Hovedopgave

Master i informationsteknologi
Linien i softwarekonstruktion

“Bringing a Legacy System under Test Control”

Af
Niels Hauge Bock
9. juni 2013

_______________________                          _______________________
Niels Hauge Bock, studerende                          Clemens Klokmose, vejleder
Abstract

In his book "Working Effectively with Legacy Code" Michael C. Feathers defines all code not being covered by unit tests as legacy code.

Are you using this definition, the Danske Bank Gemini system is for the main part a legacy system.

Striving for making the Danske Bank Gemini system more maintainable, there is an opportunity in bringing the system under test control.

Having a system under test control gives confidence in making changes having tests that preserve the existing functionality. Furthermore it gives confidence in the process of refactoring the system into a more flexible and maintainable design.

Michael Feathers have defined a set of techniques that can help break the dependencies in the code, which discourage you from writing unit tests.

Using these techniques on examples from the Gemini system codebase, show that it is possible to bring a legacy system under test control, which enables you to increase the systems level of software quality.
Content
1 Motivation ......................................................................................................................4
2 Hypothesis .....................................................................................................................5
3 Method ............................................................................................................................5
  3.1 Is it convincing ...........................................................................................................5
  3.2 The report ...................................................................................................................5
4 Analyses and Results .......................................................................................................6
  4.1 Why is it relevant to look at Feathers’ techniques .....................................................6
    4.1.1 Maintainability .....................................................................................................6
    4.1.2 Testability ............................................................................................................7
    4.1.3 Challenges that we face .......................................................................................7
  4.2 Unit testing ..................................................................................................................7
    4.2.1 Working with unit tests .......................................................................................7
    4.2.2 Enabling code for unit testing .............................................................................9
    4.2.3 Test Driven Development ..................................................................................10
    4.2.4 Using an automated test framework .................................................................11
  4.3 Getting a good design ...............................................................................................11
  4.4 The Gemini System ....................................................................................................13
    4.4.1 Is Gemini a Legacy system ...............................................................................13
    4.4.2 Gemini system “components” ..........................................................................13
    4.4.3 Development environment ..............................................................................16
  4.5 Feathers approach .....................................................................................................16
    4.5.1 The Book .............................................................................................................16
    4.5.2 The Legacy Code Dilemma ...............................................................................16
    4.5.3 Changing the code .............................................................................................18
    4.5.4 The Legacy Code Change Algorithm ...............................................................18
  4.6 Using Test Driven Development and Feathers’ approach .........................................21
  4.7 Bringing the Gemini system under test control .......................................................21
    4.7.1 Obstacles in the organization ............................................................................21
    4.7.2 Obstacles in the codebase in terms of Feathers’ ..............................................25
4.8 Reflections on the approach ......................................................... 45
4.9 Implementing a new feature ....................................................... 45
  4.9.1 The Gemini Label control ..................................................... 45
  4.9.2 Getting Gemini Label control under test control and adding the feature 46
4.10 Do a bug fix .............................................................................. 54
  4.10.1 The UserParameterLoader class .......................................... 54
  4.10.2 Getting Gemini UserParameterLoader class under test control and fixing a bug .............................................................. 55
4.11 The use of Mocking Frameworks ............................................... 65
5 Future work .................................................................................. 67
6 Conclusion ..................................................................................... 68
7 References .................................................................................... 69
8 Appendix ....................................................................................... 70
1 Motivation
In Danske Bank we have an internally developed system called Gemini.

Gemini is a system used for, with ease, developing and running web applications following a Danske Bank standardized design.

The web applications developed using the system are both internal systems, available through the intranet, but also externally available applications like the Danske Net Bank.

The system is composed of several parts, but the central parts are:

- A windows forms based development tool, which is used for design of web parts and pages, and persisted as metadata.
- A runtime part which, on the basis of the metadata, generates the web pages using a combination of proprietary web controls and standard Asp.Net controls.

This webpage functionality and the proprietary web controls, which are used at runtime, are also available as a software development kit (SDK), for use developing more advanced webpages.

There are constantly needs for new controls and adding of new features, which do that more advanced functionality can be developed using the Gemini system.

Before releasing new versions of the system, a more thorough test is going on, which also stretch over a longer period of time.

The wish is to cover the code with unit tests, which can preserve against introducing new errors, and thereby reduce the need for manual tests and shorten the test period, moving towards continues delivery.

There is also a wish of better the performance with which new functionality can be developed, this are going to be done through achieving a better system design, which is easy to change and extend.

We have decided on a strategy of implementing Unit tests and Test Driven development along with the development of new features and bug fixing.
2 Hypothesis

By the use of Feathers’ techniques, it is possible to bring the Gemini system under test control.


3 Method

By the use of the terms defined by Feathers, I will describe the problems that typically appears in the Gemini system codebase, when starting writing or adding unit tests. I will look at his techniques and investigate which of his techniques, which with advantage, can be used to bring the Gemini system under test control.

I will by solving a task of adding a new feature and fixing a bug, try out some of the techniques and evaluate their strength.

By this work I will evaluate whether or not we can believe in the hypothesis, that the techniques can bring the Gemini System under test control.

3.1 Is it convincing

Doing these tasks, using Feathers techniques, will not in it self prove the hypothesis of being able to bring Gemini under test control. It will still demand hard work and dedication to writing unit tests, but if the tasks can show that it can be done in small and with fast and convincing results, it will mean that we can believe in Feathers techniques and thereby the hypothesis.

3.2 The report

The fact that more of Feathers’ techniques refer to each other and are used together, entail that I have chosen to make a lists of the techniques used in the appendix, in order to be able to refer to techniques, that have not yet been used in an example.

Many of the examples that I use build on the same class. I haven chosen to use this single class because it is not that big but is fairly descriptive for the web controls that we have in the Gemini system and it holds many of the problems that we run into when trying to write test cases for our web controls. Another reason for concentrating the examples around this single class is that we in this way gets to know the class pretty well.
4 Analyses and Results

4.1 Why is it relevant to look at Feathers’ techniques

Looking at the motivation for this analysis, you can say that we are striving to better the quality attributes “Maintainability” and “Testability” [Christensen 2010].

4.1.1 Maintainability

The ISO Standard 9126, which is the international standard for measuring the product quality in software engineering, defines maintainability like this:

**Maintainability (ISO 9126)**

The capability of the software product to be modified. Modifications may include corrections, improvements or adaption of the software to changes in the environment, and in the requirements and functional specifications.

[Christensen 2010]

If we also include the definition of modifiability by L. Bass et al. [Bass 2003]

**Modifiability**

“Modifiability is about the cost of change and refers to the ease with which a software system can accommodate changes.”.

[Bass 2003]

We can see that maintainability is a cost measure, and a system with a good maintainability is a system where changes can be introduced at a low cost. Making changes at a low cost is one of the motivations for looking at Feather’s techniques to bring the Gemini system under test control.

The ISO standard defines 4 subquality attributes of maintainability

1. Analyzability
2. Changeability
3. Stability
4. Testability

As seen number 4 is testability, which means that in order to strengthen the maintainability of the Gemini system, we have to strengthen the systems testability.

Strengthen the testability is what Feather’s techniques are going to help us to do by getting the Gemini system under test control.
4.1.2 Testability
The ISO standard 9621 defines testability

Testability (ISO 9621)
The capability of the software product to enable a modified system to be validated

[Christensen 2010]

A system covered with unit tests helps to preserve existing functionality of the system. Adding tests covering new functionality enables together with the existing tests the system to be validated after modification.

But also in the process of fixing bugs, unit tests can be a help. Using unit test builds on direct feedback running the unit tests in an automatic test framework. Having a system which is under test control, makes it easier to write new test cases that can demonstrate a bug in the system.

Using unit tests and an automatic test framework supports the testability quality attribute of a system.

4.1.3 Challenges that we face
The challenges that we face up front, are that the code design not in all aspects have a design that makes the code testable by unit testing.

The Gemini codebase are developed in the C# language, which is a language, where methods are not open for inheritance and overriding by default, the method need to be marked virtual. Hence many classes cannot have there dependencies broken by inheritance and overriding of methods.

Another thing is that not many classes have there contract and responsibility defined by an interface, and not many classes have dependencies loosely coupled by the use of constructor or method dependency injection. It is therefore difficult to break dependencies in order to write tests using test doubles. If we are going to do this we have to refactor parts of our codebase.

The risk of introducing bugs and loose behavior makes many people give up on improving code design and writing unit tests.

4.2 Unit testing

4.2.1 Working with unit tests
What is a Unit test and what qualities must a Unit test have:
Unit test:

It’s an automated piece of code that invokes a different method and then checks some assumptions on the logical behavior of that method or class.

It’s written using a unit-testing framework.

It can be written easily.

It runs quickly.

It can be executed repeatedly by anyone on the development team.

[Osherove 2009]

If we add Feathers’ definition of what a unit test is not:

A test is not a unit test if:

It talks to a database

It communicates across a network

It touches the file system

You have to do special things to the environment (such as editing configuration files) to run it

[Feathers 2004]

We get a good indication of what dependencies we are going to break in order to bring existing code under test.

When doing unit testing what we want to do is testing a method or a class in isolation.

In order to do this we must be able to instantiate the class and be able to call the methods, which we are going to test.

Furthermore we must be able to provide input being in control of the variables that influences our class or method under test.

To verify the correctness of the work, which our class conducts, we also have to be able to sense key variables and the influence it has on objects of other classes.
By doing unit testing we get some benefits (I presuppose using an automated test framework)

- Preserves existing functionality, executing existing test will show if any functionality are broken.
- Instant feedback – executing the test after a small change, makes it easier to find the bug, because very little code has changed and you know what code has changed.
- The test code documents the functionality of the code being tested
- “When we write tests we feel the pain of bad design” [Feathers 2012]

Writing unit test, we must in order to control the influence of and on other objects, use some substitutes for the depending objects. Till that we make use of test doubles:

Test Doubles

It’s a common technique to create implementations of Dependencies that act as stand-ins for the real or intended implementations. Such implementations are called Test Doubles, and they will never be used in the final application. Instead, they serve as placeholders for the real Dependencies, when these are unavailable or undesirable to use.

[Seemann 2012]

Developing classes for use as test doubles enables us to create test specific methods to set the value of variables and methods that return a predefined result.

4.2.2 Enabling code for unit testing

To be able to use unit testing to test our codebase, some requirements to the code and the design of the system have to be fulfilled.

Sensing

It must be possible to get access to and sense the values our code computes.

An example could be method that does a complex calculation, but the method is private or the result is stored in a private variable in the class under test, so that we cannot read the value in our unit test.

