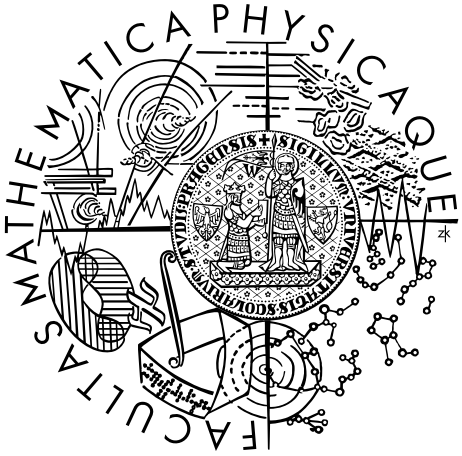
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Marek Linka

Universal Blog Manager

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek

Study programme: Computer Science

Specialization: IP

Prague 2012

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.
I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... Date ..... Signature .....

Název práce: Universal Blog Manager

Autor: Marek Linka

Katedra/Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek

Abstrakt: Sdílení názorů a pocitů na Internetu (blogování) se s přibývajícimi lety stává stále populárnější. Vývojáři aplikací na tento trend reagovali a vytvořili mnoho desktopových aplikací ulehčujících publikovaní obsahu. Ale téměř žadná z dostupných aplikací nedokáže uživateli spřístupnit celý potenciál blogovací platformy. Rozhodli jsme se proto zaplnit tuto medzeru tím, že navrhneme desktopovou aplikaci rozšiřitelnou pomocí zásuvných modulů, která by byla schopna spřistupnit jakýkoliv blog a využít celý jeho potenciál, zatímco by zůstala intuitivní a jednoduchá na použití. Dokončením našeho záměru jsme vytvořili aplikaci dostatečně mocnou pro experty, ale stále dostatečně jednoduchou i pro běžného uživatele. Tato aplikace účinně zaplňuje mezeru vytvořenou aplikacemi, které se víc zaměřují na publikovaní obsahu.

Klíčová slova: Blog, Zásuvný modul, Windows, Internet

Title: Universal Blog Manager

Author: Marek Linka

Department/Institute: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek

Abstract: Sharing opinions and feeling on the Internet (blogging) is becoming increasingly popular with passing years. Application developers reacted to this trend and created many desktop applications making publishing simple. But almost none of these applications bring the whole power of the blogging engine to the user. We decided to fill this gap by designing a plugin-extensible desktop application capable of accessing any blog and harnessing all its potential, while still remaining simple and intuitive. By completing our intent, we created an application powerful enough for an exert user, yet simple enough for an average blogger. This application effectively fills the gap left open by more publishing-focused blogging tools.

Keywords: Blog, Plugin, Windows, Internet

Dedication

To my father - for teaching me that hard work is the only way to achieve a true success.

1. Introduction

1.1. What is a blog?

> Blog: a Web site that contains an online personal journal with reflections, comments, and often hyperlinks provided by the writer; also: the contents of such a site.
> — Merriam-Webster Online Dictionary[1]

In short, a blog is a user's own personal space on the web. The user can use it to write about anything and everything he or she wants – from his/her day to day life to the latest trends in software engineering. It is a place to express and share opinions about the world.

The term *blog* evolved as a blend of the words ‚web log'. This notion alone implies the nature of the earliest blogs – they were just day to day logs of people's lives. As the Internet penetration increased over the years, blogging became available to more and more people. As many of the potential users were not skilled enough in HTML and similar technologies, blogging tools and platforms were created to simplify the process.

There are many blogging platforms nowadays, ranging from very simple mail-based web applications designed for personal blogs to complex and robust Content Management Systems (CMS) employed by large companies and international corporations.

Among the most popular platforms are: WordPress[2], Tumblr[3], Blogger[4], Posterous[5], Movable Type[6] and many more. It should be noted however, that basic blogging concepts are platform independent and the terminology is usually common for all of the blogging platforms.

