Software Quality Assurance and Testing (SQAT)
Black-box testing (pt.1)

dr. Joost Schalken-Pinkster
Windesheim University of Applied Science
The Netherlands

joost@schalken.me

The contents of these course slides is (in great part) based on:
Neil Taylor (2019) *Course presentations for Software Quality Assurance and Testing (SQAT)*, Aberystwyth University.

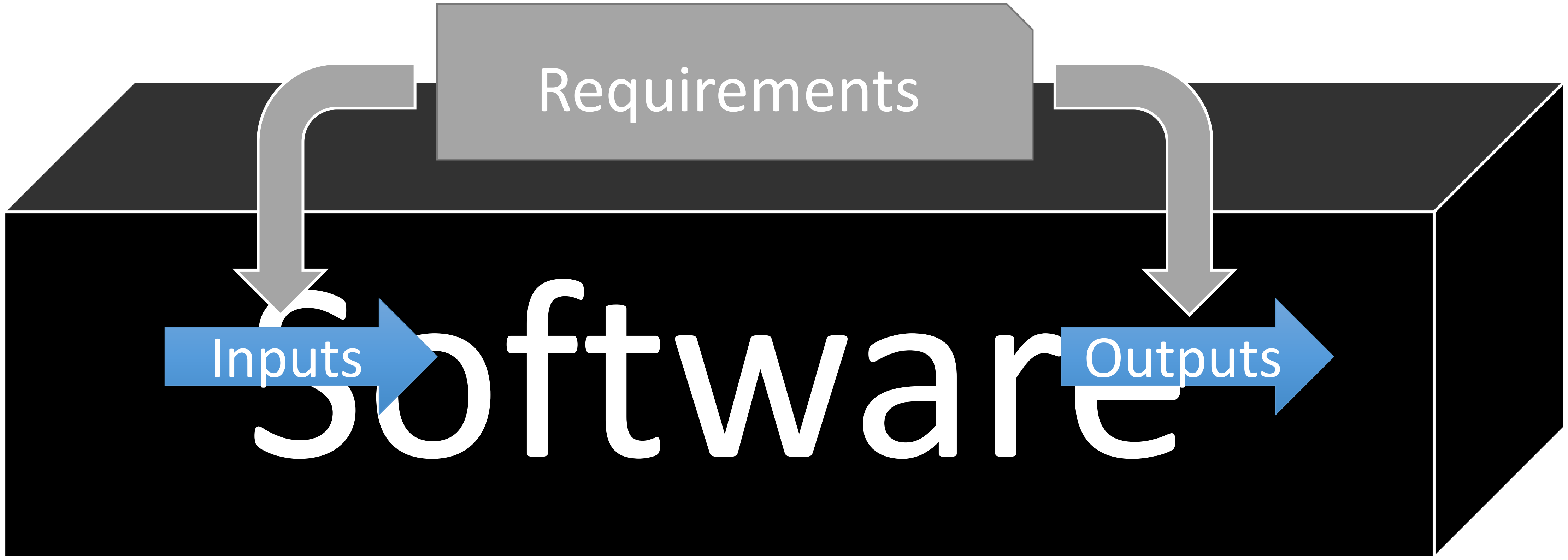
Introduction

- How do we identify test cases?
- One general approach:
  - Black-box testing – testing based on a specification
- Techniques including:
  - Equivalence Partitions
  - Boundary Value Analysis
- Examples

Learning Outcomes

- Discuss the role of Black Box Testing, including any advantages or disadvantages
- Discuss the term Equivalence Partition and how this can be used to identify some test cases.

