Ausarbeitung

im Kurs

Wahl-Pflicht-Veranstaltung

Dienst- und Protokollentwicklung

Projektbetreuer : Prof. Frank

Eingereicht am : 10.01.2016

Matthias Beyer
Immatrikulationsnummer: CENSORED

CENSORED, CENSORED, CENSORED

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

ASCII American Standard Code for Information Interchange

ASN.1 Abstract Syntax Notation One

BNF Backus-Naur-Form

HTTP Hyper Text Transfer Protocol

HTTPS Hyper Text Transfer Protocol Secure

IDE Integrierte Entwicklungsumgebuch (aus dem englischsprachigen: Integrated Development Environment)

IMAP Internet Message Access Protocol

IP Internet Protocol

JSON JavaScript Object Notation

MSC Message Sequence Chart

NFS Network File System

GCC GNU is Not Unix (GNU) Compiler Collection

GDB GNU Debugger

GNU GNU is Not Unix

GPS Global Positioning System

POP Post Office Protocol

RPC Remote Procedure Call

RFC Request For Comments

UDP User Datagram Protocol

SDL Specification and Description Language

SMTP Simple Mail Transfer Protocol

TCP Transport Control Protocol

VCS Version Control System

WebDAV Web-based Distributed Authoring and Versioning

XML eXtensible Markup Language

1. Einführung

Ein Programm durchläuft bei seiner Entwicklung von der Idee bis hin zum Produkt mehrere Phasen, in denen verschiedene Werkzeuge zum Einsatz kommen. Welche Werkzeuge zur Anwendung kommen ist dabei von vielen Faktoren abhängig, wie zum Beispiel der Programmiersprache, in welcher das Programm geschrieben werden soll, auf welchem System das Programm ausgeführt werden soll und nicht zuletzt mit welchen Werkzeugen das Programmes entwickelt wird.

So wird bei der Entwicklung eines Programmes in der Programmiersprache "Ruby" Wert auf andere Tools gelegt als bei der Entwicklung eines Programmes in der Programmiersprache "C". Zwischen beiden Programmiersprachen bestehen Unterschiede, die eine Einschränkung der Werkzeuge mit sich bringen: "Ruby" ist eine interpretierte Programmiersprache, "C" eine Sprache welche von einem Compiler übersetzt wird. Der Unterschied besteht darin, dass der Programmcode einer interpretierten Programmiersprache zur Laufzeit des Programmes in den entsprechenden Maschinencode Maschinencode übersetzt wird. Bei einer Sprache welche von einem Compiler übersetzt wird (Compilersprache), wird der Programmcode vor der Ausführung in eine für die Zielplattform ausführbare Form übersetzt (siehe auch: [Sch01, S. 16]). Dies hat auch Auswirkungen auf die Werkzeuge, welche für diese Sprachen benutzt werden. So werden Programme welche in "Ruby" geschrieben sind eher mit "bundler" und "rake" zu Softwarepaketen übersetzt, Programme in "C" mit "autotools" oder "cmake". Selbstverständlich hat die Programmiersprache auch Auswirkungen auf die Testumgebungen, welche für Programmtexte verwendet wird. Dies hängt nicht zuletzt von der Art der Programmiersprache selbst ab, dem Programmierparadigma. Ist eine Programmiersprache in die Kategorie der Objektorientierten Programmiersprachen einzuordnen, so werden andere Test-Frameworks verwendet als zum Beispiel bei einer Programmiersprache mit funktionalem Programmierparadigma.

Diese Arbeit stellt verschiedene Entwicklungswerkzeuge wie Editoren, Compiler, Build-Tools, Debugger und Testumgebungen sowie Werkzeuge zur Versionskontrolle vor.

Die Aufgaben, welche ein Programm erfüllt, können als Dienste bezeichnet werden.

Es ist dabei nicht von Belang, ob diese Dienste über ein Netzwerk, lokal für einen Nutzer oder lokal für das System angeboten werden. Zudem ist zu unterscheiden welche Dienstleistung das Programm selbst anbietet und in welchem Format (Protokoll) die Kommunikation mit dem Programm abläuft. Die Unterscheidung zwischen einer Dienstleistung und des zu benutzenden Protokolls ist essentiell, so kann ein Protokoll von verschiedenen nicht semantisch verwandten Diensten benutzt werden, die Dienstleistung eines Programms ist allerdings eindeutig.

2. Entwicklungswerkzeuge

In diesem Kapitel werden die herkömmlichen und verbreitetsten Programmier- und Entwicklungswerkzeuge behandelt.

2.1. Editoren und Integrierte Entwicklungsumgebungen

Für die Entwicklung von Software unerlässlich ist ein Editoren oder eine IDE. Hierbei handelt es sich um Werkzeuge zum Schreiben von Quellcode. In dieser Arbeit wird auf keinen speziellen Editor und keine spezielle IDE eingegangen, stattdessen soll die Notwendigkeit eines solchen Werkzeuges verdeutlicht werden.

IDEs sind Sammlungen von Werkzeugen, welche einen Editor beinhalten. Zudem gehört zu einer IDE

- Compiler bzw. Interpreter
- Linker
- Debugger
- Quelltextformatierung

[Wik15e]

IDEs sind oft auf die Entwicklung in einer bestimmten Programmiersprache ausgelegt, wie zum Beispiel die IDE "Code::Blocks", welche zur Entwicklung von "C" und "C++" ausgelegt ist. Demgegenüber stehen Editoren, welche eine Vielzahl von Programmiersprachen bedienen, allerdings nicht (oder nur über Erweiterungen) über die Eigenschaften einer IDE verfügen oder diese integriert haben. Zum Beispiel verfügt der Editor "vim" über Möglichkeiten zur Autovervollständigung von Quelltext in nur sehr rudimentärer Form, Compiler, Interpreter, Linker, Debugger oder Quelltextformatierung stellt er nicht zur Verfügung. Dennoch werden auch Editoren gern und weit verbreitet eingesetzt, da diese oft sehr individuell anpassbar sind.

Tabelle 1.: Prominente IDEs

IDE	Programmiersprachen
Code::Blocks	C, C++
Eclipse	Java, C++, theoretisch alle
KDevelop	C, C++, PHP, Python
NetBeans	Java, JavaScript, Python, C, C++, Ruby, UML, PHP, Groovy, Scala, Clojure
Qt Creator	C++ mit Qt
Visual Studio	C, C++, C#, Visual Basic

Prominente Editoren sind

- vim
- emacs
- TextMate
- Kate
- Notepad++

2.2. Compiler/Interpreter

Tabelle 2.: Beispiel: Compiler von reinen Compilerspachen

Compiler	Sprache
GNU Compiler Collection (GCC)	C, C++, Objective-C, Objective-C++, Fortran, Ada, Go
Clang	C, C++, Objective-C, Objective-C++
dmd	D
Borland C++	C, C++
GHC	Haskell

Compiler werden benutzt um Quelltext in ausführbare Programme zu übersetzen, während Interpreter den Quelltext während der Ausführung in Maschinenbefehle übersetzen. Naturgemäß sind interpretierte Programmiersprachen deswegen tendenziell langsamer als Compilersprachen, beinhalten dafür allerdings Sprachkonstrukte, die mit Compilersprachen nicht möglich sind (z.B. Dynamische Bindung).

Oft werden Compiler eingesetzt um eine Zwischensprache (sogenannter "Bytecode") zu erzeugen, welcher dann zur Laufzeit interpretiert wird. Dies ermöglicht die Plattformunabhängigkeit einer Interpretersprache mit den Performanceeigenschaften einer Compilersprache zu kombinieren.

Tabelle 3.: Beispiel: Compiler für Bytecode-Sprachen

Compiler	Sprache
GCC	Java
JScheme	Scheme
Racket	Racket

Tabelle 4.: Interpreter für Interpretersprachen

Interpreter	Sprache
mruby	Ruby
jruby	Ruby
rubinius	Ruby
cpython	Python
PyPy	Python
perl	Perl
php	PHP
HipHop	PHP
GNU Smalltalk	Smalltalk
Squeak	Smalltalk

Tabelle 2 zeigt, dass Compiler oft nicht nur für eine einzelne Programmiersprache genutzt sondern können mehrere Programmiersprachen in ausführbare Dateien übersetzen. Oft existieren mehrere Compiler für die gleiche Programmiersprache, wie auch mehrere Interpreter für die gleiche Interpretersprache existieren (siehe Tabelle 3, Tabelle 4).

2.3. Build-Tools

Sogenannte "Build-Tools" werden in der Softwareentwicklung verwendet um Quellcode automatisiert in ausführbare Programme zu übersetzen. In diesem Kapitel wird der Begriff "Build-Tool" verwendet, welcher auch im deutschsprachigen Raum verbreiteter ist als äquivalente deutsche Begriffe.

2.3.1. Make

Als erster Vertreter der Palette der Build-Tools ist "Make" zu nennen, da andere Werkzeuge darauf aufbauen.

"Make", oder vielmehr "GNU Make" (da dies die weitverbreitetste Implementation des zugrundeliegenden Konzepts ist), ist ein Werkzeug, welches Abhängigkeiten

zwischen Aufgaben auflöst und diese Aufgaben dann ausführt.

Listing 2.1: Einfaches Makefile

```
all: ausgabe

ausgabe:
	echo "Hallo"
```

Listing 2.1 zeigt ein einfaches Makefile, welches zwei Aufgaben enthält: "all" und "ausgabe". Letztere Aufgabe benutzt dabei das Unix-Programm "echo" um den Text "Hallo" auszugeben. Die Aufgabe "all" hat zur Ausführung die Abhängigkeit "ausgabe". "make" kann nun feststellen, dass es erst "ausgabe" ausführen muss, um danach "all" ausführen zu können. So können Abhängigkeiten in Makefiles angelegt werden.

Diese Vorgehensweise kann nun benutzt werden um Quellcodedateien mittels eines Compilers in ein ausführbares Programm zu übersetzen. Ein einfaches Makefile compiliert dabei (zum Beispiel mittels des "gcc" Compilers) eine Menge an Quellcodedateien in sogenannte "Objectfiles" um diese dann mittels eines Linkers in eine ausführbare Programmdatei zu übersetzen. Listing 2.2 zeigt ein Beispiel für ein einfaches Makefile welches oben beschriebenes durchführt.