Separation

There must be a separation or a loose coupling between the classes, which enables instantiating the class under test in the automatic test framework.
As an example this will demand the possibility of inheriting from a class and override the dependencies or the possibility of using dependency injection, so that the class under test can be injected with test double versions of the depending objects.

**Seams / Variability points or hotspot** [Christensen 2010]

It must be possible to reach the “Seams”, which is a point in the code where enabling of different behaviors can take place. As and example programming against an interface instead of a concrete type creates such a seam in the code.

4.2.3 **Test Driven Development**

We will first and foremost use the Test Driven Development approach for the implementation of new features.

Test Driven Development is an approach presented by Kent Beck [Beck 2002] and an approach that will secure that the implementation of new code are done by small steps and thereby simplifying the development process.

Test Driven development is based on 3 concepts

- Use an automated test framework
- Write tests first
- Maintain a list of tests for the new functionality

Using Test Driven development when adding new features lead to a testable design, and you don’t have to bother breaking dependencies, when building your system on Test Driven development.

For the implantation of the new feature the TDD rhythm is used

The TDD Rhythm

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

If we will succeed we most focus on the TDD principle of taking small steps, running the necessary tests for every small change in order to optimize the development process by catching failures as early as possible.

In order to be able to run the test again and again we must focus on the performance of our unit test, and optimize if taking too long.
4.2.4 Using an automated test framework
To bring a system like Gemini under test control, we need an automated test framework, where we can specify our test, make them to run and follow how they succeed or fail. I will not get deeper into the automated test framework, but only assume that we are using the Microsoft Unit Testing Framework that is build into Microsoft Visual Studio.

4.3 Getting a good design
Getting a maintainable system is a lot about getting a system designed for change.

Both the principles of flexible design [Christensen 2010] and Feathers techniques for making a change, having got parts of a legacy system under test control, prioritize refactoring after implementation.

Refactoring according to Flexible, Reliable Software [Christensen 2010]

Refactoring
Refactoring is the process of modifying and restructuring the source code to improve maintainability and flexibility without affecting the system’s external behavior when executing.

[Christensen 2010]

Doing the refactoring as part of making a change is an important step in order to keep and improve the maintainability of the system.

To support getting a good design, there are some known principles that also Feathers’ have in mind constructing his techniques for breaking dependencies and enabling test.

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.

[Feathers 2004]

Which secures that the system are build of small classes with a well defined responsibility. Small classes that make the code easier to read analyze and test.

In large systems like the Gemini system the dependencies between the different parts of the system are of great importance.
The degree of dependencies can be expressed in degree of coupling and cohesion.

Weak coupling means one change does not influence all other parts of the software. High cohesion means that a change is likely localized in a single part of the system, both making the system more maintainable, lowering the cost of change.

Having a low coupling and high cohesion have some practical influence on the maintainability of a large system because it makes parallel development possible [Seemann 2012].

The Open/Closed principle [Feathers 2004] or the Change by addition rule [Christensen 2010]

<table>
<thead>
<tr>
<th>Change by addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change by addition us behavioral changes that are introduced by adding new production code instead of modifying existing.</td>
</tr>
</tbody>
</table>

[Christensen 2010]

states that a maintainable system must be open for change but closed for modification, which make sure that the system is maintainable without running a big risk of breaking existing functionality.

The background for getting a flexible system is the principles defined by the Gang of Four. [Christensen 2010]

1. Program to an interface, not an implementation
2. Favor object composition over class inheritance
3. Consider what should be variable in your design

This secures a compositional design with a high degree of flexibility enabling adding new features to the system.

Again here the degree of coupling is in play, because indirect use through an interface lowers the coupling.

Everybody in the Gemini development team agree that is a good idea to write unit tests, and more and more to take the time to get it done. But to get the big time reduction when developing new features, it is important that we also focus on refactoring for building a system designed for change.
4.4 The Gemini System

A few statistics about the Gemini System:

Lines of c# code: 138799

Number of developers working on the code: 28

Development sites: 2 - Brabrand, Denmark and Bangalore, India

4.4.1 Is Gemini a Legacy system

In Danske Bank, what we normally designate as a Legacy System, are systems executing on a mainframe and developed in languages as COBOL and PL/1. It is systems that are not object oriented and is hard to analyze, hard to make changes to and hard to add new features to.

In Feathers terminology a Legacy system is a system without unit tests, because the absence of unit tests makes it hard to make changes and hard to add new features.

Using Feathers definition you can say that we are still developing Legacy code as long as we are not writing unit tests.

The Gemini System does have a set of unit tests and parts of the code are covered with unit tests, but big parts of especially the original code and the webcontrols are not under test control and can therefore by this definition go under the term Legacy System.

4.4.2 Gemini system “components”

As mentioned in the motivation the Gemini system is a system used for, with ease, developing and running web applications following a Danske Bank standardized design.

The main parts are the windows form based development tool called Gemini Studio used for designing web parts being parts of the webpages used in different webapplications. The Gemini Studio development tool have user interface as shown below.
The other main part of the system is the runtime environment and infrastructure that enables using the metadata defined web parts in the web applications.

Below is a simplified drawing of the main players in the environment that the Gemini Studio and the Gemini Runtime are parts of.
As seen from the drawing the Gemini Studio windows application uses the Gemini backend webservices for retrieving and storing metadata descriptions of the webparts developed.

The Gemini Frontend internet server uses the Gemini backend webservices for retrieving metadata for the webparts that have to be rendered into the webpages accessed by the users of the webapplications.

To access the data used by the webparts the metadata holds descriptive information of the Danske Bank backend services retrieved through the UDDI and information of the control usage as defined in Gemini Studio when developing.
At runtime these metadata are used for accessing the Gemini Backend and mapping data to and from the Danske Bank backend services and the webcontrols.

**4.4.3 Development environment**

Gemini is developed using the .Net and Asp.Net frameworks. As integrated development environment Visual Studio is used with the possibilities of using the standard build in refactoring tools. Visual Studio also includes a standard automatic test framework for executing unit tests.

**4.5 Feathers approach**

**4.5.1 The Book**

Feathers’ book is divided into 3 parts.

The first part describes the overall considerations, we do before we start changing the code.

The second part goes through the change process of changing Legacy code using Feathers’ purposed techniques.

The third part is a list of techniques for breaking dependencies, which stands in our way when we wish to write unit test when making code changes. The techniques are described in step guides and as examples in part two, so this third part is more a dictionary of techniques.

I will use the book by looking at the considerations Feathers’ describes in the first part of the book, but combine them with the considerations regarding getting a good design and the techniques we have looked at in the work with flexible and reliable software.

I will reflect on the differences also because Feathers’ book is some years old.

I will in the same way as Feathers’ do in the second part of the book go through the different techniques that he purpose, and find the techniques that we can use in the work with getting the Gemini codebase under test. I will illustrate the use with examples from the Gemini code.

Feathers’ book do cover a lot of different programming languages. As we do only work with .Net and specifically c#, I have chosen to focus on the techniques, which address this type of language.

**4.5.2 The Legacy Code Dilemma**

When working with bringing existing code under test control, the ideal is to write tests without having to change the code.
Very often, if not always, this is not possible and this is what Feathers calls “The Legacy Code Dilemma”

**The Legacy Code Dilemma**

When we change code, we should have tests in place. To put tests in place, we often have to change code.

[Feathers 2004]

What Feathers techniques will help us to do is bringing tests in place without breaking the code and preserve existing functionality.

Feathers work with 4 techniques to avoid introducing errors when changing code.

These 4 techniques are general techniques, which can be used, when using the more specific techniques for breaking dependencies, to avoid breaking the code, when making changes without having tests in place.

**Hyperaware computing**

Hyperaware computing, describes a mental condition, where you are very focused on every single letter you write in order to avoid making errors.

Another example can be the use of refactoring tools build into the integrated development environment. These tools can make changes without having tests in place at a very low risk, but calls for much focused use.

**Single goal editing**

Single goal editing is having focus on changing only one thing at a time, take small steps and assure finishing one change before starting on the next.

**Preserve signatures**

Avoid changing the method signature when moving a method from one class to another or new class. You can move with less risk if you only have to change the object reference and not the signature.

**Lean on the compiler**

Use to compile to find what have to be changed in code.

An example is when extracting an interface for an existing class, create an empty interface, change the references to objects of the by references to the new interface type and “lean on the compiler” to find which methods from the class that are referenced and now have to be part of the interface.
4.5.3 Changing the code
Our strategy of creating new tests in connection with developing new features and fixing bugs very well matches the approach Feathers have to bringing legacy code under test control.

Feathers describe the “The Legacy Code Change Algorithm” where he built the writing of tests into the change process.

4.5.4 The Legacy Code Change Algorithm

“The Legacy Code Change Algorithm”

1. Identify change points
2. Find test points
3. Break dependencies
4. Write tests
5. Make changes and refactor

[Feathers 2004]

4.5.4.1 Identify change points
The first thing to do is to identify where in code to make the change.

In the book Feathers describe different ways he attacks the code, when he have to build up an overview of the code in order to identify change points.

The techniques he uses are:

- Making notes and sketches reading through the code, using his own syntax of sketching, just to get an overview.
- Print code listing and markup the code:
  Group and separate responsibilities by marking things differently.
- Scratch refactoring
  Start refactoring, trying to get the code cleaner and learn about the code. Undo refactoring after having learned about the code.

I will not get deeper into these techniques. I think we all have our techniques to do this work and it have not strictly relation to getting the code under test control and making the code more maintainable.

4.5.4.2 Find test points
When needing to find out where write our tests, we have to reason about what effects the change will have.
Feathers’ practice sketching the associations and effects of a change.

The way most of us do this in the Gemini team, is by using the tools in the integrated development environment to find where methods are called from, where variables are referenced and to draw class diagrams showing the hierarchy of classes and use the tool “Show on Codemap” to show a drawing of the dependencies between classes, methods and properties.

Still it can be practical to sketch on paper or whiteboard, but I think everyone have their favorite approach to this task.

As part of finding test points, a step can be to use Feathers’ method use rule

**The Method Use Rule**

Before you use a method in a legacy system, check to see if there are tests for it. If there aren’t, write them. When you do this consistently, you use tests as a medium of communication. People can look at them and get a sense of what they can and cannot expect from the method. The act of making a class testable in itself tends to increase code quality. People can find out what works and how; they can change it, correct bugs, and move forward.

[Feathers 2004]

### 4.5.4.3 Break dependencies

Why break dependencies:

- Make it possible to create an instance of the object and call a method in the automated test framework.
- Isolate the class or method from the influence of other objects.
- Make it possible to sense the variables that can verify that our test is successful.

