

# CS608

# Software Testing

Dr. Stephen Brown

Room Eolas 116

[stephen.brown@mu.ie](mailto:stephen.brown@mu.ie)

# Tutorial: Lab 6

- OO/DT testing for `SpaceOrder.acceptOrder()`
- Testing in class context
- Combinational testing using DTs

# CS608

## Testing Object-Oriented Software in More Detail

(Essentials of Software Testing, Chapter 9, Sections 9.4-9.7)

# A More Detailed Look at OO Testing

- Examine object-oriented programming
- Fault models for OO-testing
- Consider advanced OO test techniques

# EXAMINE OO PROGRAMMING and IMPACT ON TESTING

# Object-Oriented Programming

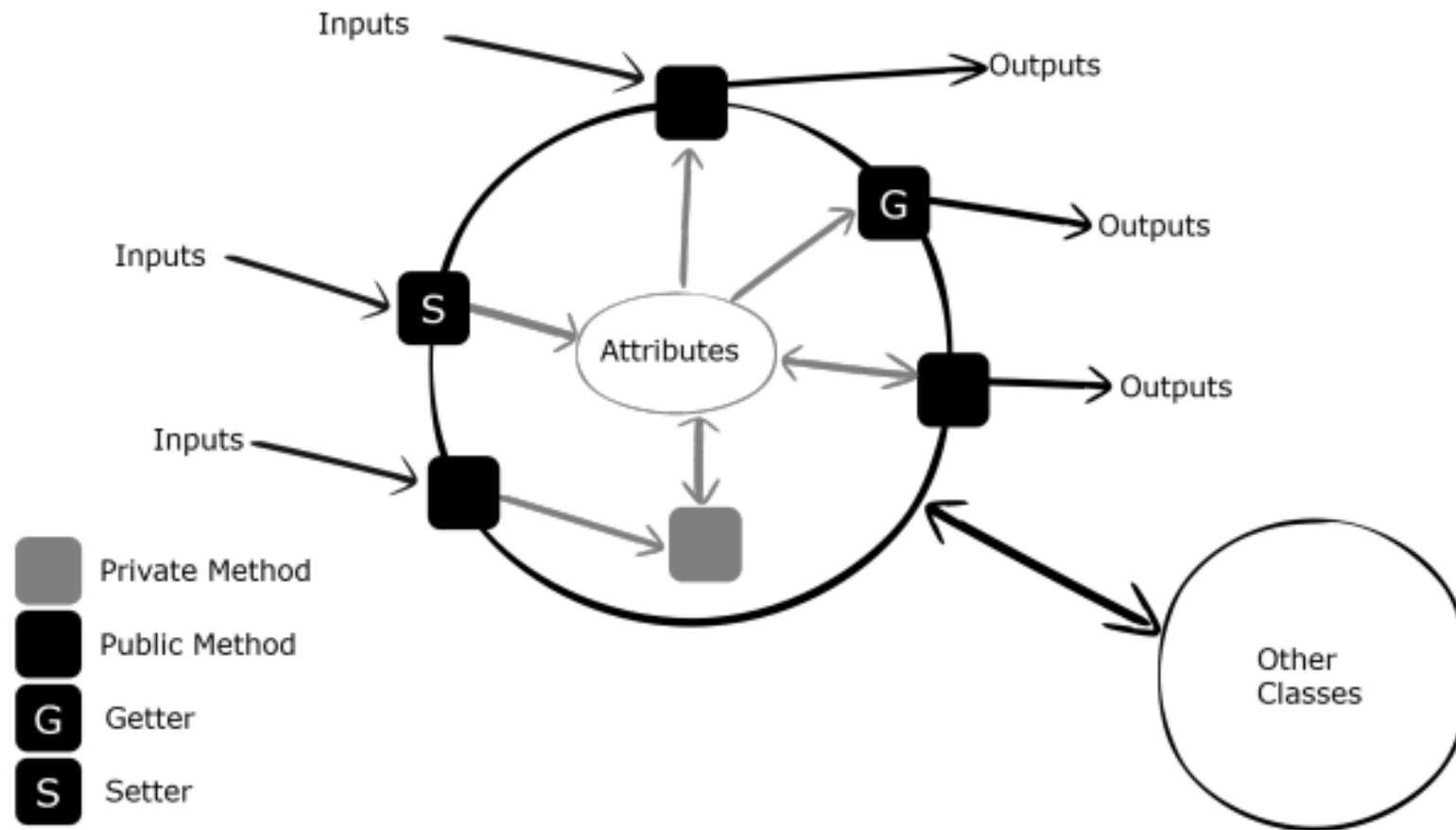
From a testing viewpoint, object-orientation can be defined as programming with classes, inheritance, and messages

