Exploratory Testing

Software Testing: IN3240 / IN4240

Part I: Testing an application

Your Horoscope

You are now going to test a horoscope program that's sets your horoscope based on your date of birth.

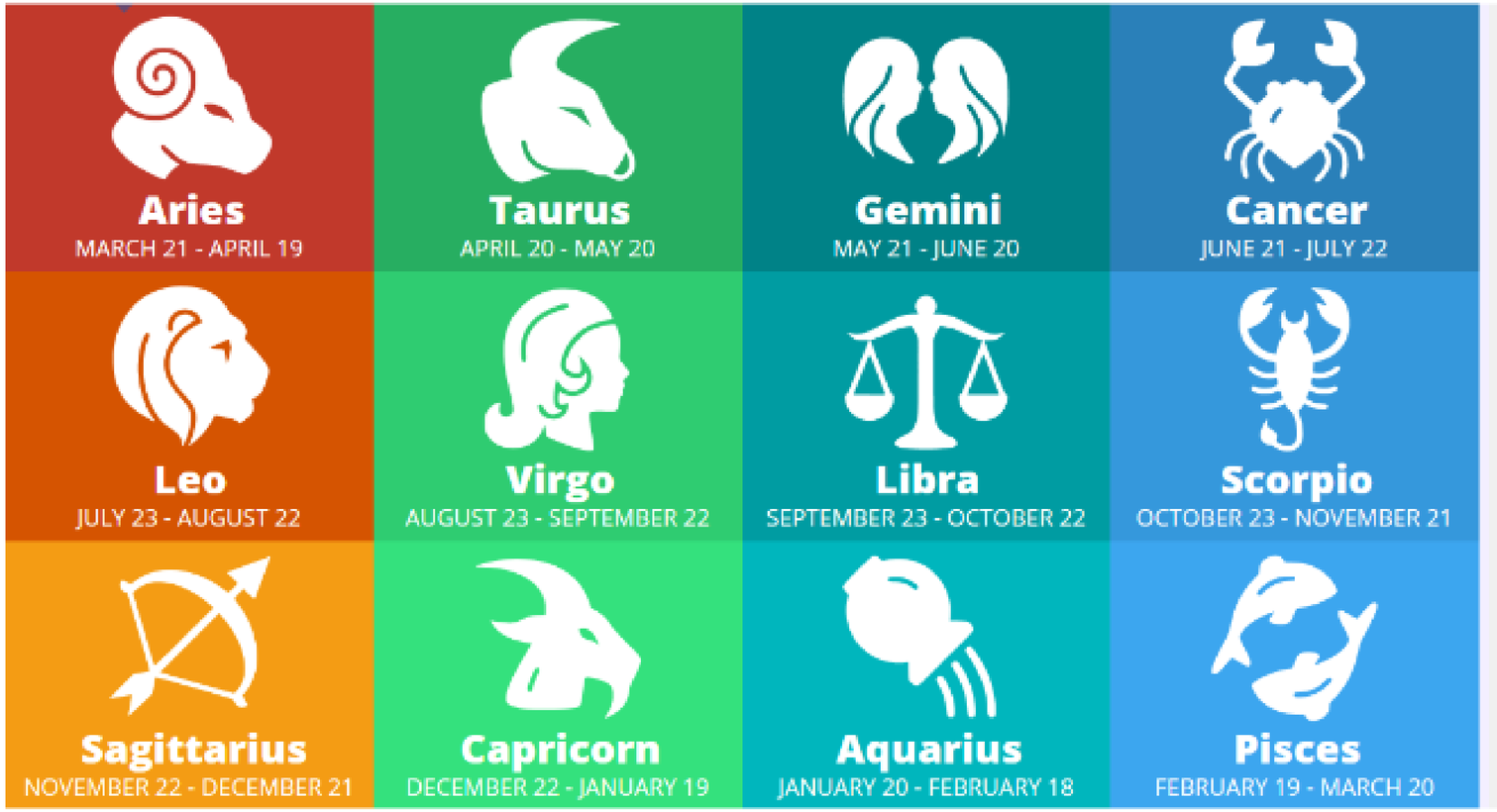
Click here to open and run the program.

Your Horoscope

Unfortunately, there are at least three bugs in the program that you shall try to detect. You do not have access to the test basis, except the zodiac signs defined in this table.

Clues:

Use Equivalence partitions and boundary value analysis!



