Flexible software solution for automatic maintenance of thick-client systems

Olof Olsson

Department of Electrical and Information Technology
Lund University

Advisor: Anders Ardö (EIT), Richard Houltz (Cenito Software AB), Johan Nilsson (Cenito Software AB)

February 15, 2013

Printed in Sweden
E-huset, Lund, 2013

Abstract

Many distributed systems are implemented with very simple thin clients than can operate without much locally bound software. This makes them fairly easy to update since the application is stored on a central server. When on the other hand a distributed system need to access local hardware, a thick client using local software must be used.

To enable surveillance and maintenance over thick client systems, it is necessary to automate the update and surveillance procedure with software. This software should be able to control and survey the behavior of unique installations and be dynamic enough to maintain and survey any thick client. The software needs to be flexible in its design to allow support for multiple system types.

The purpose of this study was to propose a design for a system that could enable cost-effective maintenance and surveillance of a distributed thick client system. To develop a possible design for this system an article study was conducted into usable design patterns and frameworks. The proposed design was also evaluated by the development of a prototype system. This system was tested on a thick client where its usefulness could be evaluated.

Acknowledgements

I would like to take this opportunity to thank my advisors at Cenito Software AB, Richard Houltz and Johan Nilsson, and my advisor at LTH Anders Ardö. I would also like to thank my colleagues at Cenito Software AB for all the interest shown in the study and the advice given.

Table of Contents

1 Introduction **1**
1.1 Environment . . . 1
1.2 The prototype . . . 2

2 Pre study **5**
2.1 Extensibility . . . 5
2.2 Communication . . . 7
2.3 Queueing . . . 9
2.4 Data storage . . . 12
2.5 Updating the thick client . . . 14
2.6 Heart beat . . . 15
2.7 Publishing data . . . 15

3 Possible solution **17**
3.1 Prototype specification . . . 17
3.2 Extensibility . . . 18
3.3 Queueing . . . 19
3.4 Communication module . . . 20
3.5 Data storage . . . 21
3.6 Updating the software . . . 21
3.7 Publishing data . . . 22

4 Final design **23**
4.1 Plugin structure . . . 23
4.2 Client . . . 23
4.3 Heart beat . . . 24
4.4 Server . . . 25
4.5 Publishing data . . . 25

5 Implementation issues **27**
5.1 Duplex WCF with MSMQ bindings . . . 27
5.2 Choosing solutions for maximum extensibility . . . 27
5.3 Database design for generic data . . . 28

6 Testing the prototype on a thick client 29

7 Conclusion 31

8 Future development 33
8.1 A better user interface 33
8.2 Filtering of accepted data in the server 33
8.3 Simplifying the plugin development 33

9 Terminology 35

References 37

Chapter 1

Introduction

Many distributed systems today rely mainly on so called thin clients, logic-less user interfaces that mainly communicate back to smart servers. These thin clients has some great benefits from a viewpoint of maintenance as they can be updated or changed without major downtime. They also have a smaller Achilles heal than its opposite, thick client, as their only dependency is that the server and the communication line are online. These thin clients do however have a limitation to what they can be used for and when more complicated logics or hardware is needed in the client application there is a need for a distributed client application that also has to be maintained and surveyed. As these thick client can be of some complexity it might not always be a simple task to keep them up to date and under surveillance. Without any way to view the system operate the thick clients are left on there own in case of a malfunction or unexpected change in operation that might not be visible to someone taking care of said system. Maintenance and surveillance systems for servers are not uncommon and while specialized surveillance systems for thick client services exist they are limited and specialized. For a company maintaining several types of systems, all working on distributed thick clients, the issue of surveying these machines might be a costly one. I have prior to this study worked for Cenito Software with such a distributed system and I have through that experience seen that there is indeed a huge need for a system that can help with the maintenance and surveillance. The issue with a system such as this is that the kind of data that is interesting to survey in the thick clients might change rapidly. It might one day be of interest to collect data specific to a current bug and another day to collect data regarding the usage of the system. In this case there is a need for a dynamic and changing system that can track data in any way the user sees fit.

1.1 Environment

Cenito works with .Net and C# as their preferred language and have therefore requested that the prototype, developed for testing in this study, should be built in the same environment. This affords the prototype the benefit of having a first testing ground in one of the Cenito projects; a distributed kiosk system. .Net comes with some suitable features for the server hosting, namely Azure, which was to be considered as a potential hosting environment. The final testing environment for the prototype is a mix of windows versions that runs a sun tanning payment system. It will there collect suitable data and communicate that data to the server. The data collection will be superficial and will hopefully show the final potentials of the system.

The user interface will for practical reasons be built for standard web browsers. It will not contain more features than is necessary for showing the potential of the server and client. For the testing purposes, no sensitive data will be collected as to not force the user interface to be run with security.

1.2 The prototype

To test the theories of a fully agile and extendable surveillance system for generic distributed systems a prototype was considered. A system such as this might work great in theory but when moving into the realm of practical application the only way to really test the theory is by prototyping. To start the prototype some limitations had to be set both because of the limitation in time and because of the span of this thesis. The main focus was put on the extensibility and as such some features had to be left out. Other important aspect of a potential final product, such as security and remote controlling of the host machine had to be left for a future update of the system, outside of this thesis scope.

1.2.1 Requirement specification

For a finished product to meet the expectations and uses, the following use cases needs to be considered. The use cases are specified for a finished product and all of the following use cases will not be implemented in the prototype software developed for this study. The use cases that will not be considered for the prototype is mentioned in more detail together with the prototype specification (See chapter 4.1). The use cases has been revised several times and their numbering is constructed from an id and version number separated by a dot.

General requirements

- 1.3

  The systems user should be able to schedule or push updates to the thick client.

- 2.3

  The user should be able to see information about the thick clients hardware status. Hardware information therefor needs to be collected from the thick client.

- 3.2

  The user should be able to view activity information from the thick clients to know if the thick clients is running correctly.

- 4.2

The user should be alerted if any configurations or software is changed from an outside source. The user should also be able to view working configurations and software versions from any thick client.

- 5.3

  The user should be able to plug in an advanced information collector to the thick client and view the collected information.

- 6.2

  The user should be able to view log files from the thick client on demand enabling fast error tracking. The log files therefore need to be collected and uploaded automatically.

- 7.3

  The User should automatically and on demand get potential malfunction alerts from sources found in the log files. The log files therefore needs to be analyzed automatically or continuously.

- 8.2

  The Thick Client needs to update itself automatically to allow the user to use new features and security updates.

- 9.2

  The user should be able to schedule and run tasks on the thick client such as backups.