- ***Classes*** provide a wrapper to encapsulate data (attributes) and associated code (methods)
- ***Inheritance*** provides a mechanism for code reuse, especially useful for implementing common features
- ***Messages*** (or method calls) provide the interface to an object

# Classes, Attributes and Messages

- **Classes** are the foundation of object-oriented programming, and contain a collection of attributes and methods
- The **attributes** are usually hidden from external access, and can only be accessed via methods: this is referred to as data encapsulation
- Classes have relationships with other classes: they can inherit attributes and methods from other classes, and can **invoke methods** on objects of other classes

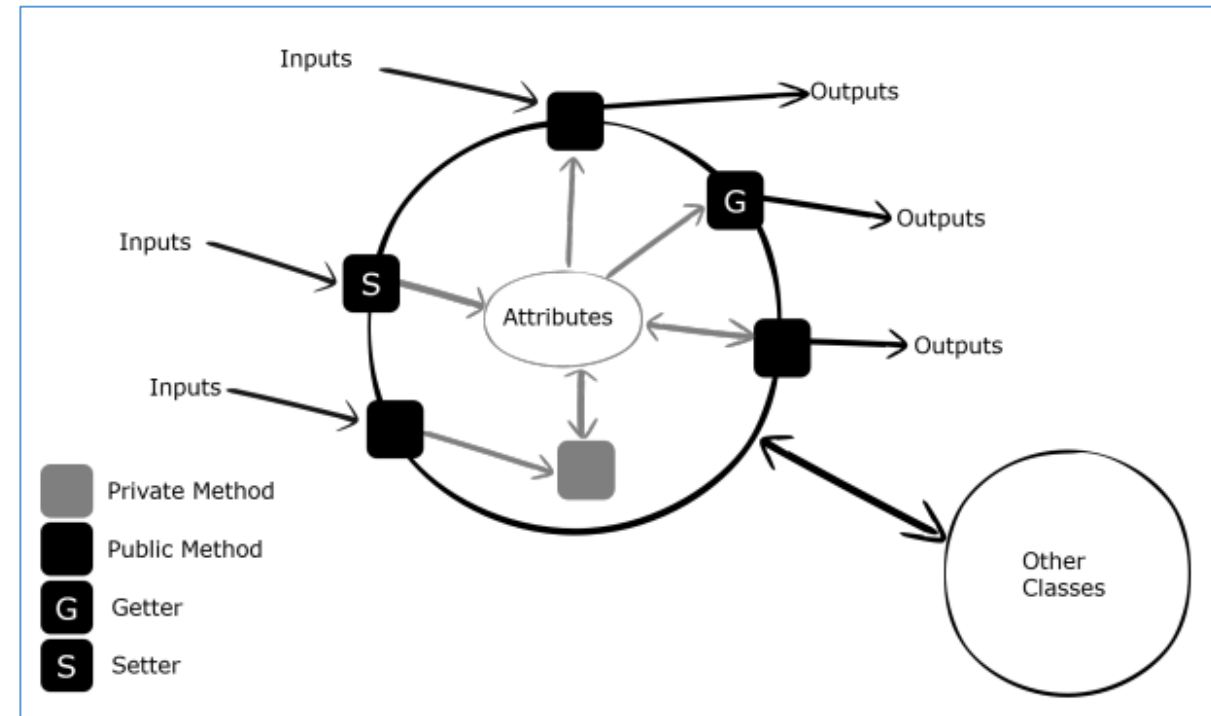
# OO Execution Model (see following slides)





