Testing und Debugging: Übung 02

Abgabetermin: siehe Canvas Kurs

1 Definitions- und Wertebereich

In der Vorlesung haben Sie einige generelle Hinweise und Kriterien erhalten, wie man gute Eingabewerte von Funktionen auswählt um Testfälle zu erstellen. Wenden Sie nun dieses Wissen an:

1.1 Luhns Algorithmus (40 Punkte)

Gegeben sei der folgender Code:

Kreditkarten Prüfung (luhn.py)

```
def luhn_digit(digit):
    assert isinstance(digit, int)

    digit = digit * 2
    if digit > 9:
        return digit - 9
    else:
        return digit


def luhn_checksum(digit_string):
    assert isinstance(digit_string, str)
    assert digit_string.isdigit()

    l = len(digit_string)
    chksum = 0
    if l % 2 == 0:
        for i in range(l):
            if (i + 1) % 2 == 0:
                chksum += int(digit_string[i])
            else:
                chksum += luhn_digit(int(digit_string[i]))
    else:
        for i in range(l):
            if (i + 1) % 2 == 0:
                chksum += luhn_digit(int(digit_string[i]))
            else:
                chksum += int(digit_string[i])
    return chksum % 10
```

Aufgaben

1. *Wieviele Werte für* digit_string *benötigt man mindestes um die Funktionen vollständig zu testen? Begründen Sie Ihre Antwort.(15 Punkt)*

   ```
   .
   .
   ```

2. *Geben Sie die Werte für* digit_string *aus der vorangegangene Aufgabe explizit an. (15 Punkte)*

.
.

3. *Muss man beim Testen von `luhn_checksum` mit versteckten Eingaben rechnen? Wenn ja, wie muss dann exemplarisch ein Testfall aussehen, der dem Rechnung trägt? Wenn nein, begründen Sie kurz diese Antwort. (10 Punkte)*

.
.
.
.
.
.
.

1.2 GUI (Graphical User Interface) Testing (50 Punkte)

Beim Testen einer Funktion oder eines Stücks Software gilt es sich erst einmal darüber im Klaren zu sein, aus welchen Elementen der Definitions- und Wertebereich besteht um entsprechende Eingabewerte zu erstellen und Prüfroutinen für die Ausgabe zu erstellen.

Zum Beispiel hat die Funktion `int testme(int a, char *buf)` als Definitionsbereich Vektoren deren erste Komponente ein Integer ist und deren zweite Komponente ein Zeiger auf einen String ist. Als Wertebereich sind die gesamten Integerzahlen angegeben.

Wie sieht das aber bei Testen einer GUI Applikation aus?

Aufgaben

1. *Aus welchen Elementen besteht der Definitionsbereich beim Testen einer GUI Applikation (10 Punkte)*

.
.

2. *Aus welchen Elementen besteht der Wertebereich? (10 Punkte)*

.
.

3. *Eine mögliche Sammlung von Tests für eine GUI Applikation währe es, die Applikation gemäß ihrer Spezifikation, für 5 Minuten zu Benutzen (z.B. wenn Sie OpenOffice testen, könnten Sie einfach einen kurzen Brief schreiben und diesen dann formatieren). Nennen Sie zwei weitere Möglichkeiten für Sammlungen von Tests, um eine GUI Applikation zu testen. (je 5 Punkte)*

.
.
.
.
.
.
.

4. *Muss man beim Testen von GUI Applikationen mit versteckten Eingaben rechnen? Wenn ja, wie muss dann exemplarisch ein Testfall aussehen, der dem Rechnung trägt? Wenn nein, begründen Sie kurz Ihre Antwort. (10 Punkte)*

.
.
.
.
.
.
.

5. *Wie gründlich müssen Graphical User Interfaces im Allgemeinen getestet werden? Begründen Sie Ihre Antwort kurz. (10 Punkte)*

.
.
.
.
.
.
.

2 Programmieraufgabe: Fault Injection (100 Punkte)

Wenn man ein Stück Software testet kann es vorkommen, das einige, der benutzten Funktionen in besonderen Ausnahmefällen ein spezielles Verhalten haben, welches in der zu testenden Software natürlich beachtet werden muss. Beim Testen dieser Software steht der Tester nun häufig vor dem Problem, dass er diese Ausnahmefälle nicht beliebig wiederholbar auslösen kann — er kann also bestimmte Teile des Codes nicht zuverlässig triggern um herauszufinden, ob diese Teile auch genau das Machen, was sie sollen.

Um auch diese Teile des Codes zu testen, bleibt einem oft nichts anderes übrig, als um die fragliche Funktion eine Wrapperfunktion herum zu schreiben, welche dieses Ausnahmeverhalten zuverlässig simuliert. Während der Tests ersetzt man im fraglichen Codestück diese Funktion durch die Wrapperfunktion und kann so auch den Teil des Codes testen, der im Ausnahmefall das spezielle Verhalten der Funktion behandelt. Viele Compiler besitzen zu diesem Zweck spezielle Schalter, mit deren Hilfe die Wrapperfunktionen statt der originalen Funktionen aufgerufen werden, ohne dass Änderungen am Quellcode notwendig sind.

