

BERUFSAKADEMIE MANNHEIM  
Information Technology International

Lecture Notes

Programming II  
Part 7b Testing

Dipl.-Betriebswirt (BA) VOLKAN YAVUZ  
{xmachina GmbH

November 2001–February 2002  
Classes: TIM01AGR && TIT01EGR  
Term: 1

Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Phase Model to Approach Testing . . . . .	2
1.2	Terminology . . . . .	2
<b>2</b>	<b>Modeling the Software’s Environment</b>	<b>3</b>
<b>3</b>	<b>Test Scenarios</b>	<b>3</b>
<b>4</b>	<b>Structural Testing</b>	<b>4</b>
4.1	Control Flow Oriented Testing . . . . .	4
4.1.1	Statement Coverage . . . . .	5
4.1.2	Branch Coverage . . . . .	5
4.1.3	Condition Coverage . . . . .	5
4.1.4	Path Coverage and Boundary Interior Loop Testing . . . . .	5
4.2	Dataflow Oriented Testing . . . . .	9
4.2.1	Defs/Uses . . . . .	9
<b>5</b>	<b>Functional Testing</b>	<b>9</b>
5.1	Equivalence Class Partitioning . . . . .	9
5.2	Boundary Value Analysis . . . . .	9
5.3	[FIXME: Test spezieller Werte!] . . . . .	9
5.4	[FIXME: Zufallstest!] . . . . .	9
5.5	[FIXME: Test von Zustandsautomaten!] . . . . .	9
<b>6</b>	<b>Instrumentation</b>	<b>9</b>
<b>7</b>	<b>Debugging</b>	<b>9</b>
7.1	The Java Debugger: JDB . . . . .	9

<b>8 Exercises</b>	<b>10</b>
<b>References</b>	<b>10</b>

## 1 Introduction

Program testing can be used to show the presence of bugs,  
but never to show their absence  
– *Edsger Dijkstra*

In addition to quality measures taken during software construction in terms of a rigid specification, good design, object oriented programming, *testing* still is an important analytical and indispensable quality assurance mechanism. The objective during testing is to discover as many errors as possible. It gives evidence that a system meets certain quality criterions that are defined by the test cases performed. This, of course, implies that the errors discovered through the testing process have been fixed. It does *neither* prove the correctness of the program *nor* does it prove the absence of further errors. Testing can be supported by appropriate tools. Of course, it is more favorable to *prevent* errors in the first place during implementation than to *discover* them later by testing. Another problem is to determine when testing can be finished, thus answering the question “have we tested enough?”. Still another problem is the question whether the test cases really do verify the specification.

**Note:** Each program is faulty unless the opposite has been verified.

### 1.1 Phase Model to Approach Testing

Whittaker proposes an approach to testing consisting of four phases:

1. Modeling the software’s environment
2. Selecting test scenarios
3. Running and evaluating test scenarios
  - Exercise a module
    - Use predetermined inputs (*test cases*)
    - Capture actual outputs
    - Compare to expected outputs
  - Test case *succeeds* or *fails*
4. Measuring testing progress, thus answering the question “have we tested enough?”

### 1.2 Terminology

The subject of testing has its own terminology (see also [Six et al., 1998, pp. 359–402] and Denert [1992].)

**Unit Testing** Each program unit is tested in isolation.

**Integration Testing** Program units are tested during assembly.

**Regression Testing** ensures that changes made during maintenance do not destroy existing functionality or introduce new defects. A permanent suite of test cases saves efforts and provides a record of existing functionality.

**Test Case** A *test case* is a set of input data and an associated set of output. Test cases are derived with the intention to actually execute as much code as possible by selection of a wide variety of input data.

**Test Driver** The *test driver* is a setup used to perform unit testing. The unit to be tested is executed directly by the test driver instead of the normal environment when that unit will be in productive use.

**Test Item** The *test item* is the piece of code under test.

**Stub, Dummy** *Stubs* or *dummies* are used to provide functionality that hasn't been implemented yet, normally during unit tests. Functionality that the unit under test is dependent on is replaced by dummies.

