departure from the expectations [outlined in a software requirements specification](https://www.techtarget.com/searchsoftwarequality/tip/How-to-write-and-structure-worthwhile-SRS-documentation#:~:text=A%20software%20requirements%20specification%20(SRS,for%20any%20software%20development%20project.&text=performance%20requirements%2C%20including%20speed%2C%20availability,resource%20limitations%20and%20database%20needs.). Defects also occur in live software when pre-production tests fail to detect functional or performance problems. Here are six examples:

1. **Errors, oversights or gaps in the original software requirements**. These defects can occur when a requirement is omitted or forgotten, phrased poorly, not properly understood by stakeholders or misunderstood by developers.
2. **Errors in the design or architecture of the software.** These problems occur when software designers create an inefficient software algorithm or process, or when that algorithm or process doesn't yield the required precision in its results.
3. **Errors in the coding or implementation.** These defects include traditional bugs caused by everything from missing brackets to ungraceful error handling.
4. **Errors in the test planning or test activities.**These defects stem from inadequately tested features and functions.
5. **Errors or oversights in the deployment.**An example of these defects would be when a team provisions inadequate VM resources.
6. **Errors in the process or policies a team uses to govern the development cycle**. These defects crop up when, for example, a team obtains signoffs or approvals without adequate design, coding or testing review.

Once root cause analysis discovers the issue, the team can take proactive steps to remediate the defect and prevent it from future occurrences. If the defect resulted from design error, for example, developers can review the design and requirements documents to make corrections. If a testing mistake caused the defect, developers can update the test cases and metrics.

**Troubleshooting vs. root cause analysis**

RCA and troubleshooting are different processes. Troubleshooting and general problem-solving methodologies solve specific problems. For example, if an [application's health monitoring](https://www.techtarget.com/searchitoperations/opinion/Forget-monitoring-alerts-turn-to-IT-root-cause-analysis) reveals that a software instance crashed and is unresponsive, the team may resolve the problem by restarting the software instance or rebooting the server.

Root cause analysis for software defects, however, might reveal that the software becomes unresponsive because of a certain error condition. Perhaps the application can't access data, and the software isn't designed to handle such errors gracefully. In response, the team can release a software patch that addresses the error handling and will likely prevent the problem from recurring.

**Benefits of root cause analysis for software defects**

Root cause analysis can save an organization money by helping to find and address problems earlier in the SDLC.

If a business uses RCA to find and fix problems earlier in the development cycle, the enterprise can create better quality software in a faster, more cost-effective way. Root cause analysis that prevents problems in live software also promotes customer satisfaction and protects company reputation. Some advantages of root cause analysis in software development include:

* lower software defect rates;
* improved software quality (eliminates the same defects and repetitive mistakes);
* reduced development costs;
* shortened development cycles by reducing troubleshooting fixes and remediations;
* improved user and customer satisfaction;
* [improved developer productivity](https://www.techtarget.com/searchsoftwarequality/tip/3-dramatic-ways-to-increase-developer-productivity) (allows a team to focus its effort on new features and improvements rather than fixes); and
* identification of problems elsewhere in the development and production environments.

**How to perform a root cause analysis**

A team can perform RCA in a wide variety of ways, but an organized, logical and objective approach is usually considered most appropriate and effective. The analysis will typically examine log data, help desk and trouble ticket details, and other evidence from an incident. As an RCA team scrutinizes this information, its members can begin to understand a defect's underlying causes and formulate strategies and recommendations to address them.

For the purposes of this discussion, consider an RCA team to be any group that gathers to discuss or determine root causes in search of corrective actions.

**Prepare for the meeting.** RCA meetings can be held as needed -- perhaps in the wake of an unexpected, critical fault -- or as regularly scheduled occurrences within the software development team. The RCA team leader will usually gather details and data about each fault, including logs, screenshots, reporting and other resources.

RCA team members can include representatives involved in each stage of the software's lifecycle, such as requirements, design, implementation, testing, operations and anybody else involved in development. The team can also consist of individuals who worked on and fixed the initial problem. Each RCA team member reviews the details and comes to the meeting prepared to discuss the issue from their own lifecycle stage.

**Blameless reporting and recommendations**

Root cause analysis for software defects only has value if a team objectively receives and implements RCA results. The biggest challenges with RCA initiatives involve the human concepts of [blame perception](https://about.gitlab.com/handbook/customer-success/professional-services-engineering/workflows/internal/root-cause-analysis.html) and responsibility assignment. In other words, no one wants it on the record that a defect was their fault. Unfortunately, when an analysis points fingers, the resentment and morale loss that follows can often undermine the benefits of root cause analysis, and in turn lead to resistance from team members, managers and business leaders.

It's crucial that all RCA efforts include blameless objectivity. Reporting and recommendations should always be framed as actionable steps that don't solely place the blame on an individual or team. When reporting and recommendations are blameless, a team is more likely to receive and implement changes without resentment or resistance.

