ROS Unit Tests Workshop

Lukas Hoyer

Otto-von-Guericke-Universität Magdeburg

12.06.2017

Motivation: Ariane 5

- Übernahme von Hard- und Software für das Trägheitsnavigationssystem (SRI) aus der Ariane 4
  - SRI keine Überlaufkontrolle für bestimmte Variablen aufgrund der maximalen Beschleunigung der Ariane 4
  - keine Aufnahme von Flugbahndaten der Ariane 5 in die Spezifikation des SRI

Motivation: Ariane 5

- Übernahme von Hard- und Software für das Trägheitsnavigationssystem (SRI) aus der Ariane 4
  - SRI keine Überlaufkontrolle für bestimmte Variablen aufgrund der maximalen Beschleunigung der Ariane 4
  - keine Aufnahme von Flugbahndaten der Ariane 5 in die Spezifikation des SRI
- Ariane 5 höhere Beschleunigungen als Ariane 4 -> Überlauf
  - Ausfall von zwei redundanten Modulen aufgrund gleicher Software
  - Fehlercode wurde vom Onboard-Computer fälschlicherweise als Flugbahndaten interpretiert
  - Falsche Korrektur der Flugbahn
  - Selbstzerstörung

Motivation: Ariane 5

- Übernahme von Hard- und Software für das Trägheitsnavigationssystem (SRI) aus der Ariane 4
  - SRI keine Überlaufkontrolle für bestimmte Variablen aufgrund der maximalen Beschleunigung der Ariane 4
  - keine Aufnahme von Flugbahndaten der Ariane 5 in die Spezifikation des SRI
- Ariane 5 höhere Beschleunigungen als Ariane 4 -> Überlauf
  - Ausfall von zwei redundanten Modulen aufgrund gleicher Software
  - Fehlercode wurde vom Onboard-Computer fälschlicherweise als Flugbahndaten interpretiert
  - Falsche Korrektur der Flugbahn
  - Selbstzerstörung
- 370 Mio. $ Schaden

Motivation: Ariane 5

https://www.youtube.com/watch?v=A1gGGDG580E

Softwareentwicklung

Spezifikation

= Beschreibung des geforderten Systems

- Was soll von einem System geleistet werden?
- Was sind die funktionalen und nicht-funktionalen Anforderungen?

Verifikation

= entspricht Software der Spezifikation

- Arbeitet das Programm korrekt?
- Erfüllt das Programm die Anforderungen?

Softwaretests

Softwaretests

- Ausführen eines Systems oder einer Komponente unter bestimmten Bedingungen
- Evaluierung bestimmter Aspekte des Systems bzw. der Komponente
- "Program testing can be used to show the presence of bugs, but never show their absence!" – Dijkstra

Softwaretests

Softwaretests

- Ausführen eines Systems oder einer Komponente unter bestimmten Bedingungen
- Evaluierung bestimmter Aspekte des Systems bzw. der Komponente
- "Program testing can be used to show the presence of bugs, but never show their absence!" – Dijkstra

Unittest

- Prüfung von funktionalen Einzelteilen (Units) auf korrekte Funktionalität

Integrationstest

- Testen des Zusammenspiels voneinander abhängiger Komponenten eines Systems

Unittests

Vorteile von Unittests

Unittests

Vorteile von Unittests

- keine "Wegwerftests"
- Frühzeitiges Erkennen von Fehlern
- Gute Eingrenzung von Fehlern
- API-Dokumentation
- Refactoring überprüfen
- Verhindern von erneutem Auftreten eines Bugs
- Schneller als andere Testarten
- Design Hilfe
- Durch zwingende Vermeidung von Abhängigkeiten bleibt Quellcode modularer

Einrichten des Workspaces

- Erstellen eines Workspaces

```
1  mkdir -p ~/workshop_ws/src
2  cd ~/workshop_ws/src/
3  catkin_init_workspace
4  cd ~/workshop_ws && catkin_make
5  source devel/setup.bash #(temporary)
```

Einrichten des Workspaces

- Erstellen eines Workspaces

```
1  mkdir -p ~/workshop_ws/src
2  cd ~/workshop_ws/src/
3  catkin_init_workspace
4  cd ~/workshop_ws && catkin_make
5  source devel/setup.bash #(temporary)
```

- Neues Package anlegen

```
1  cd ~/workshop_ws/src/
2  catkin_create_pkg ros_unit_tests_workshop roscpp
```

- package.xml anpassen (Entwickler, Beschreibung, usw.)

