Technische Universität Braunschweig

Institut für Flugführung

Dokumentation und Testing

Qualitätssicherung in der Projektarbeit

Prof. Dr.-Ing. Peter Hecker, Dipl.-Ing. Paul Frost, 16. Mai 2017

Agenda

04. April	Kick-Off
11. April	Projektmanagement
18. April	Prozessmodelle
25. April	Versionsverwaltung
02. Mai	Einführung Arduino/Funduino
09. Mai	Entwicklungsumgebungen und Debugging
16. Mai	**Dokumentation und Testing**
23. Mai	Dateieingabe und -ausgabe
30. Mai	GUI-Erstellung mit Qt
06. Juni	Exkursionswoche
13. Juni	Bibliotheken
20. Juni	Netzwerke
27. Juni	Projektarbeit
04. Juli	Projektarbeit
11. Juli	Vorbereitung der Abgabe

Teil I

Wiederholung

Praxisdemonstration: Debugging

Teil II

Dokumentation

Hintergrund

Für den Kunden

- Nachweis des Arbeitsfortschritts

Für das Projektmanagement

- Risikobewertung
- Projektstatus

Für andere Entwickler

- Ein Problem, viele Lösungen $\Rightarrow$ Personalabhängigkeit
- Kommunikation zwischen Mitarbeitern ermöglichen/erleichtern
- Definition von Schnittstellen zu anderen Modulen

Für den Entwickler selbst

- Nachvollziehbarkeit auch nach längerer Zeit
- Absicherung in Haftungsfragen (sicherheitskritische Software)

