5DV097 - Degree project in Computing Science Engineering,
30 ECTS-credits
Umeå university, Sweden

Performance and Usability Improvements for Massive Data Grids using Silverlight

Report

Adam Holmström c06ahm@cs.umu.se
2011-04-04

Internal supervisor: Thomas Hellström
External supervisor: Daniel Hellström
Examiner: Fredrik Georgsson

Abstract

TRIMMA Affärsutveckling AB is developing and marketing a business intelligence solution called INSIGHT. INSIGHT presents tables showing possibly very large data sets and the performance and user experience is sometimes suffering. The main objective of this thesis is an evaluation of the pros and cons of replacing the existing ASP.NET/HTML table component in INSIGHT with a component developed in Silverlight.

This thesis examines two techniques to speed up a Silverlight application showing a lot of data: UI- and data virtualization. UI virtualization intends to render only the user interface elements that appear on the screen and are visible to the user, while data virtualization intends to fetch (from the data source) only the section of the data that is visible to the user.

The result of the project is a fully working prototype integrated into a test version of INSIGHT. Performance testing results indicate that the prototype performs approximately the same as the ASP.NET/HTML version of INSIGHT for small tables but significantly better for large data sets.

The prototype also contains a few extra features, not available in INSIGHT, exemplifying the possibilities to create highly responsive user interfaces in Silverlight.

Contents

1 Background

Windows Presentation Foundation (1), WPF, is a new and powerful development framework from Microsoft for development of rich Windows client applications in .NET. WPF is intended to complement and perhaps eventually replace the older Windows Forms technology (2). Silverlight (3), another new technology from Microsoft, is a subset of WPF designed specifically for development of interactive web applications in .NET.

Unlike most other types of web applications, a Silverlight application is running in the user's browser. It gives the developer great potential for creating interactive user interfaces since the application can respond to user actions without having to communicate with the web server. Communication with the web server, through web services, is of course still required to access databases or other types of server resources.

In ASP.NET (4), Microsoft's older Web development platform, a web application runs on the web server and is inherently less responsive. Therefore, JavaScript and AJAX are often used to execute code in the browser and communicate asynchronously with the web server in order to create interactive user experiences in ASP.NET. In Silverlight, all communication with the web server is asynchronous via web services (5), making it easier for developers to create responsive applications.

Unlike WPF, Silverlight runs on both Microsoft Windows and Apple Mac OSX and in most popular browsers, such as Microsoft Internet Explorer, Mozilla Firefox, Apple Safari and Google Chrome (6). Moonlight is an open-source implementation of Silverlight which makes it possible to run Silverlight also on Linux (7). Because Silverlight is running in the browser and also has to be compatible with multiple platforms, not all features of WPF are available in Silverlight.

Silverlight is often compared to Adobe Flash (8), since Flash and Silverlight are both well suited for development of graphically rich web applications. Silverlight includes, for example, animation and hardware accelerated 3D effects (using the GPU) (9).

WPF, Silverlight's big brother, implements a technique called UI virtualization (10). UI virtualization intends to render only the user interface elements (UI elements) that appear on the screen (and are visible to the user), while other elements are created, initiated and rendered when they are needed, e.g. when the user scrolls down a list. Since version 3, Silverlight also supports UI virtualization in some built-in controls, albeit not to the same extent as WPF (11). The purpose of the technology is to decrease rendering time and hence to increase performance of Silverlight/WPF applications.

Another useful technique to speed up an application that displays a lot of data is data virtualization. This technique is similar to UI virtualization but concerns application data instead of UI elements. A common example is a simple list bound to a large collection of items. Whereas UI virtualization only renders the items visible to the user, data virtualization intends to loads only the sections of the collection (the data) that are displayed in the user interface (12). More data is loaded from the server when needed, e.g. when the user scrolls down the list. The intent is to reduce the initial response time by loading fewer items from the server.

There is a clear correlation between response time of websites/web applications and end user experience. End users are rarely willing to wait more than a handful of seconds for a web page to load (13). Reduced load times and improved performance will lead to better usability and thus increase customer satisfaction.

Silverlight is good at much more than graphics. The latest version, Silverlight 4, offers in combination with .NET 4 and the development environments Visual Studio 2010 and Expression Blend 4 lots of features for design and development of modern business applications.

2 Introduction

TRIMMA Affärsutveckling AB (14) is developing and marketing a business intelligence solution called INSIGHT (or INSIKT in Swedish). INSIGHT is a complete system helping customer organizations make better and more informed decisions, by providing support all the way from budgeting and planning to analysis and benchmarking. The application, which is entirely web-based, is based on the business intelligence platform provided by Microsoft (Microsoft Analysis Services).

INSIGHT is based on techniques and products from Microsoft, such as .NET, the web application framework ASP.NET, the database engine Microsoft SQL Server and the OLAP server Microsoft Analysis Services (which is part of Microsoft SQL Server).

Since INSIGHT in many cases presents tables showing very large data sets, performance and user experience is sometimes suffering. TRIMMA is constantly working to improve the user experience in the application and wanted to evaluate the possibility of using Silverlight to improve the performance of INSIGHT. TRIMMA therefore requested an evaluation of the pros and cons of replacing the existing parts of the application that displays large amounts of data with components developed in Silverlight.

TRIMMA also requested a prototype implementation of a table component whose performance can be compared to the performance of the current ASP.NET/HTML solution used by INSIGHT today. Consideration must be given both to the actual performance, such as the time it takes to retrieve and present a certain number of rows from the server side, and perceived performance from a user perspective.

Daniel Hellström, software developer at TRIMMA and former computer science student at Umeå University, has served as external supervisor. The result of the work has been presented for the management and software development teams at TRIMMA.

2.1 Problem Statement

The main tasks of this project were to evaluate

- Silverlight as a technique,
- design patterns for building business applications in Silverlight ,
- integration of Silverlight components in existing ASP.NET applications and
- techniques for enhancing performance in a Silverlight application (mainly data and UI virtualization) and
- to implement a prototype application in Silverlight whose performance can be compared to the ASP.NET/HTML solution employed by INSIGHT today.

The prototype application was supposed to provide functionality similar to that provided by the grid component in INSIGHT.

The advantages and disadvantages of techniques such as data and UI virtualization were also to be documented. The expected result of the work was a fully working and well documented prototype integrated in a test version of INSIGHT along with performance benchmarks.

A good performance benchmark requires knowledge in basic performance testing theory. A short study of performance testing is therefore included as a part of the work.

2.2 Goals

The goals of the project were to achieve good performance in the resulting prototype, gain experience in designing and building business applications in Silverlight and to learn about data and UI virtualization.

TRIMMA's goals with this project were to evaluate pros and cons of using Silverlight generally and in the specific case with a table showing large amounts of data. In the end, there are also plans to replace the existing table component in INSIGHT with a component based on the prototype (or at least inspired by), provided that the advantages of the new solution outweigh the disadvantages, and that performance is noticeably improved.

2.3 Requirement Specification

This section lists functional requirements defined in the beginning of the project that the prototype should satisfy.

- The data grid must be implemented in Silverlight.
- The data grid must be able to show 2D tabular data.
- The data grid must be able to show hierarchical row and column headers.
- The data grid should feature spanning cells.
- The user should be able to resize columns in the grid.
- The data grid must handle complex cell content (e.g. a text box and a list of buttons).
- The data grid must feature context menus on data cells and header cells.
- The Silverlight application shall communicate with a web service running on the web server.

To be able to compare performance with the current version of INSIGHT, the Silverlight grid must be provided with the same type of data. This means that the Silverlight application prototype must be integrated in INSIGHT and that the server implementation supplying the client with data must communicate with an instance of Microsoft Analysis Services.

2.4 Related Work

Techniques to achieve good performance in Silverlight applications are discussed all over the Internet. Microsoft defines UI virtualization briefly (10) and says that it is intended to improve performance but does not present performance benchmarks. A blog post by Bea Stollnitz (11) is a great article on UI Virtualization and its limitations in Silverlight compared to WPF, but it does not contain a performance benchmark.

Since data virtualization is not implemented in Silverlight and WPF, Microsoft does not provide documentation about this feature, although it is discussed in their forums. (12), another blog post by Bea Stollnitz, summarizes data virtualization and points out example implementations, one of which is described more thoroughly in an article by Vincent Van Den Berghe (15). Another simple example implementation of data virtualization is outlined in an article by Paul McClean (16). None of these sources presents figures on how data virtualization affects performance.

The only source of Silverlight versus HTML data grid performance comparisons found during the project is a blog post (17) on MSDN. This post discusses performance characteristics of the Silverlight Toolkit DataGrid and presents a rendering performance benchmark. The benchmark compares HTML and Silverlight (with and without UI virtualization) grid rendering performance as the number of cells in the grid increases. The result is that the Silverlight Toolkit grid performs worse than the HTML table without UI virtualization and that the Silverlight Toolkit grid renders faster than the HTML table for more than 20 rows[1]. For 100 rows the Silverlight Toolkit grid renders in about 3-4 seconds while the HTML table requires about 20 seconds. This benchmark however does only use the Silverlight Toolkit DataGrid supplied by Microsoft and is limited to one test case.

Silverlight is a relatively new technology, but there are many sources describing common design patterns and recommended practice. The most common design pattern on the XAML based platforms, Silverlight and WPF, is Model View ViewModel (MVVM). Two of the first well known descriptions of MVVM can be found in two separate articles in MSDN Magazine from February and March 2009 (18) (19). (18) concerns mainly WPF and a basic framework for building MVVM applications, while (19) is more targeted towards Silverlight. Another well-known MVVM walkthrough is the book Advanced MVVM by Josh Smith (20). The author describes the MVVM design pattern and how MVVM was used to create a simple application.

[1] 70 columns are used in the test.

3 Tools and Techniques

The prototype application has been developed in the .NET 4 environment using Silverlight 4, C#, Visual Studio 2010 and Expression Blend 4. Communication between the Silverlight client and web server is handled with a web service implemented in Windows Communication Foundation (WCF).

This section describes the tools and techniques used in the project.

3.1 Silverlight

Silverlight is a RIA (rich Internet application) web application platform from Microsoft. The applications run in the user's web browser using a browser plug-in (21). According to RIAStats.com, a site providing "a publicly accessible and easily readable set of statics about the proliferation of Rich Internet Application (RIA) players" (22), the Microsoft Silverlight plug-in has a market penetration of more than 60 %.

Silverlight runs on both Microsoft Windows and Apple Mac OSX and in most popular browsers, such as Microsoft Internet Explorer, Mozilla Firefox, Apple Safari and Google Chrome (6). Moonlight is an open-source implementation of Silverlight that makes it possible to run Silverlight also on Linux (7). Unfortunately the Moonlight releases lag behind Microsoft's releases. Currently there is a preview of version 3 of Moonlight, while Microsoft has released the fourth version.

3.1.1 History

Windows Presentation Foundation (WPF) was released with .NET Framework 3. WPF is a new and powerful development framework from Microsoft for development of rich Windows client applications in .NET. WPF is intended to complement and perhaps eventually replace the older Windows Forms technology (2) and introduced and demonstrated the power of the XAML language for defining user interfaces (21).

After the release of WPF, Microsoft started a project called WPF Everywhere, aiming to create a striped down version of WPF running in a browser plug-in (21). This project was renamed to Silverlight and the first version, Silverlight 1, was released in September 2007. The first version focused on media (video, animation and vector graphics) and was not enriched by the power of the CLR and the .NET Framework but instead required the developers to write code in JavaScript (21).

Silverlight 2 was a big step forward since it now built on a version of .NET Framework allowing developers to write code in managed .NET languages, e.g. C# (23). Silverlight 2 was released in October 2008.

With Silverlight 3, Microsoft began to focus on business application development by for example providing richer data binding support, a few more advanced user interface controls, the capability to run applications as a standard application (outside the browser, OOB) and a new framework for communication between Silverlight client and the web server tier (21). Silverlight 3 was released in July 2009.

Silverlight 4 continued in the same direction, by adding for example support for printing, right mouse button events, even more user interface controls and better integration with Visual Studio and Expression Blend (24). Silverlight 4 was released in April 2010.

3.1.2 Features

Silverlight is often compared to Adobe Flash (8), since Flash and Silverlight both are well suited for development of graphically rich web applications. Silverlight includes, for example, animation and hardware accelerated 3D effects (9). Silverlight is good at much more than graphics. The latest version, Silverlight 4, offers in combination with .NET 4 and the development environments Visual Studio 2010 and Expression Blend 4 lots of features for design and development of modern business applications.

A Silverlight 4 client application can be developed in any .NET language (e.g. C# and Visual Basic .NET). User interfaces are defined in a language called XAML (described in section 3.1.4).

The SDK (including the Silverlight Toolkit) features more than 60 customizable user interface controls (24); from simple controls such as text boxes and labels to more advanced controls such as data grids and tree views.

A Silverlight project is compiled into a single file, called a XAP file, with the `.xap` extension. A XAP file is actually a compressed file using the ZIP file format, containing compiled assemblies, XAML files and other resources (e.g. images) (21). Silverlight applications are hosted in HTML pages using an `<object>` tag referencing the XAP file. When a user first visits the web page, the XAP file is downloaded and loaded into the browser plug-in (21).

The Silverlight runtime/plug-in installation file is about 6 MB large (or small) (21) and is everything required to run Silverlight applications in any of the supported browsers.

3.1.3 Silverlight Toolkit

Silverlight Toolkit is an open source project developed by Microsoft containing additional user interface controls. Silverlight Toolkit, hosted on CodePlex, adds a set of more advanced controls (e.g. a data grid/table (`DataGrid`) and a tree control (`TreeView`)) to the basic controls provided in the Silverlight 4 SDK (21).

3.1.4 Extensible Application Markup Language (XAML)

Extensible Application Markup Language (XAML; pronounced *"zammel"*) is a declarative language based on XML allowing developers to structure objects hierarchically (25). XAML can create, initialize and set properties on objects and express relationships between objects (25). XAML is not used only in Silverlight and WPF but also in Windows Workflow Foundation (a Microsoft technique to define workflows) and XML Paper Specification (a Microsoft electronic document format) (21).

In Silverlight (and WPF) XAML is the primary way to define user interfaces (21), similar to HTML for web pages. In XAML, an XML element is an object (an instance of a .NET class) and XML attributes are used to set properties on objects (21).

Code Snippet 1 shows a very simple XAML user interface. The root element is a `UserControl`, which is used to create reusable components encapsulating Silverlight user interfaces (26). The `x:Class` attribute is used to specify the name of the created component (including the namespace). Inside the `UserControl` is a `Grid`. The `Grid` is a layout component organizing children in rows and columns, similar to the HTML table. In this case, the `Grid` has white background, specified with an XML attribute, and a single cell containing a `TextBlock` centered vertically and horizontally. The `TextBlock` is a user interface control showing text. In this example the `TextBlock` displays the text "Hello World!".

```
<UserControl
    x:Class="SilverlightApplication.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid Background="White">
        <TextBlock Text="Hello World!"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" />
    </Grid>
</UserControl>
```

Code Snippet 1. Very simple XAML code defining a user interface containing a text block vertically and horizontally centered in a layout grid.

Event handlers can be attached directly in the XAML file as is illustrated in Code Snippet 2. The `Click` attribute attaches a method called `Button_Click` (in the corresponding code-behind) to the `Click` event of the `Button`. Code Snippet 3 shows an excerpt of the corresponding code-behind file. The `Button_Click` event handler simply changes the `Content` property of the button. The `Content` property of the `Button` controls is what is displayed inside the button, in this case a text string.

```
<Button
    x:Name="Button"
    Content="Click"
    Click="Button_Click" />
```

Code Snippet 2. A button with attached event handler.

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        Button.Content = "Clicked!";
    }
}
```

Code Snippet 3. An example code-behind file written in C#.

3.1.4.1 Content Element Syntax

Some user interface controls accept content between the start and end tags in the XAML code. This content is mapped to a property of the control (21). One example is the `Button` control whose content is mapped to the `Content` property. In Code Snippet 2 the text of the `Button` is set using the `Content` property (the XML attribute called "Content"). In Code Snippet 4 the exact same effect is achieved by setting the content of the `<Button>` XML element. In both cases, the `Content` property of the `Button` is set to "Click".

```
<Button
    x:Name="Button"
    Click="Button_Click">
    Click
</Button>
```

Code Snippet 4. A button with attached event handler using the content element syntax.

Two things doing the same thing might seem unnecessary, but the content element syntax is more powerful than this. In Code Snippet 1 for example, the `UserControl` and `Grid` controls both have complex XAML content that could not have been specified using XML attributes. The same thing applies to the `Button` control; it accepts complex XAML content which makes it very flexible. Code Snippet 5 lists example code for creating a button containing an image and a text block stacked horizontally within a `StackPanel`. Figure 1 shows the resulting button.