### Instrumentation

**Reproducibility** Test runs have to be repeatable in order to perform regression tests. All tests that were performed have to be logged with the relevant results. Performing tests in an automated way makes it possible to run a large number of test cases without user interaction (which might be a source of errors).

## 2 Modeling the Software's Environment

During software development, testing is performed for different components, and each component consists of a specification and an implementation.

**Unit tests** testing of single code units, methods, classes, functions.

**Integration tests** testing of the interfaces among integrated units.

**System tests** testing of the complete system.

**Acceptance tests** testing of a complete system for satisfaction of requirements.

## 3 Test Scenarios

Determining what *type* of testing to do occurs in this second phase:

**Dynamical Testing** assumes that the code to be tested is actually executed.

**Structural Testing** (*white-box testing*) test cases are derived solely based on the source code or its data structures.

- control flow oriented
- data flow oriented

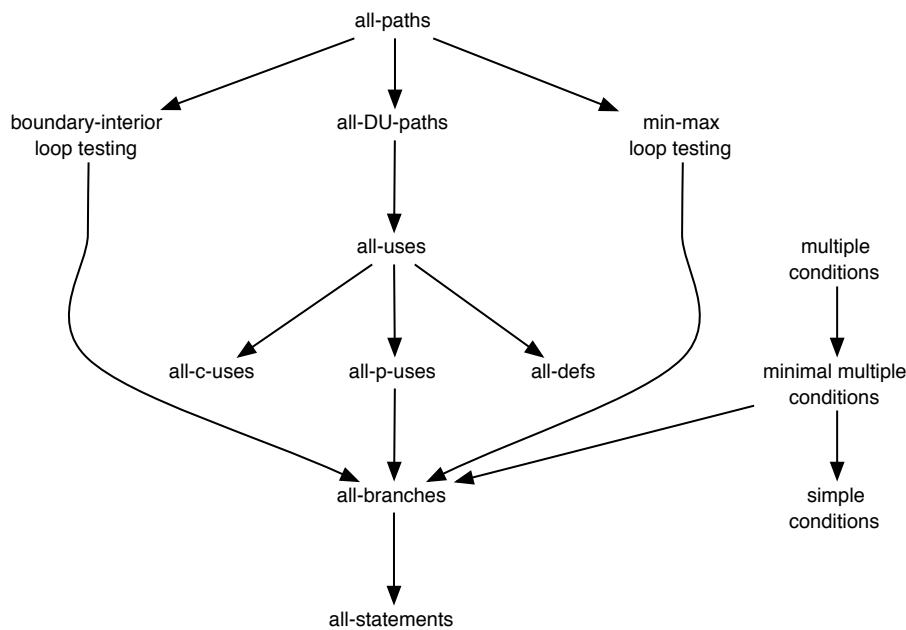
**Functional Testing** (*black-box testing, specification-based testing*) test cases are derived without knowledge of the source code structure or the implementation. Test cases are derived solely based on attributes of the specification or operational environment.

- equivalence class partitioning
- boundary value analysis
- [FIXME: Test spezieller Werte!]
- [FIXME: Zufallstest!]
- [FIXME: Test von Zustandsautomaten!]

**Statical Testing** is conducted on the source code and the code to be tested is *not* executed.

- Inspections
- Reviews

**Figure 1:** Subsumption Hierarchy of Structural Testing Coverage



- Walkthroughs
- Static Analysis

Functional testing can be used in conjunction with structural testing: after test scenarios have been derived based on the specification, an instrumented test item can be used to determine the actual coverage achieved (property of structural testing), and if needed, additional test cases can be derived.

## 4 Structural Testing

The objective is to *cover* the software structure, executing as much code as possible. Metrics are derived on the percentage of coverage achieved, which in turn is based on control flow or data flow. Formally, coverage is defined in terms of *flow graphs*. Figure 2 on page 6 show the subsumption hierarchy of structural testing coverage criterions.