1.2. Blogging basics and terminology

The basic component of a blog is a *blog post* (post). Posts represent a journal entry written by the blog author and displayed to the blog visitors. Posts usually contain text, images, audio and video content.

There are several ways a user can create a blog post – by sending an e-mail to a special e-mail address, using an online blog post editor or by using a special desktop/mobile application. Since blogs are basically web pages, most blogging platforms use HTML to store blog content. However, no average computer user can write well-formed

HTML code, therefore most of the blogging platforms offer *WYSIWYG* (What you see is what you get) mode editing. In this mode, the user sees the document exactly as it will be rendered on the web, without the need to actually see and understand HTML.

However, writing a post is just the tip of the iceberg – most platforms offer additional features, such as publishing blog pages (they are basically static posts and are usually displayed in a different manner), tracking number of people visiting the blog, the ability to modify the look and feel of the blog etc.

More complex platforms even offer their own *application programming interface* (API) to allow third party application developers to integrate their applications with blogs.

1.3. Problem statement

As stated before, there are many different blogging platforms available to the user. But the moment a user chooses a specific platform, it is very difficult to switch it for a different one. Since each of the platforms has its own user interface and there is basically no standardization, the user might choose to stay with his/her platform just to avoid the need to learn a new “set of tricks”.

The aim of this thesis is to solve this problem - to hide the big and small differences behind a unified user interface, so the user need not remember whether he/she is currently publishing to a Blogger account or a WordPress account. Migrating a blog from one provider to another will be also made less painful for its owner - the provider will change, but the user interface will remain the same.

Admittedly, there are several applications that offer similar features, but their approach is different - they concentrate on the most common tasks and ignore the advanced features of the respective blogging platforms. On the other hand, this thesis aims to be as flexible as possible, allowing the user to complete tasks he would otherwise be able to complete only by the use of the original blogging platform's user interface.

1.4. Goals of this thesis

The overall goal of this thesis is to create a platform independent blog management tool (by “platform” we mean a blogging platform).

This can be decomposed into the following project goals:

- **Universality** - the ability to support any popular blogging platform
- **Extensibility** - the process of extending the list of supported platforms must be as simple as possible
- **User-friendliness** - the application must remain simple to use and user-friendly even when working with many different blogging platforms

To meet the project goals we decided to split the application into two parts – the main application (user interface) and plugins (functionality). This way the user interface remains platform-independent and the task of extending the list of supported platforms is as simple as installing a new plugin.

1.5. Project history and evolution

This project began several years back as a simple idea to write a library for interacting with the WordPress API. After the library was finished, we decided to write a simple application demonstrating the capabilities of our new library. The demo application slowly grew in size and complexity and when it was finished, it was a full-fledged WordPress management solution designated as version no. 1.

The next iteration of development brought technological advancements as well as vast improvements to the abilities and overall application robustness. Upon finishing, the application reached version no. 2.

After that, we decided to incorporate other platforms into our application and that lead to creation of the Universal Blog Manager (UBM) project covered in this thesis.

2. Analysis

2.1. Basics

This chapter describes the most basic decisions made at the start of the project and states the areas the analysis will focus on.

2.1.1. Target platform

Before we could start analyzing the issues we will face while working on the project, we had to choose a target platform for out application. We wanted UBM to be a desktop application for Windows, so we had basically two options: either use C++ or a language from the Microsoft .NET family. Of course, there are several more possibilities, but those are more exotic and not so widespread. In the end, the author decided to use Microsoft C# and .NET Framework 2.0 (later upgraded to 3.5), because he is more skilled in the language and because he considered C++ an overly complicated language for writing this kind of application.

2.1.2. Areas of interest

There are several key areas we needed to analyze while working on this thesis:

1. **Plugins**

   The most important area of all - how to implement the extensibility of our applications in a simple and efficient way, including exception handling and plugin management.

2. **Data structures**

   How to store the application and user data coming from different sources in different form in a way that would allow out API to remain as simple as possible.

3. **Application's user interface**

   The application's user interface must be flexible enough to accommodate as many different blogging platforms (plugins) as possible while remaining simple and intuitive.