Dokumente in der Softwareentwicklung

Lastenheft

Skripte

Endbenutzer-Dokumentation

Roadmaps

Quelltextdokumentation

DDL-Dokumente

Bug-Reports

Controlling

Pflichtenheft

Lizenz-Dokumente

Quellcode

Product-Backlog

Retrospektiven

ICDs

Gesprächsprotokolle

Testspezifikationen

Maschinencode

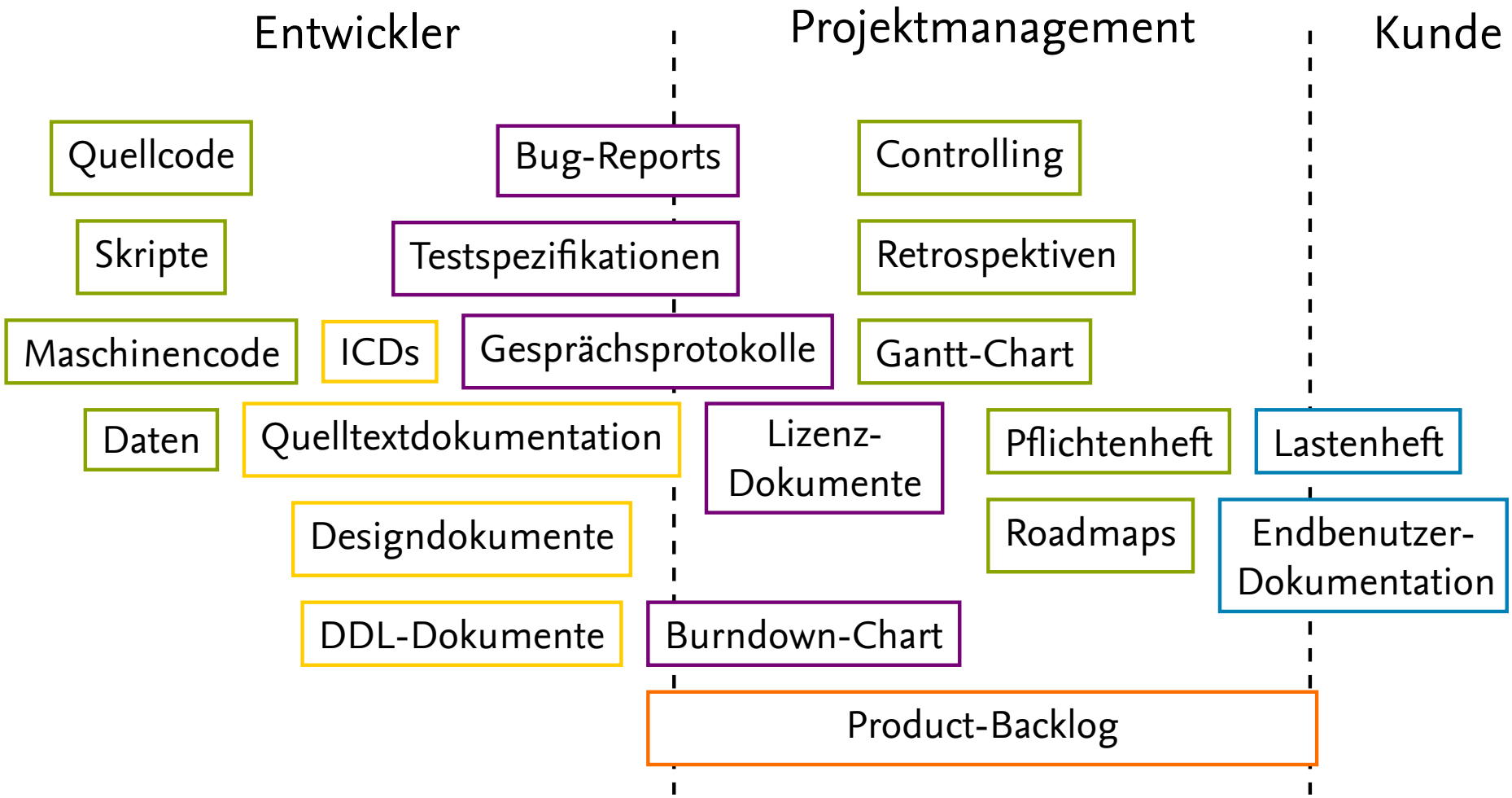
Designdokumente

Gantt-Chart

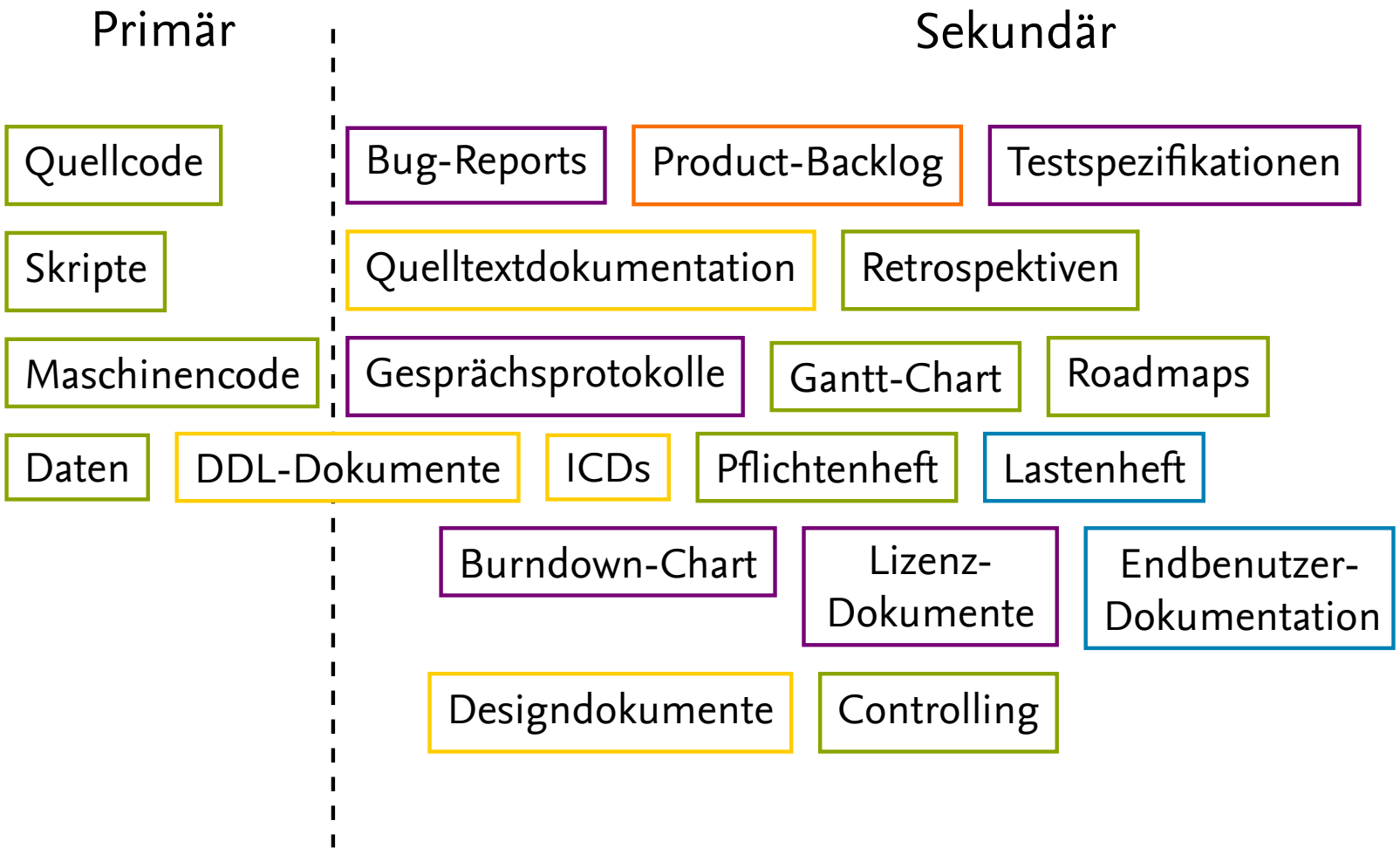
Daten

Burndown-Chart

Klassifikation nach Nutzergruppen



Klassifikation nach Primär- und Sekundärdokumenten



Softwaredesigndokumente

- Möglichkeiten eingrenzen um Austausch zu vereinfachen
  - Weniger Lösungswege
  - Weniger Probleme bei der Übergabe von Code
- Coding Conventions
  - Geben die Schriftform vor
  - Einrückungsstil, Namenskonventionen
  - (Un-) Erwünschte Konstrukte
- Software-Architektur
  - Algorithmen schematisch beschreiben
  - Beschreibung verwendeter Design-Muster (Pattern)
  - Definition von Modulen und deren Abhängigkeiten
  - Repräsentation unabhängig von Programmiersprache, aber mit Bedacht, welche Sprache zur Implementierung verwendet werden soll

Unified Modeling Language (UML)

- Standardisiert nach ISO/IEC 19501
- Systemmodellierung durch Begriffe und deren Beziehung zueinander (Modell), Visualisierung über diverse Diagramme
- Statisch: Klassendiagramm, Kompositionsstrukturdiagramm, Komponentendiagramm, Verteilungsdiagramm, Objektdiagramm, Paketdiagramm
- Dynamisch: Aktivitätsdiagramm, Use-Case-Diagramm, Interaktionsübersichtsdiagramm, Kommunikationsdiagramm, Sequenzdiagramm, Zeitverlaufsdiagramm, Zustandsdiagramm
