Cross-Platform, Parallel Development in Xamarin

Design and Implementation of the Platform for Building Extensible, Cross-platform GUI Applications Based on Plug-in Framework in Xamarin

Mariusz Waclawek

Computer Game Programming, bachelors level
2016

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Preface

This thesis concludes my three years of studies of Computer Game Programming at Luleå University of Technology. During last two months of education my thesis job was conducted at Data Ductus, an IT consulting company.
Abstract
This thesis describes research of plug-in frameworks, their application and benefits and also a way to develop cross-platform GUI applications in Xamarin with Xamarin.Forms. It describes also the limitations that mobile systems set on plug-in frameworks. The result of the thesis is a basic and extensible platform for building GUI applications based on the MVVM design pattern with modularity and extensibility of MEF 2.0.

Sammanfattning
Abbreviations and Terms

Abbreviations

DI – Dependency Injection
DIC – Dependency Injection Container
GUI – Graphical User Interface
IoC – Inversion of Control
MEF – Managed Extensible Framework
MVC – Model View Controller
MVP – Model View Presenter
MVVM – Model-View-ViewModel
PCL – Portable Class Library
OS – Operative System
OSGi – Open Services Gateway initiative
XAML – eXtensive Application Markup Language
XML – eXtensive Markup Language
WPF – Windows Presentation Foundation
Terms

Assembly – in C# it is a single project built into a library or executable.

Client plug-in – a plug-in that is used by another plug-in.

Export – MEFs definition of an extension.

Extension – a component of plug-in, dependency, implementation of single functionality.

Host plug-in – a plug-in that uses one or many plug-ins.

Import – MEFs declaration of an import.

Plug-in – a component package, assembly that extends the functionality of an application.

Slot – a term used in Plux.NET to declare import of an extension.
# Table of Contents

Preface ..................................................................................................................................................... i  
Abstract ................................................................................................................................................... ii  
Sammanfattnig ........................................................................................................................................ ii  
Abbreviations and Terms ....................................................................................................................... iii  
1. Introduction ........................................................................................................................................ 1  
   1.1 Background ................................................................................................................................... 1  
   1.2 Goals and Purpose ........................................................................................................................ 1  
   1.3 Limitations ..................................................................................................................................... 2  
   1.4 Data Ductus ................................................................................................................................... 2  
2. Theory ................................................................................................................................................. 3  
   2.1 Plug-in ........................................................................................................................................... 3  
   2.2 Plug-in framework ......................................................................................................................... 3  
   2.3 Interfaces ...................................................................................................................................... 4  
   2.4 DI - Dependency Injection ............................................................................................................. 4  
   2.5 DIC - Dependency Injection Containers ........................................................................................ 4  
3. Method ............................................................................................................................................... 5  
   3.1 Social, Ethical and Environmental Considerations ........................................................................ 6  
   3.2 Plug-in frameworks ....................................................................................................................... 6  
      3.2.1 Eclipse .................................................................................................................................... 6  
      3.2.2 CAP.NET/Plux.NET .................................................................................................................. 7  
      3.2.3 MEF – Managed Extensibility Framework ............................................................................. 7  
      3.2.4 MEF 2.0 .................................................................................................................................. 9  
   3.3 IoCs Performance ........................................................................................................................ 10  
   3.4 Cross-platform ............................................................................................................................ 11  
      3.4.1 Xamarin & Xamarin.Forms ................................................................................................... 11  
      3.4.2 XAML .................................................................................................................................... 11  
      3.4.3 MVVM – Model-View-ViewModel ....................................................................................... 12  
      3.4.4 Shared Code ......................................................................................................................... 13  
      3.4.5 Limitations ............................................................................................................................ 13  
         3.4.5.1 No Dynamic Libraries ................................................................................................... 13  
         3.4.5.2 PCL == Low .NET Profile ............................................................................................... 14  
   3.5 Platform ...................................................................................................................................... 14  
      3.5.1 Interfaces ............................................................................................................................. 14
3.5.2 Importing.............................................................................................................................. 14
  3.5.2.1 Exclusively ImportMany............................................................................................... 15
3.5.3 Metadata.............................................................................................................................. 15
3.5.4 Plug-in Versioning ................................................................................................................ 16
  3.5.4.1 OSGi Semantic Versioning............................................................................................ 16
3.5.5 Resolvers .............................................................................................................................. 17
3.5.6 Custom Export Attributes .................................................................................................... 18
3.6 Platform Basic Structure ............................................................................................................. 18
4. Results ............................................................................................................................................... 19
  4.1 IoCs Performance Results ........................................................................................................... 19
  4.2 Platform overview ....................................................................................................................... 22
  4.2 Available Interfaces ..................................................................................................................... 23
  4.3 UI Composition ........................................................................................................................... 24
  4.4 Navigation ................................................................................................................................... 25
  4.5 Metadata & Resolvers ................................................................................................................. 25
  4.6 Development Pipeline ................................................................................................................. 26
  4.7 Maintaining the Platform ............................................................................................................ 26
    4.7.1 Core ...................................................................................................................................... 26
    4.7.2 Plug-ins................................................................................................................................. 26
5. Discussion .......................................................................................................................................... 27
  5.1 Future Work ................................................................................................................................. 29
6. References ........................................................................................................................................... 30
1. Introduction

This Section presents a brief description of background and problems. It also specifies the limitations for the scope of the project and presents a company, DataDuctus, at which this thesis was carried out.