4. **WYSIWYG HTML editor**

   As stated before, most blogging platforms provide their own WYSIWYG HTML editors. To compete with them, we had to implement one of our own.

5. **Application management**

   Blogging platforms evolve fast and so do their APIs. We had to create a way to deliver updates to the end user reliably and quickly.

2.2. Plugins

When writing a plugin-enabled application, the first question that should be asked is “What is a plugin?” By definition, *plugin* (or plug-in) is “a small piece of software that supplements a larger program” [7].

This chapter discusses the decisions made regarding the plugin architecture, including exception handling and the way the main application uses the plugins.

2.2.1. Plugin architecture basics

Most plugin-enabled applications use plugins only to extend its features, but are fully functional even without them. UBM is a little different in this matter. The whole application is designed to manage blogs that are usually stored in the Internet, but only the plugins know what address to connect to and what data to send. In other words, the application does not know where the data is coming from. Basically, the main application is a front-end, while plugins provide the back-end. There is a clearly specified contract between the to, which provides a layer of abstraction for the application - the application only knows “what” to do, but doesn't care “how” to do it.

Implementing a plugin support imposes a specific set of requirements on the main application. First, the application must be able to load plugins dynamically, by which we mean that the plugins are not hard-coded into the main application, but can be added later by the end user.

Second, the application must provide a standardized interface that the plugins must implement in order to be recognized by the main application. As a side-effect, this also enforces the developer to analyze the functions of the main application to a deeper extent, thus improving the overall application robustness.

There are more issues to take into consideration when writing a plugin-enabled application, such as exception handling and plugin management, all of which will be covered in the subsequent sections.

2.2.2. Dynamic loading

Dynamic loading is the cornerstone of any viable plugin architecture. In short, it means that the main application must be able to find, load and use available plugins at run-time.

Without the ability to load plugins at runtime, the application would have to be recompiled every time a “plugin” is added to it. This is not desirable for an application that is to be used by an average computer user.

Fortunately, .NET Framework contains everything necessary for implementing dynamic load in a relatively simple way - using .NET Reflection.

.NET Reflection is a part of .NET Framework that allows for programmatic access and analysis of .NET assemblies. By using reflection, it is possible to load an external assembly into memory, inspect its content (classes, methods etc.) and even create instances of its classes and call methods.

There are two technologies built on top of the reflection API - Managed Add-in Framework (MAF) and Managed Extensibility Framework (MEF). Both technologies provide a way to create plugin-enabled applications. MAF is older of the two and shipped with .NET Framework 3.5, MEF is available from .NET Framework 4. Using these technologies would most likely result in a simpler and more robust application design, but we could not use them from historical reasons - the original project targeted .NET Framework 2.0.

When implementing a plugin architecture in .NET without any third-party libraries (or aforementioned technologies), reflection API is virtually the only way to go, because .NET Framework does not provide any other way to load assemblies at runtime.

It is also important to mention that from the beginning of the project we expect a single plugin to be stored in a single .NET *assembly* (compiled code library used for deployment, versioning, and security). While theoretically possible, it is not recommended to spread one plugin among several assemblies, nor to include multiple plugins in a single assembly. The reason behind this recommendation is that it could greatly complicate plugin management for the end user (what if

the user wanted to uninstall a plugin stored in an assembly containing several other plugins?).

2.2.3. Unified architecture

In order for our main application to make any use of available plugins, we must provide means of communication between the application and its plugins. In more technical terms, we must create an interface through which the components exchange data and commands.

Both sides of the relation must “understand” the interface, meaning that the interfaces must be visible to both the plugins and the application, otherwise it would be impossible to call plugin methods from the application.

If we kept all the shared data in the main application assembly, we would also complicate plugin development - every developer considering to write a new plugin would have to install the whole application. In such a case, plugins would have to reference the main application assembly to see the required interfaces. And since assembly references are version-sensitive (if assembly A.dll references assembly B.dll with version 1.0.0.0, this reference will be broken when updating B.dll to version 1.0.0.1), upgrading the main application to a newer version would be a breaking change. All installed plugins would be unusable after every application update.

