Fachhochschule Vorarlberg GmbH
University of Applied Sciences

Diploma Thesis

in the Degree Program

iTec – Information and Communication Engineering

Evaluation of Plug-In Architectures for a Test Suite

implemented by

Jürgen Berchtel

0310109021

cooperation with

OMICRON electronics GmbH

Oberes Ried 1, A-6833 Klaus

Klaus, **September 2007**

Academic Supervisor: **DI Wolfgang Auer**

Danksagung

Ich möchte mich bei meinen Eltern Helga und Bernhard Berchtel herzlich bedanken, da sie mein Studium ermöglicht und mich all die Jahre in allen Belangen unterstützt haben.

Ganz besonders möchte ich mich an dieser Stelle auch bei allen bedanken, die mir bei der Diplomarbeit hilfreich zur Seite standen:

- DI Wolfgang Auer, für die Betreuung dieser Diplomarbeit;
- DI Thomas Hensler, für die Betreuung seitens der Firma OMICRON electronics GmbH;
- Carolin Dobler, Christoph Dobler und Christopher Pritchard für das Korrekturlesen dieser Arbeit;
- Und all denen, die mich bei dieser Diplomarbeit mit hilfreichen Informationen und Verbesserungsvorschlägen unterstützt haben.

Kurzfassung

In dieser Diplomarbeit werden verschiedene Plug-In Architekturen für eine Test Suite evaluiert. Bei der Test Suite handelt es sich um eine Windows-basierte Anwendung, welche bei Prüfverfahren im Bereich der elektrischen Energieanlagen zum Einsatz kommt. Das Plug-In Konzept ist gemäß Mayer u. a. weit verbreitet. Das Konzept wird in vielen Anwendungen verwendet, um Kunden und Fremdanbietern die Möglichkeit zu bieten, die Funktionalität der Anwendung zu erweitern. Spezielle Adaptionen des Plug-In Konzeptes können auch für die Entwicklung von komplexen Anwendungen verwendet werden. In diesem Fall wird der Quellcode in kleine überschaubare Plug-Ins aufgeteilt, was zu einer höheren Modularisierung der Anwendung führt.

Die Evaluierung der Plug-In Architekturen wird stark von den Anforderungen der Test Suite beeinflusst. Diese Anforderungen wurden von der Firma OMICRON electronics GmbH definiert. Die meisten dieser Anforderungen sind sehr allgemein und ähneln somit auch Anforderungen anderer Projekte.

Für die Evaluierung wurden drei geeignete Lösungen untersucht. Diese basieren alle auf dem .NET Framework, welches durch die Anforderungen vorgegeben ist. Evaluiert wurden der Composite UI Application Block (CAB) mit den Smart Client Software Factory (SCSF) Erweiterungen, das Spring .NET Framework und die SharpDevelop Anwendung.

Die Stärken und Schwächen dieser Lösungen werden im Kapitel Evaluierung diskutiert. Wie in der Arbeit festgestellt, ist in Bezug auf die Test Suite der Composite UI Application Block die beste Wahl. Gründe die für CAB sprechen sind zum einen die gute Benutzerschnittstellen-Integration für die Module und zum anderen die Unterstützung für lose gekoppelte Komponenten.

Der Composite UI Application Block erfüllt nicht alle definierten Anforderungen. Deshalb wird mit einem Prototyp gezeigt, wie diese mit der Hilfe von CAB vollständig erfüllt werden können. Der Prototyp besteht aus einer Infrastruktur und verschiedenen Plug-Ins. Die Infrastruktur bietet die von den Plug-Ins benötigten Dienste an, wie zum Beispiel die Benutzerschnittstellen-Integration. Die Plug-Ins zeigen mögliche Lösungswege für die Erfüllung aller Anforderungen. Diese Diplomarbeit zeigt, dass CAB den Aufwand für die Realisierung einer Test Suite verringern kann.

Abstract

This diploma thesis evaluates various plug-in architectures for a Test Suite application. The Test Suite is a Windows-based application which is used for test procedures in the field of electrical power systems. The plug-in concept is widely used according to Mayer et al. [MMS02]. The concept can be found in many applications where it enables customers and third party manufacturers to extend the functionality of the application. A special adoption of the plug-in concept is to use it for building complex applications. In this case the separation of the code into manageable small plug-ins can increase the modularity of the application.

The evaluation of the plug-in architectures is highly influenced by the requirements for a Test Suite application. These requirements are defined by the company OMICRON electronics GmbH. Most of the requirements are also adaptable to other projects, even though they might be of other domains.

Three suitable solutions are chosen for the evaluation. One of the requirements defines that the Test Suite has to be a .NET application. Therefore, all solutions are based on the .NET Framework. Evaluated are the Composite UI Application Block (CAB) with the Smart Client Software Factory (SCSF) extensions, the Spring .NET framework and the SharpDevelop application.

These solutions all have different strengths and weaknesses which are discussed in the evaluation chapter. The thesis shows that the Composite UI Application Block is the most applicable solution for the requirements of the Test Suite and therefore it is further investigated in this thesis. Reasons for using CAB are the good UI integration for the modules and the support for loosely coupled components.

Because the framework does not support all requirements out of the box, a prototype implementation was created to show a possible way to fulfill all defined requirements. The prototype consists of an infrastructure part and several plug-ins. The infrastructure provides common services which are needed by the plug-ins e.g. the UI integration. The plug-ins show how the different requirements can be fulfilled. This diploma thesis shows that CAB can reduce the effort for developing a Test Suite.

Contents

1 Introduction

1.1 Motivation

According to Mayer et al. the *plug-in* concept is widely used [MMS02]. It can be found in many applications to enable customers and third party manufacturers to extend the functionality. These applications are spread in various domains. Some examples are:

- Office (e.g. Microsoft Office, OpenOffice)
- Browser (e.g. Microsoft Internet Explorer, Mozilla Firefox)
- Communication (Miranda IM, IBM Lotus Sametime 7.5)
- Audio (e.g. Steinberg Cubase, Nullsoft Winamp)
- Development (e.g. Eclipse, Microsoft Visual Studio)

Providing a powerful *plug-in architecture* in a software product can help to differentiate from competitors. These days a rich variety of systems are running in the IT environments of the customers. Therefore, the manufacturer is seldom able to support all of them. A plug-in architecture let others create the connection between the application and the various IT systems while the manufacturer is able to concentrate on its core business. For example the customer or third parties can connect the application with an *enterprise resource planning (ERP)* system by writing a new plug-in[1].

An approach of building complex applications and particular *GUI* applications is to use a plug-in architecture not only to offer an extension mechanism[2] for others. It can also be used to separate the own code into plug-ins and increase the modularity this way.

> *The concept of plug-in-based application development (…) goes one step further. Thereby, it is possible to divide the development of big systems into manageable small components which can be evolved independently [MMS02].*

The plug-in architecture supports a fundamental architectural design principle called *Separation of Concerns*. That is to separate cohesive concerns into different independent plug-ins. One advantage of this principle is that developers (or teams) do not have to care about the whole application. They are able to concentrate on their specific requirements and implement them in plug-ins.

> *Additionally, the use of plug-ins reduces the complexity of the design and makes it more understandable [MMS02].*

[1] The term plug-in is also known as add-in, add-on, snap-in or extension [GK07].

[2] An extension mechanism allows adding of new functionality without rewriting or recompiling the main application. See also Extensibility in the Glossary.

It is common for this approach that the plug-ins can also extend each other in a well defined way. This improves the extensibility of the application as new functionality can be introduced by adding new plug-ins. The important point is that the existing code does not have to be changed as long as adding of functionality does not violate the application design. Changing of already reviewed and tested code should be avoided whenever it is possible.

The plug-in architecture allows the deployment of different plug-in sets to create different application editions. This enables product managers to react in a flexible way on market changes. Even upgrading to a more capable version of the product can be done by installing only the necessary plug-ins.

1.2 Objective

The goal of this thesis is to find a plug-in architecture which fits the requirements of a Test Suite. The Test Suite consists of different test modules which must be developed and deployed independently. The test modules still require some kind of communication mechanism for information exchange.

An important point is to analyze existing plug-in architectures on the fulfillment of the specified requirements. The chosen architecture has to be adapted to the requirements defined in this thesis because these architectures are mostly kept abstract to cover as many scenarios as possible. The requirements that are not covered by the architecture have to be addressed with own solutions. A prototype application presents a way for fulfilling the requirements.

1.3 Document Structure

This thesis is divided into eight chapters. The content of these chapters are as follows:

Chapter 2: Requirements

This chapter describes the requirements for a Test Suite product.

Chapter 3: Plug-In Architectures

Gives an introduction into modular application design and plug-in architectures.

Chapter 4: Fundamentals

Describes the Dependency Injection and Service Locator design pattern which are common in application frameworks. Additionally, the chapter discusses the concept of `Attributes` as they are used in some frameworks for configuration.

Chapter 5: Current Solutions

This part describes the investigated plug-in frameworks. These are the Composite UI Application Block, the Smart Client Software Factory, the Spring .NET framework and the SharpDevelop application.

Chapter 6: Evaluation

Shows the evaluation criteria for the chosen solutions. Further, the chapter includes a discussion of each criterion in association with the plug-in frameworks.

Chapter 7: Prototype

Presents the Test Suite prototype which deals with the requirements defined in chapter 2. This chapter discusses the implementation and shows workarounds for occurring problems.

Chapter 8: Final Remark

Contains the conclusion of this diploma thesis and lists some ideas for future work.

1.4 Intended Audience

This diploma thesis is primarily written for software architects and software developers. It is advantageous if the reader has the following skills for understanding this thesis:

- Good knowledge about the principles of object-oriented programming.
- Basic understanding of the programming language C# and the .NET Framework. All the source code examples in this thesis are written in C# 2.0.
- Capability of reading UML class, UML sequence and UML component diagrams. All *UML* diagrams in this diploma thesis use the UML 2.0 notation.

2 Requirements

This chapter describes the requirements for a Test Suite product. The Test Suite is a Windows-based application which is used for test procedures in the field of electrical power systems. The Test Suite is intended to become a successor product for the Test Universe 2.x [Omicron07a]. The requirements discussed in this thesis are far not complete as only the relevant ones are listed here. Relevant means that the requirements are in the field of plug-in architectures. The requirements addressed in this thesis include:

- The runtime platform
- The test modules
- Handle complexity
- Dependencies between the modules
- Module loading
- Dependency resolution and lazy loading
- Deployment and versioning
- GUI integration
- Command service
- Extensions and communication between the modules

Most of these requirements are very general and can be adapted to other projects, even projects in other domains. The requirements are defined by the company OMICRON electronics GmbH. OMICRON is an international company providing solutions for primary and secondary testing in the field of electrical power utilities and industries [Omicron07]. All the listed requirements are of the category must-have if they are not otherwise defined.

The Runtime Platform

The most important requirement is to define the *runtime platform* because it sets the boundaries of the software architecture. The strategy of the company OMICRON is to use the Microsoft .NET Framework 2.0 or a higher version as runtime platform for new client applications which are running on a *PC*. The company's preferred programming language is C# which open source solutions should be written with. The decision for using the Microsoft .NET Framework limits the deployment of the Test Suite to PCs on which the Microsoft Windows operating system is running.

The Test Modules

The Test Suite consists of test modules which enables the user to use different kinds of test procedures on various targets. The modules implement whole *use cases* with their own *GUI elements*, domain logic and module specific infrastructure. The GUI elements need to integrate seamless into the Test Suite. A user should not be aware of the different modules behind the GUI elements that he sees in the application. Many test modules share the same requirements regarding the infrastructure like logging, error handling, security, etc. To avoid implementing the same infrastructure code in every test module separately, they require access to a shared infrastructure implementation.

Handle Complexity

The complete Test Suite application is going to be a huge software product. The predecessor is known as OMICRON Test Universe 2.x and has about 2 million lines of code. Most parts are written in C++ whereas some newer parts are already developed with C#. A single project for rewriting the Test Universe from scratch with all its features is not manageable. Thus, the project needs to be divided into smaller feasible ones. The dependencies of these projects have to be as low as possible to remain controllable. Therefore the test modules need to be coded, tested and deployed independently. Nevertheless, the modules have to cooperate together. This cooperation includes extension and interaction with other modules.

Dependencies between the modules

Modules can use and extend functionality of other modules. Therefore, they have dependencies among each other. The functionality is mostly represented by *services*. Likewise the services have a dependency between the provider and the consumer. The dependencies on service level can be divided into hard and soft ones. Whereas the services defined by hard dependencies have to be available in order to work, the services defined by soft dependencies do not have to. A module is still able to work if some soft dependent services are not loaded but they run with reduced functionality.

Module Loading

The modules have to be loaded by the application at runtime. They are not allowed to have static references to other modules, because at compile-time it is not known which modules are going to work together. The module loader has to support configuration by a human-readable file and alternatively by command line parameters. This provides a flexible way of exchanging the modules as only the configuration file needs to be modified. It is especially useful in unit testing of a specific module. During testing, all the dependent modules are replaced by some mock modules. So the specific module is tested in an isolated environment.

Dependency Resolution and Lazy Loading

The module loader needs the information of the module dependencies. The dependent modules must be loaded before the defined module. Important to note is that dependent modules can also be dependent on other modules. These hierarchical dependencies can be represented in a *tree data structure*. Furthermore, the modules should be loaded on demand. This is also known as lazy loading. It means that modules are only loaded if they are used by the application or any other module. This approach improves the application start-up time and saves resources like memory.

Deployment and Versioning

Another point of the requirements is a strategy for the deployment of the Test Suite with its modules. This includes installing, uninstalling and updating the whole application or only a single module. Developers should be able to deploy new module versions without affecting the application or other modules. From this it follows that it can be necessary to deploy different versions of the same module on the identical machine. Additionally, the different versions need to be loaded side-by-side in the same client process. Not fulfilling this requirement results in a problem known as DLL-Hell [Löwy05, p. 11]. A nice-to-have feature would be to do the deployment tasks like installing, uninstalling or updating of the modules without restarting the application.

GUI Integration

A module needs to integrate seamlessly into the Test Suite. Consequently it requires an interface for the extension of the application GUI. The elements to extend are the menu bar, toolbar, status bar, option dialog, open / save dialog, etc. These extensions should be independent of the underlying user interface technology. At the moment it is planned to use the reliable *Windows Forms* framework but in future it will be the new *Windows Presentation Foundation (WPF)* which is introduced in Microsoft .NET Framework 3.0. Writing GUI components is a time consuming work. Therefore, it is not possible to change the GUI framework used by the whole application in a single step. The Test Suite application has to support both GUI technologies and even allow the mixing of modules which are based on the different GUI frameworks.

Command Service

Additionally to the GUI extensions, some kind of *command* service is required. The command service has to group different GUI elements (e.g. Menu item, toolbar button, etc.) together. It needs to be possible to hide or disable all GUI elements which are connected to the same command. This should be controlled by a command state. The state is mostly dependent on the active status of the module but in some cases a custom handling is required. Furthermore, the command object has to be associated with one or more command handlers.

Extensions and Communication between the Modules

The GUI extensions are already discussed in this chapter. Event though, other types of extension are also necessary. This can be an extension of the domain functionality of a module. A module has been able to provide various extensions and also consume extensions of other modules. Beside extension, a loosely coupled mechanism for communication between the modules should be available.

3 Plug-In Architectures

The previous chapter shows the requirements of the Test Suite product. Some of them can be fulfilled by a modular application design. Fowler writes that modularity is about hiding a secret in its implementation that is not apparent from the interface [Fowler04]. An example can be seen in Figure 1 whereas the modules hide their implementation behind an interface.

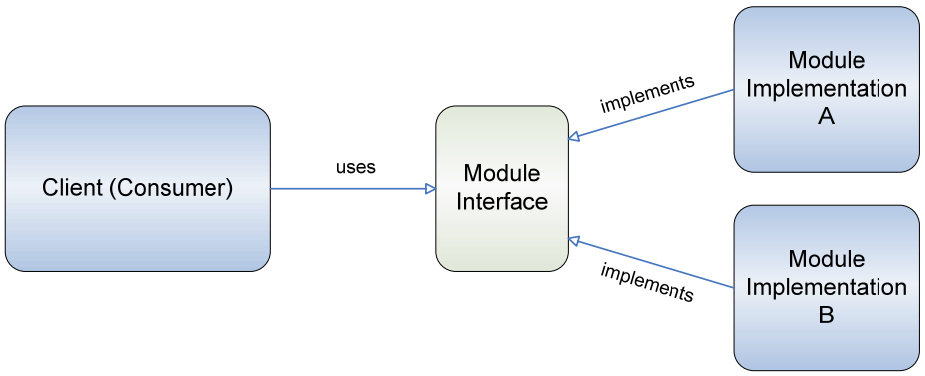

Figure 1: The client uses one of the modules through an interface.

A modular application design has two essential advantages. The first one is that the developer who writes the client does not have to understand the implementation of the module to use it. The second advantage is that the implementation can be exchanged without rewriting and recompiling the client code. To prevent recompiling, a *binary compatibility* is necessary between the client and the module. The .NET Framework provides this by compiling the source code (e.g. C#) into *Intermediate Language (IL)* code. The IL code contains tokens to identify the fields and methods instead of using offset memory addresses as it is common in machine code. This is possible because the IL contains the meta-data of all types. Before the IL code can be run, it must be converted by the *Just-In-Time (JIT)* compiler to native machine code. The JIT compiling is an automatic process which is started at runtime by the .NET Framework.

Separation of interface and implementation, and binary compatibility are a subset of the principles of *component-oriented programming* [Löwy05, p. 6].

