

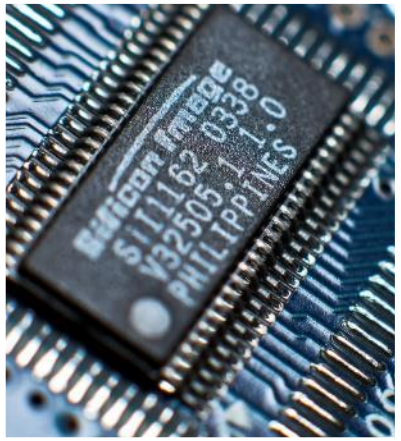


Die (automatische) Reduzierung Technischer Schulden in Legacy-Codebasen 13:30-14:15

VECTOR software

VECTOR

Ingo Nickles
Sr. Field Application Engineer
ingo.nickles@vectorcast.com
Vector Software





Agenda

- “Technical Debt”: Was versteht man unter “technischen Schulden”
- Raus aus den Schulden
  - Unit- und Integrations-Test
  - Automatische Unit Test-Framework Erstellung
  - „baselining“: Input Daten
  - „baselining“: Expected Daten
- Live Demo

Technische Schulden (technical debt) in der Software Entwicklung

Technische Schulden (technical debt)

- Was ist das
  - Zusätzlicher Aufwand, den man für Änderungen und Erweiterungen an einem schlecht entwickelten Produkt leisten muss.
  - Meist im Zusammenhang mit mangelnder Software Qualität
  - Kredit auf Kosten zukünftiger Erweiterungen

- Wie entsteht das
  - Vernachlässigen von Engineering Regeln wegen
    - *Unwissenheit*
    - *Zeitmangel*
    - *Geldmangel*
    - *Faulheit*
- Beispiel
  - Testphase am Ende des Projekts wird als Puffer der Entwicklung verwendet („Schwarze Peter Spiel“)
  - Erstes Release Datum wird eingehalten
  - Mangelhafte Tests führen zu erhöhten Aufwändungen für Fehlerbeseitigungen
  - Entwicklungsphase des nächsten Releases belastet durch technische Schulden

Technische Schulden können in vielen Varianten erscheinen

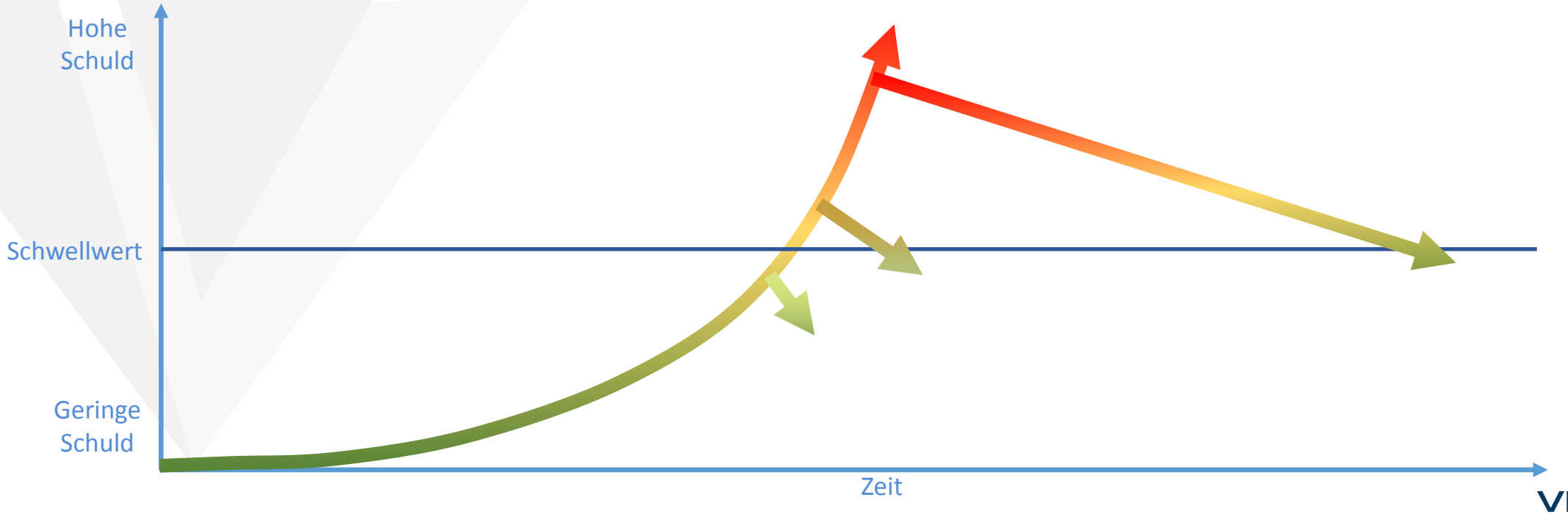
- Softwaredokumentation
- Technische Infrastruktur
  - Versionsverwaltung
  - Datensicherung
  - Build-Tools
  - Kontinuierliche Integration
  - Statische Code Analyse
- Modultest, Integrationstest, Systemtest (Regressions-Test)
- Compiler-warnings und Anmerkungen der statischen Code-Analyse
- Coding
  - Coding rules (Einrückung, Schreibweise, Kommentardichte, Fehlervermeidung, Verständlichkeit...)
  - Codewiederholungen
  - Zu großer oder zu komplexer Code
  - HAL
  - Trennung von Datenverabeitung und Control Flow
  - ...

