Extended Testing using GMock

Christoph Steup

November 3, 2015

Table of contents

Summary of Testing

- Testing small units of Code
- Simple tests expressing the expected behaviour
- Automated test execution to catch regressions
- Integration of the tests to the build process (catkin, rostest)

Interface: Definition

Definition: Specification

Specification is used to describe the functionality of a piece of software without any implementational detail. It can either be done in written form, in pseudo-code or even in a mathematical form.

Interface: Definition

Definition: Specification

Specification is used to describe the functionality of a piece of software without any implementational detail. It can either be done in written form, in pseudo-code or even in a mathematical form.

Definition: Interface

An interface is transformation of a (part of) specification in a concrete machine readable description. It defines the access points of a functional unit regarding ingoing and outgoing data as well as control.

Flavours of Interfaces

Class Interface : Collection of methods and members the class provides to other parts of the program

Library Interface : Collection of static functions and statically defined variables.

Communication Interface : Definition of data in a specific format usable to be transmitted byte by byte through a network connection.

User-Interface : Collection of user-interactable fields and switches.

Flavours of Interfaces

Class Interface : Collection of methods and members the class provides to other parts of the program

Library Interface : Collection of static functions and statically defined variables.

Communication Interface : Definition of data in a specific format usable to be transmitted byte by byte through a network connection.

User-Interface : Collection of user-interactable fields and switches.

Commonality

Interfaces define the possible interaction between entities. One entity may only use the features of another entity through the interface as specified.

Examples

A Class Interface

```
class RoadInterface {
  public:
    enum class Cell { Free, Blocked, Car };
    virtual ~RoadInterface() {}
    virtual Cell readCell( unsigned int y,
                           unsigned int x ) const =0;
    virtual void writeCell( unsigned int y,
                            unsigned int x, Cell cell ) =0;
};
```

A ROS Network Interface

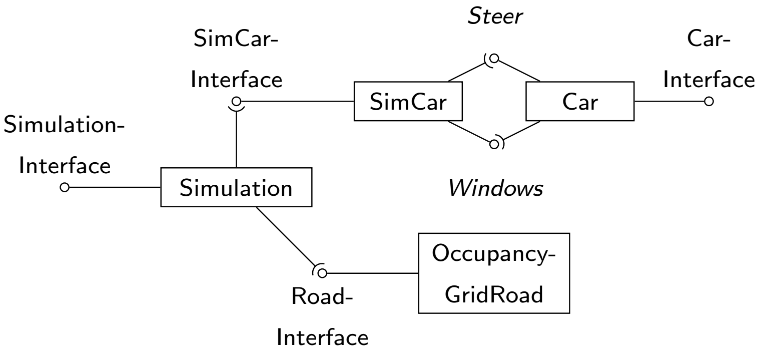
```
Header  header
int32   cmd

int32 Straight =  0
int32 Left     = -1
int32 Right    =  1
```

Component-based Software Design

Definition: Software Component

A software component is a functional unit with clearly specified incoming and outgoing interfaces. The incoming interfaces need to be fulfilled by other components for this component to be used.

Component structure of the ROS Testing Example



Benefits of Software Components

Decoupled Devopment : Developers may focus on a single piece of software. Limits the effects of software changes to the connections between the components.

Reusage of Components : A developed component may be used in any context as long as the interface specifications are fulfilled.

Easier Testing : Each component may be tested individually against the specification of its interface, without using any other component

How to test

The Problem

How can the dependancy of a software component be fulfilled without using another component fullfilling the interface.

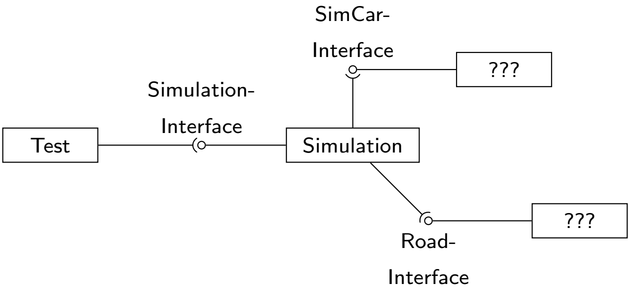


Figure: ROS Example Simulation Testing Challenge

Solutions

Definition: Fake Object

Fake objects have working implementations, but usually take some shortcut (perhaps to make the operations less expensive), which makes them not suitable for production. An in-memory file system would be an example of a fake.[1]

[1] GMock for Dummies: https://code.google.com/p/googlemock/wiki/ForDummies

Solutions

Definition: Fake Object

Fake objects have working implementations, but usually take some shortcut (perhaps to make the operations less expensive), which makes them not suitable for production. An in-memory file system would be an example of a fake.[1]

Definition: Mock Object

Mocks are objects pre-programmed with expectations, which form a specification of the calls they are expected to receive.[1]

[1] GMock for Dummies: https://code.google.com/p/googlemock/wiki/ForDummies

Mocking an Object

Mock Methods

A Mock Method replaces the abstract or incomplete implementation of a **virtual** method of a base class. It is defined without any implementation while fulfilling the class' interface.