Funktionalität

- Quellcode unter:
  `https://github.com/lhoyer/ros_unit_tests_workshop/`
- Hinzufügen my_math.h

```
1  #pragma once
2  class MyMath {
3  public:
4       MyMath() {}
5       int knobel(int x, int y);
6  protected:
7       int mLastBase = 0;
8       int nextSquare();
9  };
```

Funktionalität

- Hinzufügen my_math.cpp

```
1  #include "my_math.h"
2  #include <stdexcept>
3  int MyMath::knobel(int x, int y) {
4      if (x < 0 || y < 0) throw std::invalid_argument("
           received␣negative␣value");
5      if (x == 0)
6          return y;
7      while (y != 0) {
8          if (x > y) x = x -y;
9          else y = y - x;
10     }
11     return x;
12 }
13 int MyMath::nextSquare() {
14     int a=0, b=0, c=0;
15     mLastBase++;
16     while (a <= mLastBase) {
17         a = a+1;
18         b = a+a-1;
19         c = b+c;
20     }
21 }
```

Funktionalität

- Anpassen CMakeLists.txt

```
1  ## Declare a C++ library
2  add_library(${PROJECT_NAME}
3    src/my_math.cpp
4  )
5  ## Add dependencies
6  add_dependencies(${PROJECT_NAME} ${${PROJECT_NAME}
       _EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
7  ## Specify libraries to link a library or executable
       target against
8  target_link_libraries(${PROJECT_NAME}
9    ${catkin_LIBRARIES}
10 )
```

Einrichten von gtest unter ROS

- Anpassen CMakeLists.txt

```
1  # Compile as C++11, supported in ROS Kinetic and
       newer
2  add_compile_options(-std=c++11)
3  # ...
4  include_directories(
5    ${catkin_INCLUDE_DIRS}
6    src/
7  )
8  # ...
9  # Add gtest based cpp test target and link libraries
10 catkin_add_gtest(${PROJECT_NAME}-test test/mytest.cpp
       )
11 if(TARGET ${PROJECT_NAME}-test)
12   target_link_libraries(${PROJECT_NAME}-test ${
         PROJECT_NAME})
13 endif()
```

GTest Frame

```
1  #include <ros/ros.h>
2  #include <gtest/gtest.h>
3  #include <thread>
4  #include "my_math.h"
5
6  // Here you can add your test cases
7
8  int main(int argc, char** argv){
9      ros::init(argc, argv, "MyTestNode");
10     testing::InitGoogleTest(&argc, argv);
11     ros::NodeHandle roshandle;
12     std::thread t([]{while(ros::ok()) ros::spin();});
13     if( ros::console::set_logger_level(
           ROSCONSOLE_DEFAULT_NAME,
14          ros::console::levels::Info) ) {
15        ros::console::notifyLoggerLevelsChanged();
16     }
17     // ::testing::GTEST_FLAG(filter) = "MyTest.knobelTest
          ";
18     auto res = RUN_ALL_TESTS();
19     ros::shutdown();
20     return res;
21 }
```

GTest Frame wichtige Funktionen

InitGoogleTest(...) : Aufsetzen der Testumgebung

set_logger_level(...) : Setzen des Loggerlevels für die Tests (Debug, Info, Warn, Error, Fatal), ab dem Meldungen angezeigt werden

GTEST_FLAG(filter) = "Suite.Test" : Nur einen bestimmten Test Case ausführen

RUN_All_TESTS() : Ausführen aller Test Cases

Beispiel Test Case

```
1  TEST(MyTest, knobelTest) {
2      MyMath mymath;
3      EXPECT_EQ(1, mymath.knobel(7,13)) << "gcd(7,13)=1";
4      ASSERT_EQ(27, mymath.knobel(27,27)) << "gcd(27,27)=27
           ";
5      EXPECT_EQ(6, mymath.knobel(12,18)) << "gcd(12,18)=6";
6      EXPECT_EQ(5, mymath.knobel(5,0)) << "gcd(5,0)=5";
7      EXPECT_THROW(mymath.knobel(12,-18), std::
           invalid_argument) << "Our␣gcd␣isn't␣defined␣for␣
           negative␣values";
8  }
```