- 10.2

  The system should be able to run independent of hardware and software of the thick client so that the user is able to combine information and status collection for multiple different systems.

- 11.2

  The user should be able to set thresholds for alarms so that the user can decide how involved he will be in maintenance.

- 12.2

  The user should be able to sort out information from the thick clients and view only what is wanted

- 13

  The user should be able to see system status and information from the thick clients in real time so that malfunctioning systems can be spotted immediately

- 14

  The user should be able to view information and system status through a web page for easy access.

- 15.2

  The user should be able to push a message making the thick client update itself

- 16.2
  The user should be informed by mail if a thick client is malfunctioning.
- 17.2
  The user should be able to get notifications by push

Chapter 2

Pre study

Before starting on the development of this project some effort was put into the search for similar projects and the documentation of such. The search did not reveal any documentation of fully developed projects in the same area of functionality and design but a more detailed search in the areas of extensibility gave much to build on. During the pre study the goal of the project shifted slightly in the direction of extensibility as the research opened up new grounds for what could be possible. The bigger part of the software seemed to theoretically work in a module model where close to all parts could be exchanged depending on the underlaying system from which the software would be run.

2.1 Extensibility

"In software engineering, extensibility (sometimes confused with forward compatibility) is a system design principle where the implementation takes into consideration future growth. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The central theme is to provide for change - typically enhancements - while minimizing impact to existing system functions."[6]

For a system to be able to survey another independent generic system the surveying system needs to be either very general in its surveillance or very easy to modify for detailed surveillance. As the main aim of this paper is to explore if a surveying system can gather detailed data from an independent generic system a high priority is put on making the surveying system highly extensible.

Basing the system on a plugin model enables us to exchange the parts of the system that collects information and therefore makes our software able to survey any system we can write data collection plugins for. Plugins could solve the most necessary extensibility problem that we face and it could also enable us to loosely bind any part of the system thus enable users to choose the parts any way that they would want.

The first step to create the desired extensibility in the system is to look at inversion of control and dependency injection.

2.1.1 Inversion of Control containers and Dependency Injection

“In software engineering, Inversion of Control (IoC) is an object-oriented programming practice where the object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis. ”[10]

The main aim of IoC containers is to decouple systems. Loose bindings between objects creates more flexible and manageable design which opens up for change and enables mocking of modules in testing.

Dependency injection was first introduced by Martin Fowler in his article about the pattern[14] and was by the new term separated from the general Inversion of Control that is very common in software engineering.

Dependency injection is a concept or pattern that we can derive from Inversion of Control. We do not always want to let an object be responsible for all its dependencies. Therefore we inverse the control and let the host of the object inject the dependencies into the object.

Dependency injection is a software design pattern that allows a choice of component to be made at run-time rather than compile time. This can be used as a way to introduce third party add-ins, or plugins. Dependency injection can also be used to support different kinds of runtime environments. As an example it is possible for an application to use more than one database by letting the logic handle the data transactions against a contract that would have several implementations with the specific database logic. These implementations could be inserted into the system by use of dependency injection.[12]

“At its core, IoC, and therefore DI also, aims to offer a simpler mechanism for provisioning component dependencies and managing these dependencies throughout their lifecycle.”[7]

With the use of the Dependency injection pattern we can create a suitable plugin support for the prototype and get the extensibility needed for the system.

2.1.2 Suitable IoC containers

There are several IoC containers that should be considered for the implementation of the prototype. There is a vast amount of available IoC container frameworks and not all of them can be discussed in detail in this study because of time limitations. A number of frameworks was selected from the book Dependency Injection in .Net[30] for a closer look and as the wanted features was found, no further research was conducted.

Unity

“The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection.

Unity addresses the issues faced by developers engaged in component-based software engineering. Modern business applications consist of custom business objects and components that perform specific or generic tasks within the application, in addition to components that individually address cross cutting concerns such as logging, authentication, authorization, caching, and exception handling.”[9]

Unity tackles the Dependency injection pattern very well and is able to create a very loosely coupled design. It does however compose its components by explicit registration. It is therefore not optimal to use for a plugin pattern.[13]

Managed Extensibility Framework

Managed Extensibility Framework (MEF) is not strictly speaking an Inversion of Control container as it does not depend on a clear definition of the dependencies that it procures for the target. However in our case it acts as any other Inversion of Control container in the sense that it provides the dependencies that are needed.

MEF composes the components by way of filling components marked with the attribute [Import] with other components marked with the attribute [Export]. The importing components signals which imports it needs by adding the contract the export needs to fulfill in its import attribute. The exporting component fulfills contracts by implementing interfaces representing the contracts. Each import is matched with a list of exports by way of its contract by the composition engine.[1]

MEF can be used for dynamic discovery of components. An application using MEF could, with a DirectoryCatalog, be designed so that simply putting compiled code, in the form of dlls, in that catalog enables the application to find all classes marked by the export attribute. This makes for a very flexible plugin structure.

Other dependency injection frameworks

There is some other dependency injection frameworks worthy of a mention such as Castle Windsor, StructureMap and Autofac. These framework does however like Unity need a explicit registration of the components and is therefore not as suitable as MEF for a plugin pattern.

2.2 Communication

2.2.1 Possible issues

The main feature of the system could be said to be the data communication. It is also of the highest necessity that the client can work unhampered by the statues of the server and vice versa as to be able to continue collect data even in situations where the server is not responding. The following list describes some potential communication problems that have to be considered in the choice of solution model.

- Network issues; the network is down and the message can not be sent from the client to the server or vice versa.
- Dropped connections; A package is sent after the client checks that the connection to the server is up but is then immediately dropped resulting in a lost package.
- Unpredicted lost package through unknown error; If we can not be completely sure of all parts in the communication between the client and the server we have to protect the communication from the unpredictable errors

as well. The client and server both should be able to keep running and functioning as normal even in the advent of a lost package.

- Out-of-order messages; The packages that are sent from the client arrives in the incorrect order therefor changing the intended result of handling those packages.[2]

As the system will be relying information to the surveilling user, a proper reliable and robust communication has to be used between the data collecting clients and the data analyzing and storing server.

Three main solutions were considered for the communication, namely windows communication foundation, TCP/IP socket communication and ASP.Net. A fourth alternative with UDP communication was not considered as a possible solution due to the likely dataloss in UDP communication.

2.2.2 Socket communication over TCP/IP

A socket communication would work very well for the communication between server and client but a protocol for the communication would have to be built to handle both the sending of collected data and the update push communication from the server. Setting up such a protocol and data communication would be a very big and unnecessarily complicated task in itself, therefor a shortcut is needed for this prototype.

