Master Project
Refactoring the Legacy Code

CARL QUIT. HE'S THE ONLY ONE WHO KNOWS HOW TO PROGRAM THE LEGACY SYSTEM.

IT CAN'T BE THAT HARD. GO FIGURE IT OUT.

EVU Master Thesis
Aarhuus University
2013 – 06 – 09

Student: Carsten Sørensen (201200089)
Teacher: Henrik B. Christensen
Abstraction

Over time, all code turns into legacy code, mainly because the original developers leave the project and are taking the knowledge they have with them. This can make it problematic when having to update the old parts of the systems or adding new futures that influences the old parts, because there are no knowledge about the structure or behavior.

In the book “Working Efficient with Legacy Code” by Michael C. Feathers, he describes some principles and techniques that he has used to work with legacy code.

The problem with legacy code is that it is code without any tests, and thereby no verification of the impact on the behavior when making changes. Michael C. Feathers guides the developer through different steps, that shows how to, find code areas that needs to be verified by tests, break some dependencies in a safe way, write the tests and finally refactoring the code in a safe tested environment that should verify that the behavior doesn’t change.

The principles and techniques in the book are made for efficient refactoring and are possible to apply on small parts of the code-base. This lets the developer take small steps in the process of working with the legacy code refactoring.

This report goes through the process of working with legacy code and discussing the gained knowledge, guided by Michael C. Feathers principles and techniques, by using the legacy code-base from the InnoScan’s CVT system.
# Table of Contents

Abstraction .................................................................................................................. 1

1 Motivation .................................................................................................................. 5

1.1 What is the CVT program ....................................................................................... 6

2 Problem statement ...................................................................................................... 7

2.1 Hypothesis ............................................................................................................... 7

2.2 Problem statement ................................................................................................. 7

2.3 Assumptions ........................................................................................................... 7

3 Method ....................................................................................................................... 8

4 Analyses ..................................................................................................................... 9

4.1 Current system design ............................................................................................ 9

4.1.1 Description of the classes .................................................................................. 10

4.1.2 Dependencies problems ................................................................................... 11

4.1.3 “I don’t understand the code” ......................................................................... 11

4.1.4 Current system design review .......................................................................... 14

4.2 Reason for change .................................................................................................. 15

4.2.1 Improve the design ............................................................................................ 15

4.2.2 Software modules has three functions ................................................................ 16

4.2.3 Ensure the behavior doesn’t change .................................................................. 16

4.3 Quality Attributes ................................................................................................ 17

4.3.1 QAS introduction ............................................................................................. 17

4.3.2 CVT QAS .......................................................................................................... 20

4.3.3 Michael C. Feathers focus on “QAS” ............................................................... 22

4.4 Structure less system design ................................................................................ 23

4.5 Why layered architecture ..................................................................................... 23

4.5.1 Presentation ....................................................................................................... 25

4.5.2 Domain ............................................................................................................. 25

4.5.3 Data Source ...................................................................................................... 26

4.6 Design patterns ...................................................................................................... 26

4.6.1 SRP - Single-Responsibility Principle .............................................................. 26

4.6.2 ISP – Interface Segregation Principle ............................................................... 26

4.7 Dependencies-breaking techniques ....................................................................... 27

4.7.1 Parameterized constructor .............................................................................. 27

4.7.2 Parameterize method ....................................................................................... 27

4.8 Legacy code change algorithm .............................................................................. 28
5  Refactoring .......................................................................................................................... 32
   5.1  Data source layer refactoring .......................................................................................... 32
      5.1.1  CCVTDatabase structure ......................................................................................... 32
      5.1.2  Identifying change points ......................................................................................... 33
      5.1.3  Find test points ........................................................................................................ 37
      5.1.4  Break dependencies .................................................................................................. 39
      5.1.5  Write tests (CCVTDatabase) ................................................................................... 40
      5.1.6  Make changes and refactor ....................................................................................... 43
      5.1.7  Evaluation of refactoring the Data source layer ......................................................... 45
   5.2  Domain layer refactoring ................................................................................................. 47
      5.2.1  Structure .................................................................................................................. 47
      5.2.2  Identifying change points ......................................................................................... 47
      5.2.3  Find test points ........................................................................................................ 48
      5.2.4  Break dependencies .................................................................................................. 48
      5.2.5  Write tests ................................................................................................................ 50
      5.2.6  Make changes and refactor ....................................................................................... 51
      5.2.7  Evaluation of refactoring the Domain layer .............................................................. 52
   5.3  Results of the refactoring process .................................................................................... 53
      5.3.1  Updated change algorithm ....................................................................................... 53
      5.3.2  Writing the tests for legacy code .............................................................................. 54
6  Discussions on legacy code refactoring ............................................................................. 56
   6.1.1  Anti-patterns vs Dependency-Breaking techniques ....................................................... 56
   6.1.2  Architectural Prototyping ........................................................................................... 57
7  Conclusion ............................................................................................................................... 60
8  References .............................................................................................................................. 61
9  Word list .................................................................................................................................. 62
10 Appendixes ............................................................................................................................ 63
    10.1  CVT user interface ......................................................................................................... 63
    10.2  CCVTDatabase grouping of methods .......................................................................... 64
    10.3  Preserve signature example ......................................................................................... 67
10.4  CCVTDatabase method: Open ................................................................. 68
10.5  CCVTDatabase split method: Open......................................................... 70
      10.5.1  Step 1. ............................................................................................. 70
      10.5.2  Step 2. ............................................................................................. 70
10.6  CCVTDatabase find place method Open is used....................................... 71
10.7  Tests for during refactoring ..................................................................... 72
      10.7.1  Tests for data source layer ............................................................... 72
      10.7.2  Tests for domain layer .................................................................... 72
10.8  IBatchPersistence interface .................................................................... 73
10.9  Domain layer – Batch Sequence update ................................................ 74
      10.9.1  BatchPersistence ............................................................................. 74
      10.9.2  CCVTDatabase .............................................................................. 74
      10.9.3  IBatchPersistence .......................................................................... 74
      10.9.4  BatchStateHandler ....................................................................... 74
10.10 Domain layer – Use of IBatchPersistence .............................................. 76
       10.10.1  BatchStateHandler ................................................................. 76
       10.10.2  BatchPage.xaml.cs ................................................................. 76
1 Motivation

InnoScan\textsuperscript{1} is a company that for more than 20 years, have been developing medical inspection machines, for vials and ampules. At the beginning, InnoScan only made the software and another company supplied the machines. This changed over the years and today the full production of a machine is located at InnoScan.

Over time, the machines design have changed from generation 3 to generation 5, and the software has evolved greatly in that time span. However, do to not having enough resources on the projects (Software developers) and deadlines, the software architecture suffered.

This has led to a design where everything in the code is tightly coupled and with strong dependencies. The structure that should have consisted of separated layers (Presentation, domain and data source) [M. Fowler] have been lost over time and a global class container was created, for faster solutions building and simpler access to general functions. A solution that was fast and simple to maintain, when there were only one developer working on the project and that one developer knew the entire system. This has changed and now there are four developers working on the same codebase, that has grown. Due to unstructured complexity, the code has and no unit tests of the functionality, it is difficult to predict the changes each programmer makes in the different parts of the code base, as have been seen when deploying new updates to the machines.

Today the system is only consisting of legacy code. Legacy code in the understanding that it is difficult to make changes, because there is no test suits to very that old features still works as expected.

\begin{quote}
\textbf{Michael C. Feathers – Legacy code}

\textit{Legacy code is not so much that good or bad code, but Legacy code is code without tests}
\end{quote}

The motivation for this project is to look at the aspects from Michael C. Feathers (MFC) principles in the book “Working effectively with Legacy code” [Michael C. F.] and try to apply the methods onto InnoScan’s legacy code, in an effort to try to refactor a more correct architecture in the project and for the developer team. Therefore, the code should end up, having less strong dependencies and be easy modifiable for future changes. The outcome is that some parts of the system is under test control, to validate that behavior of the functionality is the same after changes and to learn new methods to deal with legacy code.

The goal is not to fix the entire system, but to make a starting point, in trying to add test suits and remove some dependencies and to reflect on the methods on how to work with legacy code, in the way MCF recommends it.

At the end it should give a more clear view, on how InnoScan in the future could get rid of the global class and the legacy code by adding test suits and refactoring the current code in a safe way, so changes or new features to the system, don’t make unexpected changes to the behavior of the system.

\textsuperscript{1} InnoScan’s homepage can be viewed, for more information: \url{http://www.innoscan.dk}
1.1 What is the CVT program

The machines that are produced by InnoScan are called CVT machines, which stands for Computer Vision Technology machines. Example of a CVT machine can be seen in Figure 1.

![Figure 1 - CVT Inspection machine](image)

The CVT program is the software that is controlling the machine, from the operator’s point of view. The overall user interface (UI) is very complex, with a lot of sub-menu’s and functionality. One of the reason to get a cleaner separation between the UI and the rest of the code is to be able to change or update the UI fast and without any unforeseen consequences. A screen dump of the UI can be seen in appendix 10.1.

The CVT program gives the operators the possibility to control the production of batches\(^2\). Meaning that it gives the operators the possibility to create batches and create sub-batches, run test sets to verify the inspection quality are still functioning as expected and also allows to print old batch reports.

The CVT program also talks to the pharmaceutical companies own systems, when the machine is running on site. This could be to share alarms, counter values or overall efficiency information.

For InnoScan it is important to have a good architecture for the software, because customers are getting more and more demanding, in their requirements for future functionality and integration with other systems in their factories. Therefore InnoScan needs to be sure that new functionality or changes doesn’t make unpredictable behavioral changes.

\(^2\) A batch or Sub-Batch is an amount of production items that are inspected together. Therefore, if a problem is detected, it is possible to trace back what boxes needs to be taken off the marked.
2 Problem statement

2.1 Hypothesis

By critically applying the principles in Michael C. Feathers book “Working Effectively with Legacy Code” [Michael C. F.], is it possible to refactor the batch handling functionality to use a layer architecture, to remove tightly coupled dependencies and get a clear and correct separation between the presentation, domain and data source layers.

Secondly, are MFC’s techniques sufficient and comprehensive; or is there a need for adaption or novel techniques?

2.2 Problem statement

The software solution made by InnoScan has the problem that the code has strong dependencies. This is the case because of global instance of different classes are being used, for simpler access to functionality. In addition, most of the classes contains different responsibility’s, to be able to do updates in the UI and make computations without having to delegate objects around in constructors. This makes it difficult when modifying current behavior or adding new functionality, because the impacts to the current system can’t be foreseen.

The CVT project has become a solution with legacy code and needs to be handled with care, each time a modification is made. This can cause problems for updates requested by a customer, because there are no tests in place, to verify that the old behavior of the CVT program has not changed.

The system needs to be decoupled in its dependencies and unit test needs to be added, to ensure that changes don’t change behavior and avoid future legacy code state. In addition, new techniques about refactoring legacy code needs to be acquired, by using MFC’s book and reflecting on the guidelines he provides.

2.3 Assumptions

The report will focus on the batch handling functionality, by looking on how to separate the code in correct layers, so the UI does not use the database through a global container and to add unit tests to the functionalities, to ensure that batch behavior does not experience unpredicted changes after the refactoring.

The reason for choosing the batch handling area of the system is that it is the central logic for the customer’s functionality, to send production items on to the next step in there process in their factories. The system is also too extensive to remove all dependencies or to add tests for all functions.

The batch handling contains classes in the different layers, with strong dependencies. There for it represents a starting point for the beginning of the refactoring.
3 Method

By reading the book “Working Efficiently with Legacy Code” [Michael C. F.], it should gave a base knowledge on how to remove the strong dependencies in the code and get the current legacy code, under control in a test safe environment.

In addition, books about design patterns, will also looked at to get the correct design for decoupling the global dependencies in the code.

This will at the beginning be Martin Fowlers book “Patterns of enterprise application architecture” [M. Fowler], for guidelines on how a layered architecture should be.

A last book to cover some of the fundamentals would be “Agile Software Development” [Robert C. M.]. This book covers Principles, Patterns and Practices of these.
4 Analyses

The first step is to analyze the current system and to get an overview of how the architecture is in the current state. Look at the goals for the system and pitfalls to lookout for in the future. Other tings will be to look a tools and techniques on how to refactor and create a more correct design by using layered architecture and design patterns.

Many of the steps in this analyzing process will be from MCF’s book [Michael C.F]. Nevertheless, evaluating if the techniques are useable or other techniques are better suited.

4.1 Current system design

Before starting with analyzing the CVT program code by MFC techniques, a short introduction and overview of the current system design will be given.

The Unified Modeling Language (UML) class diagram below (Figure 2) shows a small part of the CVT program code base, related to the classes concerning the batch handling and some of its relations and the dependencies other classes have to it.

![Figure 2 - Simplified CVT Class diagram](image)
4.1.1 Description of the classes

**CBaseWindow:**
The main UI windows, which is the handler for all sub-pages shown in the UI. It is also partly the main logic, because most of the UI logic uses functionality from CBaseWindow object.

