Freie Universität Berlin

Lutz Prechelt

Softwaretechnik, SoSe21

Übung 10

TutorIn: Samuel Domiks
Tutorium 02
Materialien: Latex, Skript

Jonny Lam & Thore Brehmer

20. Juni 2021

---

1 Begriffe

(a) *Erklären Sie den Unterschied zwischen Verifizieren und Validieren.*

- **Verifizieren [1]:** Bei der Verifizierung wird überprüft ob eine Software seiner Spezifikation entspricht.
  So können wir überprüfen, ob die zuvor gesetzten Ziele mit den erreichten Zielen erreicht werden können.
  Mit Hilfe der Verifizierung können wir Fehler und Probleme in einem Entwicklungsprozess frühzeitig erkennen und so auch die Kosten reduzieren.
  Mit Hilfe der Verifizierung können wir jedoch keine Fehler in der Spezifikation feststellen.

- **Validieren [2]:** Erst nach der Verifizierung folgt die Validierung.
  Bei der Validierung wird überprüft ob die Software auch wirklich das erreicht, was gefordert wird.
  Das passiert meistens im Rahmen eines Feldversuchs, wo mehrere Anwender/Nutzer der Software sich mit dem Produkt auseinander setzt.
  Der Nutzer soll das Produkt nach der Zielerreichung untersuchen und unter verschiedene Rahmenbedingungen.

Quellen:

[1] "Verifizierung", https://de.wikipedia.org/wiki/Verifizierung . [aufgerufen am 20.06.2021]

[2] "Validierung", https://de.wikipedia.org/wiki/Validierung_(Informatik) [aufgerufen am 20.06.2021]

(b) *Woraus besteht ein Testfall? Wann nennt man einen Testfall erfolgreich und was ist der Gedanke dahinter?*

- Ein Testfall besteht aus dem **Systemzustand**, der **Eingaben** und dem (erwarteten) **Systemverhalten**

- Wenn wir von einen Defekttest sprechen, ist der Testfall **erfolgreich**, sofern eine Abweichung vom erwarteten Systemverhalten und den wirklichen Rückgabewert vorliegt.
- Der Testfall ist genau deswegen erfolgreich, weil wir diese Versagen (failures) finden wollen.

(c) *Wie hängen Versagen, Fehler und Defekt zusammen?*

- **Versagen** ist ein falsches Verhalten des Programmes. Das falsche Verhalten kommt durch einen **Defekt** im Program. Der defekt entsteht durch **Fehler** von Softwareentwicklern. (Falschtun oder Versäumnis)
- In Kurz: Ein Versagen entsteht durch ein Defekt. Ein Defekt entsteht durch einen Fehler.

(d) *Erklären Sie jeweils die Gemeinsamkeiten und Unterschiede zwischen*

1. Strukturtest und Durchsicht
   - In **Strukturtests** wird das Programm dynamisch geprüft während in **Durchsicht** das Program manuell statisch geprüft wird.
2. Lasttest und Stresstest
   - Ein **Lasttest** wird das Systemverhalten bei einer realistischen Last beobachtet (z.B. viele Benutzer), während beim **Stresstest** das Systemverhalten bei einer unrealistischen Last (Überlastung) beobachtet wird. (z.B. sehr viele unrealistische Eingaben)
3. Testen und Debugging
   - Beim **Testen** versucht man Versagen(failures) im Programm zu finden, während man beim **Debugging** versucht die Ursache des Versagens zu finden und diesen zu beheben.
4. Funktionstest und Akzeptanztest
   - Mit **Funktionstest** werden Testfälle genutzt um Versagen zu finden, während man mit **Akzeptanztests** zeigen möchte dass das Produkt nun nutzbar (tauglich ist).
5. Top-Down-Testen und Bottom-up-Testen
   - **Top-Down** und **Bottom-up** sind beides Verfahren die einen eine Vorlage bieten, welche Teile des Systems man als erstes testet.
   - Dabei fängt man bei Bottom-up mit den untersten Teilen an (da sie keine anderen Teile als vorraussetzung haben) und bei Top-Down mit dem Hauptprogramm. (da für sie keine Treiber nötig sind, weil alles, was sie aufruft, schon verfügbar und getestet ist)

2 Testtechniken anwenden und bewerten

(a) *Spezifizieren Sie die Vorbedingung für die Operation klassifiziereDreieck in OCL.*

```
context klassifiziereDreieck(seite1: int, seite2: int, seite3: int)
    pre:
        #Keine negative Seitenlängen
        seite1 > 0 and seite2 > 0 and seite3 > 0 and
        #Zwei Seiten zusammen müssen größer sein als die dritte
        #sonste => kein Dreieck
        (
        seite1 + seite2 > seite3 and
        seite1 + seite3 > seite2 and
        seite2 + seite3 > seite1
        )
```