VECTOR software

Entstehung von technischer Schuld

Technische Schulden in der Software Qualität

- Üblicherweise kein Problem bei neuer SW
- „Continuous-Feature-Integration“ ist aber die Zukunft
- Langlebige SW muss refakturiert werden (ja, auch SW altert)
- Ansonsten steigen die technischen Schulden exponentiell



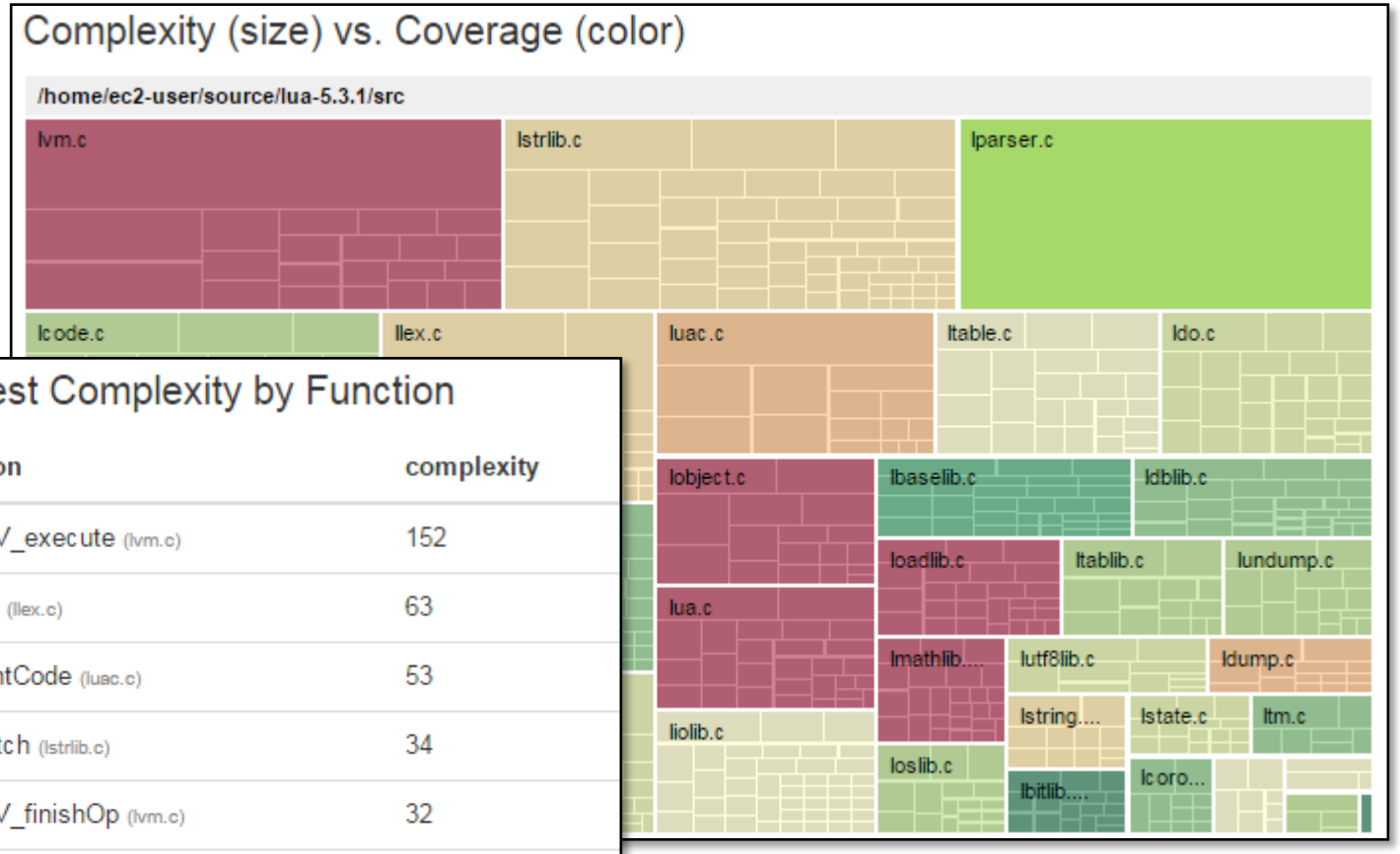
Raus aus den Schulden

Probleme

- Refakturierung hat keinen sichtbaren Mehrwert
  - Kostet Zeit und Geld
  - Kein neues Feature
- Bis zu einem gewissen Punkt wird das Problem ignoriert
  - Release-Zyklen verlängern sich bzw. Fristen können nicht eingehalten werden
  - Software Qualität verschlechtert sich
  - Software Entwickler sind mehr mit Fehlerbeseitigungen beschäftigt als mit Neuentwicklung
- Software Entwicklung muss Refakturierung als Feature verkaufen
  - Metriken können eine Argumentationshilfe sein
    - *Entwickler Einsatz-Zeiten (Fehlerbeseitigung vs. Neuentwicklung)*
    - *Fehler-Statistiken*
    - *Software-Metriken*
      - Zyklomatische Komplexitäten
      - Code Größen
      - Testfall-Abdeckung
      - Code-Abdeckung

Measure. Report. Act.