**CMainWindow:**
Is a subclass of CBaseWindow. Controls main logic of the CVT program.

**CBatchPage:**
Contains the UI of batch handling functionality. The page allows the user to create batches, start sub-batch inspections, close sub-batches, close batches and view previous batch reports.

CBatchPage uses CBatchStateHandler to handle batches, but also to get information for updating the UI buttons and operator text explanations, when the batch state changes by listening to events from CBatchStateHandler.

**CBatchStateHandler:**
Controls the state of the batches (This influences the entire machine state: Maintenance mode / Maintenance production mode / Production mode). CBatchStateHandler publishes events, when the state is changed.

**CBatchAction:**
Is a class consisting of static methods, too perform different batch operations, like create and close batches and sub-batches. It also stores the information about the current running batch. CBatchAction uses CCVTDdatabase class, through the Globals class that contains an instance of the CCVTDdatabase class, too save and get information regarding batches and sub-batches.

**Globals:**
Is a global container class, which contains references to all other major classes in the CVT system. In itself, it has very little functionality. However all classes uses it.

**CCVTDdatabase:**
The main purpose is to get and save persistence information in the database.

One big class that controls persistence of all information about Machine configuration, Product configuration, Batch information, user information and more.

**CBatchInformation:**
Data object, containing information about a batch or sub-batch.

---

3 The different states, limits the settings that can be changes by a user, to insure the quality of the product. Production mode is the most limited mode, where nothing can be changed to the machine setting.
4.1.2 Dependencies problems

There are many strong dependencies between the different classes in the class diagram above (Figure 2) and no clean separation of the different layers, regarding interfaces or abstractions.

Almost all classes in the entire system, uses CCVTDatabase directly through the public member variable within Globals, instead of using dependencies injection.

This can cause problems, when adding new features or changing old functionality, because it is impossible to predict what is depending on the classes that is being changed.

In addition, unit tests are very difficult to add, when there are no abstraction or interfaces. It is very difficult to make mock objects to limit the unit test to single methods in the objects.

4.1.3 “I don’t understand the code”

“I don’t understand the code well enough to change it” is a chapter in MCF’s book [Michael C. F.], which should have been read, before creating the class diagram above, because before starting this report, the understanding of the software architecture was limited.

What this chapter suggest is to start read the code and when things get complicated, the reader should take a pen and some paper, to note down the current information in mind. This should not necessarily be UML notations but simple sketching. However, the more complicated it gets, the more need for correct terminologies could be a help full thing.

This technique was actually the same used, to generate the class diagram and to understand the code better, for this report. However, instead of drawing on a paper, when the understanding got tough, a UML program (Visual Paradigm 10.0) [VisualP10] where used, while reading the code.

This where found to be a very helpful way to handle the class diagram, because it was possible to update and make changes, as a better understanding of the code was acquired. Nevertheless, this would be around the same as using a pen and a piece of paper.

The technique is simple and straightforward, when having to learn an unknown legacy code base.
4.1.3.1 Other techniques

In the book “Software architecture in practice” [Bass2003], they are using another technique, to obtain information about a legacy system. However, this technique goes into more details of the system.

The technique they are describing in chapter 10 (Reconstructing Software Architectures), have a few steps, that the user can follow, to get the overview of the system.

- Information extraction
- Database construction
- View fusion
- Reconstruction

**Information extraction:**
Analyzing the system, to find information about the relations between the different files, classes or methods. The information that is obtained in the analyzing part is placed in a table, as in the example below (Table 1). The information analyze could be done either with a profiler/analyzer program or manually by the user.

<table>
<thead>
<tr>
<th>Source element</th>
<th>Relation</th>
<th>Target element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateBatch</td>
<td>calls</td>
<td>SetStartUpStateParam</td>
<td>Function calls function</td>
</tr>
<tr>
<td>CloseBatch</td>
<td>calls</td>
<td>GetSqlDataAdapter</td>
<td>Function calls function</td>
</tr>
<tr>
<td>CloseBatch</td>
<td>creates</td>
<td>CBatchInformation</td>
<td>Function creates instance of class</td>
</tr>
</tbody>
</table>

Table 1 - Information extraction - Bass

**Database construction:**
After the placing of the information about the system in the table (Table 1), all information is stored in a database. This is either done by the tool gathering the information or manually.

The information should be stored in tables, where a link between the tables are made, with foreign key constrains.

Example of tables could be:

```sql
create table calls( caller text, callee text );
create table access( func text, variable text );
```

The reason for placing the information in the database, is so that it is possible to selections on different parts of the information.
**View fusion:**
In this step, the user has to compare the collected view information to be sure the extracted information by the tools are fully complete, and not missing any information. Some tools can’t detect all relations, and thereby a manual verification should be performed.

The relations in the database are than updated to include the correction information.

**Reconstruction:**
When all information is collected and stored in the database, tools for making reconstruction are used, to give the user a graphical view of the architecture.

4.1.3.2 Reconstruction tools

By using tools for reconstruct, it could ease the job for the developer that is trying to understand the system, by generating the diagrams or extracting the classes and methods.

However, it is not always the case, that the tools will make the process simpler. Some tools have the problems, that the miss to extract information or extract too much information, making it impossible to use practical [Koschke2005a]. Koschke also mentions that the reconstruction tools lose relations types: Aggregation, composite or others. Thereby the developer has to regain the relation type’s information manually and update the diagrams afterword’s.

By using an example from Visual Studio 2008 (VS), it is possible to see that the class diagram (Figure 3) extracted from the CVT project is not helping to give a quick overview.

![Visual Studio class diagram](image)

**Figure 3 - Visual Studio class diagram\(^4\)**

---

\(^4\) The figure is not scaled down, to show the full class diagram, generated by VS.
VS generates boxes, for all classes in the project, but as it can be seen, there are no relations between many of the classes. The developer still have to go through each class and find the relations between the classes.

4.1.4 Current system design review

By looking at the different ways to get information about the current system, the person needs to figure out, what the intention of information is.

Comparing the technique describe above (4.1.3.1) against the why MCF describes how to get an overview of the system, MCF’s technique by sketching the system is a simpler and more direct way to get a understanding overview, for refactoring. The user are sitting with the code and can note down on the paper at the same time, which keeps the learning about the system direct and easy to update. However, by using a class diagram tool, the information can be note correctly from the beginning and updated, without any confusing.

The technique from Bass [Bass2003] above seams more appropriated for creating documentation, for a system, that has no documentation, and not so much for fast refactoring. Thereby this would be a suitable thing to do after the refactoring is done. Therefore, the system architecture is saved for future use.

Using refactoring tools does not ensure that the person gets a better understanding. By removing the process of going through the code, to see the different dependencies, then the “learning by doing” information is lost. This could lead to a poorly understanding of the code, and in the end a more struggling refactoring process. Nevertheless, the tools could be fine, when having to generate system documentation.

The preferred way to get an understanding of the code is to use the techniques provided by MCF, because it gives a fast overview, which is simple to change/update if any new information is observed. However, with the small twist of using a sketching tool.
4.2 Reason for change

When working with a legacy code base, for a system, there are different reasons for a change in the code to take place. This can for simplicity be limited to four reasons:

- Adding a new feature
- Fixing a bug
- Improve the design (Refactoring)
- Optimizing resource usage

MCF [Michael C. F.] made a table showing what normally will change, when changing in the four aspects (Table 2).

<table>
<thead>
<tr>
<th></th>
<th>Adding a feature</th>
<th>Fixing a bug</th>
<th>Refactoring</th>
<th>Optimizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>Changes</td>
<td>Changes</td>
<td>Changes</td>
<td>-</td>
</tr>
<tr>
<td>New functionality</td>
<td>Changes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Functionality</td>
<td>-</td>
<td>Changes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Resources</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Changes</td>
</tr>
</tbody>
</table>

Table 2 - Where changes typically occur in software

By looking at MCF table (Table 2), we see that almost all changes (Adding a new feature, fixing a bug or refactoring), will have influences on the Structure of the code.

For InnoScan’s entire CVT program, this is a troubling thought. Most of the classes contains hidden dependencies; most of the classes are tightly coupled and are not cohesive.

This could in the future lead to problems, because there is no logical structure, and it can be very unpredictable what the changes will influence. It is important to have a good design/architecture for the system, to be able to ensure faster and more safe changes, when doing any of the four things listed above.

4.2.1 Improve the design

The reason for improving the CVT design is to make the code safer to add new features or fixing bugs.

It is not just to change the design as required to make it more correct. The goal is to refactor the design of the current system, to make a more correct and safe structure, and make it easier for new developers to understand the architecture faster.

Refactoring

The act of improving design without changing its behavior is called Refactoring. [Michael C. F.]
By doing refactoring, it is critical to be sure, that behavior of the program is not changed. This is where testing of the current legacy code is important, to try to insure that the behavior does not change.

Martin Fowler has the same aspect to refactoring as MCF, when looking on the fact, the both state the behavior must not change. A part of Martin Fowlers statement is: *Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior* [M. FowlerHP].

Freeman and Pryce [FreemanPryce] also states that refactoring is to make small changes to code, without changing the behavior.

This overall shows that more of the leading software developers have the same view on refactoring.

### 4.2.2 Software modules has three functions

Robert C. Martin [Robert C. M.] states that every software module has 3 functions. Where the first is the functionality the software module has, while executing.

The second function is that modules must be able to change, because all software changes over time. Therefor the developers are responsible for writing the software so it is easy to change.

The third function of a module is to communicate to its readers. If a module is too complicated to read, it will be difficult to understand and change the behavior if the requirements should change.

The second and third step are related to refactoring, and is the reason why refactoring is important.

Robert C. Martin and MCF are in the overall view, stating the same things, however in different ways. MCF is looking on the reason for changes, where Robert is looking on the reason to keep the software structure clean and simple, to make it simpler to implement changes and new features. Nevertheless, both states that refactoring should be guided by tests, to ensure the behavior of the system doesn’t change.

### 4.2.3 Ensure the behavior doesn’t change

The problem is that it is not always possible to write tests for the code, because it was not written for testing. Thereby it can be necessary to change the structure of the design, before tests can be written to ensure that the behavior of the program does not change. MCF calls this "breaking dependencies" (See 4.7 for information about dependencies breaking techniques).

It is critical to verify the behavior afterword’s by testing; this should both be unit tests of the classes that are changed and integration-/acceptant-tests.
4.3 Quality Attributes

Quality attributes are used to describe the qualities of the system, that the architect should use as guidelines, when designing the architecture or deploying the system.

Bass [Bass2003] gives 6 qualities of the system:

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability

Some Quality Attribute Scenarios (QAS) have to be defined, when designing the system, to insure that some guidelines are created for the goal of the architecture of the system. It is the stakeholders (ex. Costumer, Director, Developer or Tester) that are working on the system have to decide what qualities are most important for the system.

The CVT system has no QAS defined, and therefore the design of the architecture came from the ideas the developers that were working on the system in the beginning. Meaning that there was not stated any focus areas with QAS, that the system needed to follow in the architecture.

Nevertheless, because the goal is to refactor the system architecture, an important step is to define some QAS, to guide the architectural changes.

However, before continuing, a short introduction to QAS is required, to get a better understanding of their functionality (4.3.1). After the introduction, there will be created some QAS for the CVT system (4.3.2).

Finally a short look on MCF’s focus on what could be compared to QAS (4.3.3).

4.3.1 QAS introduction

Quality Attribute Scenarios works by having a Stimulus (input), that affects the system and thereby generating a Response (output). The goal of the QAS is to have a measurable response, so it is possible to verify it the QAS is full filed or not.

In depth, the QAS consist of 6 parts [Bass2003]:

\[
\text{The Legacy Code Dilemma} \\
\text{When we change code, we should have tests in place. To put tests in place, we often have to change code. [Michael C. F.]} \\
\]
<table>
<thead>
<tr>
<th>QA part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source of stimulus</td>
<td>This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.</td>
</tr>
<tr>
<td>Stimulus</td>
<td>The stimulus is a condition that needs to be considered when it arrives at a system.</td>
</tr>
<tr>
<td>Environment</td>
<td>The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.</td>
</tr>
<tr>
<td>Artifact</td>
<td>Some artifact is stimulated. This may be the whole system or some pieces of it.</td>
</tr>
<tr>
<td>Response</td>
<td>The response is the activity undertaken after the arrival of the stimulus.</td>
</tr>
<tr>
<td>Response measure</td>
<td>When the response occurs, it should be measurable in some fashion so that the requirement can be tested.</td>
</tr>
</tbody>
</table>

Table 3 - Quality Attributes parts

The flow/connection between 6 parts is best illustrated in Figure 4.

![Figure 4 - Quality Attributes parts flow](image)

Each of the 6 QA areas, have their own general scenarios, as an example can be seen for the QA Modifiability [Bass2003] in Table 4.
Portion of Scenario | Possible values
---|---
Source | End user, developer, system administrator
Stimulus | Wishes to add/delete/modify/vary functionality, quality attribute, capacity
Artifact | System user interface, platform, environment; system that interoperates with target system
Environment | At runtime, compile time, build time, design time
Response | Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
Response Measure | Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or quality attributes

Table 4 - QA Modifiability General Scenario Generation

An example of setting up a QAS for modifiability is illustrated in Figure 5.

