Chapter 14

Testing Programs

Abstract When programming, chances are that errors or bugs are created. Some are syntactical, which dotnet is very good at finding, but others are semantical, which can be very hard to find. To systematically seek semantical bugs, you will in this chapter learn how to

- test sofware without from the persective of a user or a customer, who has no knowledge about the internal structure of the software, but who is interested in its functionality. This is known as black-box testing
- test software from a perspective of the software developer, with full knowledge and with a focus on every line of code. This is of known as white-box testing

A software bug is an error in a computer program that causes it to produce an incorrect result or behave in an unintended manner. The term 'bug' was used by Thomas Edison in 1878[1,2], but made popular in computer science by Grace Hopper, who found a moth interfering with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 14.1. Software is everywhere, and

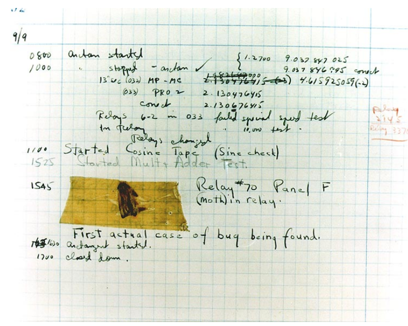

Fig. 14.1 The first computer bug, caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

errors therein have a huge economic impact on our society and can threaten lives[3].

The ISO/IEC organizations have developed standards for software testing[4]. To illustrate basic concepts of software quality, consider a hypothetical route planning system. Essential factors of its quality are:

Functionality: Does the software compile and run without internal errors. Does it solve the problem it was intended to solve? E.g., does the route planning software find a suitable route from point a to b?

Reliability: Does the software work reliably over time? E.g., does the route planning software work when there are internet dropouts?

Usability: Is the software easy and intuitive to use by humans? E.g., is it easy to enter addresses and alternative routes in the software's interface?

Efficiency: How many computer and human resources does the software require? E.g., does it take milliseconds or hours to find a requested route? Can the software run on a mobile platform with limited computer speed and memory?

---

[1] `https://en.wikipedia.org/wiki/Software_bug`

[2] `http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487`

[3] `https://en.wikipedia.org/wiki/List_of_software_bugs`

[4] ISO/IEC 9126, International standard for the evaluation of software quality, December 19, 1991, later replaced by ISO/IEC 25010:2011

Maintainability: In case of the discovery of new bugs, is it easy to test and correct the software? Is it easy to extend the software with new functionality? E.g., is it easy to update the map with updated roadmaps and new information? Can the system be improved to work both for car drivers and bicyclists?

Portability: Is it easy to port the software to new systems such as new server architecture and screen sizes? E.g., if the routing software originally was written for IOS devices, will it be easy to port to Android systems?

The above-mentioned concepts are ordered based on the requirements of the system. Functionality and reliability are perhaps the most important concepts, since if the software does not solve the specified problem, then the software design process has failed. However, many times the problem definition will evolve along with the software development process. But as a bare minimum, the software should run without internal errors and not crash under a well-defined set of circumstances. Furthermore, it is often the case that software designed for the general public requires a lot of attention to the usability of the software, since in many cases non-experts are expected to be able to use the software with little or no prior training. On the other hand, software used internally in companies will be used by a small number of people who become experts in using the software, and it is often less important that the software is easy to understand by non-experts. An example is text processing software like Microsoft Word versus Gnu Emacs and LaTeX. Word is designed to be used by non-experts for small documents such as letters and notes and relies heavily on interfacing with the system using click-interaction. On the other hand, Emacs and LaTeX are for experts for longer and professionally typeset documents and relies heavily on keyboard shortcuts and text-codes for typesetting document entities.

The purpose of *software testing* is to find bugs. When errors are found, then we engage in *debugging*, which is the process of diagnosing and correcting bugs. Once we have a failed software test, i.e., one that does not find any bugs, then we have strengthened our belief in the software, but it is important to note that software testing and debugging rarely removes all bugs, and with each correction or change of software there is a fair risk new bugs being introduced. It is not exceptional that the testing-software is as large as the software being tested.

