

Testing, debugging and verification

*Notes based on lectures for DIT084 (Testing, debugging and verification)
at Gothenburg University, Autumn 2023*

Sebastian Pålsson

Last updated: **January 07, 2024**

Contents

1. Introduction	2
1.1. The contract methaphor	2
1.2. Specification, failure, correctness	2
2. Testing	2
2.1. The State of a program	2
2.1.1. Program to graph	3
2.2. Complexity of testing (and why we need models)	3
2.2.1. Software testing foundations	3
2.2.2. RIPR model (or fault and failure model)	4
2.2.3. The V model	4
2.3. Test automation	5
2.4. Regression testing	5
2.5. Software testability	5
2.6. Observability	5
2.7. Controllability	5
2.8. Test suit construction	6
2.8.1. Black box testing	6
2.8.2. White box testing	6
2.9. Coverage criteria	6
2.10. Control flow graph	6
2.10.1. Control flow graph notions	6
2.11. Types of coverage criteria	7
2.11.1. Statement coverage (SC)	7
2.11.2. Branch coverage (BC)	7
2.11.3. Path coverage (PC)	7
2.11.4. Logic coverage (LC)	7
2.11.5. Decision coverage (DC)	7
2.11.6. Condition Coverage (CC)	8
2.11.7. Modified Condition Decision Coverage (MCDC)	9
2.12. Input domain modeling	9
2.12.1. Strategies for generating blocks	10
3. Debugging	11
3.1. Motivation	11
3.2. The Steps Of Debugging	12
3.3. Problem Simplification	12
3.3.1. AUTOMATION OF INPUT SIMPLIFICATION	12
3.3.2. Short Quiz On Delta debugging	13
3.4. Observing outcome, State Inspection	13
3.4.1. The Quick-And-Dirty Approach : Print Logging	13
3.4.2. BASIC LOGGING IN JAVA	14
3.4.3. Using Debuggers	14
4. Formal Specification	14

4.1. Unit specification	14
5. Dafny	14
5.1. Fields	15

1. Introduction

How do we judge failure in software development? In general, we need a measure to judge failure. This is given by means of satisfaction of a specification. A specification is a set of requirements that a system must satisfy. A specification can be given in many forms, such as a formal mathematical description, a set of examples, or a set of requirements in natural language. The specification is the basis for testing, debugging and verification.

In simple terms:

$$\text{Spec} = \text{Require} + \text{Ensure} \quad [1]$$

where **Require** is the set of requirements the program must satisfy in order to behave correctly, and **Ensure** is the set of guarantees that the program must satisfy.

Example

Imagine a programs that sorts an array of integers. The program must satisfy the following specification:

- **Require:** Accept only input as an array of integers.
- **Ensure:** Return a sorted array of integers.

1.1. The contract methaphor

Same priciple as a **legal contract**, between supplier and client.

- **Supplier:** aka implementer, here a *class or method* in Java.
- **Client:** mostly *caller object or human user* calling main-method.
- **Contract:** One or more *pairs of ensures-requires clauses*, defining *mutual obligations* between supplier and client.

*“If the caller (the client) of the method **m** (the supplier) gives inputs which fulfils the required **precondition**, then the method **m** ensures that the **postcondition** holds after **m** finishes execution.”*

1.2. Specification, failure, correctness

A failure/correctness is relative to a specification.

- A method **fails** when called in a *state fulfilling the required precondition* of its contract and it *does not terminate in a state fulfilling the postcondition* to be ensured.
- A method is **correct** whenever it is started in a *state fulfilling the required precondition*, then it *terminates in a state fulfilling the postcondition* to be ensured.
- **Correctness:** Proving absence of failures.

2. Testing

Some terminology regarding faults, errors & failures:

- **Fault:** a static defect in the software.
- **Error:** an incorrect internal state that is the manifestation of some fault.
- **Failure:** external, incorrect behaviour with respect to expected behaviour.

2.1. The State of a program

A program can be seen as a function that maps a state to a state. The state of a program is the values of all variables at a given point in time.

Inputs in the program in Figure 1 does not always trigger failures. This is the case for most programs!