Another issue with using socket communication between the client and the server would be the amount of connections that would have to be opened and maintained if there were to be multiple clients.

On the whole, socket communication seems to be a possible solution but to create the robustness and flexibility in that communication a considerable amount of work would be needed.

2.2.3 Windows communication foundation

Setting up a web service with Windows Communication Foundation (WCF) is a much simpler task than building a communication protocol for socket communication from scratch and therefor enables this study to focus more on the extensibility part than on the communication problems. WCFâĂŹs implementation uses sockets over TCP/IP in its lower levels but takes care of the difficulties and issues mentioned above so that the exposed interface towards the developer is far simpler to use. WCF therefore enables building the communication part of the prototype in a very extensible and agile way. The interface for communication is communicated with a XML schema and it is therefore possible to build clients potentially independent of platform. The WCF web service comes with a number of easily configured bindings and the potential to create new ones based on a wanted protocol.[3]

Through web services, Microsoft wanted to create a compromise between web development and component-based development. Web services were a step toward service orientation, which is a way to develop loosely coupled distributed applications.[3] WCF allows the client and server to work without any strong binding between each other and the two parts of the communication can therefore work independent of the other parts accessibility. there is also no need to keep a process running with an open socket for communication. These features solves the loose binding requirement of the prototype.

Running the server as a web service would enable minimization of the runtime of the server. It would only wake when a message was received and that could depending on the clients save a lot of runtime on the server. WCF also supports concurrent calls[5] which will be a necessity if the number of clients and plugins is large.

2.2.4 Web Api with ASP.Net

A simple web api could be set up and hosted with ASP.Net as communication between the server and the clients. ASP.Net is a framework that simplifies the building of HTTP services which can broaden the range of possible clients to browsers and mobile devices.[24]

Communicating with simple post and get verbs against a web server looks at first glance as a practical solution. The communication work only in one way however and the pushing of an update from the user interface necessitates that the clients hosts their own web server or other technology for receiving the push command. This set high demands on the openness of the thick clients network which has to allow for such a hosting. A web API could however be used as part of the communication solution as both a method of publishing data and potentially for posting data to the server.

2.3 Queueing

If the server where to be shut off for a moment or the network between the server and the client where to falter in its availability it is important that the clients collected data is not lost. It is therefore necessary for the client to hold on to the collected data until it is certain of the availability of the server. The best solution to such a problem is arguably a message queue where in the clients can deliver its data until the client can send the messages.

For the development of the queue some options must be considered. If the queue is run as a separate module it is possible to choose which queueing technique that is wanted. There is however some difficulties to deal with if the queueing is to be a built in part of the communication. WCF does give the option to create a binding using any queueing system but doing that would take a lot of time away from the rest of the prototype development and in the end it might not be any significant improvement over using the provided queue, MSMQ. This paper will not go deeper into exploring the benefits of the individual queueing systems but will instead explore if the queueing should be a part of the communication module or if it should be separated into its own module.

To evaluate whether the stand alone queueing module or a closely coupled queueing and communication module should be used the following benefits of the two options were considered.

The separated module gives us the possibility of simply exchanging the queueing module in the client and thus enable the server to keep running without changes. This certainly makes the client far more agile in the design and could be very useful if the client are to be run on a different platform than currently targeted. We could see the separated queueing module as a part of a pipe between the clients collection and communication modules, enabling us to design for more “pipe modules” that could be used to alter the collected data in preferred ways. i.e. a compressing module could be inserted in that pipe together with the queueing module.

The integrated queue and binding solution model would remove one error prone part of message handover and in some cases simplify the search for errors if communication would stop working. Using the queue as a part of the binding instead of a module in between the collection and communication also enables us to use some of the features that is part of the web service. We could let the plugins target specific communication methods depending on priority and content.

2.3.1 Microsoft Message Queuing

Windows comes with its own built in queueing system that applications can hook into. Applications can with Microsoft Message Queuing (MSMQ) create and control their own queues with a very small setup cost. MSMQ is also integrated into the WCF solution as a binding (see WCF bindings) and is therefore well suited for lossless communication between a client and a service.

Using MSMQ as a queueing buffer between the client and service and only sending the messages from the queue when the server is responding guarantees that no data will be lost because of network issues. If the network connection is down or the server is unresponsive the client will simply store the packages in the queue until the network connection is available and the service can be reached.

The MSMQ queue size is expandable as far as the disc hosting it allows, making the discs size the limitation of the queue. The client is not meant to be run without a connection to the service for a long duration but if someone wants to use the system as such, MSMQ will not be a showstopper.

It is very important for the client and service to be loosely coupled in its communication so that the client can continue to collect data even if the service or the network connection goes down. MSMQ enables the client to release the responsibility of communication and focus on the data collecting.

Another benefit of using a queueing system is that we can separate the client from the server even before mixing in the communication modules. The client will simply put messages in the queue and let the queue handle the handover to the server by which ever method chosen. This means that some failure isolation is achieved and that potential errors will be easier to spot.

MSMQ does however come with a maximum for message length and it is therefor necessary to make sure that packages over the maximum size get split up before communication with the server as well as put back together at the arrival.

2.3.2 WCF and MSMQ

A direct issue with using MSMQ as part of the binding is that it does not support duplex communication.

The server part of the prototype needs to be able to push a “update call” to the registered clients. This call is necessary for forcing an update of the clients software which is one of the requirements of the product. The duplex call is therefore a necessity and the inability to use it has to be solved by adding a second communication line between the server and the client. This solution would let us keep the robustness and reliability of the MSMQ binding for the pure data delivery from the client to the server while still using the duplex callback for the registration and callback from the server.

We could not guarantee that the registration call or the update call would be delivered safely as it would not be using the queue. Both the registration call and the callback would however both be very small messages, using minimal bandwidth and could potentially run repeatedly as to make sure that the client is registered.

Using two separate services for the communication does not result in a design that is as clean as a design with only one service and it also introduces a potential error source.

2.3.3 Queueing options

MSMQ is not the sole way of achieving reliability in a WCF service communication. MSMQ is supported by WCF out-of-the-box but one could build a transport binding with other queueing systems. MSMQ is well supported by windows but could prove to be a problem if the client was to be run on another operating system. Another queueing system could potentially solve that problem.

2.3.4 WCF and MSMQ in separate modules

The queueing could be handled separated from the WCF communication on the client side with a small module stuck between the collection of data and the communication of said data.