In order to facilitate this requirement and avoid the mentioned problems, we decided to declare our interfaces, shared data types and other constructs needed on both sides of the equation in a separate library (“proxy” library). Both the main application and the plugins reference this library, thus having access to all the shared objects contained within.

It is very important to design this shared library to change as little as possible over time. Any breaking change made to this library breaks all plugins written against its objects, meaning every plugin would have to be rewritten.

By extracting shared objects into a separate assembly we also make plugin development easier, because the only thing a third-party plugin developer needs to start writing a plugin is this shared library - no need to install the whole application.

2.2.3.1. Interface vs. class inheritance

After designing the unified architecture, it is vital to be able to enforce it. Without enforcement, the unification is only a recommendation.

Therefore we needed a way of saying “if this plugin is capable of doing A, it also must be capable of doing B and C, otherwise it is not a valid plugin”.

In .NET, such an enforcement might be done by using class or interface inheritance. The main premise of this approach is this: “If a class extends class X/implements interface Y, it is a valid plugin”. Inheritance will make sure that all the necessary properties and methods are implemented.

The matter of class vs. interface is closely related to the monolithic vs. granular problem (discussed later) - abstract class is better suited to work with monolithic plugin base, while interfaces combine better with the granular approach.

The main limiting factor in this debate is the fact that C# only allows a class to extend a single parent class, whereas a single class can implement an unlimited number of interfaces.

Since we decided to use the granular approach to plugin base, we had to use interfaces to enforce plugin integrity.

2.2.3.2. Singularity vs. granularity

There are two main approaches to defining a plugin through an interface. The first is to use a single large interface containing all the plugin methods. This is very straightforward and easily implemented both in the application and in the plugins.

The main application benefits from this approach, because the developers can be sure that when a plugin implements this interface, it **must** contain all the methods the main application needs.

The second approach is the exact opposite. Instead of a single monolithic interface, we divide the interface methods to small groups according to their usage (e.g. methods for managing blog posts, methods for managing pages etc.).

Both approaches have its benefits, but also their limitations. A monolithic plugin base would require less overhead in the main application, where the UI changes according to plugin capabilities. Checks would require less type casting and type comparisons could be dropped entirely. The downside of this solution is that it is not flexible - adding support for new features would require breaking changes in the plugin base and existing plugins would stop working.

Granular approach is much more flexible - adding a new set of features would only require creating a new interface and check

whether a particular plugin implements it. Old plugins would still remain functional, only the new features would be unavailable. In this scenario, the main application must constantly check whether a plugin implements a specific interface required for a specific function, which requires a lot of type comparisons and casting.

We decided to use the granular approach to plugin base, mainly because even though it is more complex, it is better to optimize the application API for the plugin developers, not for the main application developers. It also provides a better solution of backwards plugin compatibility.

2.2.3.3. Constructors

In C#, it is impossible to enforce existence of a constructor with a specific signature using inheritance. This greatly complicates the loading of installed plugins, since the main application must be able to create instances from the plugin classes even though it does not know what constructors are available.

After much deliberation and researching, we decided to require the plugin developers to provide a default public parameterless constructor for each plugin. This way the main application can create instances of plugins using the reflection API with little complexity. It is impossible to truly enforce the existence of this specific constructor - the closest thing (which we implemented) was to reject a plugin if it doesn't contain it. This way if a developer wants to debug a plugin, he/she must implement the default constructor in order for the plugin to at least load properly.

Using parameterless constructors to create plugin instances means that the instances will not be fully initialized - the application is unable to provide any user data (such as username and password) to the plugin. Therefore, the plugin created using its default constructor is practically useless.

We circumvented this limitation by introducing a special `LogOn` method, which is used to finalize plugin initialization by providing the user data needed.

It is very important that this method returns a **new instance** of the plugin. This is necessary, because the end user might have several accounts using the same plugin. If the `LogOn` method only finalized initialization of itself, it would be impossible to work with two same-plugin accounts at the same time.

