Object-Oriented Testing meets TTCN-3

TTCN-3 User Conference 2006
The Evolution of TTCN-3
31 May- 2 June 2006, Berlin, Germany

Prof. Dr. Mario Winter
FH Köln
Overview

- Why is testing object-oriented software difficult?
- Specification and testing of object-oriented software
- The UML Testing Profile
- TTCN-3 cases for OO

Why do we test?

- Errare humanum est! (to err is human)
- Even with formal specification and program generation ...
  - Spec could be wrong
  - Generator doesn’t work as expected
  - Platform doesn’t work as expected
  - ...
- Testing checks that the Software ...
  - ... does what it should do and ...
  - ... doesn’t what it shouldn’t do
- But always remember that ...
  - ... Testing can only show the presence of faults (i.e. bugs), not their absence!!!
  [E.W. Dijkstra]
When do we Test?

- Requirements
- Analysis
- Architecture
- Design
- Coding
- Acceptance Test
- System Test
- Integration Test
- Unit Test

Test cases based on work artifacts (documents)

Test process

Planning and control
- Analysis and design
- Implementation and execution
- Evaluation of test exit criteria
- Post testing activities

Start

Validation & Verification

Test Level

End

How do we Test?

- Static Testing
  - No executables necessary
  - Automated source code analysis
  - Manual reviews and inspections

- Dynamic Testing
  - Executables necessary
  - Test cases with input and expected behaviour/output
  - Test harness for unit and integration testing needed

Test Object

Processor

Stimulation

Observation

[ISTQB Certified Tester – Foundation Level Syllabus 2005]
Black box testing vs. white box testing

**Black Box Testing**
- Input parameter/state
- Specification needed!
- Test object
- Output parameter/state

**White Box Testing**
- Input parameter/state
- Specification and realization (sources) needed!
- Test object
- Output parameter/state

---

Why is dynamic testing difficult?

**Test driver**
- Test case 1
- Test case 2
- ...
- Test case m

**Test object**
- Point of Control
- Point of Observation

**Slubs, Mocks**
- Stub 1
- Stub 2
- ...
- Stub n

**Run time environment, analysis tools, simulators, monitors**

**Test output**
- Compare & report

**Processor**
Why is object-oriented testing even more difficult?

- Object-Orientation = "Normal programming" + Encapsulation + Inheritance + Polymorphism + Late Binding

- Encapsulation
  - Bundling of data with the operations that manipulate that data
  - Classes often hide encapsulated information by presenting only public interfaces

- Inheritance
  - Forms new classes using already defined classes
  - Generalization: is-a relationship, captures hierarchical relationship between classes

- Polymorphism
  - Ability of object (references) belonging to different classes to respond to calls of operations of the same name, each one according to a class-specific behavior
  - Programmer (and program) does not have to know the exact classes of objects in advance, so behavior can be implemented in different ways

- Late binding (aka. dynamic binding)
  - At run time, the machine ("system") decides which method to execute
  - Methods searched upwards the inheritance hierarchy

Example: Inheritance or Generalization?

- Rectangle
  - side1
  - side2
  + setSide1()
  + setSide2()
  + getArea()

- Square
  - side1
  - side2
  + setSide1()
  + setSide2()
  + getArea()

{ inv: side1 = side2 }

SomeClass

Superclass

Subclass

myR : Rectangle

foo()

setSide1(10)

A := getArea()

A = 200

OK!

myR : Square

foo()

setSide1(10)

A := getArea()

A = 100?

A = 200?

OOPS!
### Hierarchical Inheritance Testing (HIT)

<table>
<thead>
<tr>
<th>Operations of subclass</th>
<th>Black box test cases</th>
<th>White-box test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited operations</td>
<td>Execute test cases of superclass</td>
<td>Repeat test cases of superclass</td>
</tr>
<tr>
<td>Redefined operations</td>
<td>Complement test cases of superclass and complemented test cases</td>
<td>Create new test cases</td>
</tr>
<tr>
<td>(Pure generalization)</td>
<td>Execute test cases of superclass</td>
<td>Execute new test cases</td>
</tr>
<tr>
<td>New defined operations</td>
<td>Create new test cases of subclass</td>
<td>Create new test cases</td>
</tr>
<tr>
<td></td>
<td>Execute new test cases</td>
<td>Execute new test cases</td>
</tr>
</tbody>
</table>

### Where are we now?

- Why is testing object-oriented software difficult?
- Specification and testing of object-oriented software
- The UML Testing Profile
- TTCN-3 cases for OO
Black box testing of classes

- **Goal**
  - Validation of the class interface, i.e. its public operations (and members)
- **Value**
  1. Conformance of the class realization w.r.t. its specification
  2. Robustness of the class
- **Needs**
  - Assertions, i.e. pre- and postconditions for each public operation of the class under test (CUT) together with its invariant
- **Result**
  - Reusable and extendable class test cases which validate the classes interface conformance and its robustness

**Black Box Testing**

![Input parameters/State](#) Specification needed! **Class Under Test (CUT)** ![Output parameters/State](#)

---

How to specify interfaces? Design by contract (B. Meyer)