Listing 2.2: Makefile für C

```
CC=gcc
SOURCES=$(wildcard src/*.c)
HEADERS=$(wildcard include/*.h)
OBJECTS=$(patsubst src/%, build/%, $(SOURCES:.c=.o))
EXECUTABLE=programm
CFLAGS=-std=c11 -Wall -Wextra -Werror

build/%.o:src/%.c $(HEADERS)
	$(CC) -c $(CFLAGS) $< -o $@

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(CFLAGS) $(LDFLAGS) $(OBJECTS) -o $@

all: $(EXECUTABLE)

clean:
	rm -f $(OBJECTS) $(EXECUTABLE)

```

```
19 .PHONY: clean all
```

Die genaue Syntax von "GNU Make" kann in [Mec05] nachgelesen werden.

2.3.2. Autotools

Autotools ist ein Programm, welches den Entwickler im Übersetzungsprozess der Software unterstützen soll, indem Makefiles generiert und nicht mehr händisch produziert werden. Dabei lässt sich Autotools (insbesonderen Automake) zu Make vergleichen wie "C" zu "C++" [Cal10, Seite xviii, f.]. Bei diesem Vergleich ist der Ansatzpunkt des Build-Tools gemeint: Make setzt an einer bestimmten Komplexitätsebene im Build-Prozess an. Es wird verwendet um sehr detailiert zu definieren, wie welche Datei in welches andere Format übertragen wird und abstrahiert dabei Compileraufrufe und Linkeraufrufe in "Tasks", welche voneinander abhängig sind. Autotools setzt eine Abstraktionsebene weiter "oben" an. Bei Autotools wird projektweit definiert, wie die Software als Ganzes zu erstellen ist. Es wird nicht definiert, welche Tasks ausgeführt werden müssen um ein gewünschtes Ergebnis zu erzeugen, vielmehr wird nur das Gewünscht Ergebnis definiert, die Eingangsvorraussetzungen sowie die Werkzeuge und Autotools findet von selbst heraus, wie es zum Ziel kommt. Dieses Schema passt auch auf CMake.

Die Werkzeug-Suite "Autotools" besteht aus mehreren Komponenten:

- Autoconf, welches dazu genutzt wird ein Konfigurationsscript für das jeweilige Projekt zu generieren
- Automake, welches dazu genutzt wird ein oder mehrere Makefiles zu generieren
- Libtool, welches eine Abstraktion bietet um portierbare Shared Libraries zu erzeugen

Ein Paket mit dem Namen "Autotools" gibt es nicht, der Name wird als Referenz auf diese Werkzeuge verwendet [Cal10, etwa, Seite 1].

Listing 2.3: Beispiel: Makefile.am

```
1 # Gibt die Möglichkeit, "#include 'foo.hpp'" von überall zu benutzen, auch vom
2 # "tests" Ordner
3 AM_CPPFLAGS = -I$(top_srcdir)/src
4
```

```
5 # Nicht-rekursives Makefile.am für Autotools.
6 # Erzeugte Binärdateien verbleiben im Top-Level Ordner des Projektes
7 bin_PROGRAMS = foo
8
9 # Quelldateien
10 foo_SOURCES = \
11   src/main.cpp \
12   src/foo.hpp
13
14 # Extra Compileraufruf-Optionen
15 #foo_CXXFLAGS = -pedantic
16
17 # Testing setup. 'make check' oder 'make recheck' kann zum testen benutzt
18 # werden. TESTS definiert eine Liste von tests. Quellcodedateien werden wie
19 # üblich definiert
20 TESTS = passing failing
21
22 passing_SOURCES = tests/passing_test.cpp
23 failing_SOURCES = tests/failing_test.cpp
24
25 # Um ./configure lauffähig zu machen
26 noinst_PROGRAMS = passing failing
```

[Dan15]

In Listing 2.3 ist die Definition für Autotools zu sehen, welche benutzt werden kann um das Projekt zu konfigurieren. In dieser Datei wird festgelegt, welche Quellcodedateien zu berücksichtigen sind, welche Dateien für Tests relevant sind und welche Compilerflags verwendet werden sollen.

Listing 2.4: Beispiel: configure.ac

```
1 # Initialisierung mit Paketname, version, mailtainer email, etc.
2 AC_INIT([foo], [0.0.1], [foo@gmail.com], [foo], [http://www.example.com])
3
4 # Benötigte Autoconf version
```

```
5  AC_PREREQ([2.68])
6
7  # Temporäre Dateien in Unterordner "verstecken"
8  AC_CONFIG_AUX_DIR([build-aux])
9  AC_CONFIG_MACRO_DIR([m4])
10
11 # Initialisierung von Automake.
12 # subdir-objects: Erstelle Objekt-Dateien in Unterordnern
13 # foreign: GNU checks auflockern
14 # -Wall, -Werror: Instruktionen an Automake, nicht den Compiler
15 AM_INIT_AUTOMAKE([subdir-objects foreign -Wall -Werror])
16
17 # Generiere diese Dateien
18 AC_CONFIG_FILES([Makefile])
19
20 # Sicherheitscheck - Diese Datei muss unter diesem Pfad existieren
21 AC_CONFIG_SRCDIR([src/main.cpp])
22
23 # Konfigurationen in dieser Datei erzeugen, kann in C-Quellcode
24 # verwendet werden
25 AC_CONFIG_HEADERS([config.h])
26
27 # Sucht einen C++ Compiler
28 AC_PROG_CXX
29
30 # Letzte Zeile: Generierung starten
31 AC_OUTPUT
```

[Dan15]

Die Datei in Listing 2.4 definiert, wie das Projekt aus dem Quellcode in ein Binärprogramm übersetzt wird. Hier wird der Build-Prozess für das "Makefile" an sich konfiguriert. Ausserdem wird eine Datei "config.h" angelegt, die Projektinformationen beinhaltet, welche dann in den Quellcode des Projektes selbst eingebunden werden kann.

2.3.3. CMake

> CMake (cross-platform make) ist ein plattformunabhängiges Programmierwerkzeug für die Entwicklung und Erstellung von Software.

[Wik15d]

CMake kann dazu benutzt werden Makefiles zu generieren. CMake bedient sich dabei wie Autotools (2.3.2) einer Konfigurationsdatei, welche im Projektverzeichnis abgelegt wird und mit einer entsprechenden Skriptsprache befüllt wird. Diese Skriptsprache kann CMake dann übersetzen und damit das Quellcodeprojekt in eine ausführbare Binärdatei übersetzen. CMake ist ebenfalls als Kollektion von Werkzeugen zu verstehen:

- "DART"
- "CDash"
- "CTest"
- "CPack"

([Wik15d])

Der Unterschied zu 2.3.2 besteht darin, dass CMake keine Zwischendatei ("configure") erzeugt und "Make" direkt vom Nutzer aufgerufen werden kann. Das Erstellen einer Binärdatei aus Quellcode ist mit CMake in einem Schritt erledigt. Zudem gibt es bei "CMake" nicht die Notwendigkeit mehrere Konfigurationsdateien im Quellcoderepository abzulegen.

Listing 2.5: Beispiel: CMakeLists.txt

```
1  cmake_minimum_required(VERSION 2.6)
2  project(cpp11)
3
4  # Variablen:
5  set(WITH_TEST ON CACHE BOOL "Enable Test")
6
7  # Einstellungen:
8  set(CMAKE_THREAD_PREFER_PTHREAD ON)
9  if(CMAKE_COMPILER_IS_GNUCXX)
10   set(CMAKE_CXX_FLAGS "-std=c++11")
11 endif(CMAKE_COMPILER_IS_GNUCXX)
```

```
12
13 # Abhängigkeiten finden:
14 find_package(Threads)
15
16 if(WITH_TEST)
17   message(STATUS "Test is Enabled")
18
19   enable_testing()
20   find_package(GTest REQUIRED)
21 endif(WITH_TEST)
22
23 add_subdirectory(src bin)
24 add_subdirectory(test test)
```

[Yan15]

In Listing 2.5 wird ein Projekt für die Programmiersprache "C++11" definiert. Auffällig im Vergleich zu Autotools ist die sehr viel einfachere Syntax sowie die Länge der Datei. CMake wird nur mit dem Quellcodeverzeichnis sowie dem Verzeichnis für Test-Sources konfiguriert, es besteht keine Notwendigkeit die exakten Quellcodedateien zu definieren. Zudem wird der Compiler in diesem Beispiel nicht explizit definiert, CMake findet den Compiler selbst. Ausserdem wird die Abhängigkeit "Threads", sowie die Abhängigkeit "GTest" definiert, letztere allerdings nur falls die Tests eingeschaltet sind.

Der kompakten Syntax von CMake steht die Flexibilität von Autotools gegenüber. Letztendlich muss der Entwickler oder Projektleiter entscheiden, welches der beiden Tools für das jeweilige Projekt von Vorteil ist.

2.4. Debugger

Debugger werden dazu verwendet, das bereits übersetzte Programm während seiner Laufzeit zu entschleunigen und Schritt für Schritt auszuführen. Dabei werden dem Programmierer Möglichkeiten gegeben, Werte von Variablen einzusehen und gegebenenfalls auch zu verändern, um den Programmablauf genau zu analysieren. Zudem bieten Debugger oft Möglichkeiten den Quelltext während des Debugging-Vorganges einzublenden.

Bekannte Debugger sind

- Firefox JavaScript debugger
- GNU Debugger (GDB)
- LLDB
- Microsoft Visual Studio Debugger
- valgrind

Debugger sind dabei oftmals an eine Programmiersprache gebunden und können nur mit Quelltext dieser Programmiersprache umgehen. Der Terminus "Debugger" wird dabei auch für Werkzeuge verwendet, welche nicht für Compilersprachen zugeschnitten sind. So existieren beispielsweiße Debugger für "JavaScript" oder "Python".

Manche Debugger können zudem verwendet werden um automatisiert Speicherschwächen des Programms, wie zum Beispiel nicht freigegebenen Speicher, zu sammeln ("valgrind"). Diese spezielle Art von Debuggern nennt man "Memory-Profiler". Sie dienen zum Analysieren der Speicherbelegung eines Prozesses und werden deswegen nur für Sprachen verwendet, bei denen Adressierung von Speicher gegeben ist (keine Interpretersprachen).