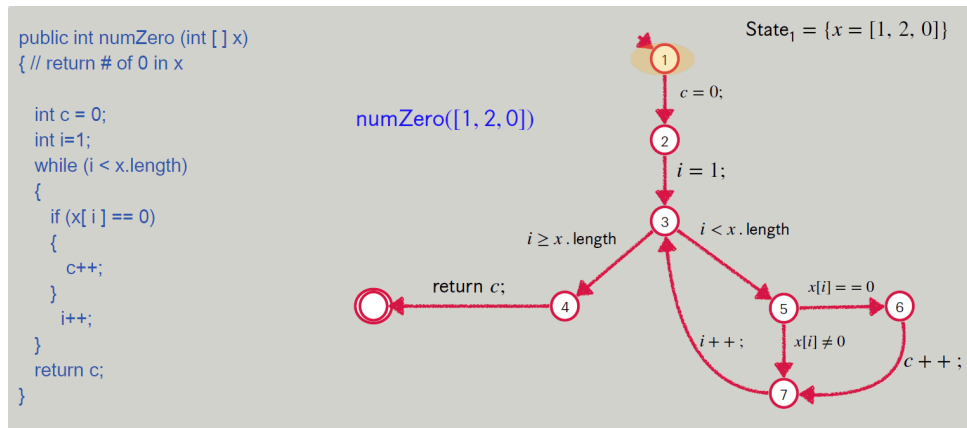


Figure 1: A graph showing the states of a program

2.1.1. Program to graph

How does one convert a program to a state graph?

A labelled graph G is of the form

$$G = (N, E, L) \quad [2]$$

where N is a set of nodes, E is a set of edges and L is a set of labels. $E \subseteq N \times L \times N$

A program P is a set of statements

$$P = \{s_1, s_2, \dots, s_n\} \quad [3]$$

The following function is used to transform the program to a graph:

$$[\cdot]^{0,f} : S \rightarrow 2^E \quad [4]$$

The function takes an initial state 0 and a final state f and a **statement** and **return a set of edges**. We use the special state \perp to mean an exit state (for return statements).

The transformation rule are as follows:

$$[s_1; s_2]^{0,f} = [s_1]^{0,1} \cup [s_2]^{1,f} \quad [5]$$

where 1 is a fresh state.

Example

Some examples of transformations:

- $[x = a;]^{0,f} = \{(0, f = a, f)\}$
- $[\text{return } a;]^{0,f} = \{(0, \text{return } a, \perp)\}$
- $[\text{if}(b)\{s_1\}]^{0,f} = \{(0, b, 1), (0, !b, f)\} \cup \{s_1\}^{1,f}$
- $[\text{while}(b)\{s_2\}]^{0,f} = \{(0, b, 1), (0, !b, f)\} \cup \{s_2\}^{1,0}$

2.2. Complexity of testing (and why we need models)

No other engineering field builds products as complicated as software. The term correctness has no meaning. Like other engineers, we must use abstraction to cope with complexity. We use **discrete mathematics** to raise our level of abstraction.

Abstraction is the purpose of *Model-Driven Test Design* (MDTD). The “model” is an abstract structure.

2.2.1. Software testing foundations

Testing can only show the presence of failures, not their absence.

- **Testing:** Evaluate software by observing its execution.
- **Test Failure:** Execution of a test that results in a software failure.
- **Debugging:** The process of finding a fault given a failure.

Not all inputs will “trigger” a fault into causing a failure

2.2.2. RIPR model (or fault and failure model)

Four conditions necessary for a failure to be observed:

- **Reachability:** The location or locations in the program that contain the fault must be reached.
- **Infection:** The state of the program must be incorrect.
- **Propagation:** The infected state must cause some output/final state to be incorrect.
- **Reveal:** The tester must observe part of the incorrect state.

The RIPR model is a framework for understanding and categorizing software bugs. It stands for Reach, Infection, Propagation, and Reveal. Reach refers to the execution reaching the defect, Infection is when the defect causes an incorrect program state, Propagation is when this incorrect state leads to incorrect behavior or output, and Reveal is when this incorrect behavior is observed.

