# SERG Test Strategy

Arie van Deursen

Version 0.2, 31 January 2010

## 1 Introduction

This document describes the test strategy adopted in software development projects conducted under the responsibility of the Software Engineering Research Group (SERG) of Delft University of Technology.

The document provides guidelines on how to conduct testing. It is inspired by

1. The concept of a strategy document as discussed by [Pezzè and Young, 2008, Ch. 20]
2. The notion of a *test catalog* as proposed by [Marick, 1995] and discussed by [Pezzè and Young, 2008, Section 11.4].
3. The collection of *test design patterns* provided by [Binder, 2000].

The present document is in use at the course *Software Testing and Quality Engineering*, and plays a role in research projects conducted at SERG.

This is a working document which is continuously in progress. During each project, the test strategy is evaluated, and if necessary adjusted. In particular, whenever a fault slips through testing, a critical *root cause analysis* is conducted [Pezzè and Young, 2008, Section 20.7], and if possible the testing strategy is adjusted so that in future faults like these are more likely to be captured.

## 2 Test Process

The basic test process involves the following steps:

- Requirements are explicitly documented.
- A requirements specification immediately includes a (number of) test case specifications.
- Design decisions are explicitly documented.
- Requirements as well design documents are used to derive test case specifications before coding starts.

- The catalog of test patterns (Section 3) is used to derive test cases where appropriate.
- Test code is written as much as possible *before* the corresponding production code (in line with [Beck, 2003]
- After implementing a feature, a coverage tool is used to analyze whether all code is indeed covered.

**Assertions and Design by Contract** Where possible, design decisions and coding assumptions are documented by means of assertions. In particular:

- Class invariants are encoded in Boolean function `invariant`. Assertion checking the invariant are included at the start and at the end of every public method.
- Method preconditions are encoded as assertions at the beginning of each method.
- Method post conditions are encoded as assertions at the end of each method, or just before the return statement.
- All assertions are side-effect free.
- Exceptions are thrown in line with the principles of Design-by-Contract [Meyer, 1992].

## 3 A Catalog of Test Patterns

### 3.1 The Catalog Format

**Pattern 0: Template** For each entry in the catalog, we present the following information:

- *Model Under Analysis*: The artefact / document that we inspect in order to arrive at a test case;
- *Fault Model*: The sort of faults that we expect to find.
- *Test Strategy*: The testing steps that must be taken to have a good chance to find these fault. The steps are derived from the model under analysis.
- *Adequacy*: A way of indicating to what extent we are done when adopting this strategy. This is typically expressed as a percentage of elements in the model under analysis that is to be covered. (see [Pezzè and Young, 2008, Chapter 9].

## 3.2 Boundary Values

**Pattern 1: Boundary values**

- *Model Under Analysis*: Design or requirements documents
- *Fault Model*: Boundaries often give rise to “off-by-one” errors
- *Test Strategy*: For each boundary test one point that is exeactly on the boundary, one that is just off the boundary. (see also the $1 \times 1$ strategy from [Binder, 2000], and the sample test catalog in [Pezzè and Young, 2008, Table 11.7] inspired by [Marick, 1995]).
- *Adequacy*: A boundary is adequately checked if it is execued with one value on the boundary as well as one just off the boundary. Coverage is given by the number of adequately tested boundaries divided by the total number of boundaries occurring in the specs / designs.

## 3.3 Decision Logic

**Pattern 2: Basic Decision Logic** Requirements often involve simple if-then-else decision logic in one way or another, which is tested through this pattern.

- *Model Under Analysis*: Design or requirements document
- *Fault Model*: Either the coding of the condition could be wrong, or the selected action taken in the “then” or the “else” branch could be the wrong one.
- *Test Strategy*: For each decision involving a “then” and an “else” part, two test cases are needed, one capturing the then, and one capturing the else part.
- *Adequacy*: Branch coverage, which also ensures that implicit (omitted) else branches are covered as well (see [Pezzè and Young, 2008, Section 12.3]).

**Pattern 3: Complex Decision Logic** More complex decision logic (compared to Pattern 2) may involve conditions composed from a range of predicates and Boolean operators.

- *Model Under Analysis*: The complex decision logic should be modeled through a decision table.
- *Fault Model*: Incorrect Boolean operators, incorrect bracketing, negation forgotten, cases forgotten, ...
- *Test Strategy*: Ensure each basic condition contained in the complex decision once yields true and once yields false
- *Adequacy*: Modified Condition / Decision Coverage (MC/DC) (see [Pezzè and Young, 2008, Section 12.4, 14.3]; called each-condition / all conditions by [Binder, 2000]).

**Pattern 4: Exception Handling**

- *Model Under Analysis*: Design documents including explicit specifications of exceptions to be raised under specific circumstances.
- *Fault Model*: Incorrect raising and handling of exceptions.
- *Test Strategy*: Throwing exceptions will usually be done in if-then-else structures, and hence testing is covered by Pattern 2. Therefore, testing exception handling should focus on ensuring that only specified exceptions can be propagated, and that other exceptions are caught and handled appropriately. See also [Pezzè and Young, 2008, Section 15.12]. A challenge in testing exception handling is usually controllability: being able to trigger the event that causes the exception.
- *Adequacy*: The “try” as well as the “catch” block of each try-catch statement should be covered.

## 3.4 File I/O

*Writing the File I/O patterns is left as an exercise.*

## 3.5 Further Test Patterns

Other test patterns that are used include:

- Category/Partition
- Pairwise Combination Testing
- Iteration
- Polymorphism
- State Machines
- Decision Tables

Describing these is work in progress.

## References

[Beck, 2003] Beck, K. (2003). *Test-Driven Development by Example*. Addison-Wesley.

[Binder, 2000] Binder, R. (2000). *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley.

[Marick, 1995] Marick, B. (1995). *The Craft of Software Testing*. Prentice Hall, New Jersey.

[Meyer, 1992] Meyer, B. (1992). Applying “design by contract”. *Computer*, 25(10):40–51.

[Pezzè and Young, 2008] Pezzè, M. and Young, M. (2008). *Software Testing and Analysis*. Wiley.