Introducing Automated Testing to a Legacy System
Master Thesis

Ulrika Malmgren

June 10, 2015

Abstract

This is where I'll write my abstract

Contents

1 Introduction

1.1 Setting the scene

1.1.1 About Nobina

Nobina is the Nordic leader in scheduled public road transport services, with a market share of 16%. The company is also one of the ten largest public transport companies in Europe. It's most wellknown brand is Swebus Express which 2 million passengers use each year for long distance traveling. Nobina has around 280 million passengers per year within the companys two business units: contract traffic and express bus traffic. The bus fleet includes 3,500 vehicles and the company has 7,500 employees.

1.1.2 About the OMS system

As support for the production of these transport services, Nobina has built a specialized system, the Operation Management System (OMS) which was first released in 2003. OMS is comprised of 15 modules within 3 different areas: staff planning, vehicle maintenance and traffic management. It is unique since there is no system on the market that combines these areas. The system has 900 active users and is used in four countries (Sweden, Finland, Norway and Denmark).

OMS is a Winform application written in Vb.Net (80 %) and C# (20 %). There is a general plan for the architecture of the modules but since each developer is responsible for one or two modules different programming styles exist.

This thesis will look more closely at the Workshop and Fuel modules in OMS. In the Workshop module, there are functions related to maintaining the vehicles in good condition such as error reports, servicing, workshop planning, and so on. Fuel on the other hand, handles the fuel consumption of the vehicles, the amount of kilometers driven, the fuel levels at the different garages, etc.

1.1.3 Releases and testing

The system is released four times per year and warmfixes are released about once a month per module or two times a week for the entire system. Before each release manual testing is performed.

For testing a release, 20 people are bought in externally and test the system for two days. Critical bugs are fixed and the system retested during an additional day.

For a warmfix, only the concerned functionality is tested but for a regular release the entire system is tested.

1.2 Problem statement

Testing in this way is expensive and Nobina wishes to lower their testing costs by using automated testing instead. They have identified two main techniques for

approaching automated testing: Graphical User Interface testing (GUI testing) and unit testing. Nobina wishes to find out which of these two approaches is preferrable to choose in order to reduce their testing costs.

However, because of the OMS system has been in development for several years and testability hasn't been in a factor in the development process, introducing testing has challenges. The code must be altered in order to introduce tests. The developers are not trained in automated testing and the test environment is not designed to support it.

The question this thesis attempts to answer is: is it preferrable to introduce automated in the form of GUI testing or unit testing to reduce testing costs for the OMS system?

1.3 Purpose

The purpose of the thesis is to discuss and compare two different methods of introducing automated testing to a legacy system.

1.4 Goal

The result of the thesis is a conclusion based on a selection of comparison criteria and lessons learned from apply the two different test automation techniques.

1.4.1 Benefits, Ethics and Sustainability

The development team at Nobina benefits from the thesis by receiving input towards selecting a strategy for introducing automated testing.

The thesis work is conducted internally at Nobina in parallell with the development team. Except for the lead developer and the thesis supervisor, no one was affected by the work. No personal data was handled during the thesis work. Hence, there are no ethical implications from this thesis.

From a sustainability point of view, the thesis does not impact the product lifecycle of Nobina products or the development team.

2 Testing

2.1 Purpose

Testing is the process of gathering information about a system with the intent that the information could be used for some purpose. [1]

By testing, information is uncovered about the system's capabilities or lack thereof. This information can be used to make decisions about the system. For example, if it is ready for release or not, if there are defects which need to be fixed or if it meets the needs of its users.

2.2 Testing techniques

There are a lot of different testing techniques which can be used to gather information. This section aims at describing a few of them which are relevant for this thesis.

2.2.1 Black box testing

Black box testing is a strategy in which no knowledge of the internal structure of the system under test is required. It can be applied at all levels of the systems from unit to system testing.

A disadvantage of black box testing is that it is impossible to be sure of the coverage of the tests since the internal structure isn't known.

An example of black box testing is GUI testing.

2.2.2 White box testing

In contrast to black box testing, in white box testing, the internal structure guides testing. It can also be applied to all levels of the system but usually requires programming skills to perform. An example of black box testing is unit testing.

2.2.3 Regression testing

Regression testing is the process of testing in order to find a regression, functionality which has worked in the past but which has stopped working. Regression testing is typically performed right before a release to make sure that in the addition to the new functionality, the old functionality is also working as intended.

2.2.4 Scripted testing

Following a list of predefined test cases is called scripted testing. The test cases can be specified in a test management tool or a simpler tool like a spreadsheet. The tester will repeat the test cases exacly as specified and write down any anomalies compared to the script.

The advantage of scripted testing is that it can be performed by anyone who knows how to read the script and that it is consistent in its coverage of the system.