> *Over the last decade, component-oriented programming has established itself as the predominant software development methodology. (…) Practitioners have discovered that by breaking down a system into binary components, they can attain much greater reusability, extensibility, and maintainability [Löwy05, p. 1].*

Popular component-enabling technologies are *DCOM*, *CORBA*, *JavaBeans*, and the .NET Framework which is a relatively new member in this field. As Löwy writes the other principles of component-oriented programming are language independence, location transparency, concurrency management, version control, and component-based security. These principles are all supported by the .NET Framework whereas a programmer does not have to adopt all of them.

A component-enabling technology like the .NET Framework already fulfills some of the Test Suite requirements. But one important aspect is not handled by component-oriented programming. Someone has to wire the client with the appropriate module implementation together. In the example seen in Figure 1 the client could use the implementation of module A or B. One of the Test Suite requirements needs the wiring dependent on the runtime configuration. For example the client uses the implementation of module A by default and during testing it uses the mockup implementation of module B. How this can be accomplished in a flexible way is described as *Plugin pattern* [Fowler03, p. 499].

> *Use Plugin whenever you have behaviors that require different implementations based on runtime environment. [Fowler03, p. 500]*

The Plugin pattern adapts the Factory pattern [Larman04, p. 440] which reads the linking (wiring) instructions from a single, external point in order to keep the configuration management easy. The linking has to be done at runtime rather than during compile time. Otherwise, a rebuild would be necessary if the configuration changes. This can be accomplished via *reflection* which is supported by the .NET Framework.

A software design that uses the Plugin pattern is called plug-in architecture in this thesis. These architectures can be divided into two categories which are based on the same concept but fulfill different requirements. The former one is used to increase the modularity of the internal application design. This one is addressed in this diploma thesis. The latter plug-in architecture is used to provide an *automation* and extension interface for third parties. It shares the requirements of the former one and comes up with new ones. The most important additional requirements are backward compatibility and isolation. An application (*host*) evolves considerably faster as the plug-ins of third parties. Thus, the interfaces for external plug-ins have to be more robust as the internal application design. Otherwise it is likely that plug-ins stop working because the application was updated. Some plug-in frameworks help to ensure backward compatibility for older plug-ins. The second requirement is the isolation of a plug-in from the host and other plug-ins. This allows the host to be unaffected of unstable plug-ins. In addition, it is often combined with sandboxing of the plug-ins to increase the security. The realization of these advanced requirements comes at high costs. A version resilient architecture is far more complex and the communication between isolated parts comes with high performance penalties. Although the Test Suite does also require this kind of plug-in architecture, it is not in the scope of this diploma thesis.

Modularization is an important principle in software engineering. It improves the reusability which can lead to faster time to market, and lower development and long-term maintenance costs. Modular software design is promoted by evolving development methodologies like procedural programming, object-orientation, component-orientation and aspect-orientation.

> *(…), separating the interface from the implementation and separating configuration from use are two vital principles in a good modularization scheme. [Fowler04, p. 67]*

4 Fundamentals

4.1 Overview

Plug-in architectures do not have to be designed and implemented from scratch since already some sophisticated *frameworks* exist. The architecture of a plug-in framework shall work with all applications of the intended domain. The Test Suite product is in the domain of Windows-based applications. It is very difficult to design a framework architecture that is applicable for a variety of applications. This means that flexibility and extensibility are essential for a framework design [GHJV95, p. 27]. Additionally, low coupling between the framework and the application is important. Modifications on the framework should not bring much migration work for the application. Gamma et al. write the following about these issues:

> *A framework that addresses them using design patterns is far more likely to achieve high levels of design and code reuse than one that doesn't. Mature frameworks usually incorporate several design patterns. The patterns help make the framework's architecture suitable to many different applications without redesign [GHJV95, p. 27].*

Basic knowledge of the most important design patterns and concepts in the field of plug-in architectures help to evaluate different plug-in frameworks. Particularly, it is easier to estimate how a framework will affect the whole application design. The previous chapter gives a short introduction into the Plugin pattern. Whereas this chapter addresses some further important design patterns and concepts used in the field of plug-in architectures.

4.2 Dependency Injection

The *Dependency Injection* (DI) pattern arose from the Java community when they tried to find alternatives to the high complex enterprise Java world. This pattern helps to wire components of different layers together. The components are often developed by different teams with minor knowledge of each other. A well-known task for an architect is to compose the components into a coherent overall application. A number of design patterns, such as Factory Method, Abstract Factory, Builder, etc. [GHJV95], are already devoted to deal with this issue. An alternative for implementing these design patterns is to use a reliable framework. Some frameworks, which deal with the wiring of components, are known as *Inversion of Control (IoC) container*. They are also referred as lightweight containers because of the minor performance impact and the lower application complexity compared to other container technologies (e.g. Microsoft .NET Framework Enterprise Services) [Caprio05].

Inversion of Control is a general principle which is often used to characterize frameworks [Fowler05]. It is also known as Hollywood principle "Don't call us, we'll call you". It means that the framework takes control over the program and calls the code of the client. For example, a GUI framework calls a method of the client if a button is pressed. Fowler writes that this term is too general and does not suite as a description for the pattern used by IoC containers [Fowler04a]. Thus the name Dependency Injection is used for this particular pattern.

In Dependency Injection a client object (`Birthday printer`) declares its dependencies (`Address book`). Dependencies are objects (`Address book Implementation`) which are required by the client to fulfill its tasks. The client is not responsible to get the dependent objects. This is done by an external mechanism which is known as `Assembler`. The specific characteristic of this pattern is that the client does not have any dependencies to the `Assembler` or any other object for locating the dependent objects. The resulting dependencies between the classes can be seen in Figure 2.

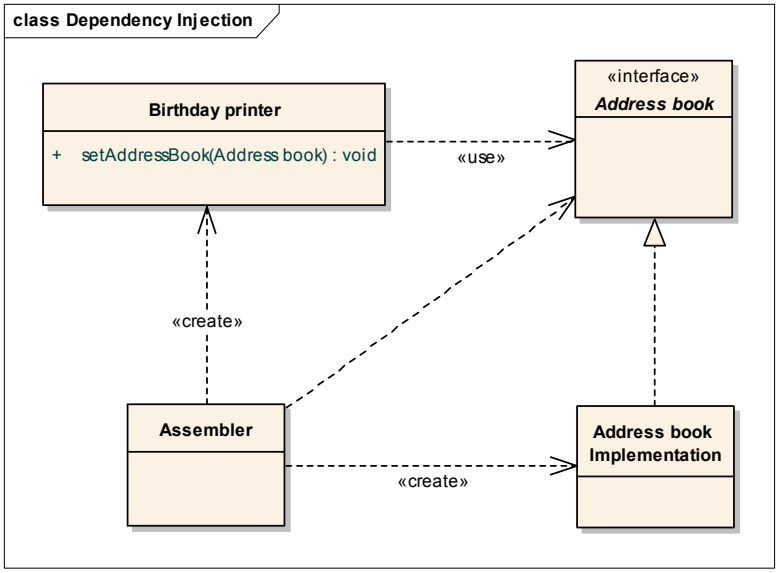

Figure 2: UML class diagram for dependency injection [Fowler04].

The tasks of the `Assembler` are:

- Read the dependency information of the client object (`Birthday printer`).
- Create or locate the dependent objects (`Address book Implementation`)
- Create the client object (`Birthday printer`)
- Inject the dependent objects into the client object (`setAddressBook`).

This process is shown in Figure 3, except of reading the dependency information.

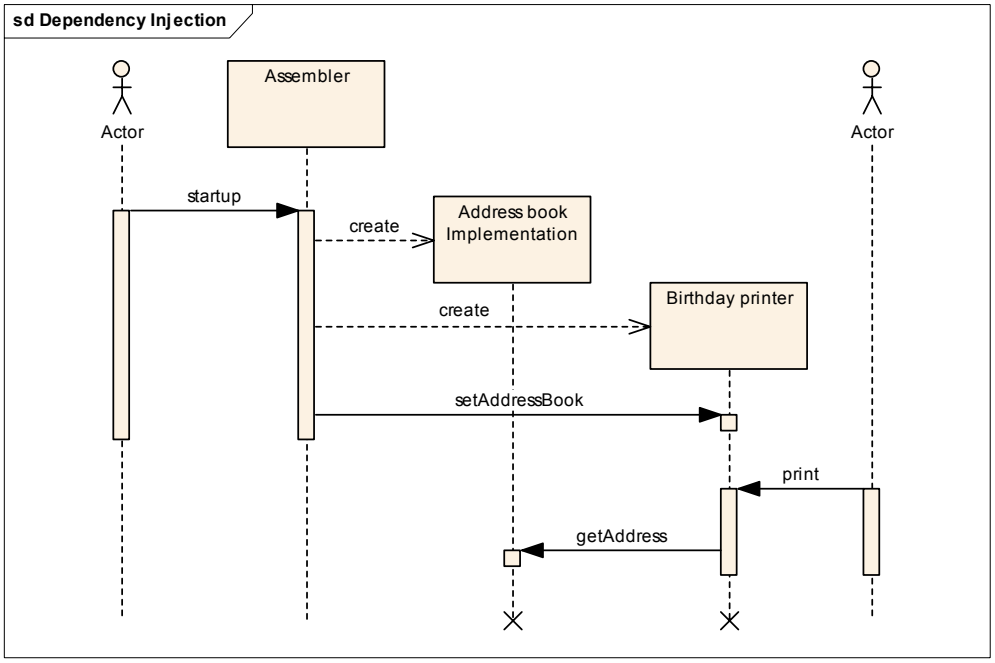

Figure 3: UML sequence diagram for dependency injection [Fowler04].

The Dependency Injection pattern does not define in which way the dependencies have to be declared. A popular approach is to write the dependencies in an external file, particular in an *XML* file. Another possible solution seen in DI frameworks is the using of associated meta-data direct in the programming language like *Attributes* in .NET or *Annotations* in Java.

How the `Assembler` locates the dependent object is not specified. The Plugin pattern can be used for this task. A common way for the configuration of the Plugin factory is the use of an XML file. XML files can be easily changed for different deployment scenarios. Nevertheless, other approaches can be useful too like retrieving the configuration dynamically from a server.

The injection can be done in various ways. Fowler writes that there are three main styles of dependency injection [Fowler04a]:

- Type 1 IoC: Interface Injection
- Type 2 IoC: Setter Injection
- Type 3 IoC: Constructor Injection

An example for Setter Injection can be seen in Figure 3. The `Assembler` calls the setter method `setAddressBook` of the object `Birthday printer` to inject an implementation of the `Address book` interface.

An alternative to the previous approach is that the `Birthday printer` class does not provide the setter method. Instead, it requires the `Address book` implementation already in the constructor (Listing 1). The `Assembler` passes the implementation of the `Address book` interface to the `Birthday printer` constructor. This procedure is called Constructor Injection.

```
1  public class BirthdayPrinter
2  {
3      private AddressBook _book;
4
5      public BirthdayPrinter(AddressBook book)
6      {
7          _book = book;
8      }
9
10     ...
```

Listing 1: Extract of the client class which is configured by constructor injection.

Interface Injection is not relevant for this diploma thesis because the investigated solutions do not support this type of injection. Most of the lightweight containers do not promote this approach. According to Fowler, the reason is the more invasive nature of Interface Injection since many interfaces are required to get it working [Fowler04a].

4.3 Service Locator

An alternative to Dependency Injection is the Service Locator pattern [Sun02]. Basically, it uses a central object (`Service locator`) that knows how to locate the dependent objects (`Address book Implementation`). The dependent objects are referred as services in this context. The client (`Birthday printer`) requests the concrete implementation of the `Address book` interface from the `Service locator`. In contrast to the Dependency Injection pattern the client takes an active role in retrieving the concrete implementation. Thus, it has a dependency to the `Service locator` (Figure 4).

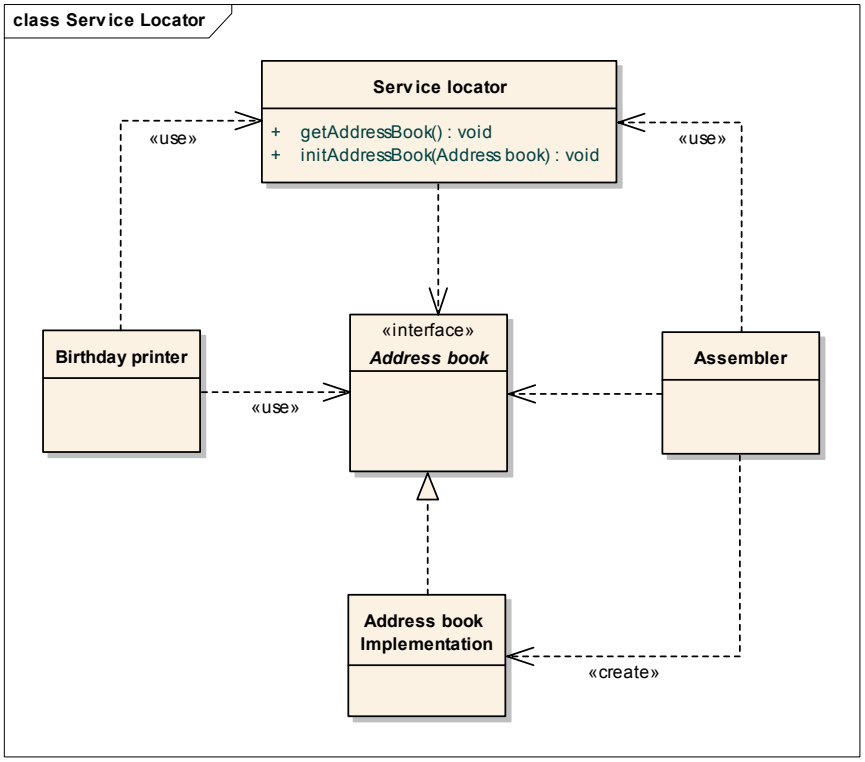

Figure 4: UML class diagram for a service locator [Fowler04].

This time the tasks of the `Assembler` are (Figure 5):

- Create the services (`Address book Implementation`).
- Pass them to the Service locator (`initAddressBook`).

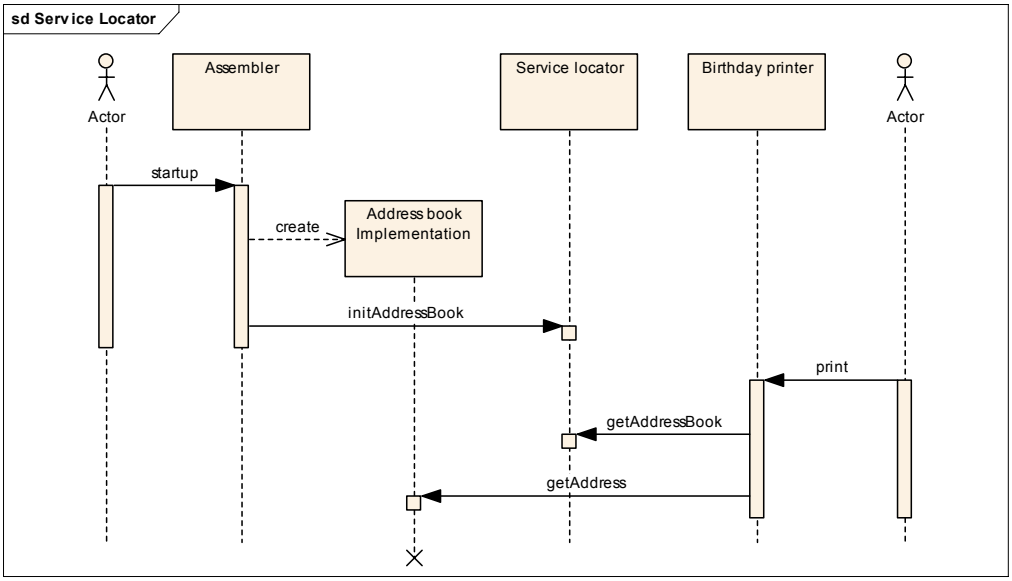

Figure 5: UML sequence diagram for a service locator [Fowler04].

How the `Assembler` is going to find the right implementation is neither specified by this pattern nor specified by the Dependency Injection pattern. In this case the Plug-In pattern is a possible solution too.

The `Service locator` class can be realized as a Singleton [GHJV95, p. 127]. If the `Service locator` should provide an implementation depending on the application context, the Registry pattern [Fowler03, p. 480] is a good alternative. For example, the Registry is able to provide a separate database connection service for every thread which simplifies the development of multi-threaded applications.

The class diagram (Figure 4) shows the `Service locator` class with the service specific methods `initAddressBook` and `getAddressBook`. These methods can be written in a more general way, so that different services can be registered and retrieved. The example code (Listing 2) shows how generics can be used to write a general `ServiceLocator` class.

```
1  public class ServiceLocator
2  {
3    public static T get<T>() { ... }   // Instead of getAddressBook
4
5    public static void register<T>(T service) { ... }  // Instead of
6                                                       // initAddressBook
7    public static void deregister<T>() { ... }
8  }
```

Listing 2: A generic service locator implementation.

In Listing 2 the type `T` is used to identify the service. Alternatively, a string or integer value could be used as an identifier. Using the types has the advantage that the refactoring and error checking capabilities of the IDE still works. The disadvantage is that only one service of the same type can be registered. Thus, this approach is not as flexible as using string or integer values as an identifier [Nilsson06, p. 373].

4.4 Attributes vs. Configuration Files

The .NET Framework provides `Attributes` for adding meta-data to an assembly, a type, a type member or other targets. The `Attributes` can rather be used to declare information in the code than creating external configuration files. This is also known as declarative programming. The meta-data can be read by an application through the reflection API of the .NET Framework.