Software as a black box

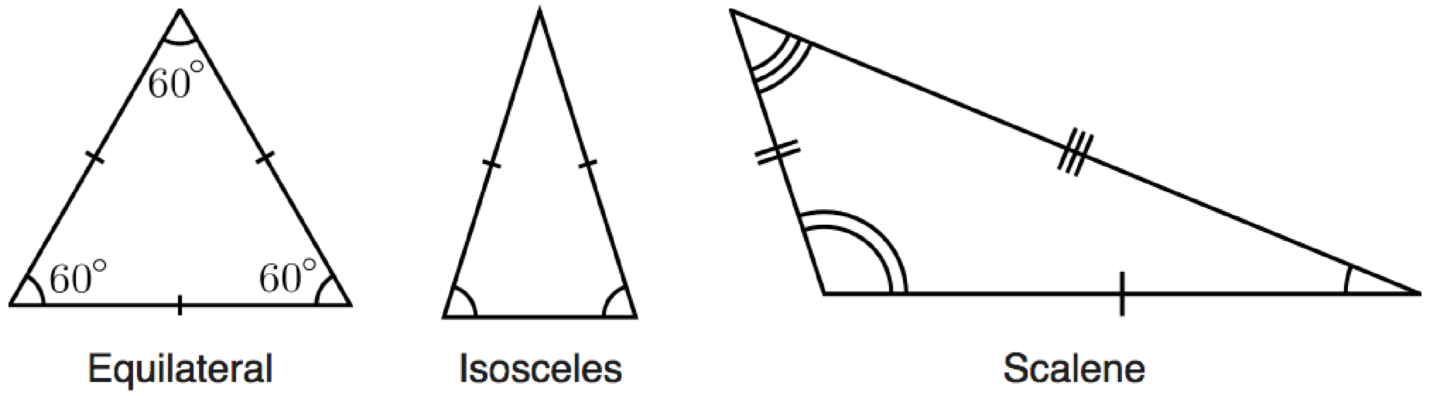


Triangle Example

Description

A Shape module has a feature to return a value to indicate if a triangle is one of the types “Equilateral”, “Isosceles” or “Scalene”.

The feature takes three values which represent the lengths of the three sides for the triangle.



Triangle Example

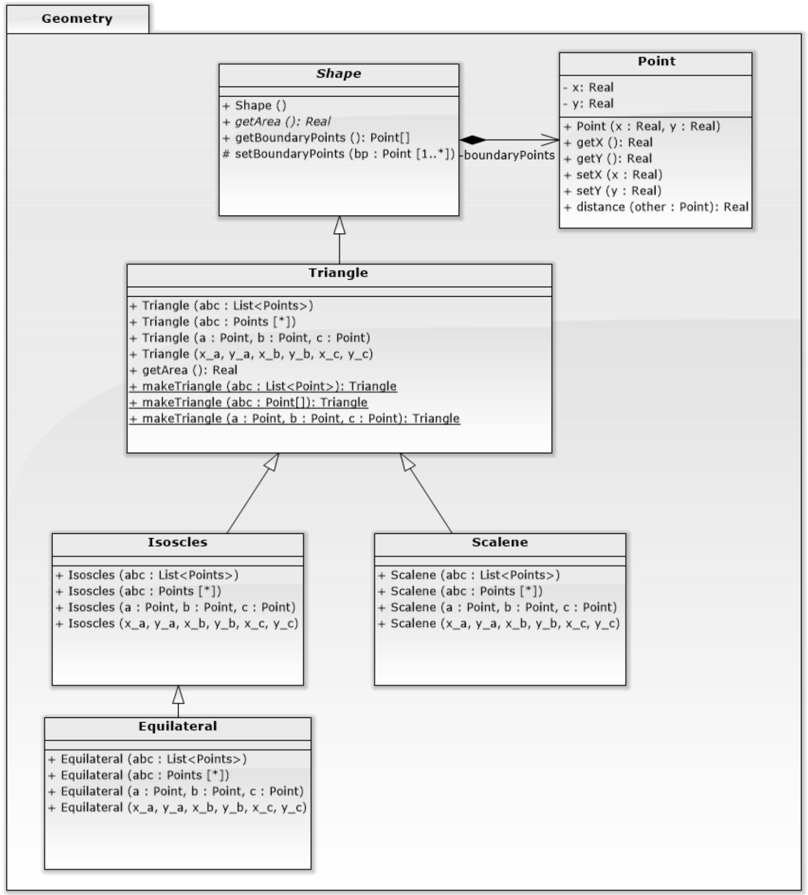
Rules

- Equilateral triangle is one that has three equal sides.
- Isosceles triangle is one that has at least two equal sides.
- Scalene is one where no sides have matching length.

- The values represent a triangle if the sum of the lengths of any two sides is greater than the length of the remaining side.

Triangle Example

Can you make a UML class diagram for a design for this problem?


Geometry
Shape
+ Shape ()
+ getArea (): Real
+ getBoundaryPoints (): Point[]
# setBoundaryPoints (bp : Point [1..*])
-boundaryPoints
Point
- x: Real
- y: Real
+ Point (x : Real, y : Real)
+ getX (): Real
+ getY (): Real
+ setX (x : Real)
+ setY (y : Real)
+ distance (other : Point): Real
Triangle
+ Triangle (abc : List<Points>)
+ Triangle (abc : Points [*])
+ Triangle (a : Point, b : Point, c : Point)
+ Triangle (x_a, y_a, x_b, y_b, x_c, y_c)
+ getArea (): Real
+ makeTriangle (abc : List<Point>): Triangle
+ makeTriangle (abc : Point[]): Triangle
+ makeTriangle (a : Point, b : Point, c : Point): Triangle
Isoscles
+ Isoscles (abc : List<Points>)
+ Isoscles (abc : Points [*])
+ Isoscles (a : Point, b : Point, c : Point)
+ Isoscles (x_a, y_a, x_b, y_b, x_c, y_c)
Scalene
+ Scalene (abc : List<Points>)
+ Scalene (abc : Points [*])
+ Scalene (a : Point, b : Point, c : Point)
+ Scalene (x_a, y_a, x_b, y_b, x_c, y_c)
Equilateral
+ Equilateral (abc : List<Points>)
+ Equilateral (abc : Points [*])
+ Equilateral (a : Point, b : Point, c : Point)
+ Equilateral (x_a, y_a, x_b, y_b, x_c, y_c)

Triangle Example

Can you implement this design?