- Austauschformat: XML Metadata Interchange (XMI)

Hintergrund

Listing 1: code/ArduinoSchlecht/ArduinoSchlecht.ino

```
1 class E{public: void shE(){
2 Serial.print("Eins");Serial.println();}
3 }; class Z{public: void shZ()
4 {Serial.print("Zwei");Serial.println();}
5 }; E e; Z z; void setup() {
6 Serial.begin(9600);} void loop() {
7 e.shE(); z.shZ();delay(1000);}
```

$\Rightarrow$ Keine Nachvollziehbarkeit des Quelltexts für Teammitglieder und den Programmierer

Programmierrichtlinien

Was zeichnet guten Code aus?

- Funktionalität
- Verständlichkeit
- Wartbarkeit
- Effizienz
- Eleganz

Welche Mittel stehen den Entwicklern zur Verfügung?

- Formatierung
- Bezeichner
- Komplexität
- Kommentare

Formatierung Klammern und Separatoren

- Bessere Wartbarkeit
- Geringere Fehleranfälligkeit

Listing 2: ohne Klammern

```
1 while(true)
2 wiederholung();
```

Listing 3: mit Klammern

```
1 while(true){
2 wiederholung();
3 }
```

Formatierung

Einrückungen

- Verdeutlichen die Zugehörigkeit
- Programmablauf wird nachvollziehbarer
- Syntaxfehler werden schneller ersichtlich

Listing 4: mit Einrückung

```
1 while(true){
2   if(quit) {
3     break;
4   }
5   wiederholung();
6 }
```

Formatierung

Einrückungen

Listing 5: ohne Einrückung

```
1 if((bedingungA && bedingungB)
2 || bedingugnC
3 || bedingungD){
4   machEtwas();
5 }
```

Listing 6: mit Einrückung

```
1 if((bedingungA && bedingungB)
2     || bedingungC
3     || bedingungD){
4   machEtwas();
5 }
```