Declarative programming is an interesting alternative for configuring frameworks to the classic configuration files. The main advantage is that the `Attributes` are associated directly with a target. This can save a lot amount of configuration as it is shown in Listing 3 and Listing 4.

```
1  [ServiceDependency]
2  public IMovieFinder MovieFinder
3  {
4      set { _movieFinder = value; }
5  }
```

Listing 3: Configuration of setter injection with an `Attribute`.

```
1  <objects>
2    <object id="MyMovieLister" type="MovieLister.MovieLister,
3        MovieLister">
4      <property name="MovieFinder" ref="MyMovieFinder" />
5    </object>
6    ...
7  </objects>
```

Listing 4: Configuration of setter injection with an external XML file.

These both code examples list a setter injection configuration for the same component. Listing 3 uses a `ServiceDependency` attribute for the configuration. The configuration is minimal as it consists only of the `Attribute` type name. The `Attribute` is directly above the `MovieFinder` property and thus, the meta-data is attached to this property. Listing 4 configures a setter injection for another Dependency Injection implementation in an external XML file. Here the configuration consists of the lines 2, 3 and 4. In this case, more information is necessary to configure the injection. Most of this information is necessary to address the `MovieFinder` property. If the configuration has to refer to the code, the approach with `Attributes` needs less amount of information. Furthermore, the maintenance is simplified because the code and configuration is at the same place. This makes in many cases of component refactoring, modifications to the configuration unnecessary. For example, the renaming of the `MovieFinder` property does not require a change to the information declared by the `Attribute`.

`Attributes` also have a few drawbacks. The main weakness is that they do not physically separate the configuration from the code. If the `Attributes` are overused, the source code can become messy [Sosnoski05]. Additionally the code requires a reference to the assembly that provides the `Attributes`. This reference can be a problem if the code should be independent of the framework or the library (Framework dependencies, p. 38). The use of a configuration file does not have these drawbacks.

Sosnoski [Sosnoski05] writes in more detail about the differences of using meta-data inlined with the code and configuration files. He uses the term `Annotations` instead of `Attributes` as it is the Java keyword for the same concept. Declarative programming and external configuration files are widespread for framework configuration. Understanding the impact of these concepts on the application design helps to evaluate the frameworks.

4.5 Summary

The Dependency Injection and the Service Locator patterns are two possible ways to wire different components together. With the Service Locator pattern the client retrieves its dependent objects by requesting a central object. In this case, the client has an active role to get hold of the needed objects. In contrast, the client in the Dependency Injection pattern has a passive role. An external mechanism is responsible that the client gets the dependent objects. This mechanism is known as injection.

These patterns are often seen in plug-in architectures. Understanding them can help to evaluate the different architectures and frameworks. Dependency Injection has a minor impact for the application design whereas Service Locator is easier to understand and to debug. In the Service Locator pattern it is also possible to provide different objects depending on the application context. Which pattern should be preferred is dependent on the requirements.

A Dependency Injection implementation requires some kind of configuration. The most popular ways for configuring are the use of `Attributes` and the use of external configuration files. Both concepts have different advantages and drawbacks. Which of them should be preferred depends on the requirements defined for the application.

5 Current Solutions

5.1 Overview

This chapter introduces the three chosen solutions which are analyzed and evaluated in chapter 6. These solutions are based on the .NET Framework which is a requirement defined in this thesis. They are well known in the .NET community, yet more sophisticated solutions can be found. Some of them are mentioned in chapter 8.3. The three chosen solutions share the same idea at the core level which is to increase the modularization with a plug-in architecture.

5.2 Composite UI Application Block

The Composite UI Application Block[3] (CAB) is a plug-in framework from Microsoft. It helps to write complex Windows-based applications which are built of independent components. The components can be composed together in a flexible way to form an overall coherent application. The focal point of this Application Block lies on *user interface integration*. The components are able to contain own UI elements which can be hosted in a *Shell*. The Shell is responsible to show the hosted UI elements and it defines a general layout to control their appearance. The components are able to extend some special UI elements of the Shell like the menu bar, tool bar, status bar, etc. The Composite UI Application Block is shipped as C# and VB.NET 2005 version. It requires the Microsoft .NET Framework 2.0.

Architecture

CAB heavily uses and implements design patterns which are common in the development of Windows-based applications. An overview of the CAB architecture is shown in Figure 6. The design patterns used by the blocks are presented by the elliptical shapes.

[3] Official Website of the Composite UI Application Block: http://msdn2.microsoft.com/en-us/library/aa480450.aspx.

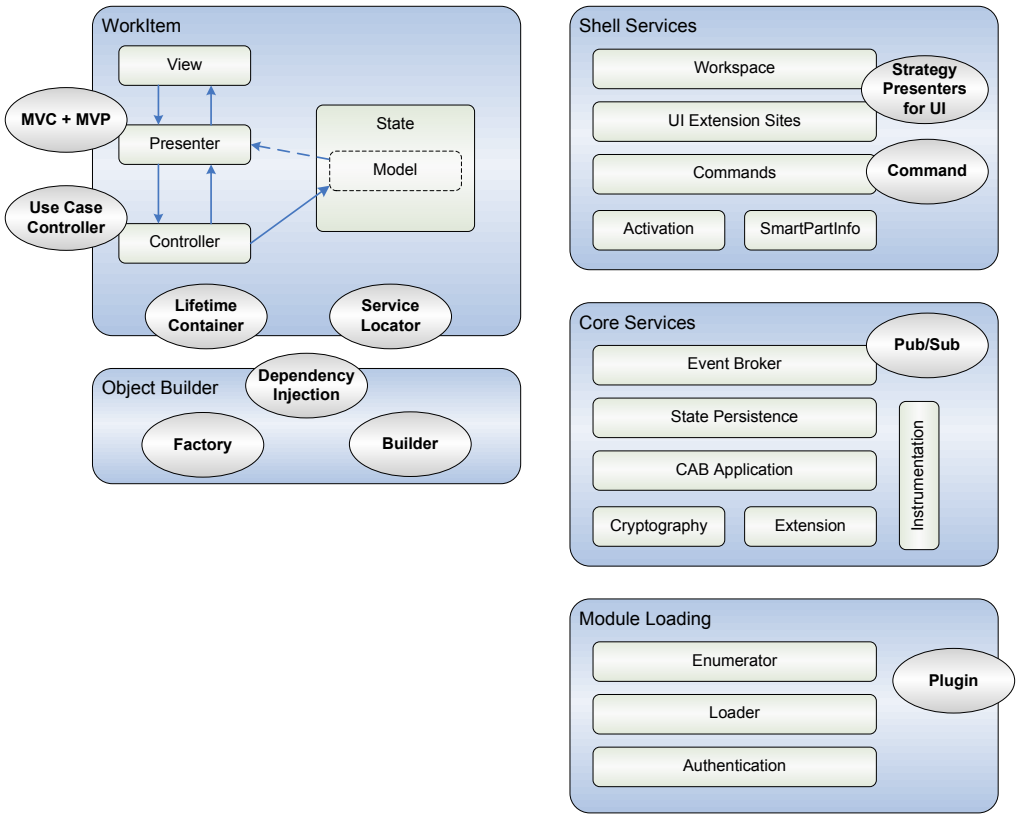

Figure 6: Patterns implemented or supported by the Composite UI Application Block [MSDN06].

WorkItem

A `WorkItem` is a lifetime container for visual and non-visual components. Typically, it contains all components which are necessary to handle a specific use case. The `WorkItem` is responsible for creating and disposing these components. Additionally, it provides an implementation of the Service Locator pattern to retrieve and register services. This is an alternative to the Dependency Injection mechanism provided by CAB.

The components managed by a `WorkItem` can access each other inside the container. If the container is not able to find a component, it delegates the request to its parent container. This behavior is known as Chain of Responsibility pattern [GHJV95, p. 223]. The pattern allows the accessing and using of components which are registered in parent containers. `WorkItems` are composed in a tree structure. A CAB application always provides a single `RootWorkItem` which contains infrastructure services used by the child `WorkItems`. Figure 7 illustrates an example for a `WorkItem` composition. The `RootWorkItem` provides two services which can be used by all child `WorkItems`. It creates a child `WorkItem` for handling a use case. This `WorkItem` creates two additional child `WorkItems` which are responsible for a second use case. These additional `WorkItems` are able to access the state of the first use case.

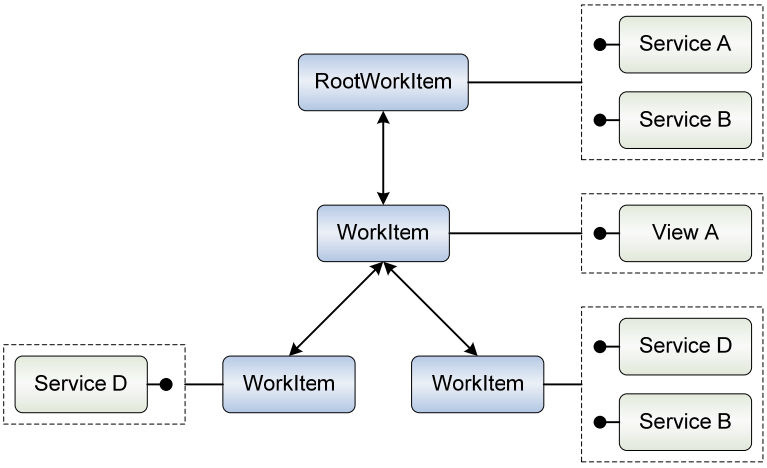

Figure 7: WorkItem hierarchy [MSDN06].

Object Builder

The Object Builder is the Dependency Injection Framework behind the Composite UI Application Block. It is responsible for wiring the independent components together. The dependencies of the components are defined by `Attributes` (e.g. `CreateNew` or `ServiceDependency`). The Object Builder reads these `Attributes` and injects the required components during the creation cycle. It supports constructor and setter injection which can even be mixed. The Object Builder can also be used programmatically without defining `Attributes`. The methods are provided by the `WorkItem`. e.g. `WorkItem.SmartParts.AddNew<OfficerView>();`

Nevertheless, the Object Builder is a general purpose framework. CAB provides the strategies for the Object Builder so that it knows how to interpret and react on these `Attributes`. The Object Builder is also used by other Microsoft products like the Enterprise Library, the Web Client Software Factory, and the Mobile Client Software Factory. The source code is included in the Composite UI Application Block. A newer compatible version can be downloaded to run in partial trust environments and it is signed by the Microsoft patterns & practices team[4].

Shell Services

The Shell Services block in Figure 6 contains some UI integration specific services. The Workspaces host the UI elements and define their appearance. CAB already includes some prefabricated Workspaces like the `TabWorkspace` which shows the UI elements inside tabbed pages. The UI Extensions are services to extend exposed UI elements. For example it is possible to add menu items into the Shell's menu bar via this service.

[4] Official Website of the Object Builder: http://www.codeplex.com/ObjectBuilder

It is common that more than one UI element invokes the same method. An open file button can be hosted in a menu bar and a tool bar. Each of these buttons are represented by an own UI element with different appearance settings. However, both should call the same method if the user presses one of these buttons. This issue is dealt by an implementation of the Command pattern [GHJV95, p. 233]. Furthermore, the Command implementation allows the association of one UI event with multiple methods (*command handlers*).

Core Services

The Core Services block encloses the low level services which are provided by CAB. The event broker is a loosely coupled, multicast event mechanism for components managed by the `WorkItems`. The events can be published and subscribed programmatically or via attributes. The State Persistence service can be used for saving the current application state and for reloading it at the next application start-up. If the application state contains sensitive data, the Cryptography service helps to protect them. The CAB Application is an abstract class that defines the application lifecycle and contains an instance of the root `WorkItem`. Extension and Instrumentation can be used to monitor the lifecycle of the `WorkItems`.

Module Loading

In CAB the modules are a synonym to plug-ins. The Module Loading block in Figure 6 shows the Enumerator service which knows how to retrieve a list of the modules to load. By default, a XML catalog file holds this list. The Loader service is responsible for the module loading. It can use the Authentication service which only loads the modules, the user has permission for. When a module has dependencies to other modules it expresses the dependencies with the `ModuleDependency` attribute. Behind the surface of these services the Plug-In pattern can be found.

License

The software can be used for every commercial and non-commercial purpose without any fee. It is allowed to distribute modified versions and to combine it with own products or services. Although, the license allows the modification of the source code, it can complicate the migration to a newer version of the Composite UI Application Block. The product does not come with any warranty or guarantee from Microsoft. A specialty is that this software is only allowed to run on the Windows platform. Thus, it is not permitted to run an application built on the CAB on the Linux operating system with Mono as .NET runtime platform.

5.3 Smart Client Software Factory

The Smart Client Software Factory [5] (SCSF) assists during the creation of a composite Windows-based application which is build on the top of the Composite UI Application Block. In this thesis the May 2007 release is used. SCSF can be seen as an extension to CAB which uses additional software assets:

- Composite UI Application Block Extensions for WPF
- Enterprise Library 3.1
- Guidance Automation Extensions (for MS Visual Studio 2005)
- Guidance Automation Toolkit (for MS Visual Studio 2005)
- Application Blocks for supporting occasionally connected clients

Software Factory

SCSF uses the concept of a Software Factory. A Software Factory is a collection of software assets, software tools and documentation. It helps to build applications that share an architecture and a feature set. In the case of SCSF it supports all composite Windows-based applications. The software assets can be reusable code components and reference implementations. The software tools can be wizards, code generators and visual designers. It is common to integrate these tools into the IDE. For example the Smart Client Software Factory provides a wizard to create a view with an associated presenter class. Typical parts of the documentation are an architecture guidance, description of common patterns, how-to topics, and an explanation of the reference application. A key concept of Software Factories is that architects can customize them to their own needs. The use of Software Factories helps to increase the consistency and quality of an application and it also boosts the productivity by reusing software assets [SCSF06].

Composite UI Application Block Extensions for WPF

This application block extends CAB with a Windows Presentation Foundation (WPF) integration layer. The layer allows the Shell to host WPF user controls in the same way as it hosts Windows Forms controls. To activate the WPF integration layer the `WPFFormShellApplication` can be used to initialize the Composite UI framework with the needed services. The `WPFUIElementAdapter` service is responsible for wrapping all WPF controls with `ElementHost` objects. The `ElementHost` class is part of the .NET Framework 3.0 and can be used in Windows Forms-based applications to host WPF controls. The WPF support, which is provided by this application block, does not include the Shell and the UI infrastructure elements. They still need to be implemented via Windows Forms controls.

[5] The official Website of the Smart Client Software Factory: http://msdn.microsoft.com/smartclientfactory.

License

The Smart Client Software Factory uses the same license as the Composite UI Application Block.

5.4 Spring .NET

Spring .NET[6] is an *application framework* which provides lots of functionalities to simplify the building of enterprise applications. The functionalities are divided into independent modules. An exception is the Core module which represents the fundament of this framework. Most of the other modules require the Core to work properly. The modular architecture allows to chose just the necessary modules for own applications. Figure 8 shows an overview of the various modules shipped with Spring .NET 1.1 RC1.

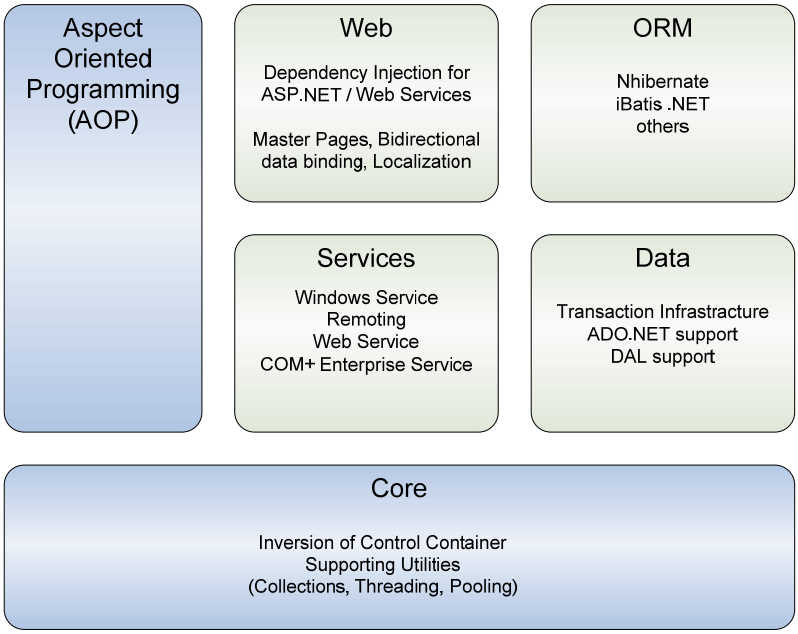

Figure 8: Overview of the modules in Spring .NET [Spring07].

Spring .NET supports the development of web applications and server components but it lacks of assistance for Windows-based applications. Therefore, only the Core block of the Spring .NET modules is taken into consideration for this thesis. Although, other blocks (e.g. ORM) can be useful too for the development of a Test Suite but they are not in the scope of the requirements defined in chapter 2.

[6] The official Website of Spring .NET: http://www.springframework.net.

IoC Container

Spring .NET contains a flexible IoC Container for the wiring of collaborating components. It supports constructor injection, setter injection and it is possible to call a static factory method instead of a constructor. The preferred way of configuring the managed objects and their dependencies is by an XML file. However, the framework is flexible enough to be extended by other configuration mechanisms.

