

# Master Thesis

Ulrika Malmgren

May 3, 2015

## **Abstract**

This is where I'll write my abstract

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.1.1	About Nobina . . . . .	1
1.1.2	About the OMS system . . . . .	1
1.1.3	Releases and testing . . . . .	1
1.2	Problem statement . . . . .	1
<b>2</b>	<b>Testing</b>	<b>2</b>
2.1	Purpose . . . . .	2
2.2	Different levels of testing . . . . .	2
2.3	Prerequisites . . . . .	2
<b>3</b>	<b>GUI testing</b>	<b>2</b>
3.1	Introduction . . . . .	2
3.2	Advantages . . . . .	2
3.3	Drawbacks . . . . .	2
<b>4</b>	<b>Unit testing</b>	<b>2</b>
4.1	Introduction . . . . .	2
4.2	Advantages . . . . .	2
4.3	Drawbacks . . . . .	2
<b>5</b>	<b>Legacy code and refactoring</b>	<b>2</b>
5.1	Introduction . . . . .	2
5.2	Legacy code . . . . .	2
5.3	Refactoring . . . . .	3
5.3.1	Purpose . . . . .	3
5.3.2	Advantages . . . . .	3
<b>6</b>	<b>Approach</b>	<b>4</b>
6.1	Procedure . . . . .	4
6.2	Case selection . . . . .	5
6.2.1	Past releases . . . . .	5
6.2.2	Future releases . . . . .	5
6.3	Comparison criteria . . . . .	5
6.3.1	Time . . . . .	5
6.3.2	Changes . . . . .	6
6.3.3	Test runtime . . . . .	6
6.4	Frameworks . . . . .	6
6.4.1	Project White . . . . .	6
6.4.2	NUnit . . . . .	7

<b>7</b>	<b>Case studies</b>	<b>7</b>
7.1	Case 1: The Service Object List . . . . .	7
7.1.1	Method one - GUI test . . . . .	8
7.1.2	Method two - Unit tests . . . . .	9
7.2	Case 2: The Fuel Outtake and Odometer . . . . .	11
7.2.1	Method one - GUI test . . . . .	12
7.2.2	Method two . . . . .	12
<b>8</b>	<b>Results</b>	<b>13</b>
8.1	Time . . . . .	13
8.1.1	Data . . . . .	13
8.1.2	Analysis . . . . .	14
8.2	Effort . . . . .	15
8.2.1	GUI changes . . . . .	15
8.2.2	Code changes . . . . .	15
8.3	Test runtime . . . . .	15
8.3.1	Data . . . . .	15
8.3.2	Analysis . . . . .	16
8.4	Lessons learned . . . . .	16
8.4.1	Test environment . . . . .	16
8.4.2	Differences in scope between the two methods . . . . .	17
8.5	Risks . . . . .	17
<b>9</b>	<b>Further work</b>	<b>17</b>
<b>10</b>	<b>Conclusion</b>	<b>17</b>

# **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 really expensive and Nobina wishes to lower their testing costs and using automated testing instead.

## 2 Testing

### 2.1 Purpose

### 2.2 Different levels of testing

Differentiate manual testing vs exploratory testing?

#### 2.2.1 Regression testing

### 2.3 Prerequisites

What makes for a good test? Reproducible -> 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 Advantages

### 3.3 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. [?]

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

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 small units of functionality

### 4.2 Advantages

An automated unit testing practice meant a 20.9% decrease in test defects, and relative decrease of defects found by customers. [?]

Developers felt they spent less time fixing bugs found by testers. [?]

Raised awareness for error conditions and boundary cases. [?]

Understanding for code written by others. [?]

Harder for testers to find bugs [?]

### 4.3 Drawbacks

Success factors?

### 4.4 TDD

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

## 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 [1]. 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. [2]

### 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.

### 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.

(Why is it good to break dependencies, why is separation of concerns good? Do I need to add a part about design that increases testability, or do I keep it here?) In *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.

### 6.3.2 Changes

There are other ways of measuring effort as well. In *A Case study on Regression Test Suite Maintenance in System Evolution* [3], 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 [?] 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

### 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 [?]). However, this requires studying the system over a longer period of time to uncover those defects.

## 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

For this purpose, an open source Framework for .NET applications called White is used. 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. [4]

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 long as a button keeps the same automation id, it doesn't matter if it is moved to another place on the GUI.

SOURCE FOR THIS BEING GOOD!!!

### 6.4.2 NUnit

(For unit) Write about mocks/stubs?

## 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.

For 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.*

In the service object management window, all the service objects should be displayed

This is a new implementation of how the service objects are being presented and 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 isnt 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. Since this information is retrieved as a collection, a method is implemented to be able to create a collection with test data easily. With the conditional compilation we are able to either compile a normal version of the system, or one where our test data is used instead.