3. Testen von Software

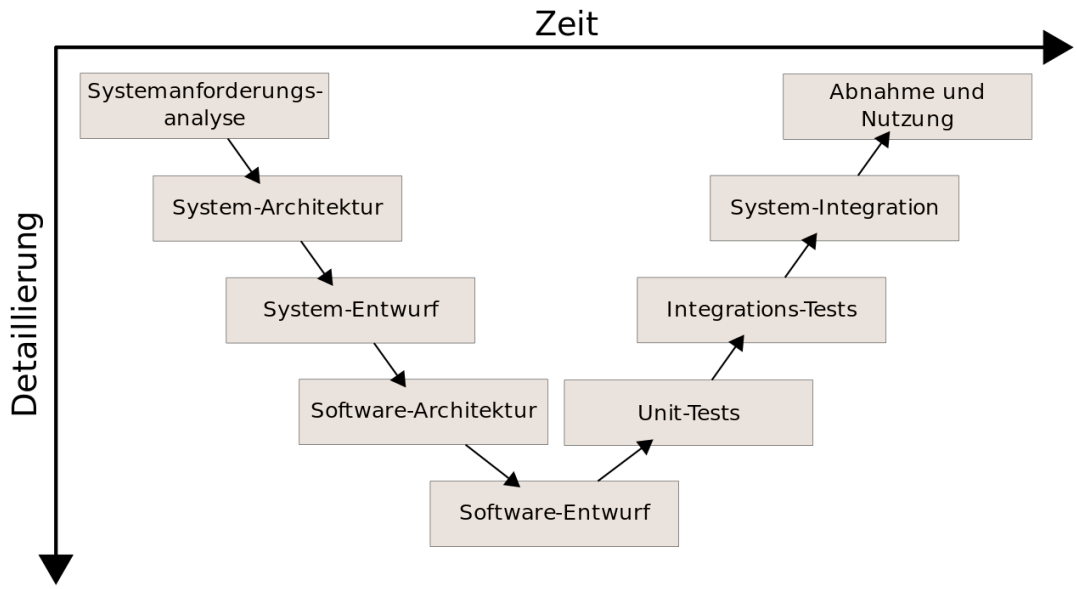

Abbildung 1.: Stufen des V-Modells
[Wik15g]

Beim Testen von Software werden verschiedene Teststufen durchlaufen. Jede Stufe beschreibt eine Art von Test, welcher verwendet wird. Diese Stufen können mit einer Entwicklungsstufe assoziiert werden, wie zum Beispiel in Abbildung 1 dargestellt.

Die einzelnen Teststufen sind:

- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

[Wik15g]

3.1. Komponententest

Bei dieser Teststufe werden einzelne Komponenten eines Programmes getestet. Die Komplexität der Komponenten ist dabei variabel und reicht von einzelnen Funktionen eines Programmes bis hin zu ganzen Programmteilen und Klassen, Unterprogrammen, Modulen oder Units. Oft wird der Terminus "Unittest" als Synonym für den Komponententest verwendet.

3.2. Integrationstest

Beim Entwickeln größerer Softwareprojekte werden oft mehrere Module zu einer Software zusammengebaut. Um die Funktionsfähigkeit der einzelnen Komponenten zu testen wird der Komponententest verwendet, wobei der Integrationstest darauf abzielt, die Interaktion zwischen den Modulen fehlerfrei zu gewährleisten.

Der Integrationstest kann dabei aus einer Reihe von Einzeltests bestehen, welche das fehlerfreie Interagieren zwischen mehreren Softwarekomponenten sicherstellen.

Oft wird als Vorraussetzung für einen Integrationstest der fehlerfreie Abschluss des Komponententest für alle Module vorrausgesetzt.

3.3. Systemtest

> Der Systemtest [...] ist ein Projekt für sich, ein Projekt, das neben dem eigentlichen Entwicklungs- bzw. Integrations-, Migrations- oder Installationsprojekt läuft. Als solches muss es wie andere Projekte definiert, kalkuliert, organisiert und geplant werden.

[SBS08, S. 45]

Beim Systemtest wird die komplette Entwicklung gegen die Anforderungen getestet. Dabei sind sowohl funktionale als auch nicht-funktionale Anforderungen zu berücksichtigen.

3.4. Abnahmetest

Beim Abnahmetest wird die zu entwickelnde Software vom Auftraggeber getestet, bevor sie produktiv zum Einsatz kommen soll.

> Der erfolgreiche Abschluss dieser Teststufe ist meist Voraussetzung für die rechtswirksame Übernahme der Software und deren Bezahlung.

[Wik15g]

3.5. Testing-Frameworks

Beim Testen von Software kommen je nach Programmiersprache verschiedene Tools zum Einsatz. Darunter vertreten sind nicht nur Testing-Frameworks, sondern auch Memory-Profiler, Debugger, Laufzeitanalysesysteme, Coverage-Werkzeuge und viele mehr.

Testing-Frameworks sind Bibliotheken, welche in das zu testende Programm integriert werden um automatisiert Funktionalitäten auf Fehler zu untersuchen. In der Entwicklung mit der Programmiersprache "C" oder "C++" sind Unit-Testing-Frameworks wie zum Beispiel "ctest", "gtest", "check", "cutter", "Boost Test Library", "CppUnit" und "cunit" verbreitet. Unit-Testing-Frameworks testen Einzelfunktionalitäten wie Funktionen oder Methoden. Oft werden zu Testing-Frameworks sogenannte "Mocking"-Frameworks hinzugezogen, welche in der Lage sind Testdaten aus einer Sammlung an Spezifikationen für diese zu generieren. Sollte eine Schnittstelle für eine zu testende Einheit zum Beispiel aussehen wie in Listing 3.1 beschrieben, so kann ein solches "Mocking"-Framework anhand einer Spezifikation, wie die entsprechenden Typen zu generieren sind, Beispieldaten für den zu dieser Funktion gehörenden Unit-Test generieren, welche dann in selbigem benutzt werden kann.

Listing 3.1: C++ Funktionsprototyp

```
1 std::function<std::shared_ptr<lib::Label>(int, std::
    string)> getLabelGetterCallback(std::shared_ptr<lib::
    Text>, std::string);
```

Ein bekanntes Mocking-Framework für die Programmiersprache "Ruby" ist zum Beispiel "mocha", welches unter anderem im Umfeld von "Ruby on Rails" Applikationen eingesetzt wird.

"Ruby" und "mocha" wurden hier aufgrund der extrem einfachen Syntax als Beispiel gewählt.

Listing 3.2: Ruby Programm

```ruby
class Document
  def print
    # doesn't matter -- we are stubbing it out
  end
end

class View
  def initialize(document)
    @document = document
  end

  def print
    # rufe 'print' auf 'document', konvertiere Ergebnis zu boolean
    !!@document.print
  end
end
```

Listing 3.2 beschreibt ein einfaches Programm, welches eine Klasse "Document" und eine Klasse "View" enthält. Letztere wird in Listing 3.3 getested, wobei anstatt der Klasse "Document" ein sogenanntes "Stub-Objekt" verwendet wird - ein Objekt, welches nur so tut, als wäre es ein Objekt des Typs "Document".

Listing 3.3: Ruby Programm - Test

```ruby
require 'test/unit' # Ruby Unit-testing
require 'rubygems' # Ruby Bibliothekverwaltung
require 'mocha' # Mocking-Framework
require 'document' # Programm

class ViewTest < Test::Unit::TestCase
  def test_should_return_false_for_failed_print
    document = stub("my document") # Stub-Objekt anlegen
    document.stubs(:print).returns(false) # Stub-Objekt mit Methoden ausrüsten

    ui = View.new(document) # View-Objekt anlegen
```

```
12      assert_equal false, ui.print # View-Object testen
13    end
14  end
```

Aufgrund der Eigenschaften der Programmiersprache "Ruby" (in diesem speziellen Fall die Eigenschaft der dynamischen Typisierung) ist das Erzeugen von solchen Objekten sehr einfach. In anderen Sprachen kommen für Mocking-Frameworks spezielle Techniken zum Einsatz um die Erstellung solcher Stub-Objekte zu ermöglichen, sofern das Typensystem der jeweiligen Programmiersprache es erlaubt. Sollte die Programmiersprache eine solche Technik wie dynamische Typisierung nicht unterstützen, so werden Daten für die entsprechenden Schnittstellen generiert. Beispiele hierfür sind "FactoryGirl" für "Ruby" [tho15] oder "QuickCheck" für "Haskell".

Listing 3.4: Haskell Quickcheck: Generierung von Daten

```
1 import Data.Char
2 import Test.QuickCheck
3
4 instance Arbitrary Char where
5   arbitrary     = choose ('\32', '\128')
6   coarbitrary c = variant (ord c `rem` 4)
```

[Wik15a]

Listing 3.4 zeigt wie Daten für den Datentyp "Char" mittels "QuickCheck" generiert werden können.

3.6. CTest und GTest

In der Vorlesung, zu welcher diese Arbeit erstellt wurde, wurden zwei Testing-Frameworks genauer besprochen:

- ctest
- gtest

Diese sollen in folgendem vorgestellt und verglichen werden.

3.6.1. ctest

CTest ist ein Testing-Tool welches im Verbund mit CMake (2.3.3) vertrieben wird. Es kann für verschiedenste Aktivitäten genutzt werden:

- Automatisiertes Updaten eines Quellcodeverzeichnises (zum Beispiel CVS)
- Konfigurieren
- Übersetzen
- Testen
- Speicherüberprüfung
- Quellcodeüberdeckung

CTest kann ausserdem Ergebnisse an ein sogenanntes "Dashboard-System" übermitteln - im Normalfall eine Website welche solche Daten aufbereitet anzeigen kann.

CTest hat zwei Modi in welchen es betrieben werden kann, wobei in dieser Arbeit ausschließlich auf den ersten der beiden eingegangen werden soll, das konfigurieren und übersetzen eines Projektes welches Tests generiert und ausführt.

Listing 3.5: CMake: Testing einschalten

```
1 enable_testing()
```

[cma15]

Dieses Modi wird in CMake mittels dem Befehl in 3.5 eingeschaltet (siehe auch [Yan15], 2.5, hier wird allerdings "GTest" zum Testen verwendet).

Mittels dem Befehl

Listing 3.6: CMake: Test hinzufügen

```
1 add_test(mytest testDriver)
```

[cma15]

können nun Tests zum Projekt hinzugefügt werden. Dabei beschreibt das erste Argument für diesen Aufruf den Namen des Tests, der zweite das Kommando welches für den Test aufgerufen werden soll. Optional können nach dem zweiten Argument noch Parameter für das Kommando folgen. Es ist egal wie diese Tests geartet sind. Sie können mittels CMake wie auch der Rest des Projektes aus dem Quellcode übersetzt

werden oder auch nur aus einfachen Skripten bestehen welche dem Projekt beiliegen. Dies ermöglicht zum Beispiel auch den Aufruf von automatisierten Speicherbelegungstests mittels "valgrind" (2.4).