Configuration

Typically, the components are described in an XML file (Listing 5). The XML attribute `ref` can be used to declare the dependent components (Line 4). The identification of the components is done via string values. Besides injecting dependent components, Spring .NET is as well able to inject intrinsic values (Line 9) and arrays. Moreover, it can modify collections which are exposed by properties.

```
 1  <objects>
 2    <object id="MyMovieLister" type="MovieLister.MovieLister,
 3        MovieLister">
 4      <property name="MovieFinder" ref="MyMovieFinder" />
 5    </object>
 6
 7    <object id="MyMovieFinder" type="MovieFinder.MovieFinder,
 8        TextFileMovieFinder">
 9      <constructor-arg index="0" value="Movies.txt"/>
10    </object>
11  </objects>
```

Listing 5: Extract of a Spring .NET configuration file.

Listing 5 shows a small example for a Spring .NET configuration. It uses the `MovieLister` example from Fowler's article about Dependency Injection (Fowler04a). The configuration defines a `MovieLister` component that requires a `MovieFinder` component in order to work (Line 2 - 5). The `MovieFinder` needs to be injected into the property `MovieFinder` of the `MovieLister class` (Line 4).

Nilsson writes that the information in the Spring .NET configuration file is redundant [Nilsson06, p. 380]. Most of this information is also available via the .NET type system. The redundant information leads to a potential maintenance problem. The configuration file has to be kept synchronous with the code. Changes in the code might need modifications in the configuration file. This maintenance problem can be reduced with a function called autowiring.

Autowiring

Spring .NET is capable to resolve the dependencies of a component automatically. This is done by inspecting the component definition via reflection. Listing 6 shows the same example as Listing 5 but uses the autowiring function. By default, autowiring is deactivated and has to be enabled for every object with the `autowire` attribute (Line 3). Spring .NET knows different modes for autowiring [Spring07, p. 38]. The mode `byType` iterates through all *reference-type* properties of the `MovieLister` component and it searches adequate objects for them.

```
1  <objects>
2    <object id="MyMovieLister" type="MovieLister.MovieLister,
3        MovieLister" autowire="byType" dependency-check="objects">
4    </object>
5
6    <object id="MyMovieFinder" type="MovieFinder.MovieFinder,
7        TextFileMovieFinder">
8      <constructor-arg index="0" value="Movies.txt"/>
9    </object>
10 </objects>
```

Listing 6: Extract of a Spring .NET configuration file with activated autowiring.

If autowiring does not find an adequate object for a property, it simply ignores it. The mode "checking for dependencies" allows guaranteeing that all properties are initialized with dependent objects (Line 3). If the dependency check finds an unassociated property, the framework throws an `UnsatisfiedDependencyException`. The `depencency-check` attribute knows further modes but they are not important for the autowiring [Spring07, p. 39].

Application Context

The application context represents the container that manages the lifecycle of the components. These containers can be structured into hierarchies. This is comparable to the WorkItem hierarchy of the Composite UI Application Block. The context provides a Service Locator style access to the registered components. This access can be used as an alternative to the Dependency Injection. Using the Service Locator is not recommended as the component would have a reference to the underlying framework. However, sometimes a component cannot be created by the framework factory and thus it is necessary to retrieve the dependent components manually.

Loosely Coupled Events

One of the services provided by the application context is the loosely coupled event propagation. A component can register a publisher of which all .NET events are routed to interested subscribers. Other components can register a subscriber which receives all the events published by a specific type. The subscribers need to have a method that matches the signature of the event *delegate*. The event wiring cannot be done via the XML configuration file. Therefore a dependency to the application context is necessary.

License

Spring .NET is licensed under the Apache License, Version 2.0[7]. Thus, it is an OSI Certified Open Source Software. It has only the restriction that the attribution and disclaimer has to be maintained. Important to note is that this software does not come with any warranties.

5.5 SharpDevelop

SharpDevelop[8] (#develop) is an open-source IDE for C#, VB.NET and Boo projects. It is a well known alternative to Microsoft Visual Studio for developing .NET applications. SharpDevelop is completely written in C#. The current released version 2.1 requires the Microsoft .NET Framework 2.0 to run. This IDE is chosen for the evaluation within this diploma thesis because of its architecture. The application consists of a small core which includes an add-in system. Everything outside the core is implemented as add-ins. This architecture supports the idea behind SharpDevelop that it should be an open IDE for various programming languages. Extending SharpDevelop with a new programming language can be done by writing a new add-in [HKS03, p. 8]. The important point is that SharpDevelop can be extended without the need of modifying existing code.

Add-In System

An add-in consists of an XML configuration file with the extension `.addin` and one or more assembly files. The configuration file contains:

- Add-in name, author, description, etc.
- A unique add-in name and the add-in version. The version attribute value can refer to an assembly to use its version number.
- Dependencies to other add-ins. They are defined by the unique add-in names and optionally by the version numbers.
- A list of assemblies used by this add-in.
- *Extension points* (The explanation follows later in this chapter)

The add-in system provides a smart search strategy to locate the installed add-ins. First, it searches for all `.addin` configuration files in the application `AddIns` directory and its sub directories. It is common to divide add-ins in separate sub directories to prevent file name collisions. The second step is the searching for the `AddIns.xml` file in the user profile directory. This file allows the deactivation of add-ins at user level and defining the location of external add-ins. The last step is to search in the user profile `AddIns` directory and sub directories for `.addin` configuration files. Concrete example directories are listed in Table 1.

[7] Apache License, Version 2.0. See http://www.apache.org/licenses/LICENSE-2.0.
[8] The official Website of SharpDevelop: http://icsharpcode.net/OpenSource/SD.

Location	Example Directory
Application `AddIns`	C:\Program Files\SharpDevelop\2.1\AddIns
User Profile	`%AppData%`\ICSharpCode\SharpDevelop2.1
`%AppData%`	C:\User\Juergen\AppData\Roaming (Windows Vista)

Table 1: Example directories of the SharpDevelop add-ins.

Deployment

The add-in system simplifies the deployment of add-ins. The installation process consists of copying the add-in files to one of the special `AddIns` directories. It is not necessary to modify the program configuration. Uninstalling is done by removing the add-in files. A limitation of the add-in system is that the deployment tasks (install, update and uninstall) require a restart of the application.

Add-In Tree

The add-ins are able to extend each other. For example an add-in can extend a toolbar which is hosted by another add-in. SharpDevelop uses a tree structure to manage the extension points [Grunwald06]. The access of a concrete extension point is done via a path (`e.g. /SharpDevelop/Browser/Toolbar`). A path contains nodes and optional sub nodes. The nodes define the behavior of the extension point. The most simple node type is `Class`. This node type is responsible for creating an instance of a defined class by invocation of the parameter-less constructor. Other node types are responsible for creating UI elements or defining file filters for the `OpenFileDialog` or `SaveFileDialog`. These are the node types supported by the core. Further node types can be added by add-ins. The nodes are represented by the `Codon` class which delegates the object creation process to a `Doozer`. The `Doozer` class represents the node type and thus, it implements the behavior of a node.

The extensions are declared in the `.addin` configuration file. This strategy is chosen because the philosophy of the SharpDevelop developer team is to extract as much data into XML files as possible [HKS03, p. 28]. An advantage of this approach is the lazy loading of the add-ins. The `.addin` configuration files are read during application start-up whereas the add-in assemblies are first loaded when one of their extension points is accessed.

Service Locator

The add-in tree can be seen as a configurable factory. One drawback of core implementation is that different add-ins cannot share the same instance of a component which is defined in the `.addin` configuration. Nevertheless, the add-in tree is extensible and an implementation of the Service Locator pattern on the top of it is simple. Listing 7 shows a `SingletonDoozer` that creates only one instance of the object specified in the `.addin` file. The `SingletonDoozer` is registered at the `AddInTree` which is a static class (Listing 8). Listing 9 shows an extract of an `.addin` file that defines a `MovieFinder` component. The xml element `Singleton` refers to the `SingletonDoozer` instance. If the component is retrieved by any add-in, the `SingletonDoozer` is responsible to return an instance of the `MovieFinder` component (Listing 10). A limitation of this implementation is that the shared component needs a default constructor.

```
1  public class SingletonDoozer : IDoozer
2  {
3    private Dictionary<string, object> instances =
4      new Dictionary<string,object>();
5
6    public bool HandleConditions
7    {
8      get { return false; }
9    }
10
11   public object BuildItem(object caller, Codon codon, ArrayList subItems)
12   {
13     string key = codon.Properties["id"];
14
15     if (!instances.ContainsKey(key))
16     {
17       instances.Add(key,
18         codon.AddIn.CreateObject(codon.Properties["class"]));
19     }
20
21     return instances[key];
22   }
23 }
```

Listing 7: A simple implementation for a `SingletonDoozer`.

```
1  AddInTree.Doozers.Add("Singleton", new SingletonDoozer());
```

Listing 8: Registration of the `SingletonDoozer`.

```
1  <Path name="/Service">
2    <Singleton id="MovieFinder" class="MovieFinder.MovieFinder"/>
3  </Path>
```

Listing 9: Define a component in the `.addin` file.

```
1  movieFinder = AddInTree.GetTreeNode("/Service")
2    .BuildChildItem("MovieFinder", null, null) as IMovieFinder;
```

Listing 10: Retrieve the `MovieFinder` component.

Conditions

Every tree node can contain conditions to indicate whether the node should be active. This is useful for dynamic changes of the nodes. For example, the predefined condition `WindowsActiveCondition` can be used to ensure that a toolbar button is only enabled if the editor window is active. The conditions are declared in the `.addin` configuration file.

Framework

SharpDevelop is not primary designed to be a framework for building own applications. Even though, the developer team state that the core can be used for other Windows-based applications. Especially the add-in system would help to build an extensible application.

> *(…) presents the AddIn architecture used in the IDE SharpDevelop, and how you can use it in your own Windows applications [Grunwald06].*
>
> *It is possible to build a completely different application on top of the AddIn tree by just putting other Run commands in the tree. This application can then benefit from the AddIn tree, just as SharpDevelop does [HKS03, p. 56].*

It is also possible to use more than only the core of SharpDevelop. Though, some code has to be changed for the own needs. Changing of the code has the disadvantage that it would be incompatible with the maintained code by the SharpDevelop team.

License

The source code of Sharp Develop is licensed under the GNU LGPL 2.1[9]. Thus, it is an OSI Certified Open Source Software. The license is comparable to the Apache License, Version 2.0 but they are not compatible because of the different dealing with patents [FSF07].

[9] GNU LGPL, Version 2.1. See http://www.gnu.org/licenses/lgpl.html.

6 Evaluation

6.1 Overview

Object-oriented frameworks like the solutions discussed in the previous chapter are difficult to evaluate.

> *One initial difficulty is to understand the intended domain of the framework and its applicability to the application under construction [BMMB97].*

The frameworks which are relevant for this thesis are known as application frameworks [GB01]. Their purpose is to provide all the domain-independent functionality needed in an application. The evaluated solutions extend another application framework, the .NET Framework. In comparison to the .NET Framework they provide additional functionality for more specific domains:

- The Composite UI Application Block can be used for Windows-based *composite applications*.
- Spring .NET supports the building of *enterprise applications*.
- SharpDevelop provides a core suitable for Windows-based applications and additionally ships services for building IDEs.

In the evaluation, the Composite UI Application Block is always used with the extensions provided by the Smart Client Software Factory (Chapter 5.3).

The evaluation of the applicability of these frameworks for the Test Suite application is done by checking the fulfillment of the requirements. This is dealt in the next chapter. The applicability is the most important part in this evaluation. Only suitable frameworks are considered in the next two evaluation parts which are about further quality issues (chapter 6.3) and strategic aspects (chapter 6.4).

In all three evaluation parts the appraisal is done by using three grades:

- (+) … The requirement is completely fulfilled
- (○) … The requirement is partly fulfilled.
- (–) … The requirement is not fulfilled.

6.2 Fulfillment of the Requirements

This chapter presents the first and most important evaluation part. It checks the fulfillment of the requirements which are defined in chapter 2. Table 2 shows a summary of this evaluation which is discussed afterwards.

Requirement	CAB/SCSF May 2007	Spring .NET Version 1.1	SharpDevelop Version 2.1
Runtime platform: Minimum .NET Framework version	(+) 2.0	(+) 1.1	(+) 2.0
Open source and programmed in C#	(+)	(+)	(+)
Test modules			
Define test modules	(+)	(+)	(+)
External configuration	(+)	(o)	(+)
Loose coupling	(+)	(o)	(-)
Lazy loading of modules	(o)	(+)	(+)
Modules deployment	(o)	(o)	(+)
GUI integration			
Support for GUI extension	(+)	(-)	(o)
Command service	(+)	(-)	(o)
Loosely coupled events	(+)	(o)	(-)

Table 2: Checking the fulfillment of the requirements which are defined in chapter 2.

Runtime platform

All solutions run on the .NET Framework 2.0 as it is specified in the requirements.

Open source and programmed in C#

The source code of the three frameworks is available and the used language is C#.

Define test modules

The .NET *assemblies* are an ideal candidate for representing the test modules. Assemblies are the basic unit for versioning, security, and deployment. Hence, they fulfill the requirements. An assembly can physically be a standalone application (`.exe`) or a class library (`.dll`) [Löwy05, p. 23]. In the case of a test module the choice would be a class library. By using assemblies the requirement is already dealt by the .NET Framework. Therefore, the investigated frameworks do not have to provide an own solution for defining the test modules.

CAB extends the .NET assemblies by the introduction of the abstract `ModuleInit` class. A concrete implementation of this class is used to initialize the module. During module loading this concrete class is searched by reflection and initialized through the framework.

Spring .NET does not extend the functionality of the .NET assemblies. Because the .NET Framework already fulfills this requirement, Spring .NET gets full grade too.

In SharpDevelop a module is defined by the `.addin` xml file. This file refers to a .NET assembly by using the `Identity` tag. The `.addin` file extends the meta-data of the assembly with additional information.

External configuration

The configuration for the module loading has to be in an external human-readable file. Changes of the configuration need to be done without recompiling the application. All investigated solutions use an adoption of the Plugin pattern to fulfill this requirement. A short introduction of the Plugin pattern can be found in chapter 3.

The module loader of CAB can be configured via a single XML file. This file contains the module assemblies and the dependencies between them. The information about dependencies assures that the modules are loaded in the correct order.

Spring .NET does not support the loading of modules in the same way as the other solutions do. It can configure the wiring of single components in XML files. For the reason that every module can contain many components the configuration gets extensive. Moreover, the one who is configuring the application needs in-depth knowledge about the dependencies of each component in the module. A way to reduce the amount of configuration is to use the autowiring function of Spring .NET. The setter injection with activated autowiring is limited because it is not possible to define that some properties of an object need to be injected and others not. Autowiring only saves the writing of the dependency information. The components still have to be defined in the configuration file. That is the reason why Spring .NET does not get full mark for this requirement.

SharpDevelop scans special directories for `.addin` files and interprets them. The `.addin` files can define dependencies to other modules in a similar way as in CAB. The main difference to CAB is that in SharpDevelop every module has its own configuration file whereas in CAB a central configuration file is used.

All three frameworks ship XML schema files (XSD) for the configuration. This simplifies the writing and editing of the configuration in an XML schema aware editor. The requirements of this thesis also specify the configuration via command line arguments. The frameworks do not provide direct support for this requirement but the application can extend them to provide this functionality. A possible solution with the CAB framework can be seen in chapter 7.5.

Loose coupling

Loose coupling between the modules is essential for fulfilling the requirements which are defined in chapter 2. It allows the developing of modules by different teams. Furthermore the modules can be isolated for testing. This simplifies the test procedure since the dependent components can be replaced by mock objects. Loose coupling can be achieved by programming to an interface (see chapter 3). Nevertheless, the loosely coupled modules have to work together in a coherent application. This means that the components of the modules have to be weird up.

CAB and Spring .NET support the wiring by the Dependency Injection and the Service Locator implementation. In both frameworks the Dependency Injection and Service Locator implementation work hand in hand together and thus, they can even be mixed in one application. The Service Locator implementation in Spring .NET has one drawback. It does not allow the replacement of a service instance during runtime.

SharpDevelop had a Service Locator implementation called `ServiceManager` in a previous version [HKS03, p. 109], but the current version 2.1 has replaced the `ServiceManager` with static service classes. This step simplifies the service usage but it does not allow the replacement of the services any more. The isolating of components during a test is impossible without replacing the dependent services. A solution for implementing the Service Locator on top of the add-in tree is demonstrated in chapter 5.5. The framework still gets the grade (-) because it does not support loose coupling by itself.

Lazy loading of modules

The Test Suite is an extensive application. For keeping the start-up time at a minimum, the modules have to be lazy loaded. This strategy handles the resources in a smart way, as only the needed ones are allocated. If a user does not use some modules during work these modules will never be loaded. Therefore, unused modules do not waste any resources.

The Composite UI Application Block is not able to load the modules on demand. This is due to the fact that the application integration is done in the module initializing code. However, CAB investigates only the modules via reflection and instantiates the subclass of `ModuleInit`. The services that the modules provide can be loaded on demand with the `ServiceCollection.AddOnDemand` method.

Spring .NET does not support modules in a special way. Therefore, it has no need to load the assemblies at the start-up process. The components, which are needed by other components, are instantiated on demand [Spring07, p. 29]. If more components have to use the same instance, the dependent component can be defined as singleton. By default, singletons are instantiated during the start-up sequence of the container. The `lazy-init` attribute allows delaying the creation until the component requested for is the first time [Spring07, p. 18].