Drawbacks include the human error factor of doing repetitive tasks over and over again and the time requirements of performing these tests. There is also concern about the amount of defects such an approach really finds compared to an approach where the script isn't followed step by step. [2]

2.2.5 Automated testing

Automated testing is writing code to perform tests on the system. By automating tests, they can be run by any one in the development team (or for example a build server) and can provide feedback fast about the state of the system. The

automated tests are very similar to the scripted tests since the main difference between them is that a computer performs the automated tests and a human performs the scripted tests.

They have an advantage over the scripted tests since they are not prone to human error when performing the tests and can be faster.

However, they are not entirely free from impact of human error since humans are still needed to write the tests and run them. [2]

For example, a risk with automated testing is that automating bad tests only means that bad tests will be performed faster. A bad test could be a test which doesn't actually test anything, it would pass in every scenario. The automated test will provide a false sense of security without actually providing any. [3]

Automated testing can be performed on a lot of different levels in the system such as API, GUI, unit, integration, and so on. In this thesis, only unit and GUI testing are considered.

2.3 Prerequisites for effective testing

What makes for a good test? Reproducable - problems with test environment are interesting - test data?

3 GUI testing

3.1 Introduction

A black box testing technique for end-to-end tests.

3.2 Record and play

I plan to write about how my GUI-test framework isn't record and play and why that is good. So should I write about record and play here or when I describe it?

3.3 Advantages

Being a black box testing technique, automated GUI testing doesn't require knowledge of the internal structure of the system and the tests can be set up without it.

3.4 Drawbacks

Gui testing pitfalls: coverage criteria. The mapping between gui events and code is not straightforward. It is impractical to try to generate all test cases, a selection has to be made but it is also hard to select a good subset. [4]

Layout changes means that the tests might need to be changed because components change. But also outputs can change. [4]

Time consuming, false positives, flaky Changing GUI means changing tests - doh!

Record and Play vs other

4 Unit testing

4.1 Introduction

A white box testing technique for the smallest units of functionality that comprise a software system: functions, classes and methods.

When unit testing is performed well, the units are tested in isolation as opposed to integration testing when the units are tested together. This means that the tests will find defects in the internal workings of the units and not in their interaction with other units.

4.2 Advantages

In *On the Effectiveness of Unit Test Automation at Microsoft*, a number of advantages with unit testing are presented. For example, an automated unit testing practice meant a 20.9% decrease in test defects, and relative decrease of defects found by customers. Also, developers felt they spent less time fixing bugs found by testers. Unit testing was said to raise awareness for error conditions and boundary cases as well as increasing the understanding for code written by others. The study reported that it was harder for testers to find bugs after the introduction of a unit testing approach than before. [5]

4.3 Drawbacks

4.4 Success factors?

4.5 TDD

[5] says 40% fewer test defects with TDD and increased performance.

I didn't use TDD, do I still mention this?

5 Legacy code and refactoring

5.1 Introduction

The OMS system has been running for seven years already at the time of the thesis and is considered to have legacy code. Before unit testing can be used to test the system, it needs to be refactored.

5.2 Legacy code

Legacy code refers to source code that isn't new and the common interpretation is code inherited from someone else. Often this constitutes a problem since the original developer's intentions can be hard to discern. For example by the use of magic numbers and unexplained variable names. It is not unusual that legacy code is left untouched simply because new developers are afraid to break it.

Maintaining and developing legacy systems comes with a lot of different challenges. A system that has been developed throughout several years can easily become a patchwork of different technologies and ideals as many different developers leave their marks on it. Some technologies are viewed as outdated by new developers and as technologies develop; bottlenecks are moved to other places.

Another definition for legacy code is code without tests [6]. Legacy code does not necessarily have to do with the age of the code but rather code that is difficult to maintain. When the system works in incomprehensible ways, developers are afraid to introduce subtle bugs. However, if there are tests written for the system, it is possible to change the code with impunity. A test can provide an invariant that lets the developers know if they have changed the behavior of the system.

5.3 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the behavior of the code yet improves its internal structure and a refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. [7]

5.3.1 Purpose

The general goals of refactoring are improved software design, improved understanding and help finding bugs.

There are several scenarios for when refactoring is used. For example, even though a piece of code, class or function has been written with a good design, as the system evolves and new functionality is added the good design might become obsolete. In order to improve the general design, refactoring the original code is needed.

In another scenario, the code might be bad to begin with. Urgency or lack of time might have forced a developer to solve a problem quickly, without having the time to create a well-designed solution.

A third scenario is the one for this thesis, refactoring in order to be able to add automated tests for the system. If the code base lacks tests it might not be written in a way which accomodates for tests to be added by lack of clear interfaces for example. Refactoring is then necessary to be able to add test to this existing code base.