```
<Button
    x:Name="Button"
    Click="Button_Click">
    <StackPanel Orientation="Horizontal">
        <Image Source="image.png" Width="16" Height="16" Margin="3" />
        <TextBlock Text="Click" Margin="3" />
    </StackPanel>
</Button>
```

Code Snippet 5. Code defining a button with an image and a text block stacked horizontally.

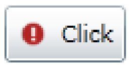


Figure 1. A button with an image and a text block stacked horizontally.

3.1.4.2 Attached Properties

Code Snippet 6 shows another powerful XAML feature. First of all the `Grid` control defines two rows and two columns. The interesting part however is that the child controls (a `TextBlock`, a `TextBox` and a `Button`) all set values of properties called `Grid.Row` and `Grid.Column`. These properties do not exist on the classes themselves, but refers to the parent `Grid`. This type of properties that are assigned a value on one control but defined in another, are called attached properties (21). The `Grid.Row` and `Grid.Column` properties set the cell position of an object within the parent `Grid` layout container.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
         <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
         <ColumnDefinition Width="Auto" />
         <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0"
         Text="Enter Name:" />
    <TextBox Grid.Row="0" Grid.Column="1" />
    <Button Grid.Row="1" Grid.Column="0"
         Content="Submit" />
</Grid>
```

Code Snippet 6. A grid with two rows and two columns exemplifying the use of attached properties.

3.1.4.3 Data binding

Data binding (27) is a technique used to connect properties of data objects to properties of user interface controls. Once the binding is in place, data flow between the two bound properties (27). If a bound property of a data object is updated, the UI elements are updated automatically (27). For two-way data binding data flows in the opposite direction as well; the data object is updated automatically when bound properties of UI controls are updated (27).

Data binding is an example of a markup extension. A markup extension, surrounded by curly brackets, returns a value that will be applied at runtime (21). Another common markup extension is `StaticResource` for assigning a pre-defined and possibly shared resource as the value of a property (21).

A class, whose instances are to be data bound, should implement the `INotifyPropertyChanged` interface. This interface exposes an event named `PropertyChanged` whose sole purpose is to notify a data binding when a property of an instance of the class is updated (27), so that the user interface can be updated automatically.

Code Snippet 7 shows two examples of data binding. The layout grid has two rows. The first row contains a text block displaying "Name:" and a text box two-way bound to the `Name` property of the data object currently in data context. The second row does the same thing for the `Age` property.

This example assumes that the `DataContext` property of the `Grid` is set, in code-behind or by a parent to the grid, to an object implementing `Name` and `Age` properties. The `DataContext` property sets the default source of data bindings of a control and its children. In this case, it is probably an instance of a class representing persons. The text in the text boxes are automatically updated when the bound properties are updated and vice versa.

```xml
<Grid>
    <Grid.RowDefinitions>
         <RowDefinition Height="Auto" />
         <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <StackPanel Grid.Row="0" Orientation="Horizontal">
         <TextBlock Text="Name:" />
         <TextBox Text="{Binding Name, Mode=TwoWay}" />
    </StackPanel>

    <StackPanel Grid.Row="1" Orientation="Horizontal">
         <TextBlock Text="Age:" />
         <TextBox Text="{Binding Age, Mode=TwoWay}" />
    </StackPanel>
</Grid>
```

Code Snippet 7. The text of two text boxes are two-way data bound to the `Name` and `Age` properties of the data object currently in data context.

In Silverlight 4, it is also possible to bind to executable actions (28). An action is in this case a property implementing the `Execute` and `CanExecute` methods of the `ICommand` interface. For example, it is possible to bind the click event of a button to a command on the data object in data context. The `Execute` method of the command will then be executed when the button is clicked, and the button will be disabled when `CanExecute` returns false.

Data binding is particularly important when using the MVVM design pattern.

3.1.5 Competitors

In the four following sections, Silverlight is compared to HTML, ASP.NET, Adobe Flash/Flex and HTML 5.

The reason to use Silverlight in this project was that it was specified as a part of the assignment specification provided by TRIMMA. This in turn depends on that Silverlight is a .NET technology and that the company's development is largely based on .NET.

3.1.5.1 HTML with JavaScript, AJAX and jQuery

HTML is very mature as a development platform and works on almost all Internet-connected devices. This is a great advantage of HTML compared to Silverlight that requires a plug-in that does not even work on all platforms.

Another disadvantage of Silverlight compared to HTML is that the application (the `XAP` file) must be downloaded to the client when it is first viewed (or when the cache has expired).

Plain HTML reloads pages on user interaction and an HTML application is thus generally not as responsive as a Silverlight application, although client-side JavaScript code can be used to improve responsibility (21). In Silverlight, the entire application (written in managed .NET) runs on the client and all service calls are asynchronous (5) by design. Hence a Silverlight application is responsive by default and remains so unless the developer forces the application to wait for service communication.

As mentioned, JavaScript can be used to improve responsiveness of HTML pages. JavaScript is an interpreted scripting language, often embedded in HTML pages, that is used to add interactivity to web pages (29). Therefore, JavaScript can be used to mitigate the limitations of HTML when it comes to responsiveness. AJAX (Asynchronous JavaScript and XML) is another technique (which is based on JavaScript and XML) used to create dynamic HTML pages. Using AJAX, it is possible to exchange data asynchronously with the web server without having to reload the entire page, which in turn makes it possible to refresh only parts of a web page (30).

jQuery is "a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development" (31). In short, it is a JavaScript library aiming to simplify client-side JavaScript programming and therefore make it easier to create responsive HTML based applications.

In summary, there are a couple of techniques for adding interactivity and responsiveness to HTML applications, but still, writing managed .NET code in a Silverlight client is probably easier than writing JavaScript.

Another advantage of Silverlight compared to HTML is that a Silverlight application renders the same regardless of platform (21) (provided that the platform is supported). In HTML, this is not always the case because different browsers (and different versions of the same browser) may interpret the HTML differently (32).

3.1.5.2 ASP.NET

Both ASP.NET and Silverlight are used to develop web applications in .NET. ASP.NET, like HTML, works on almost all Internet-connected devices (21) since the web server running the application returns standard HTML for the web browser of the client to display. Hence, unlike Silverlight, ASP.NET, like HTML, does not require a browser plug-in (21). As a development platform, ASP.NET is also more mature than Silverlight (21).

However, because ASP.NET is based on HTML, ASP.NET applications tend to be less responsive than Silverlight applications (21). As with HTML, JavaScript, AJAX and jQuery can be used to improve responsiveness. The ASP.NET Ajax Control Toolkit provides a set of pre-built controls making it easy to use AJAX in ASP.NET applications (33). Visual Studio 2010 ships with jQuery with support for ASP.NET (34).

It may be worth mentioning that Silverlight applications often are hosted in ASP.NET web applications (although they could be hosted in plain HTML as well).

3.1.5.3 Adobe Flash and Flex

According to (21), Adobe Flash and Flex is Silverlight’s biggest competitor. Adobe Flash is like Silverlight a platform for building RIAs that run in a browser plug-in. Flash is, like Silverlight, good at graphics, animations and media playback. Compared to Silverlight, Flash has broader reach and is almost ubiquitous (21).

Adobe Flex is an SDK from Adobe based on the Flash platform targeted towards development of RIAs (35). The Flex SDK includes a set of standard user interface controls, data binding support, a programming model based on a stateful client as well as ways to communicate with servers in the background (without having to reload the user interface) (35).

Flex uses a XML-based language called MXML to define user interface appearance and behavior, and thus corresponds to XAML in Silverlight. Client-side logic is coded in the ActionScript language (35).

3.1.5.4 HTML 5

A new HTML standard, HTML 5, is under development. The draft of HTML 5 includes support for video and audio playback as well as drag and drop features (36), and there are many voices on the Internet discussing whether or not HTML 5 could be the death of plug-in based technologies as Flash and Silverlight (37).

In a blog post (38) on Silverlight’s official blog, it is argued that HTML 5 will indeed include parts of the functionality (e.g. media support) previously requiring a Flash or Silverlight plug-in, but that Silverlight includes many more features that will not be available in HTML 5. It is also argued that the purpose of Silverlight is not to replace HTML, but to do things HTML cannot do.

In the same blog post, it is claimed that Silverlight has been created and shipped in four major versions during about half the time the new version of HTML has been under design and that Silverlight will have evolved even more when the specification of HTML 5 has stabilized and is fully implemented in all major browsers.

3.2 Windows Communication Foundation (WCF)

This section describes the web service framework, Windows Communication Foundation, used in the project. The section may be hard to understand without some web service knowledge and can in that case be skipped.

Windows Communication Foundation (WCF) is “*a part of the .NET Framework that provides a unified programming model for rapidly building service-oriented applications that communicate across the web and the enterprise*” (39).

A WCF web service results in one or many SOAP web services defined by WSDL interface descriptions. The main approach is to write an annotated interface in C# which represents the service interface and simple C# classes for custom data types required in the interface. The annotation attributes provide WCF with additional information about how to perform the mapping between the object oriented .NET world and the WSDL interface and SOAP messages (40).

The interface annotations specify for example that the C# interface is a service interface, the XML namespace of the interface, which methods of the interface that are to be exposed in the service and the type of each operation (in, in/out, etc.). A data contract is a simple C# class with annotations specifying how this class is to be serialized into SOAP/XML messages (41).

To implement a service, a class implementing the service interface is created. This class can be annotated with instance and concurrency mode attributes, specifying how the service will be handled by the service container (42). A service is normally deployed to IIS (Internet Information Services (43), Microsoft’s web server) for service hosting.

The code first approach described is not the only option in WCF. A contract first approach is also possible by first developing the WSDL interface description and then generating code stubs (for both client and service) using command line utility accompanying the Visual Studio IDE (44) (or a plug-in providing a GUI interface to the command line utility (45)).

Since the service interface is specified in WSDL, for both approaches, any client that meets the contract should be able to interact with the service. In this project, the client (service consumer) has also been developed in Visual Studio which means that client stubs are automatically generated when adding a service reference to the client project.

A WCF service can be exposed using different bindings. “Bindings specify the communication mechanism to use when talking to an endpoint and indicate how to connect to an endpoint” (46). The most common binding is based on plain SOAP XML over HTTP and is suitable when interoperability is prioritized. When all involved parties use WCF (interoperability with other platforms is not required) and performance is critical, this binding can be customized to use binary encoding of message payload. Binary encoding increases performance but sacrifices interoperability (47). The binary encoder encodes the SOAP XML messages binary (47) which makes the messages smaller and thus faster to send.

3.3 Tools

Visual Studio 2010 Professional has been used as the main development environment. Visual Studio 2010 is a full IDE including Silverlight/WPF specific features such as an editable design surface and drag and drop data-binding (24). Expression Blend 4 has in some cases been used for design of Silverlight user interfaces. Expression Blend is specialized in user interface design for WPF and Silverlight and makes the design process easier than using Visual Studio, especially when it comes to animation and advanced customization of user interface controls.

4 Model View ViewModel (MVVM)

Model View ViewModel (MVVM) is a design pattern used by many Silverlight and WPF developers (21). It is an alternative to the code-behind approach common in ASP.NET and Windows Forms as well as Silverlight and WPF. The main purpose of the MVVM design pattern is (as in Model View Controller (MVC) and Model View Presenter (MVP)) separation of concerns between presentation and business logic (21).

MVVM is a widely discussed topic and due to lack of standardization confusion about how to implement it is not unusual (21). Despite the lack of standardization most developers agree that designing an application based on MVVM is good practice (21). Microsoft has also been using MVVM internally on large development projects (such as Microsoft Expression Blend (18)).

This section intends to introduce MVVM and its components as well as pros and cons. The client implementation of the prototype (see 6.3 Client Implementation) is based on MVVM and the principles described in this section.

Note that MVVM is a Silverlight client application design pattern and says nothing about server-side and web service implementation.

4.1 Evolution

Design patterns for creating user interfaces have been around for a long time. Model View Presenter (MVP) is one such pattern that emerged as a variation of the older Model View Controller (MVC) pattern (18). The `View` is what the user sees on the monitor, the data displayed by the `View` is the `Model` and the `Presenter` acts as a bridge between them by populating the `View` with the `Model` data and reacting to user interaction (18).

MVVM can be seen as a version of MVP adapted for the XAML-based platforms Silverlight and WPF. In MVVM the `Presenter` has been replaced by a component called `ViewModel`. The main difference is that the `Presenter` has a reference to and manipulates (and reacts to events in) the `View` while the `ViewModel` does not need a reference to the `View` (18). Instead, the `ViewModel` simply exposes data and actions that the `View` can bind to via data binding (18).

4.2 The Components of MVVM

Figure 2 illustrates the three components of the MVVM design pattern.

- The `View` layer contains the presentation (the user interface).
- The `ViewModel` layer contains client business logic controlling the `Views`.
- The `Model` layer contains business data entities.

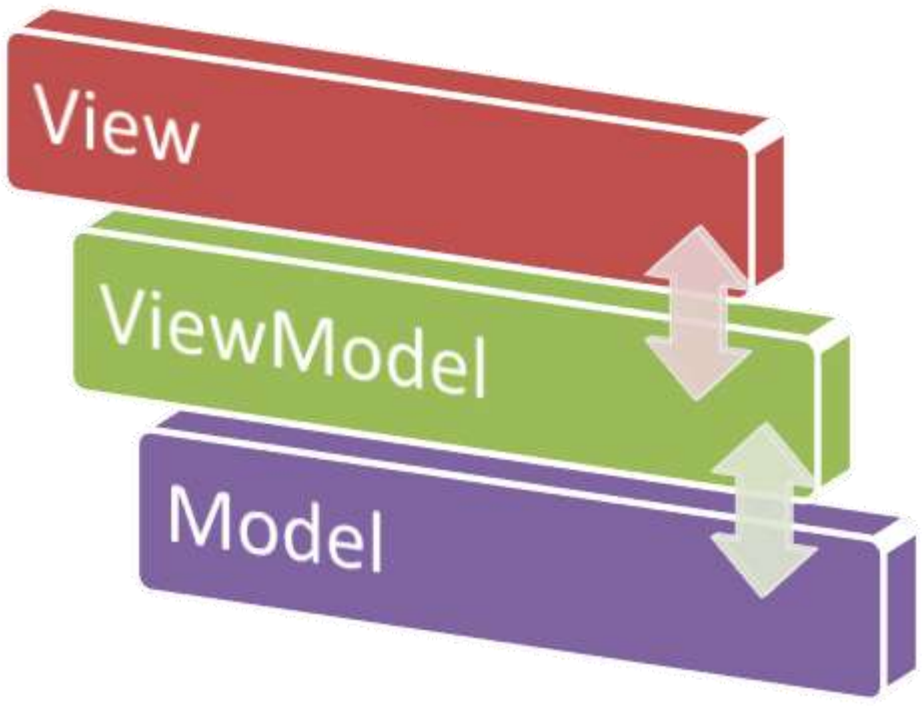


Figure 2. An illustration of the three components of the MVVM design pattern.

4.2.1 Models

`Models` are data entities (objects) that are needed in the client (21). They are usually fetched from one or more web services by one or more `ViewModels`. The data entities are instances of simple classes exposing data (through properties). If a property of a `Model` class is not constant, the class may implement `INotifyPropertyChanged` so that other objects (and `Views`) are notified when the property has changed.

4.2.2 ViewModels

`ViewModels` contain the business logic controlling what is shown in the user interface and what happens when the user interacts with the application. They basically provide the `Views` with information (based on the `Models`) that can be shown in the user interface and commands that can be executed when the user interacts with the `Views`. A `ViewModel` does not know anything about the `View(s)` connected to it, but has to notify interested parties when exposed information changes. (21) describes a `ViewModel` as "a model of the view".

A `ViewModel` exposes information and commands (executable actions) through properties that can be accessed by the `View`. To notify `Views` when one or more properties have changed, the `INotifyPropertyChanged` interface is implemented (18).

4.2.3 Views

A `View` is a XAML-file defining the elements of the user interface (21) and a code-behind file containing presentation specific logic. In the perfect case, the code-behind file is empty or at least almost empty since all logic is placed in the `ViewModel` of the `View` (21).

A `View` uses data binding to bind properties of elements in the user interface, e.g. the text of a text box, to properties of a `ViewModel`. Properties that may be changed by the user interface are bound two-way, so that the corresponding properties of the `ViewModel` are updated when the user interface is updated.

4.3 Benefits

As stated previously, the main purpose of MVVM is separation of concerns. The loose coupling between presentation and logic makes it possible to easily create several `Views` using the same `ViewModel` but presenting data in different ways (21).

MVVM makes it easier for developers and designers to cooperate. The development team can work on the `ViewModel` without interfering with the designer working on the `View` (21).

MVVM may enhance testability since the `ViewModel` is a regular class without UI elements (18). The `ViewModel` can be tested using usual unit tests.

A great benefit with MVVM is that it really unleashes the power of data binding. When a property of the `ViewModel` is updated the `View` is automatically updated as well and vice versa (18).

4.4 Criticism

The most common criticism is lack of standardization (21) and excessive complexity (48). The lack of standardization may cause confusion around implementation details but should not affect the understanding of the basic purpose of the design pattern.

MVVM is certainly not a necessity for small applications and in some cases it may increase complexity, buy as the complexity of the problem and hence the code complexity increases, a design pattern may help organize the code (18).