Bezeichner

Schlechtes Beispiel

Listing 7: schlechte Bezeichner

```
1  const int varA = 42600;
2  const float varB = 35.8f;
3  const float varC = 122.4;
4
5  char* dasisteinvielzulangervariablenname = "LH321";
6  char* Variablemitgrossbuchstabenamanfang = "EDVE";
7
8  double _ = 52.15648;
9  double __ = 9.5614;
10
11 float test = 54600.64f;
```

Bezeichner Beispiel

Listing 8: sinnvolle Bezeichner

```
1  const int A320_EMPTY_WEIGHT_KG = 42600;
2  const float A320_WING_SPAN_M = 35.8f;
3  const float A320_WING_AREA_M2 = 122.4;
4
5  char* flightNr = "LH321";
6  char* departureAirport = "EDVE";
7
8  double latitude_deg = 52.15648;
9  double longitude_deg = 9.5614;
10
11 float currentMass_kg = 54600.64f;
```

Bezeichner Namenskonvention

Variablen	Kleinbuchstabe am Anfang Trennung durch Großbuchstaben	`statusLed, date`
Konstanten	Großbuchstaben, Trennung durch Unterstriche	`MAX_WIDTH,` `MIN_WIDTH`
Methoden	Kleinbuchstabe am Anfang, Verb, Imperativ	`calculateDistance()`
	Getter-Methoden	`speed(), getSpeed()`
	Setter-Methoden	`setSpeed()`
Klassen	Großbuchstabe am Anfang	`Klasse`

Bezeichner Vorschlag

Listing 9: code/variablen.h

```
1  class EineKlasse{
2    static int s_statischeVariable;
3
4  public:
5    void setMemberVariable(int memberVariable);
6    int memberVariable() const;
7
8  private:
9    int m_memberVariable;
10 };
```

Komplexität Funktionen

- Beschreibende Funktionsnamen
- Funktionen klein halten
  - Richtwert: 7 Zeilen
  - In weitere Funktionen aufteilen

⇒ Quellcode kann leichter wiederverwendet werden

Listing 10: code/function.cpp

```
1 double calcCoordinateX(double latitude, double longitude){...}
2 double calcCoordinateY(double latitude, double longitude){...}
3
4 void calcCoordinates(double latitude, double longitude)
5 {
6   double x = calcCoordinateX(double latitude, double longitude);
7   double y = calcCoordinateY(double latitude, double longitude);
8   printPoint(x, y);
9 }
```

Kommentare Quellcode

- Verständlichkeit von Quellcode durch Anmerkungen erhöhen, die die Programmerstellung nicht beeinflussen
- Programmiersprachen erlauben in der Regel das Schreiben von Kommentaren:
  - C/C++/Java

```
// Diese Zeile ist ein Kommentar
/* Dieser Block ist ein Kommentar */
```

  - XML/HTML

```
<!- Dieser Block ist ein Kommentar ->
```

  - MATLAB/LaTeX

```
% Diese Zeile ist ein Kommentar
```

Beispiel für sinnvolle Kommentierung

Sinnvoller Kommentar (mehr oder weniger)

Listing 11: code/kommentarbeispiele.c

```
1 // berechnet den Umfang des Kreises mit dem Radius r
2 float u(float r)
3 {
4   return 2 * M_PI * r;
5 }
```

Microsoft ESP SDK Variablenbeschreibung

Variable `PLANE PITCH DEGREES`:

Pitch angle, although the name mentions degrees the units used are radians

Beschreibung von Programmierschnittstellen

- Extern verfügbare Funktionen und Objekte sollten dokumentiert werden
- Nutzer muss oft nur wissen was gemacht wird, aber nicht wie
- Schnittstellen-Dokumentation in Header-Dateien (`.h`, `.hh`, `.hpp`)

Listing 12: Deklaration

```
1 // Fuehrt eine Division aus.
2 // Bitte Divisor niemals 0 setzen!
3 float quotient(float dividend, float divisor);
```

Listing 13: Implementierung

```
1 float quotient(float dividend, float divisor)
2 {
3   return divident / divisor;
4 }
```