(b) *Black-Box: Erstellen Sie mindestens sieben Testfälle allein aufgrund der Schnittstellenbeschreibung der Operation durch Betrachtung unterschiedlicher Fälle, die sich darin unterscheiden, dass Sie jeweils ein anderes Verhalten des Systems erwarten; vermeiden Sie unnötige Dopplungen (Stichworte: Äquivalenzklassen und Randfälle). Beschränken Sie sich nicht nur auf gültige Eingaben (Stichwort: Fehlerfälle). Listen Sie die Testfälle tabellarisch auf. Beschreiben Sie bei jedem Testfall kurz, warum Sie diesen in die Menge der Testfälle aufgenommen haben.*

Nr.	Äquivalenzklasse	Beschreibung	Testfall (& Erwarteter Rückgabewert)
1	Eingabewerte	Negative Zahl	(-1,3,5), Error
2	Eingabewerte	Null	(0,3,5), Error
3	Art des Dreiecks	Gleichseitig	(1,1,1), Gleichseitig
4	Art des Dreiecks	Rechtwinklig	$(1,1,\sqrt[2]{2})$, Rechtwinklig
5	Art des Dreiecks	Gleichschenklig	(1,2,2), Gleichschenklig
6	Art des Dreiecks	Normales Dreieck	(2,3,4), Normal
7	Art des Dreiecks	kein Dreieck	(1,1,2), Error
8	Größe der Eingabewerte	sehr groß	(MAX_VALUE,2,MAX_VALUE-1), Normal

(c) *White-Box: Erstellen Sie nun Testfälle aufgrund der Struktur des Programms, mit dem Ziel, Zweigüberdeckung (C1, engl. branch coverage) zu erreichen, d.h. dass an jeder Verzweigung im Programm beide Äste ausgeführt wurden. Auf der zweiten Seite dieses Übungsblattes finden Sie eine Implementierung für klassifiziereDreieck in Java: Da für eine vollständige Zweigüberdeckung jede Steuerbedingung mindestens einmal war und einmal falsch werden muss, bestimmen Sie zunächst diejenigen Stellen im Programm, an denen es Verzweigungen gibt und listen Sie diese auf. Suchen Sie nun nach Testfällen, sodass jeder Ast im Programm einmal aktiv wird. Erstellen Sie auch hier eine tabellarische Übersicht der Testfälle und geben Sie jeweils an, warum Sie diesen Testfall in die Menge aufgenommen haben.*

Nr.	branch	Beschreibung	Testfall (& Erwarteter Rückgabewert)
1	Zeile:5 if	um den Ast zu erreichen	(1,2,2), Gleichschenklig
2	Zeile:10 if	um den Ast zu erreichen	(1,1,1), Gleichseitig
3	Zeile:18 if	um den Ast zu erreichen	$(1,1,\sqrt[2]{2})$, Rechtwinklig
4	Zeile:22 (else)	um den Ast zu erreichen	(2,3,4), Normal

(d) *Fügen Sie all die in b) und c) entstandenen Testfälle zusammen zu einer Testfallmenge und führen Sie die Testfälle gedanklich aus. Welche Ihrer Testfälle sind erfolgreich? Beschreiben Sie etwaige Defekte, die Sie auf diesem Wege im Programm gefunden haben.*

Nr	Testfall (& Erwarteter Rückgabewert)	Rückgabewert	Erfolgreich
1	(-1,3,5), Error	Error (OCL)	Ja (mit OCL)
2	(0,3,5), Error	Error (OCL)	Ja (mit OCL)
3	(1,1,1), Gleichseitig	Gleichschenklig	Nein
4	$(1,1,\sqrt[2]{2})$, Rechtwinklig	Gleichschenklig	Nein
5	(1,2,2), Gleichschenklig	Gleichschenklig	Ja
6	(2,3,4), Normal	Normal	Ja
7	(1,1,2), Error	Error (OCL)	Ja (mit OCL)
8	(MAX_VALUE,2,MAX_VALUE-1), Normal	Normal	Ja

(e) *Haben Sie eine Anweisungsüberdeckung (engl. statement coverage, C0) erreicht? Wie geeignet erscheinen Ihnen die beiden genannten Überdeckungskriterien C0 und C1?*