![Figure 5 - QA example of modifiability scenario](image)

In the illustration example of the modifiability QAS, it is possible to see that the information is used from the table information from Table 4.

However, one of the important things to notice, are that the Response Measure, actually is a measurable value. This will help in testing that the QAS actually is achieved.

By using the general scenarios for each of the 6 QA’s areas, it is possible to generate some QAS for the CVT system, to guide the architecture of the refactoring. This will be done in the next step (4.3.2).
4.3.2 CVT QAS

The reason for making Quality attributes scenarios for the CVT system, before starting any refactoring, is to keep focus on what the goal is, and where to keep the focus.

It is also to create some QAS that InnoScan, can use in the future, to ensure that the systems development (Refactoring or adding new functionality) focus are on the qualities of the system.

The normal process in finding QAS of a system would be to have a meeting with the different stakeholders. However, this is not possible now, because there are too many projects ongoing at InnoScan at the moment. Therefore, there will be generated some QAS from the points of views, that has focus on the Legacy code problem and the architecture of the system.

The CVT program is today running in production at the different customers sites. Therefore the QAS that should have been created, when the first machines was made, are not necessary the ones needed for the system today. Because the requirements can change/develop over time, depending on the requirements from different actors.

Therefor the selected architectural drivers for the current CVT system are the following ones:

- Modifiability
  - It must be possible to change the presentation (UI), without influencing the functionality of the domain source.
  - It must be easy to add unit tests, for methods/classes in the CVT system.
- Testability
  - The batch handling must be under unit tests.
  - All unit test must be able to run fast. This is not a Performance QA; the focus is not on the code performance, but on the ability to test the code, without testing agents a database or an external service.
- Usability
  - The UI must be intuitive for an Operator of the machine. The Operator should be able to use the machine, with minimal knowledge of the Batch flow and background knowledge of the inspection process.

After defining the architectural drivers, that will be the focus for future features and refactoring, the next part is creating some QAS.
### QAS for Modifiability

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Changing the UI, without modifying the domain logic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Modifiability</td>
</tr>
<tr>
<td>Scenario Parts</td>
<td></td>
</tr>
<tr>
<td><strong>Source:</strong></td>
<td>Developer</td>
</tr>
<tr>
<td><strong>Stimulus:</strong></td>
<td>Wishes to modify the UI.</td>
</tr>
<tr>
<td><strong>Artifact:</strong></td>
<td>Code</td>
</tr>
<tr>
<td><strong>Environment:</strong></td>
<td>Design time</td>
</tr>
<tr>
<td><strong>Response:</strong></td>
<td>Modification is made, without any of the current unit tests are failing.</td>
</tr>
<tr>
<td><strong>Response Measure:</strong></td>
<td>Modification is made within 4 hours.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Easy to add unit test to methods in the CVT program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Modifiability</td>
</tr>
<tr>
<td>Scenario Parts</td>
<td></td>
</tr>
<tr>
<td><strong>Source:</strong></td>
<td>Developer</td>
</tr>
<tr>
<td><strong>Stimulus:</strong></td>
<td>Wishes to modify a class, so it is possible to add unit tests</td>
</tr>
<tr>
<td><strong>Artifact:</strong></td>
<td>Code</td>
</tr>
<tr>
<td><strong>Environment:</strong></td>
<td>Design time</td>
</tr>
<tr>
<td><strong>Response:</strong></td>
<td>Modification is made, without breaking any dependencies.</td>
</tr>
<tr>
<td><strong>Response Measure:</strong></td>
<td>Method is modified and a unit test is written within 7,5 hours.</td>
</tr>
</tbody>
</table>

### QAS for Testability

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Unit tests must cover the batch handling functionality.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Testability</td>
</tr>
<tr>
<td>Scenario Parts</td>
<td></td>
</tr>
<tr>
<td><strong>Source:</strong></td>
<td>Unit developer</td>
</tr>
<tr>
<td><strong>Stimulus:</strong></td>
<td>Perform unit tests to verify functionality</td>
</tr>
<tr>
<td><strong>Artifact:</strong></td>
<td>Functionality in the system</td>
</tr>
<tr>
<td><strong>Environment:</strong></td>
<td>At development time</td>
</tr>
<tr>
<td><strong>Response:</strong></td>
<td>Class is tested and verified that the functionality is still intact.</td>
</tr>
<tr>
<td><strong>Response Measure:</strong></td>
<td>Batch handling unit tests coverage of 90% within 1 month.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Unit tests must run fast.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Testability</td>
</tr>
<tr>
<td>Scenario Parts</td>
<td></td>
</tr>
<tr>
<td><strong>Source:</strong></td>
<td>Unit developer</td>
</tr>
<tr>
<td><strong>Stimulus:</strong></td>
<td>Perform unit tests to verify functionality</td>
</tr>
<tr>
<td><strong>Artifact:</strong></td>
<td>Functionality in the system</td>
</tr>
<tr>
<td><strong>Environment:</strong></td>
<td>At development time</td>
</tr>
<tr>
<td><strong>Response:</strong></td>
<td>Unit tests must be able to run, without interacting with external services or database connection.</td>
</tr>
<tr>
<td><strong>Response Measure:</strong></td>
<td>All system unit test must run within 2 seconds.</td>
</tr>
</tbody>
</table>
QAS for Usability

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Operators must be able to use the UI to operate the machine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Usability</td>
</tr>
<tr>
<td>Source:</td>
<td>End user (Operator)</td>
</tr>
<tr>
<td>Stimulus:</td>
<td>Wants to run a production batch</td>
</tr>
<tr>
<td>Artifact:</td>
<td>System</td>
</tr>
<tr>
<td>Environment:</td>
<td>At runtime</td>
</tr>
<tr>
<td>Response:</td>
<td>The end user is able to run a production batch.</td>
</tr>
<tr>
<td>Response Measure:</td>
<td>End user has run a production batch, without asking about help, to understand the UI more than 3 times.</td>
</tr>
</tbody>
</table>

The 5 QAS main focus is on having a structured architecture, that makes it possible to have a clean separation from the presentation (UI). In addition, to add tests for the methods in the classes, to insure the behavior stays intact.

4.3.3 Michael C. Feathers focus on “QAS”

When looking at MCF’s workflow for refactoring legacy code, he is not using any QA or QAS in the wording. However, his focus is on 2 of the QA, and in that why he is using QA’s to keep the focus on what has the be done and why.

The 2 QA that MCF uses, are Testability and Modifiability. Testability is the main QA of the guideline, when working with legacy code. It is important to get the code under test control, before doing any refactoring, if this is possible.

The other QA is modifiability, where the main reason for refactoring is to get a code, that has less strong dependencies and a more structured and correct architecture. This will help people working with the code, to better understand it and make modifications or add new features.

MCF should maybe include QAS in his workflow of working with legacy code. The reason for this, would be to insure that the refactoring, still have focus on the QAS that the actors have chosen for the system.

If a developer is refactoring a legacy system, where the QA focus is on performance, it could be structured with tingly-coupled classes and bigger classes, to avoid dynamic allocations like to generate objects from interfaces or a lot of performance costing invocations between the classes. Here it could be an incorrect solution, if the developer started splitting everything up, and creating a correct architecture with loss dependencies and runtime loading of DLL’s to ensure flexibility, because the developer thought it would be a nice to have feature.

The developer must be sure that the refactoring is in alignment with the goals of the systems QA’s.

Therefore, it would be good to add QA’s and QAS to MCF’s guidelines when working with legacy code.
4.4 Structure less system design

MFC mentions some reasons in his book [Michael C.F], why a system has a structure less design. The reasons can be because the system is so complex that there is no big picture to follow, or the team of developers are dealing with emergencies all the time, that they lose sight of the big picture, when trying to close the problems. Another thing is that the system has no architecture, so the developers just add or change features, as they want.

For InnoScans CVT system, the reason is that the original developer stopped, and a new one had to continue from that design. However, he properly did not have the same understanding of the systems architecture and there were no one to ask about it, so he just added features on top of what already was there.

From that point on, the problem is mostly because the team of developers are dealing with emergencies, which needed to be put out fast. However, also because the system did now follow any design structure, and no one have had the time to sit down and discuss about the future architecture or QAS (4.3). The lack of QAS (4.3) could also be one of the reason for the structure less architectural design. Because there were no goals for the system, except the functions that the developers was focusing on at the current moment.

The learning from the current structure less design can be important, so the developer can learn why the design has become structure less. This is to notice the causes and trying to avoid the same mistakes in the refactored code. However, even with a correct architecture, mistakes still show up, but by knowing the previous mistakes, it can limit the amount of mistakes in the future code.

This is the cause for refactoring, so that if the system that has gotten out of hand can be refactored to an architecture that is more commonly known by other developers. To have a more common ground for all developers, and avoid the term “Job security” code.

The most common structure used in software, is a layered architecture, which will be described in detail in the next section.

4.5 Why layered architecture

Layered architecture is used in many features in the daily life. This is not just limited to software development, but to all from hardware to building constructions.

The layered architecture helps by separating lower levels of information from higher levels, which has no need to know of the smaller implementations ways. This has two beneficial things.

- Separating the functionalities from the different parts of logic and thereby keeping the design simpler to understand.
- Making it more possible to substitute different parts, at different levels.

---

5 Job security code is when only the developer working on the code, can make any sense of it.
Separating functionality
This makes it possible for the different parts of the developer team to work on different levels. Some developers could work on the frontend and others could work on the persistence/network communication. As long as the interfaces between the different layers are well defined.

Making substitutes
To make substation solutions is a great advantage when writing tests, but also an efficient way to make use of different functionalities during runtime.

An example could be persistence of the data, that some would like to persist data in a database, where others would want to use xml files.

In Martin Fowlers book [M. Fowler], he describes 3 layers: “The three principle layers”. These 3 layers are split into Presentation (4.5.1), Domain (4.5.2) and Data source (4.5.3).

The flow of the communication between the lays are always from top to bottom (Presentation being the top and Data source being the bottom level), as shown in Figure 6.

This is the most usual way for the most correct and decoupled design. However, some uses a design where the Presentation layer knows the Data source layer directly, as shown in Figure 7. This can make the code simpler, because data can be fetch directly. Data binding is one of the things that could benefit from this, and for C#, this was very popular when using WinForms [WinForms].
In this report, the design that the refactoring will try to follow, is the normal separation of the layers (Figure 6). This is because it will give a simple overview of the architectural structure and other programmers will know exactly where the code is accessing the different levels. Meaning that the developers know that the domain layer only is accessed from the Presentation layer and the Data source is only accessed from the Domain.

4.5.1 Presentation

The Presentation layer handles the interfaces between the user interaction and the software. This can be tools like: HTML, WPF or other programming languages ways of interacting with the user.

The UI is one of the places that is most likely to change, during the system’s lifetime. This could have been a basic C# program, were the UI was built with WinForms during construction of the program. However as time has pass, a new UI framework was released for C# (WPF [WPF]). By having the correct separation and only UI logic in the presentation layer, it would be relatively easy to replace the Winforms with a new WPF design, because no changes would have to be made in the lower layers and no domain logic (business logic) had to be copied.

4.5.2 Domain

The Domain layer (Also known as the business logic layer), is the main logic of the system. The functionality is to handle system functionality when the user interacts with the UI.

The functionality could be things like make calculations, validate user input, save information with the correct persistence or send information over the network.

The domain layer is often the only layer that can’t be replaced in a program, because it contains all the logic of the system. However, the construction should still be in a correct way, so that sub parts of the domain could be replaceable without any major difficulties (No tight cobbled code). Sub parts could be things like algorithms that should be possible to change/swap, without the core behavior should change.
4.5.3 Data Source

The Data Source layers is responsible for communication with other systems that carry out tasks on behalf of the application.

This could be things like talking to a web-service, talking to a client/server application, file handling or database access.

To be able to change the Data source layer is an important thing to have in a system, even if it is not the layer, which would change so often. However, there could be a possible chance that a system would have to change database source or add more ways to persist data, like both files and database persistence. This could be because of customer requirements, or that new faster persistence storages where found to be more suitable for the solution (like using NoSQL persistence, instead of normal SQL databases).

Also for testing point of view, it is a great benefit to be able to swap the data source layer, with a fake data source object. It is important to test to correct data source layer, to be sure that it work as intended, however this is a very slow procedure and it would slow down the test runs. Therefor to speed up and make it easier to test Domain logic, the data source layer should be replaced with fake object.

4.6 Design patterns

During the process of refactoring the CVT code, known design patterns will be applied to the code, to generate a more loosely coupled and more coherent design.

The design patterns that will to be used during the refactoring are listed below, with a short explanation on what the design pattern is responsible for.

4.6.1 SRP - Single-Responsibility Principle

Every class should have a single responsibility: It should have a single purpose in the system, and there should be only one reason to change it.