5.3.2 Advantages

Software design is improved by reducing complexity. Some techniques include removing duplication, breaking up large functions into smaller ones, breaking dependencies, and so on. Reducing complexity of the software makes it easier to understand, simpler to maintain and less prone to involve bugs.

Improving understanding is important since there are more users of the code than one might initially think of. Besides the computer which reads and processes the code, there are the developers who will be reading the code after its been written. As a developer when a new feature is to be added or a bug is to be fixed, a lot of time is spent reading and trying to understand old code.

Less effort required after changes in [8].

5.4 Testability in code

5.4.1 Single responsibility principle

Break dependencies Separation of concerns

5.4.2 Code complexity

IIn Refactoring for changeability: a way to go? and Measuring Technology Effects on Software Change Cost, it is shown in a case study that refactoring can significantly decrease the number of customer reported defects and effort needed to make changes.

6 Approach and Methodology

This section explains the approach used in this thesis which includes the procedure, the selection of cases for analysis, the comparison criteria from which the thesis will make a conclusion and the frameworks used to write the tests.

6.1 Procedure

In order to evaluate the benefit of one method over the other, on top of adding automated tests to this system, it is also necessary to look at how difficult it is to maintain the tests when the system changes.

As it is not in the scope of this thesis to implement changes in the system, changes that are planed or have been performed and that are suitable for this research will be found. By looking at the work items that the OMS team has been working on, it is possible to select a change that seems appropriate.

Because the source code is under version control, it is possible to select a certain change set from the source code repository and set the system up as it was at a certain point in time. The same procedure is done for both the GUI tests and the refactored tests.

The following procedure will be used:

1. Tests are created for the scenario prior to the change being implemented. The purpose is to pretend that there were test cases for the system before the change specified in the work item is implemented.

2. The change is rolled onto the system.

3. The test cases are changed in response to the change. Tests might be added, modified or removed.

The procedure is applied first with GUI tests and later with unit tests for each case selected.

6.2 Case selection

With the time constraints for the Master's Thesis, it is necessary to pick a representative set of changes to look at. An analysis of the three previous releases and an interview with one of the developers gives an indication of what kind of work items that are typical for this system and this module in particular.

6.2.1 Past releases

	GUI	Logic	Both	Total
February	1	2	2	5
May	7	1	3	11
September	1	0	1	2
Total	9	3	6	18

Two thirds of the work items have impact on the GUI. More than half of these are mere modifications of the GUI, whereas the rest also concern the business logic. Work items that only affect the business logic are few.

Most work items are of moderate size and a few of the logic changes are a bit larger.

6.2.2 Future releases

The interview reveals that more GUI changes are to be expected in the future as the GUI is being polished and the system owner stresses the need for increased usability.

6.3 Comparison criteria

6.3.1 Time

One important aspect of this thesis is to compare the two methods with regards to how difficult it is to maintain the tests. An obvious way of doing this is to measure the time it takes to update the tests after the system has changed.

Time measurement can be tricky to perform and during the short time span available for the thesis, it is not possible to get enough data in order to do a statistical analysis on it.

However, it is still interesting to use the recorded times when discussing the results. The time that is recorded is the time spent working on the problem at hand. Time used for example for discussions and solving problems with the test or development environment will not be recorded as efficient time spent on the test or refactoring effort.

Even though a lot of focus is placed on the maintenance effort for the tests, the time it takes to set up the tests will also be recorded. If one method is very costly it, this needs to be taken into account when comparing the two. As a matter of fact, if both methods are costly, it will still be interesting to know this.

Because the procedure used is refactoring code, adding unit tests and then switching to a newer version of the code through the version control system, the refactored code will not in place after the new version has been switched to. The time it takes to reproduce the refactoring in the new version is not accounted for. This is because the interesting scenario is to see how much effort it takes to update the refactored code and its tests.

6.3.2 Changes required

There are other ways of measuring effort as well. In *A Case study on Regression Test Suite Maintenance in System Evolution* [9], the authors log the steps of their work during its progress. Their work builds on top of a previous case study where a system has been added a feature using three different change strategies. Skoglund and Runeson look at the maintenance effort required to update the test suite of the system in each case.

The log includes information such as compilation errors in the test suite, number of lines of modified test code and number of lines of modified system code. Inspired by this approach, this thesis will also present some of the same metrics.

6.3.3 Test runtime

One of the conclusions in [5] was that it should be easy to run the unit test suite and that the run time for the test suite needs to be short.

6.3.4 Code complexity

As shown, an advantage of reducing complexity of the code is that a code base with less complaxity is less prone to errors and therefor it is interesting to observe if the code complexiy goes up or down with both methods.

6.3.5 Defects

A typical criteria for judging the effectiveness of a test technique is the number and severity of the defects it catches (such as it is done in [5]).