MVVM is widely discussed and debated (21) and opinions differ. (49) e.g. considers that the responsibilities of the `ViewModel` are too many.

5 Techniques for Improving Performance and Usability in Silverlight

This chapter introduces two performance improving techniques common in Silverlight.

5.1 UI Virtualization

WPF, Silverlight's big brother, implements a technique called UI virtualization (10). UI virtualization intends to render only the user interface elements (UI elements) that appear on the screen (and are visible to the user), while other elements are created, initiated and rendered when they are needed (11), e.g. when the user scrolls down a list. Since version 3, Silverlight also supports UI virtualization in some built-in controls, albeit not to the same extent as WPF (11). The purpose of the technology is to decrease rendering time and hence to increase performance of Silverlight/WPF applications (10).

Lots of controls in a user interface may affect performance, since the default behavior of the layout system in Silverlight and WPF is to create and initialize them all, even though most of them may be out of sight. A common example is a list with a large number of items. The standard layout system will create one layout container for every item in the list and then calculate the layout size and position for all layout containers (10). If the number of items in the list is large, typically only a small subset is visible to the user. With UI virtualization, layout container creation as well as position and size computations are deferred until the item is visible to the user (10), which potentially can reduce rendering time dramatically. The layout containers of the items that are scrolled out of view are destroyed (10).

UI virtualization in Silverlight is implemented in a layout component (class) called `VirtualizingStackPanel` (50). The standard Silverlight `StackPanel` stacks items (its content/children) horizontally or vertically without UI virtualization. `VirtualizingStackPanel` does the same thing with UI virtualization and is intended to improve performance.

5.1.1 Container Recycling

A further optimization of UI virtualization is to recycle layout containers. When an item is scrolled out of view, its layout container can be reused for another item scrolled into view. This means that it is not necessary to create and destroy items as the user scrolls the list (10). Container recycling results in less garbage collection and improved performance when scrolling because of decreased time spent initializing layout containers (11). Container recycling is implemented in the standard Silverlight list control (10).

5.1.2 Deferred Scrolling

The default behavior when a user drags the thumb of a scroll bar is to immediately update the UI (e.g. the list or whatever control it is). Deferred scrolling is a technique used to improve scrolling performance by not updating the UI when the scroll bar thumb is moved. When the thumb is released the UI is updated to reflect the scroll position (10).

The advantage of using deferred scrolling is improved scrolling responsiveness since the elements scrolled by are not rendered. The disadvantage is that the user cannot see the items scrolled through (11).

5.1.3 Example of UI Virtualization with Container Recycling

Figure 3 tries to illustrate UI virtualization with container recycling. The figure illustrates a three step scenario with a list with 30 items. A red dashed border indicates the section of the list visible to the user (the section that fits on the screen). Light blue color indicates that a list item is rendered and kept in memory.

In the first step, *a*, the first ten items are visible to the user and thus only the first ten items are rendered and kept in memory.

In the second step, *b*, the user has scrolled one row downwards. This makes the first list item disappear from the screen (indicated with a light red color) and the eleventh to appear (indicated with a darker blue color). The layout container of the first item is reused when rendering the eleventh item (indicated by the arrow).

In the third step, *c*, the user is assumed to have dragged the thumb of the scroll bar and released in so that items 13-22 are visible. Items 2-11 are not visible anymore and the layout containers used to render them will be recycled when rendering items 13-22. If deferred scrolling was enabled, the list would not have scrolled until the scroll thumb was released which means that the twelfth item (colored light green) never would have been rendered. With deferred scrolling disabled, the twelfth item would have been first rendered and then recycled.

Note that this is a constructed example and that it probably does not work exactly like this in actual implementations.

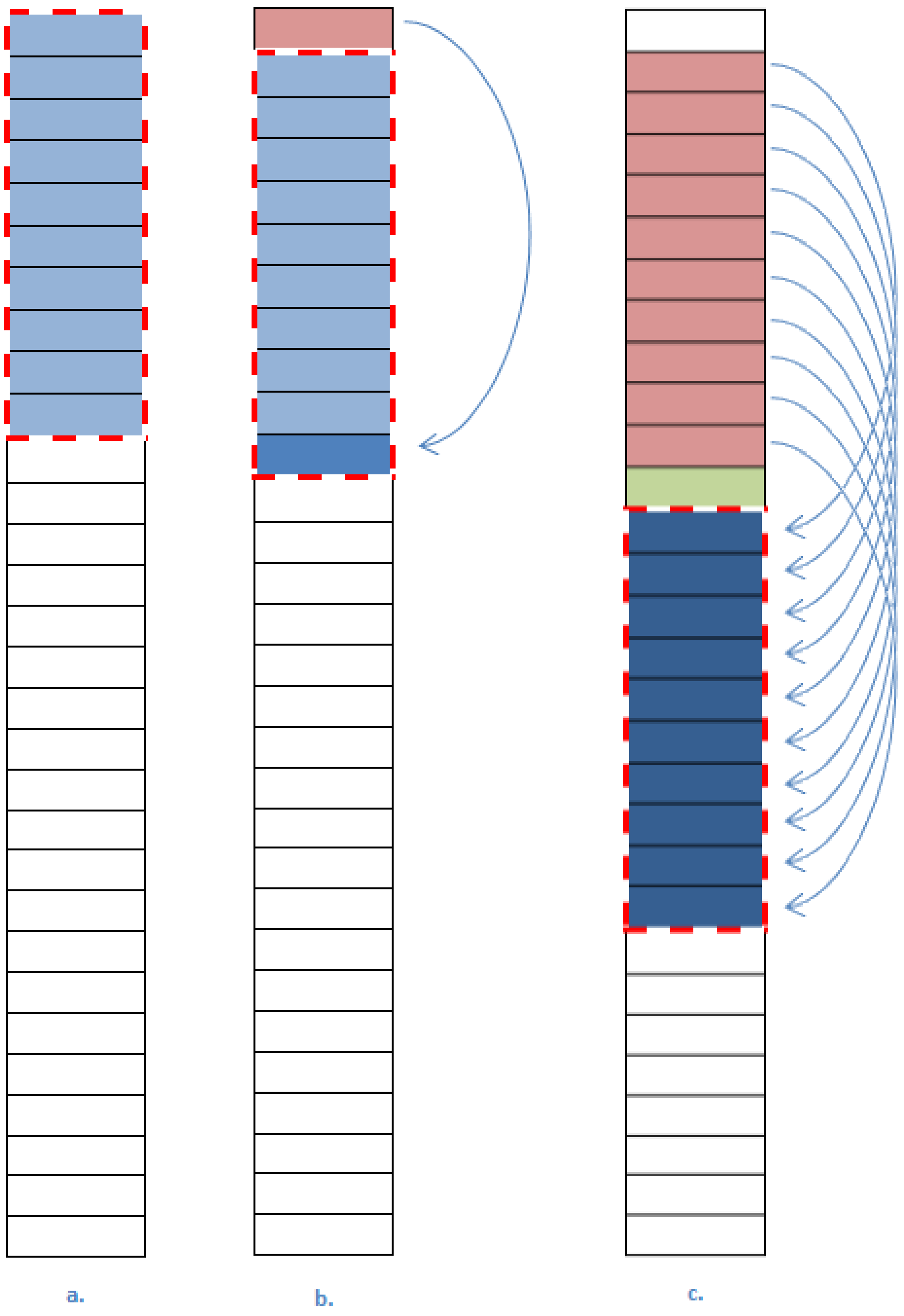


Figure 3. Illustration of UI Virtualization of a simple list.

5.2 Data Virtualization

Another useful technique to speed up an application that displays a lot of data is data virtualization. This technique is similar to UI virtualization but concerns application data instead of UI elements. A common example is a simple list bound to a large collection of items. Whereas UI virtualization only renders the items visible to the user, data virtualization intends to load only the sections of the collection that are displayed in the user interface (12). More data is loaded from the server when needed, e.g. when the user scrolls down the list. The intent is to reduce the initial response time by loading fewer items from the server (12).

There are many ways to implement data virtualization. Data could be fetched synchronously or asynchronously when more data is needed. One alternative is to fetch data only when requested. Another alternative is to fetch all data separated in smaller packages, so that the data in the first package can be displayed once it has been received by the client. In this way, the user can start interact with the application while the rest of the data is still being transferred in the background.

There are a couple of example implementations of data virtualization available on the Internet. Paul McClean outlines (16) a possible implementation of data virtualization adapted for data binding in WPF. McClean makes use of the fact that an `ItemsControl` (e.g. a `ListBox`) bound to an implementation of `IList` only accesses the `Count` property to find out the size of the collection and the indexer to retrieve the items to display. Thus, it is possible to create an implementation of `IList` that virtualizes data by knowing only the number of items in the collection and how to fetch and cache items when requested (through the indexer). Items that have not already been requested will not exist in the cache of the virtual collection, and must be fetched on demand when requested. The `ItemsControl` does not need to know that the collection is virtualized.

McClean's virtual collection implementation divides the collection in pages (continuous segment of the collection) that are fetched to a cache (a dictionary/hash table) when requested and released when no longer required.

When an item from a page already in the cache is requested, it is immediately returned. If the page of the item is not in the cache, the entire page is fetched to the cache. When a page is fetched, the previous or next page may also be fetched, depending on the location of the requested item (in its page). McClean also maintains "last access" time for every page in the cache in order to know when to release a page from the cache.

McClean also outlines an asynchronous virtualized collection based on the same principles. The asynchronous collection also implements `INotifyCollectionChanged` (the interface used by `ObservableCollection`) so that the user interface control bound to the collection is notified when the collection is updated (items are added, updated or deleted).

Vincent Van Den Berghe has another approach to data virtualization in WPF (15). This solution is based on a generic wrapper class called `DataRefBase<T>` that wraps the items in the collection. A wrapper object holds a reference to the "real" object, and knows how to fetch it. The generic wrapper also exposes all the properties of the "real" type using dynamic type information and implements `INotifyPropertyChanged`, so that the user interface control bound to the collection of "fake" objects will not notice any difference[2]. The wrapper objects are initially empty and fetch the real objects when requested.

In a second step, the wrappers' references to the "real" objects are swapped for weak references[3] allowing the garbage collector to free memory when needed. Van Den Berghe emphasizes that using the garbage collector for caching is not a good solution and that sorting this collection can be very slow (since objects may be loaded several times). Van Den Berghe, like McClean, also provides a solution loading items asynchronously without blocking the user interface.

Van Den Berghe virtualizes the list items as outlined above, but the collection still has references to all wrappers, even if they are empty. For a collection of 1 million objects for which 50 items have been loaded there will be 999 950 empty wrapper objects. Also, the wrappers never release the "real" objects (unless weak references are used) so they may all be in memory at the same time.

Van Den Berghe also implements list virtualization in a solution similar to McClean's, also making use of the behavior of an `ItemsControl` when bound to an `IList` implementation. The solution is a move to front (MTF) list cache of pages which releases the pages in the end of the list when the maximum cache size has been exceeded. Van Den Berghe's solution also adds support for sorting and filtering the virtualized collection.

In a blog post by Bea Stollnitz (12), McClean's and Van Den Berghe's solutions are compared and evaluated. Stollnitz also explains how the two solutions work with Silverlight and concludes that Van Den Berghe's solution requires several features not available in the Silverlight version of .NET while McClean's does compile with minor changes.

Although McClean's solution is compatible with Silverlight, it doesn't work as expected. The data binding behavior of the `ItemsControl` in Silverlight differs from its counterpart in WPF. The `ItemsControl` implementation accesses all items in the collection at load time which practically knocks out the data virtualization.

[2] Data binding uses only the name of the property; the type does not matter.

[3] A target of a weak reference may be collected by the garbage collector even though weak references to it still exist.

However, it's important to realize that McClean's data virtualization implementation may work perfectly with other Silverlight controls.

The data virtualization implementations discussed in this section are adapted for data binding in WPF and Silverlight, but could be used for other scenarios as well.

6 Implementation

This section describes the implementation of the Silverlight grid prototype.

6.1 Overview

The system consists of two parts; a server and a client. The first part, the server, runs on a web server and provides the second part, the client, with data via a web service. The server handles business logic as well as fetching data from the associated data source. The main server business logic can be found in the so called coordinator. The coordinator is hidden behind a layer of facades with the intent to provide a simple interface to the server logic. The service logic layer uses a data access layer to access the data sources.

The client is responsible for user interface presentation and communicates with the server via the web service. The client is implemented according to the MVVM design pattern.

Both the client and the server (from the coordinator and up) works with a so called presentation model consisting of data transfer objects (DTOs). These objects represent the data processed by the application. Figure 4 is an illustration of the system architecture.

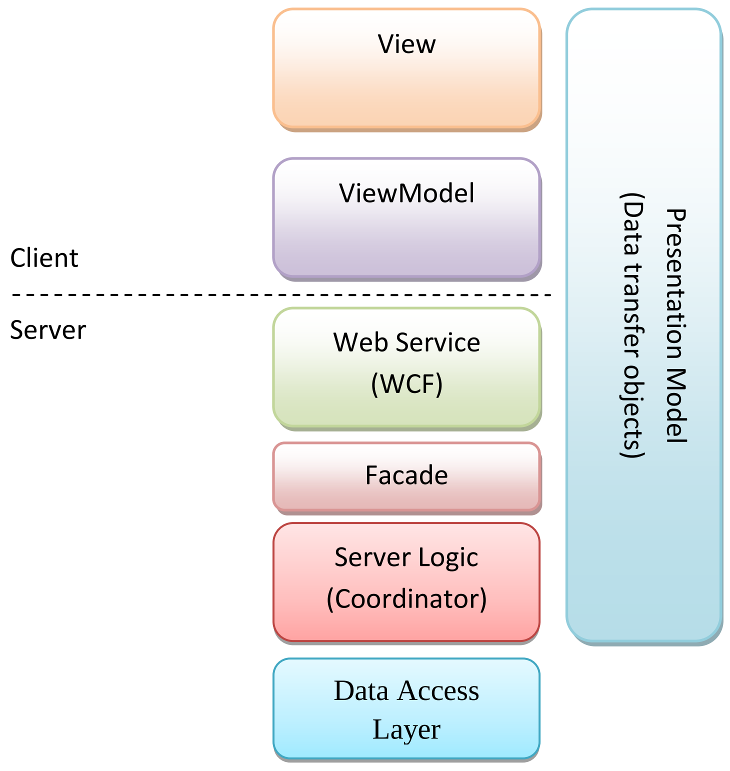

View
ViewModel
Client
Server
Web Service (WCF)
Facade
Server Logic (Coordinator)
Data Access Layer
Presentation Model (Data transfer objects)

Figure 4. An architectural overview of the system.

6.2 Server Implementation

The server implementation is based on so called data transfer objects (DTOs). A data transfer object is an instance of a class responsible for holding structured information (i.e. simple data container) communicated between components on the server and between the server and the client. The server logic manages these objects and exposes them to the client via the web service.

6.2.1 Data Transfer Objects

As mentioned, the data transfer objects are simple data containers that hold the data processed by the system. The main data structures are the ones representing the grid/table to present in the client. These DTOs are shown in Figure 5. The main class is the `Grid` class. The grid has a collection of `GridDataRow` objects representing the rows in the grid. A `GridDataRow` has a collection of `GridCell` objects representing the cells of the row (one cell per column in the grid).

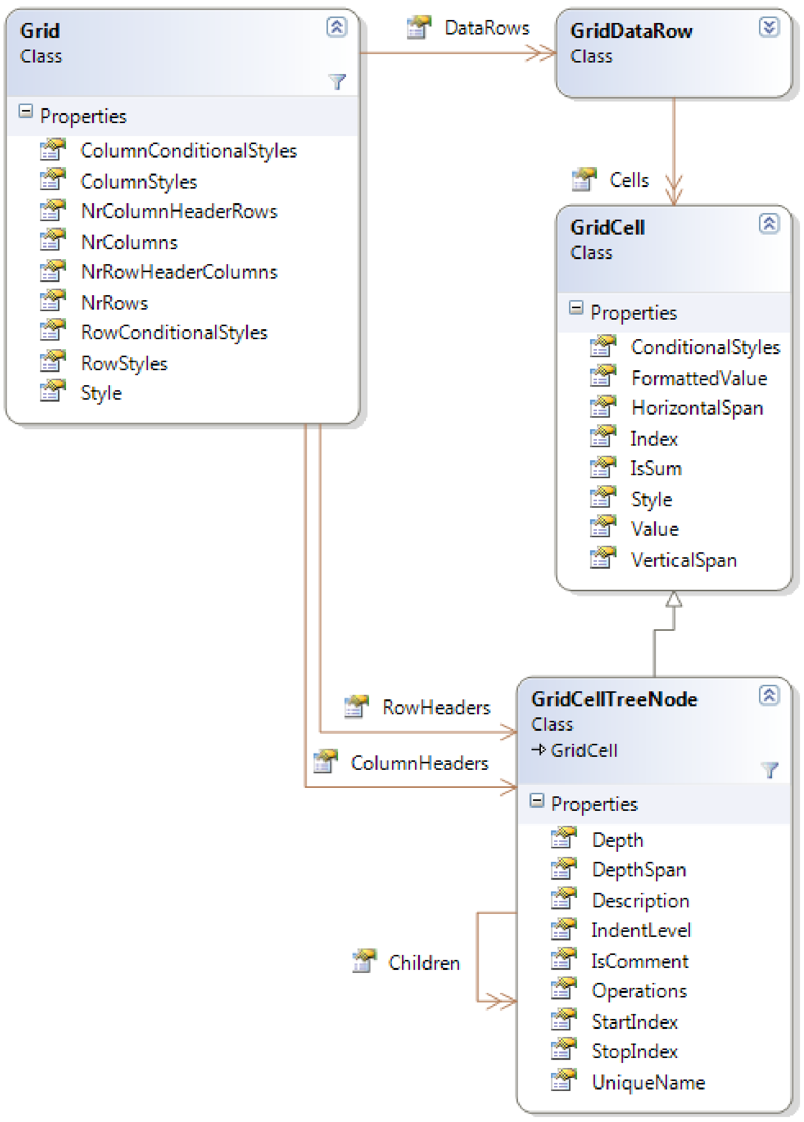


Figure 5. Class diagram of the grid DTOs.