This solution would potentially give the client a seemingly more extensible design and if the coupling between the collection, queueing and sending were built with the same extensibility framework as used in the other parts of the prototype the client could switch the queueing method without changing any parts of the communication module.

The separation would however also potentially lead to one more error creator. It could become harder to find errors between the collecting and the receiving of the data in the server. If a message were to disappear we would have to investigate yet one more area as the hand over of the messages between the MSMQ and web service is a separate procedure.

Further, the minimized runtime on the server that WCF would allow is a great feature for a service.

2.3.5 Integrating the queue in a WCF binding

While the MSMQ binding seems simple at first glance the added complexity of having two services for supporting the callbacks makes this solution cluttered compared to keeping the queueing as a separate module.

Separating the queueing into a module allows the client to change queueing without changing the communication module and therefor lets the client owner decide which queue is suitable for the platform.

The separated queueing module also opens up for the communication with non .Net clients which enables future development of use case 10.2 (See chapter 2.2.1).

2.4 Data storage

The data storage is a central part of the system and it is important that the data layer provides robustness and flexibility. If the service was allowed to talk directly to a database we would loose the flexibility and would be unable to switch the database model if that need arrises. As the plugins would talk directly to a set standard of the database we could not exchange, upgrade or in any way change the database without having to ask plugin developers to rewrite their plugins.

The solution to this problem is to separate the database with a layer of abstraction.

2.4.1 Repository pattern

“A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer.” [15]

A repository should be used to separate the logic that moves data in and out of data storage from the logic acting on the model of a system to create a loose coupling in an application. Making the business logic agnostic to the data types that makes up the data source layer enables the data storage to be build from any technology such as a database or a web service without affecting the logic acting on the data.[16]

Separating the data storing structure from the data handling plugins creates the flexibility needed in the server as it enables the change of data storage without any effect on the plugins using the repository contract to communicate with the data storage. Using a repository pattern together with a dependency injection such as MEF enables the switching or multiplication of datasources. It could therefore be possible to store the data to a database while at the same time posting the data to a second server without making any changes to the server except adding the extra data storage logic classes, implementing the repository contract, to the plugin folder provided by the dependency injection framework.

2.4.2 Choice of database

Saving the collected data from a dynamic range of collectors and potentially third party developed collectors demands a very loose way of storing the data. The type and format of the collected data will from the point of the database be unknown and that necessitates the choice of a data structure that is very flexible.

When choosing the database three categories; Sql, NoSql and XML will be considered. Picking the suitable technology and its suitable data model is a difficult job as the choices are vast. Both the NoSql databases and the XML seem at first glance to be good choices as they both provide a very high degree of flexibility and extensibility. On the other hand, using SQL might be a good way to go as it is both very robust and well supported by .Net and Azure and with the use of a blob field can provide the flexibility needed.

Structured Query Language and relational database management systems

SQL or Structured Query Language is a programming language specially designed for managing data in relational database management systems (RDBMS).[20] Relational databases are fairly simple to use and well supported in both free and for-pay versions. Using a standard SQL syntax to access the stored data is both simple and fast as long as the complexity of the database model does not grow to big. Relational databases are designed on the assumption of that the data inserted into it is of known scheme and for that kind of data they are ideal both in simplicity and in speed. However, when it comes to storing data of dynamic schemes or dynamic sizes they tend to complicate the usage. If the data is stored with a simplified scheme of some important keys for easy fetching of data and a large field for the storage of said data, the database becomes simple to design and use but will need to use a xml based field for the data as to make it usable to the publishing plugin. This is so that the plugins, which uses an unspecified data scheme for its data, will be able to manage and query the data directly when fetched from the database.

The .Net framework comes with some great advantages for coding for relational databases. Entity framework which is a part of the .Net framework since NET Framework 3.5 Service Pack 1[21] simplifies the use of a database as a data model greatly by mediating between the code and the database. With another addition of Code-First development, included in .Net 4[22], the database can be generated and maintained automatically from the coded data model. Simply creating classes that makes up the model without any requirements for inheritance of database logic is enough for automatically defining the tables for a relational database[22]. Further, there is great support for a automatic deployment with code first into Azure SQL making the development for hosting the server with azure a breeze.

To solve the issue of having no proper freedom regarding the scheme the data that is entered into the database, a field for saving blobs of xml could be used in the SQL database. This would supply a way to search the data and a way for the collectors of data to take on the responsibility of how to arrange the collected data. In this case a xml blob of data would be saved together with a couple of search friendly keys such as a reference to the collector of the data, a date and other information letting the publishing of the data access interesting entries easily.

NoSQL database

"NoSQL database systems are developed to manage large volumes of data that do not necessarily follow a fixed schema."[19]

When discussing NoSQL databases it might be wise to divide these into two categories; the document-style stores and the key-value stores.

The key value stores are designed to store a value and its key as a pair with the use of, most commonly, distributed hash tables. These databases are not of much interest to this study as it is suited for a very simple data structure.

The document-styled databases however, are indeed of interest as the allow for the use of very dynamic data in both size and scheme. The document-styled databases are usually used to store a collection of key values together with a payload of data. Examples of such databases are CouchDB and MongoDB. [23]

Using a NoSQL database allows entering data in any format while still being able to fetch data by keys that the plugin developer could specify. This could allow the collection of data a lot of freedom in what kind of and what size of data that is collected and later stored.

XML database

Much like the NoSql Databases, the XML databases stores data in free format while still keeping the data highly searchable[31]. The database stores blobs of data structured in XML tags and can therefore query the data based on key values entered together with the data much like in the NoSQL databases. In the paper Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage, the two data storage approaches are compared and the main difference between them seems to be the slightly faster querying speed of the NoSQL database. The paper further states that both the database approaches works very well with non relational complex structures of data[32].

2.5 Updating the thick client

Today most applications have the capability to update themselves or uses services such as app stores or apt get to handle its download and installation procedures. However, to fulfill one of the use cases specified the prototype needs to be able to not only update itself but also other software on the thick client. The update procedure must also be specifiable from client to client as the software to update and the availability of the software might vary from system to system.

Another requirement of the application is that it must be possible to activate the update by a forced call from the server. The important part to stress here is that it will not be the client that decides when to update but the server. The registration and callback was briefly examined when looking at the communication module and that callback needs to be extended to produce the update by the client. As all other parts of the system the updating part could be loosely bound and changed as one or several plugins. If a thick client was to run several systems that required forced updates it would be wise to separate the update procedure into several updaters. These updaters could be started by one or several updater modules in the client.