API-Dokumentation

- Programmierer benötigt Informationen zur korrekten Verwendung einer Bibliothek
  - Was bewirkt welche Methode/Funktion?
  - Welche Parameter erwartet die Methode/Funktion?
  - Was gibt die Methode/Funktion zurück?
- Die Information muss erstellt werden
- Quellcodekommentare können während des Entwickelns erstellt werden
  - Nicht jeder möchte Quellcodedateien nach Kommentaren durchsuchen
  - Oft keine gute Gliederung, kein Index und keine Möglichkeit der Verlinkung durch einfache Quellcodekommentare gegeben
- Ansatz: Dokumentation aus Quellcode-Kommentaren erzeugen

Doxygen

- Programm zur automatisierten Beschreibung von APIs
  - GPL lizenziert, Entwicklung seit 1997
  - Unterstützung mehrerer Programmiersprachen (C, C++, C#, Java)
  - In vielen Entwicklungsumgebungen integriert
  - Lauffähig unter vielen Betriebssystemen (Linux, Windows, MacOS)
- Programmablauf von Doxygen
  - Automatische Ermittlung der Quelltext-Struktur
  - Einlesen und Verarbeitung von speziellen Doxygen-Kommentaren
  - Ausgabe der API-Beschreibung in verschiedenen Formaten (HTML, CHM, PDF, etc.)

Struktur- und Abhängigkeitsgraphen mit Doxygen

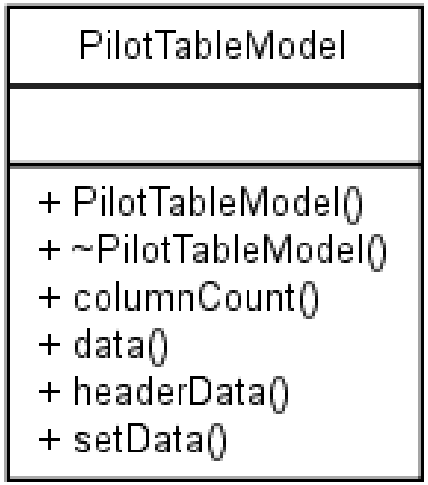


Abbildung 1: Klassendiagramm

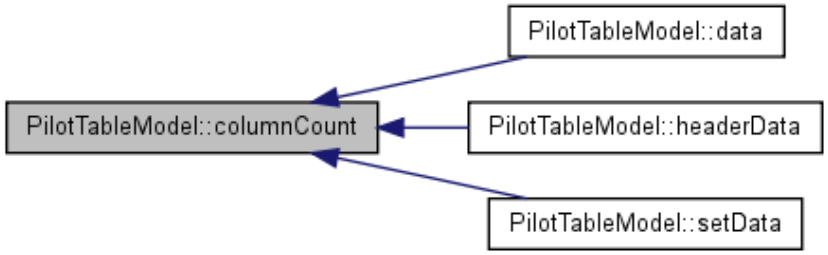


Abbildung 2: Callergraph

Doxygen-Kommentare

- Beschreibungen werden direkt in dem Quelltext der jeweiligen Klasse/Methode geschrieben
- Doxygen-Anweisungen innerhalb von BlockKommentaren

\brief	Kurzbeschreibung der jeweiligen Funktion
\param name	Beschreibung des Parameters name
\return	Beschreibung des Rückgabewertes
\author	Liste der Autoren
\warning	wichtige Hinweise zur Verwendung
\bug	Beschreibung eines bekannten Fehlers
\version	Aktuelle Version
\sa	Siehe auch...

Doxyfile

Das Doxyfile enthält Einstellungen, die für eine Erstellung der Dokumentation erforderlich sind.

Projektname	`PROJECT_NAME`	Name des Projekts
Quelltextverzeichnis	`INPUT`	Ordner zum Quelltext
Ausgabeverzeichnis	`OUTPUT_DIRECTORY`	Ordner, in dem die Dokumentation abgespeichert wird

Doxywizard

- Graphische Oberfläche von Doxygen
- Festlegung von Optionen
  - Ausgabeformat
  - Projekteigenschaften
  - Präferenzen
- Teilweise zugängliche Schritt-für-Schritt-Konfiguration
- Ausgabe von Fehlern und Warnungen
- Abspeichern einer Konfiguration