1.1 Background

Mobile applications today are often developed with an Interface First[1, 2] methodology. Those have a set scope of features and are built as one unified application which makes it hard to implement new features without drastically changing the interface and the base code.

To keep up with the growth of Internet of Things[3] it is important to design the applications in a way that would allow developers to easily expand them with support for those new devices. Also the interface has to be responsive and adapting to each device format.

In a production where multiple developers work on the same application, they are usually strongly dependent on one another. If the application is not modular or the code is not well separated, they will often make changes in same code which can lead to merging conflicts. Overall, they must have wide knowledge of the existing code someone else wrote when developing new features, even more so when they must introduce changes.

1.2 Goals and Purpose

The goals of the project were to research a way that would allow developers to easily extend applications and to let them work more independently in parallel without the need to understand or change one another’s code in cross-platform environment. As a result this would lead to reduced time and cost of development.

The purpose of the project was to get an insight into cross-platform development and plug-in framework architecture.
1.3 Limitations

This project presents a general solution for any kind of application, but because of the wide scope of research, only the base functionality for developing GUI-based applications was implemented. Plug-ins and plug-in frameworks were analyzed in general but because of platform limitations it was focused more on static composition.

1.4 Data Ductus

Data Ductus is a multinational IT consulting company. It was founded in 1989 in Sweden and today has 170 employees. It delivers technologically advanced solutions to meet client specific needs. They offer services in a large number of industries and solution areas, and they are known for their ability to quickly adapt and respond to changing business requirements.
2. Theory

This Section provides brief introduction into the subjects of plug-in frameworks and dependency injection which is essential for understanding this thesis.

2.1 Plug-in

Plug-ins are components that introduce new features or extend existing ones. Some of the plug-in examples based on the application type are:

- Web-browser: virus scanners, search engines, Adobe Flash Player, JAVA [4].
- IDE: plug-ins that e.g. add UML generation functionality [5].
- Media Player: skins, support for new audio types [6].

2.2 Plug-in framework

A plug-in framework is a software architecture that allows developers to write extensible, modular and customizable applications. In its early days it was used mostly to allow third parties to incorporate features to an application without requiring access to the source code. Eventually it has advanced into a methodology for developing applications [7, 8].

Benefits of plug-in frameworks [9]:

- Third parties can easily extend an application with new features without access to source code and therefore do not require intervention of the application developer to recompile and redistribute the application.
- Bug fixes and updates can be performed on plug-ins rather than the whole application.
- Since plug-ins are implemented against interfaces it is easier to isolate and fix bugs.
- Easier and faster testing because unit-tests can be performed against interfaces.
- Easier to publish different versions/editions of the application.
- Consumers can customize an application in a way that suits their workflow.
- Allows consumers to disable unwanted features to reduce memory footprint and improve performance.
2.3 Interfaces
An interface in object-oriented languages is an abstract type containing only the declarations of methods. An interface can’t be instantiated; it has to be inherited by either another interface or a class, where the second has to implement it as well [10, 11, 12].

Depending on the use case, an interface can contain many, one or no members. An interface that contains no members, is often used as a so-called marker interface which helps find and filter out objects of that interface in an application [13].

Interfaces provide loose dependency coupling between objects. A class using an instance of an object implementing an interface no longer depends on the specific class implementation, but an abstraction and therefore can use any object implementing that interface [12].

Interfaces and their properties are well utilized in the dependency injection design pattern [12].

2.4 DI - Dependency Injection
A very common design pattern in plug-in frameworks is dependency injection, it offers better separation of the code without hard dependencies. A dependency is an object another class needs in order to function. Dependency injection means that the dependency is provided to the class, removing responsibility of instantiating dependencies by that class [14, 15].

The Dependency Inversion Principle says that code should depend on abstractions. This means that dependencies should be provided by an external code through interfaces creating loose couplings. In this way different implementations of the same interface can be provided to the object. Implementations can be updated without altering the depending class [14, 15].

2.5 DIC - Dependency Injection Containers
Their main functions are to register classes together with their dependencies, interfaces, and to construct objects of a class by resolving their registered dependencies. The registration itself is done manually either in code or through external files, such as XML. The biggest benefit of those containers is that dependencies are resolved recursively [14, 15].
3. Method

Before the implementation process begins, it is important to research and determine what tools, solutions and environments will be used as well as their benefits and limitations.

This project is heavily based on the plug-in framework and therefore it was important to first research how one is built, how it functions and what the benefits are. This required finding the information, reading available papers as well as analyzing different implementations and existing frameworks. An approach that fitted our needs was chosen based on the research results described in Section 4.1.

Xamarin was used in this project as base environment to simplify the problem of cross-platform development as it offers support for wide range of platforms, ability to implement application in one language and a cross-platform GUI toolkit called Xamarin.Forms. Because of lack of previous experience with Xamarin the next step was to research it is capabilities of the cross-platform development, to understand how the code was shared between platforms, how the user interface was structured and how it can be implemented for different platforms. This was approached by reading the available documentation and books. At that stage it was also crucial to detect any limitations caused by either Xamarin and/or OS platforms.