Updating the surveying client software and thick client software is potentially a big security issue. If an update is pushed by the system another hostile system could potentially do a man in the middle attack and force the thick client to run malicious software. Such an attack could be made much more difficult by letting the thick client download and install its own update at the order of the server. Only a pushed call telling the thick client to start the updating procedure would be needed from the server in this solution. The client could then access the update in a more secure way using a web based deployment strategy. The web based deployment strategy would enable the thick client to access the new version of the system to download and install it from where it is securely hosted. This update procedure could be implemented with a separated system using technologies such as ClickOnes[26], Java Web Start Technology[27] or Zero Install[28].

2.6 Heart beat

The most basic part of the surveillance of a distributed system would arguably be live information about which clients are online and talking to the server. To implement this a simple heart beat could be used. The heart beat would be a very simple package of data that is sent to the server.

The heart beat could be separated out into a single plugin or be included into the client as a standard feature. If the heart beat was to be implemented into the client the user would not be able to remove it or change its behavior unless by configuration if that was to be made available. This could potentially be an unnecessary bandwidth use although a small one. The inclusion of the heart beat would on the other hand be a great way to make sure that the communication between the client and server is functioning and to cut out the communication as a potential error source in case of plugin malfunction. Depending on the implementation of the communication module the inclusion of the heart beat might also enable the usage of a higher or lower communication priority, depending on the systems needs.

2.7 Publishing data

The display of the collected data is very important for a product based on the prototype but for the prototype a simple way of displaying data through a web API will suffice. To limit this study to the essential parts of the prototype no direct research has been conducted into the area of publishing the data. Some tips has been offered from advisors and colleagues and the most appealing, in its simplicity, is the lightweight framework, Nancy[29].

Chapter 3

Possible solution

3.1 Prototype specification

All of the use cases does not fall within the frames of this paper and therefore some of them will be left for future development of the product. The prototype designed and developed for this study will, however, have to take all the use cases into account in the design for it not to make the implementations of the excluded cases impossible in the future development.

the following cases will be excluded from the prototypes specification.

- 9.2

  The user should be able to schedule and run tasks on the thick client such as backups.

  Enabling the user to schedule tasks or change configurations adds a lot of complexity to both the service that has to communicate the changes to the client and to the administrative interface that the plugins have to develop. This case is of interest for a final product but does not impact on the feasibility of building this system. Therefore this case will be excluded from implementation in the prototype.

- 10.2

  The system should be able to run independent of hardware and software of the thick client so that the user is able to combine information and status collection for multiple different systems.

  While the design of the client does not exclude any platforms the choice of using .Net does. The client could possibly be developed using other frameworks but it would take far more time and it would not be necessary to prove the feasibility of the product.

- 16.2

  The user should be informed by mail if a thick client is malfunctioning.

  While possibly being an excellent feature, this use case does not impact the feasibility of the product and should therefore be excluded from the prototype to save time for development of the higher prioritized cases.

3.2 Extensibility

"You use MEF to really manage a set of unknown things, you use IoC containers to manage a set of known things." - Glenn Block[8]

The process of importing third party plugins as dlls is something that MEF was built for and while Unity also has support for the task the configurations are complex and therefore unsatisfactory. MEF handles all the requirements for the plugin structure and can be reused in the loose binding of the clients modules as well as the service modules. It enables a repository pattern for the data storage and opens up for further changes to the system.

The main difference between using Unity or MEF in the prototype is how the classes are registered for composition. In Unity each class that is needed in the composition is explicitly registered in a UnityContainer for later composition as in the following example:

```
var container = new UnityContainer();
container.RegisterType<IFoo,Foo>();
container.RegisterType<IBar,Bar>();
...
var program = container.Resolve<Program>();
program.Run();
```

When using MEF the available classes for the composition is instead marked with an Export attribute directly in the class implementation.

```
[Export(typeof(IFoo))]
public Foo
{
    ...
}
```

The system will be set up so that the client and server can collect plugins from extension folders in the installation path. On the client side users of the system simply needs to compile the collector plugin and put it in the extensions folder. The system will take care of the rest. On the server side the data handler plugin and the data publishing plugin will both be uploaded to the extension folder and the server will be able to use the new plugins at the next call from the client. Identification for the plugins will need to be set by the developer of the plugin. A default identification will be generated for the collector plugin but it will need to be registered in the data handler and data publisher plugins for the server to be able to communicate the data to the correct handler plugin.

3.2.1 Building plugins with MEF

The system has three major actions that will be of interest to the user; The collection of data, the handling of the collected data at the server level and the presentation of that data. These three actions, while sharing the data type, has three very divided responsibilities. It is therefore reasonable to split these parts into the different contracts that the plugin developers needs to implement. The development of a plugin function is therefore done in three different stand alone plugins all sharing their identification.

Collector plugin

The client will create a thread for the collector plugin to run in and let that plugin access the queue module for delivering its collected data. How the collection is done is up to the plugin developer and as it runs in a separate thread there will not be any direct restrictions enforced by the client. The clients collection modules only responsibility is to gather and start the plugins and will there after not meddle in the way the plugins use their given thread. The responsibility of keeping the plugin to a low usage of the thick clients resources is left to the plugin developer.

Data handling plugin

The data handling plugin will be gathered by the web service when it is called by a client. The service will hand over the data received from the client to the plugin corresponding to the identification of said data. The data handling plugin is able to use the data storage module to save the received data. The data handling plugin also has access to the alarm module in the service enabling the service to send out an alarm to the user in the way specified by said alarm module.

Data publishing plugin

The publishing plugin is responsible for formatting and displaying the data corresponding to the plugins identification when prompted by the user. This structure frees the administrative system of the responsibility of knowing each plugins purpose. The graphical user interface will simply prompt the plugin, that is called on by the user, for information and the plugin fetches the data needed from the data storage, formats the data and returns it to the administrative user interface.

Other plugins

There might also be a case to argue for making the alarm module open to plugins from the user and maybe even the data storage module. Changing or doubling up on these modules behaviors might strain the server a bit but could in many imaginable cases be well worth it. This however is not a part of the first prototype.

3.3 Queueing

While using MSMQ as the queueing module will only work as long as the client is run on windows it carries the benefit of minimal setup. MSMQ works great for a system running on windows and as the prototype is mainly meant to be run on a thick client using windows for the testing, the benefits of simplicity outweigh the issue of being bound to windows in the development of the prototype.

In a future platform independent product another queueing system would work just as well for our needs. No specific MSMQ dependent features will be used and the design of the software will not be reworked based on which queueing system is used.

The queueing will be a separate module as part of a pipe between the collection module and the communication module and that pipe could in the future also enable the user to insert or change layers to the client.

