Test und Verlässlichkeit Foliensatz 7: Software: Beschreibungsmittel, Vorgehen und Testauswahl

Prof. G. Kemnitz

6. Oktober 2025

Inhaltsverzeichnis

7.2 Software

Der Begriff Software wurde 1958 von John W. Tukey als Gegenstück zu dem wesentlich älteren Begriff Hardware geprägt für

- Programme, Einstellungen,
- HW-Konfigurationen, ...

Im Vergleich zur Hardware:

- Mehr funktionale Möglichkeiten, z.B. Fehlfunktionsbehandlung.
- Entstehungsprozess*: mehr Kreativität, weniger Automatisierung, weniger Determinismus.
- Die zuerst entstehen Gedanken, Notizen, nur manuell kontrollierbar. Gute automatische Kontrollen erst nach Rechnereingabe.

Vorzüge und Probleme für Test und Fehlerbeseitigung:

- Test und Fehlerbeseitigung viel einfacher als bei Hardware.
- Möglichkeit der Instrumentierung des zu testenden Codes mit Abdeckungszählern, zusätzlichen Kontrollen, ...

- Keine fehlerfreie Beschreibung des Sollverhaltens für die Kontrolle von Testausgaben und die Modellierung von Fehlern.

---

* Gilt auch für Hardware-Entwurf. Hier Abgrenzung zur Hardware-Fertigung.

7.3 Dieser Foliensatz ...

Automatisierung ist das wirksamste Mittel zur Fehlervermeidung, für Tests mit hoher Abdeckung und gute MF-Behandlung. Software bietet viele Möglichkeiten, fehlerträchtige Routine-Aufgaben und Kontrollen auf allen Ebenen von Software ausführen zu lassen.

Wir beginnen mit der **Programmiersprache** als Schnittstelle zwischen dem manuellen und automatisierten Entwurf mit Schwerpunkt auf Beschreibungsmittel zur Verbesserung der Verlässlichkeit.

Wichtig sind auch **Software-Architektur** und **Vorgehensmodell**, beides Kern-Knowhow von Software-Firma.

Hohe **Fehlerabdeckung** und **Zuverlässigkeit** verlangt viel umfangreichere Tests, als heute üblich, nur bei Automatisierung bezahlbar:

- **Kriterienabdeckung**, auch aus testbaren Anforderungen,
- **hohe Abdeckungsanzahl** und
- **Alternativen zum Soll/Ist-Vergleich** für Testausgaben.

Die Schaffung eine **Testinfrastruktur** zur weitgehend automatisierten **Testerzeugung und -durchführung** liegt im Bereich des Möglichen.

1 Programmiersprache

7.4 Programmiersprache und Verlässlichkeit

Die Programmiersprache ist die Schnittstelle zwischen den manuellen und den automatisierten Entstehungsschritten.

Die Wahl der Programmiersprache bestimmt sehr wesentlich

- den Programmieraufwand,
- mögliche und typische Fehler,
- Fehlerentstehungsrate,
- Fehlerfunktionsbehandlung, ...

Unterteilung der Fehler- und MF-Arten nach typ. Umgang damit:

ST	Ausschluss durch **s**tatische **T**ests zur Compile-Zeit.
DT	Suche mit **d**ynamischen **T**ests.
Tol	**Tol**erierung durch Check und Fehlfunktionskorrektur.
CT	Schadensvermeidung durch **C**heck + **T**erminierung.

Als Beispiel und, um Unterschiede zu erörtern, wird Rust betrachtet, eine neue Sprache, die versucht »alles richtig zu machen«.

7.5 Rust*

- Fortschrittlich im Fehlerausschluss durch statische Tests.
- Instrumentierung von Kontrollen in Testübersetzungen.

- Fortschrittliche Schrachunterstützung der MF-Behandlung.
- Fortschrittliche Beschreibung und Verwaltung von Tests.

Insbesondere rutschen einige verbreitete »bösartige« Fehlerarten wie Speicherlecks, Deadlocks, ... aus den Kategorien:

- DT (Suche mit **d**ynamischen **T**ests)
- Tol (**Tol**erierung durch Check und Fehlfunktionskorrektur)
- CT (Schadensvermeidung durch Check + Terminierung)

in die Kategorie »statisch nachweisbar«. Dadurch in übersetzten Programmen ausschließbar und somit unproblematisch.

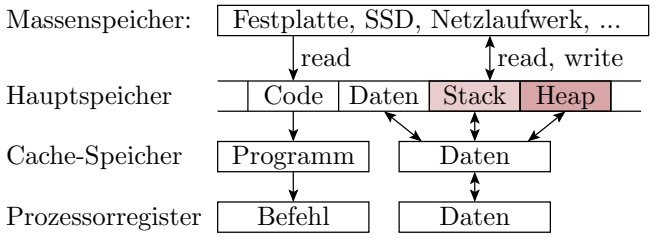
Verbesserte Sprachunterstützung für MF-Behandlung, Testprogrammierung und Testdurchführung motiviert zu mehr Kontrollen und Tests.

Viel umfangreichere Tests, als heute üblich, automatische Testerzeugung, viele Kontrollmöglichkeiten, ..., noch nicht im Fokus.

* https://rust-lang-de.github.io/rustbook-de.

1.1 Speicherlecks, ...

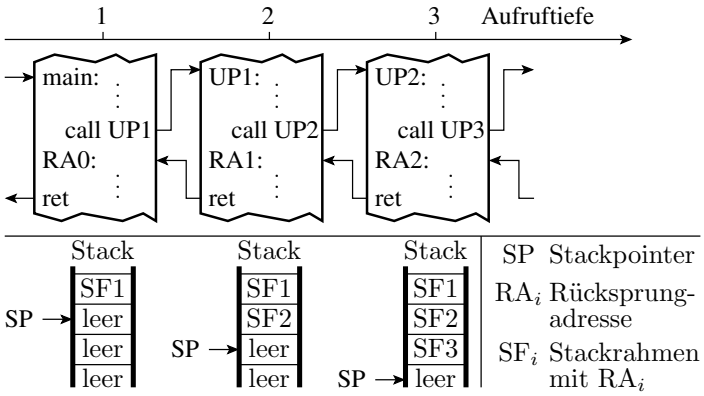
7.6 Speicherlecks, hängender Zeiger, ...



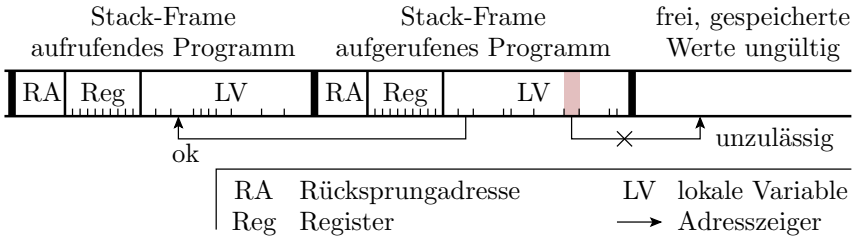
- Komplette Programme sind Dateien in Massenspeichern. Für aktive Programme werden Code-Kopien im Hauptspeicher gehalten und Platz für Daten reserviert.
- Von genutzten Code- und Datenbereichen werden Kopien im schneller zugreifbaren Caches gehalten, auf die der Prozessor zugreift.

Speicherlecks und hängende Zeiger hängen mit den Funktionsprinzipien bzw. der Nutzung von Stack und Heap zusammen.

7.7 Das Stack-Prinzip



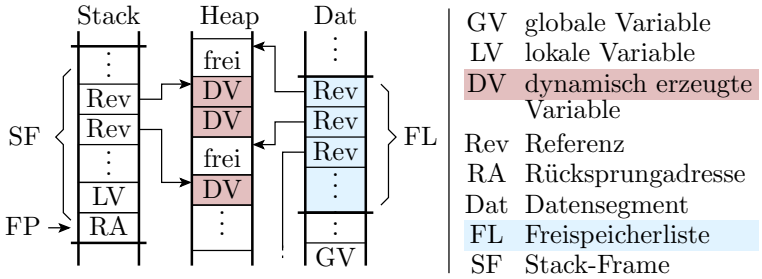
Programme sind Sammlungen von Funktionen, die aufgerufen werden und nach Beendigung zum Aufrufpunkt zurückspringen. Rückkehradresse werden auf einem Stapelspeicher (Stack) abgelegt. Das erlaubt große Aufruftiefe und sogar rekursive Aufrufe.



- In den Stackrahmen kopiert das Programmm auch Werte von Registern, die das aufrufende und das aufgerufene Programm beide benötigen, und reserviert Platz für lokale Variablen.
- Lokale Variablen werden relativ zum Stackrahmen adressiert und haben bei jedem Aufruf andere absolute Adressen.
- Nur Variablen von Programmen auf dem Stapel belegen Speicher.
- Nach Rücksprung werden alle lokalen Variablen ungültig.

Hängende Zeiger sind ungültige Adresswerte in gültigen Variablen. Im Normalfall ist es Aufgabe des Programmierers, das zu verhindern.

7.9 Dynamische Daten auf dem Heap



- Heap: von der Software verwalteter Datenbereich,
- Zugriff über Zeiger im Datensegment, Stack oder Heap.

Vereinfacht das Programmieren, ist aber eine böse Fehlerquelle:

- Hängende Zeiger (siehe Folie zuvor).
- Speicherleck: Auf dem Heap sich ansammelnde Speicherbereiche, auf die es im Programm keine Zeiger mehr gibt.
- Fragmentierung: Freispeicherzerstücklung in immer kleinere Teile.

Für Speicherlecks und hängende Zeiger bieten die meisten Programmiersprachen nur

- Suche mit dynamischen Tests (DT) und
- Programmabruch, wenn während der Laufzeit erkannt (CT).

Kein ausreichend großer zusammenhängender freier Speicher bei Anforderung durch Heap-Fragmentierung:

- durch sorgfältigen Umgang mit Freispeicher minderbar,
- mit dynamischen Tests schwer ausschließbar,

- zur Fehlfunktionsbehandlung kann versucht werden, kleine freie Speicherbereiche zu größeren zusammenzufassen oder Abbruch, Heap-Neuinitialisierung und Wiederholung.

In sicherheitskritischen Anwendungen Nutzung dynamischer Speicherverwaltung zum Teil verboten.

Rust hat ein neuartiges Beschreibungskonzept, zur Vermeidung hängende Zeiger und Speicherlecks (ST).

ST Ausschluss durch statische Tests zur Compile-Zeit.
CT Schadensvermeidung durch Check und Terminierung.
FT Fehlertoleranz durch Check + MF-Korrektur.
DT Ausschluss durch dynamische Tests.

7.11 Datenobjekte in Rust

Konstanten: benutzte Zahlenwert, Zeichen, Auszählungswerte:

```
3136, ..., 2.45, ..., "Text"
```

Variablen:

- global (static), feste Adresse im Code-Segment,

```
static x: u16 = 3136; // globale, unveränderbar
```

- lokal, Stack, feste Adresse im Stack-Frame der Funktion

```
let mut yyy: u32 = 0; // feste Größe, änderbar
let yyy = yyy + 1254; // Wertänderung
```

- dynamisch, Adresszuordnung zur Laufzeit auf dem Heap.

```
let v = vec![1, 2, 3];// änderbare Größe
```

Besonderheiten von Rust:

- `static` erzeugt globale Datenobjekte (feste Adresse),
- `let` erzeugt Datenobjekte zur Laufzeit (Stack oder Heap),
- Veränderbarkeit (**mut**ability) ist explizit festzulegen*.

* In C++ ist const explizit anzugeben. Nachteil oft nicht gemacht, wenn sinnvoll.

Erzeugung von Heap-Objekten standardmäßig als initialisierte Konstante mit Adresszeiger auf dem Stack als lokale Variable:

```
let a:<typ> = <komplexes Datenobjekt>;
```

Veränderbarkeit muss extra angegeben werden:

```
let mut b:<typ> = <komplexes Datenobjekt>;
```

Nur einen Besitzer. Beim Verlassen einer Funktion (oder Blocks) werden alle »im Besitz befindlichen« Objekte auf dem Heap gelöscht.

- Besitzweitergabe:

```
let b1 = b; // b existiert danach nicht mehr
```

- Bei Übergabe eines Heap-Objekts an eine Funktion geht der Besitz an die Funktion über und muss bei Rückkehr zurückgegeben werden. Rückgabewert ist ein Ausdruck ohne abschließendes Semikon:

```
  xxx  // Rückgabewert, auch Tupel
}      // schließende Klammer
```

Ausschluss hängender Zeiger und Speicherlecks durch Löschen der Datenobjekte bei Beendigung des Besitzers.

7.14 Referenzen

Referenzen sind Zeigerkonstanten auf Heap-Objekte, die statt der Objekte an Funktionen übergeben werden können:

```
fn get_len(s: &String)->usize {//Referenz auf s
  s.len()                      //Rückgabewert
}
```

- Referenzen besitzen das Objekt nicht, sondern borgen es nur.
- Wenn eine Referernz den Gültigkeitsbereich verlässt, wird nur die Referenz, aber nicht das Objekt selbst gelöscht, also keine Rückgabe erforderlich.
- Innerhalb eines Gültigtigkeitsbereichs sind mehrere nur lesbare Referenzen auf eine Objekt erlaubt.
- Bei einer **mut**able Referenz keine weitere Referenz zulässig.
- Löschen der Objekte mit der letzten Referenz.

Ausschluss hängende Zeiger, Speicherlecks, Read-After-Write-Hazzards, ... Programmierung gewöhnungsbedürftig. Bestimmte Konstrukte (Listen, Semaphore, ...) nur »unsafe« programmierbar.

1.2 Typ- und WB-Checks

7.15 Datentypen und Operationen

Elementare Typen fester Größe (wie alle Programmierhochsprachen):

- ganzzahlig: 8, 16, ... bit, vorzeichenfrei oder Zweierkomplement,
- Gleitkomma: 32, 64, ... bit aus Vorzeichenbit, Mantisse und Charakteristik, Sonderwerte: NaN, $+\infty$ und $-\infty$.
- Aufzählungen z.B. die Wahrheitswerte »true« and »false«, ...

An Typen sind Operationen und Kontrollmöglichkeiten gebunden:

- Operatoren und Funktionen haben zulässige Operandtypen. Aus denen resultierende Ergebnistypen (ST).
- Zulässigkeit konstanter Werte (ST)
- Zulässigkeit berechneter Wert (CT*)

Standardmäßige Zuweisung als »Konstanten« erlaubt mehr ST.

NAN Ungültig (not a number), z.B. Ergebnis der Division durch null.

ST Ausschluss durch statische Tests zur Compile-Zeit.

CT Schadensvermeidung durch Check und Terminierung.

* Rust instrumeniert in Testversionen WB-Überlaufskontrollen für Operationen, wenn nicht ausdrücklich unterbunden (fail-fast). In den Release-Code jedoch nicht (fail-slow).

7.16 Zusammengesetzte Typen

Feld als Zusammenfassung von Elementen desselben Typs:

```
// Feld mit veränderbaren Werten
let mut a: [i32; 5] = [1, 2, 3, 4, 5];
```

Verbund als Zusammenfassung von Objekten unterschiedlichem Typs:

```
// nur lesbares Tuple
let tup: (i32, f64, u8) = (500, 6.4, 1);
```

Operationen: Erzeugung, Übergabe an Funktionen und Rückgabe, Elementeauswahl, ... Kontrollmöglichkeiten auf Zulässigkeit:

- unzulässige Elementeauswahltyp (ST),
- unzulässige Index-Konstante (ST),
- unzulässiger berechneter Index (CT),
- unzulässige Zuweisungen (ST*, CT)...

ST Ausschluss durch statische Tests zur Compile-Zeit.

CT Schadensvermeidung durch Check und Terminierung.

* In Rust sind auch zusammengesetzte Objekte standardmäßig unveränderlich. Veränderbaren (mutable) Objekten dürfen nur Werte, aber nicht Typen oder Größe neu zugewiesen werden.

7.17 Kollektionen

Sammlung von Daten variabler Größe und Anzahl, die während der Programmausführung wachsen oder schrumpfen können mit unterschiedlichen Fähigkeiten. Die mitgelieferten Bibliotheken bieten:

- Vektor: gemischte Elementetypen, Erzeugen, Elemente anhängen und löschen, indizierter Zugrif Iteration über alle Elemente, ...
- Zeichenkette: Erzeugen, verketten, parsen, Zeichen suchen, ...
- Hash-Tabelle (hash map): Datenzugriff über Schlüssel.

Kollektionen nutzen den Heap und löschen automatisch ihre Heap-Daten, wenn der Gültigkeitsbereich verlassen wird.

Rust bietet komplexe Beschreibungsmittel wie Zeichenkettenverarbeitung incl. Parse-Funktionen, Hash-Tabellen z.B. für Wörterbücher, ...

- Verringert Quelltextgröße und -fehleranzahl.
- Die Beschreibungsmittel sind in der Regel für ein Maximum an statischen Tests und Kontrollen zur Laufzeit konstruiert.

7.18 Aufzählungen in Rust

Aufzählung von Werten, in der jedem Wert ein Tupel von Datenobjekten zugeordnet ist:

```
enum IpAddr {          // Datentyp für IP-Adresse
    V4(u8,u8,u8,u8), // V4-Adresse als vier u8
    V6(String),      // V6-Adresse als String
}
```

Vordefinierte Aufzählungstypen für die Fehlerbehandlung:

```
enum Option<T> {   // T generischer Typ
    None,    // kein gültiger Wert
    Some(T), // gültiger Wert vom Typ T
}
enum Result<T, E> {
    Ok(T),   // ohne MF ein Objekt vom Typ T
    Err(E),  // wenn MF, Beschreibung der MF
}
```

Option<T> und Result<T> erlauben eine einfache Rückgabe der Daten für die Fehlfunktionsbehandlung an die behandelnde Funktion.

7.19 Verarbeitung mit match

Abgleich von Werten mit einer Reihe von Mustern und Ausführung des zum Muster passenden Codes, z.B. Verarbeitung möglicherweise nicht existierender Daten:

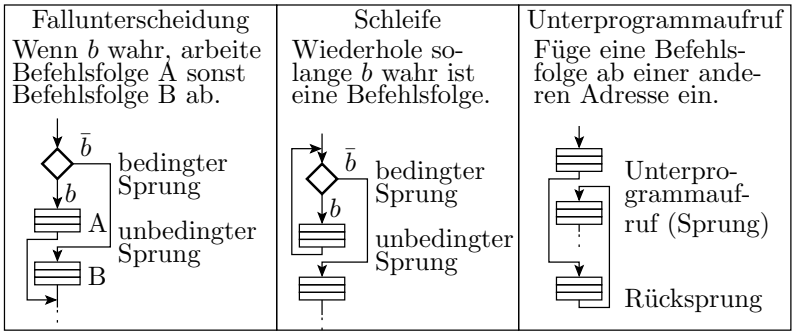
```
fn plus_one(x: Option<i32>) -> Option<i32> {
  match x {
      // for x=None return None
      None => None,
      // for x=Some(i) process i
      Some(i) => Some(i) + 1,
  }
}
```

Ein Match-Ausdruck muss für alle Mustermöglichkeiten einen Ausdruck vom selben Typ liefern. Kontrollen auf Zulässigkeit:

- unvollständige Musterabdeckung (ST),
- Fälle mit abweichendem Rückgabetyp (ST).
- Some(i) +1, kontrollieren!!

1.3 Kontrollfluss

7.20 Steuerung des Kontrollflusses



Nach jedem Befehl wird ein Folgebefehl abgearbeitet. Ausnahmen:

- Sprünge unbedingt, bedingt,
- Unterprogrammaufruf, Rücksprung.

Die Programmierung von Kontrollflüssen mit Sprüngen oder »goto« ist sehr fehlerträchtig und nach Möglichkeit zu vermeiden.

7.21 Fehlervermeidung in Hochsprachen

Alle Hochsprachen unterstützen »strukturierte Programmierung«, d.h. ein Fokusierung auf robuste (weniger fehleranfällige) Ablaufstrukturen

- Fallunterscheidungen, Schleifen,
- Unterprogramme, ...

Robuste Konstrukte für Nebenläufigkeit

- Threads, Semaphore, ...

Rust geht hier auch über das übliche hinaus:

- stärkere Restriktionen für mehr statische Kontrollen,
- mächtige Beschreibungsmittel für häufig genutzte komplexe Abläufe, insbesondere auch für die Fehlfunktionsbehandlung.

7.22 If-else und let-if in Rust

Bedingte Abarbeitung wie in allen höheren Programmiersprachen:

```
if number % 4 == 0 {
  println!("Zahl ist durch 4 teilbar");
} else if number % 3 == 0 {
  println!("Zahl ist durch 3 teilbar");
} else {
  println!("Zahl ist nicht durch 4 oder 3 teilbar");
}
```

- Bedingungen müssen Ausdrücke vom Typ boolean sein (ST).

Bedingte Zuweisung (Besonderheit von Rust):

```
let number = if condition { 5 } else { 6 };
```

Erlaubt als zusätzliche Kontrollen (ST):

- Abdeckung aller Möglichkeiten.
- Gleicher Typ aller Ergebnisausdrücke und
- Typkontrollen bei Verarbeitung und Zuweisung wie für Ausdrücke.

7.23 Schleifen in Rust

Wie in allen höheren Programmiersprachen:

```
loop {...,                // Schleife
  if <Bedingung> break    // Abbruchbedingung
};
while <Bedingung> {...}; // Abweisschleife
```

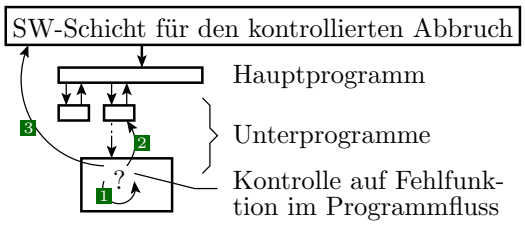
Besonderheiten mit Bezug zur Fehlervermeidung und MF-Behandlung:

- Schleifen mit Rückgabewert (Ausdruck hinter break), um z.B. in Fehlersituationen aus der Schleife zu eine MF-Behandlung zu wechseln und danach die Schleife mit Abbruchwert fortzusetzen.
- Schleifen-Label, um bei Verschachtelung mehrerer Schleifen festlegen zu können, welche Schleife mit break zu verlassen ist.
- For-Schleife über alle Elemente einer Kollektion. Häufig genutzt, kompakter und weniger fehleranfällig als mit Index-Variable:

```
let a = [10, 20, 30, 40, 50];
for element in a {
   println!("Der Wert ist: {element}");
}
```

1.4 MF-Behandlung

7.24 Fehlfunktionsbehandlung



Kontrollen im laufenden Betrieb erfolgen an vielen Stellen im Kontrollfluss. Mögliche Orte der Fehlfunktionsbehandlung:

1. Innerhalb der Prozedur, in der die Fehlfunktion erkannt wird, z.B. Ersatz nicht plausibler Werte durch Standardwerte.
2. Aufrufebenen tiefer. Rücksetzen des Stacks. Rückgängig machen der versuchten Berechnungen, ...
3. Nicht behandelbare Fehlfunktion. Kontrollierter Abbruch. Freigabe Stack, Heap, reservierte Geräte, Dateien schließen, sicherer Zustand, Fehlermeldung, Infos für Fehlersuche, ... (Abschn. 1.2.6).

