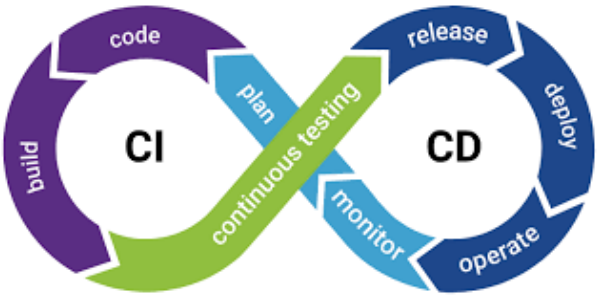
# Projektbericht

Unittests mit GoogleTest und Automatisierung mit Gitlab CI/CD




Betreuer: Lehrstuhl:

Name: Kurs:

28. Juli 2024

# Inhaltsverzeichnis

**Zusammenfassung**

Dieses Dokument erfüllt nicht die Bedingungen eines Standards.
Es kann Fehler enthalten.
Alle Angaben ohne Gewähr.
Fehler bitte melden.

# 1 Einleitung

Dieser Bericht beschreibt die Nutzung von GoogleTest als Testsuite für Unittests in C++. Forthin werden die Möglichkeit von CI/CD in Gitlab beschrieben.

# 2 Theoretischer Hintergrund

## 2.1 Tests

**Testen** von Software überprüft, ob die **Software den Erwartungen entspricht**.

Oft werden Tests wie in test.cpp geschrieben.
Dieses manuelle Vorgehen ist jedoch stark fehlerbehaftet und nicht automatisierbar. Das NIST( National Institute of Standards and Technology) kam in einer Untersuchung( The Economic Impacts of Inadequate Infrastructure for Software Testing) zu dem Schluss, dass **fehlende Tests** die amerikanische Wirtschaft **59,6 Milliarden Dollar im Jahr 2002 kosteten kosten**.

Es ist daher essentiell Tests einzusetzen.
Aus diesem Grund wurden verschieden Testarten geschaffen:

| **Testmethode** | **Beschreibung** |
|---|---|
| Unit-Test (Modultest) | Testet einzelne Komponenten oder Module des Codes isoliert. Ziel ist es, sicherzustellen, dass jede Komponente wie erwartet funktioniert. |
| Integrationstest | Testet das Zusammenspiel verschiedener Module oder Komponenten, um sicherzustellen, dass sie zusammen korrekt arbeiten. |
| Systemtest | Testet das gesamte System als Ganzes, um sicherzustellen, dass alle Anforderungen erfüllt werden und das System wie erwartet funktioniert. |
| Abnahmetest (User Acceptance Testing, UAT) | Testet das System aus der Perspektive des Endbenutzers, um sicherzustellen, dass es den Geschäftsanforderungen entspricht und einsatzbereit ist. |
| Regressions-Test | Testet das System nach Änderungen oder Updates, um sicherzustellen, dass keine neuen Fehler eingeführt wurden und bestehende Funktionen weiterhin korrekt arbeiten. |

| **Testmethode** | **Beschreibung** |
|---|---|
| Stresstest | Testet das System unter extremen Bedingungen, um seine Belastbarkeit und Leistung unter hoher Last zu überprüfen. |
| Sicherheitstest | Testet das System auf Sicherheitslücken, um sicherzustellen, dass es gegen Angriffe und unbefugten Zugriff geschützt ist. |
| Benutzerfreundlichkeits-test (Usability-Test) | Testet die Benutzerfreundlichkeit des Systems, um sicherzustellen, dass es für Endbenutzer leicht zu bedienen und verständlich ist. |
| Kompatibilitätstest | Testet das System auf verschiedenen Plattformen, Betriebssystemen und Geräten, um sicherzustellen, dass es überall korrekt funktioniert. |
| Exploratives Testen | Tester erkunden das System ohne vordefinierte Testszenarien, um unerwartete Fehler und Schwachstellen zu finden. |

Man erkennt, dass viele verschiedene Arten gibt. Ich wende mich der niederschwelligsten Methode, den Unit-Tests, zu.