SharpDevelop defines the extension points in the `.addin` configuration files. These configuration files are read at application start-up only. The loading of the add-in assemblies is delayed until one of its extension points is accessed [Grunwald06a].

Modules deployment

Deployment is a topic that is well supported by the .NET Framework. .NET provides version control for the modules and allows side-by-side execution of different module versions in the same process.

The Smart Client Software Factory includes help topics and a reference application for using ClickOnce deployment. ClickOnce simplifies the deployment tasks for the end user and the manufacturer but it has its limitations.

(…) if a program needs to carry out privileged operations that could affect other applications or data on the target machine, such as performing unrestricted file access or accessing the registry, then it may not be suitable for deployment using ClickOnce [Noyes04].

The Test Suite requires unrestricted access to the file system and it has to install native components like system drivers for the test devices. Therefore, ClickOnce is not an option.
Spring .NET does not provide any special deployment features.
SharpDevelop simplifies the deployment of add-ins since the add-in files only need to be copied into one of the specified directories. In contrast to CAB and Spring .NET it is not necessary to modify a configuration file for installing and uninstalling an add-in. A prefabricated add-in named AddIn Manager allows the end user to control the add-ins. A limitation of the AddIn Manager is that the add-ins can be installed into the user profile directory only. This issue is caused by a security restriction of the operating system because it cannot be guaranteed that the end user has write access in the application directory.
It is required to do the deployment tasks without restarting the application which is not supported by all three solutions. The reason is that the .NET Framework cannot unload .NET modules or assemblies. However, it is possible to load every module in a different *application domain*. All application domains except of the default one can be unloaded by the .NET Framework [Löwy05, p. 322]. The drawback of this strategy is that the modules have to communicate through *remoting* with each other. Solutions, which are using this strategy, are the `System.Addin` namespace introduced in the .NET Framework 3.5 (Chapter 8.2) and the CAP .NET project [Dhungana06]. If this requirement is not fulfilled, it is acceptable since it is just defined as a nice-to-have requirement.

Support for GUI extensions

The test modules have to be integrated into the Test Suite user interface. For example, a test module needs to add a new menu item in the menu bar of the Test Suite. The challenge is to create the extension without having a dependency on a concrete UI technology. Furthermore, a test module also needs also the possibility to define own GUI extensions for other modules.
CAB provides a flexible mechanism to extend the user interface. This mechanism consists of two parts which are integrated into the `WorkItem`. The first part is the `Workspace`. It is used for hosting UI controls of other modules. The second part is the `UIExtensionSite`. The extension site allows the extension of exposed UI elements. Every module can use these parts to provide own GUI extensions. Both parts are independent of the UI technology. The Composite UI Application Block supports Windows Forms controls and allows the hosting of WPF controls in the `Workspaces`. If other requirements occur, the framework can be extended.

The Spring .NET framework does not support the building of Windows-based applications out of the box. In version 1.1 the support is limited on Web-based applications which are using ASP .NET. However, the framework can be extended with the required functionality.
SharpDevelop provides GUI extensions through the `.addin` configuration files. This extension mechanism is independent of the UI technology. The core supports the handling of Windows Forms controls. If other technologies must be used, a rewriting of the core is necessary. The rewriting of code is not the best strategy to extend the functionality. After modifying the core it has to be accurately tested to assure that no side-effects occur. An advantage of SharpDevelop is that every module can register its own GUI extensions by defining a new `AddInTree` path. A reusable mechanism for hosting of UI elements like the `Workspace` of CAB is missing. Instead, SharpDevelop uses the specific `WorkspaceSingleton` class to host the Windows Forms controls.

Command service

An implementation of the Command design pattern [GHJV95, p. 233] is necessary for the Test Suite. It is required to decouple the UI elements from the command handlers.
The Composite UI Application Block contains a command service. It is managed by the `WorkItem` container. It is possible that different UI elements can register themselves as command invoker to the same command as required. An adequate `CommandAdapter` is necessary for registering a UI element. If a UI element type is not known by the framework, a new adapter can be registered in the `ICommandAdapterMapService`. Furthermore, the command supports the notification of more than one command handler. The defining of a command handler is simple because the `CommandHandler` attribute just needs to be attached to the method.
Spring .NET does not have a command implementation for user interface elements.
SharpDevelop provides a command implementation. The commands are defined in the `.addin` file as an attribute of the associated UI element. Different UI elements can use the same command class. The command class has to implement the `ICommand` interface. From this it follows that the command class is already the command handler. It is not possible that a second command handler can handle the same command. Another drawback is that the command implementation is not able to handle the command state. For example, the state is responsible for deactivating all associated UI elements if the command cannot be executed. In SharpDevelop this is done by `Conditions`. Nevertheless, the `Conditions` are associated directly to the UI elements instead of associating to the commands. If more UI elements do the same task the `Conditions` have to be applied to all of them. This means code duplication in the `.addin` file.

```
1  ...
2
3  <Condition name = "ActiveWindowState" windowstate="Dirty,Untitled"
4      nowindowstate="ViewOnly" action="Disable">
5    <ToolbarItem id = "Save"
6      ...
7
8  <Condition name = "ActiveWindowState" windowstate="Dirty,Untitled"
9      nowindowstate="ViewOnly" action="Disable">
10   <MenuItem id = "Save"
11     ...
12
13 ...
```

Listing 11: An extract of the `ICSharpCode.SharpDevelop.addin` file that shows code duplication.

Listing 11 shows the save `ToolbarItem` and the save `MenuItem`. Both items require the same condition statement since the items have the identical meaning. In the SharpDevelop `.addin` file the condition statement is duplicated.

Loosely coupled events

The framework has to support loosely coupled events for communication between the modules. Hence, two objects can register themselves as publisher and subscriber without knowing each other.

CAB supports the loosely coupled events even on dependency injection level. The publishers have to define their event declaration with the `EventPublication` attribute. The subscribers define the event handler method with the `EventSubscription` attribute. The attributes use a string as identifier for the event topic. The event publisher and subscribers are wired together by the framework during the object creation.

Spring .NET has a built in support for loosely coupled events too. The objects have to register themselves via the `IEventRegistry` interface at a central registry as publisher or subscriber. It is possible to create own event registries but it is more common to use the central event registry which is provided by the `IApplicationContext`. A Dependency Injection style of event wiring is not supported. Nevertheless, the main drawback is that a subscriber can wire itself to a specific event only if a unique delegate type is used for the event. Otherwise, the subscriber has to handle all events that match with its methods signatures. The only filter that can be applied during registration of a subscriber is the publisher type. Though, this can be a problem because the subscriber needs a reference to the publisher for applying the filter.

SharpDevelop does not support loosely coupled events at all.

6.3 Further quality issues

The main quality aspect is that the frameworks provide the required functionality. This is discussed in the previous chapter. However, this chapter deals with further quality issues of the framework. Table 3 shows a summary of this evaluation part. Afterwards the evaluation is discussed in detail.

Requirement	CAB/SCSF May 2007	Spring .NET Version 1.1	SharpDevelop Version 2.1
Framework composition	(○)	(+)	(+)
Framework dependencies	(○)	(+)	(−)
Evolution	(+)	(+)	(−)
Documentation			
Purpose of the framework	(+)	(+)	(+)
How to use the framework	(+)	(+)	(○)
Detailed design of the framework	(+)	(○)	(○)
Examples	(+)	(+)	(○)

Table 3: Shows the evaluation of further quality issues.

Framework composition

It is common that frameworks must be composed with other frameworks to get all the functionality which is required for the application. Frameworks are often designed that they are in full control. Problems can occur if two composed frameworks require both full control over the application [BMMB97, p. 11]. In case of the Test Suite all investigated frameworks are designed for working with the .NET Framework. Thus, no issues occur if the frameworks are composed together with the .NET Framework. The situation changes if other frameworks, like a persistence framework, are added.

In CAB and Spring .NET the framework takes over the control during Dependency Injection. All the necessary objects for the Dependency Injection are created by the framework. This can be an issue if the frameworks are composed with other frameworks. For example, a persistence framework is usually responsible for retrieving the persisted data and creating the necessary objects which are filled with this data. In this case, both frameworks want to create the same objects. A solution is that these objects are not created by Dependency Injection. Instead, the objects retrieve the necessary services via the Service Locator implementation. In Spring .NET the situation looks a bit better as the framework already supports some additional frameworks (e.g. NHibernate) out of the box.

SharpDevelop is not affected by this requirement because it does not provide Dependency Injection.

Framework dependencies

A characteristic of frameworks is that beside code reuse they also define the application design. The application developer can concentrate on the implementation without worrying too much about creating a good object-oriented design. However, the drawback is that a framework is closely coupled to the application.

> *As a framework evolves, applications have to evolve with it. That makes loose coupling all the more important; otherwise even a minor change to the framework will have major repercussions [GHJV95, p. 27].*

Dependency Injection helps to minimize or completely remove the dependencies between the framework and the application. The investigated frameworks, which support decoupling, use this pattern. An introduction in Dependency Injection can be found in chapter 4.2.

The Composite UI Application Block uses the Object Builder for Dependency Injection. Nevertheless, it does not completely decouple the modules from the framework since the modules use `Attributes` to control the injection. The `Attributes` are part of the framework. Therefore a reference to the framework is necessary. Other dependencies arise through sub classing of the `ModuleInit` class and the use of own sub `WorkItems`. Additionally, the Dependency Injection implementation is not able to inject primitive types. If a service has to be configured with primitive types, it has to be manually instantiated and passed to the Service Locator. The use of the Service Locator also requires a reference to the framework assembly. With all these dependencies to the framework they are still lower as without using Dependency Injection at all.

Spring .NET provides a powerful Dependency Injection mechanism that allows the complete decoupling of the modules from the framework. One exception is the use of loosely coupled events. The modules which are using them require access to the `IApplicationContext` and consequently, a reference to a Spring .NET assembly is necessary. Another drawback is the high amount of configuration which is necessary for the complete decoupling of the components.

SharpDevelop does not support Dependency Injection and thus, the add-ins are closely coupled to the SharpDevelop core.

Evolution

The number of breaking changes during the framework evolution is an important quality metric. Braking changes are all modifications in a framework that require the application to be modified too. If many breaking changes are introduced in a framework the effort for updating the applications to a new framework version is high. This metric is very difficult to define. One solution is to have a look at the past years of the framework and use this data to predict the future.

Since CAB was released in December 2005 no chances were made until September 2007. Microsoft has introduced new features with the Smart Client Software Factory, but the CAB framework itself was not updated by SCSF. The fact, that the framework did not need to be changed since its release, approves the high quality of the framework.

In Spring .NET the breaking changes of the version 1.0 release in September 2005 until the version 1.1 RC1 release in August 2007 are minimal[10]. It shows also here that Spring .NET is a well designed framework. The conclusion is further supported as version 1.1 introduces lots of new functionality without the need for changing the API.

SharpDevelop is not intended to be a framework. That is why the stability of the interfaces is not that important than in the other solutions. An example for a breaking change with high impact is the elimination of the `ServiceManager`. The book "Dissecting a C# Application - Inside SharpDevelop" that was first printed in February 2003 describes the `ServiceManager` [HKS03, p. 109]. In the version 2.1 the `ServiceManager` does not exist anymore.

Documentation

Johnson discusses the amount of documentation that is necessary for a user to understand a framework [Johnson92]. He states that the documentation should contain:

- The purpose of the framework.
- Information on using the framework.
- The detailed design of the framework.
- Examples.

The purpose of all investigated solutions is well documented. The information on using the framework is useful in CAB and Spring .NET but lacks with SharpDevelop. This results from the fact that SharpDevelop is not intended to be used as an application framework. The detailed design of the framework is only sufficiently explained for CAB. The documentation of Spring .NET shows every possible usage scenarios of the framework but does not explain the internal design in detail. The SharpDevelop team has devoted a book to the design decisions they made in the application [HKS03]. However, the book is outdated today and does not reflect the current state of SharpDevelop. CAB and Spring .NET ships useful examples which are also documented. In the case of SharpDevelop just a few examples exist regarding the usage of the core to build own applications. These examples are not in the official documentation of SharpDevelop.

6.4 Strategic aspects

The last part of this evaluation investigates some strategic aspects. These aspects should help to forecast the availability of the framework in the future. Further, the risk should be estimated, if the maintenance of the framework is discontinued soon. Important to note is that the information for this part is gathered at July 3, 2007.

[10] Breaking changes of Spring .NET 1.1:
http://www.springframework.net/BreakingChanges-1.1.txt

Requirement	CAB/SCSF May 2007	Spring .NET Version 1.1	SharpDevelop Version 2.1
License	(○)	(+)	(+)
Roadmap	(○)	(+)	(-)
Support	(+)	(+)	(○)
Number of involved Persons	(○) 14	(+) 18	(-) 2

Table 4: Shows the strategic aspects of the evaluation (July 3, 2007).

License

All investigated solutions are under an open source license. This allows viewing and editing the source code. It can be helpful for maintaining the framework by own staff if it is necessary. However, this should only be a backup plan because the frameworks are quite extensive. It would take some time for a developer to understand the framework enough for executing maintenance tasks. The CAB framework has a limitation since Microsoft does not allow their Application Blocks to run on other operating systems than Windows. Right now this is not an issue for the Test Suite but it limits the strategic decisions for the future.

Roadmap

Roadmaps are useful information for learning what direction a framework will take. Still the information is mostly a matter of change and it is not possible to rely on it. Furthermore, roadmaps do seldom show a strategic long term plan of a framework.
Microsoft does not have any plans to develop the Smart Client Software Factory and CAB any further [Block07]. The reason is the announcement of the project Acropolis which should become the successor of CAB. Microsoft states that they are working on a migration path from CAB to Acropolis.
On the official website of Spring .NET it is possible to find plans for the successor version 1.2 and 1.3 already. With this information it is possible to state that Spring .NET is further developed and maintained in the near future.
The official website of SharpDevelop shows a roadmap for the successor version 3.x. However, the roadmap does not contain any information about changes at the core level.

Support

Support is not that important as the solutions ship with the source code. Even though, it can be more economic to use the commercial support instead of an employee for specific tasks. Typical tasks for using support are bug fixing in the framework and employee training.
Microsoft provides support for the Smart Client Software Factory and CAB. This information can be found on the official SCSF website.
The company Interface21 coordinates the Spring Framework .NET project. Information about available training can be found on the official Spring .NET website.

The building of applications on top of the SharpDevelop core is not supported by the company IC#Code. This company leads the SharpDevelop project.

Number of involved persons

The number of involved persons in the projects can help to determine the risk of discontinuing an open source project.

Microsoft has already announced that the Composite UI Application Block is discontinued. A migration path to the successor project is already in work. The number of involved persons in the SCSF project is read from the SCSF community website[11].

In the Spring .NET project 18 persons are involved. The risk is minimal that Spring .NET does not survive for the next years. This number is read from the SourceForge website[12].

In SharpDevelop are just two persons which are registered to be involved. Therefore, the risk for SharpDevelop to be discontinued is high. The number is read from the SourceForge website[13].

6.5 Decision

The most applicable framework for the Test Suite application is the Composite UI Application Block with the extensions provided by the Smart Client Software Factory. CAB supports all defined requirements of chapter 2 at minimum partly. It is superior to the other solutions because it has the highest degree of fulfilling the requirements. Therefore, an implementation of the Test Suite on top of CAB causes the least effort.

Spring .NET has some advantages over CAB. However, it lacks the support for developing Windows-based applications. Due to the extensible nature of the framework it would be possible to add the missing features. It still requires a respectively high effort to implement them. Another major drawback is the amount of configuration that is necessary to build an application in the size of the Test Suite. Further, it requires in-depth knowledge of the modules to wire them together. The autowiring functionality does not help much in this case.

The SharpDevelop project is not applicable since it does not support louse coupling. This results in difficulties for separating the programming to different developer teams. Additionally, it makes the testing of the modules complicated. The high dependency to the framework and the lack of loosely coupled events does not meet the requirements. SharpDevelop is not intended to be an application framework. Thus, the interfaces are more unstable than in the other solutions investigated. Unstable interfaces in a framework result in a high maintenance effort for own applications.

[11] The project website on CodePlex: http://www.codeplex.com/smartclient

[12] The project website on SourceForge: http://sourceforge.net/projects/springnet

[13] The project website on SourceForge: http://sourceforge.net/projects/sharpdevelop

Both solutions, Spring .NET and SharpDevelop, have a few advantages over CAB. In other projects, the decision on using one of these solutions may be better. If one or none solution is applicable for an application, a study of the design ideas of these frameworks is useful too. The prototype presented in the next chapter is built on top of the Composite UI Application Block. The prototype implementation even uses some good ideas from the SharpDevelop project.

7 Prototype

7.1 Overview

The prototype is a basic implementation of the Test Suite. It presents possible solutions for the requirements which are defined in this diploma thesis. The implemented functionality in the prototype is limited to the scope of this thesis. It does not represent a full featured Test Suite. The development of the prototype has been concentrated on a few features, which are able to show the framework integration, the use of different UI technologies as well as the shortcomings of the CAB framework. The result is an infrastructure block and a collection of modules that show different aspects of the requirements.

7.2 Architecture

The prototype is primarily a .NET 2.0 application that runs on top of the Composite UI Application Block. One module uses the new UI technology called Windows Presentation Foundation already which is part of the .NET Framework 3.0 (Figure 9). Important to note is that the .NET Framework 3.0 is built on top of the .NET Framework 2.0 and introduces four new framework libraries only. WPF is one of them. The .NET Framework 3.0 does not change the common language runtime (CLR) or the base class library (BCL) of the .NET Framework 2.0. That is the reason why it is possible to mix .NET 2.0 and .NET 3.0 assemblies in one application.