6.3.6 Training

Since none of these methods are currently in place at Nobina, a certain amount of training will be required for the developers to learn how to apply them. It is interesting to compare how much training this would be in both cases.

6.4 Frameworks

Two different frameworks are used in this thesis. For GUI tests, the White framework and for unit tests, NUnit.

6.4.1 Project White

An open source Framework for .NET applications called White is used for the GUI tests. The framework is based on Microsoft's UIAutomation library and window messages. The UIAutomation library was originally developed for Assistive Technology products. That is, products designed to provide better access for individuals with physical or cognitive difficulties, impairments, or disabilities. [10]

Through this API it is possible to interact with user interface (UI) elements. White provides functions that can be used to search for, access and manipulate UI elements in an application by name, automation id or type.

For example:

```
Private Sub Click_add_serviceobjects_in(
            ByVal workorder_window As UIItemContainer)
  Dim addButton = workorder_window.Get(OfButton)(SearchCriteria.
                  ByAutomationId("btnServiceObjects"))
  addButton.Click()
End Sub
```

Listing 1: A method which finds a button with id *btnServiceObjects* and clicks on it.

The framework also provides the possibility to log the UI elements found in a certain window, enabling the developer or tester to get further information about their names, types and attributes.

Since the elements are accessed with code and not through a recording, the tests are less sensitive to changes in the GUI. For example, as a long as a button keeps the same automation id, it doesn't matter if it is moved to another place on the GUI.

6.4.2 NUnit

NUnit is an open source framework for unit testing all .Net languages. [11]

Eh... what else?

7 Case studies

7.1 Case 1: The Service Object List

In the Workshop module of the OMS system, focus is on keeping the vehicles in good shape by performing repairs and making sure that they receive servicing in due time. For this, a list of all the objects on the vehicles that can require servicing is being maintained in the database.

Each service object has a certain interval between services. This can for example be after a certain amount of kilometers driven or after an amount of time.

Some objects are external objects and service is performed on them by an external contractor, the other objects are serviced in the Nobina workshops. When an object is up for service it is said that it has a warning.

When a mechanic opens the system, a list of all vehicles at this garage that require service is displayed.

Fig x. Part of the graphical interface of the OMS system. Here a list jobs.

Original scenario *When opening a work order, all the service objects should be displayed.*

By opening one of those work orders, there is a list with check-boxes in the interface that displays all service objects. The ones that are eligible for service are checked.

Fig x. Work order, in the middle on the right is the list of service objects.

Updated scenario *When opening a work order, only the service objects eligible should be displayed.*

The feedback received says that it is hard to visualize which items should receive service since the list is quite long and the box rather small. The development team came up with the idea of showing only the eligible and external items with warnings in the list. The complete list of service object can be accessed by a button which opens up a new interface.

Fig x. Service object list management window.

In this interface, it is possible to select other service items to be added to the list that is displayed in the work order.

When the mechanic closes the work order, the system makes the conclusion that the checked items in the list have received service and updates the database accordingly.

7.1.1 GUI test

Issues with test data In order to be able to develop tests for this scenario, it is crucial to have access to a bus that is in need of service and to know which service objects will be in the list.

However this information is not stable. For one, because of the nature of service objects whose status in some cases is updated every day. Also since the

test database is updated every two days, a vehicle can be repaired and be taken off the list of work orders. This means that the test data would need to be updated every day with new information.

To solve this, the first available vehicle in need of service is selected in the work order list instead of selecting a specific vehicle.

Also, a conditional compilation statement is placed around the area of the code where the service object list is retrieved from the database. With the conditional compilation it is possible to either compile a normal version of the system, or one where specific test data is used instead. Since this information is retrieved as a collection, a method is implemented to be able to create a collection with test data.

This is needed to get the GUI tests to work without a proper test database but it is also good support for the unit tests written.

Implementation of base scenario The GUI test follows these steps

1. Open the OMS application
2. Find the correct menu option to open the workshop main form
3. Select the tab for ongoing work orders in the workshop main form
4. Double-click on one of the service orders in the list of work order to open it up
5. Verify that all the service objects in the service object list are displayed in the checked list

Implementation of the updated scenario The updated scenario follows these steps

1. 1 to 4 are similar to the original GUI test that is up until the point where the work order is opened.
5. Verify that only the eligible objects and the service objects are displayed in the checked list
6. Click the Add button
7. Verify that all service objects are displayed in the checked list

7.1.2 Unit tests

The change that is being introduced in the code concerns which items from the service object list are displayed in the checked list.

Test cases for the base scenario The method works in four steps. First a collection of service objects is retrieved from the database. Second, it goes through the collection and marks the items that are eligible for servicing. Third, it transforms this list into one that can be displayed in the checked list in the GUI and finally, it displays it. By breaking out the second and third steps into separate methods, they can be tested individually. The method that produces a collection of service object that was made for the GUI test is helpful as input to the tests.