A lot of code has a design and a structure, which makes it impossible to write unit tests for in the first place.

Feathers have developed a number of techniques to break some of these structures that stand in the way for bringing the code under test control, techniques to refactor without having tests in place, but with a minimum of risk.

### 4.5.4.4 Write tests

What tests to write?

In test driven development you write test that characterize the functionality that your new feature is supposed to deliver. Adding a new feature to a
legacy system you also want to write tests that preserve the behavior that is already in the system. To do that you can try to write tests that characterize the functionality that the system is supposed to have, but that’s not what feathers suggests he says:

“In nearly every legacy system, what the system does is more important than what it is supposed to do” [Feathers 2004]

The tests to write to preserve existing behavior are what Feathers call “Characterization tests”, tests that characterizes the actual behavior.

A technique to write characterization tests is to write a test of one method that you know will fail, look at the assertion and see what the actual result is and change your test to expect that result. Now your test will succeed and you have a test that preserves this behavior.

At the same time the Characterization tests document what the system actually do.

Feather's heuristic for writing “Characterization tests”

**Writing Characterization tests**

1. Write tests for the area where you will make your changes. Write as many cases as you feel you need to understand the behavior of the code.
2. After doing this, take a look at the specific things you are going to change, and attempt to write tests for those.
3. If you are attempting to extract or move functionality, write tests that verify the existence and connection of those behaviors on a case-by-case basis. Verify that you are exercising the code that you are going to move and that it is connected properly. Exercise conversions.

[Feathers 2004]

When writing a characterization test, you can get into something looking suspicious and maybe there is a bug in the system, that does not mean that the tests isn’t good, but that you can note that this test is suspicious and you have to look into this as a bug.

**4.5.4.5 Make changes and refactor**

Feathers’ advice is to use test driven development adding changes to the code.
An important step is therefore to write tests describing and testing the new functionality before writing the code following the steps in the test driven development rhythm.

Feathers’ book is not so concerned about getting a good design. Away to accomplish a good design is to take advantage of Feathers’ techniques to bring the code under test control and afterwards do refactor the code in favor of a flexible design having the refactor process covered by test.

Doing this before adding a new feature or fixing a bug, can lead to a simpler change process.

This refactor process must build on the design principles mentioned earlier leading to a design of a maintainable system.

4.6 Using Test Driven Development and Feathers’ approach

Even though “The Legacy Code Change Algorithm” do look a bit like the rhythm of Test Driven Development it is not. But Test driven Development steps can be added to the legacy code change algorithm at the “make changes” step.

4.7 Bringing the Gemini system under test control

In this chapter will describe obstacles in the codebase obstacles that stands in the way for bringing the Gemini system under test control.

Feathers separate his descriptions of his techniques after the overall problem or most obvious problem that you stands fore.

In the following sections I will go through some of those problems and solutions that Feathers describe and illustrate with examples from the Gemini codebase.

Every section is built on a template of problem, solution and example.

4.7.1 Obstacles in the organization

It’s not only dependencies in the code, that can be an obstacle for writing unit tests and code that can be unit tested.

4.7.1.1 “I don’t have much time and I have to change it”

Problem:

There is a strong wish to do unit testing, either it is as part of test driven development or just writing the test along with and after writing the code.
But the writing of unit test is often sacrificed in order to deliver new features fast.

**Solution:**

Feathers’ presents 2 simple techniques for at least writing unit test for the new code added and get a minimum of tests in place.

“Sprout a method (A1)” or “Sprout a class (A2)”

The steps for “Sprout a method” are the following.

1. Identify where you have to make your code change
2. Write a call for the new method that will do the work involved and comment it out
3. Add local variables as parameters to the new method if needed
4. Add return type to method signature if return of a value is needed
5. Develop the new method using test driven development
6. Remove the comment in the source method to enable the call

If you are changing a large class or you are unable to instantiate the class in the automatic test framework, as an example because of some dependencies. We can create a new class, “Sprout a class”, and implement the new method in the new class. We are now able to instantiate the new class in the automatic test framework.

**Example:**

An example of where it could have been used is in webcontrols part of the system in the class called DbgLabel. In the Render method a new Label Type “Monospace” has been added to the switch statement (Listing 3).
The switch statement is an example of using parametric based variability handling, this is not in focus here but I will look at this part of the code later.

Another more testable way of adding this new feature, without using the time for bringing the existing code under test, could be using the “Sprout a method” technique.

Using the “Sprout a method” technique you simply write new code in a separate method. This will let you write a unit test for new code, you will not get the method insertion point under test, but you will at least get the new code under test.

```csharp
public void RenderMonospaceLabel(HtmlTextWriter writer)
{
    this.Text = GetEncodeText();
    writer.AddAttribute(HtmlTextWriterAttribute.Title, this.ToolTip);
    writer.Write("<pre>");
    writer.RenderBeginTag(HtmlTextWriterTag.Span);
    writer.Write(this.Text);
    writer.RenderEndTag();
    writer.Write("</pre>");
}
```

Listing 4 Example of Sprout a method
We can now test the new method in the automatic test framework, and be sure that the new code renders the MonospaceLabel as required.

```csharp
[TestMethod]
public void RenderMonospaceLabel_MonospaceLabelDesign2006_RenderSuccessfully()
{
    const string expected = "<pre><span title=""\"">Test Text</span></pre>";

    DbgLabel sut = new DbgLabel();
    sut.LabelType = LabelTypes.Monospace;
    sut.Text = "Test Text";

    StringWriter stringWriter = new StringWriter();
    HtmlTextWriter writer = new HtmlTextWriter(stringWriter);

    // Act
    sut.RenderMonospaceLabel(writer);
    string actual = stringWriter.ToString();

    // Assert
    Assert.AreEqual(expected, actual);
}
```

Listing 5 Test of the “sprouted” method

To avoid making the new method public in order to test it or using other techniques to make it accessible, the “Sprout a class” technique can be used.

We create a new class having the responsibility of rendering the Monospace label type.

```csharp
class MonospaceLabel
{
    private string text;
    private string toolTip;

    public MonospaceLabel(string text, string toolTip)
    {
        this.text = text;
        this.toolTip = toolTip;
    }

    public void Render(HtmlTextWriter writer)
    {
        writer.AddAttribute(HtmlTextWriterAttribute.Title, this.toolTip);
        writer.Write("<pre>");
        writer.RenderBeginTag(HtmlTextWriterTag.Span);
        writer.Write(this.text);
        writer.RenderEndTag();
        writer.Write("</pre>");
    }
}
```

Listing 6: Example of Sprout a class
An advantage with the ”Sprout a class” method, is that it makes the design better with smaller classes, with a single responsibility.

We can now write a simple test of the new class testing the Render method.

```csharp
[TestMethod]
public void Render_MonospaceLabelWithTooltip_RenderSuccessfully()
{
    const string expected = "<pre><span title="Tooltip">Test Text</span></pre>);

    MonospaceLabel sut = new MonospaceLabel("Test Text", "Tooltip");
    StringWriter stringWriter = new StringWriter();
    HtmlTextWriter writer = new HtmlTextWriter(stringWriter);

    //Act
    sut.Render(writer);
    string actual = stringWriter.ToString();

    //Assert
    Assert.AreEqual(expected, actual);
}
```

To use the MonospaceLabel class we create an instance and call the Render method.

```csharp
    case LabelTypes.Monospace:
            var label = new MonospaceLabel(this.GetEncodeText(), this.ToolTip);
            label.Render(writer);
            break;
```

Again you will not get the insertion point under tests, but you will at least get the new class under test control with a few simple steps.

Instantiating the label object is from a design point of view not wishable, but according to Feathers the end justifies the means.

You can say that the above-mentioned techniques have nothing to do with getting the Gemini system under test control, but it is a way to secure that we are getting started writing unit tests.

### 4.7.2 Obstacles in the codebase in terms of Feathers’

As shown in chapter regarding the Gemini System, the Gemini System consists of many parts, and those parts have different obstacles, an example could be the webcontrols inheritance from the Asp.Net framework.

Below I will look at the general obstacles and not so much look at the different parts of the Gemini System. I will describe how they can be attacked by Feather techniques. We are with Gemini working in the Microsoft .Net framework using the Visual Studio development environment and during the description; I will look at how the Visual Studio tools can help in the use of Feathers techniques.
4.7.2.1 “Dependencies on Libraries Are Killing Me”

Problem:
The code you have to test is tightly coupled to some code that you cannot change and are unable to replace with a test double, which makes it difficult to get tests in place.

According to Feathers if possible you should try to decouple the dependencies to the framework by isolating the Framework behind interfaces. This will make it possible to replace the framework both in the test situation, but also in the situation where you wish to replace the framework with another.

If we look at the Gemini use of the Asp.net framework I don’t think that it is possible and worth the effort trying to do this.

The Asp.Net framework is an example of inversion of control, we let the framework control the life cycle of the Asp.Net controls that we develop and therefore we cannot avoid that the framework methods will play a central role in the controls.

Instead of trying to avoid the dependency to the Asp.Net Framework, I think we should decide to do as little of our coding as possible in the classes derived from the framework and delegate the functionality to separate classes. This will make the functionality testable by testing the separate classes without the framework

Solution:
Especially the Gemini web controls are tightly coupled to the Asp.net Framework.

The coupling is through inheritance, having most of the Gemini web controls inheriting from the Asp.Net framework classes and overriding protected methods.

If we do have to have code in the methods derived and overridden from the framework, we can use a specific pattern to get these methods under test.

This pattern is known as “Test specific subclass” [Osherove 2009] or Feathers “Subclass and override method (A4)”.

The steps for “Subclass and override method”:

1. Identify the method that holds the dependency that you want to separate or the place you want to sense.
2. Make the method overridable. In C# it must be virtual.
3. If needed change the access modifier of the class. Could be from private to protected.
4. Create a subclass that overrides the method.

The pattern tells us to introduce a test specific class inheriting from the class that we are going to test. By doing this we enables testing without adding any additional code to the production code implementation, all we have to do is open the method for overriding. The test specific class we can now create an instance of in our automatic test framework and call the method that we want to test through a method in out test specific class.

Using the .Net framework a special class is available for calling private or protected methods for testing, namely the .Net PrivateObject wrapper.

I believe we can use the PrivateObject in making the Asp.Net framework methods callable but shall strive to avoid testing private methods and only use this possibility when testing the Asp.Net control lifecycle methods.

*Example:*

An example is creation of a test specific class for the DbgLabel control, which is a derivation of the Asp.Net Label control.

This use of the pattern enables us to write unit test against the protected Render method.