- Metriken können helfen
  - Argumentationshilfen
  - Problemzonen identifizieren



Highest Complexity by File

file	complexity
lvm.c	319
lstrlib.c	302
lparser.c	276
lcode.c	225
lapi.c	210
lgc.c	209
llex.c	182
lauxlib.c	160
ldebug.c	153

Highest Complexity by Function

function	complexity
luaV_execute (lvm.c)	152
llex (llex.c)	63
PrintCode (luac.c)	53
match (lstrlib.c)	34
luaV_finishOp (lvm.c)	32
read_string (llex.c)	31
luaV_equalobj (lvm.c)	30
getfuncname (ldebug.c)	29
getoption (lstrlib.c)	26

Das Problem mit dem „Act“

- „All or Nothing“?
  - Ein „Refactor-Release“ ist eher die Ausnahme
  - Verbesserungen in kleinen Schritten eher die Regel
  - Mögliches Vorgehen
    - *1. Wähle die komplexeste Funktion*
    - *2. Refakturiere*
    - *3. Goto 1.*

- „Never change a running system“
  - Code Änderungen führen oft zu neuen Fehlern
    - *Auswirkungen von Code-Änderungen kaum absehbar*
    - *Auch Beseitigungen offensichtlicher Fehler können zu Fehlern führen*
    - *Ein Mangel an Testfällen verhindert ein „absichern“ von Code-Änderungen*
    - *Schlaflose Entwickler*

Vorteile eines „running System“ nutzen

- Optimal: Prozess-konforme Erstellung von Testfällen
  - Zeitaufwendig
  - Setzt Prozess-konforme Dokumente voraus (Requirements)
- Alternative: „Baselining“:
  - Refakturierung soll das Verhalten des Systems nicht ändern.
  - Also müssen wir nur:
    - *Das jetzige Verhalten aufzeichnen*
    - *Refakturieren*
    - *Das neue Verhalten mit dem alten Verhalten vergleichen*
    - *Es sollte keine Unterschiede geben*
  - Verhalten von SW aufzeichnen?
    - *Divide et Impera: Das Verhalten des Systems baselinen mit Unit- und Integrationstests*