Your Horoscope

The equivalence partitioning can be done in different ways on the same test object. Some of them will contain boundary values, others not. The following example from the textbook page 118 illustrates this:

We can also apply equivalence partitioning and boundary value analysis more than once to the same specification item. For example, if an internal telephone system for a company with 200 telephones has 3-digit extension numbers from 100 to 699, we can identify the following partitions and boundaries:

- digits (characters 0 to 9) with the invalid partition containing non-digits
- number of digits, 3 (so invalid boundary values of 2 digits and 4 digits)
- range of extension numbers, 100 to 699 (so invalid boundary values of 099 and 700)
- extensions that are in use and those that are not (two valid partitions, no boundaries)
- the lowest and highest extension numbers that are in use could also be used as boundary values

Your Horoscope

In which ways can you apply equivalence partitioning to the input of the horoscope program?

For each way, specify

- the equivalence partitions, both valid and invalid
- any boundary values

Hint: You don't need to list every equivalence partition and its boundary values. It is sufficient to describe them uniquely as sets, intervals or in words.

Part II: Close-ended questions

Question 1

Which of the following are good questions to ask oneself, in order to build quality in a software system?

I. Is the customer the same as the user?
II. How much can my customers afford to pay for my product?
III. Can I reduce the user roles even more, to reach a minimum number of user profiles?

a. I, II
b. I, III
c. II, III
d. I, II, III

Question 1: Answer

Which of the following are good questions to ask oneself, in order to build quality in a software system?

I. Is the customer the same as the user?

II. How much can my customers afford to pay for my product?

III. Can I reduce the user roles even more, to reach a minimum number of user profiles?

a. I, II

b. I, III

c. II, III

d. I, II, III

Question 2

Which of the following factors have most influence in determining which testing process to apply?

a. The tools used to report and fix bugs.

b. Product interfaces, project size.

c. The team's attitude in communicating software faults and failures.

d. Regular bug triage meetings.

Question 2: Answer

Which of the following factors have most influence in determining which testing process to apply?

a. The tools used to report and fix bugs.

b. Product interfaces, project size.

c. The team's attitude in communicating software faults and failures.

d. Regular bug triage meetings.

Question 3

Which of the following statement can, according to Cem Kaner, be used to define the term "Quality" of software?

a. The quality of software is to make a software bug free.

b. Quality software means that writing code to assert that other code returns some "correct" results.

c. Quality is value to some person(s).

d. Quality is an investigation of code, system, people and the relationship between them.

Question 3: Answer

Which of the following statement can, according to Cem Kaner, be used to define the term "Quality" of software?

a. The quality of software is to make a software bug free.

b. Quality software means that writing code to assert that other code returns some "correct" results.

c. Quality is value to some person(s).

d. Quality is an investigation of code, system, people and the relationship between them.

Question 4

Which of the following will be verified by testers, during the exploratory testing sessions?

I. Program features
II. Program data
III. Program interoperability
IV. Project management
V. Step-by-step test scenarios

a. V
b. I, II, III
c. III, IV
d. I, II, III, IV, V

Question 4: Answer

Which of the following will be verified by testers, during the exploratory testing sessions?

I. Program features
II. Program data
III. Program interoperability
IV. Project management
V. Step-by-step test scenarios

a. V
b. I, II, III
c. III, IV
d. I, II, III, IV, V

Question 5

Does software testing depend on the size of the software being tested?

YES / NO

Question 5: Answer

Does software testing depend on the size of the software being tested?

YES / NO

Question 6

Does software testing depend on the type of product being developed? (ex: experimental vs. life-critical vs. regulated software)

YES / NO

Question 6: Answer

Does software testing depend on the type of product being developed? (ex: experimental vs. life-critical vs. regulated software)

YES / NO

Question 7

_____ refers to experience-based techniques for problem solving, learning, and discovery that give a solution which is not guaranteed to be optimal.

Question 7: Answer

Heuristics refers to experience-based techniques for problem solving, learning, and discovery that give a solution which is not guaranteed to be optimal.