7.25 Kontrollierter Abbruch

Rust Makro panic!():

```
if <malfunction> panic!(<error message>);
```

Erzeugt eine Fehlermeldung mit

- Quelldateinahme und Zeilennummer des Panic-Aufrufs und
- dem Ausgabetext hinter **panic**.

Mit der Umgebungsvariable »RAST_BACKTRACE=1« wird der komplette Aufruf-Stack ausgegeben.

Automatisch eingefügte Kontrollen, z.B. auf unzulässige Index-Werte bewirken immer einen Panic-Abbruch:

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];     // Indexfehler, Abbruch mit panic
}
```

In anderen Programmiersprachen bieten vergleichbare Konstrukte, z.B. exit() in Python. Wenn kontrollierter Abbruch versagt, Neuinitialisierung des kompletten Rechners.

7.26 Behandlung mit try-except

Ausprobieren möglicherweise scheiternder Befehlsfolgen. Für behandelbare Probleme, Wiederherstellung des Zustands vor »try« und alternative Berechnung:

```
while True:           # Python-Beispiel
  try:                # Ausführungsversuch
    x = int(input("Please enter a number: "))
    break
  except ValueError: # Fehlfunktionsbehandlung
    print("Invalid number.  Try again ...")
```

Bei MF-Abbruch nach »try« wird für die aufgelisteten MF-Typen der Code nach »except« ausgeführt.

Die Wiederherstellung des Zustands vor »try« ist kompliziert. Zur eigenen Sicherheit testen, was der implementierte Automatismus wirklich leistet (Stack wieder abgeräumt, keine Daten verändert, Reservierungen Speicher und Geräte freigegeben, ...).

7.27 Rust geht einen anderen Weg

Händische MF-Behandlung ohne »try-except« auch bei Fehlfunktionen mehrere Aufrufebenen darüber mit Rückgabetyp »Result«.

```
enum Result<T, E> {
    Ok(T),   // ohne MF ein Objekt vom Typ T
    Err(E),  // wenn MF, Beschreibung der MF
}
```

Eine Beispielfunktion, die fehlschlagen kann, ist das Öffnen einer Datei:

```
let greeting_file_result = File::open("hallo.txt");
let greeting_file = match greeting_file_result {
  Ok(file) => file,
  Err(error)=>panic!("Problem beim Öffnen:{:?}",error),
};
```

Beispielausgabe, wenn die Datei nicht existiert:

```
thread 'main' paniced at 'Problem beim Öffnen: Os {code: 2, kind: NotFound,
message: "No such file or directory" }', src/main.rs:8:23
```

7.28 Durchreichen von Rücksprüngen mit »?«:

Der Operator »?« hinter einer Zuweisung eines Wertes vom Typ »result« bewirkt einen Funktionsabbruch mit Rückgabe »Err(E)«:

```
fn read_username_from_file()->Result<String,io::Error>{
  let mut username_file = File::open("hallo.txt")?; //**
  let mut username = String::new();
  username_file.read_to_string(&mut username )?;
//**
  Ok(username) // Sonst Rückgabe gelesener Datei-Text
}
// ** Abbruch bei Fehler mit Err(result)
```

Voraussetzung:

- Die Funktion muss auch Rückgabetyp »result« haben und
- gleicher Datentyp »E« für die Fehlerbeschreibung.

Die Sprachunterstützung der Fehlfunktionsbehandlung entscheidet über den händischen Programmieraufwand und darüber über den Umfang der Fehlerfunktionsbehandlung in den Programmen.

1.5 Test

7.29 Rust-Beschreibungsmittel für Tests

Gemeint sind dynamische Tests. In SW sind Tests Funktionen, die

- die zu testenden Funktionen mit Bespielwerten ausführen und
- die Ergebnisse kontrollieren.

In Rust erhalten Testfunktionen die Annotation »#[test]«:

```
#[test]
fn test_add2() {
  let test_tup = ((1, 2), (3, 5));// Testeingaben
  for (a, b) in test_tup {     // für alle Testeingaben
    let sum = add(a, b);       // Ausführung des Testobjekts
    assert_eq!(sum, a + b);    // Ergebniskontrolle
  }
}
```

Das Kommando »cargo test« führt alle Tests im Projekt aus mit Protokollausgabe der ausgeführten Tests, ob bestanden, ...

Der Test der MF-Behandlung mit Programmabruch verlangt auch Tests der MF-Behandlung mit »panic« als Sollverhalten:

```
fn add_u8(x: u8, y:u8)-> u8 {x + y}

#[test]
#[should_panic]
fn test_add_u8_overflow() {
  let sum = add_u8(128, 128); // Ergebnisüberlauf, panic
}                             // bei Testübersetzung
```

Die Annotation #[should_panic] wandelt »panic« in Test bestanden und erfolgreiche Ausführung in die Protokollausgabe »Test nicht bestanden« um.

Bei Funktionen mit Rückgabetyp »result« erfolgt der Test der Fehlerbehandlung mit ganz normalem Soll-/Ist-Vergleich.

Weitere Makros für die Testauswertung:

```
assert!(<einzuhaltende Bedingung>) // Kontrolle "wahr"
assert_neq!(<Wert 1>, Wert 2>);    // Kontrolle ungleich
```

7.31 Optionen für die Testausführung

- Auswahl oder Unterdrückung eines Teils der Tests.
- Ausführung parallel oder nacheinander im selben Thread.
- Modultests: isolierter Test einzelner Funktionen.
- Integrationstest: Test des Gesamtsystems über die externen Schnittstellen.

Einfache Testbeschreibungen und die Automatisierung der Testdurchführung motivieren zum gründlichen Testen.

Zusammenfassung

7.32 Zusammenfassung

Die Programmiersprache ist die Schnittstelle zwischen den manuellen und den automatisierten Entwurfsschritten. Sie bestimmt wesentlich

- Programmieraufwand, mögliche und wahrscheinliche Fehler,
- eingebaute Kontrollen und Fehlerbehandlung.

Rust hat gegenüber vergleichbaren Programmiersprachen

- Beschreibungsmittel für den Ausschluss hängender Zeiger und Speicherlecks,
- mehr Möglichkeiten auch für statische Tests.
- eine innovative Testunterstützung,
- Fail-Fast für den Test und Fail-Slow für den Einsatz, ...

Vorteile werden natürlich mit Nachteilen erkauft:

- Deutlich komplizierterer Übersetzungsprozess,
- Gewöhnungsbedürftige Programmierung (Lernaufwand)

Rust ist eine fortschrittliche Programmiersprachen für Fehlervermeidung, Test und MF-Behandlung, aber noch nicht die Lösung für alles.

2 Software-Architektur

7.34 Software-Architektur

Eine Software-Architektur gibt einen Rahmen vor für die

- Aufteilung eines Systems in Teilbausteine und
- die Gestaltung der Schnittstellen zwischen den Teilsystemen.
- Lehnt sich oft an die Kommunikationsstruktur der Organisationseinheit, die das System entwickelt, an (Conway's Gesetz).

Wichtige Software-Architekturmuster:

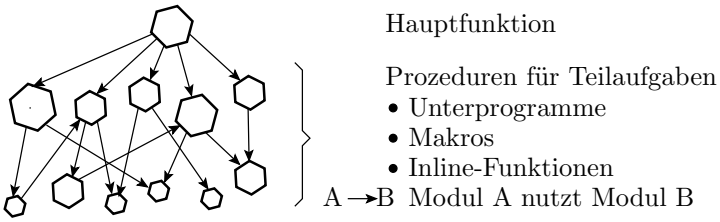
- Prozedurensammlung,
- Schichtenmodell, Client/Server-Modell, ...

Eine geeignete Software-Architektur ist die Basis für langfristig testbare, wartbare, verständliche Systeme, und damit für verlässliche Systeme.

Gesetz von Conway: Organisationen, die Systeme entwerfen ..., sind gezwungen, Entwürfe zu erstellen, die die Kommunikationsstrukturen dieser Organisationen abbilden [2].
Software-Wartung: Änderung an der Software nach Auslieferung zur Fehlerbeseitigung, Leistungsverbesserung oder Anpassungen an die veränderte Umgebung.
Weiterführende Literatur: [3]

2.1 Prozedurensammlung

7.35 Prozedurensammlung



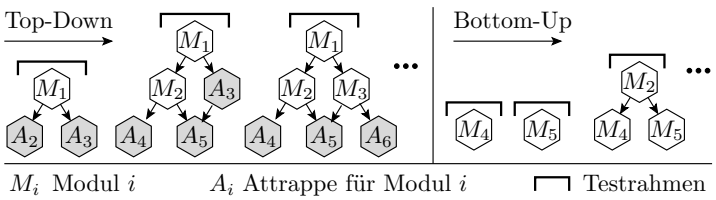
Die Architekturmuster »Prozedurensammlung« verkörpert die Idee des hierarchischen Entwurfs. Gesamtsystem als strukturiertes Programm aus Anweisungen und Teilprogrammen (Unterprogramme, Makros, ...), die wiederum Teilprogramme nutzen.

Frei gestaltbare Nutzungsbeziehungen. Schnellster Weg zu einem ausprobierbaren Programm. Mit zunehmender Systemgröße, Entwickleranzahl zu unübersichtlich und zu schwer weiterzuentwickeln.

Typisch für Teilbausteine, Einpersonenprojekte, Demonstratoren, ...

Prozeduren: Strukturierte Abläufe oder Anweisungen zur Durchführung bestimmter (Teil-) Aufgaben.
Demonstrator: Vereinfachte Impementierung zur Untersuchung und Demonstration der Machbarkeit.

7.36 Entwurf und Test

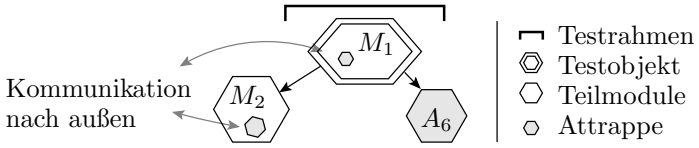


Entwurf und Test modulweise. Modultests als **Service** in Testrahmen, die Testeingaben bereitstellen, den zu testenden Code ausführen und die Testausgaben kontrollieren.

- **Top-Down**. Beginn Entwurf und Test übergeordneter Module. Nachbildung noch nicht existierender Teilbausteine durch Attrappen
- **Bottom-Up**. Beginn Entwurf und Test mit den untersten Modulen. Test der übergeordneten Module mit den bereits getesteten Untermodulen.

Service System, das auf Anforderung aus Eingaben Ausgaben erzeugt.

7.37 Attrappen und Instrumentierung



Reproduzierbare Testabläufe und -ergebnisse verlangen Entkopplung von externen Einflüssen. Das erfordert Attrappen für

- Modul, die noch nicht entwickelt sind,
- externe Datenkommunikation (Nutzer, Netzwerk, ...),
- externe Ablauf- und Zeitsteuerung (Timer, Prozesse, ...), ...

Die beiden letzten Punkte betreffen viele Funktionen, die das Betriebssystem bereitstellt.

Instrumentierung. Einfügung von Zusatz-Code in die Testübersetzungen: Attrappen, Ergebnis- und Abdeckungskontrollen, Zusicherungen, Injektion von Verfälschungen, ...

7.38 Beispiel für eine erforderliche IO-Attrappe

Das Beispieltestobjekt liest insgesamt Ct_max Zeichen und zählt dabei enthaltene Zeichen vom Typ A und vom Typ B:

```
int Ct_A, Ct_B, Ct_N;        // Programmausgaben
int CountChar(int Ct_max){   // Testobjekt
  char c;
  Ct_A=0; Ct_B=0; Ct_N=0;
  while (Ct_N<Ct_max){
    c=getchar();             // Zeicheneingabe
    if (is_TypA(c))
      Ct_A++;
    else if (is_TypB(c))
      Ct_B++;
    Ct_N++;
  }                          // Schleifenende
}                            // Programmende
```

Für einen Test als Service muss der Testrahmen auch die Zeichen, die getchar() liest, bereitstellen.

Die nachfolgende Attrappe liest die zurückzugebenden Zeichen aus einem Feld, dass der Testrahmen vor Aufruf des Testobjekts initialisiert:

```
int gct, gct_max;     // private Daten, Zähler,
char char_str[MAX];   // Testeingabezeichen
char getcharAttrappe(){
  assert(gct<gct_max);
  return char_str[gct++];
}
```

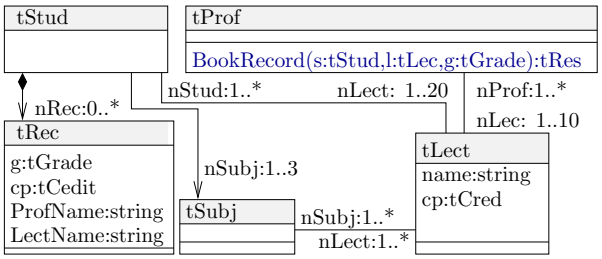
Testrahmen mit Zufallseingaben auch für die getchar()-Rückgabewerte:

```
TestCountChar(){
 while(<Testabdeckung noch nicht erreicht>){
  gct=0;              //Initialisierung Attrappe
  gct_max = <Zufallswert 0 bis MAX>;
  <beschreibe char_str mit gct_max Zeichen>
  CountChar(gct_max);// Testausführung
  <protokolliere Testein- und Ausgaben>
}
```

Je weniger Attrappen, desto besser der Code.

2.2 Objektorientierung

7.40 Objektorientierung

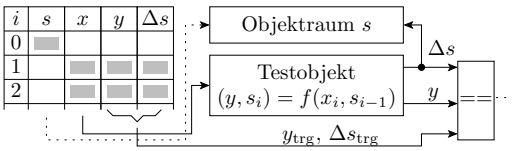


Beispiel Verbuchungssystem für Studienleistungen mit Objekten für Studenten, Professoren, Studiengänge, Lehrveranstaltungen und Studienpläne mit einer Funktion zur Leistungsverbuchung (Folie 5.49).

Die Sicht »Was (Datenstruktur) wird bearbeitet und wie (Prozeduren)?« ist näher an der menschlichen Vorstellungswelt und erlaubt eine anschauliche Darstellung **viel komplexerer Sachverhalte**.

Dafür **mehr** erforderliche **vorausschauende Planung**, mehr Programmieraufwand, ... als mit einer Prozedurensammlung.

7.41 Test von Methoden (Folie 5.50)



Der Testrahmen muss zusätzlich zu den Eingabendaten, im Beispiel Student, Lehrveranstaltung und Note, ein Beispielobjektraum erzeugen, auf den er die zu testende Methode auch mehrfach anwenden kann.

Ein Zufallstest muss im Beispiel ein zufällige Menge von Objekten je für Studenten, Professoren etc. mit zufälligen Daten erzeugen. Aus diesem Objektraum ist das Testobjekt für einen zufälligen Professor für einen Studenten einer seiner Veranstaltungen zur Verbuchung einer zufälligen Leistung aufzurufen. ...

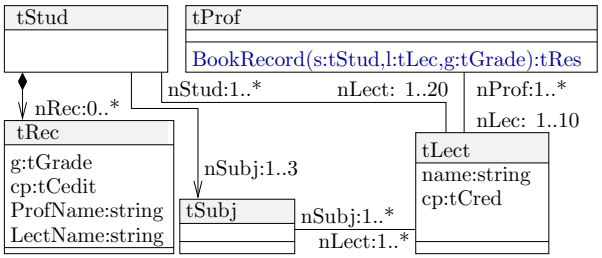
Zu kontrollierende Ergebnisse ...

---

x, y Testeingaben, Testausgaben.

s, Δs Objektraum (bearbeitete gespeicherte Daten), Änderungen des Objektraums.

y_{trg}, Δs_{trg} Sollwerte der Testausgaben und der Objektraumänderung.



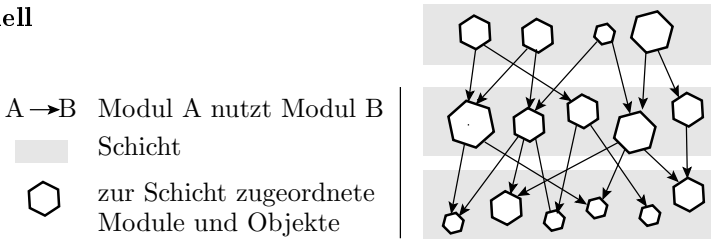
- Einen Test zu schreiben ist kein Hexenwerk, aber viel aufwändiger als der Entwurf des Testobjekts.
- Aus der Klassenbeschreibung lassen sich Beispiele für Objekträume und Testeingaben bzw. symbolische Tests anschaulich auch automatisiert ableiten, ...

Je komplexer die Datenobjekte, umso wünschenwerter ist Rechnerunterstützung bei der Codierung und Ausführung von Tests.

»Objektorientierung« ergänzt »Prozedurensammlung« um Strukturierungselemente zur Verbesserung von Verständlichkeit, Wartbarkeit, Testbarkeit, ... und macht komplexere Entwürfe beherrschbar.

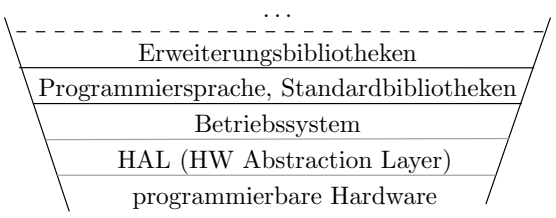
2.3 Schichtenmodelle

7.43 Schichtenmodell



- Zuordnung der Prozeduren bzw. Klassen zu Schichten.
- Prozeduren eine höhere Schicht dürfen nur Prozeduren der eigenen Schicht und der Schicht darunter nutzen.
- Die Nutzungsbeschränkungen **verbessern** Verständlichkeit, **Wartbarkeit**, **Testbarkeit**, ... erheblich und sind **statisch kontrollierbar**.
- **Einem Stilbruch** (z.B. Durchbrechen einer Schicht) erlaubt manches viel schneller und einfacher zu lösen, aber **verursacht langfristig Probleme**.

7.44 Rechner als Schichtenmodell



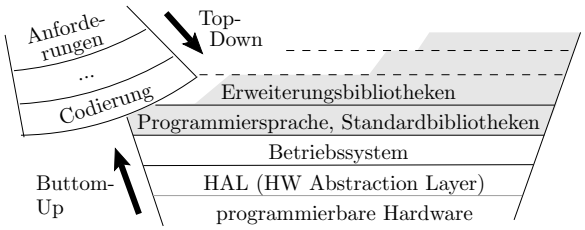
Software legt aus Programmierer- und Nutzersicht funktionale Schichten über die Hardware und ist selbst in Schichten organisiert:

- Hardware: Maschinenbefehle, Konfigurationsregister, ...
- HAL: Adapter, um die HW unter der SW austauschen zu können.
- Betriebssystem: Basisfunktionen für Ein/Ausgabe, Prozesse, ...

Die Betriebssystemschicht verbietet z.B. den direkten Zugriff auf den COM-Port. Ein Anwenderprogramm muss erst anfragen, ob COM-Port verfügbar, ..., bis Schreib- und Lesezugriffe durchgereicht werden.

HAL Hardware-Abstarktionsschicht.

7.45 Zusammenspiel Schichten und Entwurf



Entwicklung eines Anwenderprogramms:

- Programmierersicht: Progammiersprache plus Bibliotheken incl. Betriebssystemfunktionen.
- Nachbildung der Zielfunktion mit diesen Beschreibungsmitteln.

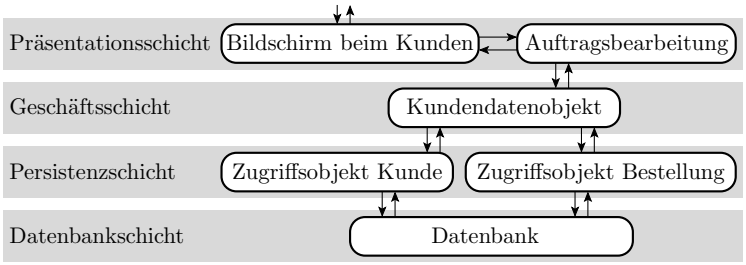
Nutzung:

- Lösen von Aufgaben mit Beschreibungsmitteln der Programme.

Entwicklung von Schichten für die Programmierung:

- ...

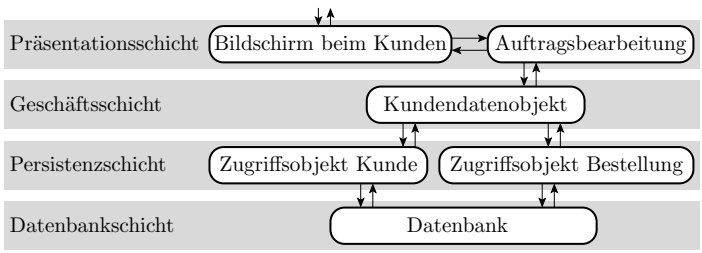
7.46 Eine oft genutzte Schichtenarchitektur



In Anlehnung an typische Unternehmens-Kommunikationsstrukturen*:

- Präsentationsschicht für die Kommunikation mit den Nutzern,
- Geschäftsschicht für die Auftragsbearbeitung,
- Persistenzschicht für die Verwaltung aufzubewahrender Daten und
- Datenbankschicht für die eigentliche Datenaufbewahrung.

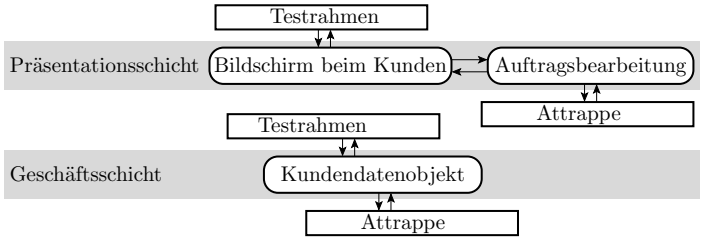
Gesetz von Conway: Organisationen, die Systeme entwerfen ..., sind gezwungen, Entwürfe zu erstellen, die die Kommunikationsstrukturen dieser Organisationen abbilden [2].



Arbeitung von Kundenaufträgen: Daten aktualisieren, bestellen, ...

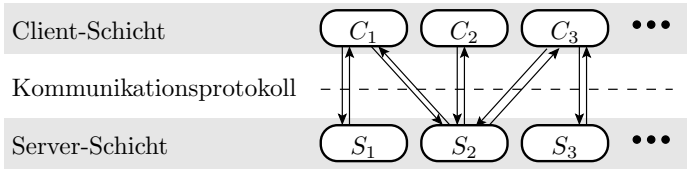
- Bildschirm: Dateneingabe und Ergebnispräsentation.
- Auftragsmodul: Auftrag erfassen, Auslösen und Ergebnisse übernehmen.
- Auftragsweitergabe an das Kundendatenobjekt der Geschäftsschicht zur Bearbeitung.
- Die für die Auftragsbearbeitung erforderlichen Lese- und Schreibzugriffe erfolgen über Zugriffsobjekte in der Persistenzschicht mit Datenbankanfragen in der Datenbankschicht.

7.49 Wartung und Test



- Die Schnittstellen der Schichten sollten sich im Projektverlauf nicht mehr ändern.
- Die Programmierer benötigen nur die Schnittstellen nach oben und unten, idealerweise mit Nutzungsbeispielen.
- Änderungen in einer Schicht sollte keine Auswirkungen auf die Tests zwischen den Schichtgrenzen haben.
- Nutzungsbeispiele nach oben sind potentielle Tests, Nutzungsbeispiele nach unter potentielle Attrappen [1].

7.50 Client-Server-Modell (2 Schichten)

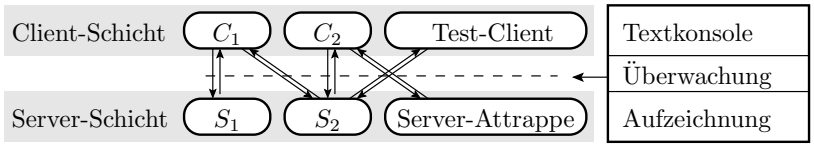


Server sind autonom arbeitende Dienstprogramme, Clients sind Applikationen, die Dienste der Server nutzen. Beispiele:

- Datenbankserver und mehrere Anwenderprogramm-Clients.
- Web-Applikationen: Webseiten als Benutzerschnittstelle, Server zur Leistungserbringung auf Rechnern irgendwo in der Welt.
- Auslagerung rechenlastiger Aufgaben auf andere Rechner.

Client-Server-Architekturen sind flexibel, leicht auf andere Plattformen portierbar und bieten zentrale Test- und Überwachungsschnittstellen.

7.51 Test und Überwachung



Eine zentrale Client-Server-Schnittstelle ist ideal für

- Dezentralisierung und Kommunikation über Netzwerke
- Überwachung / Aufzeichnung der Kommunikation.
- Server-Tests mit Test-Clients, Client-Tests mit Server-Attrappen.

Schichten mit Textkonsolen und Script-Sprachen, nutzbar auch für Überwachung, Test und Fehlersuche:

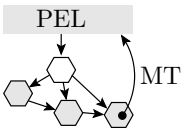
- Betriebssystem-Shell, z.B. unter Linux