A `GridCell` contains the value of the cell as well as a formatted value. In addition to the data rows, the `Grid` has trees of row headers and column headers. The reason of the tree structured headers is the hierarchical nature of row and column headers in the data sources used by INSIGHT. The tree nodes representing the header cells are modeled by the `GridCellTreeNode` class which inherits from `GridCell`. The hierarchical structure of the header cells is modeled with the recursive `Children` property. All header cells have unique identifiers (the `UniqueName` property).

In addition to the most basic properties discussed above, there are properties concerning for example cell indices and span, styles and conditional styles.

6.2.2 Server Logic

The main server business logic can be found in the coordinator. The coordinator is hidden behind a layer of facades with the intent to provide a simple interface to the server logic. For the client to be able to access the facade, the methods of the facade are exposed in a service interface.

The coordinator and facade layers are implemented in the `Coordinator` and `Facade` classes respectively. The facade simply acts as an entrance to the service logic layer and contains very little logic. The coordinator on the other hand contains the main business logic. This includes a cached grid, cached formatting rules as well as coordination of actions that can be performed on the grid, e.g. drill down and styling operations.

The grid cache contains the last retrieved grid object (the DTOs) and is invalidated when a new grid has to be fetched from the analysis server. The formatting rules cache contains the formatting rules currently applied to the grid.

The coordinator uses a data access layer (DAL) to access the data sources (OLAP cubes as well as a database). Formatting rules (associated to INSIGHT reports) are stored in the system database of INSIGHT and accessed through an existing data access layer. The cache is used to reduce traffic between the web server and the analysis and database servers as well as to increase response time of the application (by not having to access the data sources every time).

The main data source is OLAP cubes hosted by Microsoft Analysis Services. The data access layer whose responsibility is to access cubes is abstracted behind the `IData` (for read only data access) and `IEditableData` (for read and write data access) interfaces. The `IData` interface contains a single method called `GetData` that returns a `Grid` object. The `IEditableData` adds a method to update cell values.

There are three implementations of the `IData` interface (see Figure 6). The `TestData` implementation is mock data provider generating fake data without having to access an analysis server. This class has been used for testing purposes and when running outside of INSIGHT.

The `CellSetData` class is responsible for fetching data from a cube on an analysis server and parses the cell set (the data structure returned by the analysis server) and translates it to the DTO data structure. The `AnalysisData` implementation of `CellSetData` uses the current state of the analysis module of INSIGHT (information about the report currently loaded in INSIGHT) to prepare a query to send to the analysis server. This class is thus responsible for the integration with the analysis module of INSIGHT.

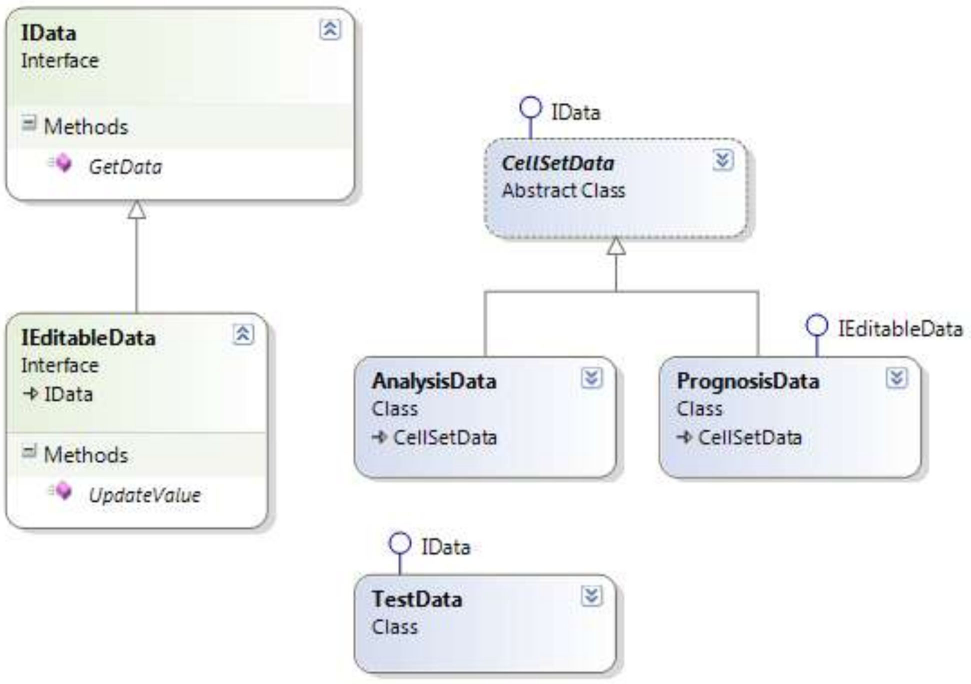


Figure 6. Class diagram of data access layer implementation.

6.2.3 Service Interface and Implementation

Figure 7 shows a class diagram of the service interface, called `IGridService`, and the facade used as an entrance to the service logic layer. The service interface contains operations to get data (the grid), to notify the server when the user has altered the styling of the grid or the content/value of a cell as well as some additional operations to get translations and to log exceptions. In summary, it contains all the operations required by the client.

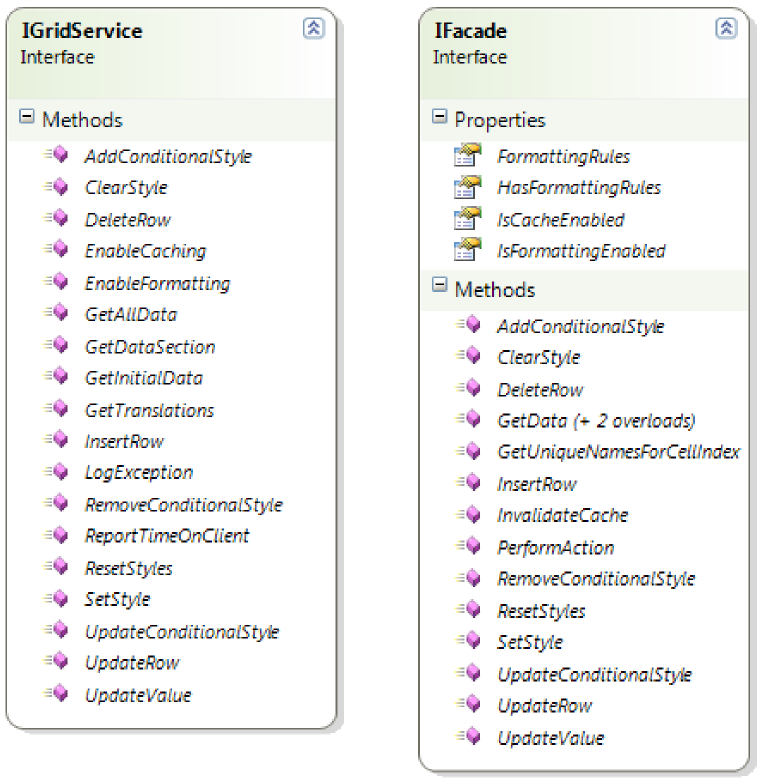


Figure 7. The facade of the service logic and the service interface.

The most important service operations are the ones to get data to the client. The `GetAllData` operation retrieves an entire grid (no data virtualization), while the `GetInitialData` operation is used for data virtualization by retrieving only the first part of the grid (a given number of rows). When using data virtualization, the `GetDataSection` is used to fetch more data rows when required.

The partitioning of the grid (for data virtualization, i.e. the `GetInitialData` and `GetDataSection` operations) is done in the coordinator by simply including only the rows requested. The cache in the coordinator contains the entire grid, so when it has first been fetched from the data access layer, subsequent request to `GetDataSection` will hit the cache (and thus avoids the delay of fetching data from the data access layer).

The service is implemented as a WCF web service according to the principles outlined in the section named Windows Communication Foundation (WCF). The DTOs exposed by the service are annotated by WCF attributes so that the framework knows how to map the objects to SOAP messages.

The service is currently exposed as an HTTP endpoint using binary encoding of the SOAP messages, but the endpoint can easily be replaced to match other requirements.

6.3 Client Implementation

The client has a reference to web service which means that Visual Studio automatically generates the required proxy class with one method per service operation. For each DTO class exposed in the service it also generates corresponding classes on the client.

The client logic is implemented in a number of `ViewModels` according to the MVVM design pattern. The `ViewModels` access the web service to fetch data and to inform the server when something has changed (e.g. styling or cell values) and prepare the information to be presented by the `Views`.

The main `ViewModel`, the `GridViewModel` class, is responsible for fetching and managing the grid DTOs and to expose them for use by the main `View`, the `SyncfusionGridPage`. Selection and styling is managed by another `ViewModel` called `SelectionViewModel`, exposed to the `Views` in a property on the `GridViewModel`.

In addition to the `ViewModels` mentioned above, there are a couple of smaller `ViewModels` responsible for specific tasks concerning formatting and filtering and each one is used by a `View`. One example is the `CreateFilterViewModel` which is responsible for creating filters and exposes properties bound to by the `FilteringUserControl` view.

The final prototype implementation uses a grid component Essential Grid (51) from Syncfusion as a starting point for the grid presentation. This component can be described as a Silverlight counterpart of the HTML table. It is basically a component for presenting tabular data. Several Silverlight grid components have been evaluated during the project and Syncfusion's was the one matching the requirements best. The other grid components evaluated during the project are the one included in the Silverlight Toolkit from Microsoft (52) as well as components from ViBlend (53), Telerik (54) and ComponentOne (55).

The grid component from Syncfusion implements UI virtualization out of the box and thus has the potential to perform well. It does not provide data virtualization though, but has a nice feature which opens for building data virtualization in the `ViewModel`. This feature is called virtual mode, and implies that the grid does not have to be initially filled with data but instead request data only when required. This is implemented by events (on the grid component object) that are raised when cell values are required and it is up to the code using it to react to those events and provide the grid with the values (cell content).

This event is handled in the code-behind of `SyncfusionGridPage`. The event handler uses properties on the `GridViewModel` to retrieve cell values. More specifically, the `GridDataRows` property (a collection of rows) of the `GridViewModel` is used to access the rows of the grid. With data virtualization enabled, this is a virtualized list. If an element requested by the grid is not in the collection, it is fetched from the web service by the `ViewModel`. Hence, all data virtualization logic is hidden in the `ViewModel` and the `View` does not even need to know about it. The `View` thinks it is accessing an ordinary collection.

6.4 Virtualized List Implementation

The data virtualization implementation is centered round the `IVirtualizedCollection<T>` interface implementing `IList<T>`. This interface simply represents a virtualized collection and adds a property of type `IItemsProvider<T>` to the interface provided by `IList<T>`. This property, called `ItemsProvider`, is responsible for fetching data when required. The items provider knows the count of the entire collection as well as how to fetch the items.

The `VirtualizedCollection<T>` implementation maintains fetched items in a dictionary. The items are associated with their index in the "real" collection. When an item not in the dictionary is requested, more data is fetched using the items provider. Items are fetched in blocks (of consecutive items) with the assumption that the spatial locality is high in most cases. Also, the previous and next blocks are fetched with the same assumption. These details and the size of the blocks can easily be tweaked. The derived class `AsynchronousVirtualizedCollection<T>` does the same thing using an asynchronous items provider.

There are two implementations of `IItemsProvider<T>`, one synchronous and one asynchronous version, that both take a C# delegate in the constructor. The delegate is responsible for fetching data from some data source.

A class diagram showing the virtualized list implementation is shown in Figure 8.

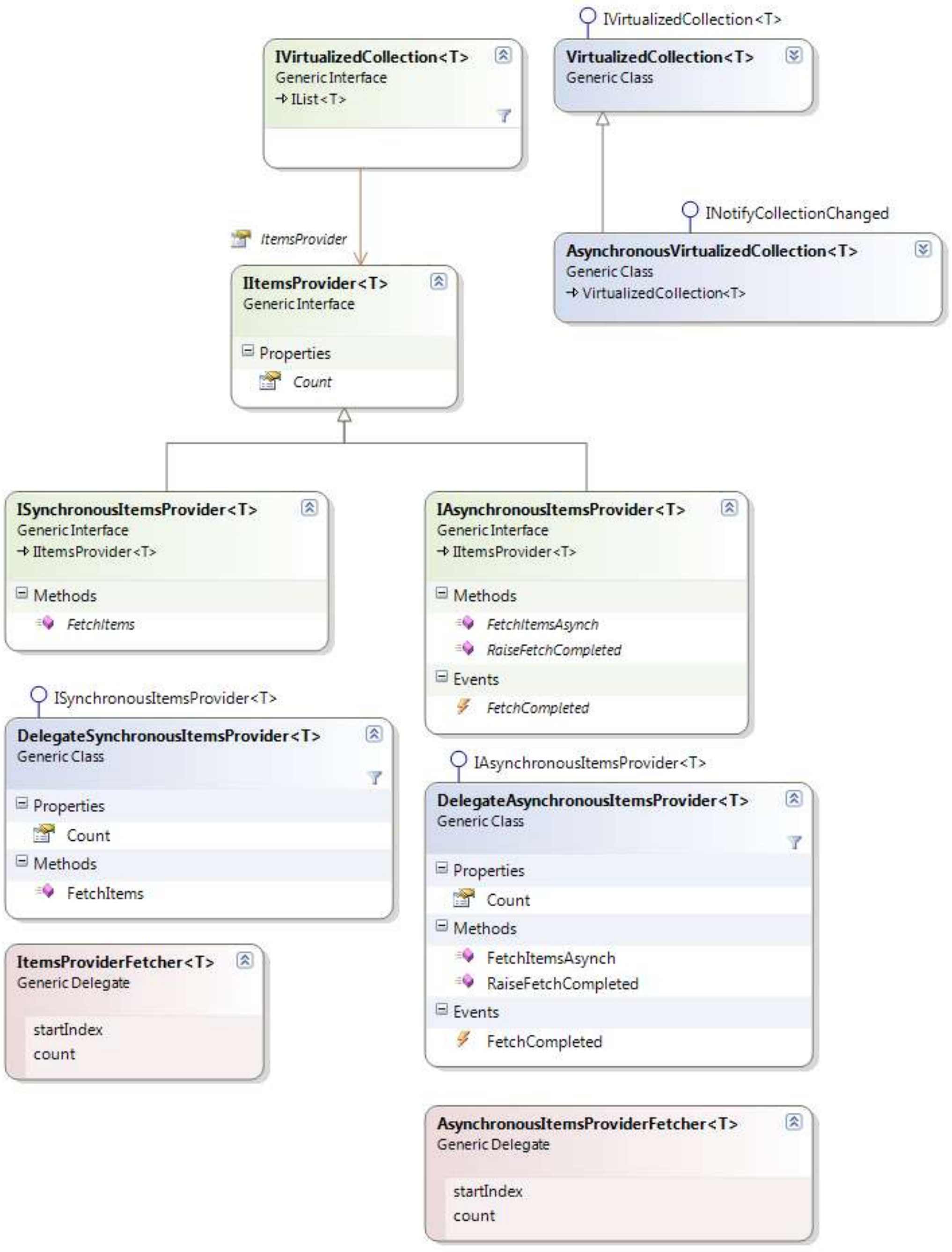


Figure 8. Class diagram of the data virtualization implementation.

6.5 Performance Testing Implementation

As performance testing is one of the tasks of the project, the implementation contains time tracking. Table 1 lists and defines the measures that have been defined and used for performance testing.

Table 1. The time measures used for performance testing.

Measure	Definition
Total Time	The total time from when the user requests a report until it is displayed in the browser.
Rendering Time	The total time for the user interface to render the grid.
Service Call Time	The total time from requesting data in the client to retrieving the results MINUS the total time processing the operation on the server, i.e. the time spent on the network and building and parsing messages.
Total Server Processing Time	The total time spent on the server MINUS the time it takes to fetch data from the data source.
Time to Fetch Data from Data Source	The time it takes to fetch data from the data source.