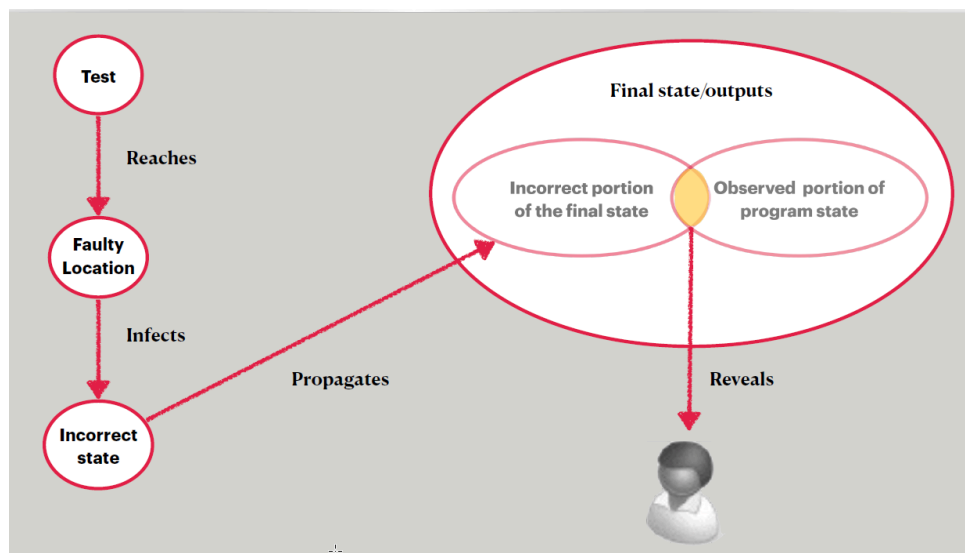


Figure 2: The RIPR model

2.2.3. The V model

The V-model emphasizes a parallel relationship between development and testing stages. Each development stage has a corresponding testing phase, ensuring that issues are identified and fixed early. *It's called the V-model due to its V-like structure, representing the sequence of execution of processes.*

Each testing phase uses a different type of testing level:

- **Acceptance testing::** Assess software with respect to user requirements
- **System Testing::** Assess software with respect to system-level specification.
- **Integration Testing:** Assess software with respect to high-level design
- **Unit Testing:** Assess software with respect to low-level unit design

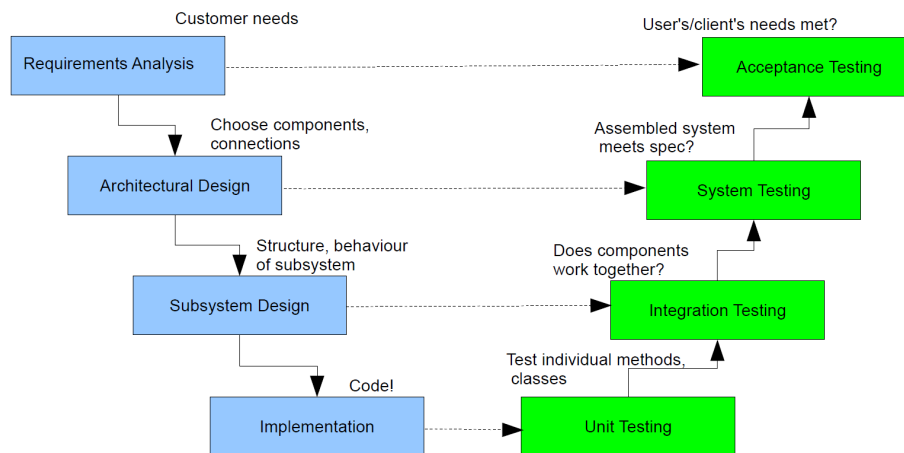


Figure 3: The V model

2.3. Test automation

The use of software to control *the execution* of tests, the *comparison* of actual outcomes to predicted outcomes, the *setting up* of test preconditions, and other test *control* and test reporting functions.

- Reduces cost
- Reduces human error
- Reduces variance in test quality from different individuals
- Reduces cost of **regression testing**

2.4. Regression testing

Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes such as enhancements, patches or configuration changes, have been made to them.

- Orthogonal to other mentioned tests
- Testing that is done **after changes** in the software (updates)
- Standard part of maintenance phase of software development

The purpose of regression testing is to gain confidence that changes did not cause new failures.