Unit-Testing ist eine Methode, bei der einzelne Einheiten des Quellcodes getestet werden, um festzustellen, ob sie korrekt funktionieren. Eine Einheit ist der kleinste testbare Teil einer Anwendung. In der prozeduralen Programmierung kann eine Einheit eine einzelne Funktion oder Prozedur sein. Unit-Tests werden normalerweise von Entwicklern erstellt. Das Ziel von Unit-Tests ist es, jeden Teil des Programms zu isolieren und zu zeigen, dass die einzelnen Teile korrekt funktionieren. Ein Unit-Test ist ein strikter, schriftlich festgelegter Vertrag, den ein Teil des Codes erfüllen muss.
Die Verwendung von Unit-Tests hat mehrere Vorteile:

- Erleichterung von Änderungen - Unit-Tests ermöglichen es Programmierern, den Code zu einem späteren Zeitpunkt zu überarbeiten und dabei sicher zu sein, dass der Code immer noch korrekt funktioniert;
- Vereinfachung der Integration - Unit-Tests können die Unsicherheit in den Einheiten selbst verringern und können in einem Bottom-up-Testing-Ansatz verwendet werden. Indem man zuerst die Teile eines Programms testet und dann die Summe der Teile, wird das Integrationstesten viel einfacher;
- Unit-Tests bieten eine Art lebendige Dokumentation für das System. Entwickler können sich den Code der Unit-Tests ansehen, um ein grundlegendes Verständnis der implementierten API zu erlangen.

## 2.2 CI/CD

Continuous Integration (CI) ist die Praxis, Codeänderungen häufig und automatisch in ein zentrales Repository zu integrieren und dabei durch automatisierte Builds und Tests sicherzustellen, dass der Code stets fehlerfrei ist.

Continuous Deployment (CD) ist die Erweiterung von Continuous Integration, bei der

jede erfolgreich getestete Änderung automatisch in die Produktionsumgebung bereitgestellt wird, ohne dass manuelle Eingriffe notwendig sind.

# 3 Methoden

In diesem Abschnitt werden die im Projekt verwendeten Methoden beschrieben. Dies umfasst die Datenerhebung, die verwendeten Werkzeuge und Techniken sowie die Analyseverfahren.

## 3.1 Unittests mit GoogleTest

GoogleTest ist ein von Google entwickeltes Framework. Es bietet viele Funktionen u.a.:

- Testfälle: Definiert einzelne Tests.
- Test-Fixtures: Ermöglicht Setup und Teardown für mehrere Tests.
- Assertions: Vergleichen und überprüfen Werte.
- Basis-Assertions: Vergleich von Werten.
- String-Assertions: Vergleich von C-Strings.
- Floating-Point-Assertions: Vergleich von Gleitkommazahlen.
- Parametrisierte Tests: Führen denselben Test mit verschiedenen Parametern aus.
- Typed Tests: Verwenden die gleiche Test-Logik für verschiedene Datentypen.
- Type-Parameterized Tests: Führen denselben Testfall für verschiedene Typen aus.
- Death Tests: Überprüfen, ob Code terminiert (z.B. bei Abstürzen).
- Mock-Methoden: Simulieren Methoden von Objekten.
- Erwartungen setzen: Definieren, wie Mock-Methoden aufgerufen werden sollen.
- Aktionen und Rückgaben: Bestimmen das Verhalten von Mock-Methoden.
- Google Mock Matchers: Erlauben flexible und ausdrucksstarke Erwartungen.

Beispiele siehe auch Gitlab; Allg.:

```
#include <gtest/gtest.h>

TEST(FactorialTest, FactorialOfZeroShouldBeOne)
{
    ASSERT_EQ(1, factorial(0));
}
```

oder

```
#include <gtest/gtest.h>

TEST(HelloTest, BasicAssertions) {
  EXPECT_STRNE("hello", "world");
  EXPECT_EQ(7 * 6, 42);
}
```

## 3.2 Testen mit CMake

Erstellte GoogleTest Programme müssen kompiliert und ausgeführt werden. Hier hilft Cmake. Es erstellt Code für eine Generator, meist make, und stellt ctest, einen Testtreiber

zur Verfügung.