3.6.2. gtest

"GTest" oder auch "googletest" ist ein Unit-Testing Framework für die Programmiersprachen "C" und "C++". GTest ist dabei für verschiedene Betriebssysteme verfügbar:

- Linux
- Mac OS X
- Windows
- Cygwin
- MinGW
- Windows CE
- Symbian

[Goo15c]

Listing 3.7: Gtest: Einfache Tests

```
1  // Tests factorial of 0.
2  TEST(FactorialTest, HandlesZeroInput) {
3      EXPECT_EQ(1, Factorial(0));
4  }
5
6  // Tests factorial of positive numbers.
7  TEST(FactorialTest, HandlesPositiveInput) {
8      EXPECT_EQ(1, Factorial(1));
9      EXPECT_EQ(2, Factorial(2));
10     EXPECT_EQ(6, Factorial(3));
11     EXPECT_EQ(40320, Factorial(8));
12 }
```

[Goo15a]

GTest beinhaltet eine Vielzahl an Funktionalitäten zum Testen von Quellcode wie zum Beispiel nutzerdefinierte Tests oder parametrisierte Tests.

So können mittels verschiedenen Präprozessormakros "Assertions" erstellt werden, welche zum Beispiel Numerische Werte, Texte oder auch komplexere Datenstrukturen auf Korrektheit überprüfen. GTest ist dabei für "C" genauso geeignet wie für "C++", wobei bei letzterer Sprache zusätzlicher Aufwand betrieben werden muss, da spezielle Testing-Klassen angelegt werden müssen (Vergleich 3.7 zu 3.8, oder [Goo15b]).

Listing 3.8: Gtest: C++

```
1  // Interface der zu Testenden Klasse:
2  template <typename E>
3  class Queue {
4      public:
5          Queue();
6          void Enqueue(const E& element);
7          E* Dequeue(); // Returns NULL if the queue is
               empty.
8          size_t size() const;
9          ...
10 };
11
12 // Test-Adapter-Klasse:
13 class QueueTest : public ::testing::Test {
14     protected:
15         virtual void SetUp() {
16             q1_.Enqueue(1);
17             q2_.Enqueue(2);
18             q2_.Enqueue(3);
19         }
20
21         // virtual void TearDown() {}
22
23         Queue<int> q0_;
24         Queue<int> q1_;
25         Queue<int> q2_;
26 };
27
28 // Einfacher Test einer leeren Queue:
29 TEST_F(QueueTest, IsEmptyInitially) {
```

```
30     EXPECT_EQ(0, q0_.size());
31 }
32
33 // Aufwendigerer Test Queue:
34 TEST_F(QueueTest, DequeueWorks) {
35     int* n = q0_.Dequeue();
36     EXPECT_EQ(NULL, n);
37
38     n = q1_.Dequeue();
39     ASSERT_TRUE(n != NULL);
40     EXPECT_EQ(1, *n);
41     EXPECT_EQ(0, q1_.size());
42     delete n;
43
44     n = q2_.Dequeue();
45     ASSERT_TRUE(n != NULL);
46     EXPECT_EQ(2, *n);
47     EXPECT_EQ(1, q2_.size());
48     delete n;
49 }
```

[Goo15a]

Ein einfacher Test mit GTest ist in Listing 3.7 dargestellt. GTest stellt Präprozessormakros zur Verfügung um Tests einfach definieren zu können.

GTest ist neben "Check" und "CppUnit" eines der bekanntesten Testing-Frameworks für C++ und wird unter anderem von dem Chromium-Projekt, LLVM sowie OpenCV verwendet.

3.6.3. Vergleich

CTest und GTest stehen zueinander eher orthogonal. Während CTest ein Werkzeug zum automatisierten Testen darstellt, ist GTest ein Testing-Framework zum erstellen von Quellcodetests, somit hat der Aufgabenbereich von CTest keine Schnittmenge mit dem Aufgabenbereich von GTest. Die beiden Werkzeuge lassen sich allerdings kombinieren um automatisiert Quellcodetests durchzuführen: Mittels GTest werden die Quellcodetests als Unittests definiert, mittels CTest werden diese dann übersetzt und automatisiert ausgeführt.

4. Versionskontrolle

Bei der Entwicklung von Quellcode arbeiten mehrere Entwickler, oft auch ganze Teams über Wochen und Monate hinweg an einem Projekt. Um die Arbeit dieser Entwickler zu synchronisieren und auch über Kontinente hinweg möglich zu machen, werden VCS eingesetzt. Doch nicht nur zum verteilten Arbeiten, auch um den Quellcode gegen Lücken zu sichern, nachzuvollziehen wer wann welche Änderungen in das Projekt eingebracht hat und zur automatische Fehlersuche können VCS beitragen.

> [Als Beispiel für ein solches Project ist] der Linux-Kernel zu nennen, die Open-Source-Alternative zu Microsoft Word, Open Office oder Libre Office, das Android-Projekt, Facebooks HipHop-Compiler, Ruby on Rails, die Desktopumgebung KDE und viele weitere Bibliotheken, Programme und Plattformen [...]

[Bey13]

Dabei sind drei Arten von VCS zu unterscheiden:

- Lokale Versionsverwaltung
- Zentrale Versionsverwaltung
- Verteilte Versionsverwaltung

[Wik15j]

Zu den Anforderungen an moderne VCS gehören

- Protokollierung der Änderungen
- Wiederherstellung von alten Versionen einzelner Dateien
- Archivierung der einzelnen Versionen eines Projektes
- Koordinierung des gemeinsamen Zugriffs von mehreren Entwicklern
- Gleichzeitige Entwicklung von mehreren Entwicklungszweigen

[Wik15j]

Da lokale VCS per Definition keine Möglichkeit für den gemeinsamen Zugriff durch mehrere Entwickler bieten und zentrale VCS oft Mängel bezüglich Performanz, Möglichkeiten zur gleichzeitigen Entwicklung oder Verifizierung der Integrität des Quellcodes aufweisen, sind verteilte VCS in den Jahren 2011 bis 2013 immer populärer geworden. Dies ist in Abbildung 6 visualisiert.

Vertreter zentralisierter VCS sind

- RCS
- CSV
- Subversion
- Perforce
- ...

Vertreter verteilter VCS sind

- git
- GNU arch
- Darcs
- Marcurial
- ...

[Wik15j]

Aus diesen Gründen hat das zentralisierte VCS "Subversion" zwischen den Jahren 2011 und 2013 stark an Popularität verloren, während "git", ein verteiltes VCS, stark an Popularität gewonnen hat (Abbildung 7). Dieser Trend wird in den kommenden Jahren verstärkt fortgeführt ([Ind16]).

Nicht nur die Art des VCS ist bei der Auswahl entscheidend, auch die Geschwindigkeiten der verschiedenen Operationen des VCS unterscheiden sich stark von VCS zu VCS. So ist zum Beispiel "git" in einigen Anwendungsfällen sehr viel schneller als Mitbewerber (Tabelle 5).

Tabelle 5.: Geschwindigkeitsvergleich zwischen Git und Subversion

Operation	Git (Sekunden)	Subversion (Sekunden)	Multiplikator
Commit Files	0.64	2.60	4x
Commit Images	1.53	24.70	16x
Diff Current	0.25	1.09	4x
Diff Recent	0.25	3.99	16x
Diff Tags	1.17	83.57	71x
Log 50	0.01	0.38	31x
Log All	0.52	169.0	325x
Log File	0.60	82.84	138x
Update	0.90	2.82	3x
Blame	1.91	3.04	1x

Quelle: [Bey13], [Git13]

5. Softwarearchitektur

> [Es gibt] widersprüchliche Vorstellungen darüber, was unter dem Begriff [Softwarearchitektur] eigentlich zu verstehen ist.
>
> [VAC+08, S. 8]

Die Definition des Begriffs "Softwarearchitektur" ist schwierig und ungenau. Das Software Engineering Institude der Carnegie Mellon University listet mehrere Definitionen des Begriffes ([sei15]) auf.

Im Groben kann man eine Architektur in der Welt der Software als Unterteilung der Software in unabhängige Komponenten verstehen. Diese Komponenten sollten möglichst wenig "gebunden", also wiederverwendbar und unabhängig von anderen Komponenten sein.

Die Komponenten selbst haben auch eine Architektur und bestehen wiederum aus Komponenten. Diese Unterteilung in immer kleinere Komponenten kann bis zur kleinstmöglichen Einheit fortgeführt werden, welche dann eine sehr spezifische Aufgabe erfüllt.

Bei der Zerlegung einer Komponente sollte darauf geachtet werden, dass zwischen den Komponenten untereinander keine starken Bindungen entstehen, um die Wiederverwendbarkeit der ausgegliederten Komponenten zu garantieren. Wiederverwendbarkeit spielt in der Softwarearchitektur eine große Rolle. Ist eine Softwarekomponente wiederverwendbar und kann dadurch in mehreren Stellen sinnvoll eingesetzt werden, spart dies Entwicklungszeit und somit Geld und trägt maßgeblich zur Quellcodequalität und -wartbarkeit bei.

Es gibt verschiedene Architekturmuster, welche in der Softwareentwicklung zum Einsatz kommen. Dabei handelt es sich um verschiedene Stile, wie die Komponenten eines Softwaresystems unterteilt werden. Zu bekannten Architekturmustern gehören

- Funktionsorientierte Architektur

  > [...] jeder Modul des Systems eine Funktion [berechnet] eine Funk-

> tion [...] [sic]
>
> [Wal, S. 2]

- Datenorientierte Architektur

  > Im Zentrum der datenorientierten Architektur steht eine Modularisierung nach dem Geheimnisprinzip. [...] Nach diesem Stil entworfene Systeme sind aufgrund der Eigenschaften des Geheimnisprinzips leicht änderbar und vor allem im Großen gut verstehbar.
  >
  > [Wal, S. 2]

- Objektorientierte Architektur

  > Im objektorientierten Entwurf wird ein System als eine Menge kooperierender Objekte aufgefasst. Wie eine Datenabstraktion besteht ein Objekt aus Daten und allen darauf möglichen Operationen. Objekte bzw. Klassen (d.h. Objekttypen) bilden die Einheiten der Modularisierung
  >
  > [Wal, S. 3]

- Prozessorientierte Architektur

  > Nebenläufige Systeme bestehen aus einer Menge unabhängig arbeitender, untereinander kooperierender Akteure, die typisch als Prozesse realisiert sind. Die Prozesse kooperieren durch Austausch von Nachrichten oder durch Zugriff auf gemeinsame Speicherbereiche. Die Prozesse können auf einem Rechner ablaufen oder auf verschiedene Rechner verteilt sein. In prozessorientierten Architekturen sind die Prozesse die Module der obersten Stufe.
  >
  > [Wal, S. 3]

- Komponentenorientierte Architektur

  > In einer komponentenorientierten Architektur versteht man unter einer Komponente eine Menge zusammengehöriger Objekte bzw. zusammengehöriger Klassen, die von ihrer Umgebung abgekapselt und nur über eine oder mehrere Schnittstelle(n) der Komponente zugänglich sind.