The `LogOn` method should be therefore considered a factory method. Factory method design pattern is usually employed whenever it is

desirable to leave the decision which class to instantiate to the derived classes. This is only partially true for UBM, as the `LogOn` method usually returns an instance of the same type as it was called on. The main benefits of using a factory method pattern in UBM are a) to be able to enforce its existence using interface inheritance and b) encapsulating the possibly complex code necessary to initialize the plugin into a single method, thus preventing potential code duplication and hiding the complexity from the main application.

2.2.4. Exceptions

It is very important to think about exception handling when designing a plugin architecture for an application. The root of the problem lies in the fact that the main application designer doesn't know how the plugins are implemented on the inside.

The plugin might throw virtually any kind of exception, including custom types. This makes exception handling very problematic – it is not feasible to prepare for every possibility. Writing a general `catch` clause in every potentially dangerous place is not considered a good practice, because it will catch even exceptions the user code is not able to handle properly - e.g. `OutOfMemoryException` or `StackOverflowException`.

We decided that the best way to circumvent this is to declare our own custom exception types in the proxy library and demand that every plugin must not throw any other type of exception.

Of course, there is no way to enforce this behavior in our interfaces and classes and we must rely on the plugin developers to respect this rule. Additionally, the main application contains an application-level exception handler for all the exceptions that are not caught. This approach allows the application to terminate gracefully even in the event of an unexpected exception (whether it comes from a plugin or the application itself).

2.2.5. Plugin management

The last non-trivial issue with plugins lies in their management. The end user must be able to install and uninstall plugins. But what does “to install a plugin” actually mean?

The answer to this question depends on the way the main application searches for plugins: a) the easiest way is to use a special folder to store plugins. At startup, the application just looks through the folder's

content and loads all available plugins. Or b) it would be possible to use Windows registry to store the necessary information. c) or something entirely different (such us custom-formatted configuration files) could be implemented.

When writing the UBM, we decided to go with the technique designated as a), because it is very easy to implement and minimizes the risk of malfunction in case of damaged registry or configuration files. Also, it is very simple to “install” a plugin – just copy the plugin to the right folder and restart the application. Every computer user can manage such a task.

However, copying assemblies is still not very user-friendly, even if it is simple. It involves opening the application installation directory, locating the necessary folder and pasting the plugin assembly. This can be tiresome for many potential end users. To improve the ease of use of UBM, we decided to write a plugin management application to improve the user experience. The main function of this plugin manager is to provide the end user with a simple and clean way of managing installation and uninstallation of UBM plugins.

As discussed above, plugin installation is simple. The user selects plugins he/she wants to install, the plugin manager verifies that the selected assemblies contain valid plugins and then performs the copying automatically.

Plugin uninstallation, however, poses a significant challenge. In order for the user to pick the desired plugin he/she wants to uninstall, the plugin manager must be able to display details about the currently installed plugins (such as plugin name, author, version etc.). Because plugins are contained within .NET assemblies and there are no files containing meta-information, the plugins must be loaded and certain methods executed to read this information.

The problem lies in the fact that once an assembly is loaded into the main application domain, its source file cannot be deleted until the application that loaded it is terminated - meaning that once the plugin manager loads a plugin and displayed its meta-information, it cannot “uninstall” it by deleting the assembly file.

To solve this problem, we at first intended to create a third executable (other than the main application and the plugin manager) that would receive the list of plugins the user decided to uninstall, wait until the manager is terminated (thus ensuring the plugin assemblies are unlocked) and then delete the files. This solution was rejected early in the analysis process, however, because even though working, it is more of a workaround than a solution to the core problem. Additionally,

it is not “clean” and the plugin manager must be terminated/restarted each time a plugin is uninstalled, which interrupts the user's workflow.

After researching the topic of dynamic unloading, we decided to use temporary application domains to load plugin information. The following section provides details on using application domains to achieve run-time assembly unloading.

2.2.5.1. Dynamic assembly unloading in .NET Framework

In order to solve the plugin uninstallation issue cleanly and robustly, we must be able to dynamically load **and unload** the plugin assemblies. After researching the subject, we designed a solution based on .NET application domains.

