Prof. Dr. A. Podelski,
Dr. B. Westphal
C. Schilling

Summer term 2018

Softwaretechnik/Software Engineering

http://swt.informatik.uni-freiburg.de/teaching/SS2018/swtvl

Exercise Sheet 6

Early submission: Wednesday, 2018-07-18, 12:00 Regular submission: Thursday, 2018-07-19, 12:00

Exercise 1 – Black-box Testing (7 Points + Bonus)

Note: You need to contact your tutor to work on this exercise, as described in the appendix.

Recall the discount calculation for the book rental company from Exercise 3 on Exercise Sheet 3. Meanwhile you have discussed the decision table (see Table 1) with the customer and agreed that it precisely reflects what the customer wants. After the discussion, you have instructed a programmer of your company to implement the book discount calculation in a program called `discount_calc`. The purpose of this program is, given the relevant data of a client and the base price of a book, to find out whether the client is allowed to borrow the book, and, if so, to determine the discounted price. The specification of the program is given by the rules according to the decision table, including the following additional instructions:

- The program `discount_calc` takes five input arguments (in this order):
  - `O` is the number of books owned by the client.
  - `B` is the number of books borrowed by the client.
  - `R` is the number of bad ratings of the client.
  - `S` is a flag for the student status of the client.
  - `P` is the base price of the new book to be borrowed in cents.

  The arguments `O`, `B`, `R`, and `P` are integers in the range [0, 32767]. The argument `S` is a Boolean flag (i.e., an integer in the range [0, 1]).

  The program is supposed to yield the final price (with the discount already applied) in cents, and rounding is performed toward zero. Lending refusal is encoded by value -1 and an invalid input by value -2, thus the program is supposed to yield values in the range $[-2, 32767]$.

Users of the program observed some calculations being performed incorrectly. Unfortunately, they cannot remember which inputs caused the program to misbehave.
You are now in the role of the test engineer and are requested to create a test suite.

(i) Submit a *test suite* to test whether the program violates its specification. Describe the strategy you used to select the test cases. For one of your test cases, discuss what exactly you can conclude from a positive and negative outcome of an execution of this test case, respectively. (5)

Submit a test suite via a test script according to the instructions in Appendix A. The test script should be a separate text file, i.e., not embedded in the PDF with your other solutions.

Hint: You can assume that the project leader allotted a time budget for the creation of the test suite equivalent to 5 exercise sheet points. Submit a number of test cases that is reasonable to achieve and explain within that time budget.

DT: Book Rental		R1	R2	R3	R4	R5	R6	R7	R8
C1	professional	×	×	–	–	–	–	*	*
C2	supporter	–	–	×	×	–	–	*	*
C3	student, ≤ 5 books	×	–	×	–	×	–	*	*
C4	1..2 bad ratings	–	–	–	–	–	–	×	–
C5	3.. bad ratings	–	–	–	–	–	–	–	×
A1	refuse	–	–	–	–	–	–	–	×
A2	no discount	–	–	–	–	–	×	×	–
A3	3% discount	–	–	–	×	–	–	–	–
A4	5% discount	–	×	–	–	×	–	–	–
A5	3+5% discount	–	–	×	–	–	–	–	–
A6	5+5% discount	×	–	–	–	–	–	–	–
$\neg[(C1 \wedge C2) \vee (C4 \wedge C5)]$									
$\not\downarrow = (\{A2, \ldots, A6\} \times \{A2, \ldots, A6\}) \setminus \{(Ai, Ai) \mid i = 2, \ldots, 6\}$									

Table 1: Decision table for the book discount calculation.
Note that the annotation of condition C3 is understood as 'the client is a student and has not yet borrowed more than five books' *including* the book for which the price is currently calculated.

(ii) Your tutor will execute your test suite (using your test script, hence it is important that you stick to the file format given in the appendix). There are the following possibilities:

- Your test suite finds one error in the program. (1 Bonus)
- Your test suite finds a second error in the program. (3 Bonus)
- Your test suite finds a third error in the program. (5 Bonus)
- Your test suite finds a fourth error in the program. (100 Bonus)
- Your test suite finds more errors in the program. (1000 Bonus each)

For the assignment of bonus points, an error is a wrongly implemented rule, or a violation of an additional specification for the program as given above, like the encoding of Boolean flags. That is, your test suite may contain more than one successful test case, but each of them might identify the same error.

(iii) What is the number of test cases required to *exhaustively* test the program *inside its specification*?