Using the queueing module as a separate module enables the design to later be made a platform independent product as it is possible to exchange the queueing module without affecting the communication with the server.

Using the queueing module in this case enables the collection module to be completely separated from the communication and as such only deposit its collected information in the queue and with that hand over the responsibility to the other modules of the client. This design also enables us to run the client with a mocked testing queue for testing in case of lost packages. This will be very useful in the process of finding potential errors.

MSMQ allows for the use of durable storage and is therefore well suited for use where packages can not be allowed to disappear. This however also means that if the client is run for to long without any connection to the server and with too heavy data collection the thick client will fill up its data storage and potentially crash.

3.4 Communication module

In the pre study the different possibilities for the communication module was discussed. The suggested design of the prototype falls well into the use cases of the WCF web service.

The web service solves the issues of having multiple clients on different platforms communicating with the server at the same time. It also allows for the deployment to hosted servers such as Azure.

Because the web service communicates based on a simple interface that can easily be communicated to the clients we get a simple and agile design and it enables us to update the web service without major repercussions.

Due to the choice of keeping the queueing module separated from the communication binding, one web service suffices. The web service will communicate over http or possible https and support only a small amount of exposed methods. The service will be of duplex type and accept a registration call or a heartbeat that registers the clients as working and listening for updates. The web service will be accepting calls from the client in form of a package of binary data and an identification of the plugin that collected the data. The web service will at the point of receiving this package forward it to the parsing plugin corresponding to the identification.

The web service exposes a contract to the clients that can be read with a meta data call. The methods specified in the contract is implemented in the web service and clients are able to call them through the specified binding. The callback uses a second interface specified inside the previous contract. With the second interface a method is called by the server to force the update on the registered clients.

The callback contract needs to be implemented on the client side and the way the client chooses to update its software is specified by the client.

3.5 Data storage

When using the repository pattern to interact with the data we can relieve the plugins collecting and handling data from the responsibility of knowing how the data is stored in the database.This also enables the change or addition of different data storage technologies.

Designing a repository contract that can switch the entire scheme of the data layout is not a simple thing to do. To go from a relational data model to a document styled model will not be a difficult task as the rules of the data model becomes simpler and the data would fit into the new database model without changes. If choosing to go the other direction, from document styled to a relational data storage, it would be hard to create a repository pattern that would allow for that change without also affecting the way the data handlers, and data publishers operate on the data.

The flexibility of the NoSQL and XML databases is certainly a great benefit for the extensibility and dynamics of the design. However, due to my prior knowledge of SQL databases and the availability of Azure SQL when hosting the prototype server on Azure, the SQL approach is suitable for implementing the prototype.

Azure also provides a document storage or blob storage but as it only admits key-value inserts it will not suffice as a storage model for the prototype. However, it might be a useful compromise to use both Azure's provided SQL database and its blob storage and reference the data blobs from the SQL tables, thereby allowing the data to use a dynamic scheme inside the blob.

For a future product it would be greatly recommended to use a NoSql or XML database as a data storage implementation as the two technologies provide ideal flexibility and extensibility for the stored data.

3.6 Updating the software

"Two advantages of Zero Install over more popular packaging systems are that it is cross-platform and no root password is needed to install software; packages can be installed in system locations writable by that user instead of requiring administrator access. Thus, package installation affects only the user installing it, which makes it possible for all users to be able to install and run new software."[17]

Because of the advantage of platform independency and not needing root password Zero Install might be the preferred system to use for the update. However as the thick client that the prototypes testing is aimed at already has an updater written in ClickOnes this one will be used for testing and evaluation purposes.

3.7 Publishing data

Nancy is a small framework created to supply a very lightweight and simple http based service to .Net. How the publication of the collected data is best handled is not a part of the scope of this study and therefore Nancy was chosen for demo purposes because of its simplicity. The user interface will be set up simply as a regular webpage displaying information from the active plugins.

Chapter 4

Final design

The final design was focused on the extensibility features and the choices for the technologies have been described in the previous chapter. To handle the essential extensibility feature, namely the possibility for users to write their own data plugins, the client and server is decorated with three points of access for a third party developer. The plugins that handle data is divided into three contracts that have the responsibility for the collection of data, the saving and usage of the data, and the publishing of data. Both the server and the client will be set up to read its plugins from a specified folder wherein the user can add dlls of plugins implementing the plugin contracts.

4.1 Plugin structure

The collector plugin is responsible for collecting the data of interest from the thick client. It might be by file reading or other kinds of surveillance and will not be limited by the client. The collector plugin will also format the data collected and put said data into a message queue provided by the client for the transfer of the data to the server. On the server side two contracts have to be implemented. One contract for the handling of data received from the client machines and one contract for publishing this data. The handling plugin will be responsible for saving the data of interest to the databases provided data storage through a repository pattern. The handling plugin is also responsible for acting on the collected data by doing calculations or alarming the user by a provided alarm contract. The publishing plugin is consulted as the user request information about a system and has the responsibility of providing formatted data in a way that can be displayed by the user interface.

4.2 Client

The client is modularized into three parts. The collection part, which collects and starts the plugins; the communication module that is responsible for sending the messages in the queue as soon as there is a connection to the server, and a message queue, which relays the messages from the collection plugins to the final messaging part. The data collection module has a very small area of responsibility as the actual data collection is handled by the plugins supplied from outside dlls.

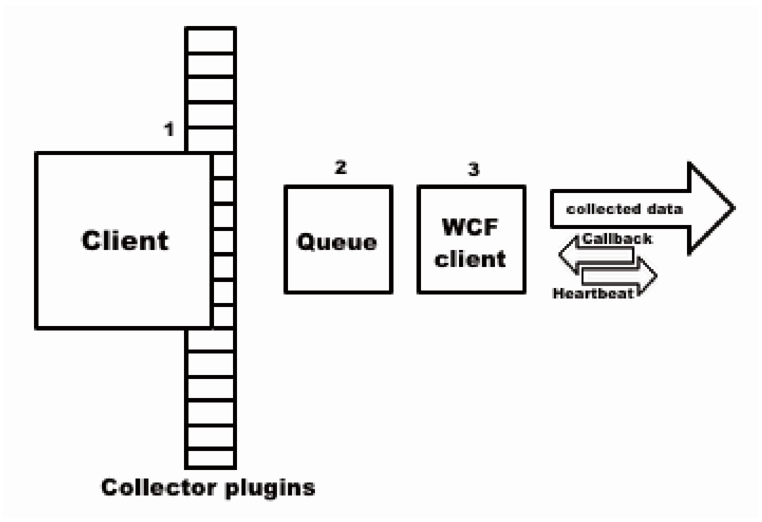

Figure 4.1: Overview of the client design. 1. The client is responsible for starting the data collecting plugins that collects data from the kiosk system. 2. The queue stores the collected data until the WCF service can safely send it to the server. 3. The WCF service client is responsible for communicating the collected data and the heartbeat to the server. It also handles a callback from the server that is relayed back to a updating module

The module uses MEF to collect and start the plugins in parallel worker threads wherein the plugins may collect their data in any way necessary.