According to the Microsoft Developer Network[25] (MSDN), application domains

> provide an isolation boundary for security, reliability, and versioning, and for unloading assemblies. Application domains are typically created by runtime hosts, which are responsible for bootstrapping the common language runtime before an application is run.
>
> — Microsoft Developer Network - Application Domains[26]

Additionally, the MSDN article describing the `AppDomain` class states the following:

> If an assembly is loaded into the default application domain, it cannot be unloaded from memory while the process is running. However, if you open a second application domain to load and execute the assembly, the assembly is unloaded when that application domain is unloaded.
>
> — Microsoft Developer Network - AppDomain class (System)[27]

That is exactly the behavior we need in order to facilitate dynamic assembly unloading - we load the assembly into a separate temporary application domain, retrieve the necessary information, pass this information to the main application domain and unload the temporary domain, thus unlocking the assembly.

However, there is still a small problem hidden in this concept - in order to prevent the main application domain from also loading the plugin

assembly (which would result in locking the assembly files), no plugin object may pass the application domain boundary. In other words, we must not use any objects from the plugin assembly in our main application domain. This complicates the process of retrieving and displaying the plugin information.

The solution to this limitation is to use a special object that can pass application boundaries safely to retrieve the data from the plugin (in the temporary domain) and pass it to the plugin manager (in the main application domain). We can create such an object by extending the `MarshalByRefObject` class.

When a class is derived from `MarshalByRefObject`, we are capable of instantiating it in the temporary domain and call it from the main application domain using a proxy. This way, we can analyse the plugin assembly exclusively in the temporary domain and pass the results safely to the main application domain, without exposing any plugin assembly objects.

The combined use of application domains and an object derived from the `MarshallByRefObject` class allows the plugin manager to dynamically load plugin information, display it to the user and unload the plugins from memory. This ensures the ability to delete plugin assemblies, effectively unistalling them from UBM.

2.3. Data structures

This chapter discusses the topic of passing data between the main application and its plugins. It covers both the basic concepts and the implementation-specific details.

2.3.1. Plugins and data

When a developer decides to write a plugin-enabled application, the reason usually is that he/she has some kind of data that can be processed in many different ways depending on the back-end/ conditions/purpose etc.

There is no reason to use plugins when this data processing is very simple – it would be much easier to hard-code such processing into the main application. Therefore most plugins perform complex operations on complex data. And here comes the problem: since every plugin in our system might have different requirements, the main application must store its data in a way that is able to accommodate all of these requirements.

An example: Suppose we have two plugins, both containing a method to retrieve the latest blog post for the current user's account. One plugin operates on WordPress blogs, the other on Blogger blogs. WordPress blog posts contain many attributes, such as post password or geolocation information. In comparison, Blogger posts are much simpler and contain only a few attributes, such as post tags and a permalink (a permanent link to a blog post/page that does not change over time). And yet, the methods must in both cases return the same data object.

Fortunately, C# programming language already provides means to overcome this problem - in the form of rooted object hierarchy and its `Object` class. In C#, everything is inherited from the `Object` class. A collection of `Objects` is capable of storing any data, whether its type is `string`, `int` or `Uri`. By using collections of `Objects`, we can accommodate any and all requirements the plugins may provide.

To improve the concept, we decided not to use simple `Object` collections, because it would unnecessarily complicate the process of retrieving and manipulating the data. Instead, we used key-value collections, where the key is of type `String` and the value is of type `Object`. This way retrieving the right resource from the collection is trivial, which greatly simplifies the inner workings of the plugins. Every record is identified by name, therefore developers don't need to operate with positions of objects in the collections.

It is also important to realize that using this approach means the main application will not know how to display this “additional” data. Since only the plugin knows what data of what type is stored where and under which name, the plugins must be able to provide the main application with descriptions specifying what data to display and how.

We designed a set of objects and methods for this purpose. You can find more details about these in the next section.