[Wal, S. 3]

Weitere, konkretere Defintionen beinhalten

- Pipes und Filter
- Schichtenarchitektur
- Serviceorientierte Architektur
- Client-Server
- Model-View-Controller
- Presentation-Abstraction-Control

[Wik15c]

5.1. Datenfluss

In einem Datenflussdiagram kann beschrieben werden aus welchen Datenquellen Daten ins System eingespeist werden. Zudem kann definiert werden welche Datensenken das System benutzt um Daten auszugeben.

Abbildung 8 zeigt ein Beispielsystem, welches verschiedene Datenquellen und -senken hat. Datenquellen sind grün, Datensenken türkis dargestellt.

Ein Datenflussdiagramm stellt die Art der Verwendung von Daten dar. Es wird dazu benutzt den Datenfluss eines Systems zu visualisieren und wiederzugeben. Es stellt keinen Kontrollfluss dar und definiert somit auch keine Operationen auf Daten.

5.2. Szenario

In dem in der Vorlesung beschriebenen Szenario handelt es sich um eine komplette Reorganisierung des Verkehrsbetriebes der Bahn in Karlsruhe. Es wurden folgende Voraussetzungen in der Vorlesung beschrieben:

- Abschnitte - also Streckenabschnitte, Weichen, Haltestellen - haben nun die Möglichkeit Daten zu generieren und zu verarbeiten (Sensoren/Aktoren).
- Es gibt eine Leitzentrale

- Blocksicherung muss berücksichtigt werden
- Die Kommunikation wird über TCP/IP realisiert, nicht UDP/IP

Es wurde definiert, dass Sensoren zur Fahrzeugerkennung die Fahrtrichtung selbstständig erkennen. Die Sensoren senden dabei ihre Global Positioning System (GPS) Koordinaten über das Protokoll, welches in dieser Arbeit definiert wird. IP und Port werden über die darunterliegenden Protokolle gesendet und sind nicht Teil dieser Ausarbeitung.

5.3. Architekturdefinition

Als Architekturmuster wurde für das in Abschnitt 5.2 beschriebene Szenario das Muster "Komponentenorientierte Architektur" gewählt. Die Problemstellung lässt sich in verschiedene Komponenten unterteilen, welche ihre Funktionalitäten für andere Komponenten über Schnittstellen anbieten. So kann ein Sensor zur Fahrzeugerkennung als Komponente definiert werden, welche eine wohldefinierte Schnittstelle zur Abfrage des aktuellen Status den anderen Komponenten anbietet.

5.4. Betrachtetes Teilproblem

In folgenden Kapiteln soll lediglich ein Teil des in Abschnitt 5.2 beschriebenen Szenarios betrachtet werden. Gewählt wurde hier das System, welches Daten über die aktuell auf einer Strecke befindlichen Fahrzeuge für einen Dienstnutzer bereitstellt.

6. Dienste

In der Vorlesung für welche diese Arbeit erstellt wurde, wurde im Rahmen des Kapitels "Dienste und Protokolle" ein Szenario entworfen, welches zur Verdeutlichung, Veranschaulichung und Unterscheidung zwischen Diensten und Protokollen dienen sollte (siehe Abschnitt 5.2).

In folgendem Kapitel soll unter Beachtung des Teilproblems aus Abschnitt 5.4 ein Dienst identifiziert und definiert werden.

6.1. Unterscheidung: Dienste und Protokolle

> Die Unterscheidung von Diensten und Protokollen ist wichtig, da ihnen verschiedene Konzepte zugrunde liegen, auch wenn sie oft gleichgestellt werden.

[Wol15]

Ein Dienst verkörpert eine Gruppe von Funktionalitäten, die der Aussenwelt zur Verfügung gestellt werden. Das heisst, dass ein Dienst zum Beispiel eine Datenübermittlung, eine Anzeige von Daten oder die Abspeicherung von Daten sein kann.

Ein Protokoll hingegen beschreibt das Format, mit welchem ein Dienst aufgerufen oder angesprochen werden kann. Weiter soll auf diesen Punkt in dieser Ausarbeitung nicht eingegangen werden.

6.2. Dienstidentifizierung anhand der Architekturdefinition

Die Einschränkung aus Abschnitt 5.4 erlaubt eine einfache Identifikation eines Dienstes. Das System, in welchem der Dienst definiert wird beinhaltet folgende Komponenten:

- Intern in dem zu beschreibenden Dienst

  - Sensor
  - Streckenabschnitt
- Extern des zu beschreibenden Dienst (Dienstnutzer)
  - Fahrzeug
  - Servicenutzer

Aufgabe des Dienstes ist es, einem Dienstbenutzer Daten über Fahrzeuge, welche sich aktuell im Streckenabschnitt befinden bereitzustellen. Ein Fahrzeug meldet sich über Sensoren am System an, die Daten werden dann vom Dienst an den Dienstbenutzer weitergeleitet. Der Dienst wird demnach von einem Streckenabschnitt zur Verfügung gestellt.

Zudem muss der Umgang mit Fehlern in der Dienstleistung genauer beleuchtet werden. Die Fehlerbehandlung sollte ausschließlich in der Entität "Streckenabschnitt" stattfinden, die Sensoren beinhalten keine Intelligenz und senden lediglich Fehlerkennungen.

6.3. Definition des Dienstes

Listing B.1 zeigt die Definition eines solchen Dienstes mit Sender und Empfänger. Die Dienstdefinition beinhaltet keine Defintion zum Verbindungsaufbau und -Abbau, da diese bereits durch den zugrundeliegenden TCP/IP-Mechanismus abgedeckt ist (Vorgabe, 5.2).

Die Daten, welche bereitgestellt werden sind nicht näher beschrieben, lediglich die Dienstleistung des Bereitstellens soll erklärt werden. Ein Fahrzeug kann dem Dienst Daten oder eine Fehlermeldung übermitteln. Beides wird an den Dienstnutzer weitergeleitet. Sollte ein Fehler bei der Nachrichtenübertragung Fahrzeug $\rightarrow$ Dienst auftreten, so wird dieser mit einer entsprechenden Antwort quittiert und die Daten müssen erneut gesendet werden.

Die Definition, wie und in welchem Datenformat der Dienst nun mit den externen Entitäten kommuniziert, ist Aufgabe der Protokolldefinition. Die Dienstdefinition legt lediglich fest, welche Daten bereitgestellt/kommuniziert werden.

7. Protokolle

Wie schon in Abschnitt 6.1 kurz beschrieben besteht ein Protokoll aus einem Datenformat und einer Ablaufdefinition, welche definiert wie die Daten ausgetauscht werden.

In Abbildung 2 ist aufgezeigt, wie die Datenformate "JavaScript Object Notation (JSON)" und "eXtensible Markup Language (XML)" und die Ablaufdefinition "Remote Procedure Call (RPC)" kombiniert werden können um ein Protokoll zu definieren, in diesem Fall zum Beispiel 'JSON-RPC" oder "XML-RPC".

Protokolle und Dienste leben in einer Symbiose, es kann kein Dienst existieren ohne dass ein Kommunikationsprotokoll für ihn existiert. Zudem ergibt es wenig Sinn dass ein Protokoll existiert welches in keinem Dienst benutzt wird, obwohl dies vorkommen kann.

7.1. Werkzeuge zur Protokolldefinition

Zur Definition eines Protokolls existieren verschiedene Werkzeuge, wovon drei in dieser Arbeit betrachtet werden sollen:

- ASN.1
- MSC
- SDL

Jedes dieser Werkzeuge legt den Fokus auf einen anderen Aspekt bei der Entwicklung eines Protokolls. Die Werkzeuge ergänzen sich also.

7.1.1. ASN.1

Bei ASN.1 handelt es sich um eine Beschreibungssprache für Datenstrukturen.

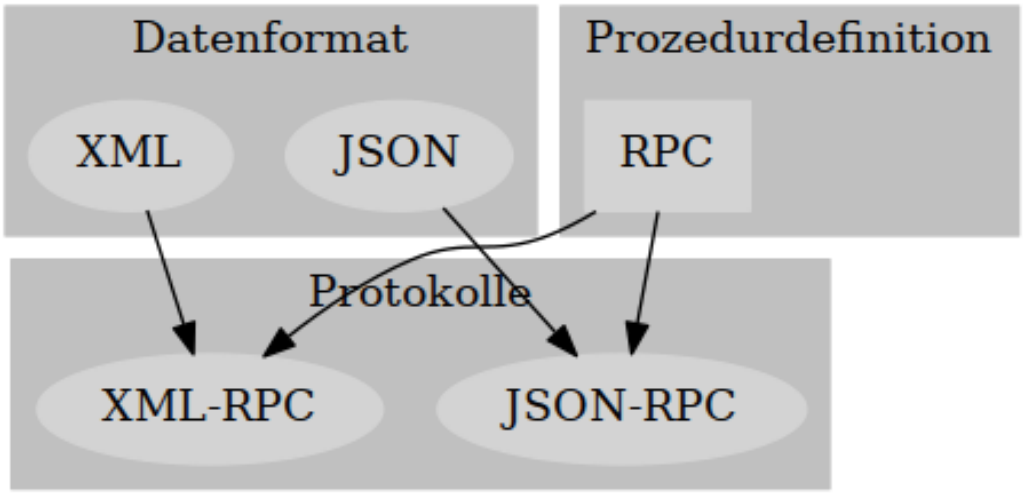

Abbildung 2.: Datenformate, Ablaufdefinitionen, Protokolle

> ASN.1 ist eine verbreitete Möglichkeit, die Nachrichtenelemente von Protokollen des OSI-Modells eindeutig zu beschreiben, und wird von OSI-konformen Techniken wie X.500 und X.509, aber auch von Internetprotokollen wie SNMP oder LDAP verwendet. Breite Anwendung findet ASN.1 auch im Telekommunikationsbereich, z. B. bei den Standards GSM für die Abrechnung von Roaminggesprächen in TAP3-Dateien und UMTS.

[Wik15b]

ASN.1 verwendet eine Backus-Naur-Form (BNF)-Ähnliche Syntax zur Definition der Datenformate und ist bereits mit einfachen Datentypen ausgetattet, welche die Definition komplexerer Datentypen vereinfacht. In ASN.1 existieren elementare Datentypen wie

- "BIT STRING“
- “BOOLEAN”
- “IA5String” (nach IA5-Tabelle kodierte Zeichenfolge, [Wik16c])
- “INTEGER”

[Wik15b]

Zudem existieren zusammengesetzte Datentypen wie

- "CHOICE“

- "SEQUENCE“
- "SEQUENCE OF <Typ>“
- "SET“
- "SET OF <Typ>“

[Wik15b]

und ein spezieller Datentyp “OBJECT IDENTIFIER” welcher zur eindeutigen Erkennung eines Objekts dient.