Given	A collection with all kinds of service objects
When	A list is created
Then	All objects are in it

Table 1: Test case 1 - the service object list should contain all service objects

Given	A collection with no objects with warnings
When	A list is created
Then	All objects are in it

Table 2: Test case 2 - the service object list should contain all service objects

Given	An empty collection
When	A list is created
Then	The list is empty

Table 3: Test case 3 - the service object list is empty if there are no service objects

Test cases for the updated scenario In the updated scenario, the entire list of service objects isn't displayed all the time. Rather, in one case only the checked objects and the external objects with warnings (in the work order) are to be displayed and in the other case the entire list is to be displayed (in the service list management window).

In code, this is being achieved by adding a new field: ShowAllObjects. Depending on whether ShowAllObjects is true or false, the list is generated with different criteria.

This means that besides the two methods created in the base scenario, one more is added that creates a list with only the eligible items. The old tests can be kept, since they still apply to the other methods but it is necessary to create a new set of tests for the new method.

To be certain of the correctness of the refactoring, both the test suite of unit tests and the GUI tests can be run. By running the GUI test, it is possible to feel confident that the code changes done have not created any defects since the

Given	A collection with all kinds of service objects
When	A list is created
Then	Only checked and external objects are in it

Table 4: Test case 4 - the service object list should contain only checked and external service objects

Given	A collection with no objects with warnings
When	A list is created
Then	Only external objects are in it

Table 5: Test case 5 - there is no error when the list has no checked objects

Given	An empty collection
When	A list is created
Then	The list is empty

Table 6: Test case 6 - the service object list is empty if there are no service objects

GUI test would likely fail in that case. This is true for both the original scenario and the updated scenario.

Implementation An example test for the updated scenario:

```
<Test()> Public Sub
    Given_DataViewWithAllKindsOfserviceItems_When_ArrayListIsCreated_
Then_OnlyItemsElligibleForServiceShouldBeInIt()
  'Arrange
  Dim myDataset As DataSet =
      CreateTestDataSetWithAllKindsOfServiceItems()
  Dim myDataView As New DataView(myDataset.Tables(0))
  Dim myArrayList As ArrayList
  myArrayList = New ArrayList

  'Action
  myArrayList = CreateArrayListOfCheckedServiceObjects(myDataView)

  'Assert
  'ArrayList has x elements
  Assert.AreEqual(2, myArrayList.Count)

  'ArrayList contains right elements
  Assert.AreEqual("[-Alkols-] (309 - 180 Days) *",
      myArrayList.Item(0).ToString)
  Assert.AreEqual("Brnslefilter (65797 - 60000 Km) *",
      myArrayList.Item(1).ToString)
End Sub
```

7.2 Case 2: The Fuel Outtake and Odometer

This case involves the Kilometers and Fuel module where it is possible to add fuel outtakes and readings for odometers, pumps, cisterns and so on. Every time a driver makes a fuel outtake they needs to add it to the OMS system with the date, time, new mileage for the vehicle and volume of the outtake.

A special case which can occur is when the odometer reaches 999 999 and rolls over to 0. When the user tries to enter a value for mileage that is smaller than the previous mileage, an error is registered and fuel outtake is flagged in the system. A manager then needs to handle it and call support to make a manual change in the database.

Base scenario *Adding a fuel outtake should be saved*

It is not possible to save a fuel outtake with an odometer value smaller than the original so the base scenario tests that saving a normal odometer value works correctly.

Updated scenario *Adding a fuel outtake when the odometer is about to roll over should trigger a dialog box*

Since this is extra work for the managers, a solution has been devised to handle one roll over of the odometer. When the system notices that the previous mileage is larger than 900 000 and that the new mileage entered by the user is smaller than 10000, a dialog box is displayed asking the keuser if the odometer has rolled over. When the user selects yes, the database is updated accordingly.

Prerequisites Because it is not possible to save a fuel outtake with invalid parameters, it is not possible to repeatedly save fuel outtakes for the same vehicle with the same parameters over and over again. This means that each time the GUI test runs, it is necessary to change the inventory number for the vehicle or the time of the outtake.

Another issue concerns the case when a previous mileage that is larger than 900 000. In that case it is necessary to identify a vehicle with that mileage and use it in the test. However, after the test is run, the database has been updated and the vehicle no longer fills that prerequisite. Since the number of vehicles with mileages around 990 000 are few, there are a limited amount of possibilities to test. This is solved by adding conditional compilation to the source code. When compiling the source for tests, the previous mileage is changed to satisfy our conditions.