Tipp
Um in der Praxis ein zügiges Testen zu gewährleisten, sollte die Wrapperfunktion mit einer Wahrscheinlichkeit von ca 99 % beim Aufrufen, das normale Verhalten der Funktion an den Tag legen und nur in 1 % der Fälle das Verhalten aus dem Ausnahmefall zeigen.

Dateien für die Programmieraufgabe

run_test.py	Datei testet Ihre Implementierung von `read_all()` in *uebung02.c*
Grading/*init*py	Hinweisdatei für Python: Das Verzeichnis ist ein Modul
Grading/Grading.py	Quellcode der Testklasse, die in *run_test.py* verwendet wird
uebung02.c *	In dieser Datei müssen Sie die Funktion `read_all()` gemäß Aufgabenstellung implementieren
test.txt	Beispieltext, der für die Tests in 'run_test.py verwendet wird

Die mit [*] markierten Dateien müssen **bearbeitet** und **abgegeben** werden.

Manpage der Linux Funktion read:

```
READ(2)                         Linux Programmer's Manual                        READ(2)



NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  attempts to read up to count bytes from file descriptor fd into the buf-
       fer starting at buf.

       If count is zero, read() returns zero and has no other  results.   If  count  is
       greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE
       On  success,  the number of bytes read is returned (zero indicates end of file),
       and the file position is advanced by this number.  It is not an  error  if  this
       number  is smaller than the number of bytes requested; this may happen for exam-
       ple because fewer bytes are actually available right now (maybe because we  were
       close  to  end-of-file,  or because we are reading from a pipe, or from a termi-
       nal), or because read() was interrupted by a signal.  On error, -1 is  returned,
       and errno is set appropriately.  In this case it is left unspecified whether the
       file position (if any) changes.
```

Ihre Aufgabe

Wie Sie in der oben gedruckten Manpage sehen können, kann die Funktion `read` unter Umständen weniger als die gegeben Anzahl von Bytes lesen.

In der Datei *uebung02.c* wurde bereits eine Wrapperfunktion für `read` mit dem Namen `__wrap_read` erstellt. Benutzen Sie beim Schreiben der Funktion `read_all` nur den Aufruf `read`. Beim Kompilieren können Sie dann entscheiden ob Sie die originale Funktion oder die Wrapperfunktion verwenden wollen — Sie müssen einfach nur den Schalter *-Wl,--wrap=read* an den *gcc* Aufruf anhängen um die Wrapperfunktion zu wählen. Wenn Sie den Schalter weglassen, so wird die originale Funktion verwendet.

Ihre Aufgabe ist es nun, die Funktion `read_all` zu implementieren, die nach einmaligem Aufruf garantiert die angegebene Anzahl Bytes in das Array einließt. Hierfür wird die Funktion `read` u.U. mehrfach mit angepassten Parametern innerhalb von `read_all` aufgerufen. Ist die einzulesende Datei kürzer als die angegebene Anzahl zu lesender Bytes wird natürlich nur die Datei bis zu ihrem Ende eingelesen.

Die Funktion gibt bei Erfolg die Anzahl der gelesenen Bytes zurück. Sie gibt -1 zurück, falls intern die Funktion `read` -1 zurück gibt.

Kompiliert wird die Datei mit folgendem Kommando:

```
gcc -Wall -Wl,--wrap=read uebung02.c -o uebung02
```

Ausgeführt wird das Programm mit der zu lesenden Textdatei und der Anzahl der Zeichen als Argument. Z. B.:

```
./uebung02 test.txt 21
```

Dieser Aufruf sollte eine Sequenz von Zahlen liefern, die sich zu 21 aufsummieren (Ausgegeben nach stderr) und den ersten 21 Zeichen der Datei *test.txt*.

Beispielsweise:

```
2
16
1
2
Dies ist ein Text zum
```

Das Skript *run_test.py* sorgt dafür, dass Ihr kompiliertes Programm mit 9 verschieden Argumenten aufgerufen und getestet wird. Ausgabe des Testskripts entspricht in Form in etwa der von Übung 01.

Der Test wird mit folgendem Kommando aufgerufen:

```
python3 run_test.py uebung02.c
```

Beachten Sie, dass Sie die Quelldatei und nicht das kompilierte Programm als Argument mitgeben und dass diese mit dem obigen Kommando kompiliert wurde.

Ist der erste Test fehlerfrei, so erhalten Sie für diesen und jeden weiteren Test, den das Testskript als bestanden anzeigt, 10 Punkte.

Viel Erfolg