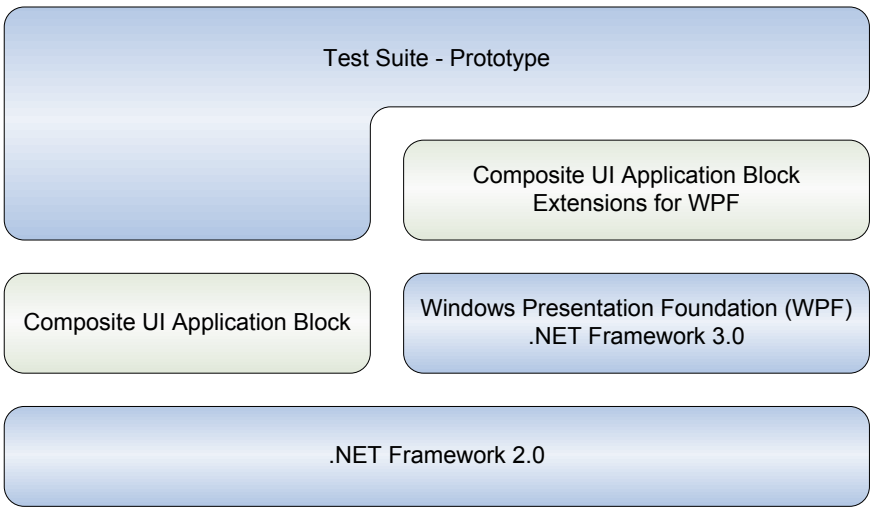

Figure 9: Prototype architecture.

Besides the usage of CAB and the extensions of SCSF as an application framework, some further libraries are used in the Test Suite:

- The *Exception Handling Application Block* allows to define a consistent handling for all exceptions that occur in the application. It is part of the Enterprise Library 3.1.
- The *Logging Application Block* extends the logging functionality of the .NET Framework. It is part of the Enterprise Library 3.1 too.
- The *DockPanel Suite* is a docking library for Windows Forms controls. It mimics the look and feel of the Visual Studio 2005 IDE. The library is licensed under the open source MIT license.
- The *CAB Extension* is a library which was developed during the implementation of the prototype. It adds some reusable features to the Composite UI Application Block.

7.3 Modules

This chapter gives a short overview of the implemented modules that can be used by the Test Suite. Figure 10 uses an UML component diagram to show the modules with their dependencies to each other. Every component with the stereotype `<<cab module>>` represents a .NET assembly which implements a CAB module. The interfaces shown in Figure 10 are implemented as separate .NET assemblies. They provide all the necessary information for a CAB module to use and extend the module which implements the interface assembly. It is also possible to divide the implementation of an interface assembly into more CAB modules like it is done with the `Infrastructure.Interface` assembly. An interface assembly consists of `Interfaces` to decouple the service implementation and `string` identifiers to access the UI integration functionality of the Composite UI Application Block. An advantage of this approach is that the modules can be replaced on both sides of the interface assembly. This is done for isolating a single module during unit testing.

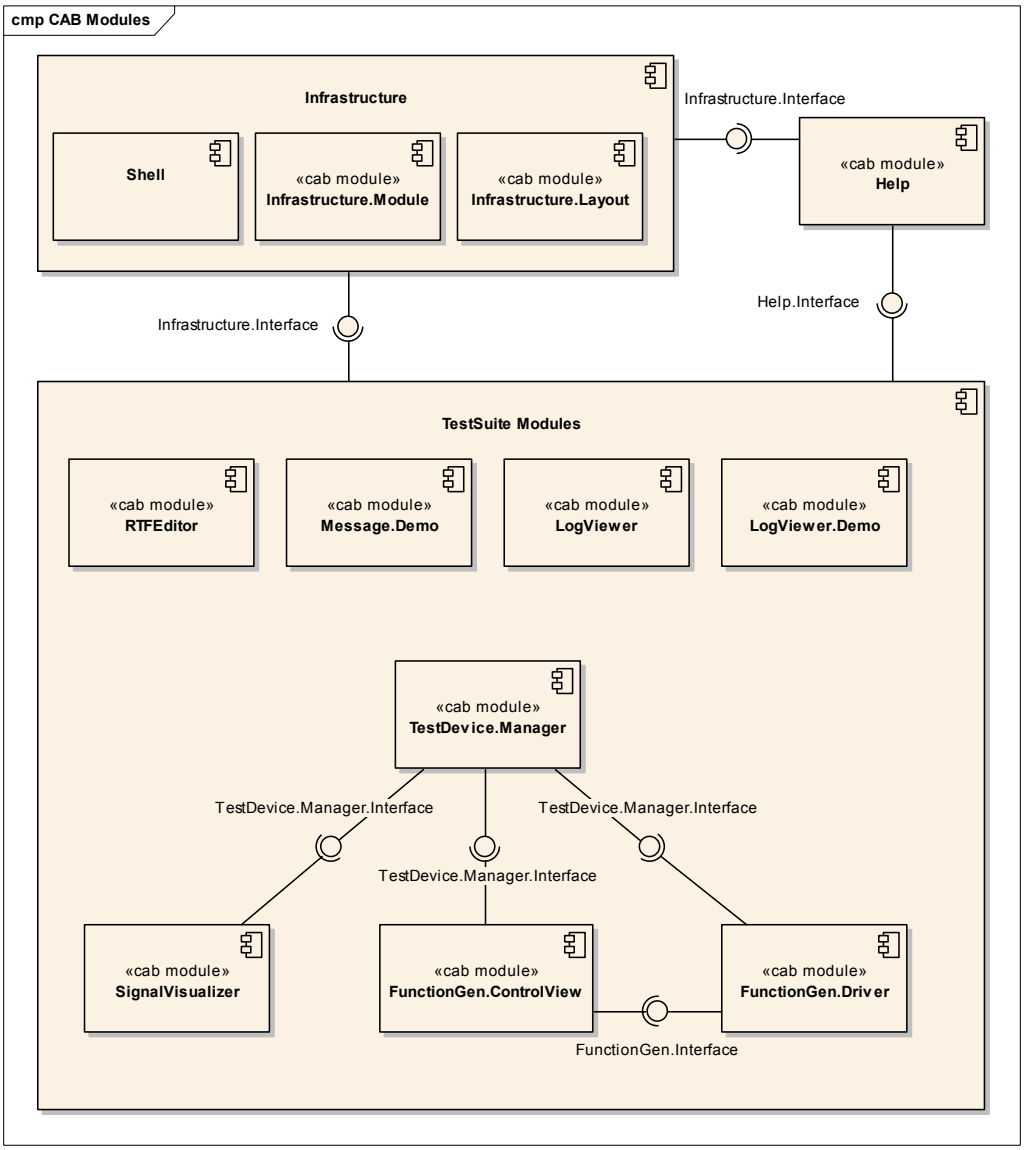

Figure 10: An UML component diagram that shows the modules of the Test Suite.

Infrastructure

The `Infrastructure` block in Figure 10 contains the core of the Test Suite application. All CAB modules require the functionality which is provided by the infrastructure. Thus, the implementation of the infrastructure must be loaded first by the application. The implementation is divided into three modules:

- `Shell`
- `Infrastructure.Layout`
- `Infrastructure.Module`

This allows the replacement of one module without affecting the others. For example, the `Infrastructure.Layout` module can be replaced by another one to define a new UI layout for the application.

Many of the infrastructure functions are implemented as CAB services. These services can be retrieved by the Dependency Injection implementation of the Composite UI Application Block. If Dependency Injection cannot be used, the services can be fetched from the `WorkItem`, which implements the Service Locator pattern.

The `Shell` is the .NET assembly that contains the start-up code of the application. It is responsible to initialize the application and to configure the CAB framework. The `Shell` contains a message service for showing all kinds of messages to the user. The reason for implementing the message service in this assembly is that an exception can already occur during the application start-up. The `Shell` processes all unhandled exceptions that are thrown in any module of the application. An unhandled exception is shown to the user with the message service and is logged by the Logging Application Block.

The `Infrastructure.Layout` module defines the appearance of the application. It uses the DockPanel Suite library to define a user interface layout that is very similar to the one of the Visual Studio 2005 IDE. A `Workspace`[14] is necessary for using the features of CAB to host views inside the DockPanel control. The CAB Extension library contains an exemplary `DockPanelWorkspace` which is used by the Test Suite application. This `DockPanelWorkspace` is able to host Windows Forms and WPF user controls. Additionally, the layout module registers UI elements as `UIExtensionSites`[15] so that other modules can extend them (e.g. the menu bar). Furthermore, this module registers common commands like copy and paste. Figure 11 shows a screenshot of the Test Suite. It contains markers to show which CAB parts are behind the UI elements.

[14] See also Shell Services (p. 21).
[15] See also Shell Services (p. 21).

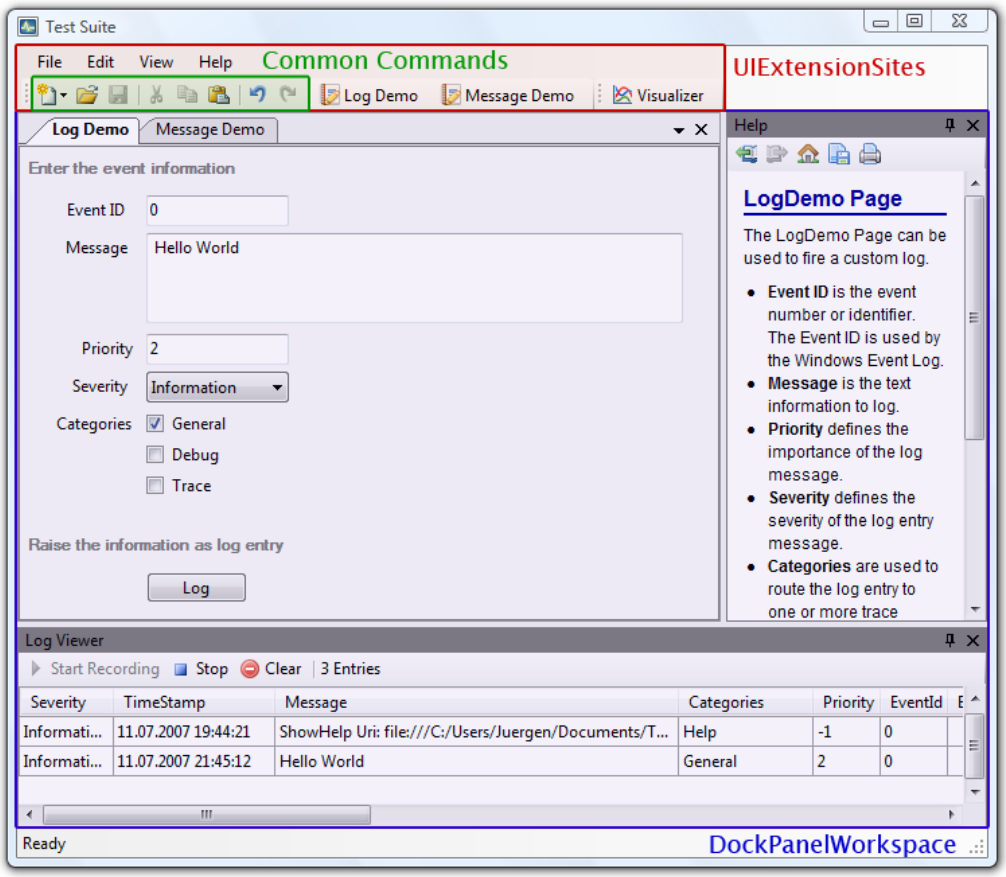

Figure 11: Shows the UI elements of the Test Suite.

The `Infrastructure.Module` contains further services:

- The `IUIElementCreationService` is used for creating UI elements which can be added to `UIExtensionSites`. This service helps to decouple the modules from the UI technology used by the `Shell`.
- The `IDocumentManager` handles the document lifecycle tasks and keeps track of all registered document types. This service mediates between the user interface and the document. A module developer, who has to implement a new document type, does not have to care about things like configuring the `OpenFileDialog` component or enabling and disabling the save buttons.
- The `IEditManager` maps the basic edit functions (e.g. copy, paste …) of an object to the edit menu of the application. Like the `IDocumentManager`, this service also mediates between the user interface and an object. This object needs to implement the `IEditHandler` interface. Many of the Windows Forms controls and the WPF controls provide some of the methods which are required by the `IEditHandler` interface. An adapter is necessary for using one of the UI controls as an edit handler. This module already contains a few adapters for common UI controls like the WPF `TextBox`. To simplify the registration of an object for the `IEditManager`, this module contains an adapter factory catalog.

- The `AdapterFactoryCatalog<IEditHandler>` service is used to register new `IEditHandler` adapters and retrieve adapters for a specific object. In Listing 12 the two WPF `TextBox` controls `scaleX` and `scaleY` are registered at the `IEditManager` service. An important fact is that these objects do not implement the `IEditHandler` interface. Thus, the `Register` method asks the `AdapterFactoryCatalog<IEditHandler>` for an appropriate adapter. By using this adapter factory catalog, the module developer does not have to care about the `IEditHandler` interface as long as an adapter is already registered for the needed object type.

```
1  [ServiceDependency]
2  public IEditManager EditManager
3  {
4      set
5      {
6          editManager = value;
7          editManager.Register(scaleX);
8          editManager.Register(scaleY);
9      }
10 }
```

Listing 12: Registering of two WPF `TextBox` controls to the `IEditManager`.

Help

Figure 10 shows a dependency between the Test Suite modules and the `Help` module. This module is optional because it is not part of the `Infrastructure`. A module developer has to keep in mind that the service, which is provided by the help module, might not be available. Thus, a module has to check if the help service is available before it can be used. The `Help` module shows the help topics inside a `WebBrowser` control. The prototype uses *HTML* files for the help pages.

Demonstration Modules

Three CAB modules in Figure 10 just demonstrate some of the functionality which is provided by the infrastructure. These modules are guidelines for using the infrastructure of the Test Suite. The modules are:

- `LogViewer`
- `LogViewer.Demo`
- `Message.Demo`

The `LogViewer` module allows the user to see the log entries in an application window. This module uses the logging mechanism of the Logging Application Block. The `LogViewer` provides a `CustomTraceListener` which can be configured in the application configuration file. The configuration can contain different filter criteria to limit the log entries, which are shown in the `LogViewer` module.

The `LogViewer.Demo` is a sample module for showing how to use the logging mechanism of the Logging Application Block. This module contains a view to define the various log entry properties. A click on the log button writes the log entry to the logger. The written log entries can be seen in the `LogViewer` module, in a text file or somewhere else. The output depends on the configuration of the logging mechanism.

The `Message.Demo` module is a demonstration of the message service provided by the `Infrastructure`. It allows the user to show messages in a modal dialog, to update the application status bar and to throw a predefined exception. The function to throw an exception is used to see the reaction of the application on unhandled exceptions. By default, the application shows unhandled exceptions through the message service and logs the occurrence of the exception via the Logging Application Block.

Editor

The `RTFEditor` module is an example to show how document oriented applications can be created with the Composite UI Application Block. In a real Test Suite the documents would be test reports with the feature to add some notes by the user. For simplicity, the document in the prototype is a *RTF* file.

The module shows that the lifecycle of a `WorkItem` can be used to represent the lifecycle of a document. The `WorkItemController`, which is aggregated by the `WorkItem`, implements all the necessary functionality of a document object. The controller also has the responsibility to show the document inside the Test Suite user interface. The `RTFEditor` module uses the `IDocumentManager` to control the document lifecycle. This service decouples the module from the application because the module is not aware of how the application shows the create, open, save and close functionality of the documents to the user. This module uses the `IEditManager` too. This service mediates between the application user interface and the `RichTextBox` control which is used to show the document. Thus, the module does not need to care about things like disabling the cut and copy button if no text is selected.

Test Device Management

The CAB modules, which are shown at the bottom of Figure 10, are responsible for the management of various test devices. The main module is the `TestDevice.Manager`. This module is responsible for the lifecycle of the test devices. It shows the connected devices in a list as it can be seen in Figure 12. The user is able to configure and to disconnect one or more connected test devices. The prototype does not contain modules for handling real test devices. Thus, virtual test devices were invented. The `TestDevice.Manager` module is in charge for creating virtual test devices. The function generators shown in Figure 12 are virtual devices too.