7.2.1 GUI test

Base scenario

1. Open the OMS application
2. Find the correct menu option to open the Km & Fuel main form
3. Enter an inventory number
4. Use today's date
5. Enter a time
6. Fill in new mileage
7. Verify that the amount of driven kilometers displayed is correct
8. Fill in a fuel volume
9. Verify that the average fuel consumption displayed is correct
10. Click save
11. Verify that the fuel outtake is displayed in the datagrid

Updated scenario IT IS NOT AN UPDATED SCENARIO, IT IS AN ADDITIONAL SCENARIO!

1. Steps 1-5 are the same as in the base scenario
6. Verify that the label displaying the previous mileage is what it should be
7. Fill in new mileage
8. Verify that a dialog box is opened
9. Answer Yes
10. Verify that the amount of driven kilometers is correct

7.2.2 Unit tests

In order to be able to create unit tests, some refactorings are needed. For example, in the source of the form there is a method with dependencies towards the GUI since it uses the information in some labels (lblReportFuelPreviousKm), the values of fields (txtKm.Text) and displays a dialog box.

```
Private Sub CheckIfKmOverOneMillionAndIfSoAskUserToCorrect()
  If IsNumeric(lblReportFuelPreviousKm.Text)
    AndAlso (CInt(lblReportFuelPreviousKm.Text) > 900000
    And CInt(lblReportFuelPreviousKm.Text) < 1000000) Then
      If IsNumeric(txtKm.Text) AndAlso (CInt(txtKm.Text) > 0 And
          CInt(txtKm.Text) < 10000) Then
        If _millionKM = 0 Then
```

```
        If
            MessageBox.Show(My.Resources.KM2Messages.msgVehicleMeterOverMillion,
          My.Resources.KM2Messages.msgVehicleMeterOverMillionTitle,
          MessageBoxButtons.YesNo,MessageBoxIcon.Question)=
              DialogResult.Yes Then
            Dim modifyKM As New KMModifyKM
            MillionKM += 1
            modifyKM.UpdKM_VehicleMeter(_vehicleMeterId, VehicleID,
                VehicleMeterType, MillionKM, AvgConsumption)
            CalculateDrivenKm()
        End If
      End If
    End If
  End If
End Sub
```

At the same time it performs some business logic based on the values in the labels, the text fields and a global variable called millionKM. This method is refactored so that the business logic is broken out into an Odometer class with the single responsibility of handling properties and methods proper to the odometer. Since we break the dependencies towards the GUI, it is possible to write unit tests to test the Odometer class separetely.

A second part of the test scenario is to verify that the calculated amount of driven kilometers is correct. By adding a helper class to the form class and extracting the logic that calculates an amount of driven kilometers given certain paramaters, it is also possible to test this functionality through unit tests.

8 Results

8.1 Time

In the tables below we can see the times for developing the test cases before the anticipated change is rolled on (base scenario) and after it has been introduced (updated scenario).

8.1.1 Data

	Base scenario	Updated scenario
GUI tests	39	13
Unit tests	13	3

Table 7: Measured time in hours for Case 1: Service Objects

	Base scenario	Updated scenario
GUI tests	20	11
Unit tests	21	12

Table 8: Measured time in hours for Case 2: Odometer

8.1.2 Designing tests

When looking at the time measurements for designing the base scenario tests, it appears that in the Service Object Case, the effort for putting the GUI tests in place is much larger than the effort for the unit tests.

This can be explained by the fact that the Service Object Case was the first case developed and thus effort had to be put into learning the framework. Another explanation is that interacting the GUI testing framework with the service object list was complex and presented problems.

The procedure for accessing the form to test requires going through a few steps before it can be opened. The components in those steps are also more complex to work with in White than the ones in the fuel outtake form.

In the Odometer Case, the results are different. Here, creating GUI tests and unit tests take a similar amount of time. In this case, the GUI components were straight forward to work with but the unit tests required mentoring to be able to complete.

8.1.3 Updating tests

In both cases and with both methods, updating the tests takes a shorter amount of time than creating them. This is because the general structure for the test is already in place.

For GUI tests, the steps leading up the to form where the change occurs are reused in the updated scenario. For unit tests on the other hand, the tests have to be changed and new added.

In the case of the Service Object List, it is interesting to see that the GUI tests which took longer to create also take longer to update. It seems that the complexity required to add the tests is maintained when the tests need to be updated.

8.1.4 Conclusion

It was known from the beginning that the time measurements would not provide any statistical evidence to conclude upon. It is possible to tell however that GUI tests might take longer to implement and update but not necessarily in all cases.

Also, applying GUI testing to complex user interface components such as the service object list will take longer than simpler components. Both when the tests are created but also when they are updated.

8.2 Changes required

8.2.1 GUI changes

	Base scenario	Updated scenario	Delta
Case 1	5	7	+2
Case 2	12	14	+2

Table 9: Amount of GUI controls used in the tests

This is interesting because of maintainability - changing gui often means breaking tests - Reference to GUI testing - drawbacks?