VECTOR software

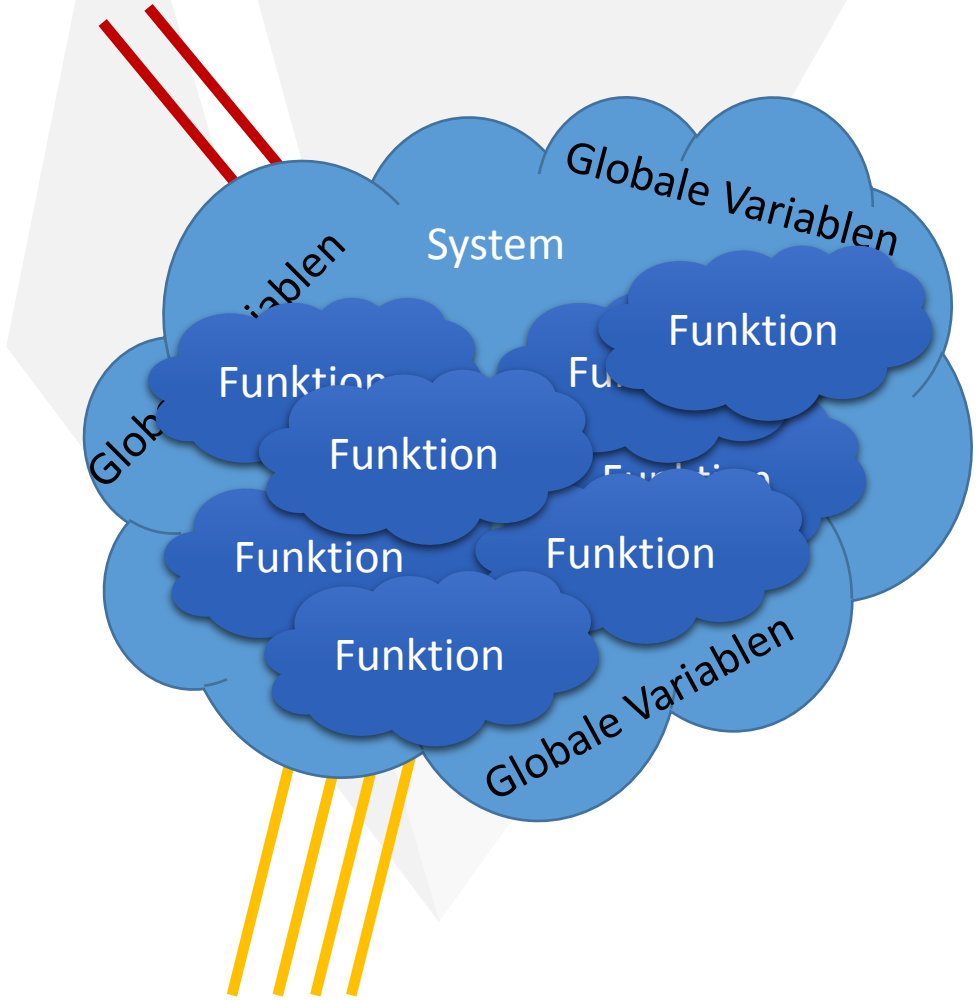
Baselining Testfall Erstellung

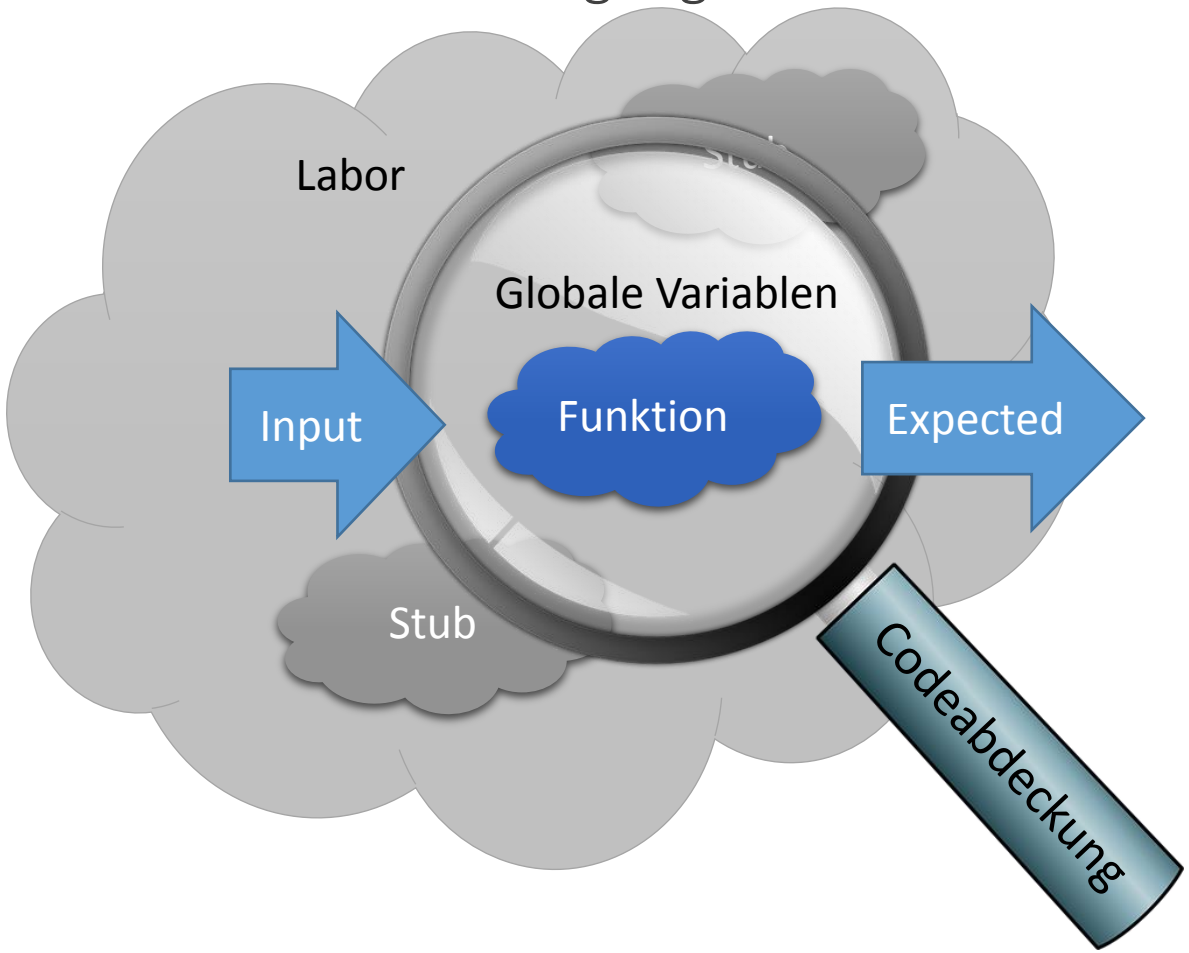
- (Automatische ?) Erstellung von Unit- und/oder Integrations Testumgebungen
- (Automatische ?) Erstellung von Test-Vektoren
- (Automatische ?) Ausführung der Testfälle mit Aufzeichnung der Ist-Daten
- Verwendung der Ist-Daten als Soll-Daten (=baselining)

Unit- und Integrations-Test

Erstellung von Unit-Testfällen: Das Prinzip

- Testen einzelner Funktionen/Methoden unter Labor Bedingungen





Erstellung von Unit-Testfällen: Input & Output Daten

Input

global data

```
g1;
g2;
```

Function

```
a
b
```

Stub Daten

```
bar()-return
```

Source file under test

global data

```
extern int g1;
int g2;
```

Funktionen

```
int foo(int a, int &b){
    int i=5;

    a=i*a;
    b=bar(a);
    if (b) g1=g2;

    return a;
}
```