4.6.2 ISP – Interface Segregation Principle

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use. ISP splits interfaces, which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
4.7 Dependencies-breaking techniques

When refactoring code, it is not always possible to write tests, for the code before doing any refactoring. If the code is written without TDD in mind, there are often a lot of hidden dependencies and other strong dependencies, which can make it more difficult to write tests, because the developer cloud have to make instances of additional classes, before being able to test only one class. It can therefore be necessary to break some dependencies before being able to write the test needed to verify that the behavior doesn’t change.

MCF gives some example of the techniques he has used with other teams, which solved the basic problems, they encountered during refactoring. The techniques focus on solving the problem, but without changing too much, so it doesn’t influence the behavior.

Below is listed a description of techniques that is used during refactoring of the CVT code.

4.7.1 Parameterized constructor

If a constructor creates a new instance of a class (this could be a persistence class), it can be necessary to remove the dependency, and place it as a parameter in the constructor.

By doing this, it is possible to keep the old constructor and add a new one, where the instance of the object is not created in the constructor. However, it set by the parameter send to the overloaded constructor.

However, this method can also be used, if the class uses a global container more places. Then that global container can be moved to a private member variable, where the value is set from the constructor. This requires a little more refactoring, but will still not change the behavior of the program.

4.7.2 Parameterize method

If a method is creating an object inside itself, it can be difficult to test. This could be a class that creates an internal object, that contains a remote connection to another service, and thereby makes it more difficult to test properly.

By using the parameterize method technique, the object is not created internally, but instead sent to the method as a parameter.

Example:

Before

```csharp
public bool IsBatchStarted (string BatchId)
{
    ...
    Database db = new Database();
    return db.IsBatchStarted(BatchId)
}
```
After

```csharp
public bool IsBatchStarted (string BatchId, Database db)
{
    ...
    return db.IsBatchStarted(BatchId)
}
```

Now it is possible to replace the Database object, when calling the method and thereby making it simpler to test the method by using a fake object instead.

### 4.8 Legacy code change algorithm

MCF [Michael C. F.] describes a change algorithm, with 5 steps that helps making the most safe refactoring of the legacy code in a system. Each step can be seen below.

The change algorithm goes through different steps; finding the places to change, find test points, break dependencies, writing tests and finally refactor the code. By following the steps, the developers have thought about the code to refactor in detail and put unit tests in place, thereby avoiding mistakes and broken functionality of the legacy code.

#### 4.8.1 Identify change points

Find places in the code that needs changes. To find the places, a technique like sketching the current system, with class diagrams or simple hand draw sketches can be efficient, to give an overview of the system. This will help understanding the current architecture and help to find the places to refactor.

#### 4.8.2 Find test points

Here the developer will find places in the code that needs to be under test control, too ensure that the behavior is still intact, after refactoring of the code. Effect sketches can help to do this. Effect sketches are simple drawings on paper, which shows the connections/usage between the variables, methods and classes in the architecture.

This is a critical step; to be sure, that the test covers all places the changes can have influence to the program behavior. In addition, to avoid writing test for all methods in a class, if many methods have to be refactored. It could make it faster to find pinch points in the code, which allows writing tests one place, to cover multiple methods.

*Pinch points*

> A pinch point is a place where tests against a couple of methods, can detect changes in many methods. [Michael C. F.]
4.8.3 Break dependencies

When refactoring a system, the purpose is to get the system under test control before doing any refactoring, to ensure that the functionality still works. However, sometimes there can be problems with writing tests, because the code is too tightly coupled.

So to get unit tests in place, before the actual refactoring, it can be necessary to break dependencies, before writing any tests.

In this step, the focus is to make sure that the code can be placed in a test harness. Therefore, it can be a requirement that some dependencies are broken beforehand, by using the Dependencies breaking techniques (4.7).

The developer that is breaking the dependencies has to be sure that no behavior is changed. Because there are no tests in place, and nothing to verify that the functionality is still correct.

MCF mentions some whys to be as sure as possible, that by making changes, behavior is not changed.

Hyperaware editing:

By having unit tests or doing pair programming, the developer will be more aware if changes have an influence on the system. The more the developer knows, if there are any changes, the better.

This is however a difficult task, regarding unit test, because the problem is to break the dependencies, to be able to add unit test.

Single goal:

Stick to one goal at the time. If one function is being refactored or a new functionality is added, sometimes the developers can lose sight of what they are doing, and thereby start making changes other places. Here the focus must stay on the current task, and the developers must note down, the other changes they wish to do later.

Preserve signature:

When copying methods too other places or to generate a static version of that method, it is not a good idea, to start generating new classes, and change the signature of the method call. Here the signature should stay the same of the method call.

An example can be seen in appendix 10.3.
Lean on the compiler:
In the Integrated development environments (IDE) that are available today, the developers can partly relay on the compiler, to detect the changes they have made. A functionality in VS gives the developer the possibility to rename a method, variable or class all places it is used, without having to search for it. The compiler does the job, by finding the signature where it is used. It is also possible to change a variable type, and compile the program, to see all the errors, where that variable is in violation with other types.

However, this is only valid for programming languages that have a compiler. Some script languages like Perl [Perl], is not able to precompile, and therefore the error will first show itself when the script is run.

Still with languages like C#, it is not possible just to trust the compiler completely. C# features Dictionaries, that is a key/value store, where the key can be anything from an integer to an string. If variable was used, the compiler would find spelling errors, but if a dictionary was used, the compiler would only detect type violations and not spelling errors.

So even if the compiler can help detecting errors, the developers still need to keep in mind, what he is changing, and know the limitations of the compiler.

Pair programming:
Pair programming is a good way to verify the changes that are being done and what impact they can have. The pair can also discuss the current design and help each other, when there are things they do not understand.

This will help insure the quality of the work the pair is doing.

4.8.4 Write tests
Do not write test on what the system is supposed to do, but what the system does now, these types of test are called Characterization tests.

The point is to be sure that the testing is focusing on the current behavior of the system, instead of what the system maybe should be doing. If tests are written for what the system should be doing, there is no guaranty to ensure that any behavior is not broken. However, by written tests for what the system is doing, it will be easy to see if behavior is broken, than the test should fail (The failsafe of having test, are only as good as the test them self).

Not to find bugs
We are not writing test to find bugs in legacy code, but to catch them, if we change the behavior that we did not expect to change. [Michael C.F.]
4.8.5 Make changes and refactor

Making the changes and refactoring the code, has two main goals. The first is to refactor the current design, to break tight dependencies and get a more correct structured code, the second is to add new features.

When changing and refactoring old legacy code, to correct the architecture, the developers just starts to make the changes, when the tests are in place. The changes should not change any current behavior, but only influence the code design.

When adding new features, the TTD method should be used, to ensure a correct way of added the new feature. This will help to keep the new code, from turning into legacy code.
5 Refactoring

In the refactoring part, the focus will be on refactoring the different classes, to obtain the architecture with 3 separated layers (Presentation, Domain and Data source).

The starting point of the refactoring will be the Data source layer (5.1), because it is the lowest level of the layer structure, and it will make it simpler to refactor the other layers, when there is a clean separation to the data source layer, so it is possible to make fake objects when writing tests.

After refactoring the Data source layer, the next point will be to refactor the Domain layer (5.2).

During the refactoring process, the point is to follow MCF’s legacy code change algorithm (4.8). This is to evaluate if the steps in the change algorithm is the right way to work with legacy code.

5.1 Data source layer refactoring

When starting to refactor the Data source layer, the goal is to get a clean separation, from the rest of the logic in the system. This is to make it easier to write tests and so the data source layer can be change, without having to find places in the domain/presentation layer, which uses SQL statements directly in the database.

The first step in this process will be to get an overview of the data source layers current structure. This is not part of the change algorithm, but more to get information about the classes and the structure.

5.1.1 CCVTDatabase structure

When looking a the CCVTDatabase class, the first thing that shows up, is that it is too big. It isent possible to show all Fields, Properties and public/private methods on one page. Therefore a short list will be shown in the table (Table 5) below, to give an example of the methods that is contained inside this class.
Totally, the class contains 111 methods (See appendix 10.2, for all methods).

The class contains all functions and information about the data source layer and persistence of all the information used in the system, related to the database.

### 5.1.2 Identifying change points

The techniques listed in Michael C. Feathers book chapter 16, where MFC writes about techniques to understand the code better (4.1.3) and chapter 17, where the focus is on reasons that a system loses its structure, so the developer can prevent the same things from happening (4.4). However, for identifying change points these are not the chapters that seams most suitable for this task.

The chapters are however giving some tools for helping the reader to understand the system and to give the reader a view of the current architecture of the system. Nevertheless, the chapter 16 (4.1.3) did lead to the current point of trying to loosen the dependencies to the data source layer.

After getting an overview of the structure and the size of the CCVTDatabase class (See point 5.1.1), chapter 20 “The class is too big and I do not want it to get any bigger”, will be the starting point, in trying to refactor the class. This is because the class is difficult to get a clear overview over, because of the many methods it contains.

One of the things, when the class is too big, is to see the responsibilities. What is the main purpose of the class and does it contain functionality that should be moved?
5.1.2.1 Seeing responsibility

When trying to refactor a big class, to ensure that it does not get any bigger, MCF have 7 experience-based techniques for trying to help the reader, to make the class smaller and avoiding that it will get bigger. This is mainly the design pattern Single-Responsibility Principle (SRP) that is the main topic of this chapter. For more information about SRP, see section 4.6.1.

Below each of the 7 heuristic given by MCF will be walked through and used if possible, to get an overview of the refactoring.

**Heuristic #1: Group Methods**

Grouping methods is about trying to find methods with similar names. By doing this, the methods with similar names, should group by different responsibilities.

It would take too much space, to list all the methods in the `CCVTDatabase` class. Therefore the table below is only showing, the methods that is relevant to the report. However, the list of the entire grouping of all methods are in appendix 10.2.

Here are the grouping of all methods with a batch related name (This table both contains batch and sub-batch methods):

<table>
<thead>
<tr>
<th>Batch:</th>
<th>Sub-Batch:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public void FireBatchEvent(EBatchEvent Reason, CBatchInformation info)</code></td>
<td><code>public bool CreateNewSubBatch(CBatchInformation BatchInformation)</code></td>
</tr>
<tr>
<td><code>public void OnFireBatchEvent(EBatchEvent Reason, CBatchInformation info)</code></td>
<td><code>public bool CloseSubBatch(CBatchInformation BatchInformation)</code></td>
</tr>
<tr>
<td><code>public bool UpdateTransportMode(String batchNumber, uint transportMode)</code></td>
<td><code>public bool GetSubBatchInformation(CBatchInformation BatchInformation)</code></td>
</tr>
<tr>
<td><code>public bool CreateNewBatch(CBatchInformation BatchInformation)</code></td>
<td><code>public bool GetSubBatchInformation(CBatchInformation BatchInformation, string RecipeNumber)</code></td>
</tr>
<tr>
<td><code>public int DeleteBatch(int BatchId)</code></td>
<td><code>public bool CloseBatches(CBatchInformation BatchInformation)</code></td>
</tr>
<tr>
<td><code>public int GetBatchId(CBatchInformation BatchInformation, string RecipeNumber)</code></td>
<td><code>public bool IsSubBatchNumberLegal(int BatchId, string SubBatchNumber)</code></td>
</tr>
<tr>
<td><code>public bool GetCurrentBatchInformation()</code></td>
<td></td>
</tr>
<tr>
<td><code>public bool GetBatchInformation(CBatchInformation BatchInformation)</code></td>
<td></td>
</tr>
<tr>
<td><code>public bool GetBatchInformation(CBatchInformation BatchInformation, string RecipeNumber)</code></td>
<td></td>
</tr>
<tr>
<td><code>public bool GetBatchInformation(CBatchInformation BatchInformation, string RecipeNumber)</code></td>
<td></td>
</tr>
<tr>
<td><code>public bool GetBatchInformation(CBatchInformation BatchInformation)</code></td>
<td></td>
</tr>
<tr>
<td><code>public bool CloseBatches(CBatchInformation BatchInformation)</code></td>
<td></td>
</tr>
<tr>
<td><code>public int[][] GetBatchCounters(DataRow dr)</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 6 - Grouping of batch methods from CCVTDatabase class
By using the Heuristic #1, it gives a quick overview, of a class’s responsibilities. This is one of the identified points that needs to be refactored, to get a better architecture for CCVTDatabase.

**Heuristic #2: Look at Hidden Methods**

In CCVTDatabase there are a few private method. However all functions are not related to the batch handling. Therefore, this point will not be used.

**Heuristic #3: Look for Decisions That Can Change**

When looking for decisions that can change in the CCVTDatabase class, some methods properly would change more than others would over timer.

An example of this would be the method Open or ExecuteNonQuery, because they are standard database methods, that only would change, if .NET would change how there ADO.net framework works. However a method like CreateNewBatch, would properly change as the requirements change for the CVT system.

Because this report only looks at the Batch handling functionality, the review of the general methods is not going to be in this point. Nevertheless, the batch handling needs some of the general methods, and it could be needed to refactor some of the methods in order to be able to refactor the batch functionality.

By looking at the methods, one method stands out because it properly would not change in the same cycle as the other methods.

This method is GetBatchCounters, because it is a method to convert the counters stored in the database to a double byte array. This should more be a helper method, that is used by the batch methods. The reason for this, is that it actually only uses other helper methods to do its work.