In Listing 7.1 wird mittels ASN.1 ein einfacher “Record” beschrieben.

Listing 7.1: ASN.1 Beispiel

```
1 Record ::= SEQUENCE {
2     kopf  Header,
3     daten Data OPTIONAL
4 }
5 Header ::= IA5String
6 Data ::= SET OF INTEGER
```

In Abbildung 2 würde ASN.1 dazu verwendet werden, JSON oder XML zu definieren.

7.1.2. MSC

> Ein Nachrichten-Reihenfolge-Diagramm [...] wird benutzt, um [...] beispielhafte Nachrichtenfolgen zwischen kommunizierenden Objekten einheitlich darzustellen.

[Wik15f]

Ein MSC kann dazu benutzt werden eine Ablaufdefinition einer Kommunikation mittels einer wohldefinierten und formal beschriebenen textualen Beschreibung zu erstellen (wie in [SM94, S. 3] aufgezeigt). Dabei ist zu beachten dass nur genau ein Ablauf einer Kommunikation beschrieben werden kann [SM94, S. 2].

> A Basic Message Sequence Chart contains a (partial) description of the communication behavior of a number of instance.

[SM94, S. 2]

Es existieren mehrere Dialekte der Beschreibungssprache.

Ein Verbindungsaufbau zwischen Client und Server würde in MSC/PR wie in Listing B.2 beschrieben, wobei Abbildung 3 die in grafische Darstellung übersetzte Repräsentation darstellt.

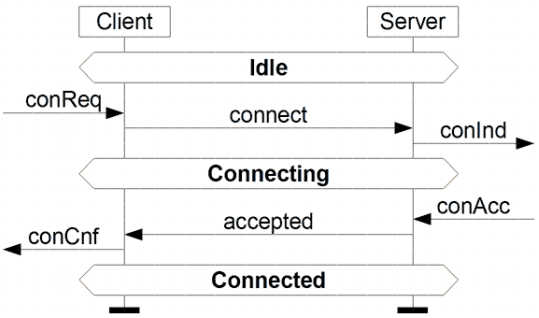

Abbildung 3.: MSC Beispiel, [Wik16d]

Bei der Erstellung dieses Dokuments kam MSC zum Einsatz (siehe Abbildung 10).

In Abbildung 2 würde MSC dazu verwendet werden, den Ablauf des Datenaustausches zu definieren, würde also in der Umgebung von RPC angesiedelt sein.

7.1.3. SDL

Bei SDL handelt es sich um eine Standardisierte Modellierungssprache welche vor allem im Telekommunikationsbereich verbreitet ist ([Wik15h]). In der Sprache existieren verschiedene Komponenten wie zum Beispiel

- Systeme
- Blöcke
- Prozesse
- Agenten
- Bibliotheken
- Prozeduren
- Kanäle
- Signale
- ...

[Wik15h] [Soc, S. 5 ff.]

welche zur Spezifikation eines Systems genutzt werden können. Zudem existieren in SDL vordefinierte Datentypen welche wiederum in

> Strukturen zusammengefasst [und] in Listen, Mengen etc. [sic] abgelegt werden können.

[Wik15h]

Die Charakteristiken von SDL sind in [Soc] wie folgt beschrieben:

- Standardkonform
- Formal
- Grafisch und Symbolbasierend
- Objektorientiert
- Testbar
- Portierbar und Skalierbar
- Wiederverwendbar
- Effizient

was die in [Soc] beschriebenen Anwendungsfalle (Spezifikation von Echtzeitsystemen, verteilten Systemen und generischen Eventgesteuerten Systemen) untermauert.

> Ein in SDL spezifiziertes System kann man entweder als Text (Textual Phrase Repräsentation/PR) oder in graphischer Form (Graphic Repräsentation/GR) darstellen.

[Wik15h]

7.2. Bekannte Protokolle

Zu den bekanntesten Protokollen im Zusammenhang mit "dem Internet" gehören Transport Control Protocol (TCP), User Datagram Protocol (UDP) und Internet Protocol (IP). Jedes dieser Protokolle beinhaltet Ablaufdefinitionen sowie Datenformatsdefinitionen. TCP zum Beispiel beinhaltet unter anderem den TCP-Handshake (Abbildung 4) und -Teardown sowie eine Definition für den sogenannten TCP-Header (Abbildung 5), welcher angibt in welchem Format die (Meta-)Daten übertragen werden.

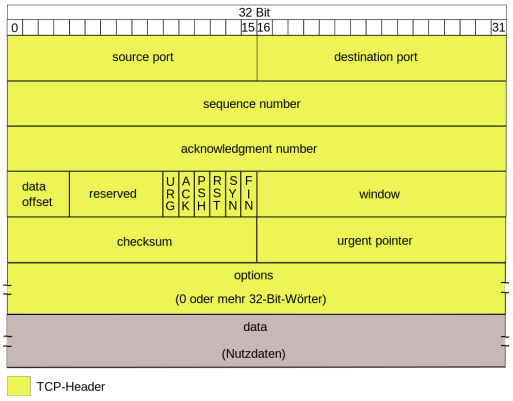

Abbildung 5.: TCP Header, [Wik15i]

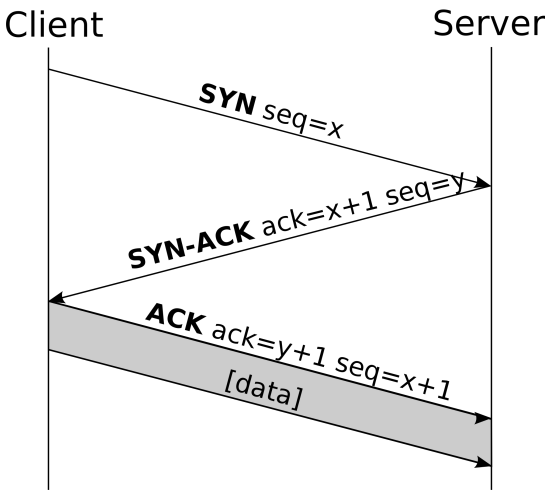

Abbildung 4.: TCP Handshake, [Wik15i]

Auch bekannte Protokolle sind Hyper Text Transfer Protocol (HTTP), Post Office Protocol (POP), Internet Message Access Protocol (IMAP) oder Simple Mail Transfer Protocol (SMTP).

HTTP ist ein zustandsloses ([Wik16g]) Protokoll welches zur Übertragung von Daten aus dem Internet genutzt werden kann [Wik16a]). HTTP wird durch verschiedene andere Protokolle erweitert (zum Beispiel Hyper Text Transfer Protocol Secure (HTTPS)) oder benutzt um darauf aufzubauen (Web-based Distributed Authoring and Versioning (WebDAV)).

POP ist ein Protokoll, über welches Emails von einem Server abgeholt werden können. Die letzte Version dieses Protokolls POP3 wird in Request For Comments (RFC) 1939 beschrieben. POP3 ist ein American Standard Code for Information Interchange (ASCII)-Protokoll, bei welchem Steuerkommandos über einen dedizierten Port an den Server gesendet werden [Wik16e].

IMAP wird wie POP benutzt um Emails von einem Server abzuholen, verfolgt

dabei allerdings eine andere Philosophie und stellt ein Netzwerkdateisystem für Emails bereit, was IMAP flexibler als POP macht. IMAP ist wie auch POP ein textbasiertes Protokoll [Wik16b].

SMTP ist ein Protokoll zum Austausch von Emails in einem Netzwerk. Es wird vorrangig zum Senden und Weiterleiten von Emails benutzt und stellt somit das Gegenstück zu POP oder IMAP dar [Wik16f].

Zudem existiert das unter Linux, BSD, AIX sowie HP-UX verfügbare Network File System (NFS), welches von 1984 an von Sun Microsystems entwickelt wurde. Die Abkürzung NFS beschreibt nicht nur einen Dienst sondern auch ein Kommunikationsprotokoll. Es wird durch die RFC 1094, 1813 und 3010 beschrieben [Mat03, S. 6]. Was herkömmlicherweise als NFS beschrieben wird, setzt sich dabei aus vier verschiedenen Diensten zusammen:

- Network Lock Manager
- Network Status Monitor
- Mount Daemon
- Network File System

wobei jeder dieser Dienste von RPC abhängt [Sel16].

Glossar

Abstraktion

Der Begriff Abstraktion wird in der Informatik sehr häufig eingesetzt und beschreibt die Trennung zwischen Konzept und Umsetzung. Strukturen werden dabei über ihre Bedeutung definiert, während die detaillierten Informationen über die Funktionsweise verborgen bleiben. Abstraktion zielt darauf ab, die Details der Implementierung nicht zu berücksichtigen und daraus ein allgemeines Schema zur Lösung des Problems abzuleiten.

[?] 7, 38

Compiler

Ein Compiler ist ein Computerprogramm, das ein (anderes) Programm, das in einer bestimmten Programmiersprache geschrieben ist, in eine Form übersetzt, die von einem Computer ausgeführt werden kann.

[?] 1, 3–5, 38

Debugger

Ein Debugger (von engl. bug im Sinne von Programmfehler) ist ein Werkzeug zum Diagnostizieren und Auffinden von Fehlern in Computersystemen, dabei vor allem in Programmen, aber auch in der für die Ausführung benötigten Hardware.

[?] 3, 38

Framework

Ein Framework (englisch für Rahmenstruktur) ist ein Programmiergerüst, das in der Softwaretechnik, insbesondere im Rahmen der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen, verwendet wird. Im allgemeineren Sinne und nicht als dezidiertes Softwareprodukt verstanden, bezeichnet man mit Framework auch einen Ordnungsrahmen.

[?] 1, 38

Interpreter

Ein Interpreter ist ein Computerprogramm, das ein (anderes) Programm, welches in Form von Quelltext vorliegt und in einer bestimmten Programmiersprache geschrieben ist, ausführt indem es die Anweißungen der Sprache in Maschinencode übersetzt und ausführt. Der erzeugte Maschinencode wird dabei nicht gespeichert. 3–5, 38

Laufzeit

Der Begriff Laufzeit beschreibt in der Informatik [...] die Zeitdauer, die ein Programm, ausgeführt durch einen Rechner, zur Bewältigung einer Aufgabe benötigt.

[?] 1, 38

Linker

Unter einem Linker oder Binder versteht man ein Computerprogramm, das einzelne Programmmodule zu einem ausführbaren Programm zusammenstellt (verbindet).

[?] 3, 6, 38

Makefile

Ein Makefile ist eine Datei, welche von dem Programm "make" ausgeführt wird.