Assume that you can execute around 1000 tests per second. How many seconds (hours? days? ...) would it then at least take to exhaustively test the program using one (single core) CPU? (2)

Exercise 2 – Testing & Coverage Measures (6 Points + 3 Bonus)

Consider the Java function `convert` shown in Figure 1. The function is supposed to convert the string representation (decimal, base 10) of an integer to the represented integer value. For example, the string "`123`" is converted to the integer 123 and the string "`-378`" is converted to the integer -378.
A digit is a character from the set {'`0`', ..., '`9`'}.

```
1  int convert(char[] str) throws Exception {
2          if (str.length == 0)
3                  return 0;
4          if (str.length > 6)
5                  throw new Exception("Length exceeded");
6          int number = 0, digit, i = 0;
7          if (str[0] == '-')
8                  i = 1;
9          while (i < str.length) {
10                 digit = str[i] - '0';
11                 if (digit <= 0 || digit > 9)
12                         throw new Exception("Invalid character");
13                 number = number * 10 + digit;
14                 i = i + 1;
15         }
16         if (str[0] == '-')
17                 number = -number;
18         if (number > 32767 || number < -32768)
19                 throw new Exception("Range exceeded");
20         return number;
21 }
```

Figure 1: Function `convert`.

The function should check the input string `str` and handle the following exceptional cases:

- The input string has strictly more than 6 characters.
- The input string has at most 6 characters and the string contains an invalid character. The first character is invalid if it is not a minus (‘`-`’) or a digit, and each further character is invalid if it is not a digit.
- The input string has at most 6 characters, none of them invalid, and the denoted integer value is outside the range $[-32768, 32767]$.

If none of the exceptional cases applies, for $n \geq 0$ let $c_0, \ldots, c_{n-1}$ be the 1st, …, n-th character in `str`. The expected return value is $\sum_{i=0}^{n-1} c_i \cdot 10^{n-i-1}$ if the first character c_0 is not ‘`-`’ and $-\sum_{i=1}^{n-1} c_i \cdot 10^{n-i-1}$ if the first character c_0 is ‘`-`’. (Unlike other implementations of string-to-integer conversions, this implementation should return 0 for the empty string and the string “`-`”.)

(i) Give an *unsuccessful test suite* for `convert` that achieves 100% *statement coverage* and 100% *branch coverage*.

Recall that submissions need a presentation that is comprehensible enough to be evaluated. Make sure that it is easy to see what your results are, which statements and branches are covered by each test case, etc. (5)

(ii) Modify your test suite from Task (i) such that the test suite is still *unsuccessful* and still achieves 100% *statement coverage*, but now achieves *strictly less* than 100% *branch coverage*. (1)

(iii) Consider the expression $expr \equiv x = \texttt{'a'} \wedge (y \leq 0 \vee (z \neq 10 \wedge v)) \vee \neg(w < 100)$ for inputs x:`char`, y:`int`, z:`int`, v:`bool`, w:`int`. Give a test suite that achieves the maximum possible *term coverage* for *expr*. What is the maximum possible term coverage? Justify your answer. (3 Bonus)

Exercise 3 – Grading Test Suites (2 Points)

Assume that there is a task to create an unsuccessful test suite for the function shown in Figure 2(a) (a complicated implementation of the function $\min(n, 255)$). The test suite should achieve 100% branch coverage and 100% statement coverage, and document how this coverage is achieved. Consider the proposed solution shown in Figure 2(b). Branches and statements are denoted by i_2, s_3, s_5, respectively, counted from top to bottom.

Assume that a correct solution would be worth 4 points. There is a proposal to grade the proposed solution shown in Figure 2(b) with 0 points. Argue in favour of this proposal.

```
1 int min255( int n ) {
2   if (n < 255)
3     return n;
4   else
5     return 255;
6 }
```

(a) Function `min255()`.

input	$i_2/true$	s_3	$i_2/false$	s_5
0	✔	✔	✘	✘
255	✘	✘	✔	✔

(b) Proposed solution.

Figure 2: Function under test and proposed test suite.

Exercise 4 – Verification with PD Calculus (5 Points)

Consider the program `multiply` shown in Figure 3. It is annotated with pre- and post-conditions and implements integer multiplication by successive addition. The operands are x and y and the result is stored in the variable *result*.

$\{x \geq 0 \wedge y \geq 0\}$

```
result = 0;
i = 0;
while (i < y) do
  result = result + x;
  i = i + 1;
od
```

$\{result = x \cdot y\}$

Figure 3: Program `multiply`.

The goal of this exercise is to use the proof system PD to show that `multiply` is partially correct. We approach this goal in three steps.

(i) Give an *invariant* for the while loop that enables you to prove the correctness of the program. (1)

(ii) Apply the rules of the proof system PD to *derive a proof* that your invariant is indeed a loop invariant. Specify which rules or axioms you use at every proof step. (3)

(iii) Apply the rules of the proof system PD to *derive a proof* that `multiply` is *partially correct*. Specify which rules or axioms you use at every proof step.