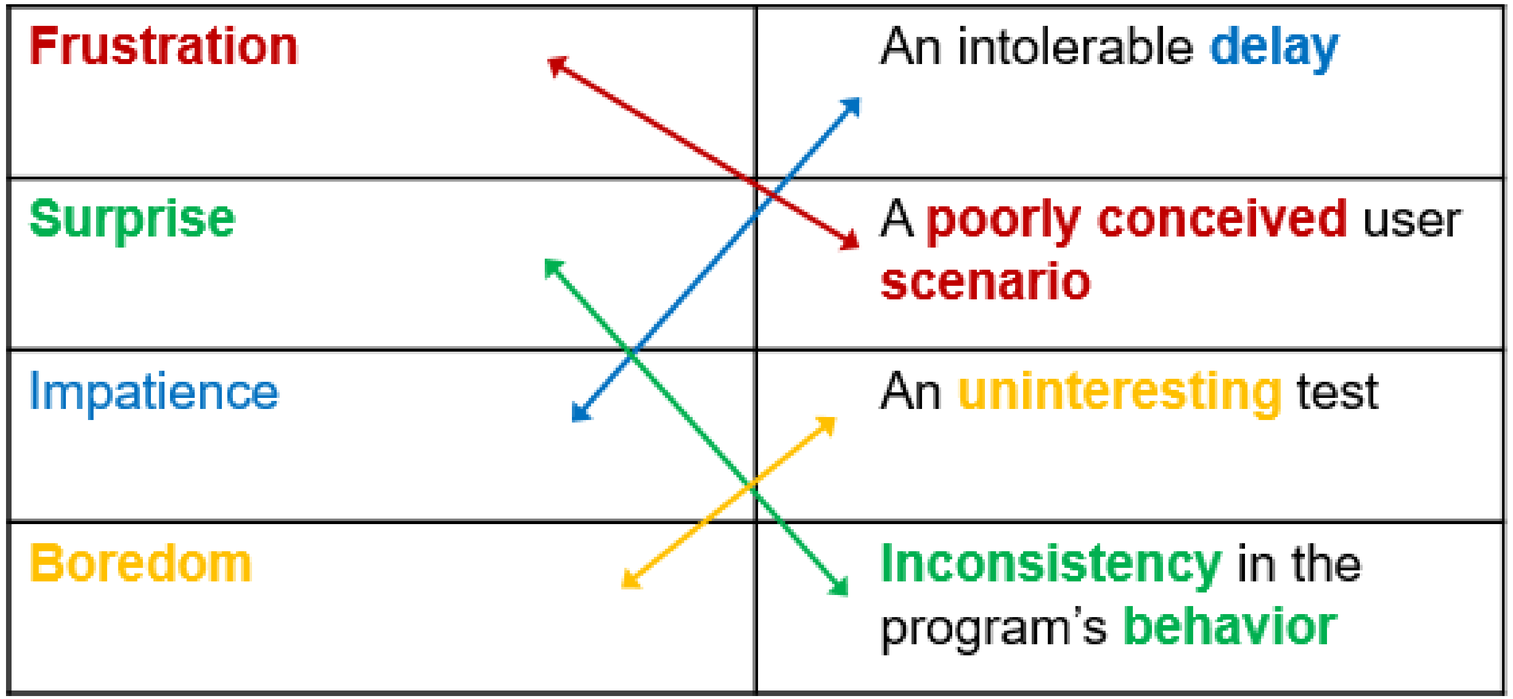
Question 8

Pair the following triggers for heuristics and their possible underlying issues:

Frustration	An intolerable delay
Surprise	A poorly conceived user scenario
Impatience	An uninteresting test
Boredom	An inconstancy in the program's behavior

Question 8: Answer

Pair the following triggers for heuristics and their possible underlying issues:



Part III: Exercises and Open-ended questions

Exercise 1

Video on what means exploratory testing:
https://www.youtube.com/watch?v=I-ItEKt_N_s

Unit testing

Software Testing: IN3240 / IN4240

Unit Testing – component testing

Unit testing, also known as Component testing verifies the modules of the software (e.g. classes, functions/methods, modules etc.) that are separately testable.

Unit Testing – component testing

The developer writes code to test modules in the software under test.

Unit test framework support the developer.

Unit testing should be done in isolation from the rest of the system.

Stubs and drivers are used to replace the missing software and simulate the interface between the software components.

Unit Testing – component testing

A stub is called from the software component to be tested.

A driver calls a component to be tested.

Test cases are derived from work products such as the software design or the data model

Unit tests and **test suites** for Java programs can be developed in an integrated development environment, e.g. Eclipse and Netbeans.

Exercise: Unit Testing

The Java program : PerfectNumbers.java finds perfect numbers up to a given limit.

- Use Eclipse to develop JUnit test cases for the three methods in the file *PerfectNumbers.java*.

- Create a JUnit test suite of all the test cases.

(To run the program, you must add the file *PerfectTest.java*.)

Exercise: Unit Testing

For an added challenge you can try to make the program yourself!

(If you want to run the program, you must add the file PerfectTest.java.)