Design and implementation of the platform begun after familiarizing with Xamarin. It was crucial to find a good way to integrate plug-in framework with Xamarin that wouldn’t introduce limitations for the developers. This is why it was important to gain good understanding and knowledge about both of those elements.

Finally maintenance of the platform and future development was considered. This required a research to compose a set of rules and a way to version the platform elements that would make it easy for developers to expand the platform.
3.1 Social, Ethical and Environmental Considerations

This project does not have any effect on social or ethical values as it is a general platform for developing applications and was done without the use of any personal information and it is only up to the other developers what kind of applications they will create with it. Based on the purpose of the project, theoretically using the platform should result in lower power consumption per project because of the parallel, more individual and independent development.

3.2 Plug-in frameworks

Plug-in framework was selected for extensibility and parallel development for this project. During research, multiple plug-in frameworks were evaluated that had different approaches which offered different extents of freedom. Some let developers extend applications via a specific API while others allowed a lot of freedom to write different kinds of extensions. Research was focused on the frameworks that offered the most freedom; where everything was a plug-in.

3.2.1 Eclipse

Eclipse is an extensible platform for building IDEs which has a thin core that offers basic functionality which can be extended via plug-ins. Main idea of its plug-in framework is that everything is a plug-in [16].

Plug-ins can contain both implementation for different extension points and own extension points, meaning that a plug-ins can extend a plug-ins. They are written in JAVA and information about each of them is stored in two kinds of XML files. The plug-in XML file contains extension points and their definitions while OSGi manifest XML contains lower-level information like plug-in id, name, version and dependencies like required contributing plug-ins [16].

XML files are automatically generated with the Eclipse dev toolkit and can be edited using the editors. All plug-ins are registered at application start but not loaded until a function of the plug-in is actually called [16].
3.2.2 CAP.NET/Plux.NET

Plux.Net is based on the same idea as Eclipse, everything is a plug-in, but Eclipse is generally a platform for developing custom IDEs and Plux.Net is a platform for building and extending whole applications [17, 18].

Just like Eclipse, this framework has similar approach where plug-ins can extend other plug-ins but instead of XML files for defining dependencies it uses custom Slot- and ExtensionAttributes. Slots are interfaces with unique identifiers which have to be implemented by Extensions while Extensions are implementations of Slots and must use same unique identifier. Both Slots and Extensions are often accompanied with Metadata attributes which are used to mark Slots and Extension with order, type, version and many more [17, 18].

The framework has its own discovery and registration technique that uses Extension-Attributes for finding and resolving dependencies. To access these attributes without loading them into memory the whole assembly, i.e. plug-in, only the reflection of it is loaded. Then valid plug-ins are added to the plug-in registry which is a data structure that represents relations between Slots and Extensions as well as their inner dependencies [17, 18].

3.2.3 MEF – Managed Extensibility Framework

MEF is a part of .NET4.0 and it is Microsoft’s polished and extended implementation of the framework presented in the Plux.NET paper.

MEF uses attributes for defining Imports and Exports [19] and just as Plux.NET it uses metadata to provide extra information of the extension. Users can implement derivatives of Export and Metadata attributes [20]. This information can be retrieved before an object is instantiated and it is often used to tag extensions of same interface for filtering or to provide extra information like versioning or ordering.

Discovery and registration can be performed on both static and dynamic assemblies but MEF was primarily designed with dynamic assemblies in mind. It supports lazy instantiation and creation of new instances which will create new instance when it is called for the first
time, just like in Eclipse [20]. MEF also provides three ways to import extensions: as a
collection, single object or optional single object [20].

While MEF is used for dependency injection and resolution, it is not considered a true IoC
container. This is mostly because it was designed to discover unknown dependencies from
assemblies while IoC containers rely on registration of each dependency.
3.2.4 MEF 2.0

MEF 2.0 is a part of .NET 4.5 and it is portable, lightweight and faster version of MEF which has been optimized for static compositions. Since it is portable it can be used inside PCL for cross-platform development. [21]

First MEF provided functionality to either declare an Extension to be instantiated either as single or global instance. Single instance was unique for each object while global instance was shared between different objects. Not only the Extension was limited to one type of instantiation but it also was instantiated only once. MEF 2.0 introduced ExportFactories that provided retrieval of new unique instances of an extension even if the instance is declared as global. This means that an Extension could be instantiated as global and single at the same time, but the main feature of ExportFactory was the ability to create new instances of an Extension at runtime [22].
3.3 IoC Performance

There are many IoC containers to choose from that differ in the amount of functionality and speed. The platform in this project was meant for cross-platform development which included mobile devices. It was important to make sure that the selected framework was fast and lightweight.

For that reason a benchmark written by Daniel Palame[23] was used to test the performance of many IoC containers in different scenarios. Every container was initialized with a couple of interfaces and their corresponding implementations. The types were registered with different lifetimes to support the various benchmarks.

Every interface was resolved 500,000 times during the benchmark. Time was measured in milliseconds. Benchmark scenarios were divided into two categories, basic and advanced based on their complexity. The results of the benchmark are presented and analyzed in Section 4.1.