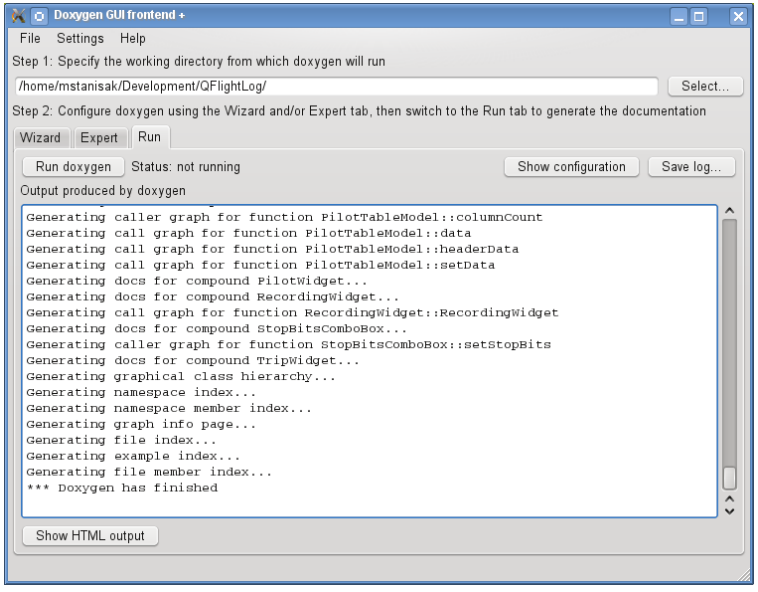


Abbildung 3: Doxywizard

Doxygen-Demonstration

Doxygen-Demonstration

Teil III

Testing

Grundsätze

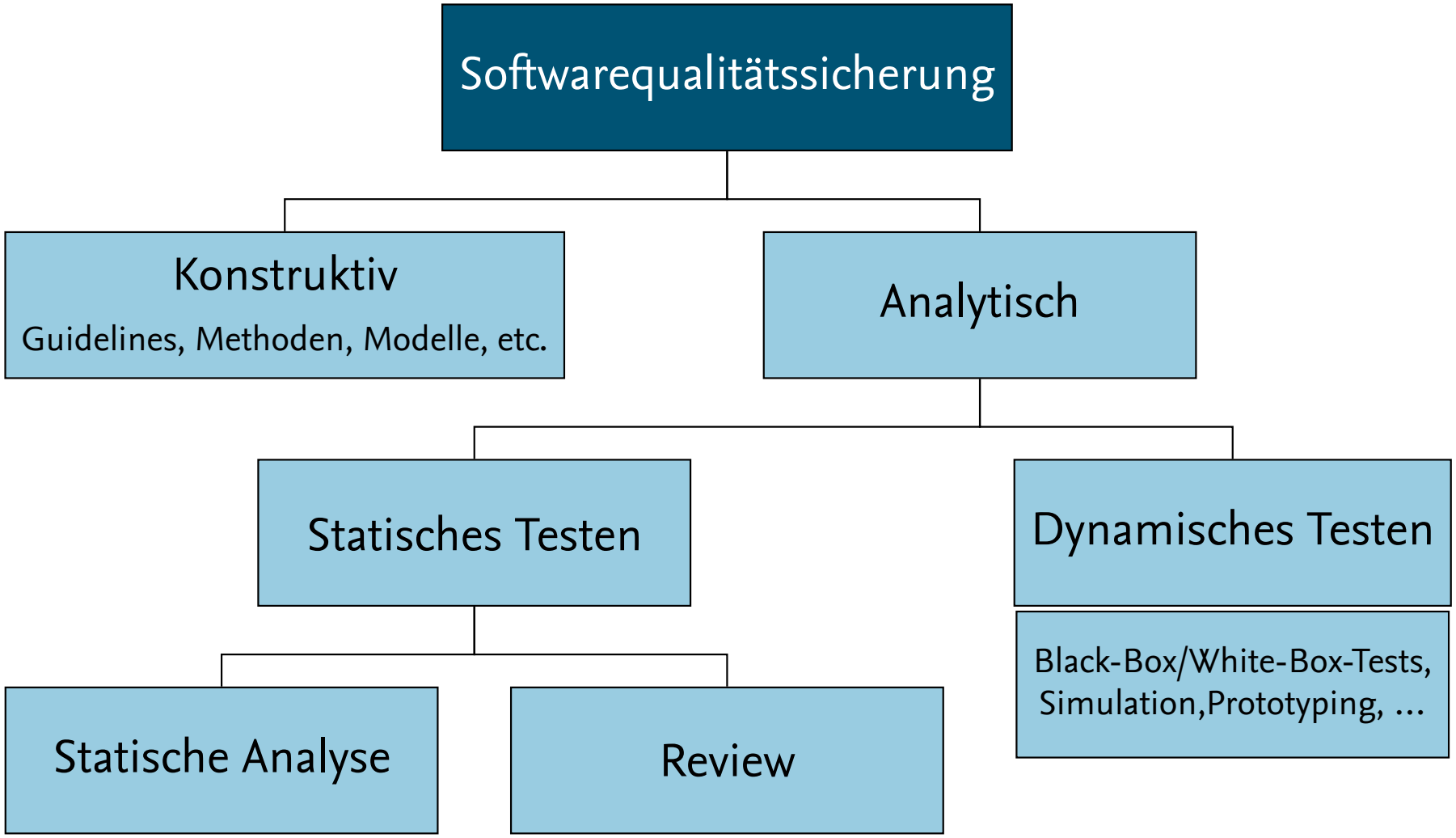
„Program testing can be used to show the presence of bugs, but never to show their absence!“