Instead of making the method override and create a new method for calling the overridden method, we use the possibility in c# of hiding and make the new method call the protected method in the base class.

```csharp
{
    new public void Render(System.Web.UI.HtmlControlsTextWriter writer)
    {
        base.Render(writer);
    }
}
```

*Listing 7: Test specific class for the Dbglabe class*

That we have to open the method for overriding can seem a violation of the restriction decided in the first place, but Feathers recommend this in order to make the code testable and recommend controlling misuse by having described code conventions.
4.7.2.2 “The Undetectable Side Effect”

Problem:
The problem with classes with a lot of instance variables is that it can be difficult to get an overview of the use of the variables and how they are updated. If we in making a change to the class, changes an assignment of an instance variable, it can have side effects.

Testing the separate methods using the instance variable does not necessarily show these side effects.

It will therefore be a good idea to be sure not to have these side effect possibilities implemented.

Solution:
A way to avoid the possibility of implementing code with the risk of side effects is to follow the Command/Query separation design principle.

Command/Query separation design principle
A method should be a command or a query, but not both. A command is a method that can modify the state of the object but that doesn’t return a value. A query is a method that returns a value but that does not modify the object.

[Feathers 2004]

Using this principle, we achieve that we with confidence can call any query method several times without causing side effects.

Being not a part of Feathers’ advice, but being an add on to the Command/Query separation principle, is the principle of keeping the query methods as pure functions if possible, making the methods use of instance variables visible and variable by passing the needed instance variable values as parameters to the query method.

Having the qualities of a pure function:

- No side effects. The pure function does not change any variables or data of any type outside of the function.
- Consistent. The pure function, given the same set of input data, will always return the same output value.

Makes it easier to test the method given quality number two.
**Example:**

An example from the DbgLabel class is the use of the property: Text.

The property can be used from the outside but is also used and assigned several places in the class itself.

Three times we see this statement in the class.

```csharp
this.Text = GetEncodeText();
```

The `GetEncodeText()` method is from the name clearly a query method, and it is, but the way it is used, it could also have been a command method setting the value of the Text property.

```csharp
private string GetEncodeText()
{
    if (this.Text.StartsWith(" ") & this.Formatter.FormatString.Trim != TrimType.Left)
    {
        if (!String.IsNullOrEmpty(this.Text.Trim()) & !this.LabelType == LabelTypes.Nospace)
            return GetSpaceEncodedText() + HtmlUtility.HtmlEncode(FormattedText.TrimStart(), true);
        else
            return HtmlUtility.HtmlEncode(FormattedText, true);
    }
    else
        return HtmlUtility.HtmlEncode(FormattedText, true);
}
```

This is a good example of a use of instance variables and properties in a way that are difficult to cope with.

The method uses another property: FormattedText that is a formatted version of the Text property and another method GetSpaceEncodedText that also uses the Text property.

When setting the value of Text, which is used by both the method and the FormattedText property, we clearly have a possible side effect.

To avoid the side effects we can do two things:

Instead of referring the Text property we can invoke the method in places, where we know we have to use the encoded text.

We can use “Parameterize method (A5)” technique, and deliver instance variable values as parameters.

The method can be rewritten in this way:
We do now a method without side effects, and a pure function, which always returns the same result, given the same input.

4.7.2.3 “The Hidden Dependency”

**Problem:**

The fact that a constructor creates a new object or depends on other resources is what Feathers calls “The hidden dependency”.

The overall problem with “The hidden dependency” is that it creates a very tight coupling between the object and the creator, thus making them extremely sensitive to changes in each other.

If a constructor creates a lot of new objects, we have what Feathers’ calls a construction blob.

**Solution**

The solution is to use constructor dependency injection. In order to enable dependency injection we have to replace the constructor object creation with a constructor parameter.

We do this by the use of the technique:

“Parameterize constructor (A6)”

The steps for “Parameterize Constructor”:

1. Identify the constructor that you want to parameterize and make a copy of it.
2. Add a parameter to the constructor for the object whose creation you are going to replace. Remove the object creation and add an assignment from the parameter to the variable that holds the object.
3. If possible remove the body of the old constructor and replace it by a call to the new constructor. If not possible you may have to extract duplication among the constructors to a new method.

There are other solutions to this problem, but I prefer the "Parameterize constructor", because it closest to the compositional design approach by implementing the strategy pattern.

**Example:**

In Gemini I haven’t found and exact example of this problem, but we have a lot of examples of this hidden dependency problem in our use of static factory methods.

An example of this hiding is in the constructor of the ActionInvoker class

```java
public ActionInvoker(IAction action)
{
    _action = action;
    _actionSA = ServiceAgentFactory.CreateActionServiceAgent();
}
```

**Listing 8: Using a static factory method for object creation**

The instance of the _actionSA instance variable is created using the ServiceAgentFactory factory method which is defined as a static method holding the object creation

```java
public static IActionServiceAgent CreateActionServiceAgent()
{
    return new ActionServiceAgent(  
        EndpointConfigurationFactory.GetEndpoint("ActionService", ServiceConfigurationSingleton.Instance));
}
```

**Listing 9: Static factory method**

Following the steps for “Parameterize Constructor (A6)” provide us with a new constructor
public ActionInvoker(IAction action, IActionServiceAgent actionServiceAgent)
{
    _action = action;
    _actionSA = actionServiceAgent;
}

public ActionInvoker(IAction action)
    : this(action, ServiceAgentFactory.CreateActionServiceAgent())
{
}

Listing 10: New Constructor open for dependency injection

The newly created constructor for the ActionInvoker class enables us to create an instance in our automatic test framework, replacing the object creation of the static factory method with a test double.

4.7.2.4 “The Irritating Global Dependency”
The overall problem with global dependencies in the context of writing unit tests is that they do not fit into the concept of testing a separate unit and they are hard to fake in a unit test arrangement.

There is also a risk of transferring global value from one unit test to another, making it impossible to predict the result.

4.7.2.4.1 Singletons

Problem:
Singletons are classes build on the Singleton pattern and used to permit only one instance of a class in an application. In c# the Singleton pattern is one of the mechanisms used to make global variables.

The problem with singletons is that they are hard to replace with test doubles in a unit test arrangement.

Solution:
Feather’s presents 2 techniques for dealing with Global objects in form of Singletons.

"Introduce instance delegator (A8)"

And

"Introduce static setter (A7)"

The steps for introducing a static setter are the following:

1. Decrease the protection of the constructor so that we can create a fake object by subclassing the Singleton.
2. Add a static setter method to the Singleton.

**Example:**

In Gemini we have an example of a Singleton in the Logging part of the system. To ease the availability of the Logging system to all classes, the MessageHandler class is implemented as a Singleton and can therefore be referenced as a Global object.

The Logging system depends on external resources in order to log the messages. To write unit tests on classes that uses the MessageHandler global object we must be able to break this dependency and inject a MessageHandler test double implementation.

To do this we expose a static setter property to enable replacing the singleton object instance with our fake object.

```csharp
static MessageHandler()
{
    instance = new MessageHandler();
}

private MessageHandler()
{
}

public static IMessageHandler Instance
{
    get
    {
        return instance;
    }
    set
    {
        instance = value;
    }
}
```

**Listing 11: MessageHandler Singleton**

The Singleton already holds its instance as a reference whose type is interface, so we do not need to subclass the MessageHandler, just implement a test double class implementing the interface.

We can now use this static setter method in our unit tests like this
4.7.2.4.2 Static methods and Helper classes

Problem:
Static methods are a natural part of the Singleton pattern, we worked with in previous chapter. But often you also create helper classes with small static helper methods. Often the helper methods are very simple and are used as function definitions, but they can also be used for accessing the Asp.Net session or configuration files like it is the case with Gemini.

The problem with the latter static methods is that we have to replace them with fake implementation in order to avoid including external resources in our test, but static methods cannot be replaced.

Solution:
Feather’s presents a technique for dealing with static methods in helper classes.

"Introduce instance delegator (A8)"

The steps for introducing instance delegator are the following:

1. Identify a static method that is problematic to use in test.
2. Create an instance method for the method on the class, give it another name but preserve the signature
3. Find places where the static method is used in the class you have under test. Use the “Parameterize method” (A5) to supply an instance to the location where the static method call is made.
4. Replace the problematic call to the original static method with a call to the delegator on the instance introduced in the last step.
With this technique we wrap the static methods in instance methods an enables replacing the static methods by using an instance of the class holding the static methods and delegating to this instance using the instance methods.

**Example:**

A Gemini example is method `BuildLabelImageRenderLink` on the `DbgLabel` class. The `BuildLabelImageRenderLink` method uses the static method `GetRevisionValue` on `WebResourceUtility` class.

```csharp
public string BuildLabelImageRenderLink(string text, string design, string labelType)
{
    string encoding;
    string encodedText = thisLabelTextEncoder.EncodeText(text, out encoding);

    string parameters =
    "text=" + encodedText + 
    "&amp;design=" + Uri.EscapeDataString(design) + 
    "&amp;labelType=" + Uri.EscapeDataString(labelType) + 
    "&amp;rev=" + WebResourceUtility.GetRevisionValue();

    return this.baseUrl + "/ImageRender.ashx?" + parameters + "}" + encoding;
}
```

Using the technique we create an instance method with a new name `GetSystemRevisionValue` and let this instance method call the static `GetRevisionValue` method. This will mean that we do not break our production code, we still retrieve the same value.

```csharp
public string GetSystemRevisionValue()
{
    return WebResourceUtility.GetRevisionValue();
}
```

In order to inject a fake instance of the `WebResourceUtility` we extract an interface using “Extract interface (A10)”

```csharp
public interface IWebResourceUtility
{
    string GetSystemRevisionValue();
}
```

We can now use the “Parameterize method (A5)” making it possible to supply the `BuildLabelImageRenderLink` method with a fake implementation of the `WebResourceUtility` class in order to test this the method without having external resources in play.
public string BuildLabelImageRenderLink(string text, string design, string labelText, IWebResourceUtility webResource) {
    string encoding;
    string encodedText = this.labelTextEncoder.EncodeText(text, out encoding);

    string parameters = 
    "text" = encodedText + "&amp;design" = Uri.EscapeDataString(design) + "&amp;labeltype" = Uri.EscapeDataString(labelType) + "&amp;prev" = webResource.GetSystemRevisionValue();

    return this.baseUrl + "/[ImageRender.ashx?" + parameters + "&amp;encoding=" + encoding;
}

Listing 13: Replace use of static method with instance reference

4.7.2.5 “This Class Is Too Big and I Don’t Want to Get Any Bigger”

Problem:
Many of the bug fixes and new features that are added to a system only require adding a small count of code lines and maybe a few new methods. Maybe some of the new code must use data from the existing class and methods and the easiest way is to add new code to the existing.