```
ls -l *.pdf # Ausführen ls('-l', '*.pdf')
```

- Windows Low-Level Benutzerinteraktion:

```
send_xevents keydn Control_L
send_xevents keyup Control_L
```

7.52 Fehlfunktionsbehandlung

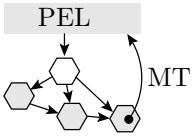


PEL Programmausführungsschicht
⬡ Hauptprogramm
⬡ Teilmodul
• erkanntes Fehlverhalten
MT kontrollierter Abbruch

Die Schicht für die Ausführung von Anwenderprogrammen unter einem Betriebssystem ist auch die Schicht für den kontrollierten Abbruch. Wenn ein Programm sich beendet, auch bei Fehlerabbruch, müssen

- alle reservierten Ressourcen (reservierter Speicher, geöffnete Dateien, reservierte IO-Schnittstellen, ...) freigeben,
- eventuell weitere Funktionen für Aufräumarbeiten, zur Herstellung eines sicheren Zustands, ... ausgeführt,
- Daten für die Fehlersuche, z.B. Bereiche des Datenspeichers und der Aufrufstack, zur Fehlfunktionsbeschreibung hinzugefügt werden, ... (vergl. Abschn. 1.2.6).

PEL, MT Programmausführungsschicht, kontrollierter Abbruch.



Debugger und andere Testhilfen

- benötigt Daten innerhalb der Programmausführungsschicht, müssen deshalb Teil dieser Schicht sein,
- sind aber selbst Programme, die abstürzen können.

Die normalen »geschützten« Funktionen für Speicherreservierung etc. nicht verfügbar.

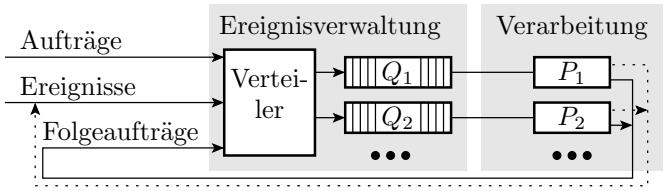
Der C-Debugger unter Visual Studio verwendet z.B. Kernel-Speicher, der bei Absturz nicht freigegeben wird. Ohne genug freien Kernel-Seicher stürzt irgendwann das Betriebssystem ab.

Schichtenmodelle ergänzt die Architekturmuster Prozedurensammlung und Objektorientierung um Restriktionen für die Nutzungsbeziehungen, wieder zur Verbesserung von Verständlichkeit, Wartbarkeit, Testbarkeit, ... d.h. zur Beherrschung noch komplexerer Systeme.

PEL, MT Programmausführungsschicht, kontrollierter Abbruch.

2.4 Weitere Architekturen

7.54 Ereignisgesteuerte Architektur



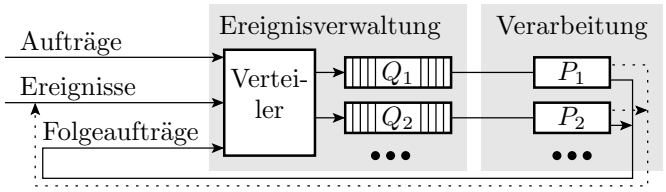
Aufträge werden in Warteschlangen sortiert. Sobald alle Abarbeitungsbedingungen erfüllt, Weitergabe an die passende Ausführungseinheit.

Verarbeitung stoppt bei der nächsten Wartebedingung und generiert Folgeaufträge (Broker Topologie).

Alternative Erzeugung der Ereignisfolgen in der Verwaltungsschicht (Mediator Topologie). Weniger flexibel, aber weniger anfällig für Systemverklemmungen und einfacher zu testen.

Q_i, P_i Warteschlange und Verarbeitungs-Service jeweils für Aktionen eines Typs i.

Systemverklemmung: Sich gegenseitig ausschließende Bedingungen für die Weiterarbeit, z.B. zwei Aktionen A und B warten gegenseitig auf den Abschluss der anderen.

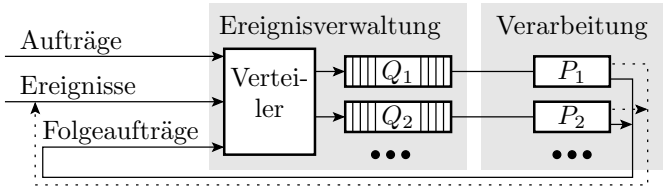


Ein Beispielsystem ist eine Robotersteuerung:

- Aufträge sind Bewegungen, die in Teilaufträge für einzelne Gelenke aufgeteilt werden.
- Die Aufträge in den Warteschlagen warten auf Bereitschaftssignale von anderen Aktivitäten und der Hardware.
- Aktivitäten sind Service-Leistungen mit Datenübernahmen von Sensoren, Berechnungen, Ausgaben auch in Form von Signalen an die Hardware (Aktorausgaben, Sensordatenanforderungen, Start von Timern, ...) und Bereitschaftsmeldungen und Folgeaufträgen für die Ereignisverwaltung.

Q_i, P_i Warteschlange und Verarbeitungs-Service jeweils für Aktionen eines Typs i.

7.56 Entwurf und Test

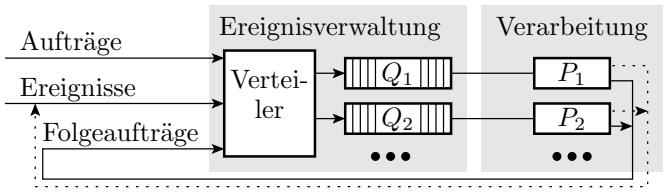


Test der Verarbeitungseinheiten als Service-Leister in einem Testrahmen und mit Attrappen für intern angeforderte Sensor- und andere angeforderte Eingaben und Protokollierung interner Ausgaben incl. generierter Signale und Folgeaufträge zur Kontrolle.

Für die Ereignisverwaltung gibt es spezielle Modelle zur Verhaltensbeschreibung (Petrinetze, Automaten, Aktivitätsdiagramme, ...) zur

- Veranschaulichung und Simulation der Ablaufmöglichkeiten,
- Untersuchung auf mögliche Verklemmungen,

Fortsetzung nächste Folie.

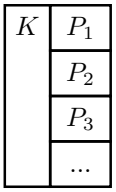


- Generierung von Ablaufkontrollen ###,
- Ableitung von symbolischen Tests und Testabdeckungskriterien.

Idealerweise automatische Code-Generierung der Ereignisverwaltung aus der Verhaltensbeschreibung.

Kapselung der Verarbeitung in gut testbare Service-Leister.
Ereignisverwaltung in ein möglichst anschauliches gut untersuchbares Modell, aus dem sich idealerweise die Tests und der Code automatisch generieren lassen. Codegeneratoren sind gut für die Fehlervermeidung, insbesondere auch bei der Änderung und Wartung.

7.58 Mikrokernel-Architektur



Aufteilung in ein Kernsystem und Module zur Funktionserweiterung:

- Kernsystem: Minimalsystem mit für den Betrieb erforderlichen Mindestfunktion.
- Zusatzmodule: voneinander unabhängige Zusatzfunktionen, idealerweise zur Laufzeit nachladbar.

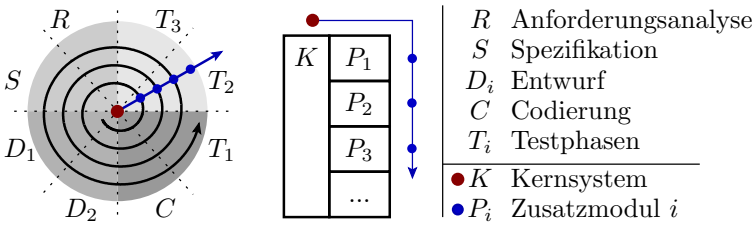
Beispiele:

- Web-Browser,
- Betriebssysteme (Linux),
- Entwicklungsumgebungen (Eclipse), ...

Bevorzugtes Architekturmuster für Anpassbarkeit an unterschiedliche Kundenbedürfnisse (z.B. landesspezifische Besonderheiten), geplante schrittweise Funktionserweiterung, ...

Q_i, P_i Kernsystem, Zusatzmodul i.

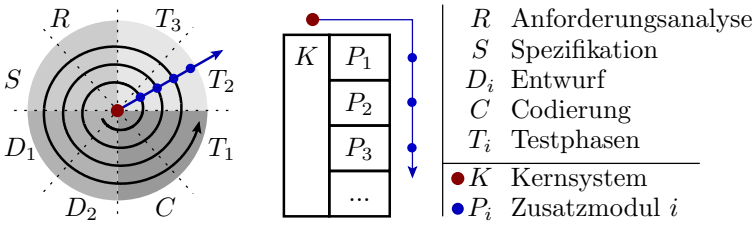
7.59 Vorgehen und Architektur



Nach dem Spiralmodell

- Durchlauf alle Stufen ab Anforderungsanalyse bis Test und Inbetriebnahme.
- In jeder Iteration Sammeln von Ideen und Änderungswünschen für die nächste Iteration als spezielle Organisiationsform für das Lernen aus Fehlern.

Mit Architekturmuster Mikrokernel Entwurfsreihenfolge Kernsystem, Zusatzmodule, jeweils von Anforderungsanalyse bis Test.



Ein strenger Entwurfsablauf Kernsystem, Zusatzmodule, jeweils von Anforderungsanalyse bis Test natürlich nicht umsetzbar:

- Kernel-Schnittstellen müssen auf alle Erweiterungen vorbereitet sein, also Anforderungsanalyse späterer Iterationen vorziehen.
- Jede Iteration verlangt Anpassungen der bereits entworfenen Bestandteile, ...

Das Architekturmuster Mikrokernel berücksichtigt außer Modularisierung von Entwurf und Test auch Aspekte der Entwurfsorganisation, insbesondere zum Lernen aus Erfahrungen (Fehlern) zu lernen.

Zusammenfassung

7.61 Zusammenfassung

Eine geeignete Software-Architektur ist die Basis für langfristig testbare, wartbare, verständliche Systeme, und damit für verlässliche Systeme.

Die skizzierten Architekturmuster

- Prozedurensammlung, Objektorientierung,
- Schichtenmodell, Client-Server,
- Ereignissteuerung,
- Mikrokernel und es gibt weitere

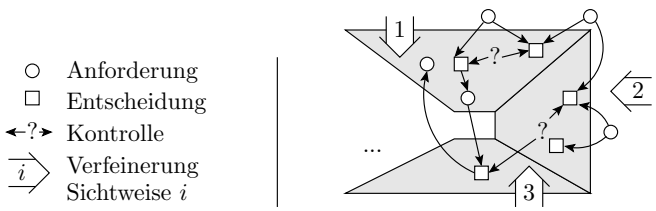
definieren kombinierbare Regeln für die Systemgestaltung mit unterschiedlichen Schwerpunkten.

Je komplexer die Systeme, desto mehr Erfahrung und Aufwand erfordert die Suche einer geeigneten Software-Architektur.

Stilbrüche, z.B. das Durchbrechen einer Schicht, erlauben manches viel schneller und einfacher zu lösen, verursachen aber in der Regel später bei Wartung und Weiterentwicklung langfristig Probleme.

3 Entwurf

7.62 Entwurfsprozesse

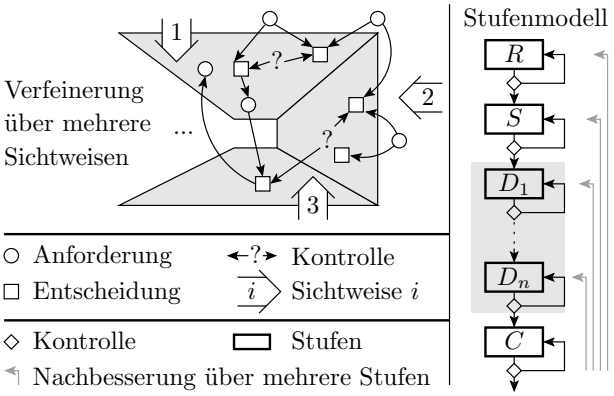


Der Entwurf komplexer IT-Systeme ist ein Entscheidungs- und Ververfeinerungsprozess über mehrere Sichtweisen:

1. Ressourcen (Knowhow, Personal, ..., benötigt/verfügbar),
2. Nachnutzung, Unteraufträge,
3. Struktur: Modulaufteilung, Schnittstellen, ...

Verfeinerungsprozess: Entscheidung. Kontrolle der Konformität mit anderen Entscheidungen aus anderen Sichtweisen. Präzisierung, Ableitung neuer Anforderungen, Entscheidung, ...

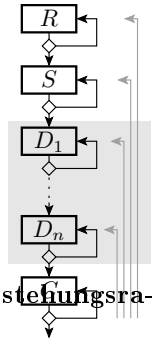
7.63 Vorgehensmodell



Zum Lernen aus Fehlern wird dieser Verfeinerungsprozess in Vorgehensmodelle aus Stufen und Kontrollen gepresst.

R, S Anforderungsanalyse, Spezifikation.

D_i, C Schritt *i* des Architektur- und Funktionsentwurfs, Codierung.



Denn Projekte sind **einmalige Vorhaben**, aber Fehlervermeidung durch **Lernen aus Fehlern** setzt viele Wiederholung voraus. **Vorgehensmodelle** vereinheitlichen das Vorgehens für eine große Anzahl vergleichbarer Projekte (Abschn. 2.4.4).

Zu Beginn eines Projekts Erstellung eines Projektplans mit konkreten Aufgaben, Meilensteinen, Verantwortlichkeiten, ... unter Nutzung alter Projektpläne. Bewährtes wird beibehalten. Für erkannte Probleme, die Unvorhersagbarkeiten und Fehler verursacht haben, wird nach einem verbesserten Vorgehen gesucht.

Ziele sind **bessere Vorhersagbarkeit** Ergebnisse, Aufwands und **Minderung Fehlerentstehungsrate**.

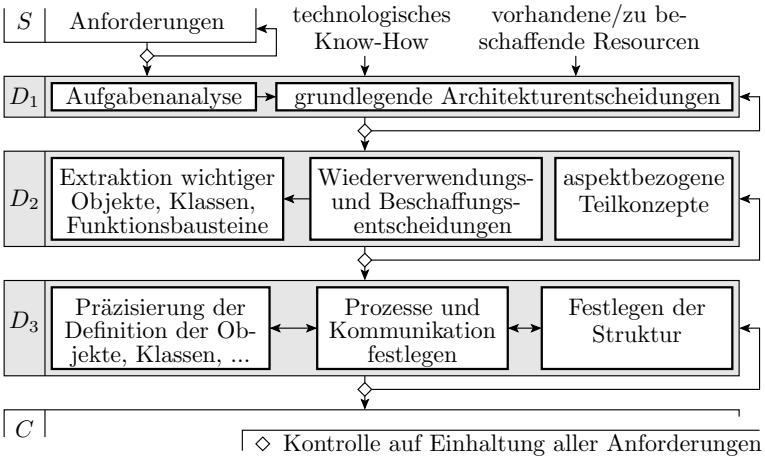
Entwurfstechnologie ist wesentliches Firmen-Knowhow und verantwortlich für die Fähigkeit eines Unternehmens, fehlerarme Produkte zu liefern (vergl. auch CMMI-Fähigkeitsstufen, Folie 2.127).

R, S Anforderungsanalyse, Spezifikation.

D_i, C Schritt *i* des Architektur- und Funktionsentwurfs, Codierung.

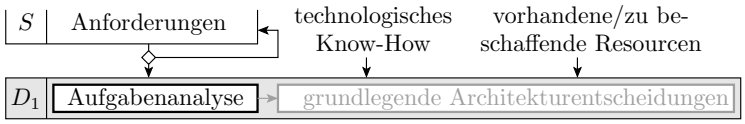
3.1 Lösungsfindung

7.65 Skizze für einen Projektablauf



Lösungsfindung zwischen Spezifikation (S) und Codierung (C). Dreistufiger Beispielprozess.

7.66 Aufgabenanalyse



Aufgabenanalyse:

- Wie lässt sich die Aufgabe lösen?
- Was braucht man dafür für Hardware, Entwicklungszeit?
- ...

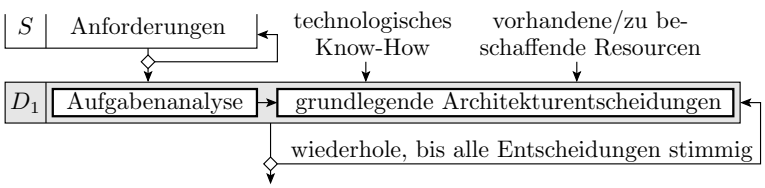
Dazu sind nicht nur die Anforderungen auszuwerten, sondern auch das vorhandene und erforderliche technologische Know-How:

- Erfahrungen mit ähnlichen Projekten,
- nachnutzbare Software-Bausteine und Tests,
- alte Projektpläne, ...

Ressourcen müssen geplant werden:

- Rechner, Software, Personal, ...

7.67 Grundlegende Architekturentscheidungen



Parallel zur Aufgabenanalyse erfolgen grundlegende Entscheidungen:

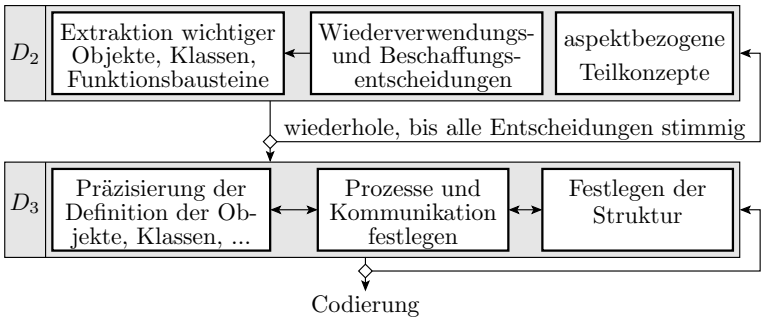
- Software-Architektur,
- File-System oder Datenbank, ...

- Wiederverwendung, Vergabe von Unteraufträgen, ...

Enthält Grundentscheidungen für Verlässlichkeit:

- Möglichkeiten für die Durchführung von Systemtests (Abschn. 5.2),
- Überwachung und Fehlfunktionsbehandlung (Abschn. 1.2.1),
- Unterstützung der Reifeprozesse (Abschn. 2.3.7),
- Reparierbarkeit, Ausfalltoleranz (Abschn. 6.6).

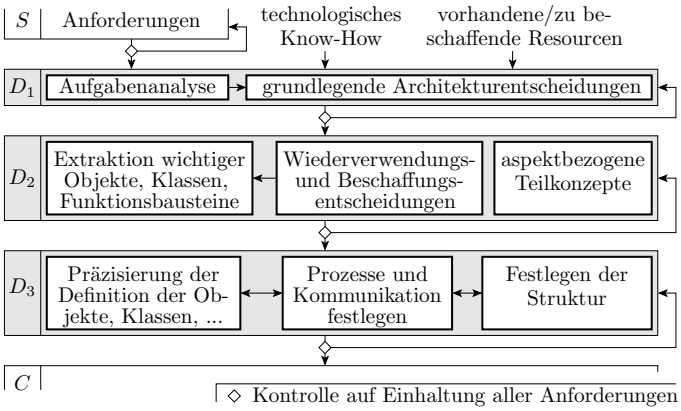
7.68 Objekte, Klassen, ...



Nach initialer Festlegung wichtigern Objekte, Klassen, Funktionsbausteine, ... incrementelle Verfeinerung:

- Objekte, Klassen, Module, Prozesse, Schnittstellen,
- Kommunikation, physikalische Struktur der Hardware.

Ergebnis ist eine in sich stimmige Aufteilung in Teilaufgaben.



Ein richtiger Projektplan ist umfangreicher, enthält Meilensteine, Verantwortlichkeiten, Kontrollen, ...

Das Vorgehen reift durch Auswertung der Projektpläne nach Abschluss. Bewährtes wird im nächsten Projektplan beibehalten und für beobachtete Probleme werden neue Wege gesucht.

7.70 Neue Trends

- BDD (Behavioral Drive Developement): Implementierung einer Zielfunktion von der Planung bis zur Codierung nach der anderen (vergl auch Spiralmodell plus Mikro-Kern-Architektur, Folie. 7.59, [Bec03], [Sma14], [EAD14]).
- TDD (Test Driven Developement): Beginn der Entwicklung neuer Funktionen mit Testbeispielen, die auch als eine Art Spezifikation gesehen werden [DD16].
- DevOps (Developement and Operation): Erweiterung der Vorgehensmodelle um Einsatzfreigabe und Reifeprozess, insbesondere auch die Werkzeugunterstützung dafür (Built-Prozess, Versionsverwaltung, ...)

[DD16] J. Davis and R. Daniels. Effective DevOps - Building a Culture of Collaboration, Affinity, and Tooling at Scale. Sebastopol: O'Reilly Media, Inc., 2016. isbn: 978-1-491-92642-0.

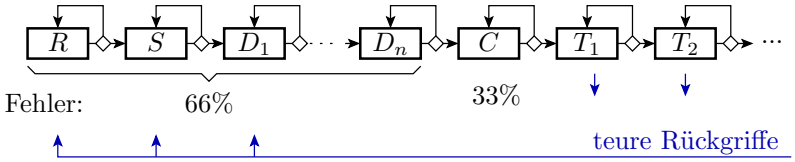
[Bec03] K. Beck. Test-driven Development - By Example. Boston: Addison-Wesley Professional, 2003. isbn: 978-0-321-14653-3.

[Sma14] J. F. Smart. BDD in Action - Behavior-driven development for the whole software lifecycle. Birmingham: Manning Publications, 2014. isbn: 978-1-617-29165-4.

[EAD14] F. Erich et al. Report: DevOps Literature Review. In: (Oct. 2014). doi: 10.13140/2.1.5125.1201.

3.2 Teure Rückgriffe

7.72 Teure Rückgriffe



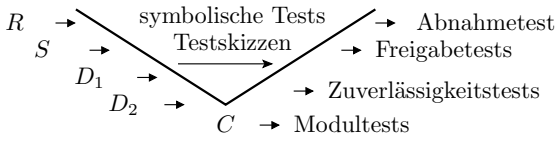
In den Vorcodierungsstufen entsteht ein erheblicher Anteil der Fehler, aber die Kontrollen (manuelle Inspektion) haben geringe Fehlerabdeckung (Abschn. 5.1), so das viele dieser Fehler erst in den Testphasen erkannt werten.

Mit jeder Stufe, die ein Fehler unbemerkt bleibt, vervielfachen sich der Nachbesserungsaufwand und die fehlerbezogenen Kosten.

R, S Anforderungsanalyse, Spezifikation.

D_i, C Schritt *i* des Architektur- und Funktionsentwurfs, Codierung.

7.73 V-Modell



Das V-Modell ordnet den Entwurfsstufen Teststufen zu:

- Anforderungen kontrolliert der Kunde mit Abnahmetests,
- die Einhaltung der Spezifikation der Hersteller mit Freigabetests,
- der Architektur- und Funktionsentwurf liefert Zusatzinformationen zum Code für die Modul- und Zuverlässigkeitstests.

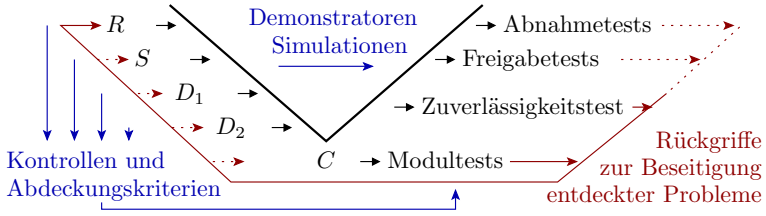
Es symbolisiert, dass für alle spezifierten Anforderungen und Entscheidungen Überlegungen erforderlich sind, wie bzw. dass später kontrollierbar (siehe prüfgerechter Entwurf ####).

Wie teure Rückgriffe vermeiden, wenn die ersten Entscheidungen im Entstehungsprozess den letzten Tests zugeordnet sind?

R, S Anforderungsanalyse, Spezifikation.

D_i, C Schritt i des Architektur- und Funktionsentwurfs, Codierung.

7.74 Minderung der Häufigkeit teurer Rückgriffe



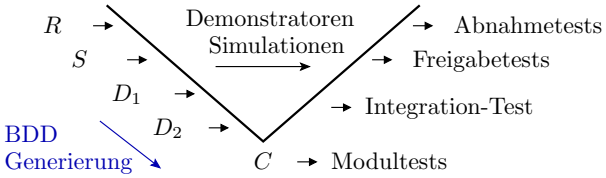
- Kontrolle früher Entwurfsentscheidungen mit Demonstratoren oder durch Simulationen.
- Abdeckung der Fehler aus frühen Entwurfsphasen überwiegend mit kurzen Modultests wärend und gründlich mit umfangreichen Zuverlässigkeitstests nach dem Entwurf.
- Freigabe- und Abnahmetest nur noch zur Kontrolle der Tests.

R, S Anforderungsanalyse, Spezifikation.

D_i, C Schritt i des Architektur- und Funktionsentwurfs, Codierung.

Demonstrator: Vereinfachte Impementierung zur Untersuchung und Demonstration der Machbarkeit.

7.75 Fehler vermeiden, schnell bis zum Test, ...