Beispiel Test Case

```
1  TEST(MyTest, knobelTest) {
2      MyMath mymath;
3      EXPECT_EQ(1, mymath.knobel(7,13)) << "gcd(7,13)=1";
4      ASSERT_EQ(27, mymath.knobel(27,27)) << "gcd(27,27)=27
           ";
5      EXPECT_EQ(6, mymath.knobel(12,18)) << "gcd(12,18)=6";
6      EXPECT_EQ(5, mymath.knobel(5,0)) << "gcd(5,0)=5";
7      EXPECT_THROW(mymath.knobel(12,-18), std::
           invalid_argument) << "Our gcd isn't defined for
           negative values";
8  }
```

TEST(Suite, Test) : Test Case mit Namen *Test* in Test Suite *Suite*

EXPECT_EQ(expected, value) : Prüfen, ob Werte gleich sind

ASSERT_EQ(expected, value) : Prüfen, ob Werte gleich sind; bei Fehlschlag Abbruch des Tests

..._NE() : Prüfen, ob Werte unterschiedlich sind

..._LT() : Prüfen, ob erster Wert $<=$ zweiter Wert

..._FALSE() : Prüfen, ob Wert false ist

..._STREQ() : Prüfen, ob Strings gleich sind

Test Case ausführen

- Roscore starten

```
roscore
```

- Alle Test ausführen

```
catkin_make run_tests
```

- Nur Test des Pakets ausführen

```
catkin_make run_tests_ros_unit_tests_workshop
```

- Beispielausgabe eines Tests

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from MyTest
[ RUN      ] MyTest.knobelTest
[        OK ] MyTest.knobelTest (0 ms)
[----------] 1 test from MyTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 1 test.
```

Fixtures

Definition

Ein Fixture ist eine Klasse, die als Umgebungsvorlage für die Ausführung eines Tests benutzt wird.

Fixtures

Definition

Ein Fixture ist eine Klasse, die als Umgebungsvorlage für die Ausführung eines Tests benutzt wird.

```
1  class MyMathTestSuite : public ::testing::Test, public
       MyMath {
2  public:
3      MyMathTestSuite() {
4          mLastBase = 5;
5      }
6  };
7
8  TEST_F(MyMathTestSuite, nextSquareTest) {
9      EXPECT_EQ(5, mLastBase);
10     EXPECT_EQ(36, nextSquare()) << "Square␣of␣6␣should␣be
           ␣36";
11     EXPECT_EQ(6, mLastBase);
12     EXPECT_EQ(49, nextSquare()) << "Square␣of␣7␣should␣be
           ␣49";
13     EXPECT_EQ(7, mLastBase);
14 }
```

Was ist schiefgelaufen?

Ausgabe:

```
[ RUN      ] MyMathTestSuite.nextSquareTest
ros_unit_tests_workshop/test/mytest.cpp:24: Failure
Value of: nextSquare()
  Actual: 49
Expected: 36
ros_unit_tests_workshop/test/mytest.cpp:26: Failure
Value of: nextSquare()
  Actual: 64
Expected: 49
[  FAILED  ] MyMathTestSuite.nextSquareTest (0 ms)
```

Was ist schiefgelaufen?

Ausgabe:

```
[ RUN      ] MyMathTestSuite.nextSquareTest
ros_unit_tests_workshop/test/mytest.cpp:24: Failure
Value of: nextSquare()
  Actual: 49
Expected: 36
ros_unit_tests_workshop/test/mytest.cpp:26: Failure
Value of: nextSquare()
  Actual: 64
Expected: 49
[  FAILED  ] MyMathTestSuite.nextSquareTest (0 ms)
```

Auflösung:

```
1  int MyMath::nextSquare() {
2      int a=0, b=0, c=0;
3      mLastBase++;
4      while (a < mLastBase) {
5          a = a+1;
6          b = a+a-1;
7          c = b+c;
8      }
9      return c;
10 }
```

Unittests

Good Practices Unittests

- Test Cases unabhängig voneinander
- Jede Funktionalität in separatem Test Case
- Ergebnisse testen, nicht Implementierung
- Überspezifizierung vermeiden
- Tests müssen deterministisch sein
- Tests müssen schnell sein