In this way our existing classes tend to end up being very large and having very large methods. The consequence is that it takes more and more time to add new features and to understand how the existing features work.

Solution:
Our classes in the web control layer are quite big, much of the functionality is in the control class itself. The way to solve the problem is breaking down the single classes into smaller classes.

Large classes do often have several responsibilities, and the way to define which classes the single class can be broken into is by looking at responsibilities and striving for fulfill the Single-Responsibility Principle.

Feathers describe a technique to see these responsibilities and to separate the code after responsibilities. The heuristics for the technique are:

- Group methods
  Similar method names etc.
- Look at hidden methods
  Maybe another class is hidden in the responsibilities of the private methods.
- Look for decision that can change
  Decisions in the code, that maybe could change. The database used etc.
• Look for internal relationships
  Are there relationships between certain instance variables and methods.
• Look for the primary responsibility
  Are there methods that falls fare from the primary responsibility of the class.
• When all else fails. Do some scratch refactoring
• Focus on the current change to make, you may see a responsibility that could allow for substitution.

*Example*

We look at the DbgLabel class and conclude “This class is too big and I don’t want it to get any bigger”.

One way of getting a big class separated in to smaller pieces is looking at Responsibilities.

If we look at the class
We can look at the names of the methods to see if we can find some reflections of responsibility.

There are some groups

<table>
<thead>
<tr>
<th>Group</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image link</td>
<td>BuildLabelImageLink</td>
</tr>
<tr>
<td>CSS class</td>
<td>CalculateCSSClass</td>
</tr>
</tbody>
</table>
When having a set of private methods like we have here, Feathers uses what he calls a feature sketch to get overview of, which methods uses which variables or is calling other private methods. We can use the Visual Studio tool “Show on Codemap” to show a similar drawing of the dependencies.

Figure 15: Feature sketch of DbgLabel class, generated by the "Show code on Codemap" tool in MS Visual Studio

As seen from the drawing, the two RenderHeadline methods are called from the RenderGraphic method, which is why I have put those in the same group.

A way making this large class smaller and to have classes that live up to the Single-Responsibility Principle is to create new classes corresponding to the groups listed and move the code to the new classes.

An example extracting one of the methods to a new class is by using the “Breakout method object (A3)” technique.

The steps for “Breakout method object”:

1. Create a class that will house the method code
2. Create a constructor for the class and preserve signatures by using the exact copy of arguments used by the method.
3. For each argument in the constructor create an instance variable and in the constructor assign it the value of the argument.
4. Create and empty method body on the new class
5. Copy the body of the original method into the new method
6. Lean on the compiler to find places where argument references have to be replaced by reference to the new instance variables.
7. When the new class compiles, go back to the original method and change it so that it creates an instance of the new class and delegates its work to it.

Using these steps we work with the GetPresenceMarkup method.

Listing 16: Original GetPresenceMarkup method

```csharp
private string GetPresenceMarkup()
{
    string imageUrl = string.Format("{0}/Resources/DbgStyles/GFX/Presence/1enhdr.gif", Page.Request.ApplicationPath);
    string emailId = this.EmailId.Contains("@") ? this.EmailId : string.Format("{0}@danskebank.dk", this.EmailId);
    string userMarkup = string.Format("<span class="ucspan"><img class="ucimage" alt="" src="{(0)}" onload="IM"
    return userMarkup;
}
```

Listing 17: The new breakout method class

The GetPresenceMarkup method does not have any parameters in its signature but it references instance variables of the original class. To make these references into instance variables we combine the “Breakout method object (A3)” with the “Parameterize method (A5) in order to create our new class.

```csharp
public class RenderPresence
{
    private readonly string presenceId;
    private readonly string emailId;
    private readonly string applicationPath;

    public RenderPresence(string presenceId, string emailId, string applicationPath)
    {
        this.presenceId = presenceId;
        this.emailId = emailId;
        this.applicationPath = applicationPath;
    }

    public string GetPresenceMarkup()
    {
        string imageUrl = string.Format("{0}/Resources/DbgStyles/GFX/Presence/1enhdr.gif", this.applicationPath);
        if (emailId.Contains("@"))
            emailId = string.Format("{0}@danskebank.dk", emailId);
        string userMarkup = string.Format("<span class="ucspan"><img class="ucimage" alt="" src="{(0)}" onload="IM"
        return userMarkup;
    }
```
We can change the original methods to create an instance of the new class and delegate the work to the instance of the new class.

```csharp
private string GetPresenceMarkup()
{
    var renderPresence = new RenderPresence(this.PresenceId, this.EmailID, Page.Request.ApplicationPath);
    return renderPresence.GetPresenceMarkup();
}
```

Listing 18: Delegation to the breakout object

To separate the responsibilities in the big class after being able to get tests in place, the technique recommended by Feathers’ is not one of his own techniques, but the Refactoring technique “Extract class” described by Martin Fowler in the book “Refactoring: Improving the Design of Existing Code [Fowler 1999]”

4.7.2.6 “I Need to Change a Monster Method”

**Problem:**
Large methods are what Feathers calls “monsters”.

The problem with large methods are that they have grown that big that they are hard to analyze in situations of fixing a bug or adding a new feature, this is also called code bloat, when a method is unnecessarily hard to read. [Christensen 2010]

We do not have very large methods in the Gemini system, but we do have methods that could grow big and do have some of the characteristics that very large methods can have according to Feathers.

**Bulleted methods**

Bulleted methods are methods that do several more or less depending things and serves like a pattern for an order of performing tasks.

The problem with bulleted methods is that they very often work at the same local variables, and therefore can be hard to break up into more methods.

**Snarled methods**
A snarled method is large methods with highly indented code. The typical snarled method consists of a large conditional statement.

We want to avoid large conditional statements, considering the “Switch creep” anti pattern.

**Anti pattern ”Switch creep”**

”Switch creep is the tendency that conditional statements become more and more complex as software ages”

[Christensen 2010]

**Solution:**

Many large methods holds both bulleted and snarled parts and an obvious solution is to break them down into smaller methods or even extract parts to new classes.

Again according to Feathers large can mean that the method, like the large classes, covers several responsibilities and an analysis of the responsibilities should also be performed for the large methods.

The way to break the large class into smaller pieces is to use the “Extract method (A5)” technique and maybe the “Extract Class” refactoring method.

**Example:**

In the Gemini DbgAsyncPagePart class, we have in the OnPreRender method a larger conditional statement
Listing 19: Example of a "snarled" method

Here we can use a combination of the “Extract method(A9)” and “Parameterize method(A5)” and create a new method that handles the creation of the “Default button”.

protected override void OnPreRender(EventArgs e) {
    base.OnPreRender(e);
    if (ScriptManager.IsInAsyncPostBack) {
        this.Page.RegisterContainerControl(this);
        this.Page.RegisterMessageControl(this);
        if (this.registerCloseScript)
            RegisterCloseScript();
        if (!string.IsNullOrEmpty(this.DefaultSubmitButton))
        {
            ObgButton defaultBtn: this.FindControl(this.DefaultSubmitButton) as ObgButton;
            // if the button can not be found with this.FindControl we have to conrtuct the button
            // on our self,
            if (defaultBtn == null)
            {
                defaultBtn = ControlFinder.FindControl(this, this.DefaultSubmitButton) as ObgButton;
                if (defaultBtn != null)
                {
                    string defaultBtnID = defaultBtn.ID;
                    ObgControl namingContrainer = defaultBtn.NamingContainer;
                    while (namingContrainer.GetType() != typeof(ObgAsyncPostBack))
                    {
                        defaultBtnID = namingContrainer.ID + this.IdSeparator + defaultBtnID;
                        namingContrainer = namingContrainer.NamingContainer;
                    }
                    this.DefaultSubmitButton = defaultBtnID;
                }
            }
            SetDefaultSubmitButton(defaultBtn, true);
        }
    }
}
We have now changed our snarled method into two more reader friendly methods. Changing the access modifier of the create button class and/or using the “Extract and override method (A4)” technique now enables us write a test for the CreateDefaultButton method.

### 4.7.2.7 “The Hidden Method”

**Problem:**

“The Hidden Method” of a class is the private method, that because of the restrictions on the access modifier cannot be called from the automatic test framework.

Some private methods of a class can be just helper methods and seen as implementations of a utility, that with no problem can be made public and thereby testable.

Other private methods can be tested through the public methods of the class.

Other private methods again can hold advanced logic or dependencies on other objects that make them candidates for extraction into a new class, taking responsibility for this part of the system logic.

**Solution:**

The methods that are candidates for new classes can be extracted using the following techniques:
• “Breakout method object (A3)”
• “Extract class” refactoring method

**Example:**

Many of the Gemini classes hold a lot of private methods. Many of these methods are small methods that can be tested through test of the public methods that calls them, but there are also private methods that should be tested by themselves because they hold advanced logic or depends on other objects.

The use of Feathers techniques for solving these problems, the “Breakout method object” can be used the same way as with “This class is to Big and I don’t want it to get any bigger”.

### 4.8 Reflections on the approach

During the use of the techniques, breaking dependencies and writing test, I have realized that breaking dependencies and writing tests is an iterative process. When you have broken one dependency in order to get tests in place, you realize that in order to get the test to run, another dependency have to be broken.

Feathers approach is obviously build on the thesis, that we start getting the Legacy code under test control, and then afterwards looks at the design and the scope of the class and methods that may have been broken to accomplish the wish of getting the code under test control.

Even though he look at the separation of the code according to responsibilities found, like described in the chapter “This Class is too big and I don’t want it to Get Any Bigger”. I think either we must have the flexible design principles in mind when doing this separation or reconsider the separation after having the code under test control.

### 4.9 Implementing a new feature

In this chapter I will implement a new feature into a web control rendering a label.

To do this I will use “The Legacy Code Change Algorithm” as described by Feathers.

#### 4.9.1 The Gemini Label control

The Gemini Label control is a webcontrol inheriting from the ASP.NET Label control. The control implements formatting and rendering of a text string into html format. A special functionality is that it can also render a
link to a picture generator, which generates a picture containing the text string.

I have chosen the Label Control because it is considered being a fairly simple control, but when you look at the code it is pretty hard to get an understanding and an overview of the code and the connections between variables and methods.

Apart from some Asp.net Framework classes and helper methods in a helper class, the functionality is within one class, the DbgLabel class.

I wish, in conjunction with implementing the new feature, to do some refactoring in order to get as much of the DbgLabel control as possible under test control.