Output

global data

```
g1;
g2;
```

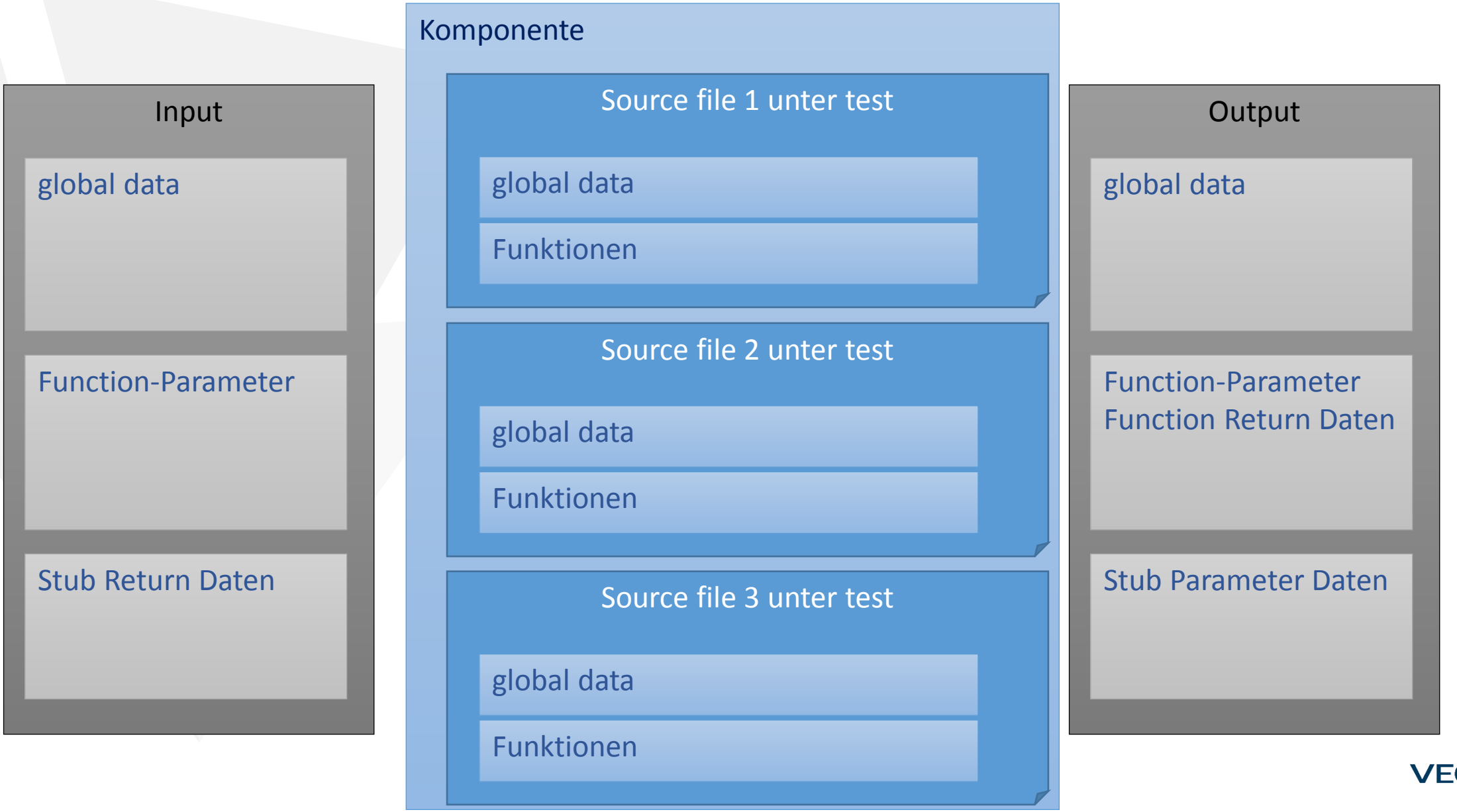
Function

```
b
foo()-return
```

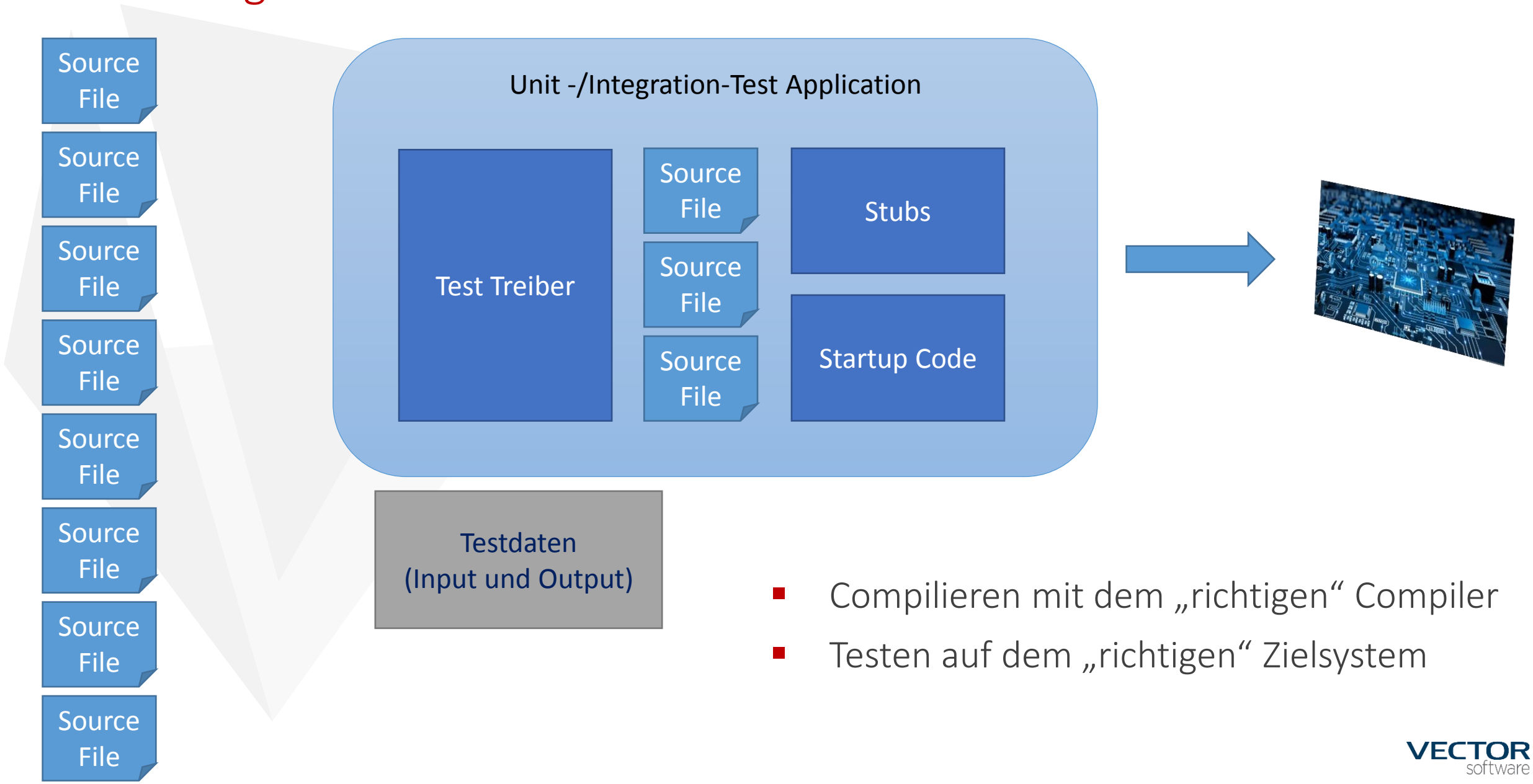
Stub Daten

```
bar()-parameter
```