Benchmarked scenarios were:

Basic

- Singleton: Objects with is singleton lifetime are resolved
- Transient: Objects with is transient lifetime are resolved
- Combined: Objects with two dependencies (singleton and transient lifetime) are resolved
- Complex: Objects with several nested dependencies are resolved

Advanced

- Property: Objects which require property injection are resolved
- Generics: Objects with a generic dependency are resolved
- IEnumerable: Several objects that implement the same interface are resolved
- Conditional: Objects with a conditional dependency are resolved
- Child Container: Objects are resolved through a child container
- Interception With Proxy: Objects with a dynamically generated wrapper are resolved
3.4 Cross-platform
Native applications are developed in a language specific for the OS running on the device e.g. Android applications are written in JAVA while iOS in objective C or Swift. Developing a cross-platform application the usual way would require writing completely different implementations for each platform. Therefore to provide easier cross-platform support different technologies, design patterns and methodologies were researched.

3.4.1 Xamarin & Xamarin.Forms
Xamarin is a platform for developing cross-platform applications recently acquired by Microsoft. With it developers can write native applications using one language, C#, and with Xamarin.Forms they can create one UI while preserving the native look of the different platforms.

3.4.2 XAML
Extensible Application Markup Language or XAML in short is an XML-based markup language developed by Microsoft and a part of the Windows Presentation Foundation, WPF. XAML is used for defining the visual side of the user interface for the Windows- and Web browser-based applications [24] as well as applications based on the MVVM design pattern [25]. With Xamarin it was possible to implement views both in code and XAML.
3.4.3 MVVM – Model-View-ViewModel

In complex applications it is a good idea to separate the data, view and logic because this way, changes can be made to the logic and data implementation without affecting the view. It is also a lot easier to write unit-tests without the involvement of view. If a view was containing a logic as well, then the tests would have to be performed against the user interface [26].

There are many design patterns that help separate the user interface, business logic and back-end logic, like MVC or variation of it, MVP. Another such pattern is MVVM (see Figure 1) developed by Microsoft architects [25]. It is a variation of Presentation Model proposed by Martin Fowler [27, 28].

MVVM was specifically tailored for WPF and Silverlight platforms to simplify event-driven programming. In fact MVVM is not using events for GUI interaction but commands and bindings. It depends on the use of data templates, commands and data bindings’ through XAML or optionally through the code [25, 29].

MVVM relations [25]:

- View knows nothing of the Model and vice-versa.
- ViewModel is unaware of the View while View uses the ViewModel through the bindings, commands and data templates. ViewModel has to be passed to the View. ViewModel implements and exposes commands and properties which can be used by the View through the bindings.
- Model knows nothing of ViewModel while ViewModel uses Model. Model has to be passed to the ViewModel.

![Figure 1 – Graphical representation of MVVM](image-url)
Xamarin integrates well with XAML and thanks to the Microsoft partnership they continue to improve the development by extending integration and releasing new tools. It was possible to apply other design patterns in Xamarin but MVVM was very natural and straightforward to adapt.

**3.4.4 Shared Code**

A key component of building cross-platform applications is the ability to share code across various platform-specific projects.

In Xamarin it is possible to share the cross-platform code by using shared projects and portable class libraries, PCL [30]. Shared projects are good for packaging native implementations of all platforms of in one project with use of OS defines [31]. PCL on the other hand can only contain pure cross-platform code with one implementation for all platforms [32].

Shared projects are the only way for writing code that requires more advanced and native functionality that is not available through Xamarin [31]. PCLs are good for using functions available through all platforms and GUI programming thanks to the Xamarin.Forms [32].

To handle the extensions in elegant manner PCLs were used as much as possible with exception for native functionality which had to be contained in shared projects.

**3.4.5 Limitations**

Support of mobile devices and other platforms has applied some constraints. Because of these, some of the basic features and benefits of the plug-in framework were lost and many building blocks and ideas fell. [33, 34]

**3.4.5.1 No Dynamic Libraries**

One of these was lack of support for dynamic libraries in iOS [33]. Remoting for communication between AppDomains and use of multiple AppDomains for loading plug-ins into secondary domain to avoid crashing the application because of faulty plug-in was no longer possible. Users were unable to customize the functionality of the application because it was impossible to load or unload plug-ins at runtime. Libraries had to be referenced in the main application since it was impossible to load these dynamically. Also there are no static
libraries in C# but Xamarin automatically merges all referenced libraries into the application when it is built.

To add, remove or edit plug-ins, whole application had to be rebuilt and deployed for redownload. This eliminated possibility of releasing and deploying per plug-in updates. Third party plug-in development was possible only if the project was open source. On the other hand there was no need to worry that changes to the core would force all plug-ins to be recompiled since they were all referenced and the whole application would be built anyway.

3.4.5.2 PCL == Low .NET Profile

With Xamarin it was possible to develop applications in one language, C#, as well as utilize many of the .NET features but the more platforms one wanted to support the smaller the shared .NET core would get [32]. At the same time any referenced libraries in the PCL had to be cross-platform as well, like MEF 2.0.