[?] 6, 7, 10, 38

Maschinencode

> Maschinensprache (auch Maschinencode oder nativer Code genannt) ist eine Programmiersprache, in der die Instruktionen, die vom Prozessor eines Computers direkt ausgeführt werden können, als Sprachelemente festgelegt sind [...].

[?] 1, 38

Objectfile

Eine "Objektdatei" (engl. "Objectfile") wird beim Übersetzen einer Quellcodedatei in ein ausführbares Programm in einem Zwischenschritt des Compilers erzeugt. Siehe: [?], [?]. 6, 38

portierbar

Die portierbarkeit eines Programms ist definiert über die Plattformunabhängigkeit eines Programms:

> Die Plattformunabhängigkeit, genauer als plattformübergreifend wird – in der Informationstechnik – die Eigenschaft genannt, wenn ein Programm auf verschiedenen Plattformen ausgeführt werden kann.

[?] 7, 38

Programmierparadigma

> Ein Programmierparadigma ist ein fundamentaler Programmierstil.„Der Programmierung liegen je nach Design der einzelnen Programmiersprache verschiedene Prinzipien zugrunde. Diese sollen den Entwickler bei der Erstellung von gutem Code unterstützen, in manchen Fällen sogar zu einer bestimmten Herangehensweise bei der Lösung von Problemen zwingen.

[?] 1, 38

Shared Library

Eine "Shared Library" (engl. für "geteilte Bibliothek") ist eine von mehreren Programmen nutzbare Programmbibliothek.

Eine Programmbibliothek bezeichnet in der Programmierung eine Sammlung von Unterprogrammen/-Routinen, die Lösungswege für thematisch zusammengehörende Problemstellungen anbieten. Bibliotheken sind im Unterschied zu Programmen keine eigenständig lauffähigen Einheiten, sondern sie enthalten Hilfsmodule, die von (anderen) Programmen angefordert werden.

[?] 7, 38

Literaturverzeichnis

[Bey13] Beyer, Matthias: *Git - Einführung in die Befehle des verteilten Versionsverwaltungssystems Git*. Hochschule Furtwangen, 04 2013

[Cal10] Calcote, John: *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, 2010

[cma15] *CMake/Testing with CTest*. cmake.org. `https://cmake.org/Wiki/CMake/Testing_With_CTest`. Version: November 2015

[Dan15] Daniels, Philip: *autotools-template*. github. `https://github.com/PhilipDaniels/autotools-template`. Version: October 2015

[Git13] Git Projekt: *Git*. Website. `http://git-scm.com`. Version: Mai 2013. – Projektseite von git

[Goo15a] Google: *Google C++ Testing Framework - Primer*. code.google.com. `https://code.google.com/p/googletest/wiki/Primer`. Version: November 2015

[Goo15b] Google: *Google C++ Testing Framework Samples*. code.google.com. `https://code.google.com/p/googletest/wiki/Samples`. Version: November 2015

[Goo15c] Google: *googletest*. github. `https://github.com/google/googletest`. Version: November 2015

[Ind16] Indeed: *Subversion, Git Job Trends*. `http://www.indeed.com/publicanalytics/jobtrends?q=Subversion%2C+Git&l=`. Version: 01 2016

[Mat03] Mathes, Juliane: *Netzwerkdateisysteme*, Rheinisch-Westfälische Technische Hochschule Aachen, Proseminar, 2003. – Lehrstuhl für Informatik IV

[Mec05] Mecklenburg, Robert: *GNU make*. O'Reilly Germany, 2005

[SBS08] Sneed, Harry M. ; Baumgartner, Manfred ; Seidl, Richard: Der Systemtest. In: *Hanser, München* 2 (2008)

[Sch01] Schmaranz, Klaus: *Softwareentwicklung in C*. 1. Springer, 2001, 2001. – ISBN 978–3540419587

[sei15] *Community Software Architecture Definitions*. website. `https://`

www.sei.cmu.edu/architecture/start/glossary/community.cfm. Version: November 2015

[Sel16] SELFLINUX: *NFS - Network Filesystem*. http://www.selflinux.org/selflinux/html/nfs02.html. Version: 1 2016

[SM94] S. MAUW, M. A. R.: An Algebraic Semantics of Basic Message Sequence Charts. In: *The Computer Journal* 37 (1994), Nr. 4. – Department of Mathematics and Computer Science, Eindhoven University of Technology

[Soc] SOCIETY, SDL F.: Specification and Description Language (SDL).

[tho15] THOUGHTBOT.COM: *factorygirl: GETTING STARTED: Defining factories*. http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md#Defining_factories. Version: 10 2015

[VAC+08] VOGEL, Oliver ; ARNOLD, Ingo ; CHUGHTAI, Arif ; IHLER, Edmund ; KEHRER, Timo ; MEHLIG, Uwe ; ZDUN, Uwe: *Software-Architektur*. Springer Science & Business Media, 2008

[Wal] WALLWITZ, Fabian: Objektorientierte Architekturen, Frameworks und Architekturmuster.

[Wik15a] WIKI, Haskell: *Introduction to QuickCheck1*. https://wiki.haskell.org/Introduction_to_QuickCheck1. Version: 10 2015

[Wik15b] WIKIPEDIA: *Abstract Syntax Notation One*. https://de.wikipedia.org/wiki/Abstract_Syntax_Notation_One. Version: 11 2015

[Wik15c] WIKIPEDIA: *Architekturmuster*. https://de.wikipedia.org/wiki/Architekturmuster. Version: 11 2015

[Wik15d] WIKIPEDIA: *CMake*. https://de.wikipedia.org/wiki/CMake. Version: 10 2015

[Wik15e] WIKIPEDIA: *Integrierte Entwicklungsumgebung*. https://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung. Version: 10 2015

[Wik15f] WIKIPEDIA: *Message Sequence Chart*. https://de.wikipedia.org/wiki/Message_Sequence_Chart. Version: 11 2015

[Wik15g] WIKIPEDIA: *Softwaretest*. https://de.wikipedia.org/wiki/Softwaretest. Version: 10 2015

[Wik15h] WIKIPEDIA: *Specification and Description Language*. https://de.wikipedia.org/wiki/Specification_and_Description_Language. Version: 11 2015

[Wik15i] WIKIPEDIA: *Transmission Control Protocol*. https://de.wikipedia.org/wiki/Transmission_Control_Protocol. Version: 12 2015

[Wik15j] WIKIPEDIA: *Versionsverwaltung*. https://de.wikipedia.org/wiki/Versionsverwaltung. Version: 10 2015

[wik15k] WIKIVS: *Git vs Subversion*. https://www.wikivs.com/wiki/Git_vs_Subversion. Version: 10 2015

[Wik16a] WIKIPEDIA: *Hyptertext Transfer Protocol*. https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol. Version: 1 2016

[Wik16b] WIKIPEDIA: *Internet Message Access Protocol*. https://de.wikipedia.org/wiki/Internet_Message_Access_Protocol. Version: 1 2016

[Wik16c] WIKIPEDIA: *ISO 646*. https://de.wikipedia.org/wiki/ISO_646. Version: 1 2016

[Wik16d] WIKIPEDIA: *MSC Message Sequence Chart Beispiel*. https://commons.wikimedia.org/wiki/File:MSC_Message_Sequence_Chart_Beispiel.png#/media/File:MSC_Message_Sequence_Chart_Beispiel.png. Version: 1 2016. – MSC Message Sequence Chart Beispiel“ von D235 aus der deutschsprachigen Wikipedia. Lizenziert unter CC BY-SA 3.0 über Wikimedia Commons

[Wik16e] WIKIPEDIA: *Post Office Protocol*. https://de.wikipedia.org/wiki/Post_Office_Protocol. Version: 1 2016

[Wik16f] WIKIPEDIA: *Simple Mail Transfer Protocol*. https://de.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol. Version: 1 2016

[Wik16g] WIKIPEDIA: *Zustandslosigkeit*. https://de.wikipedia.org/wiki/Zustandslosigkeit. Version: 1 2016

[Wol15] WOLFINGER, Prof. Dr. Bernd E.: *Dienste*. website. https://www.informatik.uni-hamburg.de/TKRN/world/lernmodule/LMint/Popup/dienste.htm. Version: November 2015

[Yan15] YANG, Ian: *cmake-template*. github. https://github.com/doitian/cmake-template. Version: October 2015

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Furtwangen, 10.01.2016

A. Abbildungen

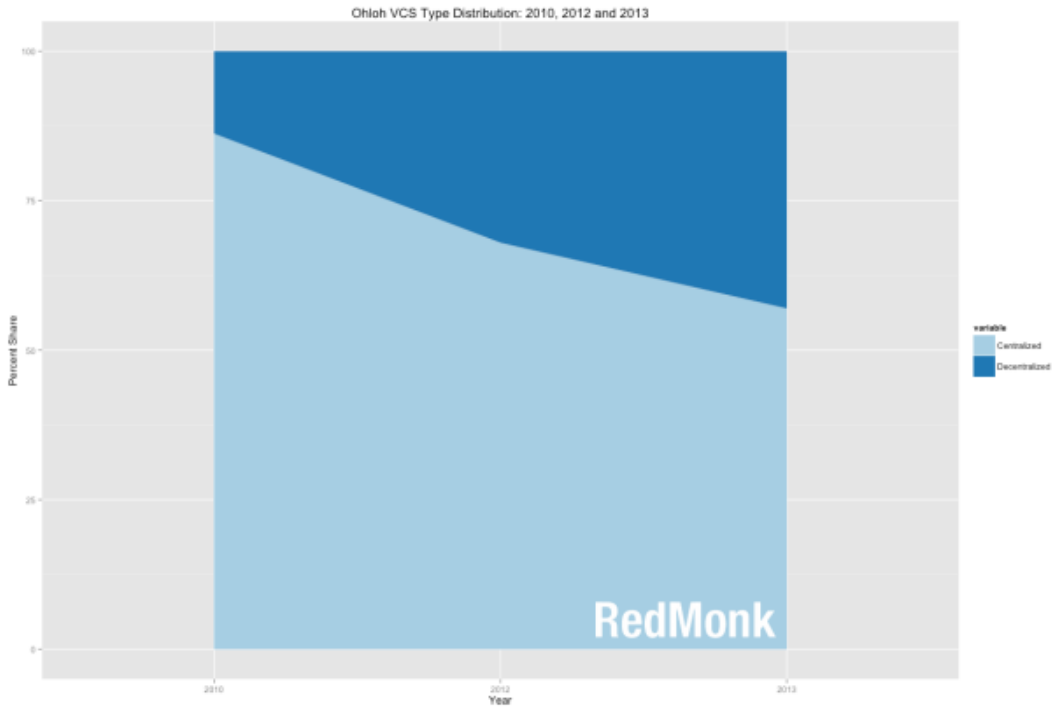

Abbildung 6.: Vergleich Benutzung VCS, [wik15k]