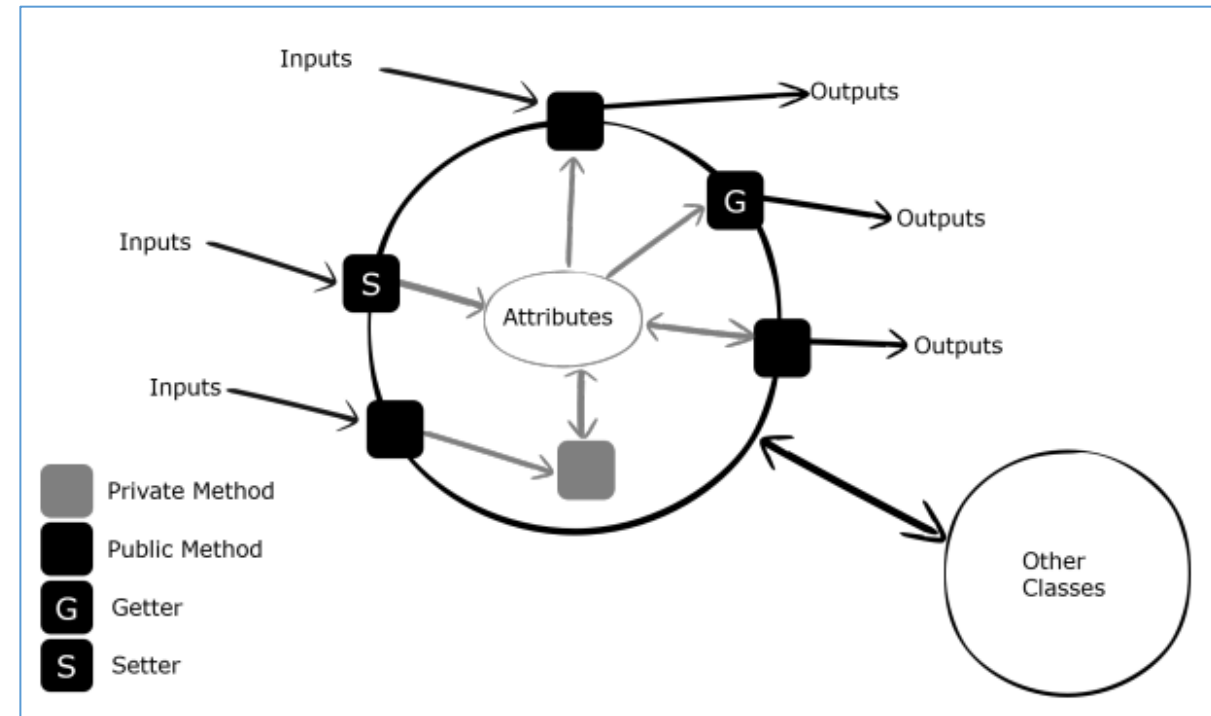
# OO Execution Model

- **Inputs** are the explicit inputs or parameters passed in the method call
- **Outputs** are the explicit outputs or values returned by the method call



# OO Execution Model

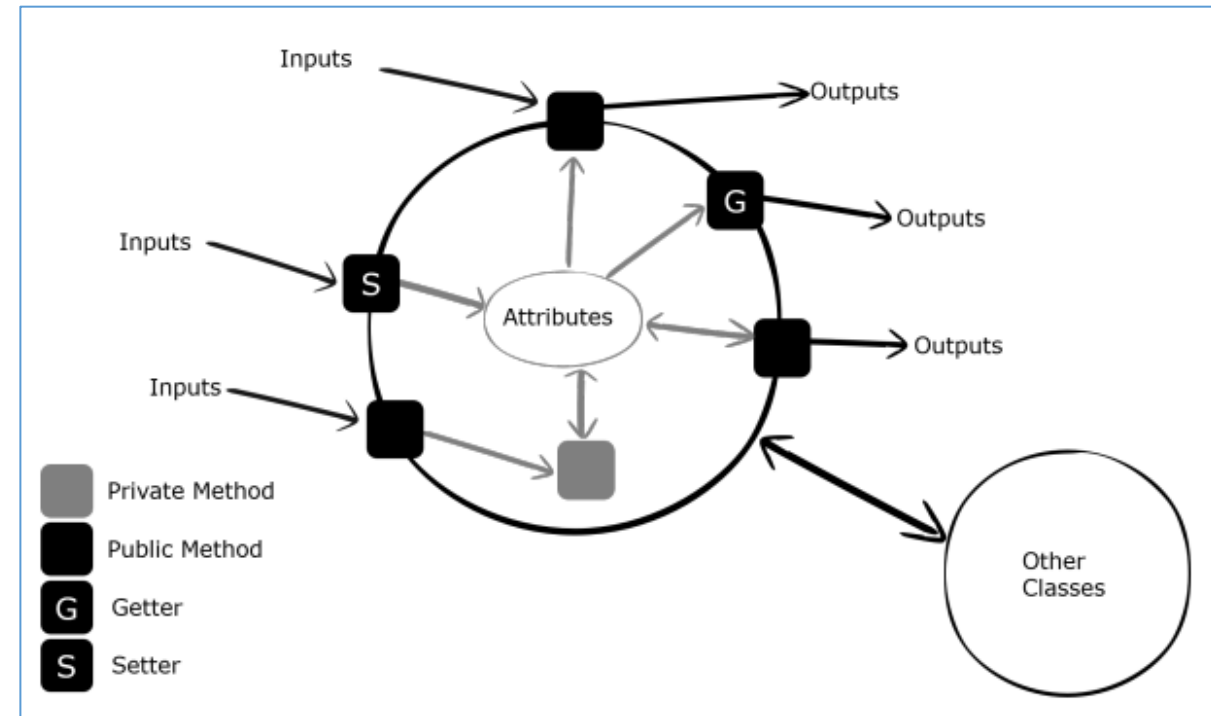
- **Public methods** are placed on the class boundary: they are accessible from outside the class
- **Private methods** and **private attributes** are hidden inside the class – they are not accessible from outside the class, and cannot be accessed or used by the tester