It is possible to conclude that the changes that have been implemented here are relatively small and comparable in size. In both cases, two new controls are accessed after the change has been introduced. We know from the previous analysis of the work items for the two modules that these types of changes are typical for the Workshop and Fuel modules.

8.2.2 Code changes

Here I'll put a table with measurements for amount of code changed.

8.2.3 Test data

For the OMS system, there is a test database that is a copy from the real database. Every other day, a fresh copy is made. All the developers share the same database. This presents a problem for this thesis.

Since it is not possible to know if some data will be present in the future, it is not certain that the tests written today will still work tomorrow. The tests are not able to run over and over again on the same entities.

Because of the complex structure of the database it is not possible to create specific test data that can be used for testing in any easy way.

If the test requires a bus that has been driven a certain amount of kilometers, it is not possible to simply create such a bus. Neither is it possible to change the amount of kilometers for an existing bus because it might violate existing rules in the database.

Also, some operation can only be performed once. For each time a test is rerun, a new object to perform the same action on has to be found.

Because of these problems, some workarounds have been introduced for the purpose of completing the thesis by adding conditional compilation statements before running the tests. This means that test data was programmatically inserted into the system when the application was started with a '- test' flag.

This was a problem for the GUI tests because it was necessary to set up certain conditions for the tests to be able to run. The test data in the test environment was not in such a state that it was possible to design the tests without first figuring out how to create special data for the tests. This took some time and effort and created additional complexity for the tests.

8.3 Test runtime

8.3.1 Data

Using the testrunner, data on the time it takes to run the tests has been gathered.

	Scenario	Number of tests	Time (ms)
GUI tests	Base	1	37000
	Updated	1	61000
Unit tests	Base	5	45
	Updated	7	46

Table 10: Measured average runtime after 10 runs for case 1 - Service Object List

	Scenario	Number of tests	Time (ms)
GUI tests	Base	1	6000
	Updated	1	3000
Unit tests	Odometer	10	18
	Helper	5	17

Table 11: Measured average runtime after 10 runs for case 2 - Odometer

8.3.2 Analysis

There are interesting observations to make regarding the runtime measurements.

Differences between the GUI tests Surprisingly, the second case utilizes more components (see table x in amount of change) yet runs faster. An explanation for this is that when White searches the GUI for a specific component it goes through all of the components in the target window. This means that in a window with many components search will be slower than in one with fewer components. This is the case here as the window containing the list of jobs in the service object list case is more complex than the fuel outtake window in the other case.

Differences between unit tests A difference in runtime for the unit tests in the different cases is noticed with around 46 ms in the first case and around 18 ms in the second case. An explanation for this is the different type of inputs for the different tests. In the first case, we manipulate collections of data (lists of service objects) whereas the tests for the second case handles short strings (mileages).

Differences between the two test types More importantly, it is possible to tell that the gui tests are 150 to 1300 times faster than the unit tests. Also with the unit tests it is possible to run several tests in a single run very fast compared to the GUI tests. In order to test a lot of different inputs with GUI it would take some time but this can be achieved swiftly with the unit tests.

WHEN WILL THE TESTS RUN? DIFF OCCASIONS!

8.4 Code complexity

8.5 Defects

8.5.1 An odometer bug

During the work with the case study of the odometer, one defect was discovered in the refactoring process. Regardless of if the user answered 'yes' or 'no', the code would update the odometer when it was supposed to only update it on 'yes'. The GUI test doesn't discover this defect because in the scenario it went through, the user always answered 'yes'. One would have to create a new scenario to catch it.

This is an example where the white box nature of the unit tests makes a defect apparent since it was discovered while designing the tests. It wasn't the actual test which caught the bug but rather the process of reading the code carefully in order to do the refactoring. A code review or pair programming would probably have found the defect as well.

However, if the defect had had impact on the GUI by for example making certain controls unusable, the unit test would not have been able to capture that behavior.

8.5.2 Abilities to capture defects

This was the only defect recorded during the thesis work and because the system wasn't observed for a long time, it isn't possible to know at this point which method would find the most defects over a longer period of time.

8.5.3 Tests as safety net for refactoring

During the work of the thesis, the GUI tests were implemented first in both case studies. An interesting observation is that those GUI tests could then be run during the refactoring work to make sure that refactoring did not create any serious defects. The GUI tests were used as a safety net and created confidence about the code change.

8.6 Training

8.6.1 GUI tests

Getting started with White Framework and the GUI tests was not hard. The framework had a decent amount of documentation and an active user group

with forums for support. A lot of the issues which arise during the test creation can be solved with trial and error. Some issues required more investigation and research but were possible to solve without external help.

Because White runs from outside the application and the GUI tests are black box tests, it is not necessary to understand the inner workings of the OMS application to be able to start creating tests.