The listed entities are quite easy to measure in code. The client can easily measure the total time by starting a stop watch when the user presses a button and stop it when the rendering is completed (there is an event for this). In the same way it is easy to measure the rendering time since it is known when rendering starts. The total time spent on the server is measured by starting a stop watch the first thing when a service call arrives and stopping it when it returns.

The service call time is the trickiest part. It is calculated as the total time from calling a service method until it returns minus the total processing time on the server. The result is the total time WCF spends on sending and receiving messages.

To be able to compare the results, an existing version of INSIGHT has been modified to measure the same things. The server times are measured the same way, except that the start of server processing is defined as the entry point in the ASP.NET page constructor and the end of server processing is defined as the `Unload` event of the page. Service call time is also a bit different. It is measured as the time from the `onclick` JavaScript of the button (the button used to load/refresh a report) until the web browser starts to parse the JavaScripts in the response (the new HTML page). The time when rendering is completed is measured when the `document.onreadystatechange` JavaScript event is raised.

7 Performance Testing

This section describes basic theory of performance testing, the performance testing targets and goals of this project and finally the test cases designed specifically to meet the performance testing goals and to check if the targets are met.

7.1 Performance Testing Theory

This section briefly summarizes the most important things learned when studying performance testing theory.

According to Wikipedia (56) there are three common objectives of performance testing:

- Demonstrate that the system meets performance criteria (or does not meet performance criteria).
- Compare two or more systems to find which performs better.
- Find out what parts of systems or workload causes the systems to perform badly.

The performance testing in this project is mainly targeted towards comparing two systems, but also includes a little from the other two objectives listed above.

Alberto Salovia (57) describes accurate performance testing as a complex task, since collecting and analyzing irrelevant data is easy. Salovia mentions overestimation and underestimation of application performance as two common scenarios; in some cases by an order of magnitude. The main reason to this, Salovia continues, is oversimplification.

To reduce the risk of underestimation, it is, according to (57) important to plan the task of performance testing from start to finish. The following seven core performance testing activities are given:

1. Identify Test Environment
2. Identify Performance Acceptance Criteria
3. Plan and Design Tests
4. Configure Test Environment
5. Implement Test Design
6. Execute Tests
7. Analyze, Report and Retest.

Molyneaux (58) describes last minute performance testing as a reason to bad performance. According to Molyneaux it is important to include performance considerations in application design.

In all cases, it is important to arrange a test environment and testing conditions as similar to the production or intended environment as possible. This includes for example hardware, network, software and tools. However, this is very hard and exact replicas are rare (57). Molyneaux (58) also stresses the importance of setting realistic and appropriate targets and to define a testing environment that matches the intended environment as closely as possible. According to Molyneaux it is important to use automated test tools for larger performance testing activities.

7.2 Performance Testing Goals and Targets

The ultimate goal of the performance testing performed in this project is to determine if Silverlight can be used to enhance performance of ASP.NET/HTML applications showing large data grids, and more specifically if the Silverlight grid application implemented as a part of this project performs better than the older ASP.NET/HTML solution used by INSIGHT.

Another goal is to find out the importance of UI- and data virtualization when trying to achieve good performance in a Silverlight application.

The ultimate performance testing target is that the INSIGHT version running the Silverlight grid prototype performs better than the latest version of INSIGHT. The following three bullets define the target less ambiguously.

- Small reports (less than 100 rows) are loaded at least as fast as the ASP.NET/HTML version.
- Medium sized reports (100 – 1000 rows) are loaded faster than the ASP.NET/HTML version.
- Large reports (more than 1000 rows) are loaded much faster than the ASP.NET/HTML version.

A *report* in INSIGHT is a tabular presentation of data from an OLAP cube. The report contains settings specifying what dimensions to show on rows and columns and what measure to show in the table. An example is a report showing customer sales with time as column dimension and region as row dimension. The report is dynamic in the sense that the user can drill down into the data along the selected dimensions. To continue the example, the user could for example isolate a few regions and drill down into the time dimension to analyze customer sales per month and region.

What is meant in the performance testing targets above is the time it takes to load a predefined report (prepared and saved at an earlier time).

7.3 Performance Test Cases

This subsection describes the test cases performed. The test cases have been designed to examine the pros and cons of UI- and data virtualization and to compare the resulting Silverlight application with the existing ASP.NET/HTML solution employed by INSIGHT.

The first test case, List With and Without UI Virtualization, uses a custom Silverlight application created specifically for the test. All other test cases use the final implementation of the Silverlight grid application prototype integrated in INSIGHT.

The specifications of the computers used throughout the tests are listed in Appendix B – Test Machine Specifications.

7.3.1 UI Virtualization

Three test cases for examining UI virtualization have been designed. All three tests use test machine A as client computer.

7.3.1.1 List With and Without UI Virtualization

To further evaluate UI virtualization a simple test case with the standard Silverlight `ListBox` (59) has been carried out.

The Silverlight `ListBox` is a simple user interface control displaying a collection of items. By default, the items in the list are listed vertically and a vertical scroll bar is displayed if necessary. In Silverlight 4 the `ListBox` control has UI virtualization enabled by default, by utilizing the `VirtualizingStackPanel` as items container.

To investigate the significance of UI virtualization for the `ListBox` control, a very simple Silverlight application has been created. The application user interface contains two `ListBox` controls; one with UI virtualization enabled and the other one with UI virtualization disabled. The user interface also contains a text box allowing the user to enter the number of items to generate (and populate into the lists). Below both lists, there is an update button used to refresh the list and a text field displaying the rendering time last time it was updated. Figure 9 show a screenshot of the test application.

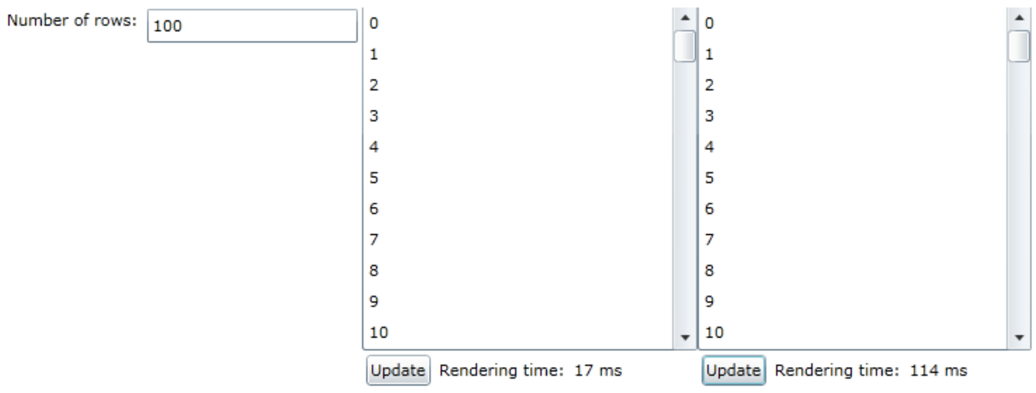


Figure 9. A screenshot of the application used to examine `ListBox` performance.

Using this application, it is simple to obtain average rendering times by updating the lists multiple times and observing the results. In this case, samples of ten updates have been used to calculate average rendering times with and without UI virtualization for 10, 25, 50, 100, 250, 500, 750, 1000, 5000, 10000, 50000 and 100000 rows.

7.3.1.2 Rendering Time vs. Number of Cells Visible

This test case has been designed to examine the relationship between the rendering time and the number of cells visible in the grid. In a perfect implementation, UI virtualization would lead to a rendering time that is linearly dependent on the number of cells visible in the grid.

With a fixed grid size of 100 by 100 cells, the number of cells visible can be varied by resizing the browser window. In this case, a sample size of three has been used to calculate average rendering times for 24 different windows sizes with from 9 to 800 cells visible.

This test case has been performed on the final prototype based on Syncfusion's grid component.

7.3.1.3 Rendering Time vs. Number of Rows

This test case has been designed to examine the relationship between the rendering time and the number of rows in the grid (with a constant browser window size). In a perfect implementation of UI virtualization, the rendering time would increase up to the point where no more rows fit on the screen and then stay constant as more rows are added.

This test case has been performed for 10, 20, 30, 40, 70, 100, 500, 1000, 10000 rows and 10 columns using the final prototype based on Syncfusion's grid component. The maximum number of rows visible on the screen was 30. The results are average rendering times with a sample size of ten.

7.3.2 Data Virtualization

This test case has been designed to show the pros and cons of data virtualization. The service call time (when first loading a grid) has been measured, by subtracting the server processing time from the total time from initiation of service call to arrival of response. Average service call time (sample size of ten) has been measured for 1000, 2000 and 4000 rows and 10 columns with data virtualization enabled and disabled. The size of the data sections (the smallest number of consecutive items sent between server and client) was set to 100 rows.

This test case also measures the total delay when the user requests more rows (by scrolling). This delay depends only on the size of data sections and has been performed with a data section size of 100 rows (sample size of ten).

The web server (test machine A) and the client computer (test machine B) were connected to the same 100 Mbit LAN.

7.3.3 INSIGHT Performance

The final performance test compares the performance of the final Silverlight prototype integrated in INSIGHT with an existing version of INSIGHT (version 5.4). Test results are presented for five typical INSIGHT analysis reports or varying sizes. Typical reports are used since the intention is to match the typical use case as closely as possible. The sizes of the selected reports are listed in Table 2. The reason to the uneven numbers (compared to previous test cases) is that the selected reports happened to have these sizes.

Since performance optimizations in communication with the cube (the analysis server) is not in focus in this project, the "Time to Fetch Data from Data Source" measure should be approximately the same for the two versions of INSIGHT (the existing version and the new prototype). This makes the comparison as fair as possible, without including data source optimizations.

The theory about performance testing stresses the importance of a test environment matching the intended environment as closely as possible. This test case tries to do this but to a limited extent. Since it is very hard to simulate several users running at the same time and other systems competing for server and network resources this test case does not attempt to do this.

The web server (test machine A) and the client computer (test machine B) were connected to the same 100 Mbit LAN. Test machine B also acted as analysis server. For data virtualization, a data section size of 100 rows has been used.

Table 2. The sizes of the reports used for testing.

	Rows	Columns
Tiny report	25	7
Small report	100	7
Medium size report	652	7
Large report	1985	9
Huge report	54787	2

8 Performance Testing Results

This section presents the results of the performance testing. The data behind the diagrams presented in this section can be found in tabular form in Appendix A – Performance Test Result Tables.

8.1 List With and Without UI Virtualization

Figure 10 shows rendering performance for a standard Silverlight `ListBox` with and without UI virtualization. During the test, a maximum of 46 list box items were visible in each `ListBox` due to monitor real estate. From the figure it can easily be found that UI virtualization improves performance drastically when the number of rows exceeds what is shown in the user interface. For 10, 25 and 50 rows UI virtualization does not improve performance at all since all (or almost all) elements are visible and have to be rendered. A performance penalty of having UI virtualization enabled (but not enough rows to make use of its advantages) cannot be spotted either.

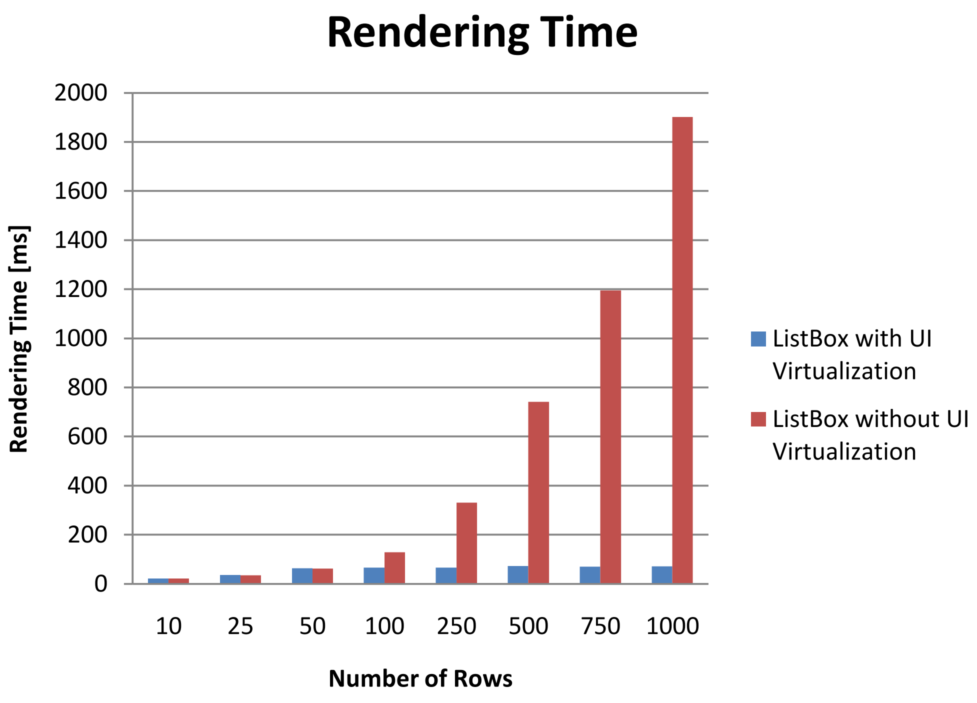


Figure 10. `ListBox` rendering performance with and without UI virtualization for an increasing number of rows.

Figure 11 clarifies the trend for when the UI virtualization technology is enabled. The rendering time increases until the user interface cannot hold more rows and then stabilizes around 70 milliseconds. For 50 to 1000 rows the additional rows not visible in the user interface amounts to a very small overhead and the rendering time is more or less constant. What is surprising is the numbers for 10000, 50000 and 100000 rows which indicate a fast increasing rendering time. The rendering time for 100000 rows with UI virtualization enabled is still less than that for 500 rows with UI virtualization disabled, but the `ListBox` implementation of UI virtualization is clearly not perfect. If it had been perfect, the rendering time for 100000 rows would have been more or less the same as the rendering time for 50 rows, since the same number of rows are visible in the user interface.

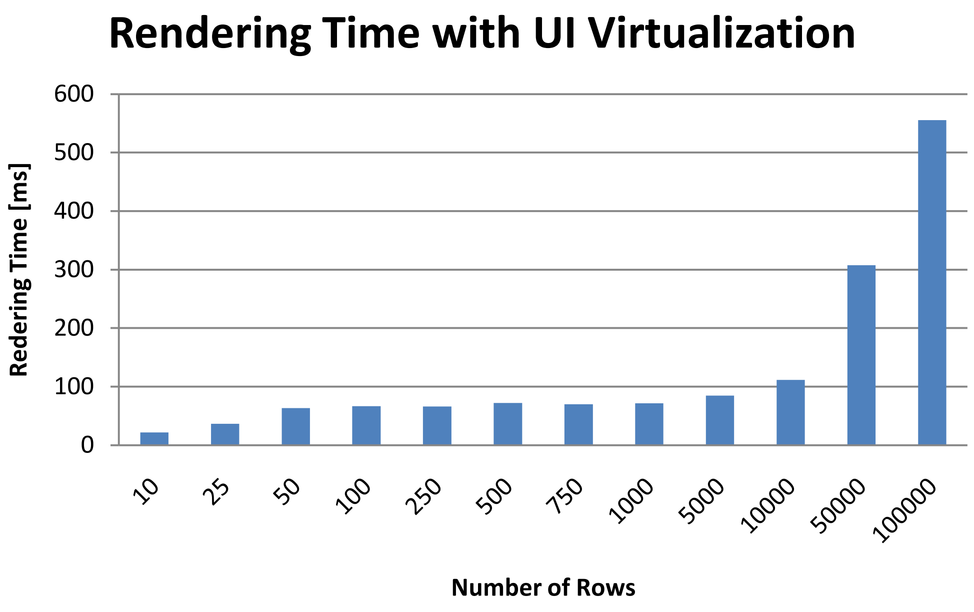


Figure 11. `ListBox` rendering performance with UI virtualization enabled for an increasing number of rows.

Figure 12 shows the rendering time per row with and without UI virtualization for an increasing number of rows. Clearly, the rendering time per row with UI virtualization disabled does not decrease as the number of rows increase.