This is needed to get the GUI tests to work without a proper test database but we will see that it also good support for the tests written in method 2.

**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.

[Code snippet]

**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.

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 1 - 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 2 - 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 3 - the service object list is empty if there are no service objects

**Implementation** Here is 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 he 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. 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.

The basic test scenario is Adding a fuel outtake should be saved properly. The new scenario is Adding a fuel outtake when the odometer is about to roll over should trigger a dialog box.

**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, we either need to change the inventory number for the vehicle or the time of the outtake. Another issue concerns the case when we want to have a previous mileage that is larger than 900 000. In that case we would need to identify a vehicle with that mileage and use it in our 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, we have a limited amount of possibilities to test. This is solved by adding conditional compilation to the source code. When compiling the source for tests, we change the previous mileage 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

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 the source of the form there is a method

```
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
```

It has dependencies towards the GUI since it uses the information in some labels (lblReportFuelPreviousKm), the values of fields (txtKm.Text) and displays a dialog box. 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 separately.

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 parameters, it is also possible to test this functionality through unit tests.

## 8 Results

### 8.1 Time

Both the efficient time spent on developing the tests and the total time spent on development has been measured. The efficient time is only the time used for coding whereas the total time accounts for problems with the development environment, time spent discussing issues, and so on. In the tables below we can see the efficient and total times for developing the test cases before the anticipated change is rolled on and after it has been introduced.

#### 8.1.1 Data

	Base scenario		Updated scenario	
	Efficient	Total	Efficient	Total
GUI tests	17	39	4	13
Unit tests	4	13	1	3

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

	Base scenario		Updated scenario	
	Efficient	Total	Efficient	Total
GUI tests	6	20	2	11
Unit tests	6	21	2	12

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

In both cases, developing the GUI tests is more costly than doing the refactoring effort. In both cases, the before and after proportions are maintained when comparing the GUI tests to the unit tests. The original effort is four times as large as the effort for updating the tests in case 1 and three times as

large in case 2. The maintenance effort is in both cases smaller than the original cost for setting up the tests.

In the second case both GUI tests and unit tests take as much time both to set up but also to update according to the change introduced. In the other however, the GUI tests take four times as much time to set up and update compared to the unit tests.

### 8.1.2 Analysis

One explanation for the fact that the first scenario takes so much longer is that the GUI test for the Service object scenario was the first GUI test scenario implemented. It took time to learn how to access the different components used.

Also, the procedure for accessing the form to test is more complex in the Service Object scenario. Instead of opening it immediately from the Menu, it is required to go through a few steps before it is opened. The components in those steps are also more complex to work with in White than the ones in the fuel outtake form.

Another interesting observation is the fact that in the first case, creating a GUI test for a single test case takes almost as long as one complete work week when taking into the account the entire effort of setting up the test, that is the total time measured. In the second case, it takes a half work week. These are high numbers, but it is possible to stipulate that when creating new test cases for the forms used, the effort should be smaller. New test cases could mean trying different kinds of input in the fields that we have used (for example, different values for mileage in case 2).

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.

For the second method on the other hand, creating tests for a few different test cases takes about 2 work days. During this time we have produced tests that cover several different inputs. It is however harder in this case to estimate how much of the work done can be reused for other scenarios.

## 8.2 Effort

### 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.

## 8.3 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 tests from method 2 are 150 to 1300 times faster than the tests from method 1. 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 tests in method 2.

## 8.4 Lessons learned

### 8.4.1 Test environment

Confirms the studies in Testing - Prerequisites?

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. In some cases, it is certain that they won't work. Why?

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

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. In another scenario, I would recommend to solve this issue in a more permanent manner. The workarounds are explained in the different cases.

This is a difficulty for the GUI tests but it does not impact the unit tests.

### 8.4.2 Differences in scope between the two methods

Some reasoning about the fact that the unit tests will only test a small part of the code but also that the gui tests don't test what happens with an empty list, etc...

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

## 8.5 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 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

## References

- [1] Atif M. Memon, *GUI Testing: Pitfalls and Process*, Computer, 2002
- [2] Laurie Williams, Gunnar Kudrjavets and Nachiappan Nagapan *On the Effectiveness of Unit Test Automation at Microsoft*, Department of Computer Science, North Carolina State University
- [3] Michael Feathers *Working Effectively with Legacy Code* 2004: Prentice Hall
- [4] Martin Fowler *Refactoring - Improving the design of everyday code* 1999: Addison-Wesley Professional
- [5] Skoglund and Runeson *A Case Study on Regression Test Suite Maintenance in System Evolution* 2004
- [6] *Project White* <https://github.com/TestStack/White>