(Shape, Point, Triangle, Isoscles)

Triangle Example

Can you think how you would this code?

Triangle Example

A Shape module has a feature to return a value to indicate if a triangle is one of the types “Equilateral”, “Isosceles” or “Scalene”.

The feature takes three values which represent the lengths of the three sides for the triangle.

Writing tests based on the above description would be Black-box testing.

- You don’t know how this is implemented.
- Test possible inputs and expected outputs.

Advantages of Black-box Testing

What advantages are there with Black-box testing?

Advantages of Black-box Testing

What advantages are there with Black-box testing?

You don't need to know how it is implemented

- You can plan tests earlier
- You can write tests earlier
- You do not need to be the person who implements the code
- When the implementation changes, the tests should still work

How do you decide what values to test?

How many values would you need to test, to test all possible values of:

- `public static Triangle makeTriangle(Point a, Point b, Point c)`

- For many methods, you cannot test all possible values
- If you don't test all values, how do you know if your tests cover all of the possibilities?

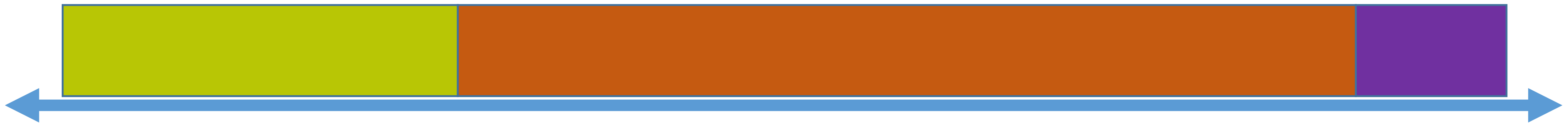
- You use different techniques to identify good test values:
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Decision Tables
  - Cause-Effect Graphing
  - Error guessing
  - …

Equivalence Partitioning

Input data and output results often fall into different classes, where all members of the class are related.

Each of these classes is an **equivalence partition** or **domain** where the program behaves in an equivalent way for each class member.

Test cases should be chosen from each partition.



Range of a value

Choosing Classes (1)

Valid equivalence classes

- This would be sets of data where all values can reasonably be treated as “the same”
- They can be used to check that the application carries out the predicted functions correctly

Example:

- In a game, a player has an amount of health points – represented as whole numbers between 0 and 100. The following rules apply:
  - A Good number health points is 40 or more
  - A Low number of health points is between 15 and 39
  - A Dangerous number of health points is below 14
- What classes can we create for this data?

Choosing Classes (2)

Invalid equivalence classes:

- This would be data that the specification describes as invalid
  - What about the situation where the specification doesn't clearly identify invalid data?
- May need more checking if a method takes user input.
- Makes sure that the invalid data is processed correctly.
- Even if no user input, still need to worry about sensible data for the test cases.

Guidelines for variables (1)

If variable is a **number** with a legal range:

- You have one equivalence class for the legal range and two equivalence classes for illegal ranges.
- For example: there is a method that uses the age of people in work. The legal range is between 18 and 67 in the NL.

If variable is a **string**:

- You have at least one class containing all legal strings and one containing all illegal strings.
- A legal (valid) string might have a maximum (and minimum) length and there might be a required format to the string, e.g. a phone number.

Guidelines for variables (2)

If variable is an **enumerated type**:

- Each value is a separate equivalence class
- For example: a valid set of colours for a car might be **red**, **yellow** and **white**.

If variable is an **array**:

- One equivalence class containing all legal arrays, one containing only the empty array and one containing arrays larger than the expected size.
- Depending upon the language, is there a difference between an empty array and **null**?

Designing Reasonable Test Cases

- Identify the equivalence classes.
- Design a new test case, with a unique ID, that covers a valid equivalence class which has not yet been tested.
  - Repeat this step until all valid equivalence classes are included
- Design a new test case, with a unique ID, that covers an invalid equivalence class which has not yet been tested.
  - Repeat this step until all invalid equivalence classes are included

Summary

- Definition of Black Box Testing
  - Testing, based on the specification
- Advantages of Black Box testing
- Equivalence Partitioning as a technique to decide on tests values
  - Valid and invalid equivalence classes
- Approach to developing test cases based on equivalence partitioning

Any Questions?