2.5. Software testability

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. *In simple terms: how hard is it to find faults?*

Testability is a condition of two factors:

- How to provide the test values to the software?
- How to observe the results of test execution?

2.6. Observability

How easy it is to observe the behaviour of a program in terms of its outputs, effects on the environment and other hardware and software components.

Software that affects hardware devices, databases, or remote files have low observability!

2.7. Controllability

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviours.

- Easy to control software with inputs from keyboards
- Inputs from hardware sensors or distributed software is harder

2.8. Test suit construction

A test suite is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviours.

- Most central activity of testing
- Determines if we have enough test cases (stopping criteria)
- Determines if we have the right test cases (representative test cases)

The quality of test suites defines the quality of the overall testing effort. When presenting test suites, we show only relevant parts of test cases.

2.8.1. Black box testing

Deriving test suites from external descriptions of the software, e.g.

- Specifications
- Requirements / Design
- Input space knowledge

2.8.2. White box testing

Deriving test suites from the source code, e.g.

- Conditions
- Branches in execution
- Statements

Modern techniques are a hybrid of both black- and white box

2.9. Coverage criteria

Most metrics used as quality criteria for test suites, describe the degree of some kind of coverage. These metrics are called coverage criteria.

These are crucial for testing *safety critical software*.

There are certain categories of coverage criteria:

- Control flow graph coverage (white box)
- Logic coverage (white box)
- Input space partitioning (black box)

2.10. Control flow graph

Represent a method to test as a graph, where:

- Statements are nodes
- Edges describe control flow between statements
- Edges are labelled with conditions

Important

Rules of transformation can be read in Section 7.3.1 “Structural Graph coverage for source code” from the book: Introduction to software testing. **This is required**

2.10.1. Control flow graph notions

- **Execution path:** a path through the control flow graph that starts at the entry point and is either infinite or ends at one of the exit points.
- **Path condition:** a condition causing execution to take some path p
- **Feasible execution path:** an execution path for which a satisfiable path condition exists.

Note

A branch or statement is feasible if it is contained in at least one feasible execution path.

2.11. Types of coverage criteria

2.11.1. Statement coverage (SC)

Satisfied by a test suite TS , *iff* for every node n in the control flow graph there is at least one test in TS causing an execution path via n .

2.11.2. Branch coverage (BC)

Satisfied by a test suite TS , *iff* for every edge e in the control flow graph there is at least one test in TS causing an execution path via e . *Note that this is a stronger criterion than statement coverage.*

2.11.3. Path coverage (PC)

Satisfied by a test suite TS , *iff* for every feasible execution path p in the control flow graph there is at least one test in TS causing an execution path via p . *Note that this is a stronger criterion than branch coverage.*

2.11.4. Logic coverage (LC)

Satisfied by a test suite TS , *iff* for every predicate p in the control flow graph there is at least one test in TS causing an execution path via p .

Logical (boolean) expressions can come from many sources. We focus on decisions in the source code (*if*, *while*, *for*, *etc.*).

2.11.5. Decision coverage (DC)

Let the decisions of a program p , $D(p)$, be the set of all logical expressions which p branches on.

For a given decision d , DC is satisfied by a test suite TS if it:

- Contains at least two tests,
- one where d evaluates to true,
- one where d evaluates to false

For a given program p , DC is satisfied by TS if it satisfies DC **for all** decisions $d \in D(p)$.

Example

Example

For a decision `if(((a < b) || D) && (m ≥ n * o))`,
 DC is satisfied for instance if TS triggers executions with:

`a = 5, b = 10, D = true, m = 1, n = 1, o = 1`

and

`a = 10, b = 5, D = false, m = 1, n = 1, o = 1`

D is not important

Inner Value Problem

- The above values are **not test case inputs**, but **values at the time of executing the decision**.
- Separate problem to find corresponding input values

Implicit Value Problem

- Java has implicit decisions (e.g. potential null-pointer access)

Figure 4: Decision coverage example

2.11.6. Condition Coverage (CC)

Let the **conditions of a program** p , $C(p)$, be the **set of all boolean sub-expressions** c of decisions in $D(p)$, such that c does not contain other boolean sub-expressions.