- **Client uses suppliers (public) operations**
  - Precondition = clients price
  - Postcondition = suppliers obligation
  - Invariant = "eternal" assumptions
- **Contract**
  - If the client satisfies the precondition, then the supplier guarantees the postcondition!
  - Client and supplier satisfy their invariants before and after each (public) operation execution
- **A contract is a prescription, not only a description!**

---

**Invariant**

- **Contract**
- **Invariant**
  - If you promise to call me with the **precondition** satisfied then I, in return, promise to deliver a final state in which the **postcondition** (and my **invariant**) is satisfied
Specification of interfaces with assertions

- **Precondition (of an operation)**
  - States the properties that must hold whenever the operation is called.
  - Refers to input-parameters of the operation and the state of the component

- **Postcondition (of an operation)**
  - States the properties that the operation guarantees when it returns (assuming its precondition was satisfied).
  - Refers to output-parameters of the operation and the state of the component

- **Both preconditions and postconditions describe properties of individual operations**
  - but often there are more general properties

- **Invariant (of the class)**
  - Expresses global properties of all instances of a class, which must be preserved by all operations.
  - Refers to the state/instant variables of the class.
  - Must be satisfied before and after each execution of each (public) operation.

Why is testing OO-SW difficult?

Specification and testing of OO-SW

The UML Testing Profile

TTCN-3 cases for OO

---

**Example: BoundedStack (Stack with bounded capacity)**

```java
class BoundedStack

State preserving operations
size():integer; // Number of elements
MAXSIZE(): integer; // Maximum count of elements
top():Object; // Pointer to topmost element

State changing operations
BoundedStack(maxSize: integer);// Constructor
~BoundedStack(); // Destructor
push(element: Object); // Stack element on top
pop(); // Removes topmost element

push(element: Object) 2 ← size
pop() 1
```

---

Why is testing OO-SW difficult?

Specification and testing of OO-SW

The UML Testing Profile

TTCN-3 cases for OO
### Contract of BoundedStack

```java
class BoundedStack {
    /** invariant@ self.size() >= 0 AND self.size() <= self.MAXSIZE() */

    public BoundedStack (Integer maxSize) {
        /** pre@ maxSize >= 0 */
        /** post@ self.MAXSIZE() = maxSize */
    }

    public void push (Object item) throws FullStackException {
        /** pre@ self.size() < self.MAXSIZE() */
        /** post@ self.size() = self.size()@pre + 1 AND self.top() = item@pre */
    }

    public Object top () throws EmptyStackException {
        /** pre@ self.size() > 0 */
        /** post@ return != null */
    }

    public Object pop () throws EmptyStackException {
        /** pre@ self.size() > 0 */
        /** post@ self.size() = self.size()@pre - 1 */
    }

    public Collection all () {
        /** pre@ true */
        /** post@ return.size() = this.size() AND (self.size() > 0 implies return.size() = self.size()) AND (self.size() = 0 implies return = null) */
    }
}
```

### Contract-based test cases for BoundedStack

**Context BoundedStack**

<table>
<thead>
<tr>
<th>Invariant@ self.size() &gt;= 0 AND self.size() &lt;= self.MAXSIZE()</th>
<th>True</th>
<th>True</th>
<th>True</th>
<th>True</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedStack@ maxSize &gt; 0</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>BoundedStack@ (post@ self.size() = 0 AND self.MAXSIZE() = maxSize)</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>push@ (pre@ self.size() &lt; self.MAXSIZE() AND self.size() = 0)</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>push@ (post@ self.size() = self.size()@pre + 1)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>top@ (pre@ self.size() &gt; 0)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>top@ (post@ return = null)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>pop@ (pre@ self.size() &gt; 0)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>pop@ (post@ self.size() = self.size()@pre - 1)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>all@ (pre@ true)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>all@ (post@ self.size() &gt; 0 AND (return.size() = self.size()) OR not self.size() &gt; 0 AND return = null)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

- **Conformance testing:** precondition satisfied
- **Robustness testing:** precondition not satisfied
- **Test oracle:** postcondition satisfied

**Don't Care**

---

**OO-Testing meets TTCN-3**
State-based specification for BoundedStack

Three (five) states:
- **empty**: \( \text{size}() = 0 \);
- **filled**: \( 0 < \text{size}() < \text{MAXSIZE}() \);
- **full**: \( \text{size}() = \text{MAXSIZE}() \);
- **initial**: before construction;
- **final**: after destruction;

---

Life cycle-based test cases (aka. state based)

Three (five) states:
- **empty**: \( \text{size}() = 0 \);
- **filled**: \( 0 < \text{size}() < \text{MAXSIZE}() \);
- **full**: \( \text{size}() = \text{MAXSIZE}() \);
- **initial**: before construction;
- **final**: after destruction;
Testing operation calls and late binding

Operating object (as “client”), parameter object und server object classes: How many different combinations of classes and states?

\[(3 \times 2 \times 3) \times (3 \times 3 \times 3) \times (3 \times 3 \times 3) = 486 \text{ possibilities!}\]

Idea: Not all combinations, but all pairwise combinations

- Orthogonal arrays (OA)
- Two states \(SA_1\) and \(SA_2\)