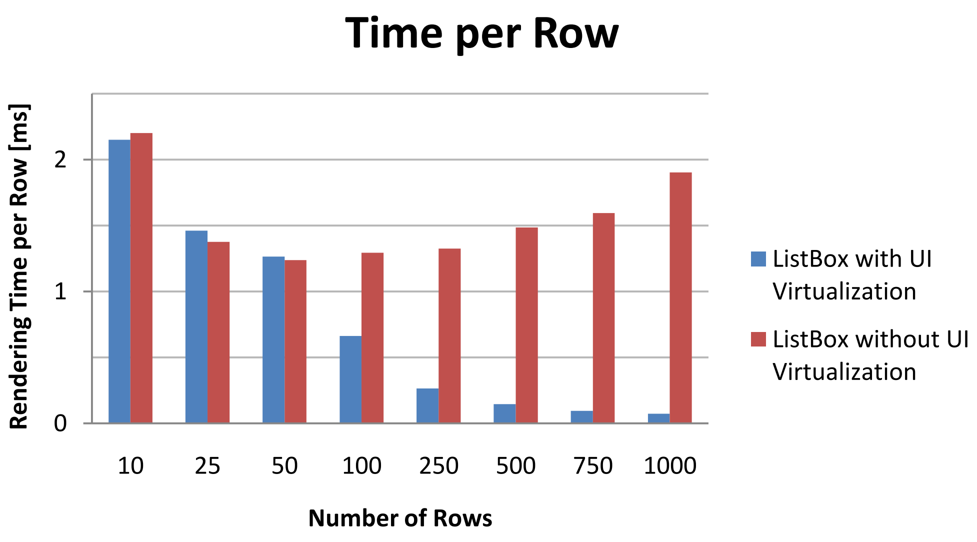


Figure 12. `ListBox` rendering time per row for an increasing number of rows.

It should also be noted that the user experience when scrolling is still very good for 100 000 rows when UI virtualization is enabled, but very bad for 10 000 rows with UI virtualization disabled.

8.2 Rendering Time vs. Number of Cells Visible

Figure 13 shows the rendering time for a grid with 100 rows and columns as the browser window size is increased to fit more and more cells. The rendering time is clearly related to the number of cells visible, just as expected. From the figure it is also easy to imagine a linear relationship.

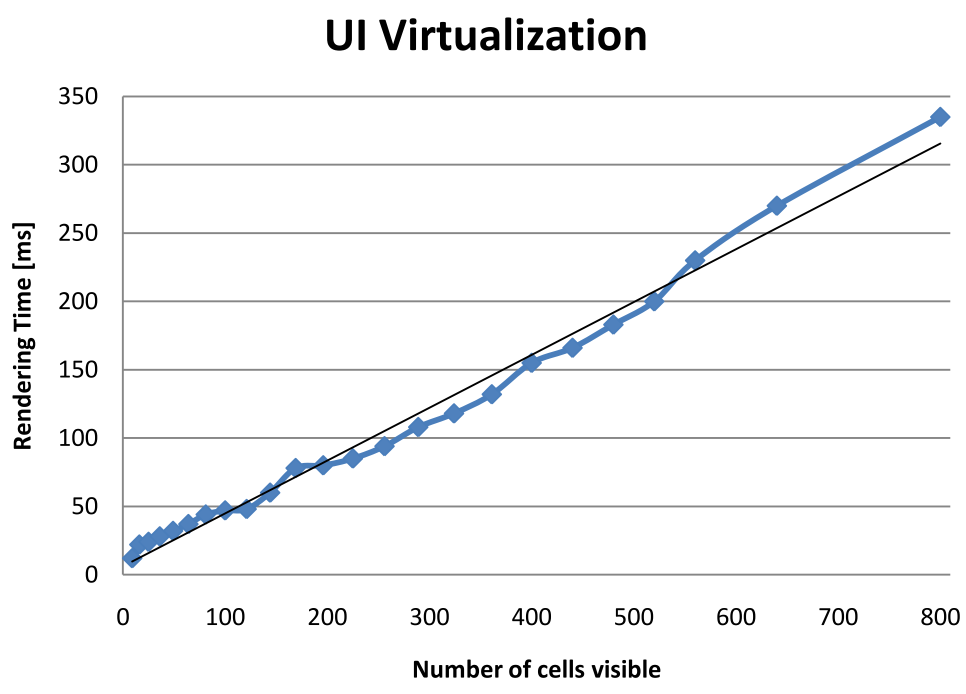


Figure 13. Rendering time for a grid with 100 rows and columns with an increasing number of visible cells (and increasing size of browser window).

The values in the last column in Table 3, rendering time per cell, seem to converge around 0.40 milliseconds. These results strengthen the suspicions that the relationship is linear or at least close to linear.

Table 3. Rendering time for a grid with 100 rows and columns with an increasing number of visible cells (and increasing size of browser window).

Number of Cells Visible	Rendering Time [ms]	Rendering Time per Cell [ms]
9	12	1,33
16	22	1,38
25	24	0,96
36	28	0,78
49	32	0,65
64	37	0,58
81	44	0,54
100	47	0,47
121	48	0,40
144	60	0,42
169	78	0,46
196	80	0,41
225	85	0,38
256	94	0,37
289	108	0,37
324	118	0,36
361	132	0,37
400	155	0,39
440	166	0,38
480	183	0,38
520	200	0,38
560	230	0,41
640	270	0,42
800	335	0,42

8.3 Rendering Time vs. Number of Rows

Figure 14 shows the results of the third test case. The average rendering time increases up to the point where no more rows can fit in the visible portion of the user interface. The user interface could show a maximum of 30 rows and as expected the rendering time does not change much when the number of rows are increased further. The rendering seems to stabilize around 300 milliseconds.

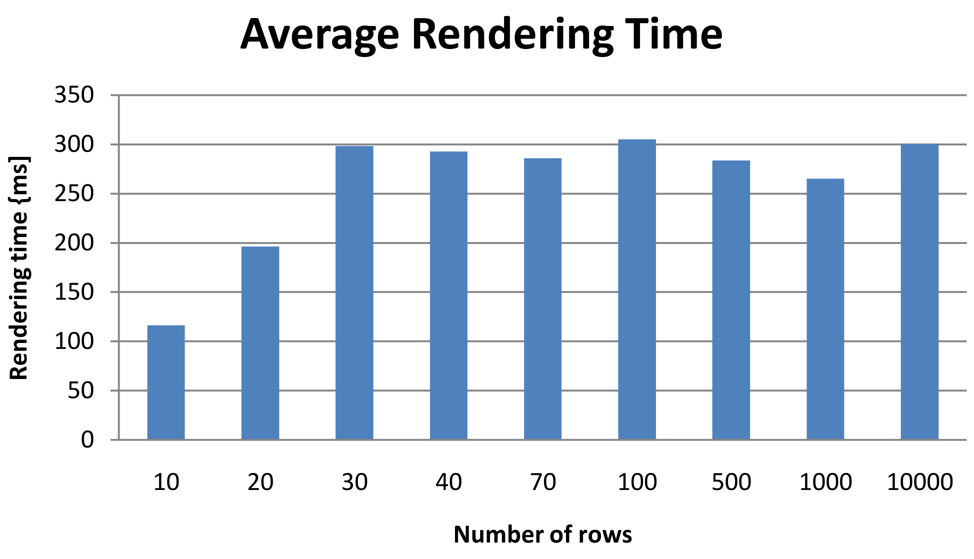


Figure 14. The average rendering time of the grid when the number of rows increases.

8.4 Data Virtualization

Figure 15 presents the result of the data virtualization test case. It is clear that data virtualization decreases the service call time and hence the time the user has to wait for a report to load. These results are in line with the expectations since less data is sent with data virtualization enabled. In this test case, with a data section of 100 rows, only the first 100 rows are sent when loading a report regardless of the size of the report. That is why the service call time with data virtualization enabled is more or less constant (as long as the report is larger than the data section).

Although the benefit is clear it is a bit smaller than expected. For a report of 1000 rows, only one tenth of the rows are sent with data virtualization enabled, but the service call time is reduced with only about 35 %. This is probably due to a pretty large overhead for sending and receiving SOAP XML messages.

The benefit of data virtualization does of course increase as the report size is increased. A report with 4000 rows is loaded about 1.4 seconds faster with data virtualization enabled.

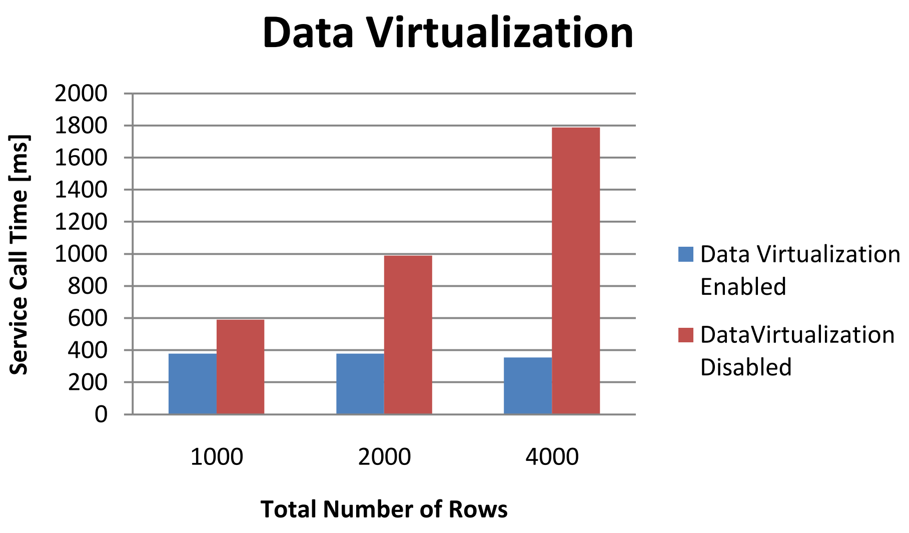


Figure 15. The average service call time with and without data virtualization.

The drawback of data virtualization is the increased complexity in the implementation, and the additional delay when scrolling the report. Say that the user views a report with 1000 rows. The report is indeed loaded 200 milliseconds faster, but when scrolling there is an additional delay of approximately 755 milliseconds every time a new data section has to be fetched. The significance of this additional delay depends on the expected behavior of the typical user. If the user always scrolls through the entire report, the additional delay is probably unacceptable. If the typical behavior is to view only small parts of reports it may be perfectly acceptable.

8.5 INSIGHT Performance

Figure 16 shows the result of the comparison between the prototype and the latest version of INSIGHT. For the tiny report, the latest version of INSIGHT is slightly faster. For the other reports, the prototype is faster; by small margin for the small report but above that the margin is significant. For the medium report the prototype is more than twice as fast and for the large report it is more than three times as fast.

The existing version of INSIGHT could not handle the huge report. During the rendering of the report, Internet Explorer kept alerting the user that something may be wrong. After rejecting several warnings the report finally rendered, but it is very hard to measure the load time accurately and that is why no exact figures are presented. A rough estimate is that the total load time is somewhere between one and two minutes.

Figure 16 also indicates that the advantage of data virtualization is small up to and including the medium report. For the medium report there is a small benefit (a few milliseconds), but it is not significant. For the large report the benefit is almost half a second and for the huge report the total time is more than halved with data virtualization enabled.

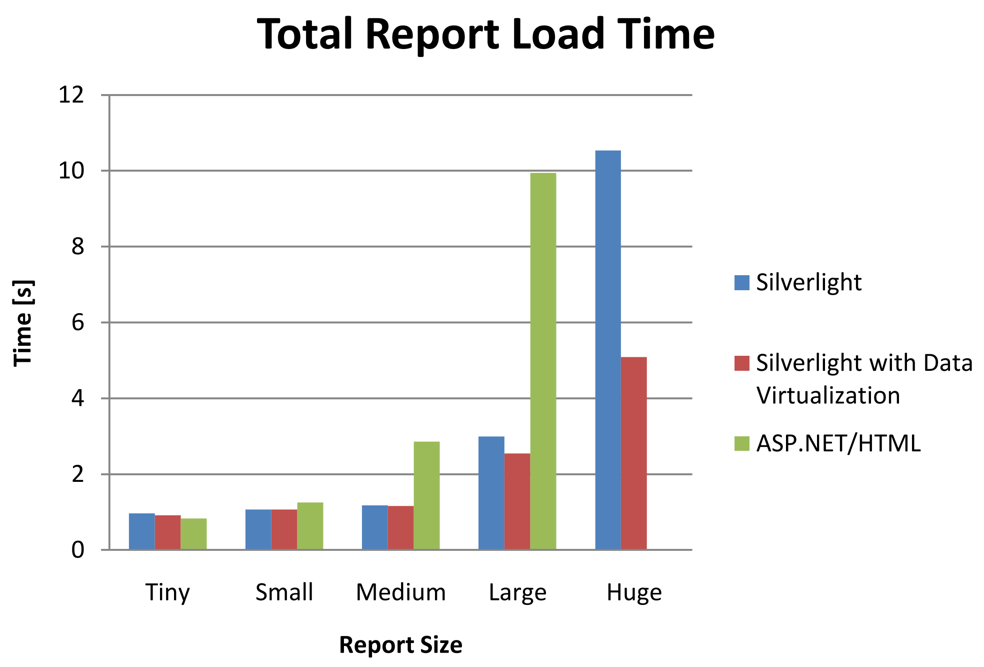


Figure 16. Total time to load reports of different sizes for the Silverlight prototype (with and without data virtualization) and the latest version of INSIGHT.

Figure 17 shows the total time loading the tiny report broken down in five measures. As previously noted, the existing version of INSIGHT performs slightly better in this case. As expected, the time to fetch data from data source is approximately the same for all three versions and this is also the case for the four other reports (see below).

What is remarkable in this figure is that the rendering time is about the same for all three versions, but that the service call time is a bit smaller for the ASP.NET/HTML version.

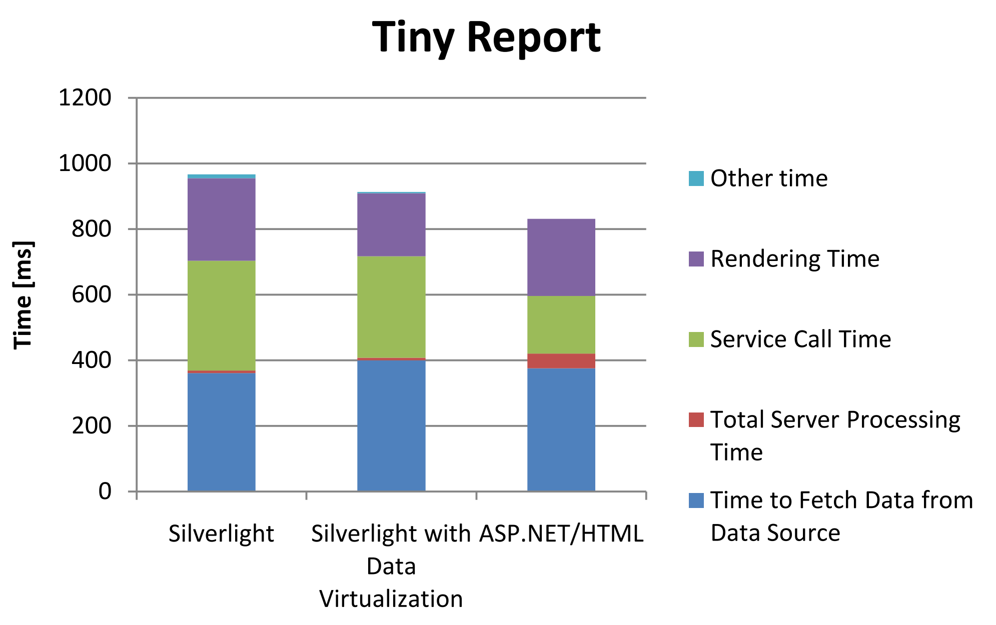


Figure 17. Total time, broken down in five measures, to load the tiny report.

Figure 18 shows the same type of chart for the small report. It is now apparent that the rendering time is smaller in the Silverlight prototype, which is expected since UI virtualization now is in effect.

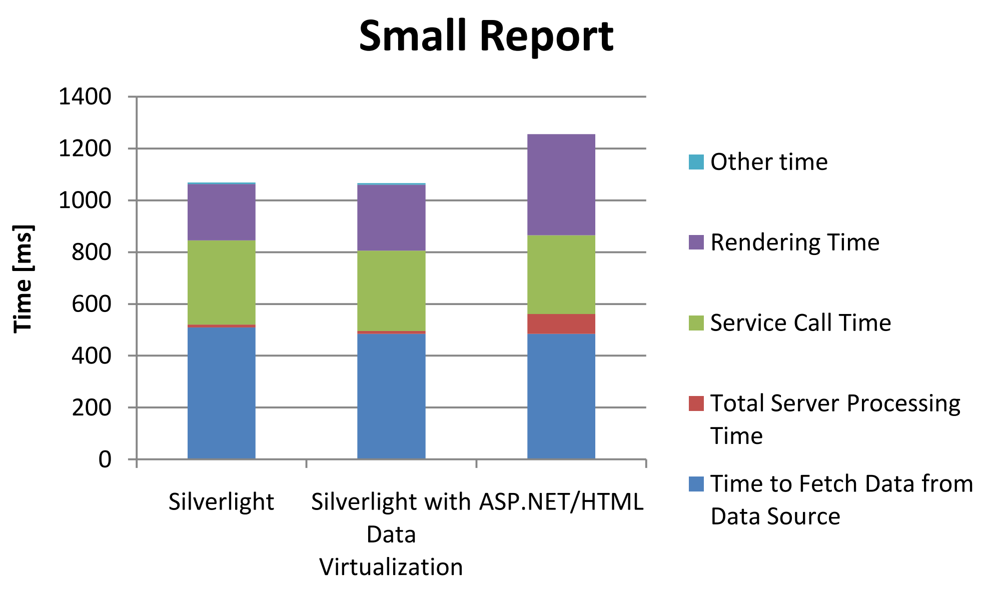


Figure 18. Total time, broken down in five measures, to load the small report.

