7. Verifikation und Validierung

Verifikation und Validierung (I)

Ziele:

Finden von Fehlern,

Beurteilung der Benutzbarkeit.

Validierung:

Bauen wir das richtige Produkt?

Entspricht das Programm den Erwartungen des Auftraggebers?

Verifikation:

Bauen wir das Produkt richtig?

Entspricht das Programm seiner Spezifikation?

Verifikation und Validierung (II)

Dynamisch:

Test der laufenden Implementierung,

während/nach der Implementierungsphase,

Verifikation *und* Validierung.

Statisch:

Untersuchung der Programmdokumente (Anforderungsspezifikation, Entwurf, Programmcode),

z. B. Programminspektion, formale Verifikation,

in allen Phasen der Programmentwicklung,

nur Verifikation.

Testen (I)

Durch Testen kann man **nur die Existenz** von Fehlern nachweisen.

(Wenn man beim Testen keine Fehler findet, dann bedeutet das nicht, daß das Programm keine Fehler hat, sondern wahrscheinlich, daß der Test das Programm nicht ausreichend gefordert hat.)

Testen (II)

Testen:

Nachweis der Existenz von Fehlern.

Debuggen:

Finden von Fehlern (durch Tracen des Codes, zusätzliche Testbeispiele, Vergleich mit bekannten Fehlern).

Häufig Implementierer $\neq$ Tester:

Vermeidung persönlicher Anhänglichkeit.

Testen (III)

Testen:

in kleinen Einheiten,

Komponententest $\rightarrow$ Integrationstest $\rightarrow$ Akzeptanztest.

Fehlerkorrektur in einer späteren Phase macht häufig Wiederholung der Tests der früheren Phasen notwendig,

$\rightsquigarrow$ sehr aufwendig,

$\rightsquigarrow$ Abhängigkeiten feststellen.

Testen ist teuer:

für Realzeitsysteme bis zu 50% der Entwicklungskosten.

Top-down und Bottom-up

Bottom-up-Testen:

Untere Komponenten zuerst testen,

$\rightsquigarrow$ Testumgebung erforderlich,

getestete Unterkomponenten für Test der höheren Komponenten verfügbar,

Validierung erst spät möglich.

Top-down-Testen:

Höhere Komponenten zuerst testen,

dabei Unterkomponenten durch Stubs (Dummys) simulieren,

wie Unterkomponenten simulieren? (zufällige Werte liefern, Tester fragen?)

Validierung kann früh beginnen,

Entwurfsfehler werden früh gefunden (vermeidet Reimplementierung).

Defekt-Testen

Idealerweise: vollständig

geht nicht: Anzahl der zu untersuchenden Fälle zu groß oder sogar unendlich.

$\rightsquigarrow$ Testfälle müssen ausgewählt werden:

Fehler wiegen unterschiedlich schwer (unaufgeräumter Bildschirm, totaler Datenverlust),

typische Situationen wichtiger als Grenzfälle,

aber: Grenzfälle besonders fehleranfällig.

Funktionales Testen

Funktionales Testen:

- ohne Kenntnis der Implementierung (Black-Box),
- allein Spezifikation bestimmt Auswahl der Testdaten.
- Menge der möglichen Eingaben $\rightarrow$ Äquivalenzklassen.

Strukturelles Testen

Strukturelles Testen:

- Kenntnisse über Implementierung beeinflussen Auswahl der Testdaten,
- Äquivalenzklassen gegenüber funktionalem Testen verfeinern,
- Pfadtest:
  - Anzahl der Pfade durch ein Programm in der Regel unendlich,
  - aber mindestens: jede Kante des Ablaufgraphen einmal durchlaufen,
  - Vorsicht: Kontrollkomplexität $\neq$ Datenkomplexität.

Technisches zum Testen

Tests dokumentieren.

Häufig: Testdaten automatisch generieren.

Falls vorhanden: gegen Prototyp oder frühere Version testen.

Testdaten und -ergebnisse aufbewahren:

- wenn Wiederholung notwendig ist, kann Test automatisiert werden.

Statische Verifikation

Statische Verifikation:

- Programminspektion
- Formale (mathematische) Verifikation
- Statische Programmanalyse

Programminspektion (I)

Team:

Autor, Vortragender, Tester, Moderator.

Autor:

hat die Programmkomponente geschrieben.

Vortragender:

stellt den Code dem Team vor.

Tester:

begutachtet den Code (z.B.: Checkliste).

Moderator:

leitet die Sitzung.

Programminspektion (II)

Checkliste: z.B.:

Sind Variable initialisiert?

Haben Konstante Namen?

Sind Bedingungen korrekt?

Terminieren Schleifen?

Werden Array-Grenzen korrekt überprüft?

Falls Zeigerstruktur modifiziert wird, zeigt jeder Zeiger danach auf das richtige Datum?

Sind zusammengesetze Anweisungen geklammert?

Werden alle möglichen Fehlersituationen berücksichtigt?

Programminspektion (III)

Sitzungen:

kurz (und dafür häufiger),

ausschließlich: Feststellung von Fehlern und Nicht-Einhaltungen von Programmierstandards.

Nicht auf Programmcode beschränkt: funktioniert auch bei Anforderungsspezifikation oder Entwurf.

Formale Verifikation

Idee:

führe mathematischen Beweis, daß Programm Spezifikation erfüllt (oder: daß zwei Programme äquivalent).

Vorteil:

liefert möglicherweise Korrektheitsbeweis (im Gegensatz zum Testen).

Probleme:

hoher Lernaufwand für Programmierer,

hoher Zeitaufwand,

Beweise sehr lang und unübersichtlich,

$\leadsto$ Rechnerunterstützung erforderlich,

formale Semantik der Programmiersprache? (ML: ja!)

Statische Programmanalyse

Gewisse Anomalien in Programmen können automatisch erkannt werden, z. B.:

- unerreichbarer Code,
- uninitialisierte Variable,
- nie benutzte Variable,
- mehrfach nacheinander beschriebene Variable,
- manche Arraygrenzüberschreitungen.

Zum Teil werden diese Funktionen von Compilern automatisch ausgeführt,
sonst: zusätzliche Optionen oder separate Programme (z. B.: lint).

Ersetzt keine Tests/Programminspektion/formale Verifikation.

Aber: wenn vorhanden, **immer** benutzen.