# Getters and Setters

**S** *Setters*: methods which have a single function, to set an attribute value

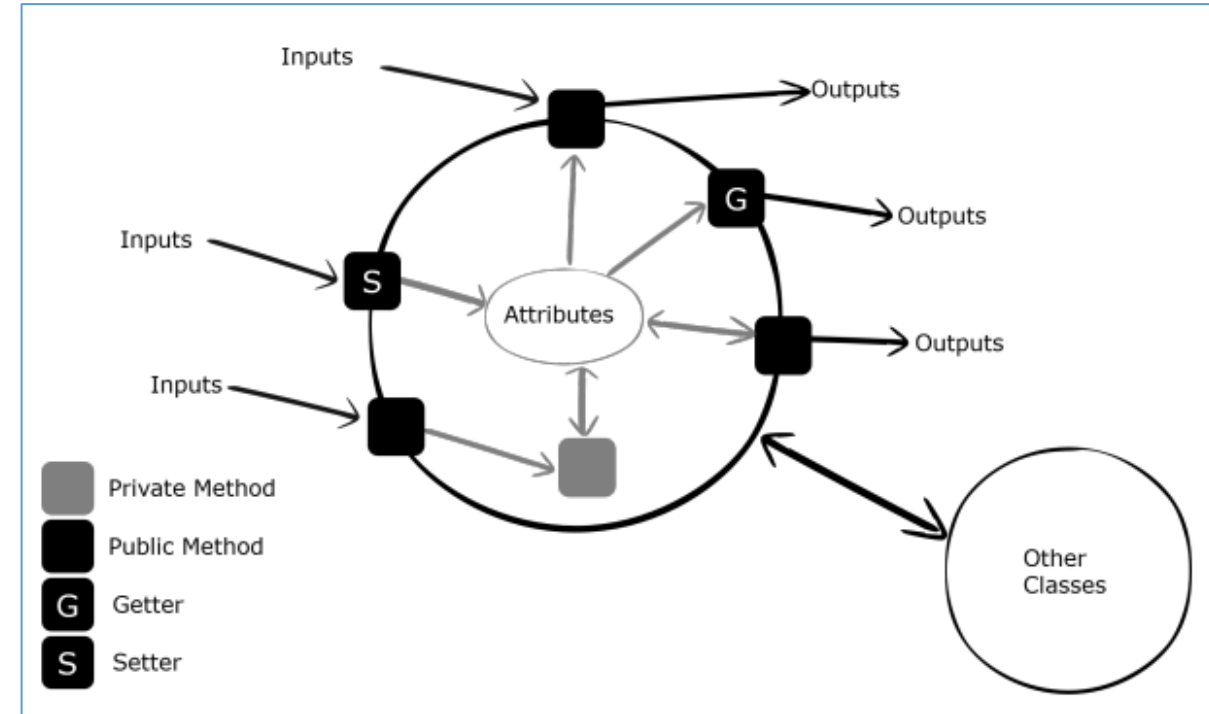
**G** *Getters*: methods which have a single function, to get an attribute value