8.6.2 Unit tests

When creating unit tests, it necessary to understand how they fit into the structure of the existing code base. Refactoring the code was sometimes easy when, for example, extracting a method for readibility but sometimes complex when it comes to introducing software patterns for testability. Guidance by the thesis supervisor was required to be able to perform these refactorings.

8.6.3 Conclusion

While the work of creating GUI tests was possible to conduct independently from the development team and with only a basic understanding of programming, refactoring and creating unit tests requires a deeper knowledge of programming, in particular unit testing. If the development team does not possess this knowledge before hand they would require mentoring and training while acquiring it.

8.7 Lessons learned

8.7.1 Test environment

A big obstacle for the GUI tests was the state of the test environment against which the tests ran. For the thesis, conditional compilation was used to provide test data for the tests.

In another scenario, I would recommend to solve this issue in a more permanent manner. The conditional compilation works when used for a few scenarios but when the amount of tests grows, this will quickly become a new source of complexity.

This is has a high impact on the feasibility of creating a large suit of GUI tests. Without being able to set up the test environment properly, an automation effort could be very costly.

8.7.2 Gathering information

It wasn't possible to draw any conclusions about the amount of defects caught by the two methods. However, it is possible to reason about it based on the experience when creating the tests.

While the GUI tests give a broad confidence about the stability of the system, the unit tests provide confidence about the business logic inside of the GUI code. since the GUI tests start up the system and navigate through a set of views to

perform their tests, they indicate that the system is at least in such a shape that it is possible to run it and navigate through it. This information cannot be gathered from the unit tests since they do not start the system at all.

In the example with the odometer bug it was observed that in order to catch this bug with GUI tests, several test runs would be required. Each test run for the odometer takes 6 minutes. Adding a new scenario means doubling this time in the worst case. This means that if an extensive test suite to cover the OMS application is desired, it would take a long time to run. It is more likely that a carefully selection of scenarios would be made in order to keep the suite's size and run time under control.

Estimation of time it would take to cover the same amount of scenarios with gui tests?

8.7.3 GUI tests

The GUI tests were faster to

When the GUI tests handle complex user interface components, implementing tests takes longer both to create and to update when they change. These components also influence runtime of tests.

Perhaps something about reusing code to add tests and to do be able to work faster

"In both cases, some of the original test case can be reused for the updated or added test case. For example, in the first case it takes four steps before accessing the form that we wish to test. These four steps are not required to reproduce when writing other tests for that form. "

8.7.4 Unit tests

Unit tests are fast to run and because of their white box nature, can find defects in the functions that they test. A developer is able to run the unit tests in between small code changes without loosing much time, thus ensuring that the changes haven't broken anything.

However they require a refactoring effort before being put in place and this effort requires advanced training.

8.8 Risks

I know what happens and might optimize my code for that scenario. For example, the test cases for case 1.

9 Further work

- analysis of what kind of bugs to the methods catch - further analysis of maintainability of longer periods of time - introduction of the Page Object Pattern for more stable gui tests

10 Conclusion

A separate test environment for gui testing

With regards to a lot of the metrics that we have selected, the methods are uncomparable. Comparing automated GUI tests and unit tests does not make sense when it comes to code complexity, the amount of changes required and the amount or sort of defects that they capture.

There are costs involved in introducing any of the methods but those are placed in different areas. For GUI testing, the costs are mainly in setting up usable test data, the effort of creating the tests and the time it takes to run the tests.

For the unit tests, the cost are mainly in training.

Testing is the process of gathering information about a system with the intent to use it for some purpose, we can conclude that the different test methods gather different kinds of information at different costs.

References

[1] Gerald M. Weinberg, *Perfect Software: and other illusions about testing*, Dorset House, 2008

[2] James Bach, *Test Automation Snake Oil*, http://www.satisfice.com/articles/test_automation_snake_oil.pdf, 1999

[3] Cem Kaner, James Bach and Bret Pettichord, *Lessons Learned in Software Testing*, Wiley, 2002

[4] Atif M. Memon, *GUI Testing: Pitfalls and Process*, Computer, 2002

[5] Laurie Williams, Gunnar Kudrjavets and Nachiappan Nagapan *On the Effectiveness of Unit Test Automation at Microsoft*, Department of Computer Science, North Carolina State University

[6] Michael Feathers *Working Effectively with Legacy Code* 2004: Prentice Hall

[7] Martin Fowler *Refactoring - Improving the design of everyday code* 1999: Addison-Wesley Professional

[8] Birgit Geppert, Audris Mockus, and Frank Rler, *Refactoring for Changeability: A way to go?*, 2005

[9] Skoglund and Runeson *A Case Study on Regression Test Suite Maintenance in System Evolution* 2004

[10] *Project White* https://github.com/TestStack/White

[11] *NUnit* http://www.nunit.org/