**Define the problem.** With details available, the RCA team can meet to collectively assess the defect and its effect on the software. This phase of the discussion focuses on what happened by answering a variety of common analytical questions, including:

* What is the problem?
* What events or triggers led to the issue?
* What systems or services did the issue affect?
* How long did the issue last?
* What effects did the issue have?
* Who (if anyone) was involved?

**Brainstorm the underlying causes.** After RCA team members review the evidence and clearly define the problem, they can consider the possible root cause or causes. Focus on *why* the defect happened and [brainstorm with tools](https://www.computerweekly.com/blog/CW-Developer-Network/AI-developer-toolset-series-Applitools-on-AI-assisted-root-cause-analysis) to identify the root cause. The RCA team leader typically moderates this part of the meeting and ensures that all members can contribute ideas.

**Select corrective action.** Once the RCA team identifies the likely root cause of a defect, it can decide on the most appropriate root cause corrective action. Here, the team should determine how to address the underlying causes. Corrective actions can vary dramatically depending on the RCA finding, such as updating requirements, enforcing coding styles and standards, making specific changes or fixes to the software, adding test cases or making changes to the deployment environment.

The team should decide if it will add to the codebase fixes already made at the software level and if those changes require retesting. Be sure that a fix doesn't affect any other features and functionality.

**Select preventative action.** The real value of root cause analysis is ongoing improvement. Defects cost money to find and fix. By understanding the underlying cause of a problem, an RCA team's recommendations can show how to prevent similar problems in the same or other applications. The final part of an RCA process should result in explicit guidance on how to prevent similar issues from recurring. These suggestions are known as root cause preventative actions (RCPAs) and may involve a wide range of ongoing recommendations such as better documentation, more team training or skill set enhancements, process changes or IT infrastructure improvements.

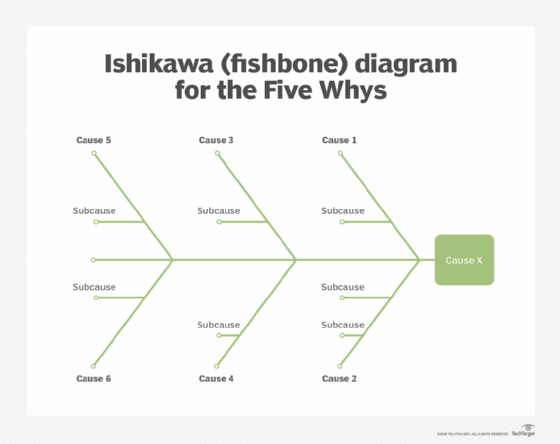
**Approaches to root cause analysis for software defects**

Software teams can draw from numerous tools to approach root cause analysis tasks, including:

* [fishbone diagrams](https://www.techtarget.com/whatis/definition/fishbone-diagram)
* [Five Whys](https://www.techtarget.com/whatis/definition/5-Whys)
* scatter plots
* [Failure Mode and Effects Analysis (FMEA)](https://www.techtarget.com/searchcio/definition/FMEA-failure-mode-and-effective-analysis)
* Pareto charts

Fishbone diagrams and Five Whys are the most popular techniques.

**Fishbone analysis.** A fishbone analysis -- also called an Ishikawa diagram or cause-and-effect diagram -- is designed to help analysts visualize a root cause by sorting possible causes into categories that branch off from the original issue. The resulting diagram resembles the skeleton of a fish -- thus the name. In practice, the underlying problem or issue is written at the "head" of the fish. The visual's "bones" are categories of possible causes. Analysts then identify the primary causes under each category; if necessary, the diagram-makers can add secondary and tertiary causes.

Example of Fishbone (or Ishikawa) diagram

**Five Whys analysis.** Asking *why* allows people to drill down into successive layers of a problem. The answer to each why becomes the basis for the next successive question. The process is similar to a child asking successive why questions -- each time the adult answers, the child uses that answer as the basis for the next question. The technique relies on brainstorming.

Five Whys analysis can be subjective since it doesn't use data or statistics, so the approach isn't suited for complex cases. The analysis may also demand more than five questions to reach a root cause, but five questions is often a starting point. Consider a simple example with just four "whys":

Problem: *The log file from a software application is missing.*

* Why is the log file missing?
  + The log file is not present in the logical unit number or folder where it was anticipated.
* Why was the log file not present?
  + The log file was not enabled in the software application.
* Why was the log feature not enabled?
  + The software application was not configured properly.
* Why was the software not configured properly?
  + A team inadequately documented the application or failed to complete a process to set up and use the software. The ultimate answer may be to enable the log and provide better documentation and user training.

MAVEN

Here's a step-by-step guide to create a Java project using the serenity-cucumber-archetype, compile, test, and package it using Maven from both an IDE and the command line.