For the medium report, see Figure 19, the rendering time of the ASP.NET/HTML version continues to increase relative to the rendering time of the Silverlight prototype. It is also remarkable that the service call time now is much larger in the existing version of INSIGHT. Since the benefit of data virtualization still is not significant, this was not expected.

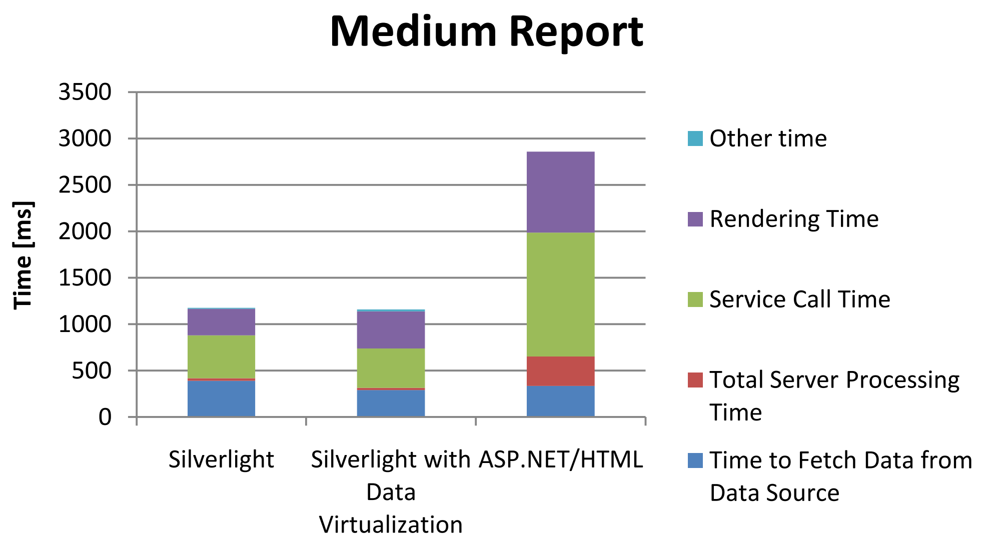


Figure 19. Total time, broken down in five measures, to load the medium report.

Continuing with the corresponding chart for the large report (Figure 20), the rendering time in the existing version of INSIGHT is almost ten seconds compared to less than 350 milliseconds for the Silverlight prototype.

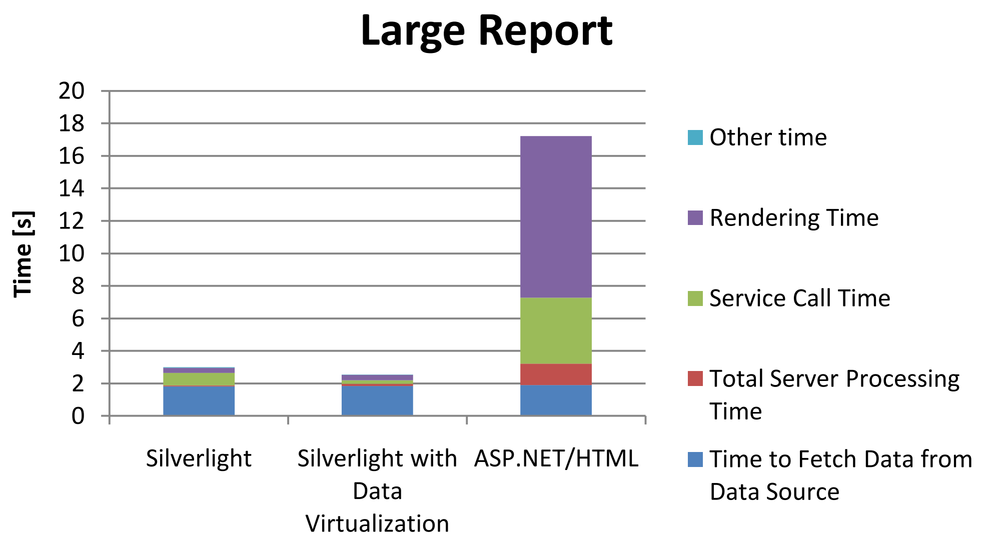


Figure 20. Total time, broken down in five measures, to load the large report.

Figure 21 shows the same type of chart for the huge report. Since the ASP.NET/HTML version could not handle this report the chart only examines the effect of data virtualization on the service call time. It is clear that data virtualization decreases the service call time from about six seconds to less than 400 milliseconds. These results are expected since less data is transferred between the server and the client at load time. In this case the version with data virtualization enabled is more than twice as fast as the version without data virtualization.

Note however that an additional delay is present whenever the user requests more data by scrolling the report (as is presented in section 5.2).

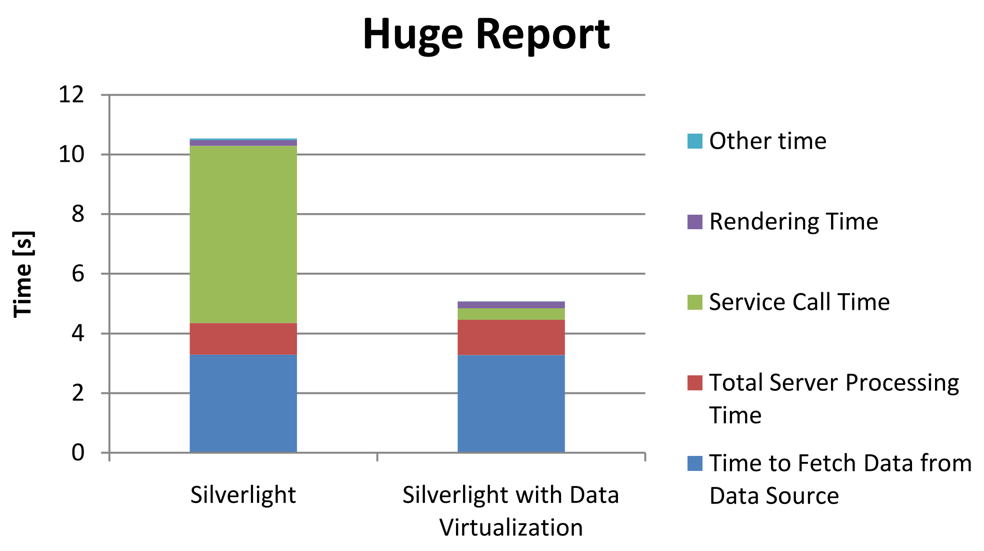


Figure 21. Total time, broken down in five measures, to load the huge report.

To summarize, the Silverlight prototype is faster than the existing version of INSIGHT except for very small reports for which the existing version has a small advantage. For medium to large reports, the prototype wins with significant margin.

From an application perspective, the relative benefit of the prototype depends on the time it takes to fetch data from the data source, in this case from the analysis server. If this time is large, the benefit of the prototype gets relatively smaller. This is especially important with data virtualization enabled. In Figure 21 for example, the load operation is twice as fast with data virtualization enabled. The relative difference could though have been larger if the time to fetch data from the data source had been smaller.

9 RIA Advantages

To exemplify the advantages of rich Internet applications when it comes to creating responsive web applications, the Silverlight prototype adds some new functionality that is not available in earlier versions of INSIGHT. The new functionality allows users to format their INSIGHT reports using a user interface similar to what can be found in for example Microsoft Excel.

The user can select ranges of cells, rows or columns using the mouse and can then format selected ranges using the floatable popup window shown in Figure 22. Figure 23, Figure 24 and Figure 25 shows additional formatting features used to create conditional styles and icon sets.

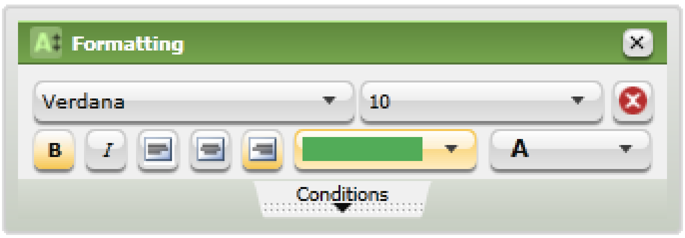
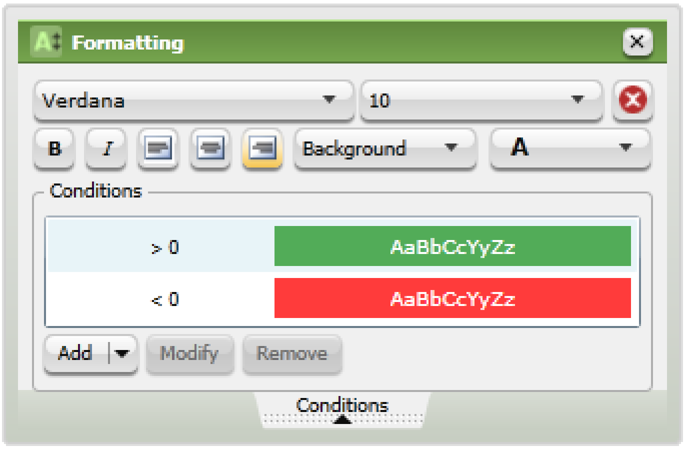


Figure 22. The formatting window.



> 0	AaBbCcYyZz
< 0	AaBbCcYyZz

Figure 23. The formatting window with the conditional styles panel expanded.

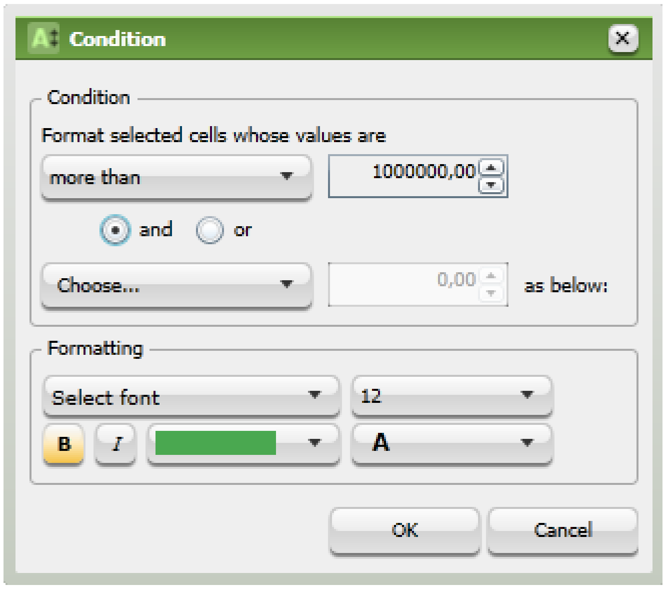


Figure 24. Dialog for creating and editing conditional styles.

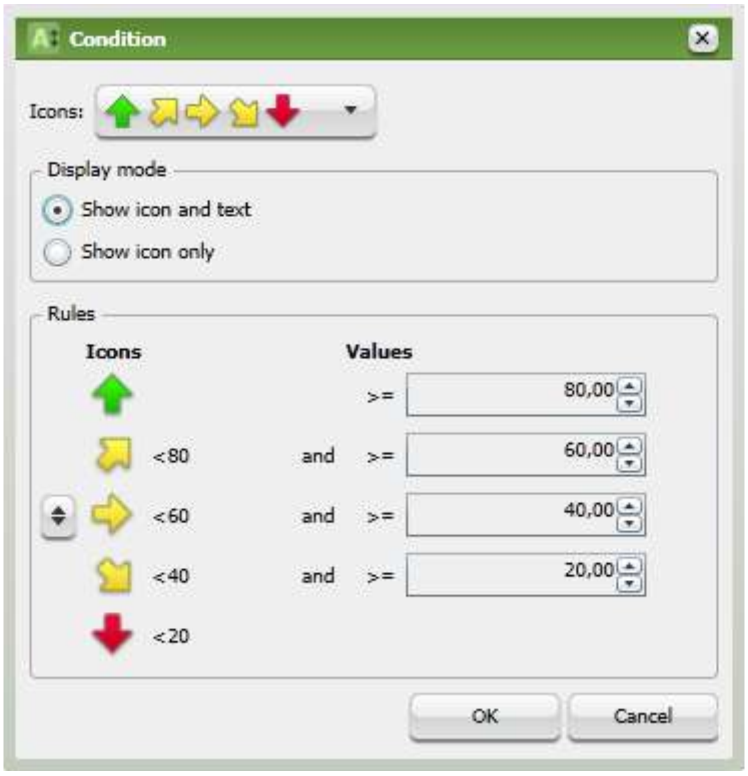
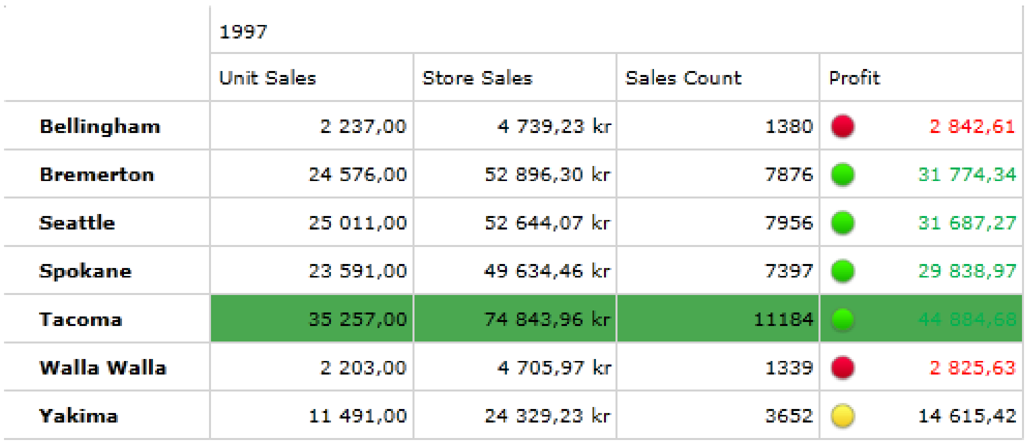


Figure 25. Dialog for creating icon series.

Figure 26 shows report formatted using this functionality.



	1997				
	Unit Sales	Store Sales	Sales Count	Profit	
Bellingham	2 237,00	4 739,23 kr	1380	🔴	2 842,61
Bremerton	24 576,00	52 896,30 kr	7876	🟢	31 774,34
Seattle	25 011,00	52 644,07 kr	7956	🟢	31 687,27
Spokane	23 591,00	49 634,46 kr	7397	🟢	29 838,97
Tacoma	35 257,00	74 843,96 kr	11184	🟢	44 884,68
Walla Walla	2 203,00	4 705,97 kr	1339	🔴	2 825,63
Yakima	11 491,00	24 329,23 kr	3652	🟡	14 615,42

Figure 26. A formatted INSIGHT report.

When a new formatting action is performed, the grid is instantly and automatically updated. Since the formatting is expected to be persistent, every action has to be sent to the server as well (so that the server can maintain formatting rules). This is all done by communicating with the web server in the background and thus it was possible to keep the user interface responsive.

This could of course have been implemented using HTML, JavaScript and Ajax, but it is probably easier to do this kind of things in Silverlight (or any other platform for building RIA). This implementation does not intend to prove that this is the case.

10 Conclusions

UI virtualization is without doubt a critical component for reaching high rendering performance in Silverlight applications presenting large data sets. The tests presented in this report indicate that with a good implementation of UI virtualization the rendering time is more or less constant as the number of items (in the list of grid) increases (given that the user interface does not fit more items). The strength of UI virtualization is that only the rows visible in the user interface are rendered. This means that new items have to be rendered when the user scrolls the list or table which may result in a somewhat slow scroll experience. This is probably the major drawback of UI virtualization.

Data virtualization is probably, in most scenarios, not as critical as UI virtualization. The tests cases show that quite large data sets are required to get a significant advantage and the additional delay when fetching new items may not be acceptable in some scenarios. For large data sets the load time is much better with data virtualization enabled, and if the typical user behavior is to go through small parts of the data it may be a large win. If the typical user behavior on the other hand is to scroll through the entire data set the additional “load more data” delay may not be acceptable. The benefit of data virtualization may be larger in environments with slow network connections.

The performance testing targets for normal and large reports have been met. Medium reports are loaded faster in the prototype than in the latest version of INSIGHT. Large reports are loaded much faster. For small reports, the prototype performs slightly worse than the latest version of INSIGHT. It is not by much, but the target saying that small reports should be loaded at least as fast as the ASP.NET/HTML version has not been met.

Microsoft recently announced next version of Silverlight, Silverlight 5. One of the new features is “Immediate mode graphics API allows direct rendering to the GPU”. This could perhaps improve the rendering performance and decrease the problem of bad scroll performance.

10.1 Limitations

The data virtualization implementation only virtualizes rows and not columns. In a case where the grid has lots of columns this could be a problem or at least a possible target for performance optimization.