### 4.1 Control Flow Oriented Testing

Control flow oriented testing belongs to the class of *white-box testing*. Based on the static control flow structure of the program, test cases are derived and the program is executed with the test cases. The partial order of statement execution as defined by the semantics of the language can be represented by a CFG (Control Flow Graph). See figure 2 on page 6 for an example of a CCFG (Compact Control Flow Graph).

**Definition 1 (Control Flow Graph, Compact Control Flow Graph)** Let  $P$  be a program. The CFG of  $P$  is a directed graph  $G = (N, E, n_{start}, n_{final})$ .  $N$  is the set of nodes,  $E \subseteq N \times N$  the set of directed nodes,  $n_{start} \in N$  the first node without any predecessor and  $n_{final} \in N$  the final node without any successor.

A directed edge from node  $i$  to node  $j$  is called a *branch* and describes a static control flow from  $i$  to  $j$ .

A *block* of a CFG is a non-empty sequence of nodes  $n_1, \dots, n_k \in N \setminus \{n_{\text{start}}, n_{\text{final}}\}$ ,  $k \geq 1$  for which the following conditions about their associated source code lines  $z(n_1), \dots, z(n_k)$  hold true:

- the block is executed exclusively at the first line  $z(n_1)$ .
- If  $z(n_1)$  has been executed and  $k \geq 2$ , all following lines  $z(n_2), \dots, z(n_k)$  will be executed in exactly this order. If  $k = 1$ , only  $z(n_1)$  will be executed.
- With respect to the first two conditions,  $k$  is maximal.

Thus, sequences of nodes that are connected by only one edge and consist of only one entrance and only one exit branch can be compacted to *blocks*, thus yielding a CCFG.

A series of nodes and edges, starting at  $n_{\text{start}}$  and ending at  $n_{\text{final}}$  is called a *path*. □

#### 4.1.1 Statement Coverage

*Statement Coverage*, also known as  $C_0$ -coverage is the least restrictive criterion. To achieve  $C_0$ -coverage, test cases are derived with the intention to execute all nodes of the control flow graph at least once. This is shown in figure 3 on page 7.

#### 4.1.2 Branch Coverage

*Branch Coverage*, also known as  $C_1$ -coverage is more restrictive than  $C_0$ -coverage. To achieve  $C_1$ -coverage, test cases are derived with the intention to execute all branches of the control flow graph at least once. This is shown in figure 4 on page 8.

#### 4.1.3 Condition Coverage

To fully achieve  $C_1$ -coverage, it is sufficient to execute each condition once to ‘true’ and to ‘false’. This does not take into account that conditions might be composed of several predicates. *Condition Coverage* tries to overcome this. *Simple Condition Coverage* ( $C_2$ -coverage) just requires that each atomic predicate are evaluated once to ‘true’ and once to ‘false’. *Multiple Condition Coverage* ( $C_3$ -coverage) requires that each possible combination of atomic and composed predicates be evaluated once to ‘true’ and once to ‘false’. The problem with  $C_3$ -coverage is that number of test cases required rises exponentially with the number of atomic predicates.

*Minimum Multiple Condition Coverage* is a pragmatic combination of  $C_2$  and  $C_3$  coverage: each predicate is evaluate to both values ‘true’ and ‘false’, be it atomic or composed predicates. Coverage is computed by this formula.  $p_t$  is the number of predicates evaluated at least once to ‘true’ during testing and  $p_f$  correspondingly the number of predicates evaluated at least once to ‘false’ during testing.  $p$  is the total number of atomic and composed predicates.