Edsger W. Dijkstra, 1970

Softwaretests

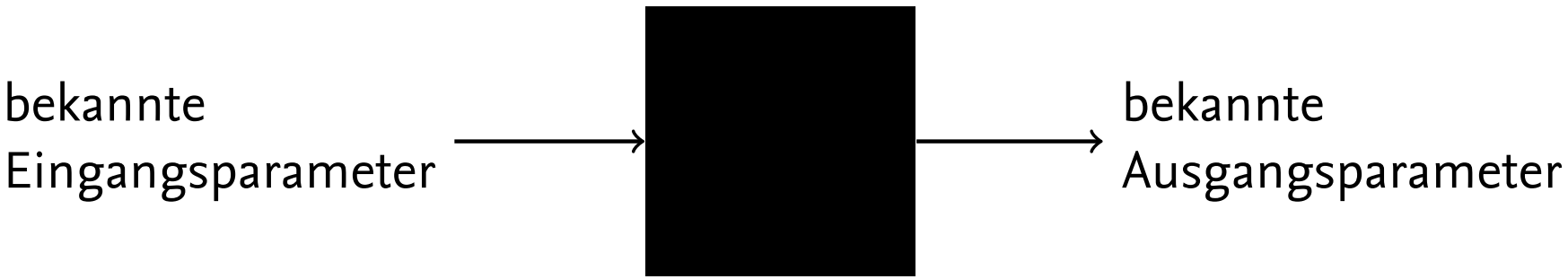
- Unterstützung des Debugging
- Dokumentieren die Erfüllung von Spezifikationen
- Ermöglichen im Entwicklungsprozess eine gezielte Suche nach Fehlern
- Sollten eindeutige Ergebnisse haben (Erfolgreich/Fehler)
- Richtige Reihenfolge in der Implementierung wichtig
  - Softwaredesign
  - Modultests
  - Integrationstests
  - Softwareimplementierung
- Begleiten den Implementierungsprozess kontinuierlich

Qualitätssicherung



Modultests

- Testen abgeschlossener Einheiten
  - Einzelne Klassen
  - Einzelne Funktionen
- Überprüfung ob Eingabewerte zu definierten Ausgaben führen
- Black-Box-Testing (Testen der Außenwirkung, keine Kenntnis der Umsetzung nötig)



Black-Box-Test

Beispiele

String-Funktion `capitalize()`

Alle Buchstaben eines Strings sollen in Großbuchstaben umgewandelt werden. Andere Zeichen bleiben unverändert.

Eingabe	Ausgabe
`abcdef`	`ABCDEF`
`aBcDeF`	`ABCDEF`
`!AbCd!`	`!ABCD!`

String-Funktion `replace()`

Ein einem String (E1) sollen Zeichen (E2) durch andere Zeichen (E3) ersetzbar sein.

E1	E2	E3	Ausgabe
`abcdef`	a	z	`zbcdef`
`aBcDeF`	a	z	`zBcDeF`
`!AbCd!`	a	z	`!AbCd!`

Integrationstests

- Testen von Schnittstellen
  - Datei lesen/schreiben
  - Netzwerkschnittstellen
  - Graphische Oberflächen
- Überprüfen ob Schnittstellen korrekt implementiert sind und Kommunikation wie erwartet stattfindet und abläuft

Manuelles Testen

- Statisches Testen möglich (Programm muss nicht lauffähig sein)
- Manuelle Interaktion mit dem Programm
- Ergebnis des Tests sollte ein Protokoll sein
  Review-Protokoll, Bug-Report, Feature-Report