Basistruktur ist immer:

```
cmake_minimum_required(VERSION 3.14)
project(my_project)

# GoogleTest benoetigt C++14.
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Die benoetigten Dateien werden hier automatisch von Google bezogen
# koennen aber auch installiert werden.
include(FetchContent)
FetchContent_Declare(
  googletest
  URL "aktueller GoogleTest build"
# Windows benoetigt:
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)

FetchContent_MakeAvailable(googletest)

# Nachfolgend wird erst das Testprogramm konfiguriert.
enable_testing()

add_executable(
  hello_test
  hello_test.cc
)
target_link_libraries(
  hello_test
  GTest::gtest_main
)

include(GoogleTest)
gtest_discover_tests(hello_test)
```

Dann im Buildverzeichnis

```
cmake ..
make
ctest
```

ausführen.

## 3.3 CI/CD mit Gitlab

Gitlab CI/CD bietet viele Möglichkeiten ohne abhängig von fremden Servern zu sein, da man selbst hosten kann. Dies tut auch die Uni-Bayreuth. Features sind u.a.:

- Native Integration mit GitLab: Nahtlose Integration in GitLab-Umgebungen.

- YAML-basierte Pipeline Definition: Definition von Pipelines durch eine einfache YAML-Datei (gitlab-ci.yml).
- Automatische Builds: Automatische Erstellung (Build) von Projekten bei Änderungen im Repository.
- Mehrstufige Pipelines: Unterstützung für komplexe Workflows mit mehreren Stufen (Stages).
- Parallele Ausführung: Möglichkeit zur parallelen Ausführung von Jobs zur Optimierung der Durchlaufzeit.
- Job-Artefakte: Bereitstellung von Artefakten nach der Ausführung eines Jobs (z.B. Binärdateien).
- Manuelle Aktionen: Möglichkeit, bestimmte Schritte innerhalb der Pipeline manuell auszulösen.
- Geplante Pipelines: Zeitgesteuerte Ausführung von Pipelines.
- Unterstützung für Umgebungen: Verwaltung von Deployments in verschiedenen Umgebungen (Development, Staging, Production).
- Sicherheitsintegration: Integration mit GitLab Security Scanning Tools für automatische Sicherheitsprüfungen.

Im Normalfall sollten an den Gitlab Einstellungen keinerlei Änderungen nötig sein. Nahezu die gesamte Konfiguration von CI/CD kann über das `.gitlab-ci.yml` File erledigt werden. Beispiel YAML:

```
# stages definiert die Abfolge der Jobs.
stages:
  - build
  - test
  - release

# Es koennen Variablen angelegt werden.
variables:
  BUILD_DIR: build

# Der Job allg wird aufgrund des .pre vor den stages durchgefuehrt.
allg:
  stage: .pre
  script:
    - mkdir -p $BUILD_DIR
    - cd $BUILD_DIR
    - cmake ..
  # Jeder Job wird in einer eigenen virtuellen Dockerumgebung ausgefuehrt.
  # Um Ergebnisse mitzunehmen muessen diese als artifacts markiert werden.
  artifacts:
    paths:
      - $BUILD_DIR

build:
  stage: build
  script:
    - ls
```

```
    - cd $BUILD_DIR
    - make
  artifacts:
    paths:
      - $BUILD_DIR

test:
  stage: test
  script:
    - cd $BUILD_DIR
    # Hier wird getestet
    - ctest --output-on-failure
  dependencies:
    - build

release:
  stage: release
  script:
    - mkdir -p release
    - cp $BUILD_DIR/hello_test release/
    - tar -czvf release.tar.gz release/
  dependencies:
    - build
  artifacts:
    paths:
    # Das Ergebnis kann dann ueber artifacts heruntergeladen werden.
    # Alternativ kann das File auch direkt gehostet werden.
      - release.tar.gz
```

In der Web Oberfläche können alle Bereiche eingesehen werden.

# 4 Ergebnisse

Tests können einfach, zuverlässig und automatisiert Fehler finden. Die oben beschrieben Methoden sind empfehlenswert. Tests funktionieren insbesondere gut mit anderen Methoden wie z.B. Formattern, Lintern, Peerreview, Guidelines und formalen Methoden. Sie stellen somit einen Baustein zu gutem sicherem, zuverlässigen und wartbaren Code dar.

# 5 Fazit

Test sind einfach zu erstellen und sinnvoll.

Weiterhin wäre eine Untersuchung weiterer Möglichkeiten in allen oben genannten Bereichen sinnvoll. Auch die Möglichkeiten von statischen Analysetools mit den obigen Methoden erscheint sinnvoll.
Es hat mir Spaß gemacht daran zu arbeiten. Ich danke Herrn