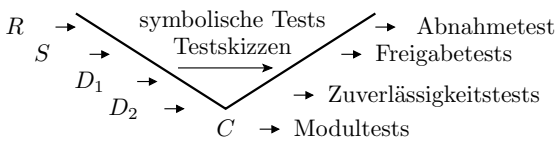
- Codegenerierung aus Anforderungsbeschreibungen. Verhalten schneller ausprobierbar. Fehlervermeidung durch Automatisierung und Neugenerierung nach Anforderungsänderungen.
- BDD: Funktionsweise Entwicklung bis Codierung und Test. Probleme schneller sichtbar, bevor sie weitere Funktionen »infiziert«.
- TDD: Beginn der Entwicklung neuer Funktionen mit Testbeispielen. Vermeidet Missverständnisse und spätere Testprobleme.

BDD Verhaltsgetriebene Entwicklung. Implementierung einer Zielfunktion von der Planung bis zur Codierung nach der anderen.

TDD Testgetriebene Entwicklung. Beginn der Entwicklung neuer Funktionen mit Testbeispielen, die auch als Spezifikation gesehen werden.

3.3 Testbare Anforderungen in UML

7.76 Testbare Anforderungen



Für die Vorcodierungsentwurfsphasen gilt als gut Praxis, getroffene Festlegungen so zu beschreiben, dass sich auch aus diesen **testbare Anforderungen** in Form von **Testabsichten** und **symbolischen Tests** ableiten.

Automatische Verarbeitung verlangt eine rechnerlesbare Anforderungsbeschreibung. Die Modellbeschreibungssprache UML ...

R, S Anforderungsanalyse, Spezifikation.

D_i, C Schritt *i* des Architektur- und Funktionsentwurfs, Codierung.

Testbare Anforderungen: Testabsichten, symbolische Tests, Abdeckungskriterien, Kontrollen.

Symbolischer Test: Beschreibungen mit Symbolen für Eingaben, Kontrollen und Zusicherungen.

Testsabsicht: Skizzenhafte Beschreibung der Testeingaben, -durchführung und Ergebniskontrolle.

7.77 UML

Graphische und rechnerverarbeitbare Darstellung unterschiedlicher struktureller, funktionaler und testspezifischer Aspekte:

- Klassendiagramm: Wesentliche Klassen, Methoden und deren Beziehungen untereinander. Für den Test Testobjektschnittstellen, zu initialisierender Objektraum, Wertebereiche für Testeingaben, Testausgabekontrollen (Abschn. 5.2.3, 5.4.3).
- OCL: Funktionale Sprache für Vorbedingungen, Invarianten und Nachbedingungen für Funktionsbausteine. Nutzbar für statische Tests, Zusicherungen und Testausgabekontrollen (Abschn. 5.4.1).
- UML-Funktionsmodelle (Aktivitätsdiagramme, Sequenzdiagramme, ...): Symbolische Tests, Abdeckungskriterien, Ablaufkontrollen.

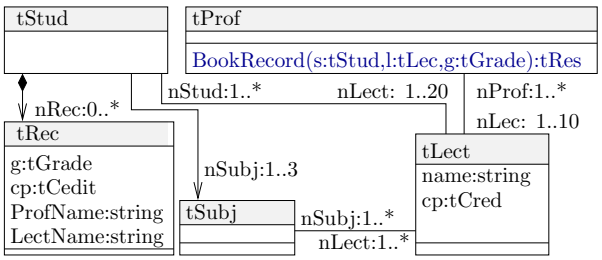
Auch geeignet für die Generierung von Code-Lückentexten (Aufwands- und Fehlervermeidung) und nach der Codierung als Referenz für Reviews auf vergessene Aspekte (statische Tests).

UML Grafische Modellierungssprache zur Spezifikation von Software-Teilen.

OCL Object Constraint Language. Modellierungssprache für zuzusichernde Eingenschaften.

Symbolischer Test: Beschreibungen mit Symbolen für Eingaben, Kontrollen und Zusicherungen.

7.78 Klassendiagramm (Beispiel Folie 5.49)



Klassendiagramme dienen im Entwurfsprozess zur Skizzierung wichtiger Klassen mit deren wesentlichen Datenstrukturen, Methoden und Beziehungen untereinander. Üblicherweise wird aus dem Klassendiagramm für die Codierung ein Lückentext generiert mit:

- Klassendefinitionen, Attributen,
- Aufrufschnittstellen der Methoden und
- Lücken für die Implementierung der Methoden.

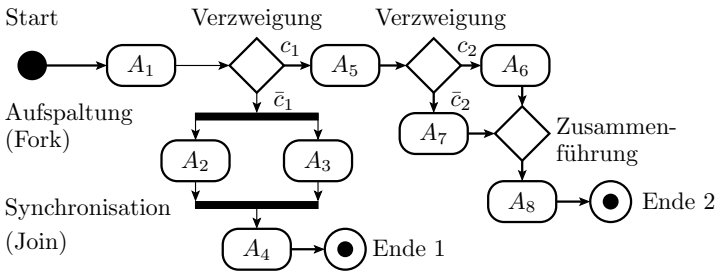
Die überarbeitete vervollständigte Klassenstruktur ist aus dem Code rückgewinnbar. Visualisierung der Änderungen in inspezierbarer Form aus Testsicht besser als Korrektur der UML-Beschreibung. Warum?

- Klassendiagramme beschränken sich auf wesentliche Schnittstellenparameter und Methoden. Eine vervollständigte Beschreibung verliert Fokus und Übersichtlichkeit.
- Umfangreicher und unübersichtlicher impliziert höherer Inspektionsaufwand und geringere zu erwartende Fehlerabdeckung.
- Ursprüngliche Absicht und Realisierung lassen eine gewisse Diversität erwarten, insbesondere in Bezug auf vergessene Aspekte.
- Inspektion der Änderungen gegenüber initialen Skizze erlaubt Hinterfragung der Gründe der Änderunge (Detaillierung, Verbesserung, vergessener Aspekt, ...).

Testbare Aspekte der Klassenstruktur: zu testende Methoden, Objekträume, Eingaberäume, ...

Diese Informationen lassen sich auch bzw. besser aus dem vervollständigten, gegen UML-Modell kontrollierten Code extrahieren.

7.80 Aktivitätsdiagramm



Aktivitätsdiagramme beschreiben Ablaufmöglichkeiten aus Aktivitäten, Transaktion, Verzweigung, Synchronisation, Signale senden und empfangen. Das Beispiel beschreibt drei Ablaufmöglichkeiten

E_1: `Start,` A_1`,` $A_2 \parallel A_3$`,` A_4`, Ende 1`
E_2: `Start,` A_1`,` A_5`,` A_7`,` A_8`, Ende 2`
E_3: `Start,` A_1`,` A_5`,` A_6`,` A_8`, Ende 2`

---

c_i, A_i Bedingungen, Aktivitäten.

7.81 Testbare Anforderungen

E_1: `Start,` A_1`,` $A_2 \parallel A_3$`,` A_4`, Ende 1`
E_2: `Start,` A_1`,` A_5`,` A_7`,` A_8`, Ende 2`
E_3: `Start,` A_1`,` A_5`,` A_6`,` A_8`, Ende 2`

Die Ablaufmöglichkeiten lassen sich betrachten wahlweise als:

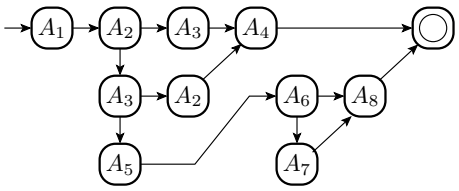
1. Äquivalenzklassen. Genutzte Eingaberäume mit ähnlicher Verarbeitung, die mit ausreichend vielen Tests abzudecken sind.
2. Symbolische Tests, für die ein oder mehrere konkrete Tests gefordert werden.
3. Abdeckungskriterien, instrumentierbar durch Zähler für die zu testenden Abläufe. Im Beispiele genügen Zähler für A_4, A_6 und A_7.

Aus allen Ablaufmöglichkeiten lassen sich Automaten generieren

1. zur Ablaufkontrolle auf Zulässigkeit (siehe nächste Folie) und
2. Würfel für (symbolische) Tests (siehe übernächste Folie).

7.82 Ablaufkontrolle auf Zulässigkeit

Ablaufmöglichkeiten:
$A_1, A_2 \parallel A_3, A_4;$
$A_1, A_5, A_7, A_8;$
$A_1, A_5, A_6, A_8;$

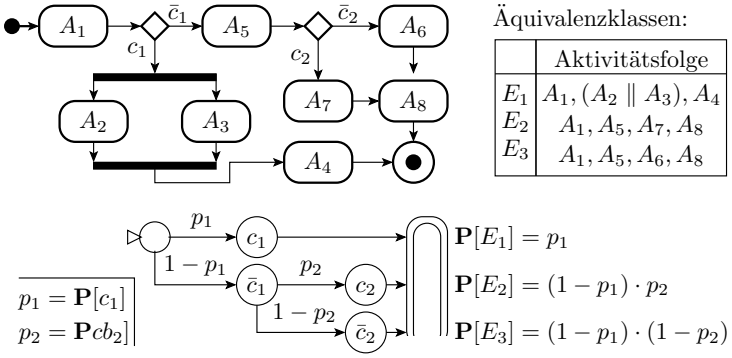


Abläufe sind Aktionsfolgen, die sich während eines Tests protokollieren, zählen und auch kontrollieren lassen. Betrachten wir jede Aktionen A_i als Zeichen, lässt sich die Menge der zulässigen Abläufe zu einem Wort einer formalen Sprache zusammenfassen, für das Beispiel:

$$\mathrm{OK} = A_1, (((A_2, A_3) \,|\, (A_3, A_2), A_4) \,|A_5, (A_6|A_7), A_8);$$

Aus der Sprachbeschreibung lässt sich ein Kontrollautomat für instrumentierte Kontrollausgaben der abgearbeitenen Aktivitäten generieren (Abschn. 5.5.3).

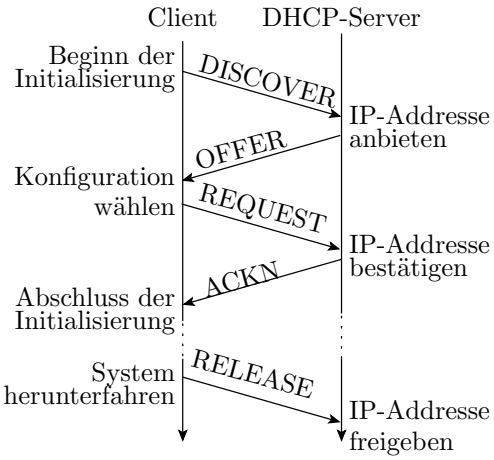
7.83 7.83 Würfel für zulässige Testeingaben



Der Automat zum Würfeln zufälliger Testeingaben muss nur die Bedingungen c_1 und c_2 auswürfeln. Wenn c_1 und c Eingabewerte, konkrete, sonst symbolische Tests (Folie 5.127).

Gleichhäufige Auswahl aus allen Äquivalenzklassen $\mathbb{P}[E_i] = 1/3$ verlangt im Beispiel Auswahlwahrscheinlichkeiten $p_1 = 1/3$ und $p_2 = 1/2$.

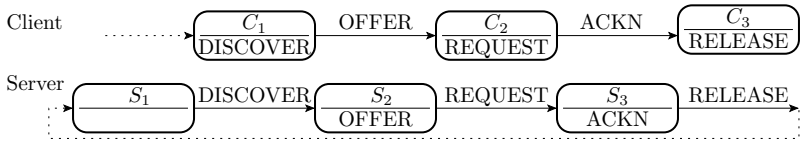
7.84 Sequenzdiagramm



Nachricht	Beschreibung
DISCOVER	Broadcast Clients, sucht Server
OFFER	Serverantwort mit Konfigurationsvorschlag
REQUEST	Braodcast Clients an bevorzugten Server Ablehnung aller anderen Server
ACKN	Server liefertIP-Adresse
NAK	Der Server lehnt die IP-Adresse ab
DECLINE	Der Server hat ein Problem mit der angebotenen IP-Adresse und lehnt ab
RELEASE	Client gibt IP-Adresse frei

Sequenzdiagramme sind Interaktionsdiagramme und zeigen Ablaufbeispiele für den Nachrichtenaustausch zwischen Objekten, Threads, Rechnern, ... in einer zeitlich begrenzten Situation. Dabei kann auch das Erzeugen und Entfernen von Objekten enthalten sein.

7.85 Testbare Anford. ähnlich Aktivitätsdiagr.

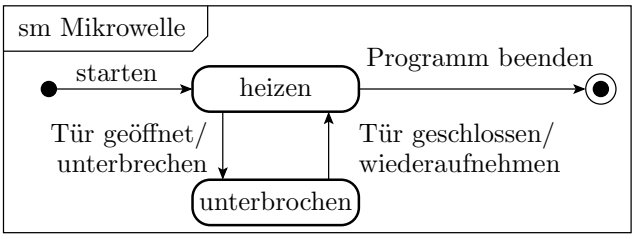


Ein Sequenzdiagramm beschreibt eine Aktionsfolge mit Nachrichten als Übergangsbedingungen, verteilt auf mehrere Akteure. Je Akteur eine Zustandsfolge mit Übergangsbedingungen und Ausgaben.

Die Zustandsfolgen beschreiben je Akteur eine Äqivalenzklasse bzw. einen symbolischen Test. Alle Sequenzdiagrammen zusammen liefern für jeden Teilnehmer die Menge alle Äuivalenzklassen und einen Zustandsautomaten zur Beschreibung gültiger Nachrichtenfolgen. Weiter transformierbar in Würfelfunktion für verarbeitbare Nachrichtenfolgen.

Für den Test nutzbare Infos wieder symbolische Tests, Abdeckungskriterien, Kontrollautomat auf Zuläs-. sigkeit und Nachrichtenwürfel.

7.86 Zustandsdiagramm

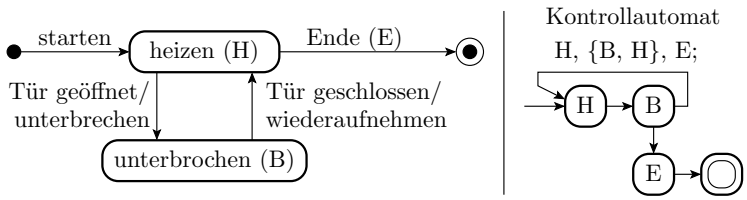


Ein Zustandsdiagramm (Verhaltenszustandsautomat, engl. behavioral state machine) beschreibt Funktionsabläufe durch:

- Zustände,
- Kanten mit Bedingungen für Zustandsübergänge und
- Zuständen und/oder Kanten zugeordnete Aktivitäten.

Im Vergleich zu Aktivitäts- und Sequenzdiagrammen werden andere Verhaltensaspekte hervorgehoben.

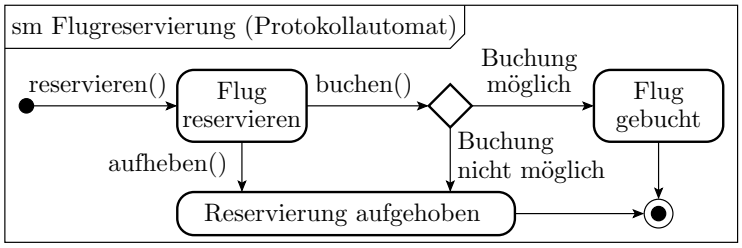
7.87 Testbare Anforderungen



Ableitbare testbaren Anforderungen:

- Typische und mögliche Abläufe und damit Äquivalenzklassen, symbolische Tests und Abdeckungskriterien,
- Kontrollautmat für Zustandsfolgen auf Zulässigkeit und
- Würfelfunktion für Testeingaben, im Beispiel Schaltfolgen der Start- und Endetaste und des Türkontakts.

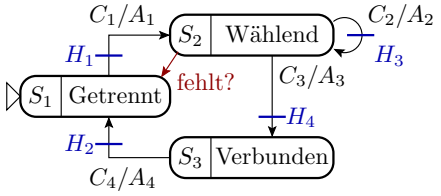
7.88 Protokollautomat



Ein Protokollautomat ist ein Kontrollautomat auf zulässige Aktionsreihenfolgen. Mögliche Aktionen im Beispiel sind die Methodenaufrufe »reservieren()«, »aufheben()« und »buchen()«. Aus dem Beispielautomaten geht hervor, dass ein Flug nur nach erfolgreicher Reservierung gebucht und dass ein einmal gebuchter Flug nicht gestrichen werden kann.

Ein Kontrollautomat ist schon der Checker auf zulässige , der aus den anderen Ablaufbeschreibungen erst generiert werden muss. Seinerseits geeignet zur Ableitung von Beispielabläufen, Abdeckungskriterien, ...

7.89 Fehlerfinden über testbare Anforderung



C/A Übergangsbedingung und Aktion
H_i Kantenabdeckungszähler

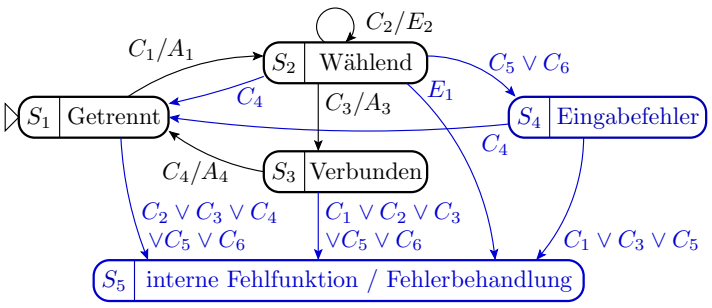
C_1	Abnehmen
C_2	Ziffer wählen
C_3	Rufnummer gültig
C_4	Auflegen
A_1	Rufnummer zurücksetzen
A_2	Ziffer zur Rufnummer hinzufügen
A_3	Verbindung aufbauen
A_4	Verbindung trennen
H_i	Abdeckungszähler

Automat für den Auf- und Abbau einer Telefonverbindung. Beschreibung als Automat. Äquivalenzklassen und symbolische Tests als EBNF-Zustandsfolgen, vergl. Folie 5.117:

- Pause: $S_1, \{S_1\}$;
- Anruf: $S_2, \{S_2\}, S_3, \{S_3\}$, Pause;
- Ungülige Nummer: $S_2, \{S_2\}$, Pause?, ...

Offenbar wichtige Äquivalenzklassen bzw. Tests nicht beschreibbar.

7.90 Vervollständigung des Automaten



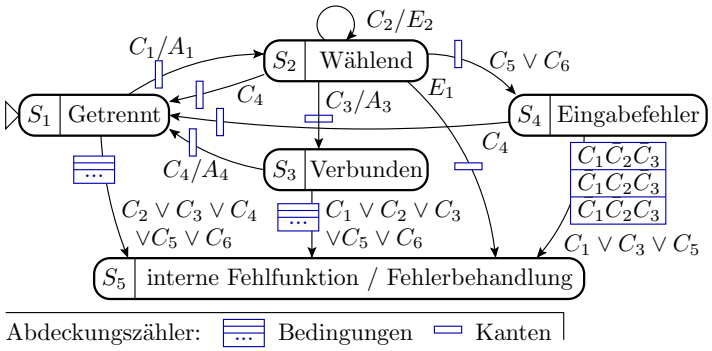
Bedingungen:

C_1	Abnehmen
C_2	Ziffer wählen
C_3	Rufnummer gültig
C_4	Auflegen
C_5	Rufnummer ungültig
C_6	Timeout

Aktionen:

A_1	Rufnummer zurücksetzen
A_2	Ziffer zur Rufnummer hinzufügen
A_3	Verbindung aufbauen
A_4	Verbindung trennen

7.91 Abdeckungskriterien



Jetzt sind auch die Betriebsarten (Äquivalenzklassen):

- falsche Nummer: $S_2, \{S_2\}, S_3$, Pause;
- Wahlabbruch: $S_2, \{S_2\}$, Pause;, ...

steuerbar. Für Kanten mit logisch verknüpften Bedingungen ist Instrumentierung von Bedingungsabdeckung möglich und zweckmäßig.

3.4 Programmierstil

7.92 Regeln für die Codierung (Good Practice)

Fehlerenstehungsrate manueller Code-Entwicklung ca. 10 bis 100 Fehler je 1000 NLOC. Minderung der Fehleranzahl vor und nach dem Test durch sog. Regeln »of Good Practice«:

- Einfach, ohne überflüssige Schnörkel. Gut testbar. Gut änderbar.
- Verzicht auf Code für eventuelle künftige Erweiterungen, weil das voraussichtlich toter Code wird.
- Ausnahme Schnittstellen, weil nachträgliche Schnittstellenänderungen viel Nacharbeit und neue Fehler bedeuten.
- Wenn man das dritte mal dasselbe Stück Code schreibt, ist es Zeit für die Auslagerung in eine Hilfsfunktion, weil dann etwa klar ist, wie diese aussehen muss.
- Tests immer nach dem Prinzip »Fail Fast« programmieren, d.h. mit strengen Kontrollen und Abbruch bei erkannter Fehlfunktion.
- Sorgfäliger Entwurf externer Schnittstellen auch mit Rücksicht auf künftige Verwendung.

- Größenbegrenzungen: Funktionen ≤ 30 NLOC, Programmdateien ≤ 500 NLOC, je schlechter testbar (z.B. nicht im Schrittbetrieb) um so kleiner und übersichtlicher.
- Fokus zuerst auf Korrektheit, dann erst auf Schnelligkeit.
- Codierung nur der benötigten Funktion statt Universallösungen mit einer Komplexität, die nicht erforderlich ist.
- Wenn ein Test versagt, zugrundeliegende Fehler sofort suchen und beseitigen.
- Zum Test der Tests sollte jeder Test einmal mit einem wohlüberlegten Bug im Testobjekt zum Versagen gebracht werden.
- Einhaltung der Architekturregeln. Die Nutzung von Funktionen unter Umgehung der Schichtenregung kann eine konkrete Lösung stark vereinfachen, aber die Folgeprobleme bei späteren Änderungen wird man nicht mehr los.
- ...

»7.94 Anti-Pattern«

Das sollte man vermeiden:

- Big ball of mud: Ein System ohne erkennbare Struktur.
- Eingabe-Hack: Unbehandelte ungültige Eingaben.
- Überladene Schnittstellen: Zu viel Schnittstellenballast kann die Implementierung extrem schwierig machen.
- Mit besseren Werkzeugen vermeidbare Programmierarbeit.
- Die Verwendung von Programmiermustern und Methoden, ohne sie zu verstehen.
- Benutzung von Konstanten ohne Erleuterung. ...

Die gelisteten Regeln und die Antipattern sind nur eine kleine Auswahl. Der Fokus liegt auf Übersichtlichkeit, Verständlichkeit, gut durchdacht, Arbeitsvermeidung, ... Schnittstellen haben eine Sonderstellung.

Außer allgemein anerkannten informalen Regeln gibt es auch standardisierte und automatisch überprüfbare Regeln.

7.95 MISRA-Standard

Die Programmiersprache C erlaubt sehr maschienennahe Programmierung und darüber schnelle und effiziente Programme. Preis »böse« Fehlermöglichkeiten, z.B. bei Nutzung Goto, Pointer, Heap, ...

MISRA: Über 100 Regeln für C-Programme für Automotive zur Vermeidung »böser« Fehler, zum Teil verpflichtend, zum Teil Empfehlungen:

- Bezeichnerlänge max. 31 Zeichen (längere Bezeichner werden von manchen Compilern nach 31 Zeichen abgeschnitten, Risiko, dass Compiler unterschiedliche Variablen zu einer zusammenfassen.
- Unterschiedliche Bezeichner für unterschiedliche Objekte:

```
int16_t i; {
 int16_t i; // Hier zwei Variablen i definiert.
 i = 4;
}
i = 3;       // Welchen Wert hat welche Variable i?
```

- Jeder Variablen ist vor ihrer Nutzung ein Wert zuzuweisen, ...

Achtung: Zu viele Regeln nerven und bewirken das Gegenteil.

7.96 Vermeidung unsicherer Konstrukte

Der Klassiker für Sicherheitslücken in C-Programmen ist die Verwendung der Bibliotheksfunktion

```
char *strcpy(char *dest, const char *src)
```

zum Kopieren von Eingabezeichenketten in einen Puffer auf dem Stack. Wegen der fehlenden Längenkontrolle lassen sich damit auf der Stack hinter dem Puffer Variablenwerte und Rücksprungadressen gezielt mit Eingabezeichen überschreiben (vergl. auch Folie 7.150).

Problemvermeidung durch statische Code-Analyse:

- Suche alle Aufrufe von strcpy(), die Eingabedaten in Puffer kopieren.
- Ersatz durch die gleichwertige Funktion mit Textlängenbegrenzung

```
char *strncpy(char *dest, const char *src,
              int n);
```

n – Puffergröße.

7.97 Refactoring

Refactoring

Strukturverbesserung von Quelltexten unter Beibehaltung des beobachtbaren Programmverhaltens zur Verbesserung der Lesbarkeit, Wartbarkeit, Testbarkeit, ...

Software-Entwicklung ist ein Lernprozess, bei dem sich immer wieder Entscheidungen nachträglich als ungünstig oder falsch herausstellen und geändert werden müssen.

Handisches Refactoring verlängert den Fehlerentstehungsprozess und erhöht die Fehleranzahl. Nach häufiger Überarbeitungen ist jeder Code zu schlecht und muss neu geschrieben werden.

Für einige Überarbeitungsregeln, z.B. die Nachbesserung von Bezeichnern, gibt es inzwischen Tool, die das fehlerarm erledigen. Das Vorgehensmodell »Extreme Programming« fordert z.B. extensive Nutzung automatisierter Refactorings.

Zusammenfassung

7.98 Lösungsfindung

Die Lösungsfindung für eine Software zwischen Spezifikation und Codierung ist ein Prozess zunehmender Verfeinerung mit Entscheidungen über mehrere Sichtweisen (Funktion, Struktur, Nachnutzung, Ressourcenplanung (Personal, Knowhow, ...), der zum Lernen aus Fehlern in ein Stufenmodell gepresst wird.

Beginn mit Analyse der Anforderungen, vorhandenes und erforderliches technologische Know-How, Ressourcen-Planung, ... Grundlegende Entscheidungen Software-Architektur, Systemtestdurchführung, Fehlfunktionsbehandlung, Wartung, Nachnutzung, Unteraufträge, ...

Extraktion wichtigen Objekte, Klassen, Funktionsbausteine, ...

Incrementelle Verfeinerung bis stimmige Aufteilung in Schnittstellen, Programmieraufgaben, Beschaffungen, Unteraufträge, ...

Das Vorgehen reift durch Auswertung der Projektpläne nach Abschluss. Bewährtes wird im nächsten Projektplan beibehalten und für beobach- tete Probleme werden neue Wege gesucht.

7.99 Teure Rückgriffe

In einem Stufenmodell entstehen zahlreiche Fehler in den Vorcodierungsstufen, die erst viel später beim Test oder erst im Einsatz entdeckt werden. Hohe fehlerbezogene Kosten. Prinzipien zur Minderung der Häufigkeit teurer Rückgriffe:

- Demonstratoren und Simulationen zum Ausprobieren von Entwurfsentscheidungen,
- Modul- und Zuverlässigkeitstests mit hoher Abdeckung, auch für Fehler aus den Vorcodierungsphasen,
- Verhaltens- und testgetriebener Entwurf,
- automatisierte Code-Generierung.

Hohe Fehlerabdeckung für Vorcodierungsphasen verlangt testbare Beschreibungen für Anforderungen zur automatisierten Gewinnung

- symbolischer Tests, Abdeckungskriterien,
- Kontrollmöglichkeiten, Würfelfunktionen für Testeingaben,
- ...

7.100 Testbare Anforderungen, Programmierung

Testbare Anforderungen aus UML-Beschreibungen:

Klassendiagramme: Generierung von Lückentexten für den Code. Generierung vermeidet Fehler. Rückgewinnnung aus dem fertigen Code und Inspektion der Änderungen auf »vergessene Aspekte«.

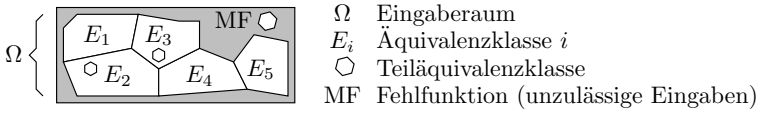
Testbare Aspekte: zu testende Methoden, Objekträume, Eingaberäume, ... besser aus dem fertigen kontrollierten Code extrahieren.

Aktivitätsdiagramme, Sequenzdiagramme und Automaten: Äqivalenzklassen, symbolische Tests, Abdeckungskriterien, Kontrollfunktionen auf zulässige Abläufe und Würfelfunktionen für Beispieleingaben oder symbolische Tests.

Fehlerarme Programmierung: Zahlreiche Regeln und Empfehlungen, informal und auch als Standards. Der Fokus liegt auf Übersichtlichkeit, Verständlichkeit, gut durchdacht, Arbeitsvermeidung, ... Schnittstellen haben eine Sonderstellung. C-typische Fehler und MISRA. Sicherheitslücken durch strcpy(). Refactoring.

4 Testabdeckung

7.101 Testauswahl für Software



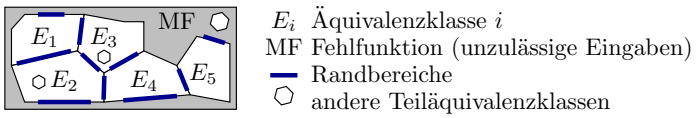
Selbst kleine Software-Bausteinen nutzen in der Regel nur ein sehr kleiner Teil des Eingaberaums. Zufällige Eingaben sind fast immer unzulässig, d.h. **Zufallstests** funktioniert **nur** sehr **eingeschränkt**.

Keine fehlerfreie Beschreibung des Sollverhaltens. Für fehlende Aspekte lassen sich nur schwache Erreichbarkeitskriterien angeben, die eine hohe Abdeckunganzahl verlangen. Die schwächsten Kriterien bestimmen den Testaufwand (Abschn. 2.1.4). Auch **fehlerorientierte Testauswahl** funktioniert **nur eingeschränkt**.

Zielführend für gründliche Tests mit hoher Fehlerabdeckung sind

- operations- und testprofilorientierte Auswahl bzw.
- Erreichbarkeits- (und Infektions-) Abdeckung mit $w \gg 1$.

7.102 Operations- und Testprofilabdeckung



Hinreichende Zuverlässigkeit verlangt für jede Äquivalenzklassen E_i einer ausreichenden Testanzahl

(2.53) $$N_i \geq h_{i.\max} \cdot N$$

Die Äquivalen
nachweisbare
abgedeckt wer

Für andere Te
ckungkriterien
Infektion und

$h_{i.\max}$ Maximale Nutzungshäufigkeit für alle potentiellen Operationsprofile.
N, N_i Erforderlich Gesamttestanzahl, Ausreichende Testanzahl für Äquivalenzklasse i.

7.103 Profil und Kritärienabdeckung

Operations- und Testprofilabdeckung rückführbar auf Abdeckung von

1. Erreichbarkeitskriterien (+Infektionskriterien) (Folie 2.26),

2. (1) + Ausbreitungskriterien,
3. symbolische Tests (Folie 7.81),
4. Mutationen (Fehlerinjektion) (Folie 2.23)

für den fertigen Code und testbare Anforderungen.

Bestimmungsaufwand von (1) bis (4) stark zunehmend. (3) hauptsächlich händische Testauswahl. (4) genauste Schätzung der Fehlerabdeckung zur Verfahrensbewertung und Kalibrierung von (1) bis (3) (Abschn. 2.1.4).

Testbare Anforderungen sind Skizzen funktionaler und struktureller Aspekte aus den Entstehungsphasen vor der Codierung (Äquivalenzklassen der Nutzungsprofile, Datenstrukturen, Funktionsbausteine, Beispielabläufe, Ausführungsbedingungen, ...) die sich im Code wiederfinden müssen und durch Erreichbarkeits- und Infektionszähler instrumentieren lassen (Folie 7.77).

7.104 Test- und Abdeckungsanzahl

Wiederholung: Schätzer der zu erwartenden Fehlerabdeckung:

1. Test mit $\#OP$ Operations- und Testprofilen, die unterschiedliche Fehler bevorzugen mit je N Tests. Im Idealfall:

   (2.54) $$\mu_{\mathrm{FC}} = 1 - \left(\frac{N}{N_0}\right)^{-\#OP \cdot K}$$

2. Abdeckung aller Kriterien mit $w \gg 1$ Tests:

   (3.21) $$\mu_{\mathrm{FC}}(w) = 1 - PR \cdot w^{-K_{\mathrm{K}}}$$

Zuverlässigkeit mit Fehlfunktionsbehandlung ohne Störungen:

(2.38) $$R \overset{(\geq 1)}{=} \frac{N}{K \cdot \mu_{\mathrm{F}} \cdot (1-\mu_{\mathrm{FC}}) \cdot (1-\mu_{\mathrm{MC}})}$$

Wenn sich die Modelle als tragfähig bestätigen, viele verschiedene Profile ($\#OP \gg 1$) oder große Abdeckungsanzahl je Kriterium ($w \gg 1$).

N_0, N	Bezugstestsanzahl, Anzahl der Tests mit Operationsprofil bzw. je Profil.
K	Formfaktor der Dichte der Fehlfunktionsrate ($0 < K < 1$).
$\#OP$	Anzahl der unterschiedlichen Profile, mit denen getestet wird.
w	Anzahl der Tests, die für jedes Abdeckungskriterium gesucht werden.
PR, K_{K}	Durchlassrate W1-Test, Formfaktor der Kriterienmenge.

7.105 Testauswahl

Wiederholung: Die Testauswahl startet z.B. mit einem Nutzerprofil oder ungewichteten Zufallseingaben und wiederholt, bis alle Kriterien ausreichend oft abgedeckt sind:

- Zufallsauswahl entsprechend Profil, für eine vorgegebene Testanzahl oder bis sich die Abdeckung nicht mehr verbessert,
- Suche eines neuen Testprofil, das noch unabgedeckte Kriterien extrem bevorzugt.

Der eingabesprofilorientierte Ansatz (1) bevorzugt eine feste Testanzahl je Profil und sucht jeweils für unzureichend abgedeckten Kriterien ein neues Profil, vergl. gewichteten Zufallstest Abschn. 6.3.4.

Der kriterienbasierte Ansatz (2) bietet mehr Flexibilität, auch unterschiedlich viele Tests je Profil, auch kriterienweise Suche, auch »ungenaue« Würfelfunktionen und Aussortieren von Testeingaben ohne Abdeckungsverbesserung (Folie 2.101).

Dieser Abschnitt mehr zu Abdeckungskriterien, Folgeabschnitt mehr zur Testprofil- und Testsuche.

4.1 Kontrollfluss

7.106 Kontrollfluss und Kontrollflussabdeckung

Erreichbarkeits- Infektions- und Ausbreitungsbedingungen lassen sich am einfachsten am Kontrollfluss eines Programms beschreiben bzw. sind letzendlich im Kontrollfluss zu instrumentieren.

Verzweigungsfreie Anweisungsfolgen werden zu Basisblöcken mit einem Eintrittspunkt und max. zwei Fortsetzungskanten zusammengefasst und daraus ein Graph mit allen Verweigungungs- und Schleifenmöglichkeiten gebildet (vgl. Compilerbau). Abdeckungsmodelle:

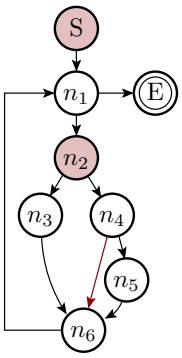
1. Anweisungsabdeckung,
2. Zweigabdeckung, schließt (1) ein,
3. Bedingsabdeckung, schließt (1 und 2) ein,
4. symbolische Tests.

Anweisungsabdeckung besagen, dass jede Anweisung, und Zweigabdeckung, dass jede Verzweigung $w \geq 1$–mal während der Tests ausgeführt werden muss. Beides mit Zählern im Kontrollfluss kontrollierbar.

Symbolische Tests siehe später Folie 7.109 und Bedingungsabdeckung siehe später ab Folie 7.113.

7.108 Beispielprogramm und Kontrollflussgraph

```
     int Ct_A, Ct_B, Ct_N;
S:   int CountChar(int Ct_max){
     char c;              ⇑Eingabe
     Ct_A=0; Ct_B=0; Ct_N=0;
n1:  while (Ct_N<Ct_max){
n2:   c=getchar();⇐Eingabe-Attrappe
      if (is_TypA(c))
n3:    Ct_A++;
n4:   else if (is_TypB(c))
n5:    Ct_B++;
n6:   Ct_N++;
     } //Schleifenende
E: }
```



Beispielprogramm zum Zählen der Zeichen vom Typ A und B. Eingaben: Gesamtanzahl bei Programmaufruf und die einzelnen Zeichen über die getchar()-Attrappe in Knoten S (vergl. Folie 7.38). Ergebniskontrolle (z.B. durch Inspektion) nach Programmabschluss im Knoten E.

7.109 Symbolische Tests

Aus dem Programmablauf lassen sich symbolische Tests in Form von Folgen auszuführender Anweisungen oder zu befriedigender Bedingungen ableiten:

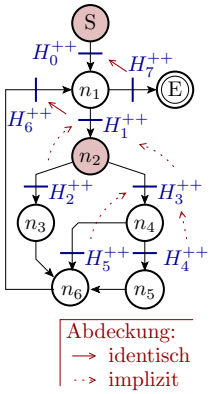
1. Eine Ausführung jeder Anweisung:
   S(2), n_1, n_2(TypA), n_3, n_4, n_6, n_1, n_2(TypB), n_4, n_5, n_6, n_1, E.
2. Eine Ausführung jeder Kante verlangt mindestens drei Schleifendurchläufe:
   S(3), n_1, n_2(TypA), n_3, n_4, n_6, n_2, n_1, n_2(TypB), n_4, n_5, n_6, n_1, n_2(weder noch), n_4, n_6, n_1, E.

Jeden symbolische Test ist mit $w \geq 1$ konkreten Tests abzudecken, z.B. mit bei Buchstaben für Typ A und Zahlen für Typ B:

	CT_max	Zeichenfolge	Sollwerte:	CT_A	CT_B	CT_N
1	2	"A1"		1	1	2
2	3	"A1!"		1	1	3

7.110 Instrumentierung von Kantenzählern

```
int H[8]={0,0,0,0, ...}; // Abdeckungszähler
...
S:  int CountChar(int Ct_max){
    char c; H[0]++;
    Ct_A=0;Ct_B=0;Ct_N=0;
n1: while (Ct_N<Ct_max){ H[1]++;
n2:  c=getchar();
     if (is_TypA(c)){
n3:   H[2]++; Ct_A++;}
n4:   else{
        H[3]++;
        if (is_TypB(c)){
n5:       H[4]++; Ct_B++; }
      }else H[5]++;
n6:  Ct_N++; H[6]++;
    } // Schleifenende
E:  H[7];}
```

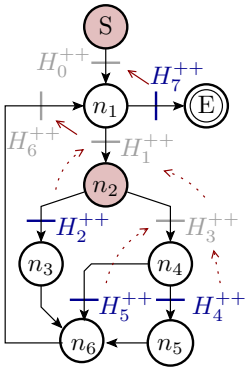


Abdeckungszähler je Kante. Bereinigung Abdeckungsidentität, ...

Für händische Auswahl weniger Tests ist der Weg über symbolische Tests anschaulicher. Für automatisierte Testsuche ist Würfeln und abhaken einfacher.

Ohne identisch und implizit abgedeckte Kriterien bleiben 4 Zähler übrig, für die der Test Mindestzählwerte erreichen muss.

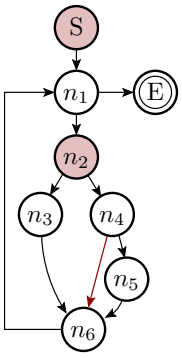
Mit Buchstaben für Typ A und Zahlen für Typ B zählt H_2 Buchstaben, H_4 Ziffern, H_5 sonstige Zeichen und H_7 Aufrufe. Einige zufällige Tests und die sich akkumulierenden Abdeckungszählwerte:



			Sollausgabe			Abdeckung			
	CT_max	Zeichenfolge	CT_A	CT_B	CT_N	H_2	H_4	H_5	H_7
1	2	"A1"	1	1	2	1	1	0	1
2	3	"A1!"	1	1	3	2	2	1	2
3	8	"axs*46.f"	4	2	8	6	4	4	3

7.112 Randwerte