3.5 Platform

The platform was designed and implemented around well tested and developed platforms, tools and methodologies. It used MEF 2.0 for extensibility and Xamarin for cross-platform development. It was designed with MVVM design pattern in mind since Xamarin.Forms integrates well with it and provided good support for XAML. Users are able to create views both in XAML and in code, giving them complete freedom over the implementation.

3.5.1 Interfaces

Each interface had its own custom Export attribute, metadata, enum Category and Resolver. Developers might want to introduce new rules and interfaces to the platform. To avoid conflicts each interface and its related code was stored in separate file.

3.5.2 Importing

At first the idea was to use Import and ImportMany whenever needed in the code but some problems have arisen. How to import with regard for version or category? How can the same instance of imported parts be shared between extensions?

Parts could be imported by either using Import or ImportMany. Most obvious difference between both methods was that ImportMany could import a list of parts of the same contract while Import could import just one.
Another difference was that Import required that at least one Export of importing contract existed for each Import, if not specified otherwise with an AllowDefault attribute. ImportMany allowed none, one or many imports into one container.

3.5.2.1 Exclusively ImportMany

To enforce flexibility and extensibility of parts, importing was made exclusively with ImportMany [20] as it offered most extensive import capabilities. There might have been only one implementation of specific type at one time, but it should be possible to implement new versions that would seamlessly fall into the place without the need for changes in the importing parts.

3.5.3 Metadata

MEF does the discovery and injection automatically through analysis of export and import attributes. Export and Import attributes are connected by contract that can be represented either by type or a name. The exporting class has to use the export attribute and specify the contract. A class that requires dependency has to use the import attribute and specify the contract for every dependency.

Because of that automation, MEF requires a specific method for handling multiple implementations of same interface. If an object required just one of those implementations there was no other way but to import all the implementations. Fortuntely lazy instantiation could be used, a feature that instantiated only the objects that were used, e.g. by calling an function in them.

The last problem was making a difference between all implementations and to solve that metadata was used for labeling each implementation. Metadata can be attached to a class by declaring an Metadata attribute above it. This metadata can be imported together with implementations and it is also accessible before instantiating the objects.

Every interface received its own metadata that contained Major and Minor versions as well as a unique enum Category for labeling implementations. This allowed developers to filter interfaces by category and version to retrieve one or many extensions.
3.5.4 Plug-in Versioning

Some larger applications today are released with various levels of functionality and often use names like Basic, Pro and Ultimate. There are also early access programs which provide access to unfinished versions of applications that are usually frequently updated.

The concept of versioning of plug-ins was introduced to support those ideas. It allows developers to produce multiple versions of same plug-in which also offers some flexibility when debugging. Having multiple versions available makes it easy to compare and switch the plug-ins. It is also easy to deploy same applications with different features just by replacing the versions or changing the set of included plug-ins.

Versioning was based on the OSGi semantic convention, a standard presented by the OSGi Alliance. It incorporates meaning for each version component [35].

3.5.4.1 OSGi Semantic Versioning

Standard OSGi version is made of four numbers separated with dots, f. ex. 1.2.3441.213414. Those numbers are named Major, Minor, Micro and Qualifier and are described in table below(Figure 2).

<table>
<thead>
<tr>
<th>Component</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related to backward compatibility</td>
<td></td>
</tr>
<tr>
<td>Major</td>
<td>Difference in versions == incompatibility</td>
</tr>
<tr>
<td>Minor</td>
<td>Consumer is compatible with provider that use higher or equal</td>
</tr>
<tr>
<td>Not related to backward compatibility</td>
<td></td>
</tr>
<tr>
<td>Micro</td>
<td>Build number used for bug tracking.</td>
</tr>
<tr>
<td>Qualifier</td>
<td>Ussuaaly indicates identity of the build like timesamp.</td>
</tr>
</tbody>
</table>

*Figure 2 – OSGi convention description*

Our versioning was represented with Major and Minor values inside the metadata. This way developers have more control over versions they want to import and support. An update of the Major version would indicate significant changes that would result in incompatibility with previous versions while an update of the Minor version would mean change of the implementation that would not break the backward compatibility.
3.5.5 Resolvers

With the current platform it was possible to import one or many client plug-ins into a host plug-in but it was cumbersome to add imports everywhere in the code. It was infected with Import attributes, properties and local variables that would handle them.

Besides the clutter introduced by the imports there was also a secondary effect, less noticeable which was the extra memory usage caused by the duplicated imports all over the code.

Another nuisance was the constructor importing rule. Dependencies simply had to be resolved recursively from the bottom to the top. Whenever a host plug-in imported dependencies in the constructor it would require those dependencies to be resolved first in their constructors, if they had any imports. This means that it was impossible to mix one plug-in with imports in constructor with another one with imports at methods or other members.

MEF allows importing of other types than interfaces like classes, properties and methods. It might not always be desirable to import dependencies in the constructor and since MEF allows imports of data members, it was important not to reduce or cripple that functionality with our platform.

To parse and retrieve the correct plug-in from the list of different implementations by using the categories and versions one would have to write the same code anywhere a dependency had to be imported.

For all those reasons, the code and importing had to be centralized. Each interface received its own manager called resolver that would import the plug-ins and offer parsing.