Below is the code from GetBatchCounters method:

```csharp
public int[][] GetBatchCounters(DataRow dr)
{
    if (DBNull.Value.Equals(dr["CounterLengths"]) ||
        DBNull.Value.Equals(dr["Counters"]))
    {
        return null;
    }

    int[] CounterLengths = CopyArray((byte[])dr["CounterLengths"]);
    int[] UintArray = CopyArray((byte[])dr["Counters"]);
    return CopyArray(UintArray, CounterLengths);
}
```

Table 7 - Method GetBatchCounters code
Heuristic #4: Look for internal Relationships

The CCVTDatabase class does not use so many internal variables, so heuristic number 4, will have no influence in this case.

The list below shows the internal variables of the class:

- private DbProviderFactory m_DataFactory;
- private DbConnection m_Connection;
- private bool m_IsInitialized = false;
- private ArrayList m_TableNames;
- private ArrayList m_ProductTableNames;
- private ArrayList m_MachineTableNames;
- private CModuleVersion m_DBVersion;
- private CBatchInformation m_CurrentBatchInfo;
- private EBatchIdLevels[] m_BatchIdFields;

The private members are only accessed from a few methods. The methods accessing the variables, are main not related to the batch functionality.

Heuristic #5: Look for the Primary Responsibility

The heuristic number 5 uses the Single Responsibility Principle design pattern (4.6.1). The goal here is to find the responsibilities of the CCVTDatabase class.

By looking at the class methods, it can be seen that there are more than one responsibility in this class. This could be compared to heuristic number 1, where the methods are grouped by name.

The fund functionality in the class are listed below:

- Communication with the database.
- Batch loading and saving.
- Sub-Batch loading and saving.
- Product configuration loading and saving.
- Machine configuration loading and saving.
- Other functions to manipulate DataTables.

This shows that the class needs to be split up, maybe even more, then just the listed responsibilities.
Heuristic #6: When all else fails, do some scratch refactoring

This heuristic will not be used, because the other heuristics have shown different responsibilities of the CCVTDatabase class. However, this heuristic could be used in a later point, when some basic refactoring has been done for the class.

Heuristic #7: Focus on the current work

This point will be a good reminder for the later work, when doing the actually refactoring. The starting point is to find the places where to refactor, to be able to keep an overview of what has to be done, and to know if the refactoring is going off the chosen path. This is limited to the CCVTDatabase class.

However, if the focus to refactor the entire class, this point is important to keep in mind, so only a small part of the program is refactored by the time.

5.1.2.2 Conclusion on finding changing points

By using the 7 heuristics shown above, it gave a clear picture of the responsibilities in the CCVTDatabase class and an overview of future refactoring points.

The class has many different responsibilities and is an indication that the class is getting to big, and needs to be refactored into smaller classes, that will be simpler to understand and maintain.

However, before any refactoring can begin, the class must be under test control, to insure the behavior is still the same before refactoring as it is after. First the places to add tests needs to be found. This will be the focus of the next point.

5.1.3 Find test points

Finding places that needs to be under test control should be a simple task. Because there should just be, tests surrounding all methods that will be change, by the refactoring. However, this is not that simple. By only testing the methods that will be changed, there could be an impact on other methods relaying on the refactored methods or parameters used by it.

The programmer needs to be aware of the side effects there are, when refactoring a method.

### Aware of side effects

Programmers needs to be aware of the side effects that could be outside of the method that is under refactoring

MCF uses a technique he calls Effect sketching, to find all the possible side effects. He uses a technique to find places in the code, that the method depends on or places that depends on the method that will be changed.
Effect sketches are simple drawings with the name of the methods or variables inside the code, where arrows are drawn between the names, indicating how they are using them. This will end up giving a map, showing the relations between the methods or variables, giving an overview of the parties that it can have an effect on. Moreover, what maybe needs to be under test control, before any refactoring.

In Figure 8 below, there is an example of an effect sketch from the CCVTDatabase class.

![Effect sketch of m_CurrentBatch example](image)

In the effect sketch around the variable `m_CurrentBatchInfo`, it is possible to see where the variable is used and what methods that could depend on changes to this variable.

It would be important to make sure, that the behavior stays intact. If any refactoring were to change the behavior that the system has now, it could have unpredictable side effects. Tests around the methods would help to ensure that the refactoring doesn’t change anything.

However, the technique (effect sketch) that MCF uses looks like it is very useful, when making changes to the behavior of the code. However, in this refactoring there should not
be any changes to the behavior of the methods. The refactoring should only move functionality to new sub-classes and break the dependencies, by using interfaces.

Effect sketching did however provide a fast and easy overview on what methods that could access the same variables.

For the CCVTDatabase class all methods that will be included in the refactoring will have to be under test control. The reason is to verify that all behavior in the methods are still the same, after moving the code to sub-class, because of the SRP design pattern (4.6.1).

5.1.4 Break dependencies

Before it is possible to get some classes under test control, it can be necessary to break some dependencies.

For the CCVTDatabase class, the refactoring goal is to make a more clean separation between the different layers, and to follow the goal of SRP (4.6.1) and ISP (4.6.2), therefore the breaking dependencies, will only be focusing on the parts related to these changes.

When looking at the class and the methods that will be under test control, there is only one method, which stands out with functionality, that is not related to the name, and that is the methods “Open”. See the method in appendix 10.4.

The problem with the “Open” method is that it has more responsibility, then just opening the connection to the database. It also collects batch information, if a batch was active before the system was closed.

Breaking the methods dependencies

The process is to split the functionality of the “Open” method into two methods. The “Open” method will only keep the code, to connect to the database and the new method “Initialize” will contain the rest of the functionality. See the updated code example in appendix 10.5.1.

In this process, the refactoring is done manually. Because of this, the compiler must be used, to verify that the code is still valid (MFC uses the technique of trusting the compiler in this section 4.8.3). This is to be sure that no local scoop variables are being used, in the new method, but haven’t been copied. As it can be seen in appendix 10.5.2, the variable “MinDBVersion” which is a method parameter, is not located in the new method “Initialize”.

The method parameter “MinDBVersion” is moved from the “Open” method to the “Initialize” method.

Then all places where the “Open” method is used, must be updated, to call both the “Open” and the “Initialize” method, to ensure the functionality is still the same. See appendix 10.6.

The class CCVTDatabase has evolved in 3 steps, see the Figure 9 below.
Now that it is possible to connect to the database, without having to take care of the current batch state, it is possible to write tests for the methods. The next step (5.1.5) will be to write tests, for the class.

### 5.1.5 Write tests (CCVTDatabase)

MFC’s states that tests should not be written for a database or an external service, because they take too long time to run. Because the CCVTDatabase class contains a lot of internal logic, it would be unsafe to refactor the class without having test. However, as stated in Freeman and Pryce’s book [FreemanPryce] it is also a good idea to have tests in place, for the real database, to insure that the query’s works as expected. However, the tests should not be included in the daily test runs. The tests should run when bigger changes occurred in the system, which could influence the database layer or at some defined intervals, to validate that the behavior is still intact.

**Setup the default database state:**

When writing tests for the data source layer (Persistence layer in Freeman and Pryce’s book), Freeman and Pryce [FreemanPryce] states that it’s important to have a known state of the database before starting the tests.

The database should be cleared or filled with defined dummy data, before running the tests, so it is possible to predict the expected test results.

They also state that the cleaning should be done before starting the tests, and not after. There are two reasons for having the cleaning before the tests and not after.

1. To be sure that the database is always in the right state. If a test fails or it is aborted, it could lead to data not being deleted, and thereby an inconsistent database state for the next test run.

![Figure 9 - CCVTDatabase - Breaking "Open" method steps](image)
2. To be able to troubleshoot if the tests fails. If the data were cleared after each test, it would not be possible to see what was actually inserted into the database. However, if the data were clear at startup, it would be possible to look at the stored data, when the tests were finished.

The developer must remember that this must be performed manually. Setting up the connection and writing manual SQL queries, to put the database in the wanted state.

This technique will be used in the tests, which will be written for the database, to ensure at correct state, for each test run.

5.1.5.1 Write the tests

The first test for the Batch handling will be for the method “CreateNewBatch”. The first step is just to see if it is possible, to run the method without any parameters, but compiler quickly tells the developer, that the method takes one parameter (See Figure 10).

![Figure 10 - CreateNewBatch parameters](image)

The method to create a batch, takes a “CBatchInformation” object as a parameter. However, that is not the only problem.

When writing the tests, some hidden dependencies starts to show up. One example is that some parameters in a global variable have not been set.

```csharp
if (Globals.CVTUnits["HMI"]).GetConfigBool(EConfigType.Machine, "Batch|Setup|ResetInspectionStatusAtBatchStart", false)
{
  Globals.InspectionInfo.RestoreToRecipe();
}
```

![Table 8 - CCVTDatabase writting test - exception](image)

The code in Table 8 looks at a configuration parameter, and if does not exists, it returns a default value. However, the problem is, that the variable “CVTUnits” is null and therefore throws an exception. This dependency should have been broken in the previous step (5.1.4), but because it where not seen, before starting to write the test, it can first be handled here.
There are more ways to handle a problem like this; one way is to check for null, and thereby skip the code if the "CVTUnits" is not initialized. Another and more correct way would be to use the technique MCF calls Parameterize Method (4.7.2).

By using the parameterize method technique, the code will change in 3 places.

1. The method signature for the copied method

```csharp
public bool CreateNewBatch(CBatchInformation cbi, bool ResetInspecStatusBatchStart)
{
    ...
}
```

Table 9 - CreateNewBatch parameterize method

2. The body where the old value was used

```csharp
if (ResetInspecStatusBatchStart)
{
    Globals.InspectionInfo.RestoreToRecipe();
}
```

Table 10 - CreateNewBatch body update

3. The body of the old method must be changed to call the new overloaded method

```csharp
public bool CreateNewBatch(CBatchInformation cbi)
{
    bool risbs =
    Globals.CVUnits["HMI"].GetConfigBool(EConfigType.Machine,
    "Batch|Setup|ResetInspectionStatusAtBatchStart", false)

    CreateNewBatch(cbi, risbs);
}
```

Table 11 - CreateNewBatch call new method from old method

In this way, it is possible to test both the method fully, with both true and false as values and because the method is overloaded, the systems current functionality is still intact.

The thing about writing test for legacy code is that it gives a more clear insight into what is actually going on in the code. When writing tests for the CloseBatch method, it was seen that if no batch id was found to close, the method returned a True value. By doing this, the developer would always think that the functionality is working, even if a batch is actually never closed, because there is not batch with that id. This is functionality that needs to be noted down, and talked about in with the other developers, to agree if the solution is correct.
The tests that is created are the following:

- TestMethodCreateNewBatchCorrect
- TestMethodCreateNewBatchFailNoRecipe
- TestMethodCreateNewBatchFailRecipeNotReleased
- TestMethodCreateNewBatchFailBatchExist

All unit test can be seen in the appendix 10.7.

Now that the tests are in place, the refactoring of the code can begin, without having to fear that, the functionality is broken (5.1.4).

5.1.6 Make changes and refactor

The changes that that will be made to the class, is to create a clean separation to the classes above, that is using the data source layer and to use SRP (4.6.1) / ISP (4.6.2). The reason why the SRP will be used, is to give clear overview of the functionality, but also later on it would be easier, to decouple the batch functionality from the entire CCVTDatabase class. The ISP will create an interface, which will make it simpler to make fake objects when writing tests.

5.1.6.1 Clean separation

The goal of giving the class a clean separation interface, for the batch functionality, is so that when refactoring the domain logic, the interface can be used, instead of the direct dependency on the actual class. Here the ISP is used (4.6.2).

This refactoring will not influence any behavior of the class. The interface can be seen in appendix 10.8. The interface is created from the grouping of the methods of the CCVTDatabase class, see appendix 10.9.3.

The only modification to the CCVTDatabase class is the line of code below:

```csharp
public class CCVTDatabase : IBatchPersistant
```

The interface can be implemented without any change, because it uses the same method signatures as the class does. This also fulfills the SRP for the interface part.

In C# a class can implement as many interfaces as needed, and thereby it is possible to use the ISP, to split the functionality into different interfaces. However, for other languages this is not always possible.
5.1.6.2 Batch Persistence class

To have the CCVTDatabase class to follow the SRP (4.6.1), the goal is to move the batch functionality to its own class and just forward, the method calls. The first method that is refactored to the BatchPersistence class, is the CreateNewBatch.

When starting to refactor the method, problems started to show up, when running the tests. In the Figure 11 below, the test fails when trying to run them. This shows, that the change made, was not done correct and shows just how important it is to have the test in place.

The reason the test fails, is that there is no “Try-Catch” around the part, that selects the recipe information, but before the refactoring, this was within a “Try-Catch”. See the two illustrations of code examples below.