```
     int Ct_A, Ct_B, Ct_N;
S:   int CountChar(int Ct_max){
     char c;
     Ct_A=0; Ct_B=0; Ct_N=0;
n1:  while (Ct_N<Ct_max){
n2:   c=getchar();
      if (is_TypA(c))
n3:    Ct_A++;
n4:   else if (is_TypB(c))
n5:    Ct_B++;
n6:   Ct_N++;
     } //Schleifenende
E: }
```



Intuitiv müsste das Programm auch mit Randwerten der Eingabeäquivalenzklassen ausprobiert werden, z.B. auch mit Ct_max=0 und den Grenzwerten für die Zeichenunterscheidung, weil sich da gern Fehler verstecken. Bei zufälligen Eingabeauswahl unwahrscheinlich.

7.113 Bedingungsabdeckung

```
      ...
n7:  if ((((A>10)&&(A<20))||((B>1)
          &&(B<85))) && (C==1)) {
n8:    ... }
      else {
n9:  ...   }
```

Bedingungsabdeckung platziert die Abdeckungskriterien bildlich an die Ränder der Entscheidungen, so dass kleine Werteänderungen jeder Entscheidungsvariablen die Entscheidung invertieren.

Verzweigungsbedingungen sind in der Regel logisch verknüpfte Vergleiche. Randwerte sind die Vergleichsschwellen und die um eins größeren bzw. kleineren Wert (Off-By-One-Fehler).

Für Gleitkomma-Variablen sind bei Vergleichen Rundungsfehler zu berücksichtigen. Wahl der Grenzwerte so, dass die zu erwartenden Verfälschungen durch Rundungsfehler nicht für eine zufällige Entscheidungsinvertierung ausreichen.

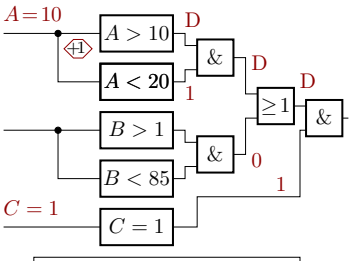
```
n7:  if ((((A>10)&&(A<20)) || ((B>1)&&(B<85)))
          && (C==1)) {
n8:    ... }
      else {
n9:  ...   }
```

Abzudeckende Kriterien der Beispielentscheidung:

Kriterium	A	B	C
A10	10	$B \leq 1$ oder $B \geq 85$	1
A11	11	$B \leq 1$ oder $B \geq 85$	1
A19	19	$B \leq 1$ oder $B \geq 85$	1
A20	20	$B \leq 1$ oder $B \geq 85$	1
B1	$A \leq 10$ oder $A \geq 20$	1	1
B2	$A \leq 10$ oder $A \geq 20$	2	1
B84	$A \leq 10$ oder $A \geq 20$	84	1
B85	$A \leq 10$ oder $A \geq 20$	85	1
C0	$((A > 10) \wedge (A < 20)) \vee ((B > 1) \wedge (A < 85))$		0
C1	$((A > 10) \wedge (A < 20)) \vee ((B > 1) \wedge (A < 85))$		1
C2	$((A > 10) \wedge (A < 20)) \vee ((B > 1) \wedge (A < 85))$		2

7.115 Instrumentierung von Abdeckungszählern