Integrations- (=Komponenten-) Test



Unit-Integrations-Test-Framework



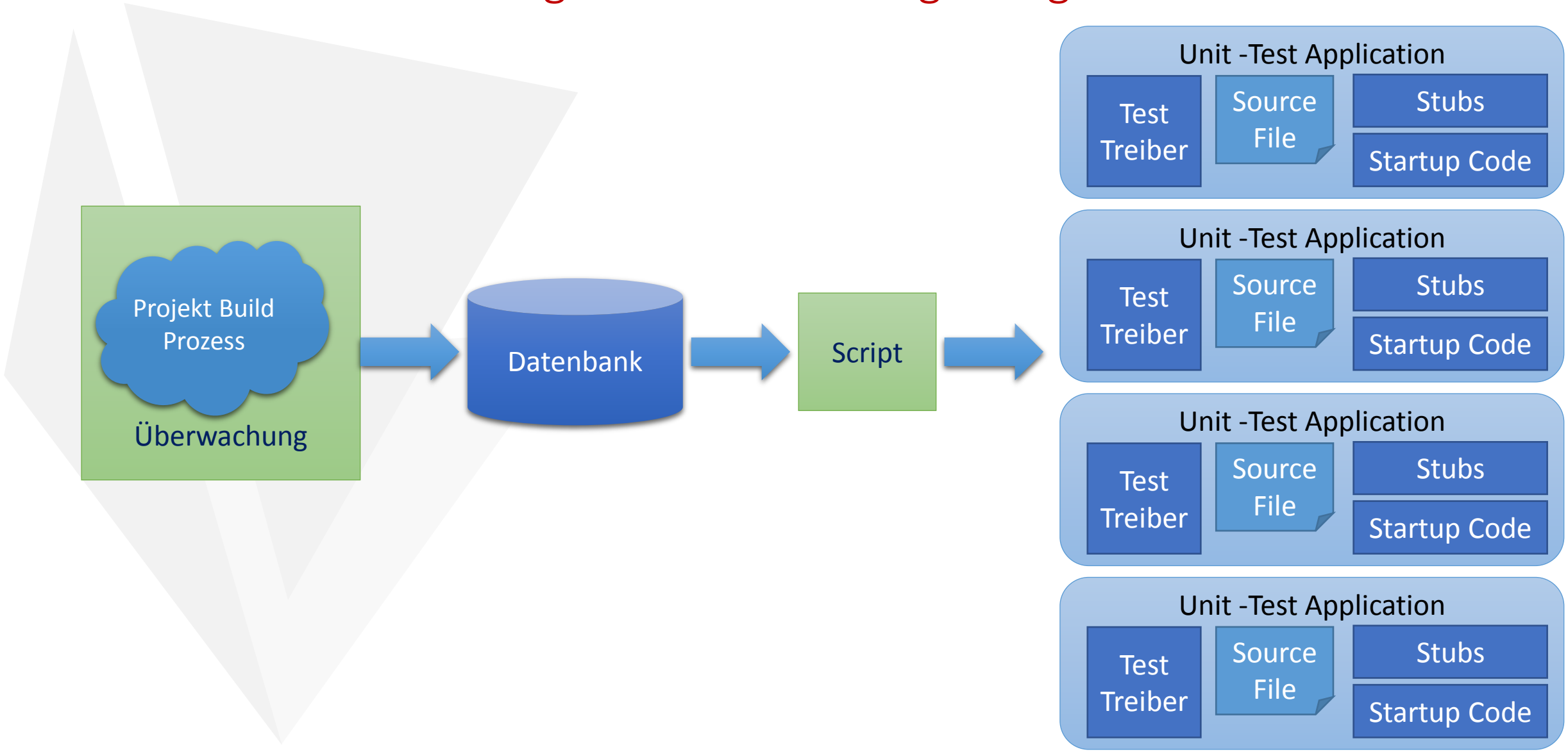
- Compilieren mit dem „richtigen“ Compiler
- Testen auf dem „richtigen“ Zielsystem

