Department of Computer Engineering

Academic Term: First Term 2023-24

Class: T.E /Computer Sem – V / Software Engineering

Practical No:	9
Title:	White Box Testing
Date of Performance:	
Roll No:	9607
Team Members:	Sanika Patankar, Lisa Gonsalves, Eden Evelyn Charles

Rubrics for Evaluation:

Sr. No.	Performance Indicator	Excellent	Good	Below Average	Total Score
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not On Time)	
2	Theory Understanding (02)	02 (Correct)	NA	01 (Tried)	
3	Content Quality (03)	01 (All used)	02 (Partial)	03 (Rarely allowed)	
4	Post Lab Questions (04)	04 (Done Well)	03 (Partially Correct)	02 (Submitted)	

Signature of the Teacher:

Lisa Gonsalves
9607
TE COMPS B

SE EXP 9: White Box Testing

CODE: The following code is used for the shake feature in our app.

```
package com.example.miniproject;
import static android.location.LocationManager.GPS_PROVIDER;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import android.Manifest;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.net.Uri;
import android.os.Bundle;
import android.os.Handler;
import android.os.Vibrator;
import android.widget.Toast;
import android.telephony.SmsManager;

public class Shakepage extends AppCompatActivity implements SensorEventListener {

    private LocationManager locationManager;
    private LocationListener locationListener;
    private SensorManager msensorManager;
    private  Sensor mAccelerometer;

    private static final int MY_PERMISSIONS_REQUEST_CALL_PHONE=1;
    private float mAcceleration;
    private float mAccelerationCurrent;
    private float mAccelerationLast;
    private long mShakeTime;
    private static final int SHAKE_THRESHOLD=50;
    private static final int SHAKE_TIMEOUT=2000;
    private Vibrator mVibrator;
    private final Handler mHandler=new Handler();
    private final int mInterval=10*1000; //5mins in milliseconds
    private final Runnable mRunnable=new Runnable() {
    @Override
    public void run() {
```

```
        openAct();
        mHandler.postDelayed(this,mInterval);
        }
        };
        public Shakepage() {
        }
        private void openAct(){
        Intent i=new Intent(Shakepage.this,check.class);
        startActivity(i);
        finish();
        }

        @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_shakepage);
        mHandler.postDelayed(mRunnable,mInterval);
        msensorManager=(SensorManager) getSystemService(SENSOR_SERVICE)
mAccelerometer=msensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

msensorManager.registerListener(this,mAccelerometer,SensorManager.SENSOR_DELAY_
NORMAL);
        mAcceleration=0.00f;
        mAccelerationCurrent=SensorManager.GRAVITY_EARTH;
        mAccelerationLast=SensorManager.GRAVITY_EARTH;
        mVibrator=(Vibrator) getSystemService(VIBRATOR_SERVICE);
        locationManager=(LocationManager) getSystemService(LOCATION_SERVICE);

        locationListener=new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {

        }
        @Override
        public void onStatusChanged(String provider, int status, Bundle extras) {
        }
        @Override
        public void onProviderEnabled(String provider) {
        }
        @Override
        public void onProviderDisabled(String provider) {

        }
        };
        }

        @Override
        public void onSensorChanged(SensorEvent event) {
        float x=event.values[0];
        float y=event.values[1];
        float z=event.values[2];

        mAccelerationLast=mAccelerationCurrent;
        mAccelerationCurrent=(float) Math.sqrt((double) (x*x+y*y+z*z));
        float delta =mAccelerationCurrent-mAccelerationLast;
```

```
        mAcceleration=mAcceleration*0.9f+delta;
        if(mAcceleration>SHAKE_THRESHOLD){
        long now=System.currentTimeMillis();
        if(mShakeTime+SHAKE_TIMEOUT>now){
                return;
        }
        mShakeTime=now;
        mVibrator.vibrate(200);
        Toast.makeText(this, "SHAKE DETECTED", Toast.LENGTH_SHORT).show();
        if (ContextCompat.checkSelfPermission(Shakepage.this,
android.Manifest.permission.CALL_PHONE) ==
PackageManager.PERMISSION_GRANTED) {
                // The app has permission to make phone calls, call the number
                callPhoneNumber();
                if (ActivityCompat.checkSelfPermission(Shakepage.this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED &&
ActivityCompat.checkSelfPermission(Shakepage.this,
Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(Shakepage.this, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION,
Manifest.permission.ACCESS_COARSE_LOCATION}, 1);
                } else {
                Location location =
locationManager.getLastKnownLocation(GPS_PROVIDER);
                double latitude = location.getLatitude();
                double longitude = location.getLongitude();
                String message = "ALERT!!! HELP ME I AM HERE \nLatitude: " + latitude +
"\nLongitude: " + longitude;
                // Get the default instance of SmsManager
                SmsManager smsManager = SmsManager.getDefault();
// Specify the phone number and message text
                String phoneNumber = "8788397498";

// Use SmsManager to send the message
                smsManager.sendTextMessage(phoneNumber, null, message, null, null);
                Toast.makeText(Shakepage.this, message, Toast.LENGTH_LONG).show();
                }
        } else {
                // Permission is not granted, request the permission
                ActivityCompat.requestPermissions(Shakepage.this,
                new String[]{Manifest.permission.CALL_PHONE},
                MY_PERMISSIONS_REQUEST_CALL_PHONE);
        }
        }
        }

        private void callPhoneNumber() {
        String phoneNumber = "8788397498" ;
        Intent intent = new Intent(Intent.ACTION_CALL);
        intent.setData(Uri.parse("tel:" + phoneNumber));
        startActivity(intent);
        }
        @Override
```

```
	public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
	super.onRequestPermissionsResult(requestCode, permissions, grantResults);
	if (requestCode == MY_PERMISSIONS_REQUEST_CALL_PHONE) {// If request is
cancelled, the result arrays are empty
	if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
		// Permission was granted, call the number
		callPhoneNumber();
	} else {
		// Permission denied, show a message to the user
		Toast.makeText(Shakepage.this,
		"Permission denied to make phone calls",
		Toast.LENGTH_SHORT).show();
		mShakeTime = 0;
	}
	}
	}
	@Override
	public void onAccuracyChanged(Sensor sensor, int accuracy) {
	}
	protected void onResume(){
	super.onResume();

msensorManager.registerListener(this,mAccelerometer,SensorManager.SENSOR_DELAY_
NORMAL);
	}
	protected void onPause(){
	super.onPause();
	msensorManager.unregisterListener(this);
	}
	protected void onDestroy(){
	super.onDestroy();
	mHandler.removeCallbacks(mRunnable);
	}
}
```