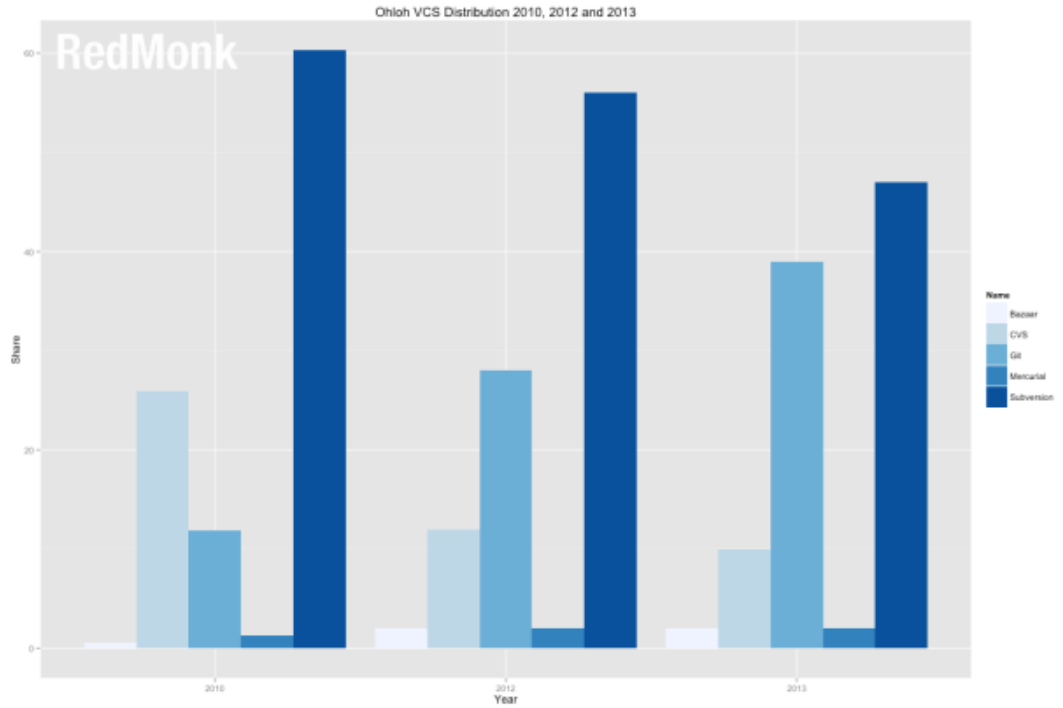

Abbildung 7.: Vergleich Benutzung spezifischer VCS

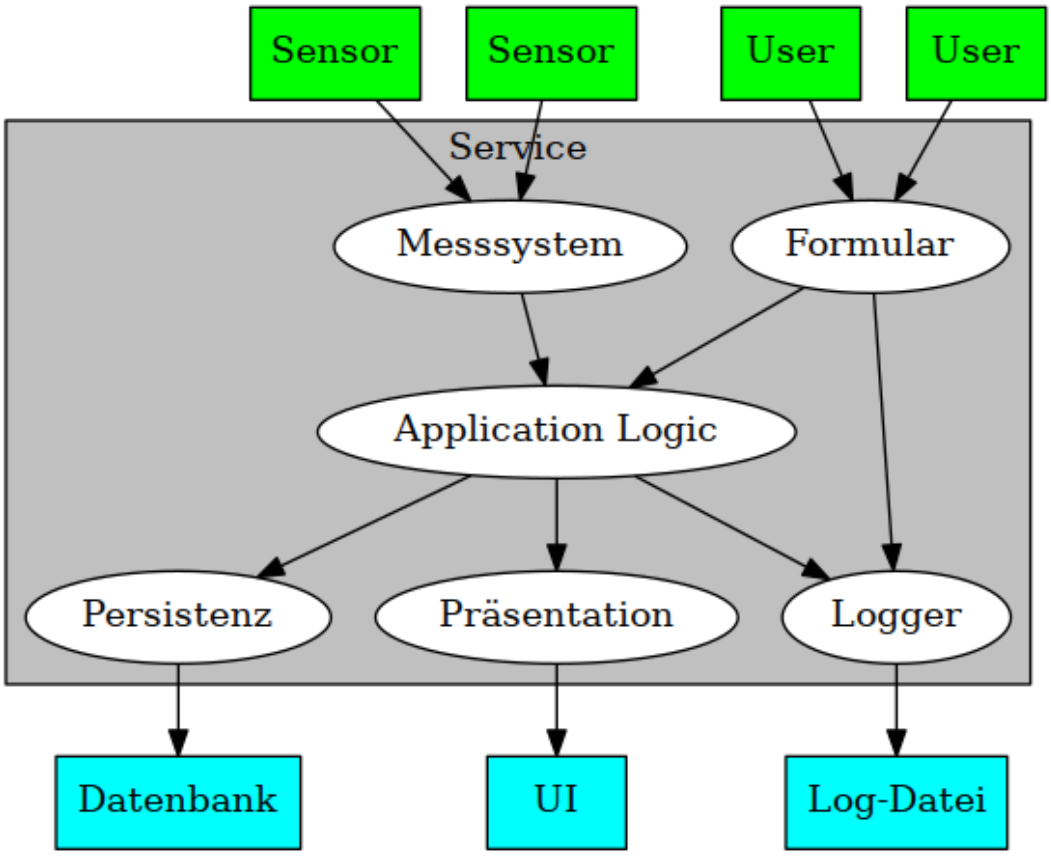

Abbildung 8.: Datenfluss

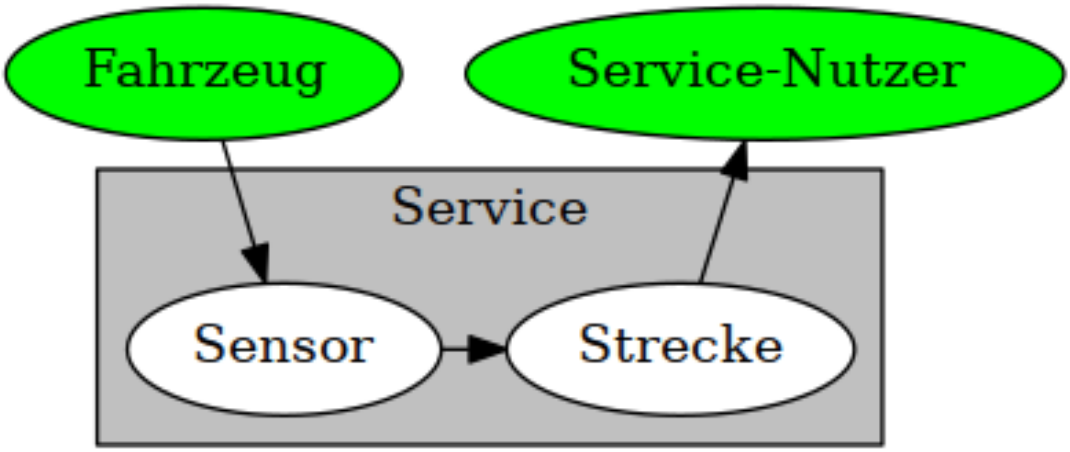

Abbildung 9.: Teilproblem

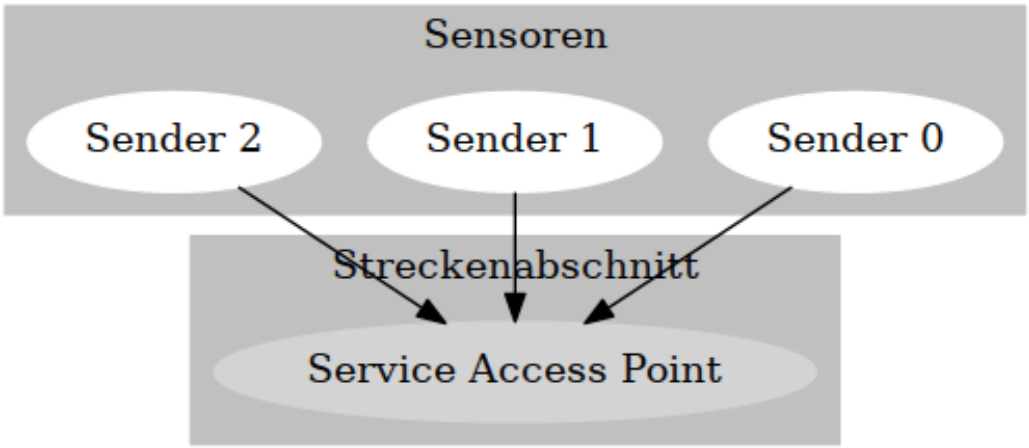

Abbildung 10.: Datenfluss Sensoren → Streckenabschnitt

B. Listings

Listing B.1: Service-Spezifikation

```
specification TrainEventBC {
    service TEBC–DATA_TRANSFER:

                    // Broadcasted data from sensor
        requested FahrzeugData(data_id: uuid,
                               fahrzeug_id: uuid,
                               data: fahrzeugData) ||
                  ErrorIndicator(data_id: uuid, errid: id
                    ),

                  // ACK from receiver
        responded DataACK(data_id: uuid);

                  // data was malformed
                  DataMal(data_id: uuid);

    sap sensor {

        DATA TRANSFER: FahrzeugData;

        par event{
            FahrzeugData:
                respond:
                    DataACK
                    send:
                        strecke.FahrzeugData
                || DataMal;
            ErrorIndicator:
                respond:
                    DataACK
                    send:
                        strecke.ErrorIndicator
                || DataMal;
        }
    }

    sap strecke { // Receiver
        par event{
            sensor.FahzeugData:
                respond:
```

```
40                      DataACK
41                      send:
42                          receiver.FahrzeugData
43                  || DataMal;
44              sensor.ErrorIndicator:
45                  respond:
46                      DataACK
47                      send:
48                          receiver.ErrorIndicator
49                  || DataMal;
50          }
51      }
52
53  }
```

Listing B.2: MSC Beispiel

```
1  msc ExampleMSC;
2      inst Client, Server;
3      condition Idle shared all;
4          instance Client;
5              in ConReq from env;
6              out connect to Server;
7          endinstance;
8          instance Server;
9              in connect from Client;
10             out conInd to env;
11         endinstance;
12     condition Connecting shared all;
13         instance Client;
14             in accepted from Server;
15             out conCnf to env;
16         endinstance;
17         instance Server;
18             in conAcc from env;
19             out accepted to Client;
20         endinstance;
21     condition Connected shared all;
22 endmsc;
```