### OA\(_{18}(2^1 \times 3^7)\)

- \(A = 1, A_1 = 2, A_2 = 3, ZA_1 = 1, ZA_2 = 2\)
- \(B = 1, B_1 = 2, B_2 = 3, ZB_1 = 1, ZB_2 = 2, ZB_3 = 3\)
Create and execute test cases

- Testscripts
  - TTCN-3
  - Parameterized
  - Executable

```
module Test of Class A {
  modulepar {
    B_Type theB_Par;
    B_Type anotherB_Par;
  }
  external const float TestExecutionTime;
  control {
    var verdicttype testCaseVerdict := none;
    testCaseVerdict := execute (connectmyB(theB), TestExecutionTime);
    if (testCaseVerdict == pass) {
      testCaseVerdict := execute (m1(anotherB), TestExecutionTime);
    }
  }
}
```

<table>
<thead>
<tr>
<th>Nr</th>
<th>Client</th>
<th>State</th>
<th>Parameter Class</th>
<th>Parameter State</th>
<th>Server Class</th>
<th>Server State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

A = 1, A1 = 2, A2 = 3, ZA1 = 1, ZA2 = 2
B = 1, B1 = 2, B2 = 3, ZB1 = 1, ZB2 = 2, ZB3 = 3

Why is testing OO-SW difficult?

Specification and testing of OO-SW

The UML Testing Profile

TTCN-3 cases for OOS
Where are we now?

- Why is testing object-oriented software difficult?
- Specification and testing of object-oriented software
- The UML Testing Profile
- TTCN-3 cases for OO

Example: Automated Teller Machine (ATM)

Test System

- Main Test Component (MTC)
  - Bank User
- User Interface
- Network Interface
- Test Coordination
- System Under Test (SUT)
  - ATM
- Parallel Test Component (PTC)
  - Bank Server

ATM system test scenario

Why is testing OO-SW difficult?
Specification and testing of OO-SW
The UML Testing Profile
TTCN-3 cases for OO

UTP: ATM system test class diagram

Why is testing OO-SW difficult?
Specification and testing of OO-SW
The UML Testing Profile
TTCN-3 cases for OO
Why is testing OO-SW difficult?

Specification and testing of OO-SW

The UML Testing Profile

TTCN-3 cases for OO

Where are we now?

• Why is testing object-oriented software difficult?
• Specification and testing of object-oriented software
• The UML Testing Profile
• TTCN-3 cases for OO
Why is testing OO-SW difficult?

- Specification and testing of OO-SW

- The UML Testing Profile

- TTCN-3 cases for OO

---

**Eclipse Test & Performance Tools Platform (TPTP)**

- Object oriented testing with TTCN-3
- Open source top level project of the Eclipse foundation
- Includes the HYADES TTCN-3 tool set
  - TTCN-3 Core Language Editor
  - TTCN-3 Compiler
- TPTP is divided into four projects.
  - TPTP Platform
  - Monitoring Tools
  - Testing Tools
  - Tracing and Profiling Tools
- But
  - TTCN-3 integration with least common denominator principle
  - Many powerful TTCN-3 concepts were left out!

---

**TTCN-3 – The standard**

- TTCN-3 Core Language
- Table Format
- Graphical Format
- IDL
- ASN.1

---

```ml
testcase myTestcase { runs on MTCType system TGType
  mydefault := activate (OtherwiseFail);
  verdict.set(pass);
  connect(PTC_ISAP1:CP_ISAP1, mtc:CP_JSAPI);
  map(PTC_ISAPI1:ISAPI1, system:TSI_ISAPI1);
  PTC_ISAPI1.start(func_PTC_ISAPI1());
  PTC_MSAPI2.start(func_PTC_MSAPI2());
  Synchronization();
  all component done;
  log("Correct Termination");
}
```
Using C/C++ with TTCN-3

- Draft ETSI ES 201 873-10
  Methods for Testing and Specification (MTS);
  The Testing and Test Control Notation version Part 10:
  Using C/C++ with TTCN-3 (lead author: Jens Grabowski)

What did we reach with TTCN-3?

- OO-Testing = Message Sequences = TTCN-3
- Excellent notation for black-box test scripts
  - Modular
  - Reusable
- Excellent tool support for TTCN-3 specification and execution
  - Commercial and open source
  - Different domains (communication, automotive, web, ...) well covered
What is still missing regarding OO-Testing?

- Subtyping for TTCN-3 structures
  - Data structures
  - Modules
  - Test Cases
  - ...
- Full UML testing profile binding
  - Subclassing (i.e. generalization)
  - Tool Support w.r.t. full UML support
- Java Binding ("Using Java with TTCN-3")
- More wishes?
Summary

- Why is testing object-oriented software difficult?
  - Inheritance
  - Polymorphism, late binding
- Specification and testing of object-oriented software
  - Unified Modeling Language
  - Testing techniques
- The UML Testing Profile
  - Black box testing notation
  - ATM case study
- TTCN-3 cases for OO
  - Eclipse TPTP
  - Using C/C++ with TTCN-3