# Other Methods

For test purposes, methods which do *any* processing are not regarded as getters or setters, and are unlabelled G or S in the diagram

These may be passed input parameters, may read attributes as inputs, do processing, call other methods both public and private, write attributes as outputs, return a value as an output, or raise an exception as an output



■ Public methods may be called by the tester

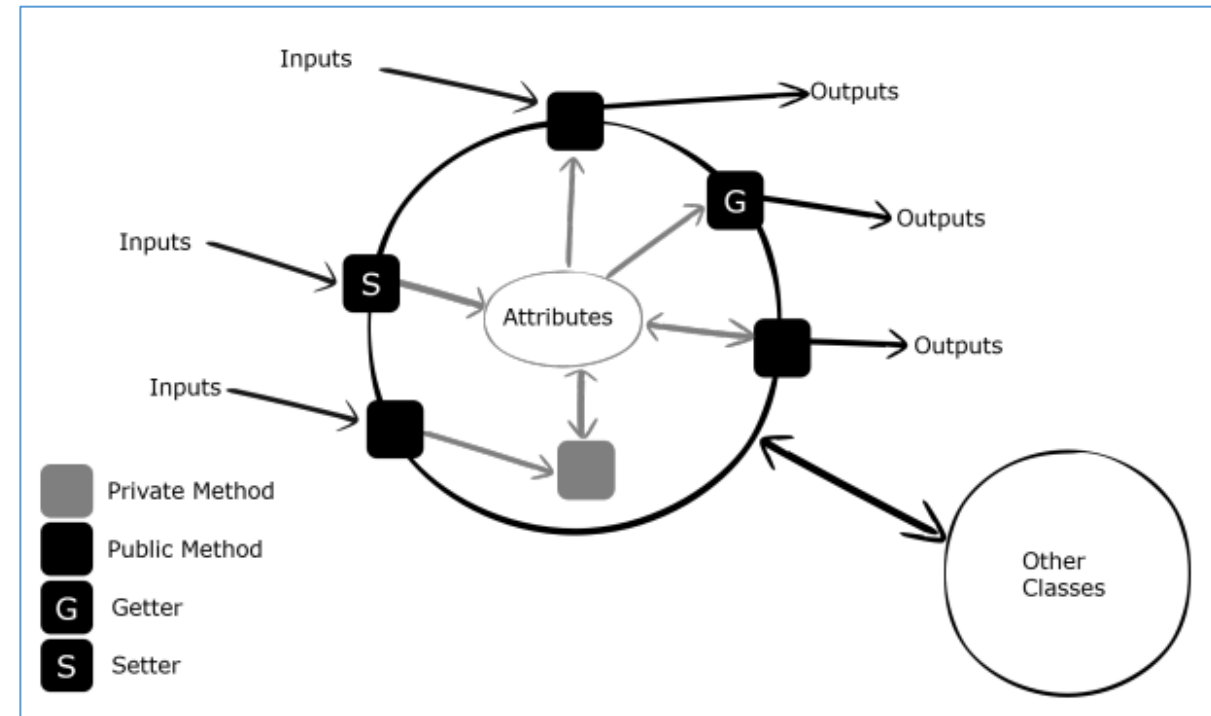
■ Private methods may only be called from within the class