4.3 Heart beat

The heart beat of the client is an implementation of the collector contract but is distributed as an intricate part of the system as it uses special rules set up in the communication module to enable the servers callback based on those heartbeats. In the server the heartbeat is handled differently than the other collectors data as it is communicating on another method with the duplex callback enabled. This way of including the heart beat in the inner parts of the clients and servers logic was chosen for two reasons: The heart beat is a central part of the system and should be as far removed from other error sources as possible; The heart beat serves as a great way for the server to set up its callback, enabling the user to push the system updates back to the clients from the user interface.

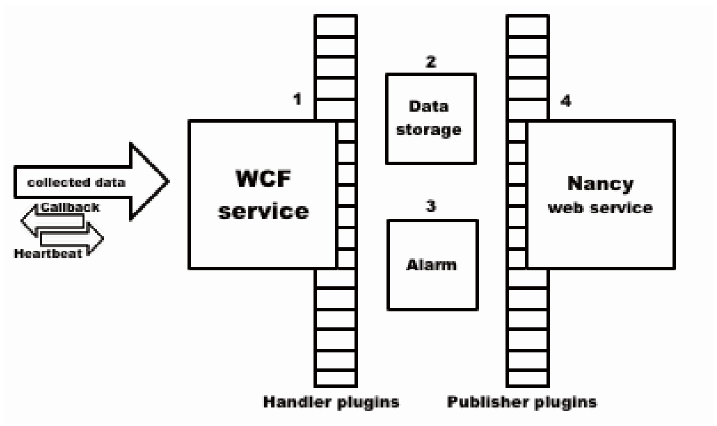

Figure 4.2: Overview of the server design. 1. The WCF service receives the collected data from the client and relays the data to the correct handler plugin. 2. The data storage is accessed mainly by the handling and the publishing plugins and is responsible for making the data persistent. 3. The alarm module is accessed by the handling plugins to notify the systems user of issues discovered in the handling of the collected data. 4. The Nancy web service uses the publishing plugins to relay data to a web browser.

4.4 Server

The server is like the client modularized into several parts. It consists of the WCF service that receives the packages from the clients and is responsible for giving the packages to the correct handler plugin. It also handles the call back to the clients when the user forces an update. The server also has a module responsible for the data storage as well as a module handling the alarm functionality. Both the data storage and alarm modules could be multiplied in the design if need arrises.

4.5 Publishing data

The publishing of data is handled by a server module that collects the publishing plugins and feeds their data into a simple web response published by the publishing module with the help of the Nancy framework.

Chapter 5

Implementation issues

During the research and the implementation of the prototype several issues arose. These issues will in this chapter get a closer examination.

5.1 Duplex WCF with MSMQ bindings

During the design an alternative binding for the WCF service, built on the use of MSMQ, was considered and tested. The issue arose in the callback that was necessary for the forced update of the client. WCF comes out of the box with several bindings also including a MSMQ binding. However, the included support for a MSMQ binding does not allow for the server responding to the client with a callback. Much effort was put into finding a way to solve this issue and after some searching a seemingly potential solution was found on Code Project [18]. The solution examined in this article enabled the server to respond with a callback immediately after a message was received but holding the communication open with several clients, being able to send the callback on the request of the user, was not possible. After much research and tinkering this approach had to be given up in favor of a communication solution with a division between the queue module and the communication module.

5.2 Choosing solutions for maximum extensibility

As the extensibility has been the main focus of the design it has also been a source of issues. The most common issue has been a constant choice between what could be called macro or micro freedom. On one hand the client could be made without a platform dependency but with less dynamic design and on the other hand the client could be designed in a very flexible way but use the features that bind it to a specific platform or framework. In the case of the prototype, developed for this thesis, the act of binding it to .Net, has also made it able to use the features that give it its dynamic build such as MEF and WCF.

When faced with these choices during the development of the prototype the decision of using the .Net dependent features was made based on that Cenito has specifically requested that the prototype be made in .Net and C#. However, for the sake of the research into how extensible and dynamic a system of this kind could be made, it is well worth specifying on which level, micro or macro, the software should be made dynamic.

5.3 Database design for generic data

When storing unspecified data for later retrieval and potentially with enabled searching the choice of a data structure is of importance. Finding the correct way of storing the gathered data from the plugins in the server database was a issue as the data does not follow a set scheme but is structured by the developer of the plugin.

To be able to keep the saved data, a database choice had to be made and there was no lack of options to consider. Due to a lack of prior knowledge and the many potential areas of finding potential solutions the study had to be cut short as to not spend to much time on only this area. A more complete study of NoSQL databases and a potential solution using such could have benefited the prototype.

Finding a model that will allow future unplanned data structures to be entered has been an issue and the chosen solution is a bit lacking in the potential features it could benefit from because of the necessary simpleness that the chosen technic brought on.

Chapter 6

Testing the prototype on a thick client

Testing the prototype on the distributed system supported by Cenito was an early goal for this study and the development and chosen solution was aimed to be run on that system. The prototypes footprint in memory, CPU, and bandwidth usage was minimal and did not disturb the thick client. The prototype was constructed with three plugins for test purposes.

- File watcher

  The file watcher is intended to keep track of configuration files, adding a message to the queue in case of changes to said file.

- Last booking time in the kiosk system

  The surveyed thick client lets users book times on a sun tanning studio and a plugin was constructed to, on an set interval, prompt the database of the thick client for the last booking time.

- Hardware watcher

  A common issue on the surveyed thick client is that the hardware often goes offline. Therefore this plugin will be watching the availability of the hardware on a fixed interval.

The results of the test was good and the plugin functionality worked well. There were some issues with the connection between the prototype server and client because of a firewall as the server was running locally on a separate machine but the issues was resolved swiftly. The test showed that the prototype worked as intended and that it could serve its purpose well. The owner of the distributed kiosk system, used for testing the prototype, was pleasantly surprised about the flexibility of the system and the possibility to add a wide range of surveillance of the hardware. The registration of the client against the server also worked as intended and the forced callback from the server was able to prompt the user to update its software.