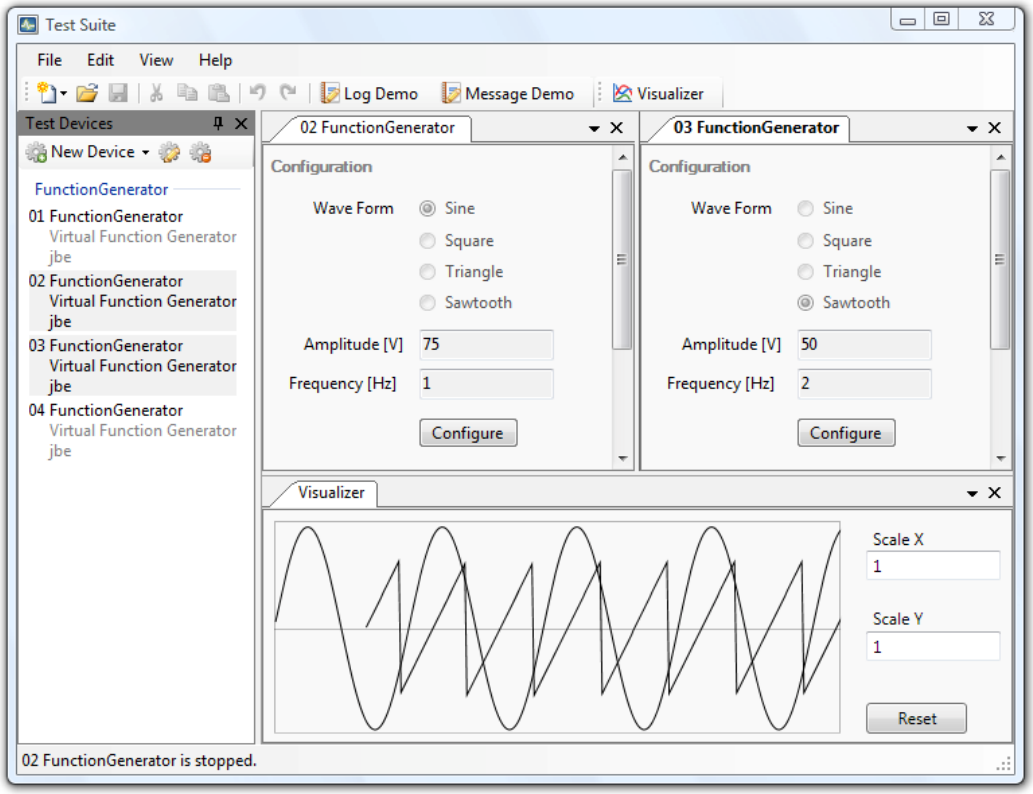

Figure 12: A screenshot of the Test Suite with the `TestDevice.Manager`.

All of the following modules are depending on the functionality which is provided by the `TestDevice.Manager` module (Figure 10). This functionality does not only consist of services. It also includes an `UIExtensionSite`, `Commands` and a loosely coupled event. This shows that every CAB module is able to provide its own user interface extensions for other modules.

The FunctionGen.Driver module represents a driver for virtual function generators. A driver module is responsible to inform the TestDevice.Manager about connected and disconnected test devices. In the case of virtual test devices, the manager triggers the creation of new devices. It is transparent for the TestDevice.Manager if a test device is a real one connected to the computer or a virtual one created by the driver module. The driver contains information about the supported test devices. This information is read by the manager. The virtual function generator of the FunctionGen.Driver module is able to create sine, square, triangle and sawtooth wave forms. Furthermore, the amplitude and the frequency can be configured. This module does not contain any UI elements to configure the virtual function generators. This is in the responsibility of the FunctionGen.ControlView.

The FunctionGen.ControlView module controls device drivers of the category function generator. It contains UI elements for the user to manage the device configuration and the device status. Figure 12 shows the UI elements of this module. The separation of the driver and controller view responsibility into different modules has the advantage that the controller view can be reused. The FunctionGen.ControlView is not bound to the FunctionGen.Driver module. The controller view can also be used for other function generator drivers. A new function generator driver just implements the interfaces of the FunctionGen.Interface assembly. Additionally, the driver has to register at the TestDevice.Manager module with the same profile name as the FunctionGen.ControlView does.

Figure 13 shows the process of how to create a new virtual function generator. The stereotypes in the sequence diagram contain the information from which module an object comes from. In this case the TestDeviceManager has the role of the actor because the manager is in charge for triggering the creation of virtual test devices. The interesting aspect of this process is that all needed services are registered at the same WorkItem. This way the FGenControlView object can access the driver services to control the test device.

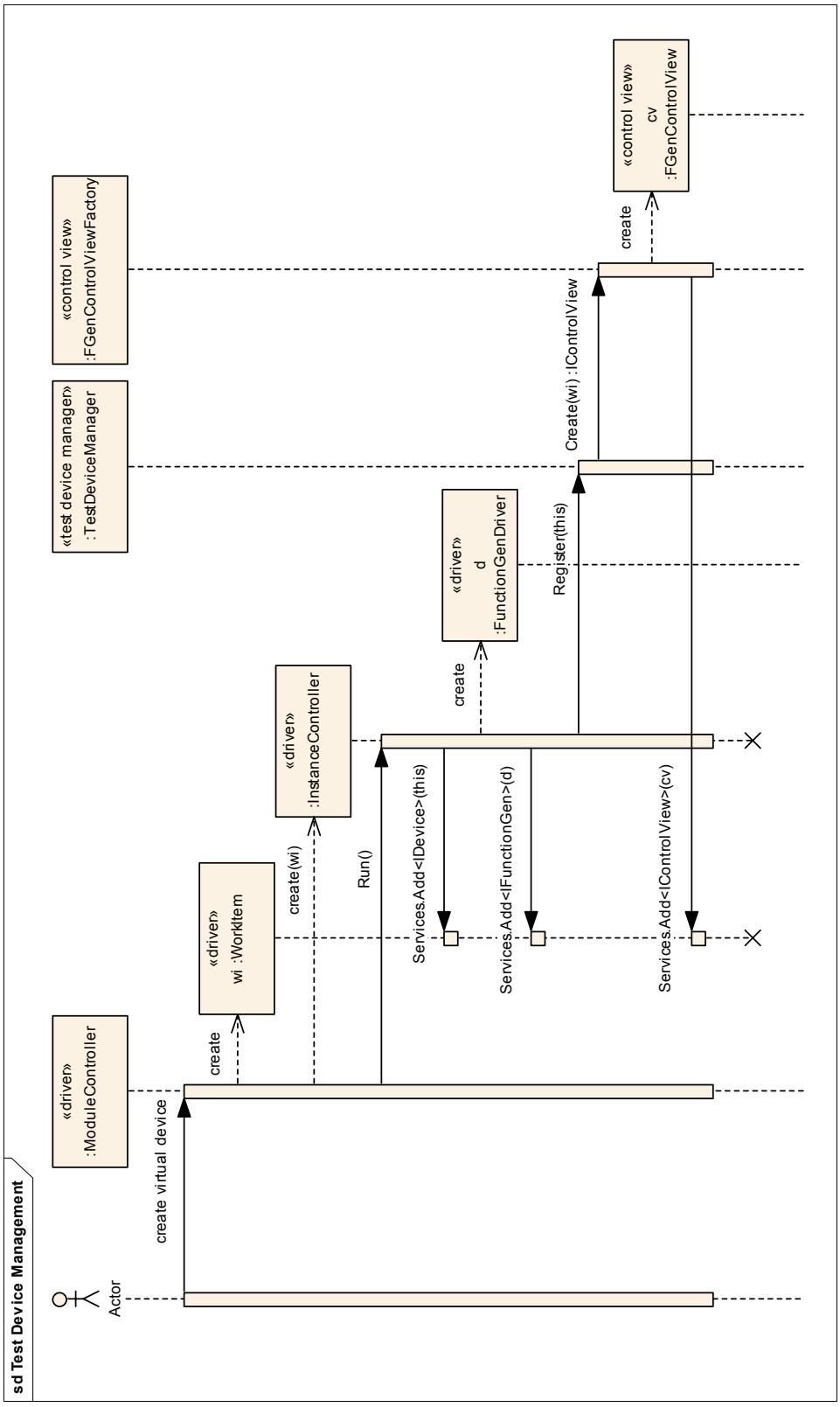

Figure 13: A simplified UML Sequence diagram of creating a new virtual device.

The `SignalVisualizer` module draws a graph of one or more signals. Figure 12 shows the visualizer at the bottom of the screenshot. In the screenshot two graphs are visible. These graphs are created by two different function generators which are running simultaneously. The signals are typically raised by the device drivers. The loosely coupled event mechanism of CAB is used to carry the signal from the source to the visualizer. The source creates a `SampleEventArgs` object which contains the amplitude and the timestamp of the signal sample. This object is sent through the loosely coupled event mechanism to all interested receivers. The visualizer is one of them. The reason for this design strategy is the decoupling of the visualizer module by using the CAB event broker. This module does not have any dependency to the `FunctionGen.Driver` module which acts as signal source. The `SignalVisualizer` is the only module of this prototype application which uses WPF controls. However, it can be completely integrated into the Test Suite application, although the application uses the Windows Forms technology.

7.4 WorkItem hierarchy

The core element of CAB is the `WorkItem`[16]. It represents the container which manages all the objects instantiated by Dependency Injection. Figure 14 presents the `WorkItem` hierarchy of the Test Suite.

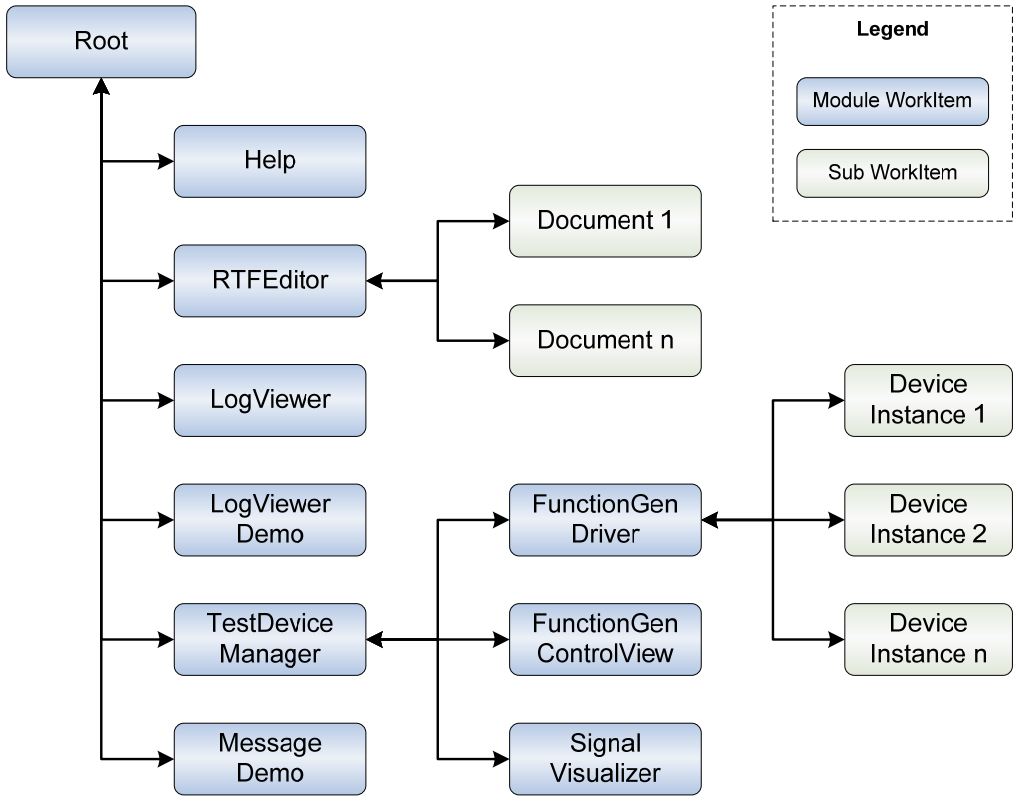

Figure 14: The WorkItem hierarchy of the TestSuite.

[16] See also WorkItem (p. 20).

Typically, every CAB module provides its own `WorkItem`. The services registered by a module are available at the module `WorkItem` or any sub `WorkItems`. Only the services registered at the root `WorkItem` can be accessed in the whole application. Due to this fact, the modules `FunctionGen.Driver`, `FunctionGen.ControlView` and `SignalVisualizer` register their module `WorkItems` in the `TestDevice.Manager WorkItem`. Hence, these `WorkItems` can access the services provided by the `TestDevice.Manager`.

Sub `WorkItems` are often used to handle a part of the use case. In Figure 14 the `Documents` are such sub `WorkItems`. A `Document WorkItem` manages the lifecycle of a document and it controls the UI window which shows the content to the user. The other sub `WorkItems` seen in Figure 14 are the `Device Instances`. Each of them represents a test device.

7.5 Implementation of the requirements

This chapter describes how the requirements are implemented by using the Composite UI Application Block. For most of the requirements the prototype only shows one of the possible ways to solve them.

Configuration of the module loader

The configuration of the module loader is done in an XML file. The prototype is using the built-in dependency module loader which is shipped with the Smart Client Software Factory. It allows the definition of dependencies inside the XML file.
Additionally, the configuration via command line arguments of the application is required. The first idea was to pass all information of the XML configuration file as command line arguments. This idea is not practical because all the information in a single line is not readable any more. Therefore, a command line argument parser is implemented to let the user choose which XML configuration file should be used. The different configurations can be defined in XML files. If the user does not set the command line argument, it reads the default file `ProfileCatalog.xml`.

Isolate the objects under test

The prototype includes a unit test project for the `TestDevice.Manager` module. It is using the Visual Studio 2005 unit test framework. The project tests all non-UI classes of the module. These are the `ModuleController`, the `TestDeviceManager` and the `TestDevicesViewPresenter` class. Additionally, the UI-class `TestDevicesView` is partially tested. A complete test of an UI-class requires special tools or frameworks because the input devices like a mouse have to be simulated.

The notable aspect is that all of these classes are completely isolated for the tests. This means that all the needed objects of these classes are replaced by mock objects. The needed objects are mostly services from the framework but they can also be collaborators of the same module. Before a unit test runs, the framework is initialized with these mock objects. The mock objects can be used to test the correctness of the interaction between them and the object under test. The abstract class `FixtureBase` initializes the framework with the test configuration. All the test classes derive from `FixtureBase` and reuse the framework setup. The unit test writing can be done with minor effort by reusing the framework setup.

Lazy loading

The Composite UI Application Block is able to load services on demand. The prototype is using this feature for the `DocumentManager`. The `DocumentManager` uses the `OpenFileDialog` and the `SaveFileDialog` class which uses native resources. With performance in mind these classes are typically lazy instantiated to save resources if they are not used. The framework already implements service loading on demand. Therefore, a software developer does not need to care about lazy loading inside the services.

Modules deployment

The projects that are created with the Smart Client Software Factory in Visual Studio build all their files in the same directory. This may become problematic because the resource filenames could be identical with the ones of other modules. It is also possible that the modules use different versions of the same assembly. If all the files are deployed in one directory, the installation of a new module could overwrite files from other modules. The Test Suite uses customized build paths for deploying every module in a separate directory to prevent the mentioned issues (Figure 15). Furthermore, the module loader configuration files have to be adapted to the new path names.

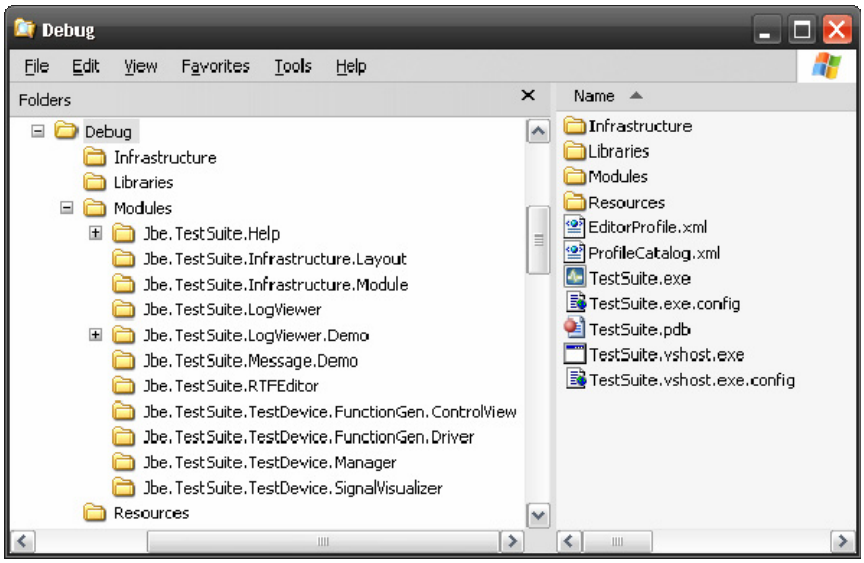

Figure 15: File structure of the Test Suite.

Beside the `Modules` directory tree, the following directories are created:

- `Infrastructure` contains all files for the infrastructure. The infrastructure files are the application assemblies that are required by all other modules.
- `Libraries` include all the necessary library files that are shared by the modules. For example, the CAB assembly files are in this directory.
- `Resources` contain all the resource files that are shared by the modules.

By using these sub directories, the *common language runtime* (CLR) has to be told where it is going to find the assemblies. A way to accomplish this, is to add the `probing` element in the application configuration file as it is shown in Listing 13.

```
1  <runtime>
2    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
3      <probing privatePath="Libraries;Infrastructure" />
4    </assemblyBinding>
5  </runtime>
```

Listing 13: Extract of the application configuration file of the Test Suite.

Support for GUI extensions

CAB supports GUI extensions in two ways. The first one is via the `Workspaces`. They are responsible to host the views of the modules. The Test Suite provides a `DockPanelWorkspace` to host views inside the shell form. Furthermore, it contains the `FormWorkspace` to host views in an own modal dialog. Both `Workspaces` are able to host Windows Forms controls and WPF controls. Due to this feature, a step-by-step migration of the older UI technology to the newer one is supported. All the modules of the Test Suite prototype use Windows Forms controls for their views except of the `SignalVisualizer`. This module implements all its views as pure WPF controls.

The second way to support GUI extensions is done through `UIExtensionSites`. The `Infrastructure.Layout` and the `TestDevice.Manager` module register UI elements as extension sites. The other modules are able to extend these UI elements. For example, the `SignalVisualizer` adds a new menu item into the drop down list of the View menu item which is defined by the `Infrastructure.Layout`. The help topics of CAB and the reference application of SCSF show how to use `UIElementSites`. In their examples, the modules create the concrete UI elements and add them to a `UIExtensionSite`. The problem is that the modules have to know which concrete UI element is behind the `UIExtensionSite`. For example, the `Infrastructure.Layout` registers a `MenuStrip` instance. All modules, which have to extend the `MenuStrip`, create `ToolStripMenuItem` elements and add them to the `UIExtensionSite`. If the `MenuStrip` in the `Infrastructure.Layout` module is replaced with another similar control, all the modules have to be modified.