# Class Relationships



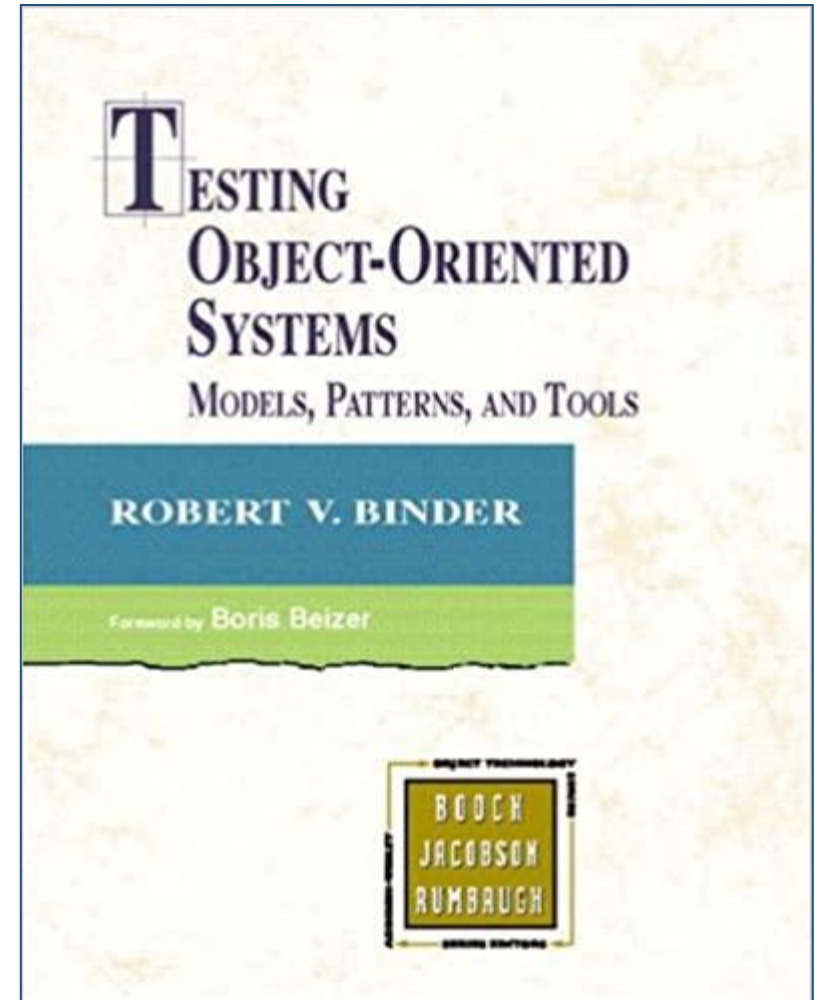
Classes may have relationships with other classes

- **inheritance** relationships, where one class reuses all the code of another class (*is-a* relationship, or generalisation/specialisation)
- **aggregation** relationships, where one class contains a collection of other classes (*has-a* relationship)



# Testing OO Software

- Very substantial topic – key highlights – for example Binder, Testing Object-Oriented Systems has over 1000 pages
- Last week: testing in class context
  - Test code in a class...
  - Where the data is protected from outside access...
  - And methods interact with each other through attributes
- This week: some more advanced topics



# FAULT MODELS FOR OO TESTING

# Fault Models

- The features of object-orientation are intended to improve code quality and support code reuse
  - Classes
  - Inheritance
  - Messages
- Complex concepts => introduce their own fault models
- In addition to those associated with software in general



# Fault Model for **Classes**

- Objects retain the values of their attributes (referred to more formally as their state) between method calls
- In many cases, the order in which these methods are called is important for correct operation
- Generally, the code which ensures this ordering is distributed over many methods
- This provides many opportunities for what are referred to as state control faults, where an object does not behave correctly in a particular state

# Fault Model for Inheritance

- Inheritance leads to a *binding* between objects and different classes and interfaces
- Dynamic binding occurs at runtime, where a decision is made by the Java VM as to which overridden method to use
- Complex inheritance structures, or a poor understanding of inheritance in Java, may provide many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage

# Fault Model for Messages

- Programming mistakes when making calls are a leading cause of faults in procedural languages:
  - the wrong parameters are passed
  - or the correct parameters are passed but mixed up
- Object-oriented programs typically have many short methods, and therefore provide an increased risk of these interface faults

# Testing in Class Context

- Methods in a class must be tested in the context of the other methods defined in the class
- Essential element of all tests for object-oriented software, as all methods execute in class context
- That is why, up till now, we have examined conventional models of testing using static methods, which accept all their inputs as arguments to the method and return the result

# A Conventional Test

---

```
1  actual_result = Classname.method( p1, p2, p3 );  
2  check_equals( actual_result, expected_results );
```

---

- Line 1
  - the method being tested is called with the test input data values (p1,p2,p3)
  - the return value is stored in a temporary variable: actual\_result
- Line 2
  - The actual result is compared with the expected result
  - Same: test passes
  - Different: test fails

# A Conventional Test on One Line

---

```
1    check_equals( Classname.method( p1, p2, p3 ),  
                  expected_results );
```