4.9.2 Getting Gemini Label control under test control and adding the feature

The new feature to implement is a new Label type called “Module Header”. The only special with the “Module Header” is that it has to render with a CSS class of name “Module Header”.

4.9.2.1 Identify change points

The main functionality of the DbgLabel class is concentrated in the Render method and methods called from within this method.

4.9.2.2 Find test points

With the concentration of the functionality in the Render method, this method will also be our starting test point.

4.9.2.3 Break dependencies

The Render method is an override of the inherited Asp.net framework Render method. The framework Render method is defined with protected accessibility, and can therefore only be reached from a derived class.

I will use the possibility of creating a test subclass and breaking the dependency by letting the test subclass call the base implementation of the Render method.

    {
        public void Render(HtmlTextWriter writer)
        {
            base.Render(writer);
        }
    }

Another dependency that we have to break is the dependency on the static methods of the WebresourceUtility helper class.
We can remove these dependencies on the configuration items by using “Replace Global References with Getter (A11)”

```csharp
protected virtual string GetBaseUrl()
{
    return WebResourceUtility.BaseUrl;
}

protected virtual string GetRevisionValue()
{
    return WebResourceUtility.GetRevisionValue();
}
```

We can now again use the “Subclass and override Method (A4), adding the two methods to our test specific class. (DbgLabelTestControl).

### 4.9.2.4 Write tests

Write Charactization tests to preserve existing behavior

Using the DbgLabelTestControl class I can write the first characterization tests of the Render method

```csharp
[TestMethod]
public void Render_HeadLineDesign2000_RenderSuccessfully()
{
    // Arrange
    const string expected = "<span title="\" class="PageHeading\">Test Text</span>");

    DbgLabelTestControl sut = new DbgLabelTestControl();
    sut.LabelType = LabelTypes.Headline;
    sut.Text = "Test Text";

    StringWriter stringWriter = new StringWriter();
    HtmlTextWriter writer = new HtmlTextWriter(stringWriter);

    //Act
    sut.Render(writer);
    string actual = stringWriter.ToString();

    //Assert
    Assert.AreEqual(expected, actual);
}
```

Listing 20 Example of unit test using derived test control

In order to find the expected result, I started running the test without the expected value initialized, I see test fail and the actual result. Afterwards I copy the actual result to the initial value of the expected variable and see the test succeed.
From the code I have seen that there are 2 types of Labels, known as the Standard Label and the Graphic Label. The first test tested rendering a Standard Label.

In order to Render a Graphic Label, two options must be fulfilled. The design standard must 2006, we can set that arranging the test. Second the configuration file must hold an application setting, allowing the use of graphic labels.

Unfortunately the DbgLabel class references the configuration file directly using the .net helper class, but luckily enough the configuration file reference is encapsulated in a private method.

To break the dependency on the configuration file, I change the accessibility of the method to protected virtual, allowing me to override the method in my test control class.

I override the method in my test control class:

```csharp
{
    private bool graphicLabelEnabled;

    public DbgLabelTestControl(bool graphicLabelEnabled)
    {
        this.graphicLabelEnabled = graphicLabelEnabled;
    }

    new public void Render(System.Web.UI.HtmlTextWriter writer)
    {
        base.Render(writer);
    }

    protected override bool GraphicEnabled()
    {
        return this.graphicLabelEnabled;
    }
}
```

Now I can arrange my test, enabling Graphic Label through my test control constructor parameter.
I run the test, but it fails with a null pointer exception. Rendering a Graphic Label depends on 2 other Configuration file settings, the dependency is in the third level of methods calls from the Render method.

In order to break this dependency I use a combination of the “Replace Global References with Getter (A11)” and the “Subclass and override method (A4)” techniques.

First I replace the reference to the configuration file with a getter method by using the Extract Method refactor functionality in Visual Studio. I now have a method that isolates the reference to the configuration file.

I change the accessibility of the method to protected virtual, which allows me to override the method in my test specific class. In this way I have eliminated the access to the configuration file and through my test specific class I can control the value of the configuration setting.
A way of finding how many tests to write is to look at the code, and if there are code that is not quite understandable, then write a test to understand the different outcome of the code.

Another way is to look at the code coverage by using the Visual Studio tool. The few Characterization tests cover 56% of the DblgLabel class, testing only through the Render method.

Looking at the code executed by the Render method in isolation, the code coverage is close to 100%.

4.9.2.5 Make changes and refactor

Having written the tests covering the existing functionality and tests covering the new feature to implement it is time to implement it.

Even though it is not part of Feathers’ techniques, it can be like mentioned in the reflections on the approach, an idea to do some refactoring moving towards a more flexible design, to achieve an easier implementation of the new feature.
With the experiences of working with the responsibilities of the class and methods in the chapter “This class is too big and I don’t want it to get any bigger” in mind. I will look at the possibilities of doing such refactoring.

To achieve a Flexible design the 3-1-2 process [Christensen 2010] can be utilized using the principles for Flexible design

Principles for Flexible design:

1. Program to an interface, not an implementation
2. Favor object composition over class inheritance
3. Consider what should be variable in your design

In 3-1-2 process we start by considering what should be variable in our design. The new feature we are going to add is a new type of Label, and it seems that what varies is the type of label going to be rendered.

We already have a parametric variability point in our code, namely the switch statement in the render method.

From the switch statement we can see that we have 3 type of label rendering, the Standard label, the Graphic label and the Monospace label rendering, the last one having the code directly in the switch statement.

```java
switch (this.LabelType) {
    case LabelTypes.defaultPageHeader:
        RenderGraphic(writer, "h1");
        break;
    case LabelTypes.localFrontPageHeader:
        RenderGraphic(writer, "h2");
        break;
    case LabelTypes.regionalPageHeader:
        RenderGraphic(writer, "h3");
        break;
    case LabelTypes.contactBoxHeader:
        RenderGraphic(writer, "h4");
        break;
    case LabelTypes.Headline:
        RenderGraphic(writer, string.Empty);
        break;
    case LabelTypes.Monospace:
        this.Text = GetEncodeText();
        writer.AddAttribute(HtmlTextWriterAttribute.Title, this.ToolTip);
        writer.Write("<pre>");
        base.Render(writer);
        writer.Write("</pre>");
        break;
    default:
        RenderStandard(writer);
        break;
}
```
To introduce a more flexible design we wish to add new functionality by composition rather than changing the existing parametric variability point.

We can show the idea in an UML diagram:

![UML Diagram](image)

This design means that the DbgLabel let delegate objects do the rendering and uses in this way the Strategy Pattern [Gamma 1993].

To utilize this design, we will have to replace our parametric variability by a variability point that uses the Factory Method pattern [Gamma 1993].

```csharp
protected override void Render(HtmlTextWriter writer)
{
  this.AppendCssClass(this.CalculateCSSClass());
  if (!_enableUserPresence && !string.IsNullOrEmpty(this.EmailID))
    writer.Write(GetPresenceMarkup());

  if (AutoSize)
  {
    this.Width = Unit.Empty;
    this.Height = Unit.Empty;
  }

  var render = CreateLabelRender();
  render.Render(this.FormattedText, this.ToolTip, writer);
}
```

Listing 22: Using the CreateLabelRender factory method
Following up on the 3-1-2 process:

3. We have identified some behavior that is likely to change:
The render of different types of Labels.

1. We have stated a responsibility that covers this behavior and expressed it in an interface: `IHtmlLabelRender`

2. The `DbgLabel` now perform rendering by letting a delegate object do it:
The Label strategy object in form of the different type of Label render classes.

We have now got rid of the switch statement and got easy to read code and we have achieved a design that shows Open/close principle in action, Open for extension, Closed for modification.

For adding the feature we using test driven development, we require a test covering the new `ModuleHeaderLabel` class:

```csharp
[TestMethod]
public void RenderModuleHeaderLabel_ModuleHeaderLabelDesignKey_RenderSuccessfully()
{
   const string expected = "<div title="" class="moduleHeader">\n\n\nTest Text</span>\n\n\n</div>";
   var sut = new ModuleHeaderLabel();
   var stringWriter = new StringWriter();
   var writer = new HtmlTextWriter(stringWriter);
   //Act
   sut.Render("Test Text", ", writer);
   var actual = stringWriter.ToString();
   //Assert
   Assert.AreEqual(expected, actual);
}
```

**Listing 23: Using TDD, writing the test before implementing**

We can now implement the class `ModuleHeaderLabel` class and test that it acts as expected.

To let the `ModuleHeaderLabel` be a part of the `DbgLabel`, we can now change the factory method to create the correct class, specifying a `LabelType` of `ModuleHeader`. 
public IMHtmlLabelRender Create(LabelTypes labelType)
{
    var textEncoder = new LabelTextEncoder();
    var linkBuilder = new ImageRenderLinkBuilder(webResourceUtility.GetStyleBaseUrl1(), webResourceUtility.GetSy
    
    if (labelType.Equals(LabelTypes.Nonospace))
    {
        return new NonospaceLabel();
    }
    if (labelType.Equals(LabelTypes.ModuleHeader))
    {
        return new ModuleHeaderLabel();
    }
    ...

Listing 24: Adding the new label creation to the factory method

We could have considered using an Abstract Factory pattern for creating the LabelRender classes instead of using the Factory Method pattern, this would have given an even more loosely coupled design. But DbgLabel is configured through its properties and created by the Asp.Net Framework and a factory cannot be specified from the outside of the class at runtime. Therefore a Factory Method using the properties assigned is chosen.

4.9.2.6 Observations writing tests for the DbgLabel class
The AppendCssClass method called by Render method, do only take effect if the label are of standard type, because in case of Graphic label type the base.Render isn’t called.

The TextFormatter have no effect on Labels of type Graphic label.

4.10 Do a bug fix

4.10.1 The UserParameterLoader class
In Gemini we have a UserParameterLoader class that gather different user data from the backend systems. The backend systems are reached through a webservice call delivering an array of a UserParameter class holding the user data as a key value pair.

The UserParameterLoader class converts the data into a dictionary of keys and values, where the values are represented as a collection of string values. (a key can have more values connected)

The data are collected from different backend sources and the joined array of parameters from the backend systems can hold two parameters with the same key.

The UserParameterLoader class has to handle this situation by ignoring subsequent parameters with the same key and by publishing a message in log system.
4.10.2 Getting Gemini UserParameterLoader class under test control and fixing a bug

4.10.2.1 Identify change points
In this case the UserParameterLoader class only consists of one public method and two private helper methods and a constructor with no parameters, which means that the change point is in this method. That the constructor has no parameters tell us, that the change cannot be conducted by composition.