```
if (!((B>1) && (B<85)) && (C==1)){
  switch (A) {
    case 10: H[n1_A10]++; break;
    case 11: H[n1_A11]++; break;
    case 19: H[n1_A19]++; break;
    case 20: H[n1_A20]++; break;}
}
if (!((A>10) && (A<20)) && (C==1)){
  switch (B) {
    case  1: H[n1_B01]++; break;
    case  2: H[n1_B02]++; break;
    case 85: H[n1_B85]++; break;
    case 86: H[n1_B86]++; break;}
}
if (A>10)&&(A<20))||((B>1)&&(B<85){
 switch (C) {
    case  0: H[n1_C00]++; break;
    case  1: H[n1_C01]++; break;
    case  2: H[n1_C02]++; break;}
}
if (((A>10)&&(A<20))||((B>1)&&(B<85))&&(C==1)){ ... }
```

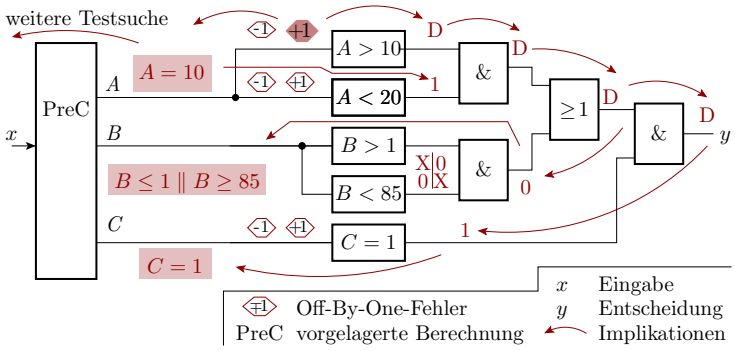


Nachweisähnlichkeit mit fehlerhaften Vergleichen.

Mit fehlenden Vergleichen nur gleiche Erreichbarkeitsbedingung.

H[xxx] Abdeckungszähler für Kriterium xxx

7.116 OFF-By-One-Fehlermodellierung



- Für jeden der fünf Vergleiche zwei Off-By-One-Fehler.
- Beseitigung identischer und redundanter Fehlerannahmen.
- Implikationsbestimmung, D-Algorithmus (Abschn. 6.2.2), ...

Ein bisschen Knowhow von digitalen Schaltungen nachnutzbar.

7.117 Entwicklungsstand, Forschungsziele

Selbst in sicherheitskritischen Anwendungen gelten **heute** noch **W1-Tests** (Abdeckung alle Kriterien mit mindestens einem Test) als ausreichend. Nach Avionic-Standard DO-178 B genügt*

- W1-Anweisungsabdeckung für nicht sicherheitskritische SW,
- W1-Zweigabdeckung für Software, die bedeutende Ausfälle verursachen kann und
- W1-Bedingungsabdeckung für flugkritische Software.

Eine hohe zu erwartende Fehlerabdeckung $\mu_{\mathrm{FC}} > 1-PR$ verlangt (3.22) $w \geq \left($

Test je Kriterium. w-fachen Aufwand, für **händische** Testerstellung **unbezahlbar**. Eingaben auswürfeln, Abdeckungskontrolle, ... sind **automatisierbar**. ...

W1-Test Einfache Testabdeckung ($w = 1$), entspricht aktuell üblichen händischen Tests.
* Zur Abwendung von Produkthaftung für Schäden durch durch nicht erkannte Fehler.
w Nachweisanzahl, Anzahl der Tests je Nachweiskriterium, Äquivalenzklasse, ...
PR, K_{K} Durchlassrate W1-Test, Formfaktor der Kriterienmenge.

4.2 Datenfluss

7.118 Datenfluss, Def-Use-Paare

Der Datenfluss beschreibt den Berechnungsablauf aus Datensicht, wie in Programmvariablen schrittweise aus Eingaben Ergebnisse gebildet werden. Grundbausteine des Datenflusses sind Def-Use-Paare:

- def(<Variable>): Basisblock mit Wertzuweisung an die Variable,
- p-use(<Variable>): Basisblock, der den Variablenwert für Verzweigungen auswertet. Eigentlich je ein Kriterium je abgehende Kante.
- c-use(<Variable>): Basisblock, der den Variablenwert für eine Berechnung nutzt.

Alle Def-Use-Paare bilden einen Graphen, nutzbar in zwei Richtungen:

- vorwärts: potentielle Weiterleitungspfade für Verfälschungen und
- rückwärts: potentielle Rückverfolgungspfade zu den Verfälschungsursachen (Folie 2.37).

Zusammenstellung der Def-Use-Paare enthält statische Tests:

- use ohne def: fehlende Variableninitialisierung und
- def ohne use: Redundante Berechnung.

7.119 Def-Use-Paare für ein Beispielprogramm

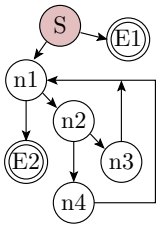
```
// Berechung  größter gemeinsamer Teiler
S:  int ggt(int a, int b){
      if(a == 0)      //def: a,b; p-use:a
E1:     return b;     //c-use: b
n1:   while(b != 0){  //p-use: b
n2:     if(a > b)     //p-use: a,b
n3:       a = a - b; //c-use: a,b; def: a
        else
n4:       b = b - a; //c-use: a,b; def: a
      }
E2:   return a;}      // c-use: a
```

def \ use	S	E1	n1	n2	n3	n4	E2
S: a, b	$b^{(p)}$	$a^{(c)}$	$b^{(p)}$	$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}$
n3: a	nr	nr		$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}$
n4: b	nr	nr	$b^{(p)}$	$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	

nr nicht erreichbar; $x^{(c)}$ c-use(x); $x^{(p)}$ p-use(x)

7.120 Umrechnung in Kontrollflusskriterien

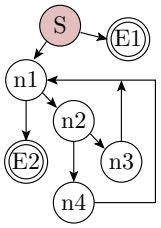
def \ use	S	E1	n1	n2	n3	n4	E2
S: a, b	$b^{(p)}$	$a^{(c)}$	$b^{(p)}$	$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}$
n3: a	nr	nr		$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}$
n4: b	nr	nr	$b^{(p)}$	$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	



1. def S → c-use E1: **Kante** S→E1
2. def S → p-use S: **Kanten** S→E1 und S→n1
3. def S → p-use n1: Pfade S→n1→E2 und S→n1→n2
4. def S → p-use n2: Pfade S→n1→n2→n3 und S→n1→n2→n4
5. def S → c-use n3: Pfad S→n1→n2→n3, **implizit (4)**
6. def S → c-use n4: Pfad S→n1→n2→n4, **implizit (4)**
7. def S → c-use E2: Pfad S→n1→E2, **implizit (3)**
8. def n3 → p-use n1: Pfade n3→n1→E2, n3→n1→n2
9. def n3 → p-use n2: Pfade n3→n1→n2→n3, n4→n2→n3→n4
10. def n3 → c-use n3: Pfade n3→n1→n2→n3, **impliziert (9)**, ...

7.121 Identität, Implikation, Redundanz

def \ use	S	E1	n1	n2	n3	n4	E2
S: a, b	$b^{(p)}$	$a^{(c)}$	$b^{(p)}$	$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}$
n3: a	nr	nr		$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}$
n4: b	nr	nr	$b^{(p)}$	$a^{(p)}, b^{(p)}$	$a^{(c)}, b^{(c)}$	$a^{(c)}, b^{(c)}$	



- Identität: P-Use gleicher Block und C-Use Folgeblöcke sind Zweigabdeckungskriterien.

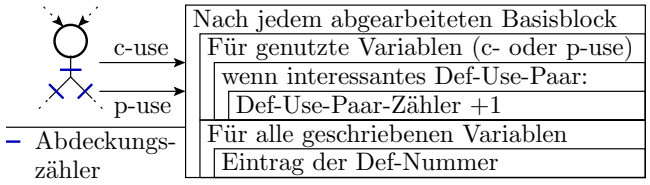
- Implikation z.B. def S → p-use n2 verlang die Abdeckung der Pfade von def S → c-use n3 und def S → c-use n4.
- Redundanz z.B. def n3 → c-use in S nicht erreichbar.

Verbleibende abzudeckende Def-Use-Pfade nach weglassen der redundanten, identisch abdeckbaren und implizit abgedeckten Pfade:

- def S → p-use n2: S→n1→n2→n3, S→n1→n2→n4
- def n3 → p-use n1: n3→n1→E2, n3→n1→n2
- def n3 → p-use n2: n3→n1→n2→n3, n3→n1→n2→n4
- ...

7.122 Instrumentierung Def-Use-Paar-Zähler

- Beseitigung aller »use ohne def« und aller »def ohne use«*.
- Definition von Variablen für die Def-Nummer aller Programmvariablen und Zähler für die abzudeckenden Def-Use-Pfade.



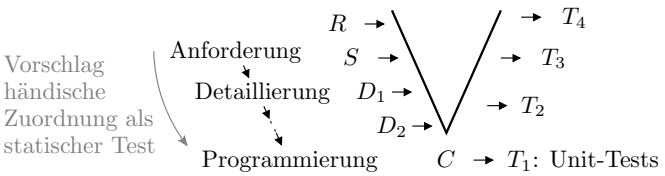
Instrumentierung und Eingabesuche** praktikabel, aber aufwändiger als für Zweigabdeckung. Dafür vermutlich höhere μ_{FC} bei gleicher Abdeckungsanzahl w. Aber, ob besser als Bedingungsabedeckung oder w erhöhen, keine Untersuchungen bekannt.

* Initialisierungfehler und ungenutzte Ergebnisse.

** Pfadsteuerung über mehrere Knoten verlangt gegenüber Pfadabdeckung die Befriedigung weiterer Bedingungen, ansonsten dieselben Suchalgorithmen.

4.3 Anforderungsabdeckung

7.123 Annotation testbarer Anforderungen

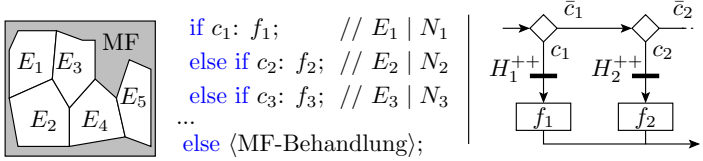


Das V-Modell sugeriert, in allen Entwurfsphasen testbare Anforderungen zu sammeln. In den Vorcodierungsphasen Operationsprofile für den Zuverlässigkeitstest sowie symbolische Tests, Funktionsbeispiele, ... Daraus gewinnbar Abdeckungs- und Kontrollkriterien, Würfel für verarbeitbare Eingaben, ...

Im weiteren sei unterstellt, dass testbare Anforderungen durch händische Annotation an den zugeordneten Code-Stellen maschinell verarbeitbar gemacht werden. Die Zuordnung ist gleichzeitig ein statischer Test, ob alle Anforderungen implementiert.

Würfel für verarbeitbare Eingaben: Zufallszahlengeneratoren für überwiegend zulässige Eingabewerte.

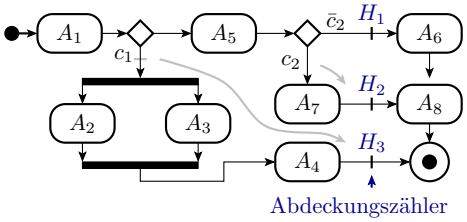
7.124 Abdeckungszähler für das Nutzungsprofil



Bei der Anforderungsanalyse entsteht u.a. eine Liste der wesentlichen Zielfunktionen, die in der Spezifikationsphase präzessiert wird. Vorschlag: Beschreibung als maschinenlesbare Tabelle von Bezeichnern für Äquivalenzklassen E_i und deren angestrebte Testanzahl N_i, optional ergänzt um Ausführungsbedingungen c_i.

Mit einer händischen Annotation der Eintritts- und/oder Endpunkte der den Äquivalenzklassen mit Namensverweis im späteren Programm (statischer Test auf vergessene Anforderungen) stehen alle Informationen für eine automatische Instrumentierung aller Abdeckungszähler incl. erforderliche Abdeckungsanzahl zur Testeingabesuche bereit.

7.125 Aktivitätsdiagramm, Automaten, ...



Äquivalenzklassen:

		Aktivitätsfolge	
E_1	c_1	$A_1, (A_2 \parallel A_3), A_4$	N_1
E_2	$\bar{c}_1 c_2$	A_1, A_5, A_7, A_8	N_2
E_3	$\bar{c}_1 \bar{c}_2$	A_1, A_5, A_6, A_8	N_3

Präzessierung der Spezifikation durch wesentliche Abläufe. Zusätzliche Äquivalenzklassen im Beispiel mögliche Aktivitätsfolgen.

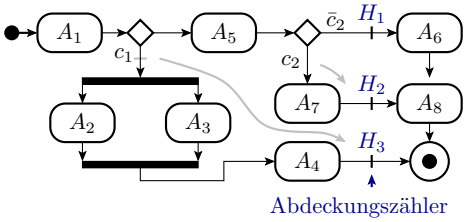
- Nutzungsprofilbeschreibung und -abdeckung wieder als maschinenlesbare Tabelle wie Folie zuvor.
- Abdeckungsanzahl aus gewünschter Zuverlässigkeit,
- Händischen Annotation der Bezeichner der Aktivitätseintritts- und/oder -endpunkte im fertigen Code.

---

c_i, A_i Bedingungen, Aktivitäten.

E_i, H_i Äquivalenzklasse i, Abdeckungszähler für Äquivalenzklasse i.

7.126 Testprofile



Äquivalenzklassen:

		Aktivitätsfolge	
E_1	c_1	$A_1, (A_2 \parallel A_3), A_4$	N_1
E_2	$\bar{c}_1 c_2$	A_1, A_5, A_7, A_8	N_2
E_3	$\bar{c}_1 \bar{c}_2$	A_1, A_5, A_6, A_8	N_3

Testprofile bevorzugen Eingaben / verlangen hohe Abdeckung für Systemteile mit schlecht nachweisbaren Fehlern, z.B.

- Aktivitäten mit komplizierter Realisierung und
- selten im Normalbetrieb eintretende Bedingungen.

Unter Nutzung von Infos aus den Vorcodierungsphasen:

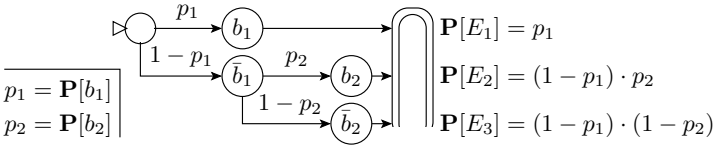
- Mindestabdeckung proportional zur erwarteten Code-Größe,
- In Anlehnung an Bedingungsabdeckung für Kontrollfluss Teilentscheidungsabdeckung (Folie 7.113), ...

7.127 Synergien

Zulässige Aktionsfolgen (Folie 7.82):

$$\mathrm{OK} = A_1, (((A_2, A_3) \,|\, (A_3, A_2), A_4) \,|A_5, (A_6|A_7), A_8);$$

Mit händischen Annotation der Bezeichner der Aktivitätseintritts- und/oder -endpunkte im fertigen Code Kontrollautomat auf Zulässigkeit der Abläufe hierzu automatisch instrumentierbar.



Aus Aktivitätsdiagrammen und anderen Ablaufmodellen ableitbaren Generatoren für Entscheidungsreihenfolgen, im Idealfall Testeingaben, (Folie 7.83) sind hilfreiche Bausteine für die Automatisierung Testeingabesuche.

7.128 Wesentliche Aussagen

Ablaufbeschreibungen (Wenn-Dann, Aktivitäts- und Sequendiagramm, Automat) bietet Abdeckungskriterien für Kanten, Aktionen, ... die sich händisch den Code-Elementen, die sie realisieren zuordnen lassen.

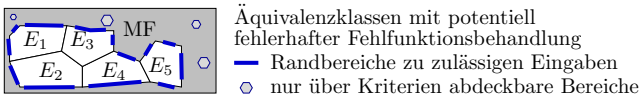
Diese Ablaufbeschreibung sind nutzbar für die Definition von Nutzungsprofilen, symbolische Tests, Würfel für symbolische Tests, ... Über Code-Zuordnung Weg für automatisierte Testerzeugung vorbereitet.

Testsuch lässt sich allg. auf die Suche konkreter Eingaben für Ablaufbedingungen zurückführen. Je nach dem, wie und woraus die Bedingungen gebildet werden, ist Automatisierung denkbar oder nicht. Das Zauberwort wird hier prüfgerechter Entwurf heißen, also System so entwerfen, dass die Tool, die es mal dafür geben wird, funktionieren.

Das Automatenbeispiel hat gezeigt, dass eine Fokusierung während er Anforderungsformulierung auf testbare Äquivalenzklassen, symbolische Tests, ... eine gute Technik ist, Unvollständigkeiten und andere Fehler offen zu legen.

4.4 Zusicherungen, NPB

7.129 Zusicherungen, Vorbedingungen



Der Bereich MF für unzulässige Eingaben enthält auch in der Regel kleine Unterbereiche mit zugeordneten Fehlern. Die Sollreaktion in diesen Bereichen ist Fehlfunktionsbehandlung.

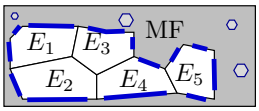
Zum Schutz vor potentiell sicherheitsgefährdenden Folgen dieser Fehler werden im Code **Zusicherungen** mit Programmabbruch bei Verletzung von **Vorbedingungen** einprogrammiert, auch für Einzeloperationen:

- Wertebereichüberlauf, Division durch null,
- unzulässige Feld-Indizes, ...

Für nicht über Zusicherungen garantierte Vorbedingungen ist eine **Instrumentierung** der **Kontrollen** während des Tests zielführend. **Erhöhung** der Fehlfunktions- und damit der **Fehlerabdeckung**.

NPB Bekannte Probleme mit Fehlerbeschreibungen, manuell oder automatisch erstellt.

7.130 Tests für Zusicherungen



Äquivalenzklassen mit potentiell fehlerhafter Fehlfunktionsbehandlung
— Randbereiche zu zulässigen Eingaben
⬡ nur über Kriterien abdeckbare Bereiche

Für (instrumentierte) Zusicherungen soll die Testsuche die Unverletzbarkeit nachweisen. Bei Zusicherungsverletzung, Fehler bzw. Phantomfehler (fehlerhafte Kontrolle) beseitigen.

Kein Test gefunden deutet auf »nur Fehler mit geringer Fehlfunktionsrate oder »Unverletzbarkeit«. Der Beweis der **Unverletzbarkeit** macht eine **Zusicherung** für die Nutzerübersetzungen **überflüssig**.

Für einprogrammierte Zusicherungen mit Abbruch, sollte mindestens **ein Test mit instrumentierter Werteverfälschung** den kontrollierten Abbruch kontrollieren.

7.131 Datenerfassung für erkannte Probleme

In einem Reifeprozess wird ein System von vielen Anwendern mit vielen Service-Anforderungen genutzt. Dabei fallen unzählige Problembeschreibungen an mit unterschiedlicher Dringlichkeit und Brauchbarkeit für die Fehlerbeseitigung:

1. Absturz, mit Aufrufstack und den weiter erforderlichen Daten für die Lokalisierung,
2. Fehlfunktionsabbruch mit Zusicherungsnummer und den weiter erforderlichen Daten für die Lokalisierung,
3. offenkundiges Fehlverhalten ohne Lokalisierungsdaten oder
4. mögliches Fehlverhalten zur weiteren Beobachtung.

Der Hersteller sortiert Problembeschreibungen in Schubladen mit vermuteter gleicher Ursache, setzt Prioritäten für die Beseitigung, sucht einen Test, der das beobachtete Fehlerverhalten reproduzierbar anregt, Experimentelle Reparatur, bis der Test durchläuft und Einpflegen der Änderung in die nächste freigegeben Version (Folie 2.83).

7.132 Automatisierte Testsuche

Moderne System generieren und übermitteln Problembeschreibungen, selbst. Ermöglicht eine Automatisierung der Testsuche der Form:

Für mehrfach* beobachtete Probleme:
Wenn aus den mitgelieferten Daten ein Test erzeugbar ist, der das Problem reproduzierbar nachweist:
händische Fehlerbeseitigung nach dem Prinzip der experimentellen Reparatur
Sonst:
Instrumentierung weiterer Zusicherungen und Datenfassungen in der Folgeversionen, die zusätzlich zur Testsuche gebraucht werden

Bei informal von Nutzern beschrieben Problemen können auch mehrere Iterationen über mehrere Folgeversionen erforderlich sein: Einfügen zusätzlicher Zusicherungen, bis automatisch Problemmeldung geliefert werden, schrittweise Erweiterung der Datenerfassung, ...

* Mehrere Problemberichte je Schublade. Nicht wiederkehrende Probleme haben oft Störungen als Ursache, z.B. Umplappen gespeicherter Bits durch radioaktiven Zerfall.

7.133 Umgang mit Zusatzkontrollen

Die zusätzlich instrumentierten Kontrollen für die Fehlereingrenzung dienen nur zur Datenerfassung für die Testgenerierung, verlangen Datenübermittlung an den Hersteller, aber keinen Fehlerabbruch beim Nutzer.

Enfernung instrumentierter Zusatzzusicherungen und Datenerfassungen nach erfolgreicher Problembeseitigung.

In heutigen Veröffentlichen zur Fehlerbeseitigungsiterationen und Fuzz-Tests, ... wird immer betont, dass nur schwerwiegende und sicherheitskritische Fehler beseitigt werden. Das ist wie bei den W1-Tests eine von der Ökonomie erzwungene Zuverlässigkeits- und Sicherheitsminderung, die nur durch mehr Automatisierung beseitigt werden kann.

Wenn in Zukunft die Testautomatisierung beherrscht wird, werden Zuverlässigkeits- und Sicherheitsabstriche bei IT-Systemen durch nicht beseitigte erkannte Probleme nicht mehr akzeptabel sein.

Zusammenfassung

7.134 Software-Test

Zielführend für gründliche Software-Tests sind

- die Operations- und testprofilorientierte Auswahl bzw.
- die Abdeckung von Erreichbarkeits- und Infektionskriterien mit je einer großen Testanzahl.

Zwei verwande Sichtweisen, Testvollständigkeit zu beschreiben. Ineinander überführbar. Statistisch untermauerte Modelle für die erforderlichen Anzahlen der Tests und für die Kriterienabdeckung.

Testauswahl:

- Start ungewichtet oder mit operationstypischer Wichtung,
- bis alle Vollständigkeitskriterien erfüllt sind:
  - viele Tests entsprechend Profil und Abdeckungssichtweise,
  - Profilanpassung an noch nicht abgedeckte Kriterien.

7.135 Kontroll- und Datenflussabdeckung

Aus dem Kontrollfluss ableitbare Abdeckungskriterien:

- Anweisungabdeckung, Zweigabdeckung und
- Bedingungsabdeckung

mit mindestens $w > 1$ Tests. Bedingungsabdeckung platziert die Kriterien bildlich an die Entscheidungsränder, so dass kleine Werteänderungen je Entscheidungsvariablen die Entscheidung invertieren.

Selbst in sicherheitskritischen Anwendungen gelten heute noch W1-Tests (mindestens eine Abdeckung je Kriterium) als ausreichend.

Hohe Fehlerabdeckungen $\mu_{\mathrm{FC}} \gg 90\%$ verlangen $w \gg 1$, also die vielfache Testanzahl. Nur mit weitgehender Automatisierung bezahlbar.

Datenflussabdeckung, heute synonym mit Def-Use-Paar-Abdeckung, umrechenbar in abzudeckende, nacheinander zu durchlaufende Kontrollflusspfade. Im Grunde strengere Kontrollflusszweigkriterien. Ob nützlich, aktuell keine belastbaren Untersuchungsergebnisse.

7.136 Anforderungsabdeckung (Folie 7.128)

Ablaufbeschreibungen (Wenn-Dann, Aktivitätsdiagramm, ...) bietet Abdeckungskriterien für Kanten, Aktionen, ... die sich händisch den Code-Elementen, die sie realisieren zuordnen lassen.

Diese Ablaufbeschreibung sind nutzbar für die Definition von Nutzungsprofilen, symbolische Tests, Würfel für symbolische Tests, ... Über Code-Zuordnung Weg für automatisierte Testerzeugung vorbereitet.

Testsuch lässt sich allg. auf die Suche konkreter Eingaben für Ablaufbedingungen zurückführen. Je nach dem, wie und woraus die Bedingungen gebildet werden, ist Automatisierung denkbar oder nicht. Das Zauberwort wird hier prüfgerechter Entwurf heißen, also System so entwerfen, dass die Tool, die es mal dafür geben wird, funktionieren.

Das Automatenbeispiel hat gezeigt, dass eine Fokusierung während er Anforderungsformulierung auf testbare Äquivalenzklassen, symbolische Tests, ... eine gute Technik ist, Unvollständigkeiten und andere Fehler offen zu legen.

7.137 Zusicherungen, erkannte Probleme

Zusicherungen sind Kontrollen für Vorbedingungen, auch für Einzeloperationen, z.B. Wertebereich, Indexbereich, ... Für potentielle Gefährdungen als Anweisungen mit Fehlfunktionsbehandlung, sonst instrumentierbar als zusätzliche Kontrolle während der Tests.

Für die Testsuche sind Zusicherungen Abdeckungskriterien, für die Tests gesucht, aber keine Tests gefunden werden sollen. Für gefundenen Tests, (Phantom-) Fehler beseitigen. Für einprogrammierte Zusicherungen mindestens ein Test mit instrumentierter Verfälschung.

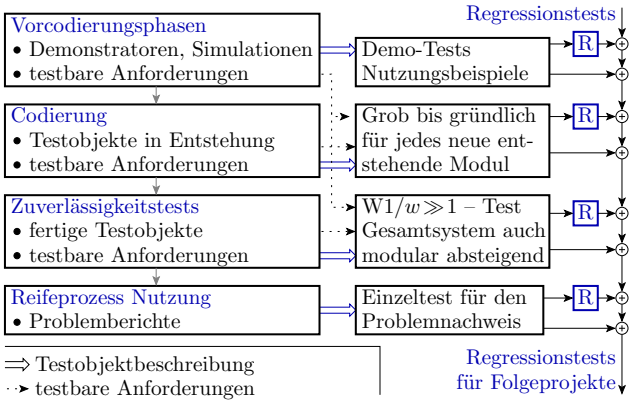
Automatische Generierung und Übermittlung von Problemberichten in Reifeprozessen erlaubt eine Automatisierung der Testgenerierung für beobachtbare Probleme. Wenn aus aktuellen Problemberichten kein Test erzeugbar, Erweiterung der Kontrollen und Datenerfassungen für die Folgeversion, bis aus den Problemberichten Tests ableitbar.

Wenn in Zukunft die Testautomatisierung beherrscht wird, werden Zuverlässigkeits- und Sicherheitsabstriche bei IT-Systemen durch nicht beseitigte erkannte Probleme nicht mehr akzeptabel sein.

5 Testinfrastruktur

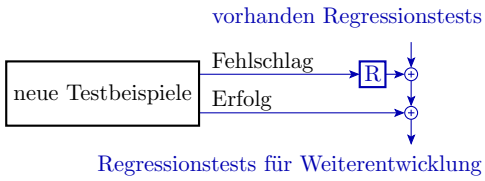
5.1 Anforderungen

7.138 Erstellung dynamischer Tests?



- Vorcodierungsphasen: spezifizierte Nutzungsbeispiel, ...
- Während der Codierung für jeden neuen Baustein.
- Nach der Codierung Zuverlässigkeitstests mit hohe Abdeckung.
- Im Einsatz für erkannte und zu beseitigende Probleme.

7.139 Regressions- und Reparaturtests



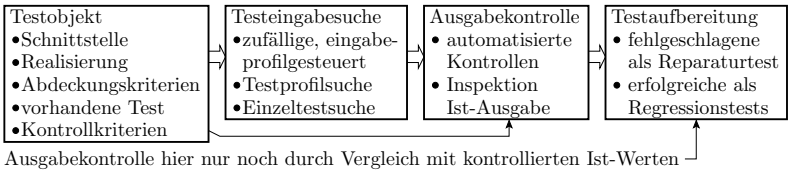
Akkumulation von Regressionstests (⊕): Erfolgreiche Test werden gesammelt und nach jeder Änderung als Regressionstest wiederholt. Wenn ein Regressionstest nach einer Änderung fehlschlägt, (Phantom-) Fehlerbeseitigung.

Fehlerbeseitigung (R): Fehlgeschlagene Tests sind als Einzeltests für die experimentelle Reparatur aufzubereiten. Nach jedem Fehlerbeseitigsversuch, wenn Fehlschlag Rückbau, sonst Änderungsübernahme und Testübernahme als Regressionstests (Folie ##).

Erfolgreicher Test: Test ohne erkennbares Fehlverhalten.

Fehlgeschlagener Test: Ein Test, der einen Fehler oder Phantomfehler aufdeckt.

7.140 Datenfluss und Ablauf der Testerzeugung



Die **Eingabesuche mit** Kriterienabdeckung benötigt für das Testobjekt Schnittstellebeschreibung (incl. erforderlicher Attrappen), übersetzbare Realisierung, **Abdeckungskriterien** und, wenn vorhanden, Regressionstests. Suchansätze später Abschn. 7.5.3.

Für **gefundene Tests** Abarbeitung **mit** den instrumentierten **Kontrollen ohne Abdeckungskriterien** zur Kontrolle auf erkennbare Fehler. Optional **händische Inspektion** der **Ist-Ausgaben**, die Sollwerte werden.

Für **Reparatur- und Regressionstest** genügt **Soll/Ist-Vergleich** mit den als korrekt klassifizierten Ist-Ausgaben, d.h. Compilierung **ohne** instrumentierte **Kontrollen und Abdeckungskriterien**.

7.141 Werkzeuge

Automatisierte Testerzeugung und -abarbeitung verlangt aufeinander abgestimmte Werkzeuge:

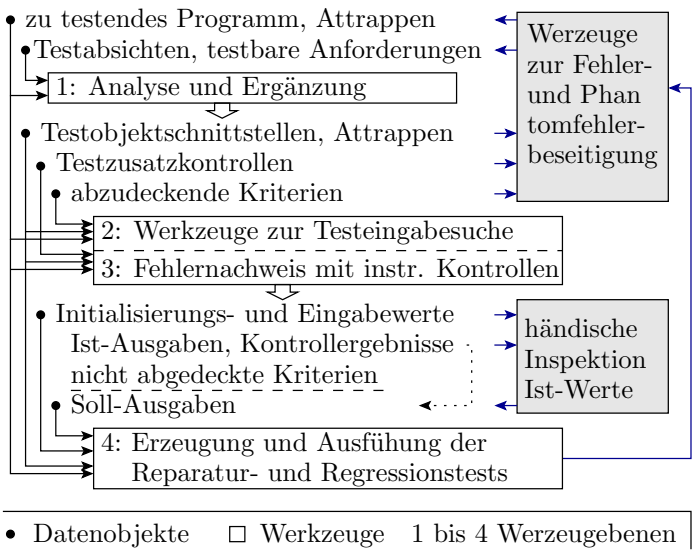
- Extraktion der Testobjekte, Abdeckungs- und Kontrollkriterien aus dem Quellcode und Beschreibungen für Testanforderungen,
- Testobjekt-Compilierung mit unterschiedlicher Instrumentierungen,
- Generierung unterschiedlicher Rahmen für die Testsuche und -ausführung, ...

Aufteilung in Werkzeugebenen:

1. Zusammenstellung Testobjekte, Abdeckungs- und Kontrollkriterien.
2. Suche von Testeingaben für ausreichede Kriterienabdeckung.
3. Kontrolle auf Fehlernachweis mit instrumentierten Kontrollen.
4. Reparatur- und Regressionstests mit Soll/Ist-Vergleich.

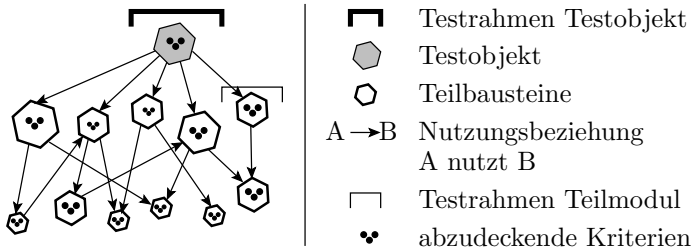
Zusätzlich Werkzeuge zur Ergebnisvisualisierung und händischen Nachbesserung, insbesondere zur Unterstützung der Ist-Ausgeninspektion und , Fehler- bzw. Phantomfehlerbeseitigung.

7.142 Incl. Datenobjekte und Verarbeitungsfluss



5.2 Extraktion der Testobjekte

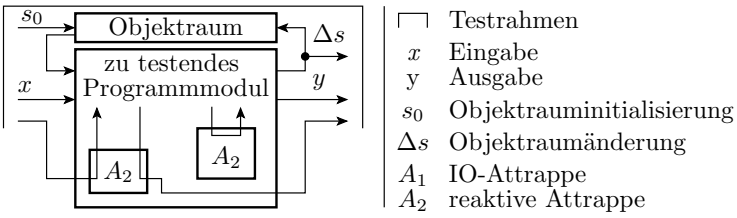
7.143 Extraktion der Testobjekte



- Testobjekte sind übersetz- und ausführbare Programme, die für den Test in eine **Service-Struktur gepresst** werden müssen*.
- In der **Codierungsphase** jeweils nach Erstentwurf und Änderung eines jeden Bausteins. Abdeckung grob bis gründlich (Folie 2.29).
- Nach der Codierung **Zuverlässigkeitstest** mit Abdeckung W1, vorgeschlagen Erhöhung auf $w \gg 1$, auch hierarchisch absteigend.

* Voraussetzung für reprduzierbare Testergebnisse und experimemtelle Reparatur.

7.144 Attrappen



Ein- und Ausgaben, Warteanweisungen auf externe Ereignisse, ... sind für den Test zur Gewähleitung von Determinismus durch Attrappen zu ersetzen (siehe getchar()-Attrappe Folie 7.38):

- IO-Attrappen: Weiterleitung der Attrappeneingaben als Testausgaben und Bereitstellung der Attrappenausgaben als Testeingaben.
- Reaktive Attrappe: Rückgabeberechnung aus Übergabewerte, in der Regel nur für ausgewählte Betriebsfälle und Eingaben.

Automatisierte Testerzeugung und Durchführung schließt automatisierte Attrappeneinbindung und soweit möglich auch Erzeugung ein.

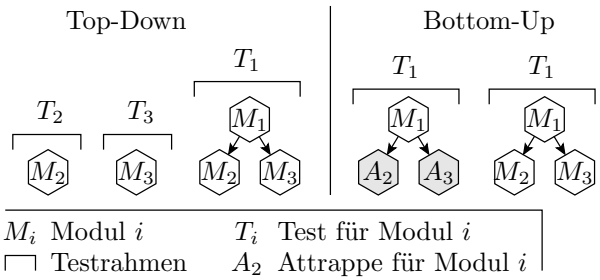
7.145 Attrappenerzeugung, Synergien

Für IO- und Zeitablaufsteuerfunktionen des Betriebssystems erscheint es zielführend, eine Attrappenbibliothek für die Testautomatisierung mit allen benötigten Zusatzangaben bereitzustellen.

Keine vorentworfene Attrappen verfügbar:

- IO-Attrappen bei Analyse aus Aufrufschnittstelle neu generierbar.
- Für reaktive Attrappen nur Lückentext mit Schnittstelle generierbar. Händische Implementierung verlangt Aufbewahrung als Quellcode.
- Reaktive Attrappen sind vereinfachte Implementierungen für ausgewählte Betriebsfälle. Erfordert für die Testsuche ein Signal an den Testrahmen, ob Attrappeneingabe gültig.
- Die Ein- und Ausgaben der Attrappen können wärend des Tests aufgezeichnet und als Regressionstests für die ersetzten Module aufbewahrt werden.
- Reaktive Attrappen sind Checker für die ersetzten Module.

7.146 Entwurfsbegleitende Test

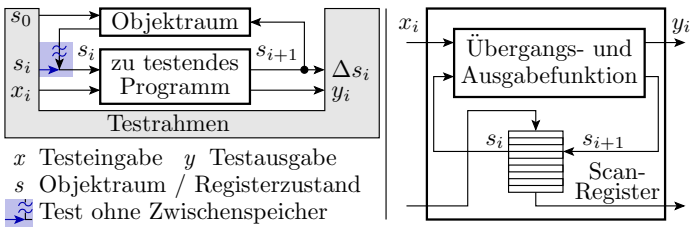


Module sind nach Entwurf und Änderung immer zeitnahe zu testen:

- Botton-Up: Entwurfsreihenfolge von unten nach oben. Alle Module benötigen einen eigenen Testrahmen und Testbeispiele.
- Top-Down: Entwurfbeginn mit den oberen Modulen. Noch nicht entworfene Teilmodule benötigen Attrappen.

In der Codierungsphase viele Nachbesserungen. Wegen Wirtschaftlichkeit nur aufwandsbeschränkte grobe bis gründliche Tests (Folie 2.29).

7.147 Objektrauminitialisierung



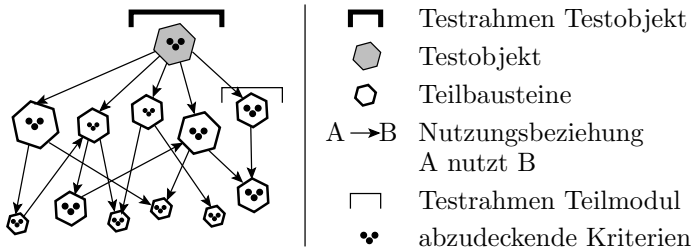
Modultests müssen den Objektraum initialisieren und können ihn dann mit **einem** oder **einer Folge** von Tests bearbeiten. **Einzeltests**

- zweckmäßig für fehlgeschlagene Tests zur Fehlerbeseitigung durch experimentelle Repartur und
- die Testsuche durch Pfadsensibilisierung, vergl. D-Algorithmus für digitale Schaltungen (Folie 6.65).

Testfolgen mit Objektrauminitialisierung nur am Anfang:

- schnellere Testausführung, weniger aufzubewahrende Testdaten,
- Voraussetzung für Fehlernachweis durch Abstürzen (Folie 7.162).

7.148 Zuverlässigkeitstests



Innerhalb der Codierungsphase wird insgesamt jedes Modul einzeln und und im Verbund grob bis maximal gründlich getestet. Dabei werden allen schwern Fehler, die Funktion immer oder oft beeinträchtigen gefunden und nach dem Pareto-Prinzip auch die überwiegende Mehrheit der Fehler. Nachweisbare Fehler werden beseitigt. Die verbleibende Fehler verursachen einzeln und auch zusammen nur moderate bis geringe Fehlfunktionsraten unverlangen Tests hoher Kriterienabdeckung:

- nach aktueller »best practice« W1 und
- für hohe Zuverlässigkeit ab der ersten freige d haben geringe Fehlf.

#

werden damit ist jedes , Die groben bis gründlichen Tests

- Testobjekte sind übersetz- und ausführbare Programme, **gepresst** werden müssen*.
- In der **Codierungsphase** jeweils nach Erstentwurf un ckung grob bis gründlich (Folie 2.29).
- Nach der Codierung **Zuverlässigkeitstest** mit Abdeck 1, auch hierarchisch absteigend.

* Voraussetzung für reprduzierbare Testergebnisse und experimemtelle Reparatur.

5.3 Testsuche

7.149 Aufgabenstellung

Für übersetz- und ausführbaren Code lässt sich nach unterschiedlichen Regeln ein große Menge von im Code instrumentierbarere Erreichbarkeits-, Infektions und mit Einschränkungen auch Weiterleitungskriterien formulieren, für die jeweils eine große Anzahl zufälliger Tests aus deren Eingabemengen zu suchen sind (Abschn. 7.4).

Verwandestes Thema in der Literatur Fuzz-Tests. Auch Software-Tests, auch Test mit viele zufälligen Eingaben, aber nur

- für fertige Systeme,
- vorrangig zur Suche von Sicherheitslücken mit
- eingeschränkte Abdeckungskriterien und Zusatzkontrollen.

Von Fuzz-Tests üblicherweise noch nicht ausgeschöpftes Potential:

- Beseitigung **aller** erkennbaren Probleme*,
- **isolierter Test** im Verbund schlecht testbare Teile und
- **weiter Maßnahmen prüfgerechter Entwurf** wie Steuerung und Beobachtung interner Werte z.B. für MF-Behandlungstest.

* Literatur betont, nur Beseitigung nachweisbar sicherheitsrelevanter Probleme.

7.150 Meilensteine der Fuzz-Test-Entwicklung

Ursprung: Ein Professor hatte die Idee, die Aufgabe zu stellen, UNIX-Werkzeuge mit unsinnigen Eingaben zum Absturz zu bringen. Ergebnis überwältigend. Am erfolgreichsten war ein Zufallsfolge von 1024 Zeichen, die über ein nicht abgefangenen Pufferüberlauf bei den meisten Werkzeugen interne Daten mit unsinnigen Werten überschrieben und die Programme darüber zum Absturz gebracht hat [UI/Application Exerciser Monkey. https://developer.android.com/studio/ test/monkey.html, gesehen 2016-05-31].

American Fuzzy Loop (AFL): Mutation bestehender legaler Eingaben mit Überwachung und Speicherung, welche Änderungen in der Eingabe welche Änderungen in der Programmausführung bewirken. Dadurch werden Tests gefunden, die nicht nur verschiedenste Bereiche zulässiger Eingaben, sondern auch verschiedenste Programmausführungen abdecken [American Fuzzy Lop: S. K. Cha, M. Woo, D. Brumley: Program-Adaptive Mutational Fuzzing. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (2015)].

Holler : Grammatik-basierter Würfel für zulässige, meist sinnlose Java-Programme für den Test des JavaScript-Interpreters, auch geeignet zur Aufspaltung gültige Java-Scripte in mutierbare Teile und Mutation [Christian Holler, Kim Herzig und Andreas Zeller: Fuzzing with code frag- ments. In: Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association (2012), Berkeley, CA, USA, 38-38].

Microsoft hat mit SAGE sein eigenen Fuzzer für den Test auf Sicherheitslücken [Patrice Godefroid, Michael Y. Levin und David Molnar: SAGE: Whitebox fuzzing for security testing. Communications of the ACM 55, 3 (March 2012), 40-44].

Google gibt für den Test ihres Crome-Browsers mit ClusterFuzz $5 \cdot 10^7$ untersuchten Fuzz-Tests pro Tag zur Suche von Sicherheitslücken an und auch, dass zusätzlich zur Kontrolle auf Abstürze mit SANITY zahlreiche Plausibiltiätstests instrumentiert werden [https://blog.chromium.org/2012/04/fuzzing-for-security.html].

Testsuche für große Systeme großer Rechenaufwand. Mit Vereinfachungspotential, isolierter Test, prüfgerechter Entwurf, ... vielleicht sogar mit moderatem Rechenaufwand gut beherrschbar.

7.152 Schwierigkeitsklassen

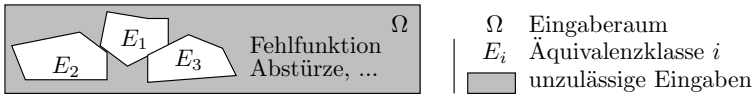
Die Testprofil- und Testsuche kann sich in Abhängigkeit von der Funktion und Struktur des Testobjekts, den Ein- und Ausgabeschnittstellen sowie der Instrumentierung mit Abdeckungs- und Kontrollkriterien unterschiedlich schwierig gestalten:

1. Zufallsauswahl erfolgsversprechend.
2. Würfelfunktionen für überwiegend zulässige Eingaben.
3. Kriterienbasierte Suche von Eingaberaumbeschränkungen.
4. Mutation gültiger Eingaben, z.B. vorhandener Regressionstests.
5. Eingabensuche nach Vorbild digitale Schaltungen.

(1) Testauswahl trivial. (2) beherrscht für Eingabespezifikation durch formale Sprachen und Nachrichtenprotokolle. (4) white-box Fuzzer, (3, 5) keine belastbaren Untersuchungen bekannt.

Für die Testsuche als Mix aus Ansätzen für Fuzz-Tests, digitale Schaltungen, prüfgerechten Entwurf, ... werden sich Lösungen finden.

7.153 Zufallssuche erfolgsversprechend

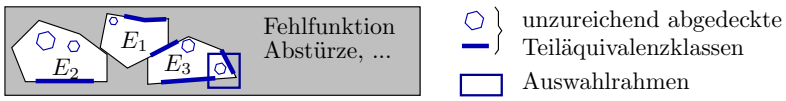


In der Regel beginnt eine Testauswahl mit zufälligen oder intuitiv gewählten Eingaben. Grundalgorithmus bei Zufallsauswahl:

- Auswürfeln zufälliger (überwiegend zulässiger) Eingaben aus dem Eingaberaum.
- Abdeckungskontrolle und wenn abgedeckt, Erhöhung der Abdeckungszähler.
- Bei Abdeckungsverbesserung Übernahme als Test.

Bei zu geringer Trefferate für neue brauchbare Eingaben, weiter mit der nächsten Schwierigkeitsklasse, modular absteigend oder anderen Maßnahmen des prüfgerechten Entwurfs.

7.154 Eingaberaumbeschränkung

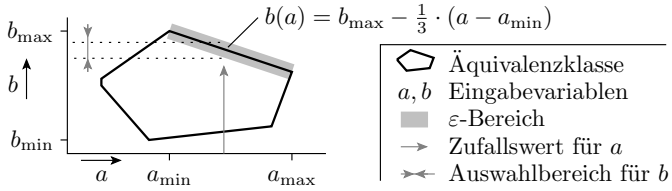


Mit geeigneten Würfelfunktionen (vergl. auch Folie 7.157 ff.) findet eine Zufallsuche für alle gut abdeckbaren Äquivalenzklassen und Kriterien ausreichend viele Tests. Diese werden für die weitere Suche gestrichen.
Für jede der verbleibenden (Teil-) Äquivalenzklasse mit bekannter Eingabebedingung $c_{i.j}$ (bzw. c_i):

- Beschränkung des Eingaberaums zur Erhöhung der Auswahlwahrscheinlichkeit.
- Zufallsauswahl, Abdeckungzählung etc. wie auf Folie zuvor.

Die Auswahlrahmen müssen die abzudeckenden Äquivalenzklassen nur so genau annähern, das je Treffer nur hunderte oder tausende Male gewürfelt werden muss (vergl. Selbsttest digitaler Schaltungen mit gewichteten Pseudo-Zufallsgeneratoren, Abschn. 6.3.4).

7.155 Würfelfunktionen für Bereichsränder



Bereichsränder sind bildlich gesehen $n-1$ dimensionale Flächen im n–dimensionalen Eingaberaum. Randbereiche umfassen alle Punkte mit einem kleinem Abstand ε zu diesen Flächen.

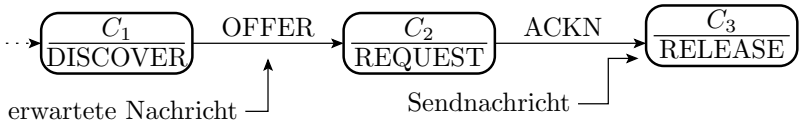
Freie Zufallsauswahl nur für die ersten Teilwerte eines Eingabevektors. Weitere Werte durch die bisherige Auswahl beschränkt, im Bild