In this chapter, we will focus on two approaches to software testing which emphasize functionality: *white-box* and *black-box testing*. An important concept in this context is *unit testing*, where the program is considered in smaller pieces, called units, and for which accompanying programs for testing can be made which test these units automatically. Black-box testing considers the problem formulation and the program interface, and can typically be written early in the software design phase. In contrast, white-box testing considers the program text, and thus requires the program to be available. Thus, there is a tendency for black-box test programs to be more stable, while white-box testing typically is developed incrementally alongside the software development.

To illustrate software testing, we'll start with a problem:

Problem 14.1

iven any date in the Gregorian calendar, calculate the day of the week.

Facts about dates in the Gregorian calendar are:

- Combinations of dates and weekdays repeat themselves every 400 years.
- The typical length of the months January, February, . . . follow the knuckle rule, i.e., January belongs to the index knuckle, February to the space between the index and the middle finger, and August restarts or starts on the other hand. All knuckle months have 31 days, all spacing months have 30 days except February, which has 29 days on leap years and 28 days all other years.
- A leap year is a multiple of 4, except if it is also a multiple of 100 but not of 400.

Many solutions to the problem have been discovered, and here we will base our program on Gauss' method, which is based on integer division and calculates the weekday of the 1st of January of a given year. For any other date, we will count our way through the weeks from the previous 1st of January. The algorithm relies on an enumeration of weekdays starting with Sunday = 0, Monday = 1, . . . , and Saturday = 6. Our proposed solution is shown in Listing 14.1. Note that this problem has been chosen such that the solution is complicated which is a typical testing scenario, where the inner workings of the code is non-triveal.

14.1 Black-box Testing

In black-box testing, the program is considered a black box, and no knowledge is required about how a particular problem is solved. In fact, it is often useful not to have that knowledge at all. It is rarely possible to test all input to a program, so in black-box testing, the solution is tested for typical and extreme cases based on knowledge of the problem. The procedure is as follows:

1. Decide on the interface to use: It is useful to have an agreement with the software developers about what interface is to be used, e.g., in our case, the software developer has made a function `date2Day d m y` where `d`, `m`, and `y` are integers specifying the day, month, and year.
2. Make an overall description of the tests to be performed and their purpose:

Listing 14.1 date2Day.fsx:
A function that can calculate day-of-week from any date in the Gregorian calendar.

```
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

let rec sum (lst : int list) j =
  if 0 <= j && j < lst.Length then
    lst[0] + sum lst[1..] (j - 1)
  else
    0

let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then 29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix[weekday] + "day"
```

1. a consecutive week, to ensure that all weekdays are properly returned
2. two set of consecutive days across boundaries that may cause problems: across a new year, and across a regular month boundary.
3. a set of consecutive days across February-March boundaries for a leap and non-leap year
4. four dates after February in a non-leap year, a non-multiple-of-100 leap year, a multiple-of-100-but-not-of-400 non-leap year, and a multiple-of-400 leap year.

Given no information about the program's text, there are other dates that one could consider as likely candidates for errors, but the above is judged to be a fair coverage.

3. Choose a specific set of input and expected output relations on the tabular form as shown in Table 14.1.

4. Write a program executing the tests, as shown in Listing 14.2 and 14.3. Notice how the program has been made such that it is almost a direct copy of the table produced in the previous step.