4.10.2.2 Find test points
The UserParameter class has only one public method, this method we have to test in order to be sure to preserve the existing functionality. The problem we are concerned about is the conversion from the array to the dictionary. This conversion is placed in a private method, so this method we will also have to test.

4.10.2.3 Break dependencies
Having a constructor with no parameters and a public method with only string parameters, should enable us to instantiate the class and invoke the method from our automatic test framework.

Looking at the code we discover that there are some dependencies that discourage us from invoking the method.

- The private method GetUserSession demands a HttpContext
- The private property EnableDecompression reads the Asp.Net configuration file
- The Load method has a hidden dependency in the instantiation of the UserParameterService.
- The Load method uses the Gemini Logging framework through a static method call. (ExceptionHandler.PublishError(…))
Listing 25: Methods and Properties of UserParameterLoader class

The Load method is not a very large method, but it is a bulleted method which, as mentioned earlier, means that it functions as a template for the order of doing different steps and can hold two or more responsibilities.
• Invoking the UserParameterService webservice
• Building the UserParameterLoader output dictionary

Simplifying the bulleted Load method.

The first technique to use for simplifying a bulleted method is the “Extract method (A9)”. With the in Visual Studio build in support for Extract method refactoring, we can without having tests do this refactoring with confidence in not breaking the code.

We extract each “bullet” into separate methods.

```csharp
public Dictionary<string, ICollection<string>> Load(string currentApplicationClass, string lid, string lidValue)
{
    Dictionary<string, ICollection<string>> parameterDictionary = new Dictionary<string, ICollection<string>>(StringComparer.OrdinalIgnoreCase);
    UserParameterService.UserParameter[] parameters = null;
    parameters = this.GetUserParameters();
    if (parameters != null)
    {
        this.BuildParameterDictionary(parameters);
    }
    else
    {
    }
    return parameterDictionary;
}

private Dictionary<string, ICollection<string>> BuildParameterDictionary(UserParameterService.UserParameter[] parameters)
{
    Dictionary<string, ICollection<string>> parameterDictionary = new Dictionary<string, ICollection<string>>(StringComparer.OrdinalIgnoreCase);
    foreach (var parameter in parameters)
    {
        string key = parameter.Name;
        string[] values = parameter.Value.Split(';', ');
        ICollection<string> value = new List<string>(values);
        parameterDictionary.Add(key, value);
    }
    return parameterDictionary;
}

private UserParameterService.UserParameter[] GetUserParameters()
{
    UserParameterService.UserParameter[] parameters = null;
    UserParameterService.UserParameterService service = new UserParameterService.UserParameterService();
    service.EnabledDecompression = this.EnabledDecompression;
    service.Url1 = String.Format("{0}\UserParameterService.axml", GeminiAppSettings.GeminiServiceBaseUrl); try
    {
        // Initialize session
        Session session = this.GetUserSession();
        string currentApplicationClass = session.ApplicationClass;
        string lidValue = session.LidValue;
        string[] lIds = session.Lids;
        string agreementOwnerId = session.AgreementOwnerId;
        string agreementOwner = session.AgreementOwner;
        string userGM = session.UserGM;
        string userValue = session.UserValue;
        string countryCode = session.CountryCode;
        string languageCode = session.LanguageCode;
        string brand = session.Brand;
        string externalUser = session.ExternalUser;
        string applicationDataString = session.ApplicationDataString;
        string viewer = session.Viewer;
        parameters = service.GetUserParameters(lidValue, lIds, agreementOwnerId, userGM, userValue, countryCode, languageCode, brand, externalUser, currentApplicationClass, applicationDataString, viewer);
    }
    catch (Exception ex)
    {
        ExceptionHandler.PublishError(new ApplicationException("UserParameterLoader: Could not load Parameter Names and Values", ex));
    }
    return parameters;
}
```
Now that we have extracted the two methods we can look at each method.

The BuildParameterDictionary method is quite simple and we can now see that this conversion of a parameter array to a dictionary is the main purpose of this class.

The GetParameters class setup and invoke the UserParameterService webservice. This dependency to a webservice is one of the dependencies that we want to break in order to bring the UserParameterLoader class under test control having in mind that accessing externals can not be part of a unit test.

One way to break the dependency is to use the “Breakout method object (A3)” technique and create a new class holding the method. In this example the GetParametersMethod do not have any parameters in the signature, but normally using this technique, parameters will become parameters to the constructor of the new class.

Looking at the Gemini architecture we can see that the way webservices are used is through a ServiceAgent which setup and handles the webservice call. Wishing to follow this architecture we can call the new class for UserParameterServiceAgent.
public class UserParameterServiceAgent
{
    public UserParameterServiceAgent() { }
}

public UserParameterService.GetUserParameter() GetParameters()
{
    var parameters = service.GetParameters();

    try
    {
        var session = this.GetSession();
        var language = session.Language;
        var customerId = session.CustomerId;
        var currencyCode = session.CurrencyCode;
        var brand = session.Brand;
        var externalCustomerId = session.ExternalCustomerId;
        var applicationDataType = session.ApplicationDataType;
        var viewer = session.Viewer;
        parameters = service.GetParameters(customerId, language, customerId, currencyCode, applicationDataType, viewer);
    }
    catch (Exception ex)
    {
        throw new ApplicationException("
UserParameterServiceAgent: Could not load parameter names and values", ex);
    }
    return parameters;
}

The two private methods are only referred from the GetParameters method, so they can move to the new UserParameterServiceAgent class.

We can now use the new class from the UserParameterLoader

public Dictionary<string, ICollection<string>> Load()
{
    var parameterDictionary = new Dictionary<string, ICollection<string>>(StringComparer.OrdinalIgnoreCase);
    var serviceAgent = new UserParameterServiceAgent();
    var parameters = serviceAgent.GetParameters();

To be able to write a unit test, testing the UserParameterLoader class, using a test double for the UserParameterServiceAgent class, the UserParameterServiceAgent contract must be described by an interface.

We use the “Extract interface (A10)” technique. The technique can be used in two ways. We can use the “Extract interface” built in tool in Visual Studio, marking methods and properties that have to be part of the interface, or we can create an empty interface, change the reference in the UserParameterLoader class, and “Lean on the compiler” to see which methods we have to have in the new interface. We have only one method, so
here it is not a problem, but with large classes with many methods, method number two is good to use in order to minimize the interface.

```csharp
public interface IUserParameterServiceAgent
{
}
```

We now refer the UserParameterServiceAgent by Interface instead of by implementation.

```csharp
public Dictionary<string, ICollection<string>> Load()
{
    var parameterDictionary =
        new Dictionary<string, ICollection<string>>(StringComparer.OrdinalIgnoreCase);
    IUserParameterServiceAgent serviceAgent = new UserParameterServiceAgent();
    var parameters = serviceAgent.GetParameters();
}
```

What we have now is what Feathers’ call a hidden dependency. We create an object of the UserParameterServiceAgent class inside our Load method, which means that we can still not constitute the class with a test double.

The technique to break this dependency is the “Parameterize Method (A5)” or “Parameterize constructor (A6)”. Usually when the object is created inside a method we use the parameterize method technique and the same for the parameterize constructor, but when the creation do not depend of any parameters from the method it can be moved to the constructor and created as an instance variable.

```csharp
private readonly IUserParameterServiceAgent serviceAgent;

public UserParameterLoader(IUserParameterServiceAgent serviceAgent)
{
    this.serviceAgent = serviceAgent;
}
```

```csharp
public Dictionary<string, ICollection<string>> Load()
{
    var parameterDictionary =
        new Dictionary<string, ICollection<string>>(StringComparer.OrdinalIgnoreCase);
    var parameters = serviceAgent.GetParameters();
}
```

Having eliminated the dependency on the HttpContext and the Asp.Net configuration file for the time being by moving the methods to the UserParameterServiceAgent class, we have only to break the dependency on the static method call on the class ExceptionHandler.
Luckily the ExceptionHandler is implemented upon a singleton class implementing the IMessageHandler interface.

```csharp
public static void PublishError(Exception exception)
{
    MessageHandler.Instance.PublishError(ConstEvent.EventId.loggingGeneralError, exception);
}
```

Therefore we can also parameterize this reference by adding a messageHandler parameter to the constructor of the UserParameterLoader class.

```csharp
private readonly IUserParameterServiceAgent serviceAgent;
private readonly IMessageHandler messageHandler;

public UserParameterLoader(IUserParameterServiceAgent serviceAgent, IMessageHandler messageHandler)
{
    this.serviceAgent = serviceAgent;
    this.messageHandler = messageHandler;
}

public Dictionary<string, ICollection<string>> Load()
{
    var parameterDictionary = new Dictionary<string, ICollection<string>>(StringComparer.OrdinalIgnoreCase);
    var parameters = this.serviceAgent.GetParameters();
    if (parameters != null)
    {
        parameterDictionary = BuildParameterDictionary(parameters);
    }
    else
    {
        this.messageHandler.PublishMessage("UserParameterLoader: UserParameterService returned null");
    }
    return parameterDictionary;
}
```

**4.10.2.4 Write tests**

Writing characterization tests to preserve existing functionality

We write a test of the UserParameterLoader load method.
To avoid calling the UserParameterService webservice, we introduce a test double to replace the UserParameterServiceAgent class.

```csharp
[TestMethod]
public void Load_OneParameter_LoadSuccessfully()
{
    // Arrange
    const string expected = "ParamValue";

    var parameter = new UserParameterService.UserParameter {Name = "key1", Value = "ParamValue"};

    var serviceAgent = new UserParameterServiceAgentStub(new UserParameterService.UserParameter[] {parameter});
    var messageHandler = MessageHandler.Instance;

    var sut = new UserParameterLoader(serviceAgent, messageHandler);

    // Act
    var actual = sut.Load();

    // Assert
    Assert.AreEqual(expected, actual["key1"].First());
}
```

To be able control the output from UserParameterServiceAgent test double stub, we implement a constructor parameter that initializes the UserParameter array with the given values.

```
public class UserParameterServiceAgentStub : IUserParameterServiceAgent
{
    private readonly UserParameterService.UserParameter[] parameters;

    public UserParameterServiceAgentStub(UserParameterService.UserParameter[] parameters)
    {
        this.parameters = parameters;
    }

    public UserParameterService.UserParameter[] GetParameters()
    {
        return this.parameters;
    }
}
```

To be able control the output from UserParameterServiceAgent test double stub, we implement a constructor parameter that initializes the UserParameter array with the given values.

Having the UserParameterServiceAgent dependency defined as a parameter for the UserParameterLoader constructor, we are now able to inject the test double into the class under test.