What observation from the proof can you make about the pre-condition of `multiply`? (1)

Exercise 5 – Verification with VCC (10 Bonus)

(i) We implemented the program `multiply` from Exercise 4 as a C function (see the file `multiply.c` in Ilias).

For a start, assume that `multiply` is used in a bigger program where it is only called with values for x between 0 and 15 and values for y between 0 and x, i.e., in the considered bigger program, all callers actually guarantee the pre-condition $\{0 \leq y \leq x \leq 15\}$.

Annotate the function with the pre-condition, the post-condition, and the loop invariant (and whatever else you consider necessary) using the VCC syntax for annotations, and use VCC[1] to *verify* the annotated function. (3 Bonus)

(ii) Discuss the relation of the verification result you obtained from VCC and your result from Exercise 4.(iii): Given the knowledge from Exercise 4.(iii), would you have expected that VCC can verify the function? (1 Bonus)

We have observed that, with testing, it is easy to provide an unsuccessful test suite for a program which is not correct wrt. its specification. How is it with VCC? To get a first impression, inject a fault into the C function or the specification such that the function *does not* satisfy the specification anymore. Describe your modification (and in how far it causes a fault) and the behaviour of VCC when applied to the modified function. (1 Bonus)

(iii) Now we want to increase the value of N. Hence we adapt the pre-condition to the following:

$$\{0 \leq y \leq x \leq N\}$$

How many (unsuccessful) test cases would you need to exhaustively test the function `multiply` within the new specification (for a fixed value of N)?

If we use VCC to verify the respective considered C function for the values $N = 15$, 150, 1500, 15000, we observe that the verification time reported by VCC is approximately the same. Is this measurement plausible? (2 Bonus)

(iv) Assume that you are unsure about your proof from Exercise 4.(iii) and you would like to confirm the result using VCC. Modify the C function such that it in particular considers the original pre-condition (cf. Figure 3), and try to verify it using VCC.

Describe what you expect to be the outcome of this experiment and what the results of the verification attempt with VCC were. Interpret the output of VCC. (1 Bonus)

(v) Consider again the implementation in `multiply.c` from Task (i). A colleague observed that the variable `i` is not actually needed. Instead, one can decrement the variable `y` in the loop until it reaches the value 0.

Provide a *correct* implementation of this idea in a new C function. (2 Bonus)

You have to convince your tutor that your implementation is correct. One way to do that is to annotate the function such that VCC can verify it.

To enable your tutor to reproduce your results, all code artifacts of this exercise (with appropriate comments or explanations, if necessary) should be submitted as separate text files, i.e., not embedded in the PDF.

[1] `http://rise4fun.com/vcc`

A Instructions for the Black-box Testing Exercise

Before you begin, you need to request a team ID from your tutor by e-mail. We have created a separate binary to test for each team. Your tutor will assign a number from 0 to 99. The team ID will be used to evaluate your results on the particular implementation of the discount calculation. In order to create a test script, login to one of the Linux machines in the computer pool and perform the following steps:

- Create a directory where you would like to store your results.
- In your directory of choice, execute the installation script as follows[2]:

  ```
  /home/westphal/testing/install.sh ID
  ```

  where `ID` is the group ID that you received from your tutor. The script will create the file `testsuite.sh` on the current directory.
- Use your favorite editor to open that file and insert one line per test case at the end of the file using the following format:

  ```
  do_test O B R S P E
  ```

  where:
  - `O` is the number of books owned by the client.
  - `B` is the number of books borrowed by the client.
  - `R` is the number of bad ratings of the client.
  - `S` is a flag for the student status.
  - `P` is the base price of the new book in cents.
  - `E` is the expected result of the test.
- Please use only the characters 0–9, a–z, A–Z, and '-' in your `O`, `B`, `R`, `S`, `P`, `E`; submissions using other characters are not eligible for bonus points.
- You may document your test suite by using *dedicated lines* starting with '`#`' anywhere among your test cases; that is, character '`#`' on a `do_test`-line will count as part of your test case (which would harm eligibility for bonus points (see above)).
- Save the file and submit it with your other exercise solutions.

Running the tests (optional)

Note that Task (i) only requires you to submit your test script. However, if you would like to execute the test script yourself, the binaries for testing are accessible. You may execute them at your discretion. The script will execute your test cases on your team's implementation and report the results.

To run the test script, execute the command

```
./testsuite.sh
```

on the host

```
login.informatik.uni-freiburg.de
```

[2]Note that you cannot use the command-line completion feature; just enter the command as given above.