Automatische Unit Test-Framework Erstellung

Analyse der Build-Umgebung

- Herausforderung: Automatische Ermittlung von
  - Compiler Optionen (Compile-defines, Include-Pfade)
  - Source Files im Projekt
- Automatisierte Erstellung durch Analyse von IDE-Projekt-Dateien möglich
  - Nachteil:
    - *Verschiedene Entwicklungsumgebungen haben unterschiedliche Formate*
    - *Projekt-Dateien sind keine offiziellen APIs*
      - Neue Versionen = geänderte Formate
    - *Projektdateien sind optional*
      - Manche Entwicklungsumgebungen nutzen „make“
- Alternativer Ansatz:
  - Aufzeichnen der Kommandos beim Build des Projekts
  - Ableiten der benötigten Informationen von der Aufzeichnung

Automatische Erstellung von Unit-Testumgebungen



„baselining“: Input Daten

Automatische Erstellung von Test-Vektoren

Äquivalenzklassen-Test

- Ableitung der Daten von der Implementierung
  - Function Interface
    - *Äquivalenzklassen-Test*
      - Mehr oder weniger Zufällige Pfad Abdeckung

```
int foo(int a, int &b);
```

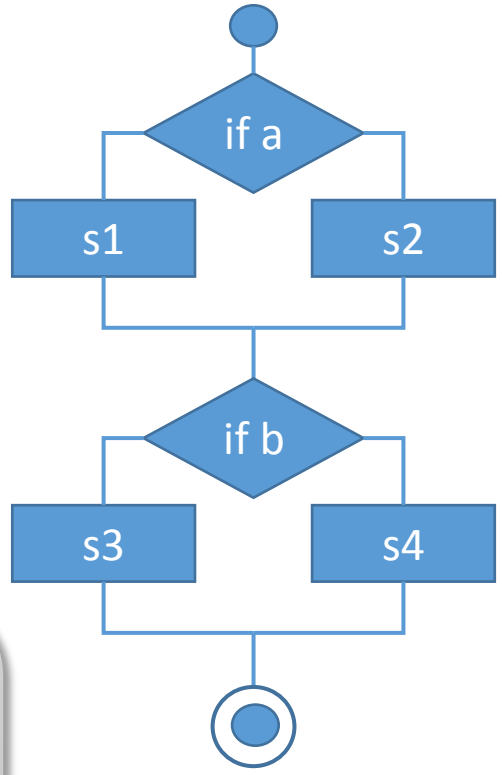
```
foo (<<MIN>>, <<MIN>>);
foo (<<MIN>>, <<MID>>);
foo (<<MIN>>, <<MAX>>);

foo (<<MID>>, <<MIN>>);
foo (<<MID>>, <<MID>>);
foo (<<MID>>, <<MAX>>);

foo (<<MAX>>, <<MIN>>);
foo (<<MAX>>, <<MID>>);
foo (<<MAX>>, <<MAX>>);
```

Code-Abdeckungs basierte Tests

- **Function coverage**
  - Jede Funktion sollte wenigstens einmal aufgerufen werden
- **Function call coverage**
  - Jeder Funktionsaufruf im Code sollte ausgeführt werden
- **Statement coverage**
  - Jede Zeile sollte ausgeführt werden
- **Decision (Branch) coverage**
  - Jede Entscheidung sollte mit beiden Ergebnissen True/False ausgeführt werden
- Condition coverage
  - Jede Unterbedingung einer Entscheidung sollte mit beiden Ergebnissen True/False ausgeführt werden
- Condition/Decision coverage
  - Kombination aus Condition und Decision Coverage
- **Modified condition/decision coverage (MC/DC)**
  - Condition/Decision Coverage mit Nachweis der Unabhängigkeit der Unterbedingungen