---

# A Conventional Test on One Line

**METHOD BEING TESTED  
RETURNS ACTUAL RESULT**

**INPUT  
PARAMETERS**

---

```
1  check_equals( Classname.method( p1, p2, p3 ),  
               expected_results );
```

---

# A Conventional Test on One Line

**METHOD BEING TESTED  
RETURNS ACTUAL RESULT**

**INPUT  
PARAMETERS**

---

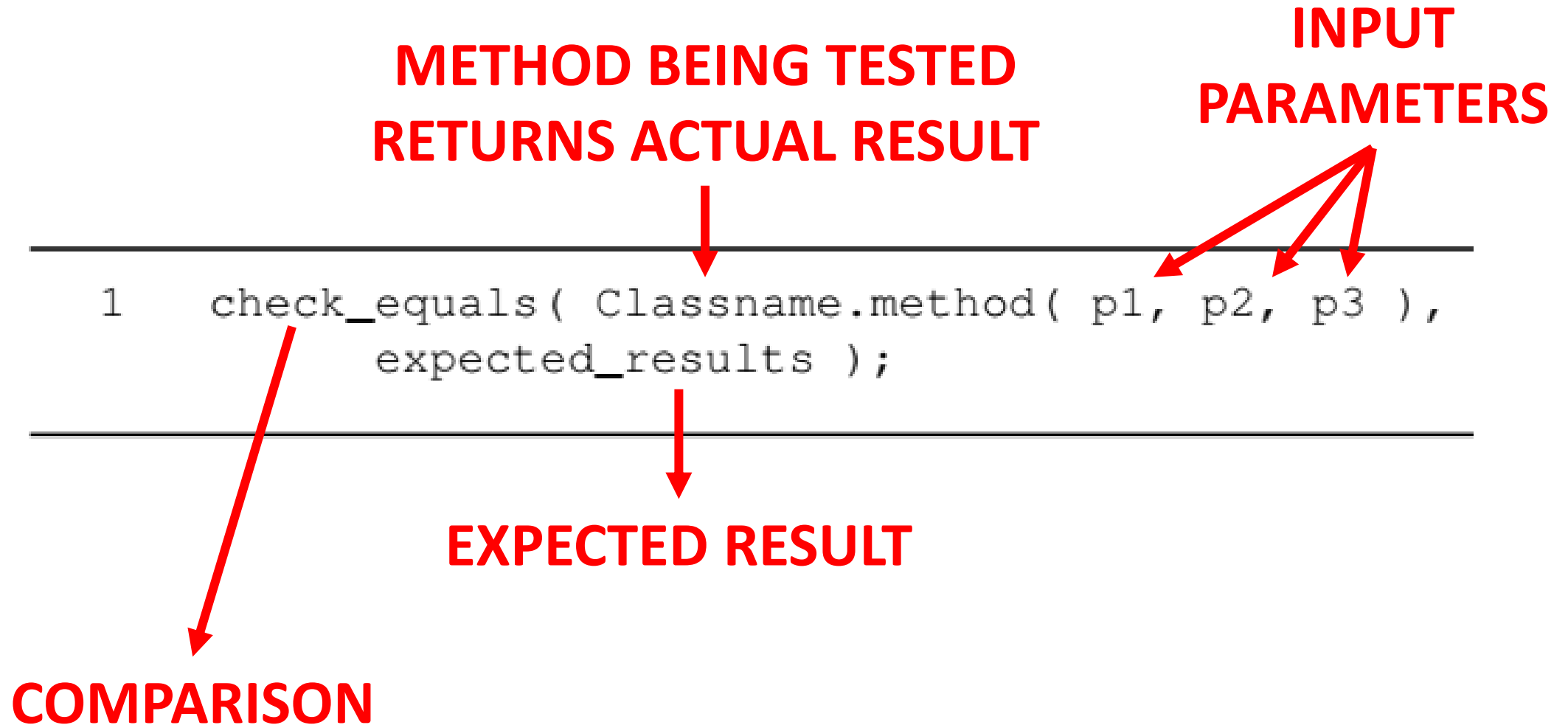
```
1  check_equals( Classname.method( p1, p2, p3 ),  
               expected_results );
```

---

**EXPECTED RESULT**



# A Conventional Test on One Line



```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

## Structure of a typical OO test

---