Resolvers as singletons were not instantiated through MEF and therefore their imports were not satisfied automatically. Luckily MEF provides a function that lets us perform post construction resolution.
Another reason to use managers was that one might want to retrieve same or new instance of an implementation for different extensions. E.g. same ViewModel instance could be resolved for three different Views. This way user could perform changes via one View and the others would get updated automatically.

MEF by default uses Shared (singleton) model for its creation policy[36] meaning all imports by default are instantiated only once no matter how many times they are retrieved and therefore can be shared between different objects. Each Resolver had regular ImportMany for retrieval of shared instances and also ExportFactories for retrieval of new instances.

3.5.6 Custom Export Attributes
To make the platform more user-friendly and to reduce the lines of code required for exporting parts, custom export attributes containing metadata were implemented.

Each interface was accompanied by an ExportAttribute and to make sure these were used properly, their default constructor enforced input of all crucial metadata.

3.6 Platform Basic Structure
Basic structure of the platform (Figure 2, 3) and relationships between modules is presented below.

![Figure 2 – Basic Structure of the platform](image)

![Figure 3 – Relationships between different modules](image)
4. Results

This Section presents the results of the tests of the different containers and the implementation of the platform.

4.1 IoCs Performance Results

Containers used in the benchmark were last updated in April 2015 so a few of them were a little outdated during the test.

Not all of the containers were cross-platform, especially when it comes to the mobile devices. For example some containers used System.Reflection.Emit which is related to the dynamic code generation at runtime which iOS do not support.

One can observe that DryIoc, Simple Injector, Hiro and HaveBox and MEF 2.0 performed well in both basic(Figure 4) and advanced(Figure 5) scenarios. While MEF and Ninject were the slowest of them all(Figure 6). The Figure with advanced tests for slow containers was not included because the only information it provided was that Ninject was the slowest of the slow containers.

DryIoc is somewhat cross-platform but is missing support for iOS and Android. Hiro is not cross-platform, has very limited features and no documentation. Simple Injector with some extra work can support iOS and Android; it is well documented and has many features. MEF 2.0 is entirely cross-platform and can be used in PCL and it is developed by Microsoft. Compared to other containers MEF uses discovery of unknown parts at compile time instead of manual registration of known parts.

Besides MEF 2.0 being quite fast and lightweight it was chosen for its discoverability, cross-platform support and because it was developed and well tested by Microsoft.
Figure 4 – Results of basic test of fast containers

Figure 5 – Results of advanced test of fast containers
Figure 6 – Results of basic test of slow containers
4.2 Platform overview

Based on the previously mentioned structure in Section 3.5 the implemented platform consists of thin core containing crucial minimal functionality and set of rules. It was adapted with Xamarin.Forms for cross-platform development of GUI applications. In its current state it is possible to easily write applications based on MVVM pattern and developers can choose either XAML or code for defining views. Figure 7 represents the platform, its navigation and GUI modules for cross-platform parallel development.

![Platform structure with modules for developing MVVM applications](image1)

**Figure 7 – Platform structure with modules for developing MVVM applications**

![A prototype design for an Home Automation App that uses the platform](image2)

**Figure 8 – A prototype design for an Home Automation App that uses the platform**

Presented on the Figure 8 above is the Home Automation App in its early stage with modular GUI, but no networking. It reveals in more detail how each device is managed as a plug-in and added to registry. One can also observe the MVVM composition for a device and how they are later imported into the Main ViewModel.
4.2 Available Interfaces

The platform contains three interfaces: IView, IViewComponent and IViewModel used for MVVM composition and one generic INavigationService<Tout,Tin> for implementing own navigation.

IViewModel as the name suggest is meant for implementing ViewModels and contains no methods. IViews are used to implement Views or so called Pages in Xamarin. They require ViewModels to be set as their BindingContext.

Views contain bindings declared as string names which they use to scan the BindingContext for data members therefore IViewModel interface does not require any special members.

IViewComponent is used for implementing the UI elements like buttons, labels or layouts, everything a View consist of.

To compose the user interfaces in simple manner it is recommended that IView and IViewComponent inherit from Xamarin.Forms classes. If for any reasons developers might not be able to do that, the two interfaces provide helper members for composition.

There is no IModel interface as Models can be very different in the implementation and are closely tied to the ViewModels.
4.3 UI Composition

Views are made of ViewComponents but a View and ViewComponent can also contain Views. Views are composited with DataTemplates available in XAML and Xamarin.Forms. It is possible to represent same data with same functionality in different Views by using same ViewModels. Resolvers can retrieve same instance of ViewModel anywhere in the code so the multiple Views can access the same data. Change made in one view will update the other one via ViewModel. If needed resolvers can retrieve the copies of original instances as well.

On Figure 9 below one can observe the more complex MVVM composition that is possible with the platform. On the left side of the tree one can observe two Views sharing the same ViewModel. At the bottom, the main View imports both View and ViewComponent. On the right side the ViewComponent imports another View. Recurrences of same resolvers are only used to easily demonstrate the modularity. In fact there is only one instance of any resolver where all implementations of respective interface are imported.