Given the decision `((a < b) || D) && (m ≥ n * o)`, the conditions are: **(a < b)**, **D**, and **(m ≥ n * o)**.

Condition Coverage (CC)

For a given condition c , CC is satisfied by a test suite TS if it

- Contains **at least two tests**
- one where **c evaluates to false**
- one where **c evaluates to true**

For a given program p , CC is satisfied by TS if it satisfies CC for all conditions $c \in C(p)$.

Figure 5: Condition coverage

Example

For each condition in $((a < b) \vee D) \wedge (m \geq n * o)$,
 CC is satisfied, for instance, if TS triggers executions with:
 $a = 10, b = 5, D = \text{true}, m = 1, n = 1, o = 1$
 and
 $a = 5, b = 10, D = \text{false}, m = 1, n = 2, o = 2$

No subsumption

- CC does not subsume DC
- DC does not subsume CC
- Consider: $p \vee q$
 - $\{p = \text{False}, q = \text{True}\}, \{p = \text{True}, q = \text{False}\} \rightarrow$ CC not DC
 - $\{p = \text{False}, q = \text{True}\}, \{p = \text{False}, q = \text{False}\} \rightarrow$ DC not CC

Figure 6: Condition coverage example

2.11.7. Modified Condition Decision Coverage (MCDC)

Modified Condition Decision Coverage (MCDC)

For a given **condition** c , **decision** d , MCDC is satisfied by a test suite TS if it

- contains **at least two tests**,
- one where **c evaluates to false**,
- one where **c evaluates to true**,
- **d evaluates differently in both**, and
- **other conditions in d evaluate identically** in both

For a given program p , MCDC is satisfied by TS if it satisfies MCDC for all $c \in C(p)$.

Figure 7: Modified Condition Decision Coverage

Example

For condition $a < b$ in decision
 $((a < b) \vee D) \wedge (m \geq n * o)$,
 MCDC is satisfied (for condition $a < b$) if TS triggers executions with,
 for instance:
 $a = 5, b = 10, D = \text{false}, m = 1, n = 1, o = 1$
 and
 $a = 10, b = 5, D = \text{false}, m = 1, n = 1, o = 1$

Note: To have MCDC for the whole decision also need test-cases for conditions **D** and **$(m \geq n * o)$** .

Figure 8: Modified Condition Decision Coverage example

2.12. Input domain modeling

- The input domain D : the possible values that input parameters can have.

- A partition q defines a set of equivalence classes (or simply blocks B_q) over D

A partition q satisfies: (completeness)

$$\bigcup_{b \in B_q} b = D \quad [6]$$

Blocks are pairwise disjoint:

$$b_i \cup b_j = \emptyset, \quad i \neq j, \quad b_i, b_j \in B_q \quad [7]$$

- Test values in the same block are assumed to contain *equally useful values*.
- Test cases contain values from each block.

Example

- Consider the domain of integer arrays.
- Are the following blocks a valid partitioning?
 - b_1 = sorted in ascending order
 - b_2 = sorted in descending order
 - b_3 = arbitrary order
- Answer: no!
 - The array [1] belongs to all blocks
 - Unclear whether the array null belongs to any block

Figure 9: Partitioning example

2.12.1. Strategies for generating blocks

- Valid vs. invalid values: all valid and all invalid (completeness)
- Sub-partition: valid values serving different functionality.
- Boundaries: values at or close boundaries often cause problems (stress testing).
- Normal use (happy path): the desired outcome.
- Missing blocks vs overlapping blocks (complete vs disjoint)
- Special values Pointers: (null and not null), collection: (empty and non empty), integer (zero a special value).