RoadInterface Mock Class

```
class MockRoad : public RoadInterface {
  public:
    MockRoad(unsigned int maxY, unsigned int maxX)
      : RoadInterface(maxY, maxX) {}
    MOCK_METHOD3(writeCell, void(unsigned int y,
                                 unsigned int x, Cell
    MOCK_CONST_METHOD2(readCell, Cell(unsigned int y,
                                      unsigned int x))
    MOCK_CONST_METHOD2(isFree, bool(unsigned int y,
                                    unsigned int x));
};
```

Expectations in Value

EXPECT_CALL declares a Mock Method to be expected to be called

- Lt is a *Matcher* comparing provided arguments to be **L**ower **t**hen a reference.
- _ is the wildcard *Matcher* stating all arguments should be matched.

Expecations on Value Example

```
using ::testing::Lt;
using ::testing::_;

EXPECT_CALL(road, isFree(Lt(road.maxY()),
                          Lt(road.maxX())
                          ));
EXPECT_CALL(road, writeCell(_, _,
                             RoadInterface::Cell::Car
                             ));
```

Expectations in Time

`.Times()` declares the expected number of calls to a mock method with the specified arguments

`AtLeast()` declares a minimum amount

Expectations on Time Example

```
using ::testing::AtLeast;

Test(SimulationTest, carCreationTest) {
  MockRoad road(10, 10);

  EXPECT_CALL(road, isFree(_, _))
    .Times(AtLeast(1))
  EXPECT_CALL(road, writeCell(_, _, _))
    .Times(1);

  Simulation<MockCar> sim(road);
  sim.createCar("TestCar");
}
```

Returning Values

.WillOnce() : Modifies the Mock Method to do an **Action** a single time

.WillRepeatedly() : Modifies the Mock Method to do an **Action** all the time

Return() : return the specifies value as the **Action**

Example of a Return Action

```
using ::testing::Return;
MockRoad mRoad(10, 10);
EXPECT_CALL(road, isFree(Lt(road.maxY()),
                         Lt(road.maxX())))
  .Times(AtLeast(1))
  .WillOnce(Return(true))
  .WillRepeatedly(Return(false));
```

Side Effects

Assign(T* ptr, T v) : assigns `v` to the address `ptr` as the methods action

Example of a Side-Effect Action

```
class MockCar : public SimCarInterface {
  public:
    MOCK_METHOD1(x, void(unsigned int x));

    MockCar(const string& numberPlate, Dir dir,
            unsigned int y, unsigned int x)
      : SimCarInterface(numberPlate, dir, 1, 1) {
      EXPECT_CALL(*this, x(2))
        .Times(AtLeast(1))
        .WillRepeatedly(Assign(&mX, 2));
      }
};
```

[0] Many more actions and receipes can be found in the cookbook: https://code.google.com/p/googlemock/wiki/CookBook

Catkin Integration

No Default Catkin Integration

Unfortunately; there is currently no GMock integration within catkin available. Therefore, a workaround needs to be used.

GMock Catkin Integration Workaround

```
if(CATKIN_ENABLE_TESTING)

  # Create a gmock target to be used as a dependency
  # by test programs
  add_library(gmock IMPORTED STATIC GLOBAL)
  set_property(TARGET gmock PROPERTY IMPORTED_LOCATION
              /usr/lib/libgmock.so)

  # Add gtest based cpp test target and link libraries
  catkin_add_gtest(${PROJECT_NAME}-mock src/Mock.cpp)
  target_link_libraries(${PROJECT_NAME}-mock
                        ${catkin_LIBRARIES} cars gmock

endif()
```

VCS History

Finding A Suspicous Commit

Using Unit-Tests the relevant file is normally easy to track down. After the relevant file or a list of files is found a look to the history of this file is necessary.

Getting the broken code

```
git checkout origin/regression -b regression
```

Run the tests

```
catkin_make run_tests
```

VCS History

Finding A Suspicous Commit

Using Unit-Tests the relevant file is normally easy to track down. After the relevant file or a list of files is found a look to the history of this file is necessary.

Getting the broken code

```
git checkout origin/regression -b regression
```

Run the tests

```
catkin_make run_tests
```

Have a look at the log

```
git log src/cars/src/Car.h
git log src/cars/src/Car.cpp
```

Rewriting History

Verify changes

```
git diff 4a1e0ef731 src/cars/src/Car.cpp
git diff 35a8437a89 src/cars/src/Car.cpp
```

Rewriting History

Verify changes

```
git diff 4a1e0ef731 src/cars/src/Car.cpp
git diff 35a8437a89 src/cars/src/Car.cpp
```

Repairing Broken Things

NEVER ever rewrite the history of the uploaded repository. If a single commit is responsible it is easiest to fix the problem by reverting the whole commit. If not change the code and commit the changes as normal.

Rewriting History

Verify changes

```
git diff 4a1e0ef731 src/cars/src/Car.cpp
git diff 35a8437a89 src/cars/src/Car.cpp
```

Repairing Broken Things

NEVER ever rewrite the history of the uploaded repository. If a single commit is responsible it is easiest to fix the problem by reverting the whole commit. If not change the code and commit the changes as normal.

Repair Everything

```
git revert 35a8437a89
git revert 4a1e0ef731
```

Left Overs

- Runtime Testing using Ros::SelfTest