Figure 9 – MVVM UI Composition
4.4 Navigation

The platform provides navigation of views accessible from anywhere in the project. It implements the generic INavigationService<TOut,TIn> interface and can be used to implement a custom navigation plug-in. Navigation has no resolver as only one should be used at any given time.

4.5 Metadata & Resolvers

Each interface uses metadata for versioning and categorizing implementations. It is also accompanied with its own singleton Resolver that imports those implementations. Developers can retrieve any dependencies they might need from Resolvers and therefore it is not necessary to import dependencies manually inside the plug-ins.

Resolvers are able to return a single dependency or a list of them. Developers can also use resolvers to request either a new or the same instance. To retrieve dependencies the user must provide a Category of the implementation and optionally Major and Minor versions. The retrieved object is a single pair or a list of pairs consisting of metadata and instances.

*Figure 10* below presents the interfaces and their relations to different elements that surround them.

*Figure 10 – MVVM modules used in the platform, with relationships represented by arrows*
4.6 Development Pipeline

Programmers can use the core to develop and test plug-ins for the application in parallel, separate environments. When a plug-in is tested and ready, it is committed to the main repository together with new interfaces. After the core is updated with those interfaces, it can be accessed and downloaded by other developers without worrying that it might break their application. A new plug-in must be referenced in the main application and added to the MEF discovery container which in no way affects the previous functionality and already existing plug-ins.

4.7 Maintaining the Platform

4.7.1 Core

Developers can add new functionality to the platform by providing new interfaces supported by their resolvers, custom attributes and metadata. Changing the existing interfaces is not recommended as it might break the existing implementations while changing categories is only permitted when adding a new one. Instead of making changes to the interface a new version should be provided, advisably with an adapter, to support previous ones if possible. All previous plug-ins will still work and the new version will be available for anyone who might like to use it.

4.7.2 Plug-ins

For similar reasons as for interfaces it is not recommended to update the existing plug-ins. Updating plug-ins in most cases is safe since they have to fulfill interface definition but there is still a slight possibility that such change might break the application. Developers have full control over the inclusion of plug-ins in the application. When new version of plug-in has been tested and it works well, the host plug-in can simply change the importing version criteria for resolving and then developer has to unload the old version from the application. If it turns out that the new plug-in is faulty then it should be simply switched with previous version in the core application.
5. Discussion

Because of the continuous growth of the Internet of Things, demand for cross-platform developer tools is increasing. It is essential to develop such applications faster and as much shared code between platforms as possible.

I think the amount of available IoCs today proves the appeal and need of such containers as it helps them write reusable, cleaner, modular code. The range in functionality, speed and cross-platform support gives a lot variety that can fit many different use cases.

Based on the results from the benchmark[23] I didn’t choose the fastest IoC possible but the one that offers speed and features that I think were more fitted for this platform. I’ve chosen the MEF 2.0 over other IoC containers because of its discovery, simplicity of adding new assemblies to the application and the extensive importing functionalities.

The standard MEF is slow but it supports dynamic libraries. It is better suited for more powerful machines than mobile phones. If I were to develop the platform only for Windows I would probably use the first MEF for the extra benefits that come when we can use dynamic libraries.

Because of the short time and the large scope of the project I was only able to design and implement the base of the platform. It provides a framework that allows developers to easily write applications with very modular user interface thanks to the provided interfaces. Resolvers reduce the boiler plate code, simplify the dependency injection which makes the code cleaner and provide helpful functions to write extensions. It uses cross-platform version of MEF for extensibility and Xamarin.Forms for cross-platform user interface which makes it good environment for developing cross-platform applications.

This platform is based on plug-in framework and dependency injection that provides loose coupling between objects. The only thing developers have to include when developing their plug-ins are the interfaces (with resolvers). This means that there are no connections between plug-ins, yet they can use and extend one another. Together those things allow parallel development where many developers can work independently without the need to know or change the source code of other developers.
At some point in my research of the plug-in frameworks, I have realized that we can’t use the dynamic libraries since some platforms didn’t support it. I’ve learned that it is important to early detect any limitations that will influence the development, to save a lot of time and avoid frustrations. Thankfully there were many cross-platform solutions available to my rescue. Many benefits of the plug-in framework were gone but today plug-in frameworks are not only about extending applications by third parties, but better project management, parallelism, testability, ability to release different alterations of the application and early access versions (EAP) which was still possible. I have to mention the testing possibilities. Since plug-ins are based on interfaces it is much easier to write unit-tests as we only need one per interface.

Developing an platform for building cross-platform applications that is simple and general for multipurpose use, yet offers many useful features, requires great knowledge of tools, platforms, technologies as well as their limitations and anything else that might be used as fundamental pieces of the platform. I think it is important to understand well the available tools at your disposal; you need to know your options because you are building something that others will use as a base for their applications. It should simplify the process of the development and reduce the boiler plate code without presenting new limitations for the developers.