```
a = rand_typ_a(amin, amax);
b = rand_typ_a(b(a) − εb, b(a) + εb);
```

Erfordert für alle elementaren Datentypen Würfelfunktionen vom Typ:

```
<tTyp> rand_ttyp(<tTyp> min, <tTyp> max);
```

7.156 Nachrichten als Ablaufbedingungen



Komplexe Teilsysteme, z.B. Client-Server-Systeme, kommunizieren vorzugsweise über Nachrichten. Im Bild Aktionsfolge zur DHCP-Anforderung einer Netzwerkadresse (Folie 7.85). Nach jedem Schritt warten Server bzw. Client auf Nachrichten bestimmten Typs und antworten mit bestimmten Nachrichten.

Nachrichtenformate oft standardisiert oder ähnlich. Pragmatik:

- für Standardnachrichtenformate Generierung von Testeingabewürfeln aus den Beispielabläufen,
- für parametrierte Formate gemeinsame Generierung von Testeingabewürfeln mit Coder und Decoder.

Nachricht: Komplexer Datenrahmen zur Darstellung unterschiedlicher Arten von Informationen.

7.157 Würfelfunktionen für Nachrichten

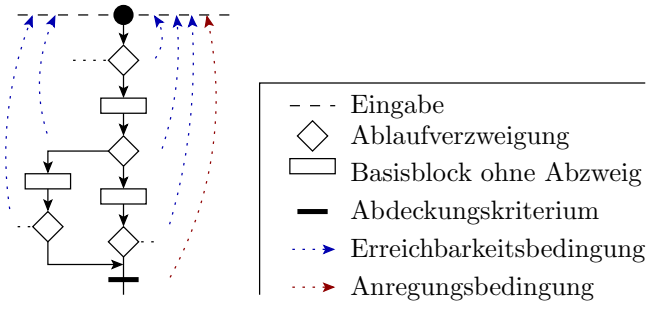
Sicherungs-schicht			MAC-Empfänger	MAC-Absender	Protokoll-typ	Nutzlast max. 1500 Bytes	Prüfkennzeichen 4 Byte	
Bitübertra-gungsschicht	Präambel	Start-byte						Lücke zum nächsten Paket
Byteanzahl	7	1	6	6	2	46 bis 1500	4	12

Nachrichten haben ein Format mit vielen kontrollierbaren Merkmalen, in der Regel auch ein Prüfkennzeichen (vergl. Ethernet-Protokoll, Folie 5.75). Zufällige Bitfolgen sind nur mit einer Wahrscheinlichkeit $\leq 2^{-r}$ (r Anzahl der Prüfkennzeichenbits) gültige Nachrichen. Unzulässige Nachrichten in Nutzungsprofilen selten.

Eine Würfelfunktion für Beispielnachrichten benötigt eine Beschreibung der Nachrichtenbestandteile, ihrer Wertebereiche, Formatmerkmale, insbesondere die Bildungsregeln für Prüfkennzeichen, Nutzungshäufigkeiten in Abhängigkeit vom Testobjektzustand, ...

Denkbar ist eine Anlehnung an eine formale Sprache, aus der sich Kontrollautomat und Würfelfunktion ableiten (Folie 5.127) mit Erweiterungen oder ein Code-Generator, der die Würfelfunktion mit erzeugt.

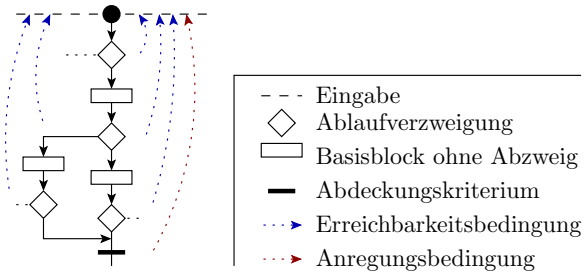
7.158 Suche von Eingabebedingungen



Statt Eingabebedingungen ist nur ein Abdeckungskriterium bekannt

- dass an eine bestimmte Stelle im Kontrollfluss
- für bestimmte Variablen bestimmte Werte

fordert. Dazu sind über die Eingaben Erreichbarkeits- und optional Infektionsbedingungen zu befriedigen.



Für mögliche Kontrollablauf vom Programmstart bis zum Abdeckungskriterien sind über die Eingabe Entscheidungen zu steuern:

ODER mehrere UND-verknüpfte Bedingungen für Entscheidungsvariablen. Schwierigkeitsklassen der Eingabesuche:

1. steuerbare Entscheidungsvariablen, Testsuche D-Algorithmus.
2. Berechnung der Entscheidungswerte nur aus Eingaben.
3. Zustandsabhängige Entscheidungen, ...

(1) D-Algorithmus, (2, 3) vermeidbar durch instrumentierte Steuerung interner Zwischenergebnisse und Zuständen.

7.160 Wertesuche

Fuzz-Test bevorzugt Wertesuche (Folie 7.150):

- Abstiegsverfahren: Suche von Wegen stetiger Verbesserung durch kleine Eingabevariationen.
- Kombination und Mutation gut bewerteter Eingaben zur Suche besserer Eingaben.

Abstiegsverfahren und genetische Algorithmen benötigen

- Startwerte, z.B. vorhandene Regressionstests und
- Bewertungsfunktionen für mutierte Testeingabe.

Vorschlag für Kriterien aus ODER, UND und Bereichskontrollen:

- Bereichsabfrage: eins wenn erfüllt, sonst eins plus Abstand,
- UND: Abstandsprodukt,
- ODER: Minimalabstand,
- nicht erreichte Abfragen: großer Standardwert (MAX).

Beispiel sei das Abdeckungskriterium:

$$(A == 20) \wedge (20 < B < 85) \wedge (C == 1)$$

A, B, C: programmintern aus Eingaben und Zuständen berechnete Werte. Abstandsberechnung:

```
/ Programmstart
int Bxx= MAX; // max. Strafpunkte Bedingung xx
...
// Instrumentierung
int sa=1; if (A>20) sa+=A-20) else sa+=20-A);
int sb=1; if (B>1) sb+=A-20 else if (B>85) sb+=85-A;
int sc=1; if (A<1) sc+=1-A else sa+=A-1;
Bxx = min(sa * sb * sc, MAX);
```

Wenn die Fachwelt die Notwendigkeit von $w \gg 1$-Tests als Vorraussetzung für verlässliche Software anerkannt haben wird, wird es genau wie für digitale Schaltungen und Fuzz-Tests innerhalb weniger Jahre praxistaugliche Werkzeuge dafür geben.

5.4 Testergebniskontrollen

7.162 Kontrolle für Testergebnisse

Problematisch wie Testsuche. Keine fehlerfreie Beschreibung, kein einfach zu automatisierender Soll/Ist-Vergleich der Testausgaben mit nahe 100% Fehlfunktionsabdeckung. Alternativen:

1. Watchdog für Abstürze (Folie 1.99), Zusicherungen (Folie 7.129),
2. Automatisiert eingefügte Kontrollen auf Wertebereichsüberläufe, Indexfehler, ... (vergl. Rust, Smart-Pointer, ...),
3. Ausgabeinspektion, händische Sollwertbestimmung (Folie 5.5),
4. Mehrfachberechung und Vergleich, Loop-Tests (Abschn. 1.2.3), Standardkontrollfunktionen (Abschn. 5.4), ...

(1, 2) Aktueller Entwicklungsstand, (3) üblich für manuelle Testentwicklung, ungeeignet $w \gg 1$-Tests, (4) Potential, dass für bezahlbare $w \gg 1$-Tests durch ein zu schaffende Testinfrastruktur nutzbar gemacht werden müsste.

7.163 Verbesserte Inspektionstechniken

Inspektion aktuell häufigste Kontrollart für Testausgaben. Wird es für erste Grobtests nach Entwurf, Phantomfehlerkontrolle, ... auch bleiben. Maßnahmen zur Aufwandsminderung und Abdeckungsverbesserung (Abschn. 5.1.2):

1. Beschränkung auf wesentliche Ergebnisse, z.B. nur die Änderungen im Objektraum (vergl. get_diff()-Funktion, Folie 5.53).
2. Aufzeichnung und ergonomische Darstellung der Werte.
3. Vier-Augen-Prinzip, Vermeidung Ermüdungsphasen (Folie 5.22).
4. Definition Einstufung kontrollierter Istwert als Sollwerte zur Vermeidung einer mehrfachen Inspektion derselben Testergebnisse.
5. unabhängig (diversitären) Sollwertberechnung (z.B. mit Demonstratoren).

(1), (2) und (5) weitgehend automatisierbar, (3) und (4) durch Tool-Unterstützung erheblich vereinfachbar.

7.164 Einbau erweiterter Kontrollen

Eine zu schaffende Testinfrastruktur sollte alle bisher üblichen automatisierter Testausgabekontrollen unterstützten:

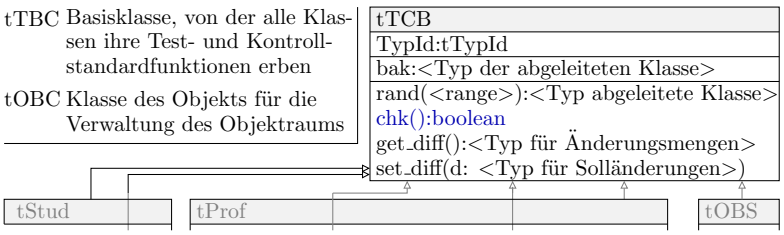
- Zeitüberwachung für Abstürze und Zeitüberschreibungen,
- Zusicherungsverletzungen (vergl. should_panic(), Folie 7.29 ff.),
- Wertebereichsüberläufe, Index-Fehler, Division durch null, ...

Hohe Fehlerabdeckung ohne Inspektion verlangt mehr Kontrollen. Notwändig viel günstiges Verhältnis aus Abdeckungsverbesserung und manuellem Zusatzaufwand als manueller Inspektion. Bereits besprochene und eine weite Idee:

- Standard-chk()-Funktionen zur Kontrolle auf Zulässigkeit,
- generierte Kontrollautomaten,
- aus »Code-Abfällen« generierte Checker, ...

Eine Vervielfachung der Laufzeit durch diese Kontrollen auch um eine Zehnerpotenz stört nicht, da Testsuche aufwändiger und Istwertkontrolle nur einmal nach der Testsuche erforderlich.

7.165 Wiederholung chk()-Standardfunktion

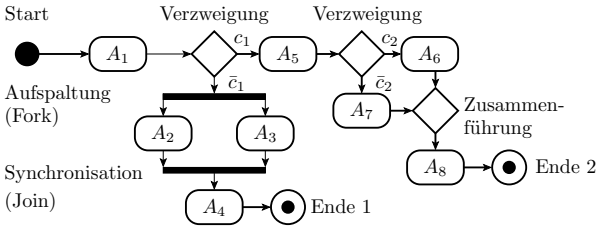


In Abschn. 5.4.3 wurde die Idee skizziert, von einer Basisklasse tTBC für alle Klassen eines Projekts und den Objektraum eine überladbare Kontrollfunktion für alle aus der Klassenbeschreibung ableitbaren Plausibilitätskriterien zu vererben. Eingebettet die Ideen:

- Klassen-ID zur Typkontrolle für zeigeradressierte Objekte,
- Indexbereichskontrollen für zugeordnete Kollektionen,
- Plausibilitätstest für alle Teilobjekte mit deren chk()-Funktion, ...

In der Übersetzung für **Testausgabekontrolle** chk()-Aufruf nach **je**dem **Berechnungsschritt**. Vielfacher Rechenaufwand **kein Problem**.

7.166 Generierte Kontrollautomaten

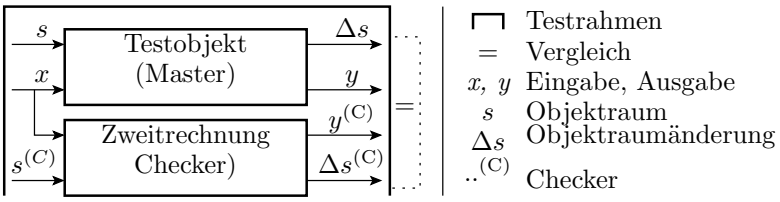


$$\text{OK} = A_1, (((A_2, A_3) \,|\, (A_3, A_2), A_4) \,|A_5, (A_6|A_7), A_8);$$

Für Abläufe, die durch Aktivitätsdiagramm, Sequenzdiagramme oder Automaten spezifiziert sind, wurde in Abschn. 7.3.3 die Idee skizziert, daraus automatisch eine Sprache für zulässige Abläufe und daraus wiederum einen Kontrollautomaten zu generieren.

Einzige manuell erforderderliche Zuarbeit ist eine Kennzeichnung der Code-Stellen für Beginn und optional Ende der Aktionen A_i. In der Übersetzung für **Testausgabekontrolle** auch **kein Kostenfaktor.**

7.167 Code-Abfälle als Checker



Mehrfachberechnung und Vergleich kann im Gegensatz zur manuellen Inspektion viel größere Datenmengen kontrollieren, erkennt alle Ausgabeabweichungen, Fehler aber nur bei Diversität.

Software-Entwurf ist ein Suchprozess, in dem oft Code-Bausteine entstehen, die später komplett neu geschrieben werden, z.B.

- zur Kontrolle der Realisierbarkeit mit einem Demonstrator,

- Top-Down-Entwurf mit Demonstrator-Attrappen,
- große nachträgliche Änderung Struktur, Algorithmus, ...

Solche Code-Abfälle sind zwar unvollständig und unzuverlässig, versprechen aber eine gewisse Diversität.

7.168 Generierung von MS-Kontrollfunktionen

Zur Master-Funktion $y = f(x)$ seien im Projekt potentielle Checker-Funktionen $fc_i(x)$ annotiert, die jeweils unter einer Bedingungen $cc_i(x)$ im fehlerfreien Fall gleiche Ausgaben liefern. Kontrollfunktion*:

```
tIn ...; tOut ...; // Ein und Ausgabetyp (oder Klasse)
tTRes check_f(tIn x, tOut y)
  if (cc1(x))          check_eq_f(x, y, fc1(x));
  else if (cc2(x))     check_eq_f(x, y, fc2(x));
  ...
  else if (no_chk(x)) no_check_f(x, y);
  else                 check_mf_handling_f(x, y);
 }
```

Davon automatisch generierbar:

- Statischer Test Schnittstellengleichheit Master und Checker**,
- Generierung der Funktionen cc*i*(..), check_f(..), no_check(..) und check_mf_handling_f(..) in manuell nachbesserbarer Form***.

---

* Im Beispiele zur Vereinfachung ohne zu bearbeitenden Objektraum und ohne Attrappen.

** in der Code-Skizze zu Datenstrukturen tIn und tOut zusammengefasst.

*** Nachbesserung z.B. durch Annotation der Änderungen im Quellcode.

7.169 Manuelle Nachbesserung

```
tTRes check_f(tIn x, tOut y)
  if (cc1(x))          check_eq_f(x, y, fc1(x));
  else if (cc2(x))     check_eq_f(x, y, fc2(x));
  ...
  else if (no_chk(x)) no_check_f(x, y);
  else                 check_mf_handling_f(x, y);
 }
```

- Schnittstellenanpassung der Checker-Funktionen an den Master,
- Zuordnung der Eingabebereiche zu Checker-Funktionen.
- Phantomfehlerunterbindung durch Ausschluss von Ausgabebestandteilen vom Vergleich oder Fenster- statt exakter Vergleich.

Die Entwurfsskizzen zeigen, dass teilautomatisierte Nutzung von Code-Abfällen« als Checker zur Testausgabekontrolle denkbar ist. Ob zieführend, hängt vom verbleibenden manuellen Aufwand, Umfang und Qualität des verfügbaren Checker-Codes und dessen Diverität ab.

5.5 Zusammenfassung

7.170 Zusammenfassung

Wenn die Fachwelt die Notwendigkeit von $w \gg 1$-Tests als Basis verlässlicher Software anerkennt, wird eine Testinfrastruktur benötigt. Wichtige Werkzeugebenen:

1. Automatisierte Extraktion der Testobjekte, erforderlicher Attrappen, Abdeckungskriterien, Kontrollmöglichkeiten, vorhandener Regressionstests, ... aus dem übersetz- und abarbeitbaren annotierten Quellcode und zugeordneter relevanten Dokumente.
2. Je Testobjekt Testeingabesuche mit geeigneten automatisch erzeugten Rahmen und Instrumentierungen von Abdeckungszählern und/oder Bewertungsfunktionen,
3. Testausführung mit instrumentierten Kontrollen.
4. Generierung und Ausführung Reparatur- und Regressionstests.

(1) und (4) bekannt/gut vorstellbar. (2) Nutzbares Knowhow: Fuzz-Tests, D-Algorithmus, prüfgerechter Entwurf. (5) Viel ungenutztes Potential.

Skizzierte Testinfrastruktur sehr ambitioniert, aber **lösbar**.

7.171 Literatur

Literatur

[1] R. Binder. *Testing Object-Oriented Systems. Models, Patterns, and Tools.* Addison-Wesley, 1999.

[2] Melvin E. Conway. How do committees invent? April 1968.

[3] Mark Richards. *Software Architecture Patterns.* O'Reilly Media, Inc., 2015.