```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
4  check_equals( actual_status, expected_status );  
5  actual_result = object.get_result();  
6  check_equals( actual_result, expected_results );
```


---

Examined in more detail in the following slides

```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

---



```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
4  check_equals( actual_status, expected_status );  
5  actual_result = object.get_result();  
6  check_equals( actual_result, expected_results );
```


---

- Line 1
  - an object is created
  - the input p1 is passed to the constructor
  - this value is stored in an attribute (e.g. paramP1)

```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

---



```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
4  check_equals( actual_status, expected_status );  
5  actual_result = object.get_result();  
6  check_equals( actual_result, expected_results );
```


---

- Line 2
  - the value p2 is set
  - this value is stored in an attribute (e.g. paramP2)

```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

---



```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
4  check_equals( actual_status, expected_status );  
5  actual_result = object.get_result();  
6  check_equals( actual_result, expected_results );
```

---

- Line 3
  - the method being tested is called with the value p3
  - the method also uses paramP1 and paramP2 as input values
  - the method returns a boolean to indicate if it worked
  - the result of the method is stored in an attribute (e.g. theResult)

```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

---

```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
➔ 4  check_equals( actual_status, expected_status );  
5  actual_result = object.get_result();  
6  check_equals( actual_result, expected_results );
```


---

- Line 4
  - the actual status is compared with the expected status
  - if they are the same, then the test continues
  - otherwise the test fails.

```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

---

```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
4  check_equals( actual_status, expected_status );  
 5  actual_result = object.get_result();  
6  check_equals( actual_result, expected_results );
```


---

- Line 5
  - the attribute theResult is retrieved using a getter method
  - this fetches the actual results

```
check_equals( Classname.method( p1, p2, p3 ),  
              expected_results );
```

# Testing in Class Context

---

```
1  object = new Classname( p1 );  
2  object.setParameter2( p2 );  
3  actual_status = object.method_under_test( p3 );  
4  check_equals( actual_status, expected_status );  
5  actual_result = object.get_result();  
 6  check_equals( actual_result, expected_results );
```

---

- Line 6
  - the actual results are compared with the expected results
  - if they are the same, then the test passes
  - otherwise the test fails



# Additional Mechanisms

- This demonstrates the various additional mechanisms that object-oriented test code uses for inputs and outputs in class-context
- The BBT and WBT techniques can all be applied to testing methods in class context: EP, BVA, DT, SC, BC
- Key differences are where you need to take the class context issues into account:
  - Analysis
  - Test cases
  - Test implementation

# Analysis for OO Testing

- There are three parts to the analysis
  1. Decide which methods will be tested
  2. Analyse the class – especially use of attributes
  3. As for conventional unit testing, analysing the test specifications (and possibly the code) by applying the test design technique(s) selected in order to identify the test coverage items

# Test Coverage Items

- The test coverage items are identified, for the test design technique(s) selected, using the results of the analysis

# Test Cases

- Not only the **data values** to be used, but the **sequence of method calls**, must be decided on
- This can be left until the implementation phase, but it is important at implementation time to know which method is to be called (as different methods may have the same parameter name for different purposes), and the sequence may have an impact on the data values
- So better to define during test design

# Methods to Develop Test Cases For

- Static methods
  - these are not dependent on the constructor
  - usually they are independent, but if they interact through **static attributes**, then they should be considered in class context along with the other methods
- Constructor(s)
  - include the default constructor
  - note that the constructor is called after the other object initialisation has taken place
- Simple methods that do not use the class attributes.
- Accessor methods (getters and setters) – **if they are to be tested**
- The other methods, which do use the class attributes

# Focus of Test Cases

- It is recommended to limit each test case to one primary method call being tested
- This may require a number of supporting method calls as we have seen
- Makes reviewing the test cases for correct ordering and completeness much easier
- It is not always necessary to use separate tests for each input error case (unlike in equivalence partition testing), as long as the input error values are inputs to different methods and do not cause error hiding

# Test Implementation

- There are two categories of OO test methods:
  1. those that can run in any order
  2. Those that require other methods to be run first
- In the example shown, the test methods create a new instance of the class for each test, and thus can run in any order
  - test priority or test dependencies can be used to enforce a particular ordering
- If one test case relies on another one to be run first, then they must either be both included in the same test in the required order, or test dependencies must be used
- Once the test cases have been defined, test implementation is usually very straightforward