Let’s perform a more detailed analysis of statement coverage, branch coverage, and path coverage for the provided Java code.

1. Statement Coverage:
Statement coverage measures the percentage of executable code statements that have been executed during testing. It's a basic metric for assessing the completeness of your testing. To calculate Statement Coverage, we count the number of executed statements divided by the total number of executable statements in the code.

Let's count the statements:
- Total Executable Statements: 33 (This includes lines with actual code to be executed)
To calculate Statement Coverage, we need to know how many of these statements have been executed during testing.

2. Branch Coverage:
Branch coverage measures the percentage of decision points (branches) in the code that have been executed. A decision point is typically an "if" statement or a switch-case

statement.To calculate Branch Coverage, we count the number of executed branches divided by the total number of branches in the code.

- Total Branches: Counting branches requires a detailed examination of conditional statements (e.g., if statements). It depends on the specific test cases and inputs used during testing. Without that information, I cannot provide an exact number for total branches.

3. Path Coverage:
Path coverage is a more advanced metric that considers all possible execution paths through the code. It provides insight into whether all logical paths have been tested.

Calculating Path Coverage is complex, as it depends on the number of conditional branches and nested conditions in the code. To determine Path Coverage, you need to analyse the logical paths and combinations of conditions tested during your testing. This involves creating a control flow graph and tracing which paths have been executed.

In practice, achieving 100% coverage for all these metrics is often challenging and may not be necessary, as it depends on project constraints and testing goals. The completeness of your testing should be based on the criticality of the code and the associated risks.

POSTLABS :

a. Generate white box test cases to achieve 100% statement coverage for a given code snippet.