4.10.2.5 Make changes and refactor
Using test driven development solving the bug we now want to write test for the bug that we have to solve.
The new implementation has to write a message to the Gemini log system. In order to sense that this message is written, we have to create a mock of the MessageHandler.

```java
public class MessageHandlerMock : IMessageHandler
{
    private string message;

    public void PublishError(int eventId, string description, Exception exception)
    {
        throw new NotImplementedException();
    }

    public void PublishError(Exception exception)
    {
        throw new NotImplementedException();
    }

    public void PublishMessage(string message)
    {
        this.message = message;
    }

    public void PublishMessage(int eventId, string description, EventLogEntryType logSeverity)
    {
        throw new NotImplementedException();
    }

    public bool VerifyMessage(string message)
    {
        return this.message.Equals(message);
    }
}
```

We implement a VerifyMessage method to verify that the MessageHandler mock is called with the expected message.

We can now write the test that test the new functionality that we going to implement to solve the bug.
We can now use the TDD rhythm and see the test fail.

We implement the change

```csharp
[TestMethod]
public void Load_OneParameter_DuplicateParameterErrorMessagePublished() {
    // Arrange
    const string expected = "UserParameterLoader: Parameter : Key1 not unique. The SI Components :
    var param = new UserParameterService.UserParameter { Name = "Key1", Value = "ParamValue"};
    var serviceAgent = new UserParameterServiceServiceAgentStub(
        new UserParameterService.UserParameter[] { param, parameter });
    var messageHandler = new MessageHandlerMock();
    var sut = new UserParameterLoader(serviceAgent, messageHandler);

    // Act
    var actual = sut.Load();

    // Assert
    Assert.IsTrue(messageHandler.VerifyMessage(expected));
}
```

Run the test list again and see that the failing test now succeeds.

**4.10.2.6 Observations writing tests for the UserParameterLoader class**

Refactoring the code enabling unit test of the UserParameterLoader class, have given us a more flexible design, and a design where we have got a separation of responsibilities. A separation of responsibilities that we should have had in the first place, if we had followed our own design principle of having Service agents in front of our webservice calls.
4.11 The use of Mocking Frameworks
Feathers writes in his book, that there are a lot of mocking frameworks, that can help you implement/generate you stubs and mocking objects.

We have used Rhino mock in the small, and it can really help you in producing mock object for use in your unit tests.

An example on implementing the unit test for the UserParameterLoader could be this. Using RhinoMock to dynamically generate a UserParameterServiceAgent, returning an array of parameters.

```csharp
[TestMethod]
public void Load_OneParameter_LoadSuccessfully()
{
    // Arrange
    const string expected = "ParamValue";
    var serviceAgent = this.mockRepository.Stub<UserParameterServiceAgent>();
    var messageHandler = this.mockRepository.DYNAMICMock< ICommandHandler >();

    var parameter = new UserParameterService.UserParameter { Name = "Key1", Value = "ParamValue" };
    SetupResult.For(serviceAgent.GetParameters()).
        .Return(new UserParameterService.UserParameter[] { parameter });

    var sut = new UserParameterLoader(serviceAgent, messageHandler);
    this.mockRepository.ReplayAll();

    // Act
    var actual = sut.Load();

    // Assert
    Assert.AreEqual(expected, actual["Key1"].First());
}
```

And using RhinoMock to mock and verify the correctness of the call to the MessageHandler

```csharp
[TestMethod]
public void Load_OneParameter_DuplicateParameterFromMessagePublished()
{
    // Arrange
    const string expected = "UserParameterLoader: Parameter 'Key1' not unique. The "
                            + "Size Components col
    var serviceAgent = this.mockRepository.Stub<UserParameterServiceAgent>();
    var messageHandler = this.mockRepository.DYNAMICMock< ICommandHandler >();

    var parameter = new UserParameterService.UserParameter { Name = "Key1", Value = "ParamValue" };
    SetupResult.For(serviceAgent.GetParameters()).
        .Return(new UserParameterService.UserParameter[] { parameter, parameter });

    messageHandler.Expect(m => m.PublishMessage(expected));
    var sut = new UserParameterLoader(serviceAgent, messageHandler);
    this.mockRepository.ReplayAll();

    // Act
    var actual = sut.Load();

    // Assert
    messageHandler.VerifyAllExpectations();
}
```
Using the mocking framework lets you specify the interfaces for which you will have mock objects generated, and let you define as in this example the outcome of the methods that you use from within the system under test (sut).
5 Future work
The work with Feathers’ techniques as a starter for writing unit tests for the existing Gemini codebase have inspired to some points of future work:

- Spread the knowledge of the techniques that matches the problems we have in the Gemini codebase, to get everyone started writing unit tests also for the existing code.
- Introduce a coding standard where we try avoid introducing problems for writing tests, like using static in the way as described earlier.
- Investigate in the possibilities in other automatic test frameworks and mocking frameworks and decide on the one to use moving forward.
- Enforce the usage of test driven development, writing tests first will put the code quality in pride of place and automatically cause a testable design.
- Spread the experience that there are some pros of writing unit test, which you will first discover when started.
  - You will find writing unit tests an interesting way of investigating existing code
  - You will find executing your code using an automatic test framework more effective, than executing through the application that it is part of.
- Agitate that getting a safety net of unit test can take us to a state where we can refactor our code in favor of a more maintainable system.
6 Conclusion
In the analysis I have pointed out the impediments that hold us from getting Gemini under test control and I have described the solutions as proposed by Feathers.

I have tried the techniques out in an example of implementing a new feature and an example of fixing a bug.

In conjunction with adding a new feature I have, with confidence in the established tests, refactored the implementation using a flexible design, changing from a parametric variability handling to a compositional variability handling, gaining the possibility of adding the new feature only changing the factory method moving towards higher maintainability.

Taking the DbgLabel control and a few other classes as an example is not covering all aspects of the Gemini code, but I think that if we look at the classes involved in the other controls and try to see them as separate classes to bring under test control, the use of Feathers techniques can take us to a state where we are much more confident refactoring, adding new features and fixing bugs in the Gemini.

Of course there are different challenges in different parts of the codebase, but all in all the experiments show that we will be able to get large parts of the codebase under test control if we do the effort.

Summing up we can answer yes to the hypothesis: “By the use of Feathers’ techniques, it is possible to bring the Gemini system under test control”.

It will not be easy, but Feathers techniques have the strength that can give us a good start.
7 References


Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides, 1995, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.


Osherove, Roy, 2009, The Art of Unit Testing, Manning

Seemann, Mark, 2012, Dependency Injection in .Net, Manning

Rhino Mock framework:
Rahien, Ayende:
http://ayende.com/wiki/Rhino+Mocks+Documentation.ashx
8 Appendix

This appendix holds a list of Feathers’ techniques and the steps involved in using them. The appendix is used for reference of the techniques, where they have not earlier in the report have been used in an example.

A1: "Sprout a method"

The steps for “Sprout a method”:

1. Identify where you have to make your code change
2. Write a call for the new method that will do the work involved and comment it out
3. Add local variables as parameters to the new method if needed
4. Add return type to method signature if return of a value is needed.
5. Develop the new method using test driven development
6. Remove the comment in the source method to enable the call

A2: “Sprout a class”

The steps for “Sprout a class”:

1. Identify where you have to make your code change
2. Think of a good name for the new class. Write code creating an instance of the new class and write a call for the method in the new class, that will do the work involved and comment it out
3. Determine what local variables from source method you need in the new class and make them arguments to the constructor of the new class.
4. Develop the new class using Test Driven Development.

A3: “Breakout method object”

The steps for “Breakout method object”:

8. Create a class that will house the method code
9. Create a constructor for the class and preserve signatures by using the exact copy of arguments used by the method.
10. For each argument in the constructor create an instance variable and in the constructor assign it the value of the argument.
11. Create and empty method body on the new class
12. Copy the body of the original method into the new method
13. Lean on the compiler to find places where argument references have to be replaced by reference to the new instance variables.
14. When the new class compiles, go back to the original method and change it so that it creates an instance of the new class and delegates its work to it.
A4: "Subclass and override method"

The steps for "Subclass and override method" :

1. Identify the method that holds the dependency that you want to separate or the place you want to sense.
2. Make the method overridable. In C# it must be virtual.
3. If needed change the access modifier of the class. Could be from private to protected.
4. Create a subclass that overrides the method.

A5: "Parameterize Method"

The steps for "Parameterize Method" :

1. Identify the method you want to replace and make a copy of it
2. Add a parameter to the method for the object whose creation or reference you are going to replace. Add an assignment from the parameter to the variable that holds the object.
3. Delete the body of the copied method and make a call to the parameterized method, using the object creation expression for the original object.

A6: "Parameterize Constructor"

The steps for "Parameterize Constructor" :

4. Identify the constructor that you want to parameterize and make a copy of it.
5. Add a parameter to the constructor for the object whose creation you are going to replace. Remove the object creation and add an assignment from the parameter to the variable that holds the object.
6. If possible remove the body of the old constructor and replace it by a call to the new constructor. If not possible you may have to extract duplication among the constructors to a new method.

A7: "Introduce static setter"

The steps for "Introduce static setter":

1. Decrease the protection of the constructor so that we can create a fake object by subclassing the Singleton.
2. Add a static setter method to the Singleton.
A8: "Introduce instance delegator"

The steps for introducing instance delegator:

1. Identify a static method that is problematic to use in test.
2. Create an instance method for the method on the class, give it another name but preserve the signature
3. Find places where the static method is used in the class you have under test. Use the “Parameterize method” (A5) to supply an instance to the location where the static method call is made.
4. Replace the problematic call to the original static method with a call to the delegator on the instance introduced in the last step.

A9: “Extract method”

The steps for ”Extract method”:

1. Identify the code you want to extract and comment it out
2. Think of a name for the new method and create it as an empty method.
3. Place a call to the new method in the old method
4. Copy the code that you want to extract into the new method.
5. Lean on the compiler to find out what parameters you will have to pass and what values you will have to return.
6. Adjust the method declaration to accommodate the parameters and the return value.
7. Delete the commented out code.

A10: “Extract interface”

The steps for “Extract interface”

1. Create a new interface without methods
2. Make the class that you are extracting from implement the interface
3. Change the object reference to reference the interface rather than the type of the implementing class.
4. Compile the system and introduce a new method declaration on the interface for each method use that the compiler reports as an error. (Lean on the compiler)
A11: “Replace Global Reference with Getter”

The steps for “Replace Global Reference with Getter”

1. Identify the global reference that you want to replace
2. Write a getter for the global reference. Make sure that the access protection enable you to override the getter in a subclass
3. Replace the references to the global with calls to the getter.
4. Create a test specific sub class and override the getter.