Illustration code before:

```csharp
void Method()
{
    try
    {
        recipe = GetRecipeInformation();
        SQL_CreateTheNewBatch(BatchId, recipe);
    } catch() {...}
}
```
Illustration code after:

```csharp
void Method()
{
    recipe = GetRecipeInformation();
    BatchPersistenceClass.CreateNewBatch(BatchId, recipe);
}

void BatchPersistenceClass.CreateNewBatch(BatchId, Recipe)
{
    try
    {
        SQL_CreateTheNewBatch(BatchId, Recipe);
    } catch() {…}
}
```

The reason is that one of the tests is trying with a recipe id that is `Null`. This will result in an exception, instead of returning `False`. Normally a developer could argue that this is more correct. However, is not the meaning to change the behavior, just to make sure that the behavior is not changed, so if it returned `False` before, it should also return `False` after.

The solution is to place a “Try-Catch” around the collection of the Recipe information and return `False` in the `Catch` statement.

After placing all `CCVTDatabase` methods regarding Batch Handling in its own class, and just delegating the method calls from the `CCVTDatabase` to `BatchPersistence`, the `CCVTDatabase` class is one-step closer to follow the SRP (4.6.1).

In the next step, the evaluation of the refactoring will be discussed and the class diagram changes will be shown.

5.1.7 Evaluation of refactoring the Data source layer

The process of refactoring the `CCVTDatabase` class was a good experience, which showed some new things that is normally not seen, when just adding new features or changing some old behavior, to tweak it.

The most difficult thing by doing the refactoring was to write the tests for the legacy code. The task in writing a test for some code, which was not meant for tests shows up, when there is too much behavior in one method and hidden dependencies. One of MCF’s suggestions is to initialize the methods without any valid parameters, just to see what happens. By doing this, the developer will learn about what the method expects and the behavior if it is used wrong.
MCF uses the breaking dependencies techniques (4.7) like Parameterized method, which is a part of this refactoring. The techniques are very helpful and very important, to be able to put the methods under test control.

Another thing when breaking dependencies, writing tests and refactoring, is actually all the small structural errors that shows up. Here it is important to keep focus on the current task that is being worked on, as MFC suggests. However, it is good to note that the changes that is seen on the way, so it can be changed later on.

The final refactoring did not change so much to the overall class diagram. A new class called BatchPersistence is added and the interface IBatchPersistence was added to the CCVTDatabase class. This can be seen in Figure 12.

![Figure 12 - Class diagram after refactoring CCVTDatabase](image)

BatchPersistence class contains all the methods from the grouping task (See appendix 10.2), that is related to the Batch Handling and also is located in the IBatchPersistence interface (10.8).

The CCVTDatabase class showed that there was much more to refactoring to be done in the future. Nevertheless, the main refactoring was to refactor the class, to give a better layered separation. This will be used when making refactoring to the domain layer in the next part (5.2).
5.2 Domain layer refactoring

Now that the data source layer have been refactored and is under test control, the refactoring process can move on to the domain layer.

Here the focus will be to follow the same guidelines in the legacy code change algorithm, but on the domain layer, where the goal is to refactor the code implement the IBatchPersistence interface and to put it under test control.

5.2.1 Structure

The current structure of the domain layer shows the use of the Globals class and still no use of the new IBatchPersistence interface.

The current structure has high dependencies on the different class and it is not possible to test the batch functionality, without the actual connection to the database.

The structure of the current architecture can be seen in Figure 13 below.

![Figure 13 - Domain layer - Batch handling structure before](image)

5.2.2 Identifying change points

The changing points for the refactoring of the Domain layer, will be to use the new IBatchPersistence interface.

The actual batch handling is split into CBatchStateHandler and BatchActions, and thereby the functionality is correctly separated from the other classes in the system.

This was actually done a short while ago, due to refactoring the batch system, to become more configurable. However, this was done without unit tests and instead with the normal trial and error technique. Two weeks after the refactoring, errors still showed up, but were found by using manually acceptance tests.
5.2.3 Find test points

For finding test points, the two classes will be looked at separately and finally an overview of where the tests will be written.

**BatchStateHandler:**

When reviewing the `BatchStateHandler`, the general properties that are available are for the UI, which are used to update the text on the buttons and other text information for the user. Examples of the properties can be seen in the list below:

- `public string ActionText`
- `public string LeftText`
- `public string RightText`

Other methods like `ButtonPressed`, `LeftButton` and `RightBtn`, are also used by the UI, when the user clicks on the different buttons on the CVT programs UI.

The basic of these properties and methods is that they return a value from the property `CurrentState`. In general, all methods use the `CurrentState` property. So the test points will be related to the public methods of the `BatchStateHandler`.

**BatchActions:**

This class only consists of static methods and is used by other classes to control the batch functionality.

The place that could be influenced by other classes is the property `BatchInfo`, because this property controls the information about the current batch that is being used. However, the point to notice is that `BatchAction` can be fully tested by `BatchStateHandler`, because it is used during the different batch state steps (Depending on the `batch state map` setup).

The `batch state map` can be seen in appendix XXXX. This is an example setup of the different steps in a batch sequence and the buttons that should be enabled or visible on the UI.

**Test points:**

The test points will be around the `BatchStateHandler` class, which then will test the `BatchActions` methods also. Thereby this can limit the unit tests that needs to be written. This is an example of a pinch point, where it is possible to write tests for one class, but it is in the end, testing two classes.

5.2.4 Break dependencies

Again when starting to look at the class, some problems shows up instantly, that will make it difficult to write tests, and it just shows that the TDD methodology wasn’t used, when writing the class.
The **BatchStateHandler** constructor is the main starting point, and here is the first big problem. The constructor gets information about the batch state directly from the database, as it can be seen below in Table 12.

```csharp
DbDataAdapter da = Globals.Database.GetSqlDataAdapter("SELECT * FROM BatchSequence");
System.Data.DataTable dt = Globals.Database.GetDataTable(da, "Tables");

foreach (System.Data.DataRow reader in dt.Rows) {
    ...
}
```

**Table 12 - BatchStateHandler constructor**

The information located in the database table **BatchSequence** is selected and for each row, the information is processed and the batch state map, is created.

The problem is that this information should be returned from the data source layer, related to a method call, and not directly, SQL calls. At the moment it will be very difficult to write tests, for this class, because the **BatchSequence** table needs to be update with new information, to test different mappings.

Here the goal will be to move the call down into the database, into the **BatchPersistence** class and also add it to the **IBatchPersistence** interface. Update the code, to use this function call instead.

This will make it possible to make a fake object that implements the **IBatchPersistence** interface and will make the writing of tests much simpler and make the tests run faster, because they do not need to have connection to the database.

Another problem that shows up is that **BatchStateHandler** has directly contact with the UI. The class calls the method **Globals.BaseWindow.ShowPage(...)**. **BaseWindow** is the main window in the UI.

This gives a bigger problem when writing test, because unit tests, can’t verify the UI changes. This will not be part of the refactoring, and can be avoided in the tests that will be written, by not setting any “GoToPage” information in the batch state map. Nevertheless, it is a very important part to notice, when writing tests for legacy code, which the developer has to take notice of all calls to the different parts of the code. Like that, **BatchStateHandler** that makes direct calls into the database and the UI.

**Breaking dependencies to the database:**

The thing to notice is that the selection of **BatchSequence** is handled in a loop, where the data is stored in a “DataRow”. By using the same functionality, this can be replaced with a “Dictionary” in C#, which holds the same functionality, because a string key is used, instead of the index position.

So the update code, were the dependencies are broken, can be seen in appendix 10.9.
Now that the selection of the batch sequence is selected from the data source layer, the second breaking of dependencies, is to remove the use of “Globals.Database”, and replace it with an object implementing the IBatchPersistence interface.

The changes can be seen in appendix 10.10.

Know that it is possible to make a fake object that implements the IBatchPersistence and parse it to BatchStateHandler, it is now possible to start writing tests, that doesn’t use the database directly and thereby follow MCF’s advice about not writing tests, that uses the database.

5.2.5 Write tests

In the beginning, the first two tests where simple to get started. Some small debugging was needed, to find where a few Null references occurred.

However, not all methods were as easily to test. Because of the using of the Global class. It is very difficult to find all places that contains strong dependencies. One of the examples was when trying to test the method “LeftButton” in BatchStateHandler. As it can be seen in the sequence diagram below (Figure 14), the method calls BatchActions, that uses Globals.User.OperationMode. This fails, because it is Null and not initialized.

The problem is that even if the code is analyzed before writing tests, there could still be hiding some dependencies, which needs to be broken, while the tests are being written. In this case, the problem is that there is no object instance of the class CUser. The place where instance of the CUser class is being used, can be seen in the code example below in Table 13.

```csharp
public static bool RunInFunctionalTestBranch {
    get {
    }
}
```

Table 13 - CUser – RunInFunctionalTestBranch
By changing the code to return `False` if the `User` object is `Null`, would solve the problem. This will not change any functionality in the program, because the system cannot function properly without initialized instance of the `CUser` class, because each action requires a user logged in, on the CVT Machines.

The following test was written for the `BatchStateHandler` class:
- `TestMethodConstructorNoBatchSequence`
- `TestMethodChangeState`
- `TestMethodLeftBtnClick`
- `TestMethodRightBtnClick`

The tests does not cover all functionality of the class, however it covers most of the basic, that is related to the batch handling that will be refactored.

The tests can be seen appendix 10.7.

5.2.6 Make changes and refactoring

Some of the refactoring for the `Domain layer` was done in the part of breaking dependencies (See point 5.2.4), but there are still some functionality left that needs refactoring.

`BatchActions` class uses the database through the `Global's class`, to handle the batch behavior, and this needs to be changed, to use the new `IBatchPersistence interface`.

First, by adding the batch persistence to the `BatchAction` class:

```csharp
private static IBatchPersistance _BatchPersistance;
public static void SetBatchPersistence(IBatchPersistance batchPersistence)
{
    _BatchPersistance = batchPersistence;
}
```

The method `SetBatchPersistence` will be used from `BatchStateHandler` in the constructor, by using parameterized constructor dependency-breaking technique (4.7.1):

```csharp
public CBatchStateHandler(IBatchPersistance batchPersistence)
{
    _BatchPersistence = batchPersistence;
    BatchActions.SetBatchPersistence(_BatchPersistence);
    ...
    ...
}
```

Then all places that the `Globals.Database` is used related to Batch functionality must be updated to use `BatchActions` internal `_BatchPersistance` member. An example can be seen below.

Before:
if (Globals.Database.CreateNewBatch(cbi))

After:

if (_BatchPersistance.CreateNewBatch(cbi))

This is the final refactoring done to the Domain layer, regarding the batch functionality, to get a more clean separation between the different layers. Unfortunately this where a more time-consuming task, than first anticipated. Therefore, there will not be any more time for refactoring.

5.2.7 Evaluation of refactoring the Domain layer

After moving up a level to refactor the domain layer, it is seen that it is more difficult to make bigger refactoring’s. The separation to the database layer, should properly have been done, for the entire Data source layer, before moving up a level.

However, one of the more noticeable tings, when following the change algorithm (4.8), was that even if the developer tries to sketch down the current system design, some information could still be missed, and this can make it a more complicated process, when writing the tests. The reason is that the developer has to go back a step, and break dependencies again, before continuing to write the tests. Therefore, there is an interruption while writing the tests and this could get the developer to lose focus on the task he is working on.

Looking at the bigger picture, the implementation of the batch persistence, into the domain layer was a success. It can be seen in the updated class diagram below, that a more clear and simpler to understand architecture is starting to show.

![Figure 15 - Final Class diagram](image-url)
The architecture in the class diagram, now follows a more top-down approach, but there are still dependencies to the *Global* class.

The legacy code change algorithm (4.8) did however supply a good way of going through the steps need to refactor the domain layer.

5.3 Results of the refactoring process

The *Legacy code change algorithm* (4.8) from MCF, was a great starting point for the refactoring process. However, there are some things, which could need some changes.

When looking at the processes of refactoring the data access layer and the domain layer, the places that were giving the most problems, where when writing tests. Here it was like that the same thing happened repeatedly, mainly that hidden dependencies kept on showing up. This should have been fixed in the step of breaking dependencies, but this was not done enough.

The problem seems to be that the code is not analyzed thoroughly enough. However, it is possible to analyze the code so much, that all collected information are too comprehensive and it would be impossible to remember it all, when having to break dependencies. The idea of having to do too much and keep all information in mind, would properly lead to mistakes.

5.3.1 Updated change algorithm

Instead, the solution would be to update the *legacy code change algorithm*, in the way that it is used. Here some things needs to be added, and some steps needs to be repeated in loop, until it is ready for the next step. This updated version can be seen in Figure 16 below.

![Figure 16 - Update legacy code change algorithm](image-url)
Description of the steps in the updated legacy code change algorithm:

1. **Understand the structure**: Is to get an overview of the place in the code, where the update should take place. This would either be a full class diagram, to find a more specific place to look at, or to go in depth with a point the developer already knows need to be changed.

2. **Find specific change points**: Here is will be to note down the specific refactoring places, to define the places that will be the focus points. This could be to make a small list and take each point, one by one.

3. **Find test places**: Look through the code and follow the guidelines from MCF to find places, where the tests should cover, to ensure the behavior of the code (Example by using effect sketching).