Here's a simple Python code snippet:

```
def divide(a, b):
    if b == 0:
        return "Division by zero is not allowed"
    result = a / b
    return result
def is_positive(num):
    return num > 0
```

To achieve 100% statement coverage, we need to ensure that every line of code is executed at least once. Here are the test cases:

```
Test Case 1: Valid division
a = 10
b = 2
result = divide(a, b)
assert result == 5  # Check if the result is correct

Test Case 2: Division by zero
a = 5
b = 0
result = divide(a, b)
assert result == "Division by zero is not allowed"  # Check the error message

Test Case 3: Check if a number is positive
num = 7
assert is_positive(num)  # Check if the function correctly identifies a positive number
```

Test Case 4: Check if a number is not positive
num = -3
assert not is_positive(num) # Check if the function correctly identifies a non-positive number

By executing these four test cases, you will achieve 100% statement coverage for the provided code snippet. Each line of code is executed at least once in these tests. Keep in mind that this is a simplified example, and real-world code may require more comprehensive testing to achieve complete coverage and handle different scenarios and edge cases.

b. Compare and contrast white box testing with black box testing, highlighting their respective strengths and weaknesses in different testing scenarios.

White box testing and black box testing are distinct software testing approaches, each with its pros and cons. Here's a comparison of these methods in various testing scenarios:

White Box Testing:

- **Definition**: Focuses on examining the internal structure and code of the software, with access to the source code and design knowledge.
- **Strengths**:
    - Offers thorough statement, branch, and path coverage, vital for complex applications.
    - Effectively localises defects, aiding developers in issue identification and resolution.
    - Detects logic errors and inconsistencies in the code early in development.
- **Weaknesses**:
    - Testers may have biases due to code access, potentially influencing test case design.
    - May not ensure comprehensive testing of user scenarios and requirements.
    - Can be resource-intensive, particularly for large codebases.

Black Box Testing:

- **Definition**: Focuses on external software behavior without knowledge of the internal code, designing test cases based on specifications and requirements.
- **Strengths**:
    - Objective and unbiased testing, suitable for independent verification.
    - Aligns with end-users' perspectives and validates compliance with requirements.
    - Efficient for user scenario and large-scale testing without needing code knowledge.
- **Weaknesses**:
    - May lack comprehensive structural coverage, potentially missing edge cases and unmentioned code paths.
    - Doesn't pinpoint exact defect locations, potentially complicating debugging.
    - May not uncover critical code logic issues.

Scenarios for Each Approach:

- **White Box Testing is Ideal When**:
    - The software has complex algorithms that require in-depth testing.
    - Specific code paths and data flows need verification.
    - Strict regulatory or compliance requirements necessitate comprehensive code coverage.
- **Black Box Testing is Ideal When**:

- Validating user scenarios and requirements is the primary focus.
- Assessing the software from the end-user perspective is crucial.
- Independent verification is required, and internal code is proprietary or inaccessible.

c. Analyse the impact of white box testing on software quality, identifying its potential to uncover complex logic errors and security vulnerabilities.
White box testing significantly contributes to improving software quality by addressing complex logic errors and security vulnerabilities:

Uncovering Complex Logic Errors:

1. **Code Path Coverage**: White box testing provides comprehensive code path coverage, including rare and complex scenarios, effectively uncovering complex logic errors.
2. **Condition Testing**: It assesses different code conditions, loops, and decision points, identifying issues like incorrect branching, unexpected loops, and logic discrepancies, which are root causes of complex logic errors.
3. **Early Detection**: By identifying complex logic errors early, it prevents them from becoming critical defects, reducing costs and efforts.
4. **Regression Testing**: Valuable for regression testing, it ensures that code changes do not introduce new complex logic errors while maintaining existing functionality.

Uncovering Security Vulnerabilities:

1. **Source Code Analysis**: White box testing examines source code, crucial for identifying security vulnerabilities like input validation issues and authentication flaws.
2. **Security Scanning Tools**: Integrates security scanning tools to detect common vulnerabilities like SQL injection, XSS, and access control problems.
3. **Authentication and Authorization Testing**: Verifies correct implementation of authentication, authorization, and access control, reducing the risk of unauthorized data access.
4. **Secure Data Handling**: Ensures proper handling of sensitive data, preventing data leaks and integrity issues.
5. **Protection Against Common Attacks**: Evaluates the software's defense against common threats, such as CSRF and buffer overflows.
6. **Encryption and Data Protection**: Assesses the correct use of encryption for data in transit and at rest, reducing data breach risks.