Automatische Erstellung von Baseline-Testfällen: Input Daten

- Ableitung der Daten von der Implementierung
  - Code Coverage
    - *Pfad-Coverage*
      - Basis Path Coverage
        - *Durchlaufen aller linear unabhängigen Pfade, ohne Iterationen*
        - *Gute Testfall Abdeckung, minimale Redundanzen*
        - *Anzahl linear unabhängiger Pfade <= zyklomatische Komplexität*
        - *Liefert 100% Statement+Branch Coverage*



Anzahl Pfade insgesamt:

a	b	Pfad
TRUE	TRUE	s1, s3
TRUE	FALSE	s1, s4
FALSE	TRUE	s2, s3
FALSE	FALSE	s2, s4

Zyklomatische Komplexität: 3

Minimale Anzahl Pfade für 100% Coverage:

a	b	Pfad
TRUE	TRUE	s1, s3
FALSE	FALSE	s2, s4

Basis Path Pfade:

a	b	Pfad
TRUE	TRUE	s1, s3
TRUE	FALSE	s1, s4
FALSE	TRUE	s2, s4

Automatische Erstellung von Baseline-Testfällen: Input Daten

- Ableitung der Daten von der Implementierung
  - Code Coverage
    - *MC/DC Coverage*
      - Unterbedingungs-Coverage
      - Unterbedingungen müssen unabhängig sein
      - Nachweis über Code Coverage
      - Anzahl Testfälle bei n Unterbedingungen = n+1

```
if (a && b && c) {...}
```

Wahrheitstabelle

a	b	c	Result
TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	TRUE	FALSE
TRUE	FALSE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE

„baselining“: Expected Daten

Automatische Erstellung von Test-Vektoren

Automatische Erstellung von Baseline-Testfällen: Expected Daten

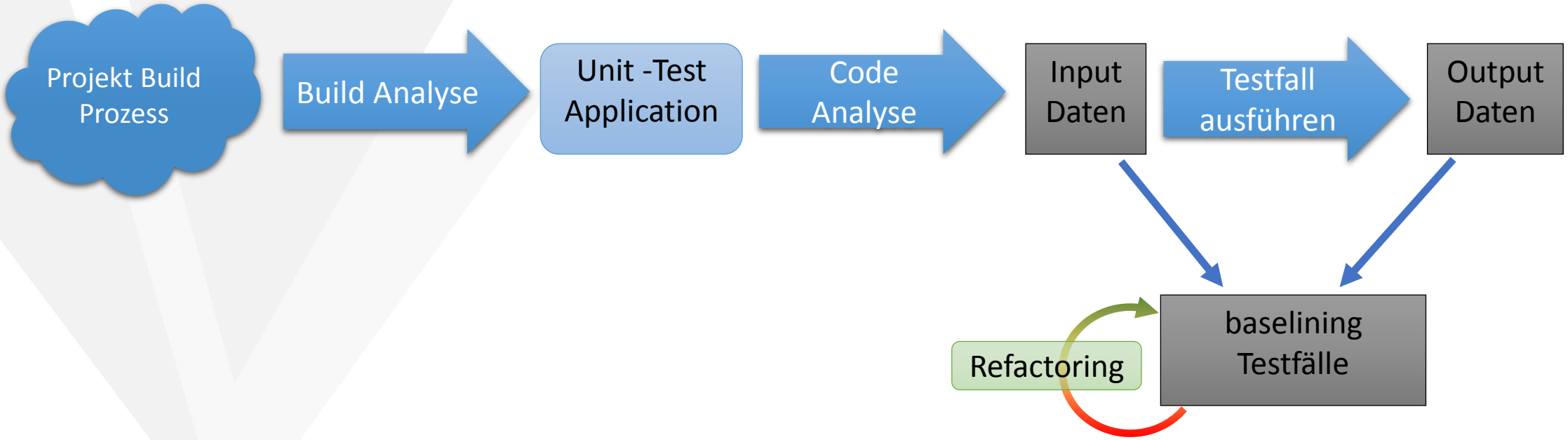
- Annahme: funktional korrektes Verhalten der bestehenden Software
- Welche Daten sollten erfasst werden?
  - Alle Output Daten
    - *Globale Variablen*
    - *Stub Daten*
      - Aufgerufene Funktionen
      - Stub Parameter
    - *Funktion unter Test*
      - Interface Parameter (Call by reference)
      - Funktion-return-value
  - Relevante Output Daten
    - *Nur Elemente, auf die schreibend zugegriffen wird*

Anwendbarkeit von baselining Testfälle

- Refactoring (technical dept)
- Re-use von Software
- Portierung von Software auf andere HW/anderer Compiler
- Ändern von Compiler Settings

Zusammenfassung

- Measure, report, act
- Automatisiertes Baselining vor Refakturierung



Live Demo





VECTOR software

Proven Test Solutions for Reliable Software

vectorcast.com