$$C_{\text{min-mult}} = \frac{p_t + p_f}{2p}$$

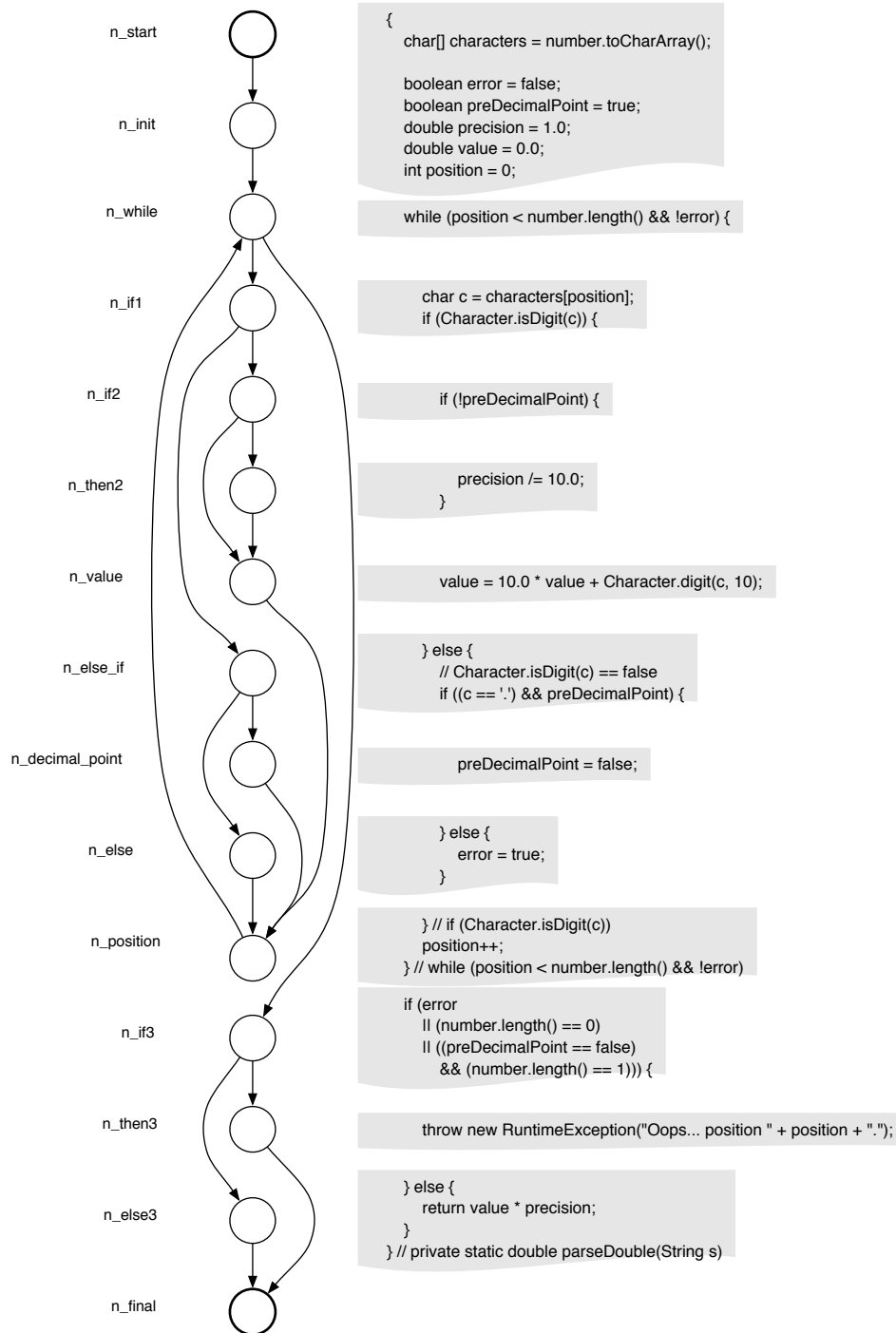
#### 4.1.4 Path Coverage and Boundary Interior Loop Testing

*Path Coverage* is the most comprehensive coverage criterion. *Complete path coverage* ( $C_4$ -coverage) means execution of all *paths* in the CFG including all possible loop iterations. The number of path for a program with just one simple loop increases linearly with the number of possible loop iterations.

The *Boundary Interior Loop Testing* focuses on testing loops but tries to minimize the number of necessary test cases.

- The first class includes all paths that to not execute the loop body.
- The second class includes all paths that execute the loop body exactly once (*boundary test*).
- The third class includes all paths that execute the loop body exactly twice thus testing the loop body (*interior test*).