Test number	Input	Expected output
1a	1 1 2016	Friday
1b	2 1 2016	Saturday
1c	3 1 2016	Sunday
1d	4 1 2016	Monday
1e	5 1 2016	Tuesday
1f	6 1 2016	Wednesday
1g	7 1 2016	Thursday
2a	31 12 2014	Wednesday
2b	1 1 2015	Thursday
2c	30 9 2017	Saturday
2d	1 10 2017	Sunday
3a	28 2 2016	Sunday
3b	29 2 2016	Monday
3c	1 3 2016	Tuesday
3d	28 2 2017	Tuesday
3e	1 3 2017	Wednesday
4a	1 3 2015	Sunday
4b	1 3 2012	Thursday
4c	1 3 2000	Wednesday
4d	1 3 2100	Monday

Table 14.1 Black-box testing

A black-box test is a statement of what a solution should fulfill for a given problem. ⋆ Hence, **it is a good idea to make a black-box test early in the software design phase, in order to clarify the requirements for the code to be developed and take an outside view of the code prior to developing it.**

After the black-box testing has failed to find errors in the program, we have some confidence in the program, since, from a user's perspective, the program produces sensible output in many cases. It is, however, in no way a guarantee that the program is error free.

14.2 White-box Testing

White-box testing considers the text of a program. The degree to which the text of the program is covered in the test is called the *coverage*. Since our program is small, we have the opportunity to ensure that all functions are called at least once, which is called *function coverage*, and we will also be able to test every branching in the program, which is called *branching coverage*. If both are fulfilled, we say that we have *statement coverage*. The procedure is as follows:

1. Decide which units to test: The program shown in Listing 14.1 has 3 functions, and we will consider these each as a unit, but we might as well just have chosen

Listing 14.2 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from Listing 14.4 has been omitted for brevity.

```
let testCases = [
  ("A complete week",
   [(1, 1, 2016, "Friday");
    (2, 1, 2016, "Saturday");
    (3, 1, 2016, "Sunday");
    (4, 1, 2016, "Monday");
    (5, 1, 2016, "Tuesday");
    (6, 1, 2016, "Wednesday");
    (7, 1, 2016, "Thursday");]);
  ("Across boundaries",
   [(31, 12, 2014, "Wednesday");
    (1, 1, 2015, "Thursday");
    (30, 9, 2017, "Saturday");
    (1, 10, 2017, "Sunday")]);
  ("Across Feburary boundary",
   [(28, 2, 2016, "Sunday");
    (29, 2, 2016, "Monday");
    (1, 3, 2016, "Tuesday");
    (28, 2, 2017, "Tuesday");
    (1, 3, 2017, "Wednesday")]);
  ("Leap years",
   [(1, 3, 2015, "Sunday");
    (1, 3, 2012, "Thursday");
    (1, 3, 2000, "Wednesday");
    (1, 3, 2100, "Monday")]);
  ]

printfn "Black-box testing of date2Day.fsx"
for i = 0 to testCases.Length - 1 do
  let (setName, testSet) = testCases[i]
  printfn "  %d. %s" (i+1) setName
  for j = 0 to testSet.Length - 1 do
    let (d, m, y, expected) = testSet[j]
    let day = date2Day d m y
    printfn "    test %d - %b" (j+1) (day = expected)
```

`date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the latter two is superfluous. However, we may have to do this anyway when debugging, and we may choose at a later point to use these functions separately, and in both cases, we will be able to reuse the testing of the smaller units.

2. Identify branching points: The function `januaryFirstDay` has no branching function, `sum` has one, and depending on the input values, two paths through the code may be used, and `date2Day` has one where the number of days in February is decided. Note that in order to test this, our test-date must be March 1 or later.

Listing 14.3: Output from Listing 14.2.

```
$ dotnet fsi date2DayBlackTest.fsx
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across Feburary boundary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

In this example, there are only examples of `if`-branch points, but they may as well be loops and pattern matching expressions. In the Listing 14.4 it is shown that the branch points have been given a comment and a number.

3. For each unit, produce an input set that tests each branch: In our example, the branch points depend on a Boolean expression, and for good measure, we are going to test each term that can lead to branching. Using 't' and 'f' for `true` and `false`, we thus write as shown in Table 14.2. The impossible cases have been intentionally blank, e.g., it is not possible for $j < 0$ and $j > n$ for some positive value n.

4. Write a program that tests all these cases and checks the output, see Listing 14.5.

Notice that the output of the tests is organized such that they are enumerated per unit, hence we can rearrange as we like and still uniquely refer to a unit's test. Also, the output of the test program produces a list of tests that should return true or success or a similar positively loaded word, but without further or only little detail, such that we at a glance can identify any test that produced unexpected results.

Listing 14.4 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
// Unit: januaryFirstDay
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

// Unit: sum
let rec sum (lst : int list) j =
  (* WB: 1 *)
  if 0 <= j && j < lst.Length then
    lst[0] + sum lst[1..] (j - 1)
  else
    0

// Unit: date2Day
let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  (* WB: 1 *)
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then 29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix[weekday] + "day"
```