The presented solution does not incorporate any order number for plug-ins. It could be used for sorting the displayed elements in the GUI or plug-in execution order. We could solve that easily by adding order number in the metadata of View and ViewComponent. As for the GUI this might be ok but in my opinion execution order or any order for that matter should not be decided by the plug-in but the application. For execution order plug-ins should provide unique values, like names or just categories, that the application can recognize and based on those it would decide on the order. There is still a problem of plug-ins that by mistake try to use same values but then as with any troublesome plug-in, developers of the consuming part can simply exclude the imported, buggy plug-in from the application until it is fixed.
There isn’t really anything else I would like to change other than extend the platform with networking. On the other hand I would like to explore other ways to separate dependencies completely by using extensively messages. We could have just one interface that has basic common functions. It would use messages for function calls and requesting dependencies when constructing objects. Another interesting approach for separating dependencies would be to incorporate the scripting language for writing extensions which in turn would require the application to be entirely open-source.

Overall I’m pleased with the project and the results. I didn’t provide the complete platform with all the features I wanted to but I still have learned a lot about design, software architecture, different design patterns, platforms and tools which will help me to continue its future development.

5.1 Future Work

Altrought the presented solution can be used for developing modular GUI applications it is still missing the networking module that was aborted in the middle of the design process because of lack of time.

Finishing the networking module would be the next step in the development of the platform. Such module would provide easy to set up connection clients, pluggable authentication, monitoring and management of power and connection type as well as its strength. The goal would be to reduce the work required by the developers to set up the connection and authentication and to provide a controlled, uniform system that informs of the usability of the connection.
6. References

Books:
[38] Mark Seemann, *Dependency Injection in .NET*, October 2011, 1st Edition
ISBN-10: 1935182501

ISBN-10/ASIN: B00NXJ8DK

Publications:
http://www.ssw.jku.at/Research/Papers/Wolfinger06/Wolfinger06.pdf
Updated 2016-05-22

Whitepapers:
[34] OSGi Alliance, *Semantic Versioning*, Revision 1.0, May 6, 2010, Technical Whitepaper
Updated 2016-05-22

Online:
[1] Interface First. Implementation Later – Perspx by Alex Rozanski
Updated 2016-06-01

[2] UI-First Software Development – Coding Horror
https://blog.codinghorror.com/ui-first-software-development/
Updated 2016-06-01

Updated 2016-06-01
Updated 2016-05-22

Updated 2016-05-22

Updated 2016-05-22

[7] MSDN – Creating a Plug-In Framework
Updated 2016-05-22

[8] Plug-in Architectures lecture by Jeff Stylos
https://www.cs.cmu.edu/~bam/uicourse/830fall04/JeffStylosPlug-inArchitectures.ppt
Updated 2016-05-22

Updated 2016-05-22

[10] Oracle Java Documentation – What is an Interface?
http://docs.oracle.com/javase/tutorial/java/concepts/interface.html
Updated 2016-05-22

http://docs.oracle.com/javase/tutorial/java/landl/createinterface.html
Updated 2016-05-22
[12] Software interfaces in object-oriented languages
Updated 2016-05-22

https://en.wikipedia.org/wiki/Marker_interface_pattern
Updated 2016-05-22

[14] Dependency Injection in .NET by Mark Seemann – Manning
http://martinfowler.com/articles/injection.html
Updated 2016-05-22

[15] Lecture at TechED 2013: Understanding Dependency Injection... by Miguel A. Castro
https://www.youtube.com/watch?v=RVpADaFIlRw
Updated 2016-05-22

[16] Notes on Eclipse Plug-in Architecture
https://eclipse.org/articles/article-architecture/plugin_architecture.html
Updated 2016-05-22

http://ase.jku.at/plux/
Updated 2016-05-22

[19] Managed Extensibility Framework, MEF.
Updated 2016-05-22

[20] MSDN – Attributed Programming Model Overview (MEF)
Updated 2016-05-22

[21] MEF Community Site
http://mef.codeplex.com
Updated 2016-05-22
Updated 2016-05-22

[23] Daniel Palame – IoC Containers Benchmark
Updated 2016-05-22

[24] MSDN What is Xaml?
Updated 2016-05-22

[25] MSDN - The MVVM Pattern, Microsoft patterns & practices
Updated 2016-05-22

[26] TechDays 2010: Understanding the Model-View-ViewModel pattern
Updated 2016-05-22

[27] Martin Fowler – Presentation Model
http://martinfowler.com/eaaDev/PresentationModel.html
Updated 2016-05-22

[28] MSDN - Presentation Model
Updated 2016-05-22

[29] Patterns – WPF Apps With The Model-View-ViewModel Design Pattern by Josh Smith
Updated 2016-05-22
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/
Updated 2016-05-22

[31] Xamarin Developer – Shared Projects
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/shared_projects/
Updated 2016-05-22

[32] Xamarin Developer – Introduction to Portable Class Libraries
https://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries/
Updated 2016-05-22

[33] Xamarin Developer – iOS Limitations
https://developer.xamarin.com/guides/ios/advanced_topics/limitations/
Updated 2016-05-22

[34] Xamarin Developer – Android Limitations
https://developer.xamarin.com/guides/android/advanced_topics/limitations/
Updated 2016-05-22

[35] Assemblies in the Common Language Runtime
https://msdn.microsoft.com/en-us/library/k3677y81%28v=vs.110%29.aspx
Updated 2016-05-22

[36] MSDN – CreateionPolicy Enumeration
Updated 2016-05-22
[37] Xamarin Developer Guides
https://developer.xamarin.com/guides/
Updated 2016-05-22