**Figure 2:** A Compact Control Flow Graph



**Figure 3:** A Compact Control Flow Graph Showing Statement Coverage

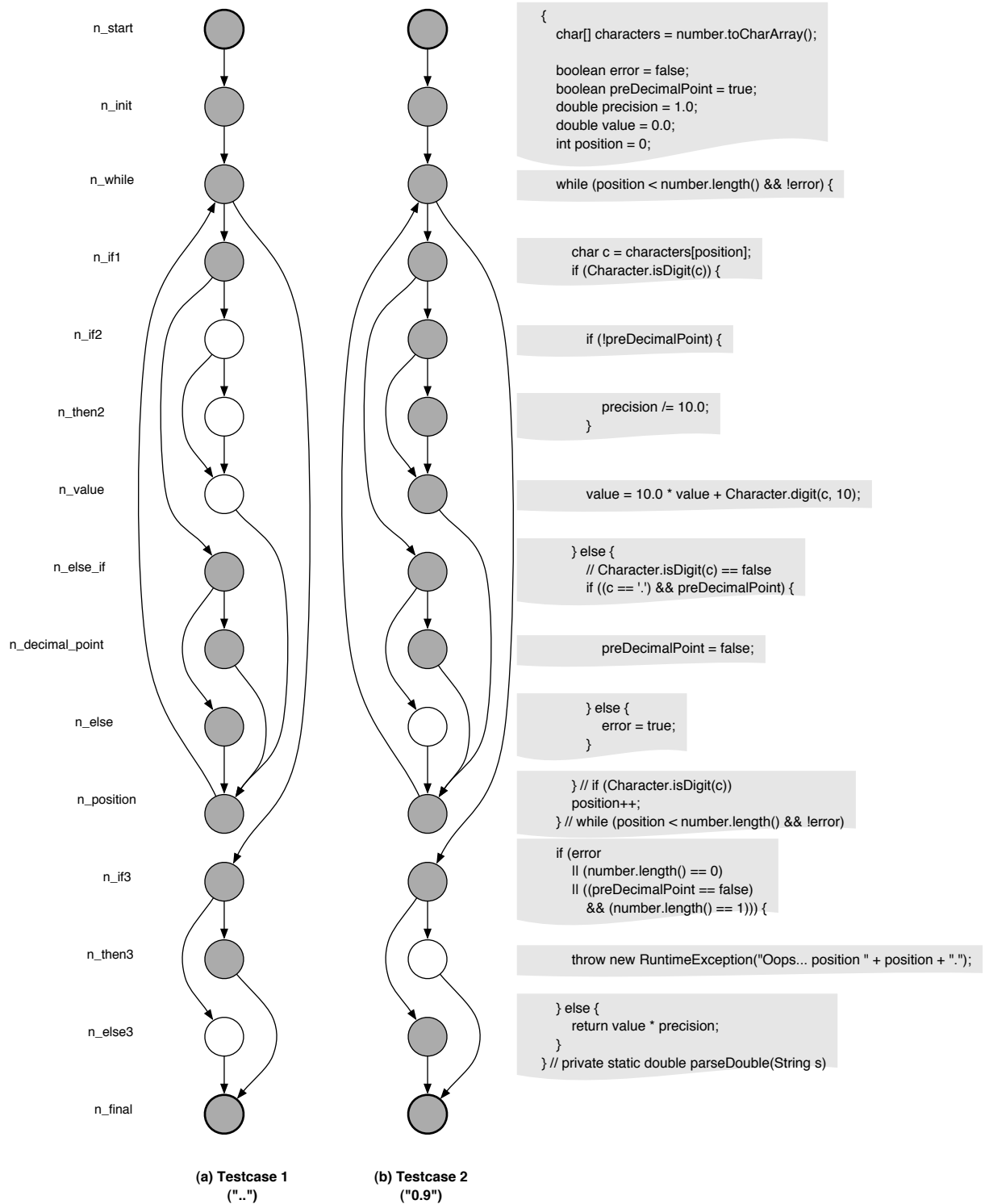
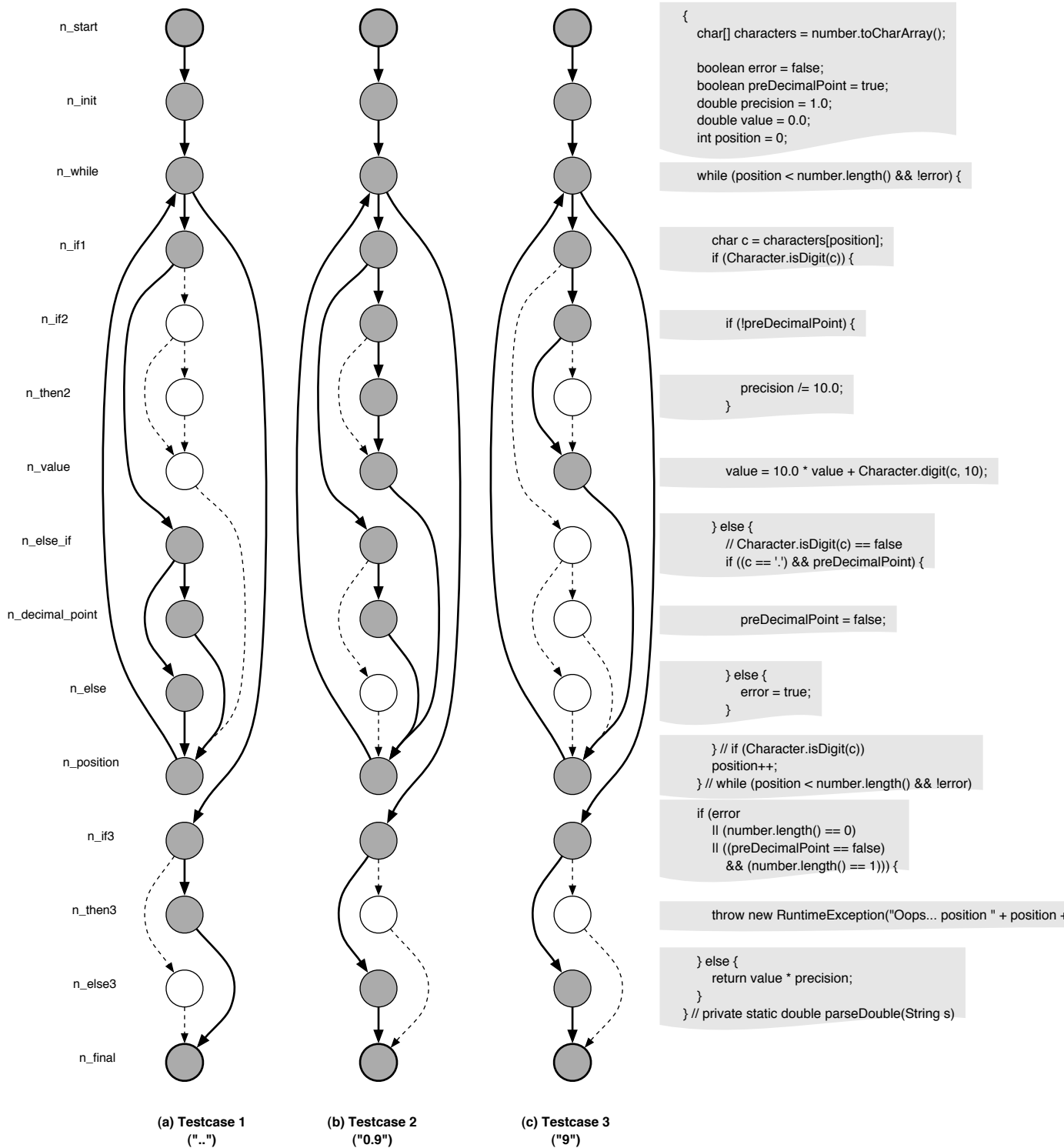


