Software Engineering 9

Solutions Manual

IAN SOMMERVILLE

These solutions are made available for instructional purposes only. Neither the author nor the publisher warrants the correctness of these solutions nor accepts any liability for their use. Solutions may only be distributed to students and it is a condition of distribution that they are only distributed by accredited instructors using 'Software Engineering, 9th edition' as a textbook. The solutions may be made available to students on a password-protected intranet but must not be made available on a publicly-accessible WWW server.

Preface

This solutions manual is intended to help teachers of software engineering courses in marking homework questions for students. Each chapter in the book has 10 exercises of different types, which you may set for students either as is or in a modified form. I have supplied answers to 50% of the exercises in this manual.

The exercises for which answers have not been supplied are, generally, of one of three types:

1. Simple exercises whose answers can be found in the text of the chapter. There are typically one or two of these questions in each chapter and they are intended to stimulate students to read the chapter.
2. Design problems for which there is a range of solutions and you have to use your judgment to decide if the solution is appropriate. Supplying a solution here would imply that there is only one right answer to the question.
3. Ethics-related questions as the aim of these questions is to encourage students to think about the ethics issues involved. The notion of a right and wrong answer does not apply in this case as the student's response to the question depends both on their cultural background and on their particular views on a topic. I suggest that these questions should be used to stimulate class discussions rather than as part of class tests.

It is important when marking the student's answers to exercises to see the supplied solutions as a guide only rather than a definitive statement of the only possible answer to the question. It is generally good educational practice to give students credit for what they know and if they produce credible answers that reveal they have thought about the exercise and have some knowledge of the topic, then this should be rewarded.

This solutions manual may be used in conjunction with the associated quiz book, which lists short questions and answers for each chapter in the book. These can be used for short class tests to assess if students have read the material or as self-assessment tests which the students complete in their own time.

If you think that I have made a mistake in some of these answers (quite possible), please let me know. In some cases, there are obviously several possible answers and you may disagree with my solutions. I'd be delighted to consider including your alternative solutions but I do not have time to engage in detailed email discussions about the exercises in the book.

Ian Sommerville
January 2010

8 Software Testing

8.2 Explain why testing can only detect the presence of errors, not their absence.

Assume that exhaustive testing of a program, where every possible valid input is checked, is impossible (true for all but trivial programs). Test cases either do not reveal a fault in the program or reveal a program fault. If they reveal a program fault then they demonstrate the presence of an error. If they do not reveal a fault, however, this simply means that they have executed a code sequence that – for the inputs chosen – is not faulty. The next test of the same code sequence – with different inputs – could reveal a fault.

8.4 You have been asked to test a method called 'catWhiteSpace' in a 'Paragraph' object that, within the paragraph, replaces sequences of blank characters with a single blank character. Identify testing partitions for this example and derive a set of tests for the 'catWhiteSpace' method.

Testing partitions are:

Strings with only single blank characters

Strings with sequences of blank characters in the middle of the string

Strings with sequences of blank characters at the beginning/end of string

Examples of tests:

The quick brown fox jumped over the lazy dog (only single blanks)

The quick brown fox jumped over the lazy dog (different numbers of blanks in the sequence)

The quick brown fox jumped over the lazy dog (1st blank is a sequence)

The quick brown fox jumped over the lazy dog (Last blank is a sequence)

The quick brown fox jumped over the lazy dog (2 blanks at beginning)

15 Dependability and Security Assurance

15.1 Explain when it may be cost-effective to use formal specification and verification in the development of safety-critical software systems. Why do you think that critical systems engineers are against the use of formal methods?

Formal methods can be cost-effective in the development of safety-critical software systems because the costs of system failure are very high and so additional cost in the development process is justified. Most safety-critical systems have to gain regulatory approval before they are used and it is a very expensive process to convince a regulator that a system is safe. The use of a formal specification and associated correctness argument may be less than the costs e.g. of additional testing to convince the regulator of the safety of the system.

Some developers of systems are against the use of formal methods because they are unfamiliar with the technology and unconvinced that a formal specification can be complete representation of the system. Furthermore, the problem with formal specifications are that they cannot be understood by system customers so they may conceal errors and give a false picture of the correctness of the system.

15.3 Explain why it is practically impossible to validate reliability specifications when these are expressed in terms of a very small number of failures over the total lifetime of a system.

To measure reliability you need to have statistically valid failure data for the system so you need to induce more failures than are specified in the given time period. However, because the number of failures is so low, this will take an unrealistically large amount of time.

27.7 Modify the insulin pump schema, shown in Figure 27.10, to add a further safety condition that the ManualDeliveryButton? can only have a non-zero value if the pump switch is in the manual position.

To specify that the manual delivery button can only have a non-zero value if the switch is in the manual position, you should add the following invariant to the state schema.

$$switch? \neq manual \Rightarrow ManualDeliveryButton = 0$$

27.8 Write a Z schema called SELF_TEST that tests the hardware components of the insulin pump and sets the value of the state variable HardwareTest?. Then modify the RUN schema to check that the hardware is operating successfully before any insulin is delivered. If not, the dose delivered should be zero and an error should be indicated on the insulin pump display.

SELF_TEST

Δ INSULIN_PUMP_STATE

(HardwareTest? = OK $\wedge$ Needle? = present $\wedge$ InsulinReservoir? = present $\Rightarrow$
status' = running $\wedge$ alarm! = off $\wedge$ display1!= "") $\vee$
(
status' = error
alarm! = on
Needle? = notpresent $\Rightarrow$ display1! = display1! $\cup$ "No needle unit" $\vee$
(InsulinReservoir? = notpresent $\vee$ insulin_available < max_single_dose)
$\Rightarrow$ display1! = display1! $\cup$ "No insulin" $\vee$
HardwareTest? = batterylow $\Rightarrow$ display1! = display1! $\cup$ "Battery low" $\vee$
HardwareTest? = pumpfail $\Rightarrow$ display1! = display1! $\cup$ "Pump failure" $\vee$
HardwareTest? = sensorfail $\Rightarrow$ display1! = display1! $\cup$ "Sensor failure" $\vee$
HardwareTest? = deliveryfail $\Rightarrow$ display1! = display1! $\cup$ "Needle failure"
)

The RUN schema should be modified to check that HardwareTest? is true before continuing operation.