Example

- Identify testable functions. (triang() has one testable function)
- Identify all parameters of that affect the testable functions. (The sides of triang)
- Generate input domain model (IDM) (Abstraction interpretation of the input). (Characteristics of the triangle)
- Generate complete set of blocks for each characteristic. (e.g. Side relation to zero)
- Select representative values from each block.

```
public enum Triangle {Scalene, Isosceles, Equilateral, Invalid}
public static Triangle triang(int Side1, int Side2, int Side3)
//Side1, Side2, Side3 represent the lengths of the triangle sides
// Returns the appropriate enum value
```

Figure 10: Example: triang(0)

Partition	b1	b2	b3
q ₁ = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q ₂ = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q ₃ = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

Partition	b ₁	b ₂	b ₃	b ₄
q ₁ = "Length of Side 1"	greater than 1	equal to 1	equal to 0	less than 0
q ₂ = "Length of Side 2"	greater than 1	equal to 1	equal to 0	less than 0
q ₃ = "Length of Side 3"	greater than 1	equal to 1	equal to 0	less than 0

Parameter	b ₁	b ₂	b ₃	b ₄
Side 1	2	1	0	-1
Side 2	2	1	0	-1
Side 3	2	1	0	-1

If Side are floats,
this is defective
partition

Figure 11: Example: triang(0) - Solution

All combinations of blocks from all characteristics must be used. The number of tests is the product of the number of blocks for each partition. *All Combinations Coverage ACoC*

For *triang()*, we have three characteristics, each with 4 blocks. Thus, we need $4 \cdot 4 \cdot 4 = 64$ test cases.

Recall: *different choices of values from the same block are equivalent from testing perspective.*

3. Debugging

- How to systematically find source of failure
- Test-case to reproduce errors
- Finding a small failing input (if possible)
- Observing execution: Debuggers and Logging
- Program dependencies: data- and control

3.1. Motivation

- Debugging needs to be systematic
- Debugging may involve large inputs

- Programs may have thousands of memory locations
- Program may pass through millions of states before failure occurs

3.2. The Steps Of Debugging

1. Reproduce the error and try to understand the cause.
2. Isolate and minimise the different factors. (**Simplification**)
3. Eyeball the code, where could it be? (**Reason backwards**)
4. Devise experiments to test your hypothesis. (**Test hypothesis**)
5. Repeat step 3 and 4 until the cause of the bug is determined
6. Fix the bug and verify the fix
7. Create a regression test (See below)

Regression testing is a type of software testing that checks if recent code changes have negatively impacted existing features. It involves re-running previously created test cases after code modifications to catch any unintended side effects and ensure the ongoing stability of the software.

3.3. Problem Simplification

As described in the different steps of debugging, simplification is a way to determine the bug. The idea behind simplification is to minimize the failing input, so that it will be easier to understand what inputs causes the bug.

Simplification can be reached by **Divide-and-Conquer**, where you ->

1. Cut away one half of the test input
2. Check if any of the halves still exhibit failure.
3. Repeat, until minimal input has been found.

Although this works in some scenarios, this method has the following problems ->

- Tedious to re-run test manually
- Boring, cut and paste, re-run ...
- What, if none of the halves exhibits a failure?

Because of the problems with **Divide-and-Conquer**, in most cases automation of input simplification is more favourable.

3.3.1. AUTOMATION OF INPUT SIMPLIFICATION

Automation of input simplification in debugging involves using tools or algorithm to automatically reduce the complexity of input data. This helps identify and isolate bugs more efficiently, especially in large or complex software systems.

One example of such an algorithm is **Delta Debugging**. Delta debugging is a software debugging technique that aims to isolate and identify the cause of a failure in a program by systematically narrowing down the input that triggers the failure. The term “delta” refers to the minimal change needed to reproduce the failure.

The algorithm **Delta debugging** or **DD-Min** as it is also called, works as seen in figure 4 below.

- Let **c** be a **failing input** (sequence of individual inputs).
- Let **test(c)** run a test on **c** with **possible outcome PASS or FAIL**.
- **n** is the number of **chunks to split c into** (initially $n = 2$).
- We will **remove one chunk at the time**, and **test the remaining input**.

```

ddMin(c, n) =
1. If  $|c| == 1$  then return c
   Otherwise, systematically remove one chunk  $c_i$ , and test the remaining input  $c - c_i$ .
2. If there exist some  $c_i$  such that  $\text{test}(c - c_i) = \text{FAIL}$ , return ddMin(c - c_i, max(n-1, 2))
3. Else, if  $n < |c|$  return ddMin(c, min(2n, |c|))
4. Else, (can't split into smaller chunks) return c

```

Figure 12: Delta Debugging

3.3.2. Short Quiz On Delta debugging

Question :

Suppose $\text{test}(c)$ returns FAIL whenever c contains two or more occurrences of the letter X. Apply the ddMin algorithm to minimise the failing input array $[X, Z, Z, X]$. Write down each step of the algorithm, and the values of n (number of chunks). Initially, n is 2.