**Creating a Java Project from an IDE (e.g., IntelliJ IDEA)**

1. **Open IntelliJ IDEA:**
   * Launch the IDE and select "New Project."
2. **Select Project Type:**
   * Choose "Maven" from the list on the left.
   * Select "Create from archetype."
3. **Select Archetype:**
   * In the list of archetypes, click on "Add Archetype" and fill in the details as follows:
     + **Group Id:** net.serenity-bdd
     + **Artifact Id:** serenity-cucumber-archetype
     + **Version:** (Choose the latest version available)
4. **Project Details:**
   * Fill in the GroupId, ArtifactId, and Version for your new project.
   * Choose a name for your project and set the project location.
5. **Finish the Setup:**
   * Click "Finish" to create the project. IntelliJ will set up the Maven project structure and download dependencies.
6. **Add Cucumber Feature Files:**
   * Navigate to the src/test/resources directory and create a features folder.
   * Create a Cucumber feature file (e.g., sample.feature) in the features directory.

**Creating a Java Project from Command Line**

1. **Open Terminal:**
   * Open your command line or terminal application.
2. **Create Maven Project:**
   * Run the following command to create a new Maven project using the serenity-cucumber-archetype:

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=my-serenity-project -DarchetypeArtifactId=serenity-cucumber-archetype -DinteractiveMode=false

1. **Navigate to Project Directory:**

bash

Copy code

cd my-serenity-project

**Compile, Test, and Package the Project Using Maven from Command Line**

1. **Compile the Project:**
   * Run the following command to compile the project:

bash

Copy code

mvn compile

1. **Run Tests:**
   * To run tests, execute:

bash

Copy code

mvn test

1. **Package the Project:**
   * Finally, package the project with:

bash

Copy code

mvn package

**Summary**

* You have created a Java project using the serenity-cucumber-archetype both from an IDE (IntelliJ IDEA) and the command line.
* You compiled, tested, and packaged the project using Maven from the command line.

**Note**

Make sure you have Maven installed and configured on your machine, and the JAVA\_HOME environment variable is set properly for Java development. You can verify your Maven installation by running mvn -v in the terminal.

**Task 02\_Hometask\_Maven**

To configure the Maven Surefire Plugin to run TestNG tests and set up the Maven Reporting Plugin to generate test reports, follow the steps below.

**Step 1: Add Dependencies**

First, ensure that your pom.xml file includes the necessary dependencies for TestNG. Here’s how to do that:

xml

Copy code

<dependencies>

<!-- Other dependencies -->

<!-- TestNG Dependency -->

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>7.7.0</version> <!-- Use the latest version available -->

<scope>test</scope>

</dependency>

</dependencies>

**Step 2: Configure the Maven Surefire Plugin**

Next, configure the Maven Surefire Plugin in your pom.xml to run TestNG tests. Add the following plugin configuration inside the <build> section:

xml

Copy code

<build>

<plugins>

<!-- Other plugins -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version> <!-- Use the latest version available -->

<configuration>

<suiteXmlFiles>

<suiteXmlFile>src/test/resources/testng.xml</suiteXmlFile> <!-- Path to your TestNG XML file -->

</suiteXmlFiles>

</configuration>

</plugin>

</plugins>

</build>

**Step 3: Create a TestNG XML File**

Create a testng.xml file in the src/test/resources directory to define your TestNG test suite. Here’s an example structure:

xml

Copy code

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name="MyTestSuite">

<test name="MyTests">

<classes>

<class name="com.example.tests.MyTestClass"/> <!-- Your test class -->

</classes>

</test>

</suite>

**Step 4: Configure the Maven Reporting Plugin**

To generate Maven test reports, add the Maven Reporting Plugin configuration to your pom.xml:

xml

Copy code

<build>

<plugins>

<!-- Other plugins -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-site-plugin</artifactId>

<version>3.9.1</version> <!-- Use the latest version available -->

<configuration>

<reportPlugins>

<reportPlugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-report-plugin</artifactId>

<version>3.0.0-M5</version> <!-- Use the latest version available -->

</reportPlugin>

</reportPlugins>

</configuration>

</plugin>

</plugins>

</build>

**Step 5: Run the Tests and Generate Reports**

Now you can run your tests and generate reports using the following Maven commands:

1. **Run Tests:**

bash

Copy code

mvn clean test

1. **Generate Reports:** After running the tests, generate the site documentation (including the reports) with:

bash

Copy code

mvn site

**Accessing the Reports**

After running the mvn site command, you can find the test reports in the target/site/surefire-report.html file. Open this file in a web browser to view the test results.

**Summary**

* You have configured the Maven Surefire Plugin to run TestNG tests.
* You have set up the Maven Reporting Plugin to generate test reports.
* You can now easily run your tests and view the results in a structured format.

4o mini

Top of Form

Bottom of Form