The Test Suite prototype solves this problem with the `IUIElementCreationService`. The modules use this service to create the necessary UI elements. Thus, the modules do not have any dependency to the UI technology which is used in the `Infrastructure.Layout` module. If the UI technology is replaced in the layout module, just the `IUIElementCreationService` class has to be updated.

Command service

The Test Suite uses the built-in command system of CAB. It can handle all the needed requirements for most use cases. One of the exceptions is the handling of the edit functions cut, copy and paste. Here, the command should be routed to the active UI element only. In Windows-based applications the active UI element is the one which has the focus. Besides command routing, the command states are depending on the active UI element. If the active UI element does not support these commands or no elements are selected, the according commands have to be deactivated. In the Test Suite, the `EditManager` takes care for the special treatment of these commands.

Loosely coupled events

The Test Suite uses loosely coupled events for various reasons. The modules can use the CAB event broker to update the text shown in the application status bar. Moreover, the communication between the `Presenter` / `WorkItemController` and the `Presenter` / `Presenter` classes is done with the loosely coupled events. A special case is the transmission of the signal samples over the event broker. The `SignalVisualizer` registers for this event topic and displays the samples. The event handling has to be synchronized since the `SignalVisualizer` runs in the UI thread and the signal samples are created by other threads. CAB provides a simple solution for thread synchronization in the event handler as it is shown in Listing 14.

```
1  [EventSubscription(EventTopicNames.Sample,
2    Thread = ThreadOption.UserInterface)]
3  public void GotSample(object sender, SampleEventArgs sample)
4  {
5    ...
```

Listing 14: Thread synchronization in the event handler.

The synchronization with the WPF control of the `SignalVisualizer` does not work properly. During the application shutdown a `NullReferenceException` is thrown in the `System.Windows.Forms.Control.WaitForWaitHandle` method. The exception is only thrown if a virtual function generator is not turned off before the application is closed. Even then, the exception does not occur every time. This is a typical behavior for threading issues. Therefore, the prototype does not use the synchronization functionality of the CAB event broker. Instead, the view synchronizes the method call manually.

```
1  public void AddSample(object device, double amplitude,
2      double relativeTime)
3  {
4    if (Dispatcher.CheckAccess())
5    {
6      InnerAddSample(device, amplitude, relativeTime);
7    }
8    else
9    {
10     Dispatcher.Invoke(DispatcherPriority.Normal,
11       new AddSampleDelegate(InnerAddSample), device, amplitude,
12       elativeTime);
13   }
14 }
15
16 private void InnerAddSample(object device, double amplitude,
17     double relativeTime)
18 {
19   signalView.AddSample(device, relativeTime, amplitude);
20 }
```

Listing 15: Shows how thread synchronization can be done in WPF controls.

Listing 15 shows the manual thread synchronization. The code extract is part of the `VisualizerView` class. The `Dispatcher.CheckAccess` method verifies if the call needs to be synchronized. `Dispatcher.Invoke` calls the `InnerAddSample` method synchronized with the UI thread.

The communication between the function generator and the visualizer via loosely coupled events is just exemplarily. Typically, signal samples have to be processed in real time and not in the way it is done in the prototype. The event broker is to slow for processing signals with high frequencies. However, it is a good example why synchronization can be necessary in association with loosely coupled events.

7.6 Summary

The Test Suite implementation presented in this chapter handles all the requirements which are defined in chapter 2. Nevertheless, many requirements are already handled by the .NET Framework or the Composite UI Application Block. The CAB framework successfully reduced the effort to create the prototype application. However, the learning time for understanding the framework cannot be disregarded. CAB is very powerful through its abstract and flexible design but it is also highly complex. Microsoft has seen that the complexity of the framework is a problem for many users. Therefore, they introduced the Smart Client Software Factory. SCFS assists the software developer in common tasks and it is delivered with extended documentation. Nevertheless, it does not help much in learning the concepts of the key parts, the Object Builder and the `WorkItem` of CAB.

8 Final Remark

8.1 Conclusion

Most of the requirements are general ones that are not limited on specifying a Test Suite. They might also be true for other applications, even applications of other domains. Many requirements can be fulfilled with a modular application design. Component-oriented programming promotes the building of modular applications [Löwy05, p. 1]. The .NET Framework is based on the principles of a component-enabling technology and thus helps to achieve this goal. One important feature missing in the current .NET Framework 3.0 is a mechanism to wire loosely coupled modules together at runtime. Here the plug-in architecture comes into play. This kind of architecture is already widely used in desktop applications. Manufacturers and institutions started the development of plug-in frameworks because the implementation of a plug-in architecture is not a trivial task. These frameworks assist the software developer to build plug-in based applications.

The Composite UI Application Block is one of these frameworks. It is used in combination with the Smart Client Software Factory for the prototype application to show how the requirements can be fulfilled. The application framework saves a lot of time in implementing the prototype since it already fulfills a couple of the requirements. Additionally, it promotes good object-oriented design by wizard driven code generation. Nevertheless, one of the major drawbacks is the high learning effort for this framework. Microsoft tried to address this issue with the Smart Client Software Factory but still the effort is not negligible.

In my opinion the uses of plug-in architectures will grow in the future. This strategy is supported by general application requirements like testing of isolated components. Furthermore, it can reduce the complexity of applications and decrease the maintenance efforts. Some IDEs already support the plug-in based development like the Eclipse Plug-In Development Environment [17]. The high learning effort is one of the main drawbacks of plug-in architectures. This drawback is weakened by the fact that an IDE like Eclipse supports the development of plug-ins and thus simplifies the usage of plug-in frameworks.

[17] The official Website of Eclipse PDE: http://www.eclipse.org/pde

8.2 .NET Framework Application Extensibility

In the previous chapter it is mentioned that the .NET Framework 3.0 does not provide an implementation of a plug-in architecture. The successor of this framework version is going to deal with this issue. Microsoft introduces the `System.Addin` namespace in the .NET Framework 3.5. This namespace contains a plug-in framework with the main focus on application extensibility for third parties. Thus, the design goal behind this namespace is to enable dynamic composition of version resilient, isolatable components [GK07]. It contains a communication pipeline between the host and the add-in to enable compatibility because both parts can evolve independently [GK07a].

The `System.Addin` namespace shares some ideas with the Composite UI Application Block but it has its strength in another domain. The requirements, which are defined in chapter 2, do not have the need for version resilient, isolatable components. Therefore, the use of the `System.Addin` namespace is not practical to fulfill these requirements.

8.3 Open Issues

This diploma thesis concentrates on the evaluation of three plug-in frameworks. In a future work further suitable solutions could be investigated and compared with the plug-in frameworks of this thesis. Examples for such solutions could be as follows:

- Castle Project Windsor Container[18]
- StructuredMap[19]

Mono.Addin

Beside of the current available solutions some promising projects have been started at the time of writing this thesis. One of them is the Mono.AddIn framework[20]. It is a further development of the SharpDevelop add-in system. The most important improvement, which is planned for this framework, is the use of `Attributes` to define the add-in description. This simplifies the configuration and the refactoring because the descriptions are at the same place as the associated code. Even though, it should still be possible to describe the add-ins via the xml files. The Mono.AddIn framework is intended to create extensible applications. Therefore, it minimizes one of the main drawbacks of the SharpDevelop add-in system. The evolution of the Mono.AddIn framework is likely to be more stable than the add-in system of SharpDevelop.

[18] Official Website of the Windsor Container: http://www.castleproject.org/container
[19] Official Website of the StructuredMap: http://structuremap.sf.net
[20] Official Website of the Mono.Addins: http://www.mono-project.com/Mono.Addins

Acropolis

Another promising project is the Microsoft framework code name Acropolis[21]. It is a set of components and tools that simplifies the building and managing of modular client applications. Microsoft intends Acropolis to be the successor of SCSF / CAB and promises to create a migration path for existing SCSF / CAB applications [Block07]. The success factor of Acropolis could be the designer environment in which a software developer shall be able to define the entire application. If the designer is well thought out, it would really simplify the development of modular applications.

Customizing the Software Factory

Another idea for future work is to customize the Smart Client Software Factory to the needs of a Test Suite. This could also be a way for simplifying the development of an application in a specific domain. It would be interesting to see how much can be gained in relation to the efforts necessary in customizing the Software Factory.

[21] Official Webste of Acropolis: http://www.windowsclient.com/acropolis

Bibliography

Block07 Block, G. 2007. My Technobabble. Glenn Block - patterns & practices client program factories, patterns and models. Acropolis, the future of Smart Client. MSDN Blogs. http://blogs.msdn.com/gblock/archive/2007/06/06/acropolis-the-future-of-smart-client.aspx. Last visited on July 3, 2007.

BMMB97 Bosch, J., Molin, P., Mattsson, M., and Bengtsson, P.O. 1997. Object-Oriented Frameworks – Problems & Experiences. Submitted. http://citeseer.ist.psu.edu/bosch97objectoriented.html. Last visited on May 14, 2007.

Caprio05 Caprio, G. 2005. Design Patterns - Dependency Injection. MSDN Magazine, September 2005. Microsoft. http://msdn.microsoft.com/msdnmag/issues/05/09/DesignPatterns. Last visited on April 13, 2007.

Dhungana06 Dhungana, D. 2006. CAP .NET - Client Application Platform in .NET. Diploma thesis, Johannes Kepler University, Linz, Austria.

Fowler03 Fowler, M. 2003. Patterns of Enterprise Application Architecture. Addison Wesley.

Fowler04 Fowler, M. 2004. Module Assembly. IEEE Software. March/April 2004. http://www.martinfowler.com/ieeeSoftware/moduleAssembly.pdf. Last visited on February 27, 2007.

Fowler04a Fowler, M. 2004. Inversion of Control Containers and the Dependency Injection pattern. http://www.martinfowler.com/articles/injection.html. Last visited on March 19, 2007.

Fowler05 Martin, F. 2005. Inversion Of Control. http://www.martinfowler.com/bliki/InversionOfControl.html. Last visited on March 19, 2007.

FSF07 2007. FSF - Licenses. Various Licenses and Comments about Them. Last modified on April 09, 2007. Free Software Foundation. http://www.fsf.org/licensing/licenses. Last visited on April 10, 2007.

GB01 Van Gurp, J., and Bosch, J. 2001. Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines. Software Practice & Experience no 33(3), p. 277-300, March 2001. http://citeseer.ist.psu.edu/vangurp00design.html. Last visited on July 25, 2007.

GHJV95 Gamma, E., Helm, R., Johnson, R. Vlissides, J. 1995. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley.

GK07 Gudenkauf, J., and Kaplan, J. 2007. CLR Inside Out: .NET Application Extensibility. MSDN Magazine. February 2007. http://msdn.microsoft.com/msdnmag/issues/07/02/CLRInsideOut. Last visited on February 27, 2007.

GK07a Gudenkauf, J., and Kaplan, J. 2007. CLR Inside OUT: .NET Application Extensibility, Part 2. MSDN Magazine. March 2007. http://msdn.microsoft.com/msdnmag/issues/07/03/CLRInsideOut. Last visited on March 30, 2007.

Grunwald06 Grunwald, D. 2006. Building Applications with the SharpDevelop Core. The Code Project. http://www.codeproject.com/csharp/ICSharpCodeCore.asp. Last visited on April 10, 2007.

Grunwald06a Grunwald, D. 2006. Line Counter - Writing a SharpDevelop Add-In. The Code Project. http://www.codeproject.com/cs/samples/LineCounterSDAddIn.asp. Last visited on June 29, 2007.

HKS03 Holm, K., Krüger, M., and Spuida, B. 2003. Dissecting a C sharp Application. Inside SharpDevelop. Wrox Press. http://www.apress.com/free/content/Dissecting_A_CSharp_Application.pdf. Last visited on April 11, 2007.

Johnson92 Johnson, R.E. 1992. Documenting Frameworks with Patterns. Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications. Vancouver, Canada. http://citeseer.ist.psu.edu/johnson92documenting.html. Last visited on July 25, 2007.

Larman04 Larman, C. 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition. Addison Wesley.

Löwy05 Löwy, J. 2005. Programming .NET Components (2nd Edition). O'Reilly

MMS02 Mayer, J., Melzer, I., and Schweiggert, F. 2002. Lightweight Plug-In-Based Application Development. NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World. Pages 87 - 102. Springer-Verlag. http://citeseer.ist.psu.edu/mayer02lightweight.html. Last visited on March 14, 2007.

MSDN06 MSDN 2006. Overview of the Composite UI Application Block. Microsoft. http://msdn2.microsoft.com/en-us/library/aa546409.aspx. Last visited on March 30, 2007.

MSDN07 MSDN Library. .NET Framework Glossary. http://msdn2.microsoft.com/en-us/library/6c701b8w(VS.80).aspx. Last visited on March 17, 2007.

Nilsson06 Nilsson, J. 2006. Applying Domain-Driven Design and Patterns. Addison Wesley.

Noyes04 Noyes, B. 2004. ClickOnce - Deploy and Update Your Smart Client Projects Using a Central Server. MSDN Magazine May 2004. Microsoft. http://msdn.microsoft.com/msdnmag/issues/04/05/ClickOnce. Last visited on June 30, 2007.

OMG07 2007. Unified Modeling Language: Superstructure. Version 2.1.1. Object Management Group. http://www.omg.org/technology/documents/formal/uml.htm. Last visited on July 30, 2007.

Omicron07 OMICRON electronics GmbH. 2007. About Us. http://www.omicron.at/aboutus. Last visited on March 17, 2007.

Omicron07a The OMICRON Test Universe. 2007. http://www.omicron.at/products/secondary. Last visited on March 17, 2007.

SCSF06 2006. Online documentation of Smart Client Software Factory – June 2006. Microsoft.

Sosnoski05 Sosnoski, D. 2005. Classworking toolkit: Annotations vs. configuration files. IBM DeveloperWorks. http://www.ibm.com/developerworks/library/j-cwt08025.html. Last visited on August 7, 2007.

Spring07 2007. Spring .NET Reference Documentation. Version 1.1 RC 1. http://www.springframework.net/docs/1.1-RC1/reference/pdf/spring-net-reference.pdf. Last visited on August 13, 2007.

Sun02 2002. Core J2EE Pattern Catalog. Core J2EE Patterns - Service Locator. Sun Microsystems. http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html. Last visited on March 28, 2007.

Glossary

Application Domain

A boundary that the common language runtime establishes around objects created within the same application scope (that is, anywhere along the sequence of object activations beginning with the application entry point). Application domains help isolate objects created in one application from those created in other applications so that run-time behavior is predictable. Multiple application domains can exist in a single process [MSDN07].

Application Framework

Application Frameworks aim to provide a full range of functionality typically needed in an application. This functionality usually involves things like a GUI, documents, databases, etc [GB01].

Assembly

A collection of one or more files that are versioned and deployed as a unit. An assembly is the primary building block of a .NET Framework application. All managed types and resources are contained within an assembly and are marked either as accessible only within the assembly or as accessible from code in other assemblies. Assemblies also play a key role in security. The code access security system uses information about the assembly to determine the set of permissions that code in the assembly is granted [MSDN07].

Binary compatibility

A core principle of component-oriented programming. It allows exchanging compatible components (i.e., binary building blocks) without the need of recompiling and redeploying the clients [Löwy05].

Common Language Runtime

The engine at the core of managed code execution. The runtime supplies managed code with services such as cross-language integration, code access security, object lifetime management, and debugging and profiling support [MSDN07].

Component

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces [OMG07, p. 146].

Extensibility

A mechanism for manipulating host application objects or extending host functionality, sometimes referred to as automation. Generally made available via an object model published as part of a host's SDK [GK07].

Framework

A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes [GHJV95, p. 360].

Intermediate Language (IL)

A language used as the output of a number of high-level language compilers (C# compiler, VB .NET compiler, etc.) and as the input to a Just-In-Time (JIT) compiler. The common language runtime includes a JIT compiler for converting IL to native code [MSDN07].

Just-In-Time (JIT) compiler

In reference to the .NET framework it means the compilation that converts intermediate language (IL) into machine code at the point when the code is required at run time [MSDN07].

Private Assembly

An assembly that is available only to clients in the same directory structure as the assembly [MSDN07].

Remoting

The process of communication between different operating system processes, regardless of whether they are on the same computer. The .NET Framework remoting system is an architecture designed to simplify communication between objects living in different application domains, whether on the same computer or not, and between different contexts, whether in the same application domain or not [MSDN07].

Service

The term service is highly overloaded in computer science. In this thesis it has the same meaning as component. See Component in the glossary for more information.

List of Acronyms

API	Application Programming Interface
BCL	Base Class Library
CAB	Composite UI Application Block
CLR	Common Language Runtime
DI	Dependency Injection
ERP	Enterprise Resource Planning
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IL	Intermediate Language
IoC	Inversion of Control
JIT	Just-In-Time
OOD	Object-Oriented Design
PC	Personal Computer
RTF	Rich Text Format
SCSF	Smart Client Software Factory
SDK	Software Development Kit
UI	User Interface
UML	Unified Modeling Language
VB.NET	Visual Basic .NET
WPF	Windows Presentation Foundation
XML	Extensible Markup Language
XSD	XML Schema Definition

List of Figures

Figures

Tables

Listings

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Klaus, am 03.09.2007 Unterschrift