Solution :

```

Initial failing input: [X, Z, Z, X]
Initial n = 2, split into two chunks:
• [X, Z] ⇒ PASS (remove 2nd chunk)
• [Z, X] ⇒ PASS (remove 1st chunk)
Update n = 4 (see step 3), split [X, Z, Z, X] into four chunks:
• [X, Z, Z] ⇒ PASS (remove 4th chunk)
• [X, Z, X] ⇒ FAIL (remove 3rd chunk)
Update n = 3 (see step 2), split [X, Z, X] into three chunks:
• [X, Z] ⇒ PASS (remove 3rd chunk)
• [X, X] ⇒ FAIL (remove 2nd chunk)
Update n = 2 (see step 2), split [X, X] into two chunks:
• [X] ⇒ PASS (remove 2nd chunk)
• [X] ⇒ PASS (remove 1st chunk)
No further splits possible, minimal failing input is [X, X] from previous step.

```

```

ddMin(c, n) =
1. If  $|c| == 1$  then return c
   Otherwise, systematically remove one chunk  $c_i$ , and test the remaining input  $c - c_i$ .
2. If there exist some  $c_i$  such that  $\text{test}(c - c_i) = \text{FAIL}$ , return ddMin(c - c_i, max(n-1, 2))
3. Else, if  $n < |c|$  return ddMin(c, min(2n, |c|))
4. Else, (can't split into smaller chunks) return c

```

Figure 13: Delta Debugging Quiz Answer

3.4. Observing outcome, State Inspection

Mainly there are three different ways to perform state inspections of a program ->

- **Simple logging** : print statements
- **Advanced logging** : configurable what is printed based on level (e.g. OFF < FINE < INFO < WARNING < SEVERE), using e.g. Java's logging package.
- **Debugging tools** : e.g. Eclipse debugger or the Java debugger jdb (hand-in assignment 2)

3.4.1. The Quick-And-Dirty Approach : Print Logging

- **Manually add print statements at code locations to be observed**
- `System.out.println("size = " + size);`

Pros ->

- Simple and easy.
- No tools or infrastructure needed, works on any platform.

Cons ->

- Code cluttering.
- Output cluttering.
- Performance penalty, possibly changed behaviour (real time apps).
- Buffered output lost on crash
- Source code access required, recompilation necessary.

3.4.2. BASIC LOGGING IN JAVA

- Each class can have its own logger-object.
- Each logger has level:
- OFF < FINE... < INFO < WARNING < SEVERE
- Setting the level controls which messages gets written to log.
- Quick Demo: Dubbel.java

Pros ->

- Output cluttering can be mastered
- Small performance overhead
- Exceptions are loggable
- Log complete up to crash
- Etc.

Cons ->

- Code cluttering - don't try to log everything

3.4.3. Using Debuggers

Assume we have found a small failing test case and identified the faulty component.

Basic Functionality of a Debugger

- Execution Control: Stop execution at specific locations, breakpoints
- Interpretation: Step-wise execution of code
- State Inspection: Observe values of variables and stack
- State Change: Change state of stopped program
- Debugging tools: Eclipse GUI debugger or the Java debugger jbd

4. Formal Specification

Describing contracts of units (methods) in a mathematically precise (formal) language.

Motivation:

- Higher degree of precision
- Automation of program analysis
 - program verification
 - static checking
 - test case generation

4.1. Unit specification

Methods can be specified by referring to:

- result value
- initial values of formal parameters
- pre-state and post-state

5. Dafny

Object oriented language desinged to make it easy to write **correct** code.

-
- Allow annotations specifying program behaviour, as part of the language.
 - Automatically proves that the code adheres to specifications.
 - Absence of run-time errors, e.g. null-pointers, index-out-of-bounds etc.
 - Termination checking of loops.

5.1. Fields