For the system to become really useful, in a future support situation, several collecting, handling, and publishing plugins has to be constructed. A better user interface, with the possibility of navigating and sorting the information, is also needed.

The testing of the prototype showed that the general design of the system holds up in a live environment and the prototype will function as a base for further development of the surveillance and maintenance system. The prototype will also continue to survey the distributed kiosk system it was tested on and will there hopefully be of much use in supporting the distributed kiosk system in the future.

Chapter 7

Conclusion

The development of a flexible and extensible surveillance system is, judged by the evaluation of prototype, very possible. The extensibility of the prototype is greatly improved by the use of MEF and therefore the system could be used for surveying data in a great range of systems. Being able to design the plugins to collect data and to specify the use of said data to alarm the user or to just be stored for statistical purposes could be of great help when taking care of distributed thick clients. There is however a need for making the development of the plugins a bit simpler for the user as some experience in programming is needed to develop useful data collectors.

The prototype showed the potential of a product running on the .Net framework and if choosing to broaden the range of platforms there would be a need for more research to find suitable frameworks that is platform independent.

The use of the system does demand a bit of work by the user and if the system were to be made into a product some effort would need to be put into making the usage of the system easier. Building the three necessary plugins for the collected data is not as simple as could be asked in the prototype model and there is a need for some default implementations that the user can inherit and specialize.

Chapter 8

Future development

8.1 A better user interface

To become really useful to a surveying user the product needs an easily operated and overviewed user interface. Delivering the user interface through a web browser works great as it is accessible from any platform but a great deal of work should go into designing a good user experience as the collected data is useless without any good way to view it.

8.2 Filtering of accepted data in the server

If the system were to survey multiple clients, each collecting big amounts of data, it might be in the users interest to filter out clients or some of the plugins data to save on the data usage of the server. For a future product it would be essential to be able to configure the server in such a way.

8.3 Simplifying the plugin development

The development of the collection plugins does demand some know how in programming if the data is not very easily accessible. There would be a great benefit in supplying a simpler way of constructing the plugins for the users without experience in programming. It could for instance be possible to supply a large number of configurable generic plugins that the user could use to target the wanted data for collection.

Chapter 9
Terminology

- Thick client: A client system which relies on locally installed software.
- Thin client: A simple client that stores its application on a central server.
- Design pattern: A general reusable solution to a commonly occurring problem within a given context in software design
- Extensibility: see chapter 2.1
- Blob: A collection of binary data stored as a single entity in a database management system

References

[1] Managed Extensibility Framework Overview, 2012-08-15,
http://msdn.microsoft.com/en-us/library/dd460648(VS.100).aspx#what_is_mef

[2] Nishith Pathak, *Pro WCF: Practical Microsoft SOA Implementation*, Chapter 8 Implementing reliable messaging and queue-based communications, Print ISBN: 978-1-4302-3368-8

[3] Nishith Pathak, *Pro WCF: Practical Microsoft SOA Implementation*, Chapter 1 Implementing reliable messaging and queue-based communications, ISBN: 978-1-4302-3368-8

[4] Optimizing BizTalk Server WCF Adapter Performance , 2012-08-13,
http://msdn.microsoft.com/en-us/library/ee377035(v=bts.10).aspx

[5] Sessions, Instancing and Concurrency in WCF Services, 2012-08-13,
http://msdn.microsoft.com/en-us/library/ff183865.aspx

[6] Extensibility, 2012-08-15,
http://en.wikipedia.org/w/index.php?title=Extensibility

[7] Rob Harrop, Jan Machacek, *Pro Spring*, Introducing Inversion of Control, 2005, ISBN 1590594614, pp. 49 - 92

[8] MEF - Managed Extensibility Framework with Glenn Block, 2012-08-15,
http://www.hanselminutes.com/148/mef-managed-extensibility-framework-with-glenn-block

[9] patterns & practices - Unity, 2012-08-15 ,
http://unity.codeplex.com

[10] Inversion of Control, 2012-08-15,
http://en.wikipedia.org/wiki/Inversion_of_control

[11] InversionOfControl, 2012-08-15,
http://martinfowler.com/bliki/InversionOfControl.html

[12] Mark Seemann, *Dependency injection in .Net*, Part 1 Putting Dependency injection on the map, ISBN: 9781935182504

[13] Mark Seemann, *Dependency injection in .Net*, Chapter 14 Unity, ISBN: 9781935182504

[14] Inversion of Control Containers and the Dependency Injection pattern, 2012-08-16,
`http://www.martinfowler.com/articles/injection.html`

[15] Repository, 2012-08-17,
`http://martinfowler.com/eaaCatalog/repository.html`

[16] The Repository Pattern, 2012-08-17,
`http://msdn.microsoft.com/en-us/library/ff649690.aspx`

[17] Zero Install ,2012-08-17, `http://en.wikipedia.org/wiki/Zero_Install`

[18] WCF: Duplex MSMQ, 2012-12-27,
`http://www.codeproject.com/Articles/41907/WCF-Duplex-MSMQ`

[19] NoSQL, 2012-12-29,
`http://en.wikipedia.org/wiki/NoSQL`

[20] SQL, 2012-12-29,
`http://en.wikipedia.org/wiki/SQL`

[21] ADO.NET Entity Framework, 2012-12-29,
`http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework`

[22] Code-First Development with Entity Framework 4, 2012-12-29,
`http://weblogs.asp.net/scottgu/archive/2010/07/16/code-first-devel`
`opment-with-entity-framework-4.aspx`

[23] *Communications of the ACM*, Volume 53 Issue 4, April 2010, Pages 10-11, ACM New York, NY, USA

[24] ASP.Net Getting Started, 2012-12-31,
`http://www.asp.net/web-api`

[25] See 4.1 Prototype specification

[26] ClickOnes, 2013-01-28,
`http://msdn.microsoft.com/en-us/library/t71a733d(v=vs.80).aspx`

[27] Java Web Start Technology, 2013-01-28,
`http://www.oracle.com/technetwork/java/javase/javawebstart/index.html`

[28] Zero Install, 2013-01-28,
`http://zero-install.sourceforge.net`

[29] Nancy, 2013-01-28,
`https://github.com/NancyFx/Nancy`

[30] Mark Seemann, *Dependency injection in .Net*, ISBN: 9781935182504

[31] XML Database,2013-01-31,
`http://en.wikipedia.org/wiki/XML_database`

[32] Ken Ka-Yin L, Wai-Choi T, Kup-Sze C. Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage. Computer Methods And Programs In Biomedicine [serial online]. n.d.;Available from: ScienceDirect, Ipswich, MA. Accessed January 31, 2013.