Figure 4: A Compact Control Flow Graph Showing Branch Coverage



**Table 1:** Minimum Multiple Condition Coverage

Predicate	Test Cases					
	"g"	"z"	".g"	".."	""	"."
Coverage achieved	46 %	71 %	89 %	93 %	96 %	100 %
<code>position &lt; number.length()</code>	t,f					
<code>!error</code>	t	t,f				
<code>position &lt; number.length() &amp;&amp; !error</code>	t,f					
<code>Character.isDigit(c)</code>	t	f				
<code>!preDecimalPoint</code>	f		t			
<code>c == '.'</code>		f	t			
<code>preDecimalPoint</code>		t	t	t,f		
<code>c == '.' &amp;&amp; preDecimalPoint</code>		f	t			
<code>error</code>	f	t				
<code>number.length() == 0</code>	f	f	f	f	t	
<code>preDecimalPoint == false</code>	f	f	t			
<code>number.length() == 1</code>	t	t	f			
<code>preDecimalPoint == false &amp;&amp; number.length() == 1</code>	f	f	f	f	f	t
<code>error    (number.length() == 0)</code>						
<code>   ((preDecimalPoint == false)</code>	f	t				
<code>&amp;&amp; (number.length() == 1))</code>						

## 4.2 Dataflow Oriented Testing

**Definition 2 (Data Flow Graph)** The flow of values from definitions of a variable to its uses. (See figure ?? on page ?? for an example.) □

### 4.2.1 Defs/Uses

## 5 Functional Testing

### 5.1 Equivalence Class Partitioning

### 5.2 Boundary Value Analysis

### 5.3 [FIXME: Test spezieller Werte!]

### 5.4 [FIXME: Zufallstest!]

### 5.5 [FIXME: Test von Zustandsautomaten!]

## 6 Instrumentation

## 7 Debugging

Removing errors?

**Definition 3 (Debugging)** *Debugging* is the activity to localize errors and fixing them. □

### 7.1 The Java Debugger: JDB

JDB

[FIXME: tracing/step-wise execution, breakpoints, watching!]

## 8 Exercises

**Exercise 1** () Shown below is a method that determines the maximum digit of a passed non-negative integer value.

- Construct the CCFG.
- Derive test cases such that  $C_0$ -coverage of 100% is achieved.
- Derive test cases such that  $C_1$ -coverage of 100% is achieved.
- There is a defect in the program. Has the defect been detected by the test cases? Correct the defect and rerun the test cases.

```

1 private static int getMaxDigit(int number)
2 {
3     int maxDigit = 0;
4     int rest = number;
5     int digit;
6
7     while (rest > 0) {
8         digit = rest % 10;
9         if (digit > maxDigit)
10            maxDigit = digit;
11        rest = number / 10;
12    }
13
14    return maxDigit;
15 }
```

### List of Tables

1	Minimum Multiple Condition Coverage . . . . .	9
---	---	---

### List of Figures

1	Subsumption Hierarchy of Structural Testing Coverage . . . . .	4
2	A Compact Control Flow Graph . . . . .	6
3	A Compact Control Flow Graph Showing Statement Coverage . . . . .	7
4	A Compact Control Flow Graph Showing Branch Coverage . . . . .	8

### List of Acronyms

CCFG . . . . . Compact Control Flow Graph  
 CFG . . . . . Control Flow Graph

### References

Ernst Denert. *Software-Engineering – Methodische Projektentwicklung*. Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, Barcelona, 1992. ISBN 3-540-52404-0.

Hans-Werner Six, Christina Esser, Wilfried Lange, Detlef Meier, Thomas de Ridder, and Mario Winter. *Konzepte imperativer Programmierung*. FernUniversität - Gesamthochschule in Hagen, 10 1998. URL <http://www.fernuni-hagen.de/LVU/guidedtour/inhalte/2-tour/5-kursangebot\%/inf11612/index.html>. 1612.

James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 2000. URL <http://www.computer.org/spftware/so2000/pdf/s1070.pdf>.