4. **Breaking-dependencies**: Here the techniques MCF describes to break dependencies are to be used. The dependencies that are found should be broken with care, to insure that no behavior is changed.

5. **Write tests**: Write tests for the methods that was defined in “Find test places”. However, if hidden dependencies shows up or other dependencies needs to be broken, the developer should go back to step 4, and break the found dependency. This loop should continue as longs as there are dependencies in the way of writing tests.

6. **Refactor code**: Follow MCF’s guide to refactor the code.

During the refactoring process with the data source layer and domain layer, the updated algorithm above was used without noticing it. Because when encountering a problem while writing tests, there was an automatic step back, to break the dependency before continuing with writing the test.

### 5.3.2 Writing the tests for legacy code

MCF guidelines for writing test’s for legacy code, didn’t help so much, and the reason could be that each code can be written very different, from developer to developer. This will make it more difficult to make one statement that could be used for all code examples.

However, some guidelines for how a test should be and what to focus on, when writing the tests, could have been a useful thing. Most of MCF’s techniques are more how to refactor or change the code, so it is possible to add tests. MCF mentions **Characterization tests**, where the goal is to write an assertion that the developer knows will fail, and then let the test-run tell the user, how the behavior works, and thereby the developer can update the test, to do what the behavior is actually doing.

Overall, the biggest problem with refactoring the code is to put the code under test control, to ensure the behavior and it is one of the most important aspects. This could have had more focus in the book and in the techniques.
Freeman and Pryce have a guide in their book [FreemanPryce] about how to handle methods that are difficult to write tests for. The illustration of the steps are shown below in Figure 17.

![Figure 17 - TDD - writing tests](image)

Freeman and Pryce state that if the test are difficult to write, there must be something wrong with the structure. By looking at this, it is possible to compare it to the updated legacy code change algorithm. When writing tests, it could be necessary to go back and break dependences before being able to write the tests. This would be step 4 and 5 in the updated legacy code change algorithm.
6 Discussions on legacy code refactoring

MFC are showing some efficient techniques when working with legacy code. The book [Michael C. F.] is covering problems in a structured way, where he has given the chapters in the book, names of problems that a developer could face.

This could be titles like, “Dependencies on libraries are killing me”, “This class is too big and I don’t want it to get any bigger” or “How do I know that I’m not breaking anything?”. The titles are easy to understand and the developer can find a title that sounds like a problem in the code-base. The title “This class is too big and I don’t want it to get any bigger”, was matching the description of the `CCVTDatabase` class, when first looking at it, which made it the first choice for a starting point, when looking at the class.

The techniques and tools shown my MCF for refactoring legacy code, gives the developer a great starting point. However, this will only help getting started, the developer still has to get his own flow and understanding of how he views and see things. If he prefers to use effect sketching or draw directly in a program that can handle UML notations and other dissensions like this.

Many of the techniques used by MCF are not following official standards, but they are the most simplest possible. This will then be a personal opinion if it is the best practice or not. Nevertheless, MCF shows the reader the simplest and quickest possible way to work with legacy code.

During the refactoring process, there wasn’t enough time to refactor all 3 layers. It was a bigger and more struggling process than anticipated. The CVT project architecture was full of hidden and strong dependencies, which made it very difficult to refactor and write tests. However, this made a starting point, for the process in getting a more clean and separated architecture for the system. Also a lot of other design problems were found, that needs to be focused on at a later point, for the InnoScan developers.

Some of the things MCF doesn’t have any information about, are Anti-patterns and prototyping, which are to common tools/processes for working with code design, the structure of the system and the developing process. In the next two sections, there will be a small overview of why these things could benefit the process of working with legacy code.

6.1.1 Anti-patterns vs Dependency-Breaking techniques

MCF wrote the chapter “Dependency-Breaking Techniques” [Michael C. F.] (4.7), where explains some techniques that can help the developer break some of the dependencies, without altering any behavior.

Another way of doing this would be to use Anti-patterns. Anti-patters are conceptually similar to patters in that they document recurring solutions to common design problems.
They describe the common mistakes made during software development as well as the solution to these problems [Smith2000].

The standard template that anti-patters follow is a simple one where the following information:
- Name: the section title
- Problem: What is the recurrent situation that causes negative consequences?
- Solution: How do we avoid, minimize or refactor the anti-pattern?

Anti-patterns are refactored (restructured or reorganized) to overcome their negative consequences. A refactoring is a correctness preserving transformation that improves the quality of the software. [Smith2000]

So what is the difference between Dependency-Breaking techniques and Anti-patterns?
MFC follows the basic of an Anti-pattern template. There is a title, a small description of the problem and finally a walkthrough of the solution, with code examples. Therefore, in general MFC’s Dependency-Breaking techniques are the same as Anti-patterns. However, the anti-patterns in the article from Smith [Smith2000], focus on the QA’s, were as MCF doesn’t use this in his book.

Anti-patterns address software architecture and design as well as the software development process itself. This could be something that could be adapted by MCF, to change the Dependency-Breaking techniques into anti-patterns. Nevertheless, using anti-patterns in his book, to help the developer to give examples of wrong structured code, so the developer have clues to what should be changed, and then having the solution to fix it.

6.1.2 Architectural Prototyping

When doing refactoring on legacy code, there are many dependencies and a lot of information to keep track on. This can in some whys be too much and confuse a person. Here a solution could be to use Architectural prototyping [Christensen2005].

Architectural prototyping is a process, were a part of a code-base from a system is copied into its own solution. Thereby a more clean separation from all other parts of the system is achieved. This is done to refactor the code, so that it could follow a different design pattern, to verify if this could perform better, related to the QAS requirements the system has.
After the code is copied into its own solution, the code is refactored, by following correct architectural guidelines, to ensure quality. If the prototype is a success, the code should be so correctly structured, that it can be copied back into the solution and used, without any major changes.

In prototyping, the 3 steps are called:

- **Harvesting**: Where the code is extracted to its own solution.
- **Retrofitting**: Refactoring/changing the code, to the wanted prototype (the change that the developer want to test, to see if it could improve the system).
- **Evolution**: Reinserting the code into the original system.

In Figure 18 below, an example flow of the architectural prototyping can be seen.

By moving the code to an external project, the developer can have a better test station, to do refactoring, without influence the normal system or being confused by many different classes. This could also help the developer in breaking the dependencies, because the user would easily see the classes that are needed, by the single class that is being refactored (retrofitted).

This why of working with the legacy code refactoring, could be a be a simplification for the developer, to get a less complicated workspace. The updated version of MCF’s legacy code change algorithm, could implement this, without the big modification. The introduction of prototyping into the update legacy code change algorithm, can be seen in Figure 19 below.
The changes that would have to be made, is located in step 4, and a new step 7 is added, as it can be seen in Figure 19 above.

Step 4, would have to be changed, so that one thing is to break the dependencies as before, however, this should be done when copying the needed classes, to a new project (Harvesting).

Step 7, is to reinsert the refactored code and the created tests, into the original project (Retrofitting).
7 Conclusion

By applying the principles from Michael C. Feather’s book “Working effectively with legacy code” [Michael C. F.], the refactoring process of the legacy code in the CVT system, can be concluded as a success. The refactoring process was setup in a basic and strait forward way, that walked through each necessary step, of what have to be thought about, when working with legacy code.

The book gives all the basic tools needed for the process, but doesn’t describe the places where the developer has to take care. An example is when using automated tools to gather information about a current system design, where Visual Studio is used to generate the class diagram (Figure 3). This class diagram doesn’t help the user if the system is to complex and also a lot of information is missing from the class diagram, like compositions or aggregation.

However, because MCF uses the simplest techniques like effect sketching, the possibility to make mistakes are limited. As long as the developer, follow the guidelines.

The process of refactoring legacy code is to follow the Legacy code change algorithm, which MCF shows. This algorithm goes though the different steps needed to perform a safe refactoring, that limits the risk of change the behavior of the system. However, the conclusion for this algorithm is that it needs to be updated, to be better to handle undetected dependencies. The updated version allows the developer loop between breaking-dependencies and writing tests (Figure 19), and also in the beginning creating an overview of the structure to refactor.

One of the important things about refactoring legacy code, is to write tests, to ensure the behavior of the system doesn’t change. MFC is only skimming the part of writing the actual tests, and more focusing on the breaking of dependencies and the process itself. This than leaves the understanding and learning of this process to the developer.

The other thing MCF doesn’t talk about is the QA’s of a system. He leaves out the focusing on the system requirements. This is an important aspect on designing software architecture.

Some of the last things that could be adapted by MCF, is Anti-patterns and prototyping (6). MCF uses Dependency-breaking techniques in the same way as anti-patterns, but he doesn’t follow the aspect of focusing on the developing process.

The aspect of prototyping is not used in the refactoring process. The reason for this could be to keep it as simple as possible. However, prototyping gives the developer the opportunity to work on the specific functionality without influencing any other parts of the system. Also it would make it easier to try different aspects/designs, that focus on the QAS of the system.

The overall conclusion of working with MCF’s book is that it gives the developer some quick and efficient tools, when refactoring legacy code. The principles guided the refactoring of the CVT system, to get a clearer and clean separation between the different layers.
8 References


[M. Fowler] Martin Fowler: *Patterns of enterprise application architecture*

[GOF] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns – Element of Reusable Object-Oriented Software*


[Koschke2005a] Rainer Koschke: *What architects should know about reverse engineering and reengineering*

[FreemanPryce] Steve Freeman & Nat Pryce: *Growing Object-Oriented Software, Guided by Tests*

[Smith2000] Connie U. Smith & Lloyd G. Williams: *Software Performance AntiPatterns*

[Christensen2005] Henrik Bærbak Christensen: *Towards an Operational Framework for Architectural Prototyping*


[Perl] Information about Perl: [http://www.perl.org](http://www.perl.org)

### 9 Word list

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCF</td>
<td>Michael C. Fathers</td>
</tr>
<tr>
<td>UI</td>
<td>User interface</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
</tr>
<tr>
<td>QA</td>
<td>Quality Attribute</td>
</tr>
<tr>
<td>QAS</td>
<td>Quality Attribute Scenario</td>
</tr>
<tr>
<td>CVT</td>
<td>Computer Vision Technology</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>WPF</td>
<td>Windows Presentation Foundation</td>
</tr>
<tr>
<td>VS</td>
<td>Visual Studio</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environments</td>
</tr>
</tbody>
</table>
10 Appendixes

10.1 CVT user interface
10.2 CCVTDatabase grouping of methods

**General database methods:**
- `public void FireDatabaseChangedEvent()`  
- `public void OnFireDatabaseChangedEvent()`  
- `public bool Open(string ConnectionString, string ProviderName, string MinDBVersion)`  
- `public void Close()`  
- `public int ExecuteNonQuery(string CommandText)`  
- `public object ExecuteScalar(string CommandText)`  
- `public DbDataAdapter GetSqlDataAdapter(string SelectCommand)`  
- `public DbDataAdapter GetTableDataAdapter(string TableName)`  
- `public DbCommand GetCommand(string CommandText)`  
- `public DbDataReader GetDataReader(string CommandText)`  
- `public DataTable GetDataTable(string TableName)`  
- `public DataTable GetDataTable(string CommandText, string TableName)`  
- `public DataTable GetDataTable(DbDataAdapter da, string TableName)`  
- `public DataSet GetTableDataSet(string TableName)`  
- `public DataSet GetTableDataSet(string CommandText, string TableName)`  
- `public DataSet GetTableDataSet(DbDataAdapter da, string TableName)`

**Batch methods:**
- `public void FireBatchEvent(EBatchEvent Reason, CBatchInformation info)`  
- `public void OnFireBatchEvent(EBatchEvent Reason, CBatchInformation info)`  
- `public bool UpdateTransportMode(String batchNumber, uint transportMode)`  
- `public bool CreateNewBatch(CBatchInformation BatchInformation)`  
- `public int DeleteBatch(int BatchId)`  
- `public bool IsSubBatchNumberLegal(int BatchId, string SubBatchNumber)`  
- `public bool CreateNewSubBatch(CBatchInformation BatchInformation)`  
- `public int GetBatchId(CBatchInformation BatchInformation, string RecipeNumber)`  
- `public CBatchInformation GetCurrentBatchInformation()`  
- `public bool GetBatchInformation(CBatchInformation BatchInformation, string RecipeNumber)`  
- `public bool GetBatchInformation(CBatchInformation BatchInformation)`  
- `public bool GetSubBatchInformation(CBatchInformation BatchInformation, string RecipeNumber)`  
- `public bool GetSubBatchInformation(CBatchInformation BatchInformation)`
- `public bool CloseBatch(CBatchInformation BatchInformation)`  
- `public bool CloseBatches(CBatchInformation BatchInformation)`  
- `public CBatchInformation LoadBatch(int CurrentBatchId)`  
- `public bool CloseSubBatch(CBatchInformation BatchInformation)`  
- `public int[][] GetBatchCounters(DataRow dr)`