Also, this work is based on one particular case. More similar projects evaluating Silverlight as a platform in terms of performance could perhaps strengthen the conclusions of this work.

The performance testing has limitations. The lack of professional automated testing tools and simulations of several users running the application at the same time may have affected the relevance of the results.

10.2 Future Work

The prototype implemented in this project concerns one particular case. It should be possible to learn from the ideas presented here and adapt it to other scenarios. The data virtualization should be easy to adapt for other scenarios, but virtualization on both rows and columns may require a larger retake.

This work does not drill into the details of how to implement UI virtualization in common controls. An in-depth walkthrough of common techniques to implement UI Virtualization may be helpful to further analyze the technique.

11 Acknowledgements

I want to thank TRIMMA for giving me the opportunity to do this project and all employees at TRIMMA for the feedback I have received. A special thanks to my external supervisor Daniel Hellström for supporting me during the project. I also want to thank my internal supervisor at Umeå University, Thomas Hellström.

12 Appendix A – Performance Test Result Tables

Table 4. Average rendering time for a `ListBox` with and without UI Virtualization for an increasing number of rows. Sample size was ten.

	List with UI Virtualization		List without UI Virtualization	
Number of rows	**Average**	**Standard Deviation**	**Average**	**Standard Deviation**
10	21.5	4.1	22	6.1
25	36.5	4.8	34.4	4.4
50	63.2	9.2	61.9	5.5
100	66.3	9.6	129.2	13.5
250	66.2	13.0	331.4	33.5
500	72.1	12.9	741.9	72.3
750	70.1	8.6	1195.2	89.0
1000	71.4	9.2	1901.4	147.1
5000	84.4	6.7		
10000	111.6	8.2		
50000	307.5	7.9		
100000	555.5	10.0		

Table 5. Average rendering time for an increasing number of visible cells (constant table size). Sample size was three.

Number of cells visible	Rendering Time	Rendering Time per Number of Cells Visible
9	12	1.33
16	22	1.38
25	24	0.96
36	28	0.78
49	32	0.65
64	37	0.58
81	44	0.54
100	47	0.47
121	48	0.40
144	60	0.42
169	78	0.46
196	80	0.41
225	85	0.38
256	94	0.37
289	108	0.37
324	118	0.36
361	132	0,37
400	155	0.39
440	166	0.38
480	183	0.38
520	200	0.38

560	230	0.41
640	270	0.42
800	335	0.42

Table 6. Average rendering time for the grid with an increasing number of rows. (UI Virtualization enabled.) Sample size was ten.

Number of rows	Average Rendering Time	Standard Deviation
10	116.5	12.0
20	196.4	21.7
30	298.1	23.7
40	292.6	24.7
70	286.0	29.8
100	305.1	28.2
500	283.7	21.9
1000	265.0	17.4
10000	300.4	44.6

Table 7. Average service call time with and without data virtualization. Sample size was ten.

	Data Virtualization Enabled		Data Virtualization Disabled	
Number of rows	Average	Standard Deviation	Average	Standard Deviation
1000	379.3	97.4	590.9	147.0
2000	379.4	142.2	988.9	244.6
4000	354.8	18.8	1788.3	282.2

Table 8. Silverlight prototype vs. INSIGHT performance comparison. Sample size was ten.

		Tiny	Small	Medium	Large	Huge
Silverlight	Total Time	966.1	1068.5	1176.9	2994.4	10535.4
	Rendering Time	252	217.4	286.7	326.6	193.5
	Total time on server	368.7	519.9	418.4	1874	4352.3
	TimeToFetchDataFromAnalysisServer	18.1	124.1	102.7	1432.7	1253.6
	TimeToParseGridStructure	6.9	8.3	22	53.9	617.2
	TimeToPrepareAnalysisServerCall	342.7	385.6	289.3	384.2	2029.1
	Service Call Time	333.9	325.3	460.9	770.7	5934.5
	Total Server Processing Time	7.9	10.2	26.4	57.1	1069.6
	Time to Fetch Data from Data Source	360.8	509.7	392	1816.9	3282.7
	Other time	11.5	5.9	10.9	23.1	55.1
ASP.NET/HTML	Total Time	831.2	1255.6	2858.5	9945.9	
	Rendering Time	235.4	390.4	873.6	2675.2	
	Total time on server	420.4	561.3	653.8	3215	
	TimeToFetchDataFromAnalysisServer	18.3	57.7	120.7	1547.1	

	TimeToParseGridStructure	0	0	0	0	
	TimeToPrepareAnalysisServerCall	356.8	426.6	214.4	350.3	
	Service Call Time	175.4	303.9	1331.1	4055.7	
	Total Server Processing Time	45.3	77	318.7	1317.6	
	Time to Fetch Data from Data Source	375.1	484.3	335.1	1897.4	
	Other time	0	0	0	0	
Silverlight with Data Virtualization	Total Time	913.3	1065.8	1159.1	2543.4	5086.2
	Rendering Time	192.5	254.4	400.8	334.9	229.7
	Total time on server	407.1	496	313.7	1985.7	4454.5
	TimeToFetchDataFromAnalysisServer	17.6	93.3	116.9	1371.2	1330.5
	TimeToParseGridStructure	7	9.5	18.8	145.5	728.4
	TimeToPrepareAnalysisServerCall	381.6	391.8	173.4	466.3	1948.9
	Service Call Time	309.6	309.7	423.6	214.2	388.8
	Total Server Processing Time	7.9	10.9	23.4	148.2	1175.1
	Time to Fetch Data from Data Source	399.2	485.1	290.3	1837.5	3279.4
	Other time	4.1	5.7	21	8.6	13.2

13 Appendix B – Test Machine Specifications

Table 9. Specifications of test machine A.

Model	Lenovo 2241WTW
CPU	Intel Core2Duo P8400 @ 2.26 GHz
Memory	4 GB DDR3 RAM 1066 MHz
Network	100 Mbit LAN
OS	Windows 7 Enterprise 64-bit
Web Browser	Internet Explorer 8
Silverlight Version	4.0.50826.0
Web Server	IIS 7.5

Table 10. Specifications of test machine B.

Model	VMware Virtual Machine
CPU	Intel Xeon E54005 @ 2.00 GHz
Memory	4 GB
Network	100 Mbit LAN
OS	Windows Server 2008 Standard SP2 64-bit
Web Browser	Internet Explorer 8
Silverlight Version	4.0.50826.0

14 References

1. **Microsoft.** Introduction to WPF. *Microsoft Developer Network.* [Online] Microsoft. [Cited: Juni 15, 2010.] http://msdn.microsoft.com/en-us/library/aa970268.aspx.

2. —. Windows Forms Overview. *Microsoft Developer Network.* [Online] Microsoft. [Cited: Juni 10, 2010.] http://msdn.microsoft.com/en-us/library/8bxxy49h.aspx.

3. —. Silverlight Overview. *Microsoft Developer Network.* [Online] Microsoft. [Cited: § 15, 2010.] http://msdn.microsoft.com/en-us/library/bb404700(v=VS.95).aspx.

4. —. The Official Microsoft ASP.NET Site. *ASP.NET.* [Online] Microsoft. [Cited: Juni 15, 2010.] http://www.asp.net/.

5. —. How to: Access a Service from Silverlight. *Microsoft Developer Network.* [Online] [Cited: Juni 15, 2010.] http://msdn.microsoft.com/en-us/library/cc197937(VS.95).aspx.

6. —. Silverlight 4 Beta. *The Official Microsoft Silverlight Site.* [Online] 2009. [Cited: Januari 19, 2010.] http://silverlight.net/getstarted/silverlight-4-beta/#whatsnew.

7. **Novell.** *Moonlight.* [Online] Novell. [Cited: Juni 15, 2010.] http://www.go-mono.com/moonlight/.

8. **Usama Alam, Muhammad.** Flash vs. Silverlight: What Suits Your Needs Best? *Smashing Magazine.* [Online] [Cited: Juni 15, 2010.] http://www.smashingmagazine.com/2009/05/09/flash-vs-silverlight-what-suits-your-needs-best/.

9. **Microsoft.** Overview. *The Official Microsoft Silverlight Site.* [Online] 2010. [Cited: October 1, 2010.] http://www.silverlight.net/getstarted/overview.aspx.

10. —. Optimizing Performance: Controls. *Microsoft Developer Network.* [Online] [Cited: Juni 16, 2010.] http://msdn.microsoft.com/en-us/library/cc716879.aspx.

11. **Stollnitz, Bea.** UI Virtualization. *Bea Stollnitz on Silverlight and WPF.* [Online] [Cited: Juni 16, 2010.] http://bea.stollnitz.com/blog/?p=338.

12. —. Data Virtualization. *bea stollnitz on Silverlight and WPF.* [Online] July 26, 2009. [Cited: October 1, 2010.] http://bea.stollnitz.com/blog/?p=344.

13. *Web Site Usability, Design and Performance Metrics.* **Palmer, Jonathan W.** 2, s.l. : Information Systems Research, Juni 2002, Vol. 13.

14. **TRIMMA Affärsutveckling.** *TRIMMA.* [Online] [Citat: den 15 Juni 2010.] www.trimma.se.

15. **Van Den Berghe, Vincent.** *Data Virtualization in WPF and beyond.* s.l. : Bea Stollnitz.

16. **McClean, Paul.** WPF: Data Virtualization. *CodeProject.* [Online] Mars 23, 2009. [Cited: 06 18, 2010.] http://www.codeproject.com/KB/WPF/WpfDataVirtualization.aspx.

17. **Tallet, Paul.** Performance characteristics of the Silverlight DataGrid. *MSDN Blogs.* [Online] April 13, 2010. [Cited: October 1, 2010.] http://blogs.msdn.com/b/mcsuksoldev/archive/2010/04/13/performance-characteristics-of-the-silverlight-datagrid.aspx.

18. *WPF Apps With The Model-View-ViewModel Design Pattern.* **Smith, Josh.** February 2009, 2009, Vol. MSDN Magazine.

19. *Model-View-ViewModel In Silverlight 2 Apps.* **Wildermuth, Shawn.** March 2009, 2009, Vol. MSDN Magazine.

20. **Smith, Josh.** *Advanced MVVM.* s.l. : Josh Smith, 2010.

21. **Anderson, Chris.** *Pro Business Applications with Silverlight 4.* u.o. : Apress, 2010.

22. **RIAStats.com.** Rich Internet Application Statistics. *RIAStats.com.* [Online] [Cited: October 4, 2010.] http://riastats.com/.

23. **Guthrie, Scott.** First Look at Silverlight 2. *ScottGu's Blog.* [Online] Februari 22, 2008. [Cited: October 4, 2010.] http://weblogs.asp.net/scottgu/archive/2008/02/22/first-look-at-silverlight-2.aspx.

24. **Microsoft.** Silverlight 4 Information. *The Official Microsoft Silverlight Site.* [Online] 2010. [Cited: October 4, 2010.] http://www.silverlight.net/getstarted/silverlight-4/.

25. —. XAML Overview. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 4, 2010.] http://msdn.microsoft.com/en-us/library/cc189036(VS.95).aspx.

26. —. UserControl Class. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 4, 2010.] http://msdn.microsoft.com/en-us/library/system.windows.controls.usercontrol(VS.95).aspx.

27. —. Data Binding. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 4, 2010.] http://msdn.microsoft.com/en-us/library/cc278072(VS.95).aspx.

28. —. ICommand Interface. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 4, 2010.] http://msdn.microsoft.com/en-us/library/system.windows.input.icommand(VS.95).aspx.

29. **w3schools.com.** JavaScript Introduction. *w3schools.com.* [Online] [Cited: October 7, 2010.] http://www.w3schools.com/JS/js_intro.asp.

30. —. AJAX Introduction. *w3schools.com.* [Online] [Cited: October 7, 2010.] http://www.w3schools.com/Ajax/ajax_intro.asp.

31. **Project, The jQuery.** jQuey: The Write Less, Do More, JavaScript Library. *jQuery.com.* [Online] [Cited: October 6, 2010.] http://jquery.com/.

32. **Dahm, Tom.** Browser Compability Tutorial. *NetMechanic.* [Online] [Cited: October 7, 2010.] http://www.netmechanic.com/products/Browser-Tutorial.shtml.

33. **Walther, Stephen.** Ajax Control Toolkit. *CodePlex.* [Online] May 5, 2010. [Cited: October 7, 2010.] http://ajaxcontroltoolkit.codeplex.com/.

34. **Guthrie, Scott.** ScottGu's Blog. *ASP.NET Webblogs.* [Online] [Cited: October 6, 2010.] http://weblogs.asp.net/scottgu/archive/2008/09/28/jquery-and-microsoft.aspx.

35. **Adobe.** Flex framework. *Adobe.* [Online] [Cited: October 7, 2010.] http://www.adobe.com/products/flex/flex_framework/.

36. HTML5 differences from HTML4. *World Wide Web Consortium (W3C).* [Online] September 19, 2010. [Cited: October 7, 2010.] http://dev.w3.org/html5/html4-differences/.

37. **Clarke, Gavin.** HTML5's Flash and Silverlight 'killer' potential chopped. *The Register.* [Online] July 8, 2009. [Cited: October 7, 2010.] http://www.theregister.co.uk/2009/07/08/html_5_media_spec/.

38. **Becker, Brad.** The Future of Silverlight. *The Silverlight Blog.* [Online] September 1, 2010. [Cited: October 7, 2010.] http://team.silverlight.net/announcement/the-future-of-silverlight/.

39. **Microsoft.** Windows Communication Foundation. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/netframework/aa663324.aspx.

40. —. Introducing Windows Communication Foundation in .NET Framework 4. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/ee958158.aspx.

41. —. Using Data Contracts. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/ms733127.aspx.

42. —. Hosting and Consuming WCF Services. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/bb332338.aspx#msdnwcfhc_topic2.

43. —. Overview. *The Official Microsoft IIS Site.* [Online] 2010. [Cited: October 1, 2010.] http://www.iis.net/overview.

44. —. ServiceModel Metadata Utility Tool (Svcutil.exe). *Microsoft Developer Network.* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/aa347733.aspx.

45. **thinktecture.** thinktecture WSCF.blue. *Codeplex.* [Online] 2010. [Cited: October 1, 2010.] http://wscfblue.codeplex.com/.

46. **Microsoft.** Configuring System-Provided Bindings. *Microsoft Developer Network.* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/ms731092.aspx.

47. —. Message Encoding. *Microsoft Developer Network.* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/ms731802.aspx.

48. Model View ViewModel. *Wikipedia.* [Online] September 21, 2010. [Cited: 10 1, 2010.] http://en.wikipedia.org/wiki/Model_View_ViewModel.

49. The MVVM Pattern is Highly Overrated. *The Inquisitive Coder - Davy Brion's Blog.* [Online] July 21, 2010. [Cited: October 1, 2010.] http://davybrion.com/blog/2010/07/the-mvvm-pattern-is-highly-overrated/.

50. **Microsoft.** VirtualizingStackPanel Class. *Microsoft Developer Network (MSDN).* [Online] [Cited: October 7, 2010.] http://msdn.microsoft.com/en-us/library/system.windows.controls.virtualizingstackpanel(v=VS.95).aspx.

51. **Syncfusion.** Essential Grid: Powerfull Cell-Oriented Silverlight Grid Component. *Syncfusion.* [Online] 2010. [Cited: November 3, 2010.] http://www.syncfusion.com/products/user-interface-edition/silverlight/Grid.

52. **Microsoft.** Silverlight Tookit. *Codeplex.* [Online] [Cited: November 3, 2010.] http://silverlight.codeplex.com/.

53. **VIBLend.** Silverlight Data Grid Control. *viblend.com.* [Online] [Cited: November 3, 2010.] http://www.viblend.com/products/net/silverlight/controls/datagrid.aspx.

54. **Telerik.** Silverlight Grid Control, GridView Component for Silverlight, Data Grid. *telerik.com.* [Online] [Cited: November 3, 2010.] http://www.telerik.com/products/silverlight/gridview.aspx.

55. **ComponentOne.** Silverlight Grid Control from ComponentOne - DataGrid for Silverlight. *componentone.com.* [Online] [Cited: November 3, 2010.] http://www.componentone.com/SuperProducts/GridSilverlight/.

56. **Wikipedia.** Software performance testing. *Wikipedia - the free encyclopedia.* [Online] [Cited: Noverber 3, 2010.] http://en.wikipedia.org/wiki/Software_performance_testing.

57. **Meier, J.D., et al.** *Performance Testing Guidance for Web Applications.* s.l. : Microsoft, 2007.

58. **Molyneaux, Ian.** *The Art of Application Performance Testing.* s.l. : O'Reilly Media, 2009.

59. **Microsoft.** ListBox Class. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: September 28, 2010.] http://msdn.microsoft.com/en-us/library/system.windows.controls.listbox(v=vs.95).aspx.

60. —. Web Services Protocols Supported by System-Provided Interoperability Bindings. *Microsoft Developer Network (MSDN).* [Online] 2010. [Cited: October 1, 2010.] http://msdn.microsoft.com/en-us/library/ms730294.aspx.