If you need a Unit Test guide, see https://www.youtube.com/watch?v=v2F49zLLj-8

Exercise: Unit Testing

What is a perfect number?

An integer equal to the sum of all its real factors, including one (1)

Real factor means a factor less than the number itself

Integrer	Real factors	Sum	Perfect?
4	1, 2	1 + 2 = 3	No 3 ≠ 4
6	1, 2, 3,	1 + 2 + 3 = 6	Yes 6 = 6
12	1, 2, 3, 4 , 6	1+ 2+ 3+ 4+ 6 = 16	No 12 ≠ 16
28	1, 2, 4, 7, 14	1+2+4+7+14 = 28	Yes 28 = 28

Exercise: Unit Testing

PerfectNumbers.java

Calculates perfect numbers

perfect(int number): boolean

Is the given number perfect?

factorSum(int number): String

Calculate factor sum of number

findPerfectNumbers(int limit)

Find perfect numbers given limit

```
public class PerfectNumbers {

  public static boolean perfect( int number ) {
  int factorSum = 1;

  for ( int divisor = 2; divisor <= number / 2; divisor++ ) {
    if ( number % divisor == 0 )
      factorSum += divisor;
  }
  return (factorSum == number);
}

public static String factorSum( int number ) {
  String sum = "1";
  for ( int divisor = 2; divisor <= number / 2; divisor++ ) {
    if ( number % divisor == 0 ) {
      sum += " + " + divisor;
    }
  }
  return sum;
}

public static String findPerfectNumbers( int limit ) {
  String result = "perfect  number less or equals " + limit + "\n";
  for ( int i = 2; i <= limit; i++ ) {
    if ( perfect( i ) ) {
      result += i + " = " + factorSum( i ) + "\n";
    }
  }
  return result;
}
}
```

Exercise 2: Unit Testing

Testing *perfect(int number)*

What to test?

Confirm perfect number is perfect

Chosen number: 6

Variables

result → Holds the returned value

expected → Set to *true*

Assert

Check that the two values match

```
import static org.junit.Assert.*;
import org.junit.Test;

public class PerfectTest1 {

    @Test
    public void perfectTest1()  {

        boolean result = PerfectNumbers.perfect( 6 );

        boolean expected = true;

        assertEquals(result, expected);

    }
}
```

Exercise 2: Unit Testing

Testing *perfect(int number)*

What to test?

Confirm non-perfect is non-perfect

Chosen number: 7

Variables

result → Holds the returned value

expected → Set to *false*

Assert

Check that the two values match

```
import static org.junit.Assert.*;
import org.junit.Test;

public class PerfectTest2 {

    @Test
    public void perfectTest2() {

        boolean result = PerfectNumbers.perfect( 7 );

        boolean expected = false;

        assertEquals(result, expected);

    }
}
```

Exercise 2: Unit Testing

Testing *factorSum(int number)*

What to test?

Confirm correct sum of factors

Chosen number: 6

Variables

result → Holds factor sum of 6

expected → Set to *"1 + 2 + 3"*

Assert

Check that the two values match

```
import static org.junit.Assert.*;
import org.junit.Test;

public class FactorSumTest {

    @Test
    public void test() {

        String result = PerfectNumbers.factorSum( 6 );

        String expected = "1 + 2 + 3";

        assertEquals(expected, result);
    }
}
```

Exercise 2: Unit Testing

Testing *findPerfectNumbers(int limit)*

What to test?

Confirm correct retrieval of PN

Chosen number: 1000

Variables

result → Holds all PN within limit

expected → Set to 6, 28, and 496

Assert

Check that the two values match

```
import static org.junit.Assert.*;
import org.junit.Test;

public class FindPerfectNumberTest {

    @Test
    public void findPerfectNumberTest() {


        String result = PerfectNumbers.findPerfectNumbers( 1000 );

        String expected = "perfect  number less or equals 1000" +
                          "\n6 = 1 + 2 + 3" +
                          "\n28 = 1 + 2 + 4 + 7 + 14" +
                          "\n496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248\n";


        assertEquals(expected, result);
    }

}
```

Exercise 2: Unit Testing

JUnit Test Suite for all test cases

Where to place test suite?

AllTests.java

@RunWith(Suite.class)

What to include?

PerfectTest1.java

PerfectTest2.java

FactorSumTest.java

FindPerfectNumberTest.java

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)

@SuiteClasses({ FactorSumTest.class, FindPerfectNumberTest.class,
                PerfectTest1.class, PerfectTest2.class})

public class AllTests { }
```