**Machine config methods:**
- `public void FireMachineConfigChangedEvent(bool Recreated)`  
- `public void OnFireMachineConfigChangedEvent(bool Recreated)`  
- `public bool WriteMachineConfig(string UnitName, ref byte[] Buffer)`  
- `public void WriteDefaultMachineConfig()`  
- `bool MachineConfigurationTableHeader(ref string TableSql, string TableName)`
Product config methods:

```csharp
public void FireProductChangedEvent(bool Recreated)
public void OnFireProductChangedEvent(bool Recreated)
public string GetRecipeVersionNumber(string RecipeNumber)
public int GetRecipeId(string RecipeNumber)
public DataRow CreateNewProduct(DataRow DataRowToCopyFrom, string ProductNumber, string Version, string Description)
public bool DeleteProduct(DataRow DataRow)
public bool ReleaseProduct(DataRow DataRow)
public bool WriteProductConfig(int ProductConfigId, string UnitName, ref byte[] Buffer)

bool ProductConfigurationTableHeader(ref string TableSql, string TableName)
```

Other methods:

```csharp
public static string EscapeQoutes(string src)
public ArrayList TableName(EConfigType ConfigType)
public bool DropAllTables()
public string LocalisedModuleName(string ModuleName)
public string LanguageLookup(DataRow dr)
public string LanguageLookup(DataRow dr, string ColHeader)
public string GetLanguageName(string TableName)
public bool SetStartUpStateParam(string Name, string Value)
public bool ExportDataTableToCSV(string TableName, char Separator)
public bool ExportDataTableToCSV(string TableName, char Separator, string FilePath)
public void ExportDataTableToCSV(DataTable dt, char Separator)
public void ExportDataTableToCSV(DataTable dt, char Separator, string FilePath)
public static void ExportDataTableToCSV(object data)
public static void RemoveQuotes(string[] strings)
public bool ImportCSVToDataTable(string TableName, char Separator, string FilePath)
public static string[] GetColumnNames(StreamReader FileReader, char Separator)
public bool ImportCSVToDataTable(DataTable dt, char Separator, string FilePath, bool bCreateColumns)
public bool GetSystemParam(string Name, ref string Value)
public string GetSystemParam(string Name, string strDefault)
public bool GetSystemParam(string Name, bool bDefault)
public int GetSystemParam(string Name, int iDefault)
public double GetSystemParam(string Name, double fDefault)
```
public bool GetStartUpStateParam(string Name, ref string Value)
public string GetStartUpStateParam(string Name, string strDefault)
public int GetStartUpStateParam(string Name, bool bDefault)
public byte[] GetStartUpStateParam(string Name, byte[] Default)
public String GetGlobalOpcTag(String NameId, String defaultTag)
public String GetGlobalOpcDefaultValue(String NameId, String defaultValue)
public int FillGridViewColumns(GridView view, DataTable table, bool bClear)
bool DropTable(string TableName)
public bool ColumnExists(string ColName, string TableName)
public void PrintXml(Stream XmlStream, string XamlHeader)
public void AppendXmlToReport(Production.BatchMenu.CReport report, Stream XmlStream, ref int iPage)
public void CreateTable(string TableName)
public void AddConfigColumns(string ConfigType)
public bool CreateTableSql(string Sql)
bool AppendForeignKey(ref string MasterTableSql, ref string MasterTableForeignKeySql, string TableName)
bool CreateUnitTable(ref string MasterTableSql, ref string MasterTableForeignKeySql, string TableName, bool bProduct)
bool CreateUnitTables(ref string ProductSql, ref string ProductForeignKeySql, ref string MachineSql, ref string MachineForeignKeySql, string StationName)
bool CreateInspectionStation(ref string ProductSql, ref string ProductForeignKeySql, ref string MachineSql, ref string MachineForeignKeySql, string StationName)
void CreateTable(string TableName)
void CreateRotationTables()
bool CreateOEETables()
### 10.3 Preserve signature example

**Not preserving the signature:**

```java
Private void TestMethod1(int id, String name, String phoneNumber, int age)
{
    TestMethodStatic(new PersonObject(id, name, phoneNumber, age));
}
Private Static void TestMethodStatic(PersonObject person)
{
    ...
}
```

**Preserving the signature:**

```java
Private void TestMethod1(int id, String name, String phoneNumber, int age)
{
    TestMethodStatic(id, name, phoneNumber, age);
}
Private Static void TestMethodStatic(int id, String name, String phoneNumber, int age)
{
    ...
}
```
10.4 CCVTDatabase method: Open

```csharp
public bool Open(string ConnectionString, string ProviderName, string MinDBVersion)
{
    try
    {
        m_DataFactory = DbProviderFactories.GetFactory(ProviderName);
        m_Connection = m_DataFactory.CreateConnection();
        m_Connection.ConnectionString = ConnectionString;
        m_Connection.Open();
    }
    catch (Exception e)
    {
        new CISEvent(e);
        throw new CDatabaseException(e.Message);
    }

    string[] split = m_Connection.Database.Split(new Char[] { '\\', '/ ' });
    string strDBVersion = GetSystemParam("DatabaseVersion", "");

    m_DBVersion = new CModuleVersion(split[split.GetUpperBound(0)], strDBVersion);

    if (m_DBVersion.Verify(MinDBVersion) != CModuleVersion.EState.VerifyOK)
        return false;

    m_CurrentBatchInfo = new CBatchInformation();

    int CurrentBatchId = GetStartUpStateParam("CurrentBatchId", 0);
    int CurrentSubBatchId = GetStartUpStateParam("CurrentSubBatchId", 0);

    if (CurrentSubBatchId > 0)
    {
        m_CurrentBatchInfo.SubBatchId = CurrentSubBatchId;
        GetSubBatchInformation(m_CurrentBatchInfo);
    }
    else if (CurrentBatchId > 0)
    {
        m_CurrentBatchInfo.BatchId = CurrentBatchId;
        GetBatchInformation(m_CurrentBatchInfo);
    }

    m_CurrentBatchInfo.BatchSize = GetStartUpStateParam("RemainingBatchSize", -1);
    m_CurrentBatchInfo.SampleSize = GetStartUpStateParam("RemainingSampleSize", -1);

    string strIdFields = GetSystemParam("BatchIdFields", EBatchIdLevels.BatchNumber.ToString());

    string[] IdFields = strIdFields.Split(new char[] { ',', ';' });
    m_BatchIdFields = new EBatchIdLevels[IdFields.Length];

    for (int i = 0; i < IdFields.Length; i++)
```
{  
  string field = IdFields[i].Trim().ToLower();

  for (int f = 0; f < (int)EConstants.BatchIdLevelCount; f++)
  {
    EBatchIdLevels Id = (EBatchIdLevels)f;

    if (field == Id.ToString().ToLower())
    {
      m_BatchIdFields[i] = Id;
      break;
    }
  }

  m_IsInitialized = true;

  return true;
}
10.5 CCVTDatabase split method: Open

10.5.1 Step 1.

```csharp
public bool Open(string ConnectionString, string ProviderName, string MinDBVersion)
{
    try
    {
        m_DataFactory = DbProviderFactories.GetFactory(ProviderName);
        m_Connection = m_DataFactory.CreateConnection();
        m_Connection.ConnectionString = ConnectionString;
        m_Connection.Open();
    }
    catch (Exception e)
    {
        new CISPEvent(e);
        throw new CDatabaseException(e.Message);
    }
    return true;
}

public bool Initialize()
{
    string[] split = m_Connection.Database.Split(new Char[] { '\', '/' });
    string strDBVersion = GetSystemParam("DatabaseVersion", "");
    m_DBVersion = new CMODULEVersion(split[split.GetUpperBound(0) - 1], strDBVersion);
    if (m_DBVersion.Verify(MinDBVersion) != CMODULEVersion.EB packet.VerifyOK)
    return false;

    m_CurrentBatchInfo = new CBatchInformation();
    int CurrentBatchId = GetStartUpStateParam("CurrentBatchId", 0);
    int CurrentSubBatchId = GetStartUpStateParam("CurrentSubBatchId", 0);
    if (CurrentSubBatchId > 0)
    {
        m_CurrentBatchInfo.SubBatchId = CurrentSubBatchId;
        GetSubBatchInformation(m_CurrentBatchInfo);
    }
    else if (CurrentBatchId > 0)
    {
        m_CurrentBatchInfo.BatchId = CurrentBatchId;
        GetBatchInformation(m_CurrentBatchInfo);
    }
}
```

10.5.2 Step 2.
10.6 CCVTDatabase find place method Open is used

Before:

```csharp
if (!Globals.Database.Open(m_ConnectionString, m_ProviderName, m_MinDBVersion))
{
    CMessageWindow.ShowDialogRes("WrongDatabaseVersion");
    Globals.CloseAllPCPrograms(false);
}
```

After:

```csharp
if (!Globals.Database.Open(m_ConnectionString, m_ProviderName))
{
    CMessageWindow.ShowDialogRes("CouldNotConnectToDatabase");
    Globals.CloseAllPCPrograms(false);
}
if (!Globals.Database.Initialize(m_MinDBVersion))
{
    CMessageWindow.ShowDialogRes("WrongDatabaseVersion");
    Globals.CloseAllPCPrograms(false);
}
```
10.7 Tests for during refactoring

Tests written during the refactoring can be view in the C# project: UnitTestsHMI.csproj

10.7.1 Tests for data source layer

Code file for test of the data source layer can be seen in file: UnitTestCCVTDatabase.cs

10.7.2 Tests for domain layer

Code file for test of the domain layer can be seen in file: UnitTestBatchStateHandler.cs

Fake persistence (data source layer) class can be seen in:
FakeObjects\FakeBatchPersistence.cs
10.8 IBatchPersistence interface

```csharp
public interface IBatchPersistence
{
    void FireBatchEvent(EBatchEvent Reason, CBatchInformation info);
    void OnFireBatchEvent(EBatchEvent Reason, CBatchInformation info);
    bool UpdateTransportMode(String batchNumber, uint transportMode);
    bool CreateNewBatch(CBatchInformation BatchInformation);
    int DeleteBatch(int BatchId);
    bool IsSubBatchNumberLegal(int BatchId, string SubBatchNumber);
    bool CreateNewSubBatch(CBatchInformation BatchInformation);
    int GetBatchId(CBatchInformation BatchInformation, string RecipeNumber);
    CBatchInformation GetCurrentBatchInformation();
    bool GetBatchInformation(CBatchInformation BatchInformation, string RecipeNumber);
    bool GetBatchInformation(CBatchInformation BatchInformation);
    bool GetSubBatchInformation(CBatchInformation BatchInformation, string RecipeNumber);
    bool GetSubBatchInformation(CBatchInformation BatchInformation);
    bool CloseBatch(CBatchInformation BatchInformation);
    bool CloseBatches(CBatchInformation BatchInformation);
    CBatchInformation LoadBatch(int CurrentBatchId);
    bool CloseSubBatch(CBatchInformation BatchInformation);
    int[,][] GetBatchCounters(DataRow dr);
}
```
10.9 Domain layer – Batch Sequence update

10.9.1 BatchPersistence

```csharp
public List<Dictionary<String, object>> GetBatchSequence()
{
    List<Dictionary<String, object>> batchSequences = new List<Dictionary<String, object>>();

    DbDataAdapter da = _database.GetSqlDataAdapter("SELECT * FROM BatchSequence");
    System.Data.DataTable dt = _database.GetDataTable(da, "Tables");

    foreach (System.Data.DataRow reader in dt.Rows)
    {
        Dictionary<String, object> batchSequence = new Dictionary<String, object>();
        foreach (System.Data DataColumn dc in dt.Columns)
        {
            batchSequence.Add(dc.Caption, reader[dc.Caption]);
        }
        batchSequences.Add(batchSequence);
    }
    return batchSequences;
}
```

10.9.2 CCVTDatabase

```csharp
public List<Dictionary<String, object>> GetBatchSequence()
{
    return m_BatchPercictence.GetBatchSequence();
}
```

10.9.3 IBatchPersistence

Added to the interface.

```csharp
List<Dictionary<String, object>> GetBatchSequence();
```

10.9.4 BatchStateHandler

Before:
DbDataAdapter da = Globals.Database.GetSqlDataAdapter("SELECT * FROM BatchSequence");
System.Data.DataTable dt = Globals.Database.GetDataTable(da, "Tables");

foreach (System.Data.DataRow reader in dt.Rows)
{
...
}

After:

string language = Thread.CurrentThread.CurrentCulture.TwoLetterISOLanguageName;
List<Dictionary<String, object>> batchSequence = Globals.Database.GetBatchSequence();

foreach (Dictionary<String, object> reader in batchSequence)
{
...
}
10.10 Domain layer – Use of IBatchPersistence

10.10.1 BatchStateHandler

```csharp
public class CBatchStateHandler
{
    private IBatchPersistence _BatchPersistence;
    ...
    ...

    public CBatchStateHandler(IBatchPersistence batchPersistence)
    {
        _BatchPersistence = batchPersistence;
        ...
        ...
        List<Dictionary<String, object>> batchSequence = _BatchPersistence.GetBatchSequence();
        ...
    }
}
```

10.10.2 BatchPage.xaml.cs

Before:

```csharp
state = new CBatchStateHandler();
```

After:

```csharp
state = new CBatchStateHandler(Globals.Database);
```