Vorteile

- Überprüfung des Quelltexts möglich
- Einhaltung von Richtlinien kann überprüft werden

Nachteile

- Hoher personeller/manueller Aufwand
- Manuelle Interaktion birgt Probleme (Unregelmäßigkeiten)
- Ergebnis evtl. subjektiv beeinflusst

Review

Bei einem Review soll der Quellcode manuell begutachtet werden.
Für API soll ein Review die folgenden Informationen enthalten:

- Name des Reviewers
- Name des Entwicklers
- Dateien
- git-Revision (`git log`)
- Untersuchte Funktionalität
- Reviewergebnisse
  Hierfür ist im Anleitungen Ordner ein Leitfaden verfügbar

Automatisiertes Testen

- Definierte Aktion im Rahmen eines Programmablaufs
- Automatisierter Ablauf von Programmschritten
- Ergebnis wird automatisiert ermittelt

Vorteile

- Schnelle automatisierte Durchführung möglich
- Ergebnisse sind objektiv und eindeutig

Schwierigkeiten

- Tests müssen sinnvoll und nützlich definiert werden
- Tests können trügerische Sicherheit vermitteln (z. B. bei Lücken)

Tests selber schreiben

Listing 14: ../../API–Testing/Arduino/testDefinitionen.h

```
1  void testAnmeldung(){ // Name frei waehlbar
2    // Die folgende Zeile sorgt dafuer, dass die Testfunktion
3    // identifizierbar ist
4    bool testResult = false;
5
6    // Eigene Testimplementierung...
7
8    // testResult kann bei einem erfolgreichen Test
9    // auf true gesetzt werden
10
11   APITest::printTestResult(testResult, "Anmeldung", "Paul Frost",
12   "Anmeldung der Studierenden Prototyp A02", "testDefinitionen.h");
13 }
```

Tests einbinden

Hinzufügen der zuvor implementierten Funktion

Listing 15: ../../API–Testing/Arduino/testDefinitionen.h

```
1 void runTests(){
2   APITest::printTestStartHeader(); // Nicht modifizieren
3
4   // Hier sollen die eigenen Tests hinzugefuegt werden
5   testAnmeldung();
6
7   APITest::printTestEndFooter(); // Nicht modifizieren
8 }
```

Tests starten

1. Tests aktivieren

   `#define TEST`
2. `testDefinitionen.h` einbinden

   `#include "testDefinitionen.h"`
3. Im Ablauf des Programms muss der Befehl `RUNTEST` eingebunden werden.

Listing 16: ../../API–Testing/Arduino/RFID–A02–A03.ino

```
1 #define TEST
2 #include "testDefinitionen.h"
3
4 /// Beginn des Setups:
5 void setup()
6 {
7   RUNTEST
```

Teil IV

Projektarbeit

1. Zyklus vom API-Spiralmodell beenden

Aufgabe 1

Führen Sie ein Code-Review durch und erstellen Sie ein Protokoll auf der Wiki-Seite **Reviews**. Ein Protokoll soll die folgenden Daten beinhalten:

- Name des Reviewers
- Name des Entwicklers
- Dateien
- git-Revision (`git log`)
- Untersuchte Funktionalität
- Reviewergebnisse

Aufgabe 2

Jedes Gruppenmitglied soll für eine seiner Anforderungen eine Testfunktion implementieren. Die Testfunktion soll dabei die folgenden Daten bei einem durchgeführten Test ausgeben:

- Name des Tests
- Autor des Tests
- Was wird getestet?
- Dateiname
- Ergebnis des Tests (erfolgreich/fehlgeschlagen)

Vorlagen finden Sie unter: `https://github.com/TUBSAPISS2017/public/tree/TestingVorbereitung/Quellcode`

Aufgabe 3

Erstellen Sie die Wiki-Seite **Testing**. Auf dieser Seite soll je eine Ausgabe der Tests aufgelistet werden.

Aufgabe 4

Erstellen Sie die Wiki-Seite **BugsFeatures**. Erstellen Sie für aufgefallene Fehler einen Bug-Report.
Ein Bug-Report soll die folgenden Punkte enthalten:

- Autor des Bug-Reports
- Welches Modul ist betroffen
- Beschreibung des Fehlers
- Wie ist der Fehler zu reproduzieren

Planen Sie anschließend den folgenden Zyklus.

Fragen?

Gibt es noch Fragen?