- In diesem Fall ist das benutzen von C_0 und C_1 zum Testen genau gleich, da C_0 auch verlangt hätte, dass wir in alle Zweige gehen. (Da alle Anweisungen einmal ausgeführt werden müssen)
- Deshalb haben wir die Anweisungsüberdeckung auch nicht erreicht, da die Anweisung der if Bedienung in Zeile 10 nie ausgeführt werden kann.
- Uns scheinen die beiden Überdeckungskritierein C_0 und C_1 als geeignet, da sie genau die gleichen Versagen wie die Black-Box gefunden haben. (Und das sogar mit weniger Testfällen)

(f) *Wie geeignet erscheinen Ihnen die beiden angewandten Testfallerzeugungsverfahren Funktionstest und Strukturtest?*

- In diesem Fall sind beide Testfallerzeugunsverfahren gleich geeignet, da wir für beide die gleichen Versagen gefunden haben.
- Wir bevorzugen etwas den Strukturtest (White box), da wir diesen schneller durchführen konnten.
- Der Funktionstest (Black box) hat dafür etwas mehr Zeit in anspruch genommen, jedoch hat sich dadurch auch unser Verständnis der Spezifikation verbessert.

(g) *Gibt es weitere Versagensmöglichkeiten, die die Testfälle nicht aufgedeckt haben? Wie haben Sie diese entdeckt? Wie nennt sich diese „Entdeckungstechnik“?*

- Unser Testfall 8 könnte trotzdem Fehlerhaft ausgeführt werden, auch wenn wir ein richtiges Ergebnis bekommen haben. Im Code wird nämlich mit dem Input gerechnet(Addition und Multiplikation). Bei großen Zahlen (also bei diesem Testfall) kann es dadurch zu einen Overflow kommen. Entdeckungstechnik: Allgemeine Erfahrung, Intuition oder durch Durchsicht.

3 Eigenen Code testen

(a) *Eine der „goldenen“ Regeln im Bereich der Softwareentwicklung lautet: „Ein/e Programmierer/in sollte nicht den eigenen Code testen.“ Mit welchen Problem müsste man rechnen, wenn man sich nicht an diese Regel hält? Erläutern Sie zwei verschiedene Probleme. Diskutieren Sie: Ist diese Regel sinnvoll?*

- **Problem 1:** Der Programmierer erschafft einen Code und testet den nur mit bestimmten Beispielen, wo er denkt, dass es die einzigen Beispiele sind. Dabei kann es vorkommen, dass andere auf andere Beispiele stoßen können, die zu Grenzen an seinem Code führen können. Oder auch triviale Sachen werden nach einer Änderung nicht getestet und am Anfang kann das sein, dass diese triviale Sachen funktionieren, aber später plötzlich nicht mehr.
- **Problem 2:** Es kann vorkommen, dass am Code alles richtig ist, doch die Grundlage ist nicht richtig. Der Programmierer kann ein falsches Verständnis der Anforderungen haben und sein Code baut darauf auf. Der Code ist richtig, aber das Ziel hat er verfehlt.
- **Problem 3:** Der Code, den der Programmierer entwickelt, testet er nur isoliert. Später wird der Code wahrscheinlich ein Teil des Systems sein. Auch da könnte es zu Schwierigkeiten kommen und sollte vorher von einer anderen Person getestet werden.

(b) *Recherchieren und erklären Sie, was man unter Test-Driven Development (TDD) versteht. Erläutern Sie, wie TDD bei der in a) angesprochenen Problematik hilft.*

- Beim TDD werden Tests benutzt um den Softwarezyklus zu steuern. Dabei wird erstmal ein Test geschrieben, der nicht funktioniert, da der Code noch nicht vorhanden ist.
- Danach wird der Code geschrieben, bis der Test erfolgreich abläuft. Als nächstes folgt das Refactoring, hierbei wird der Code und Test gleichermaßen aufgeräumt. Nach dem Refactoring fängt man wieder von vorne an.
- Die Vorteile liegen auf der Hand, der Code wird immer getestet, das bedeutet, dass triviale Sachen jeder Zeit getestet werden (Problem 1). Der Code ist sauber, das bedeutet, dass auch andere drüber gucken können und man selber kommt auch nicht durch den sogenannten SSpaghetti-Code”durcheinander.
- So kann man sich auch auf das wesentliche konzentrieren. Ein weiterer Vorteil ist auch, dass jemand anderes diesen Test schreiben kann z.B. jemand der die Anforderungen verstanden hat (Problem 2).

Quellen:

"Was ist testgetriebene Entwicklung?", https://www.it-agile.de/agiles-wissen/agile-entwicklung/was-ist-testgetriebene-entwicklung/#:~:text=Bei%20der%20testgetriebenen%20Entwicklung%20(engl,wird%20geschrieben%2C%20der%20zunächst%20fehlschlägt. [aufgerufen am 20.06.2021]