2.4. Modular user interface

This section covers all the concepts and implementation details of a user interface (UI) that supports working with extensible data structures described in the previous section.

We have already established that plugins usually require extensible data structures. But where does the data actually come from? It is either from a back-end (usually well documented) or from the user.

The first possibility is fine – the plugin developer knows exactly what data to expect and can prepare accordingly.

The situation becomes much more interesting (and complex) when a user's input is required. The complexity has its source in the fact that in order to facilitate this input, the application must provide the user with means of editing an **extensible** data structure it doesn't completely understand. In simpler terms, the user interface must be flexible enough to allow for any (supported/expected) data to be edited.

The most important step leading to a solution to this issue is to discern the types of data that are likely to be used by blogging platforms. In other words: what UI components will we need to design to support the most widely used data types? Some data can be very simple - for example permalinks, which are usually in the form of simple text. Then there are somewhat more complicated structures, such as dates and times. And finally, there are data as complex as blog post categories, which are usually in the form of a set of tree nodes (categories are often hierarchical).

After we implemented the necessary components, we had to create a set of methods that accept extensible data structures and return a description of the UI to render. More details about these methods can be found in the programmer's documentation.

2.5. The WYSIWYG editor

In short, WYSIWYG editors provide the ability to edit HTML (and other structured languages, such as XML) in a visual editor (as opposed to text-mode editing – manually writing the HTML tags).

Most blogging platforms use HTML to write and store blog posts and other blog objects. They usually also provide a browser-based WYSIWYG editor in their web administration interface to spare the users the necessity to learn and use HTML markup. In order for UBM to be a full-scale replacement of these web interfaces, we had to provide a WYSIWYG editor of our own.

To create a WYSIWYG editor, we first need an HTML rendering engine. There are several such engines available: Trident (MSHTML)[8], Gecko[9] or WebKit[10]. Gecko is used by Mozilla's Firefox browser, WebKit by Apple's Safari. Both these engines are considerably more powerful then Microsoft's MSHTML (both in performance and resource allocation), but since both of them are written in C/C++ languages, incorporating them in a .NET application means a non-trivial task of writing a very advanced wrapper library around their objects. The greatest advantage of MSHTML over Gecko and WebKit is that Microsoft already provides a complete .NET

wrapper around MSHTML. And that is also the main reason we decided to use MSHTML in UBM - writing a complete wrapper around Gecko or WebKit is currently beyond the scope of the project.

The engine library contains classes, interfaces and other objects used to manipulate HTML markup. Using this library we can take the page rendered in our UI and programmatically change it according to our demands – add formatting, insert images, modify text and many more.

2.6. Application updates

There is one last issue with blogging applications left to discuss: blogging platforms are constantly evolving (for example, WordPress releases several incremental versions per year). This means it is probable that the APIs will change overtime, which will result in malfunctioning plugins. In order to minimize the impact such an event will have on the users, we should provide an easy way to distribute application and plugin updates.

This idea has not been completely implemented yet - Universal Blog Manager is distributed together with an updater application, but there is currently no way of automatically updating individual plugins.

The updater application is quite simple - it contacts a web service in the Internet, checks the current application version number against the latest version number and if the server contains an updated version, downloads it and unpacks it.

The ideal solution to the problem of plugin updating would be to create a central repository of plugins in the Internet - this way the application could check the versions there and automatically download updated plugins. The concept is not particularly complicated and this feature will definitely be added in the future versions of the Universal Blog Manager.

3. Plugins

3.1. WordPress Plugin

This chapter analyzes our implementation of a Universal Blog Manager plugin for the WordPress blogging platform.

3.1.1. XML Remote Procedure Call

WordPress is probably the most robust and complex blogging platform currently available. Among other features, it also provides its own API, which third-party developers can exploit to integrate their applications with WordPress-powered blogs. At its core, the WordPress API (WP API) is actually a set of *XML Remote Procedure Call* (XML-RPC) methods. XML-RPC is a widely used standard for calling complex methods from remote endpoints. It uses HTTP for transport and XML for encoding.