After the white-box testing has failed to find errors in the program, we have some confidence in the program, since we have run every line at least once. It is, however, in no way a guarantee that the program is error free, which is why white-box testing is often accompanied with black-box testing to be described next.

14.3 Key Concepts and Terms in This Chapter

In this chapter, we have considered two approaches to debugging code. You have seen how to:

- write supporting code for **black-box testing**, which tests for errors focussing on the intended functionality of the code and ignoring how it is constructed

<table>
<tr><th>Unit</th><th>Branch</th><th>Condition</th><th>Input</th><th>Expected output</th></tr>
<tr><td>januaryFirstDay</td><td>0</td><td>-</td><td>2016</td><td>5</td></tr>
<tr><td>sum</td><td>1</td><td>0 <= j &&
j < lst.Length</td><td></td><td></td></tr>
<tr><td></td><td>1a</td><td>t && t</td><td>[1; 2; 3] 1</td><td>3</td></tr>
<tr><td></td><td>1b</td><td>f && t</td><td>[1; 2; 3] -1</td><td>0</td></tr>
<tr><td></td><td>1c</td><td>t && f</td><td>[1; 2; 3] 10</td><td>0</td></tr>
<tr><td></td><td>1d</td><td>f && f</td><td>-</td><td>-</td></tr>
<tr><td>date2Day</td><td>1</td><td>(y % 4 = 0) &&
((y % 100 <> 0)
||
(y % 400 = 0))</td><td></td><td></td></tr>
<tr><td></td><td>-</td><td>t && (t || t)</td><td>-</td><td>-</td></tr>
<tr><td></td><td>1a</td><td>t && (t || f)</td><td>8 9 2016</td><td>Thursday</td></tr>
<tr><td></td><td>1b</td><td>t && (f || t)</td><td>8 9 2000</td><td>Friday</td></tr>
<tr><td></td><td>1c</td><td>t && (f || f)</td><td>8 9 2100</td><td>Wednesday</td></tr>
<tr><td></td><td>-</td><td>f && (t || t)</td><td>-</td><td>-</td></tr>
<tr><td></td><td>1d</td><td>f && (t || f)</td><td>8 9 2015</td><td>Tuesday</td></tr>
<tr><td></td><td>-</td><td>f && (f || t)</td><td>-</td><td>-</td></tr>
<tr><td></td><td>-</td><td>f && (f || f)</td><td>-</td><td>-</td></tr>
</table>

Table 14.2 Unit test

- write a **white-box test**, which in its simplest form ensures that every line of code has been run at least once

- structure a white-box test in terms of **unit**, which is why this is sometimes called **unit testing**.

Listing 14.5 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from Listing 14.4 has been omitted for brevity.

```
printfn "White-box testing of date2Day.fsx"
printfn "  Unit: januaryFirstDay"
printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)

printfn "  Unit: sum"
printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)

printfn "  Unit: date2Day"
printfn "    Branch: 1a - %b" (date2Day 8 9 2016 =
   "Thursday")
printfn "    Branch: 1b - %b" (date2Day 8 9 2000 =
   "Friday")
printfn "    Branch: 1c - %b" (date2Day 8 9 2100 =
   "Wednesday")
printfn "    Branch: 1d - %b" (date2Day 8 9 2015 =
   "Tuesday")
```

```
$ dotnet fsi date2DayWhiteTest.fsx
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```