The use of XML-RPC has several advantages - it is free, it is well documented and implementations exist for all the currently popular programming languages. Full XML-RPC specifications can be found at the XML-RPC home page[12].

WordPress API specifications are available at the WordPress Codex[13]. This specification only covers the WordPress-specific methods. In addition, WordPress also completely supports Blogger API[14], Metaweblog API[15] and Movable-type API[16].

3.1.1.1. XML-RPC in .NET Framework

The first issue we encountered when working on the WordPress plugin was that while XML-RPC is a widely used technology, .NET Framework does not support it natively in its *Base Class Library* (BCL). There are several third-party implementations available for download, but using them presents a different problem: if we used a third party library, we would have to distribute it together with our plugin. We dismissed the concept, as it might easily lead to plugin and version conflicts (for example two plugins written against two different versions of the same XML-RPC library).

To solve these issues, we decided to handle the XML-RPC ourselves. We also noticed that we didn't need to write a universal XML-RPC client library, but rather only a given set of XML-RPC methods with

well documented parameter lists and endpoints - instead of solving the problem in a universal way, we would only write a code for a very specific subset of the problem. This significantly reduces the complexity of the problem.

An XML-RPC call is basically a HTTP POST request to a specific address with a payload in a specific format. Making HTTP requests from .NET Framework is fairly simple - the BCL contains all the necessary classes. The request body is filled with XML data in a specific format (see the XML-RPC specifications).

3.1.1.2. Constructing XML-RPC queries and reading responses

We already established that the data sent and received though the use of XML-RPC are in the form of XML documents. .NET Framework currently supports two different approaches to writing and reading XML data - ***Document Object Model*** **(DOM) manipulation** and **LINQ-to-XML** [17].

Since our initial implementation targeted .NET Framework 2.0, we were forced to use the DOM approach (LINQ-to-XML became available with .NET Framework 3.5). This led to a very long and difficult-to-read code - every XML object (element, attribute etc.) needed a separate line of code and it was very difficult to imagine how the resulting XML document would look like. This in turn lead to difficulties with debugging, when a simple mistake could take up to thirty (or more) minutes to track down and fix.

This problem was solved by switching the target framework version to 3.5 and rewriting the code to use LINQ-to-XML. LINQ-to-XML supports functional XML tree construction - the whole XML document can be created using a single function call. This function can be easily split across multiple lines and indented so that the code resembles the resulting XML file. The following example (taken from the MSDN) illustrates the concept:

Example 3.1. LINQ-to-XML - functional XML tree construction

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144",
                new XAttribute("Type", "Home")),
            new XElement("phone", "425-555-0145",
                new XAttribute("Type", "Work")),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

When reading and parsing the returned data, we found ourselves in a similar position - the DOM approach led to a code that was very difficult to read and maintain. On the other hand, LINQ-to-XML allowed us to make use of the Language Integrated Queries technology, resulting in a cleaner and simpler code.

3.1.2. Retrieving blog statistics

WordPress blogs track several statistics for their owners: number of visits, links clicked, search engine terms visitors used to find the blog and so on. We wanted to display these statistics in our application, but after reading the WP API documentation, we discovered that there is no (documented) API method for retrieving these statistics.

At first, we decided to omit this feature entirely and shelve it for later versions. Over time, we gave this issue a great deal of deliberation and decided that the only place where the statistics are available is the web administration interface. With this in mind, we started to write a set of methods that would scrape the statistics from the statistics web page.

That required two main steps:

1. Log into the administration interface and navigate to the statistics page

2. Parse the HTML code and retrieve the statistics

3.1.2.1. Logging into the administration interface

WordPress uses forms authentication in its administration interface - the user enters his credentials, the server verifies them and if the login is successful, sends back a cookie that identifies the user as being logged in.

In order for UBM to be able to navigate to the statistics page, it needs to retrieve this login cookie from the WordPress servers. This has been implemented by simulating the user's input on the WordPress login web page - the application crafts an HTTP POST request with the necessary information and submits it to the login URL. But a problem within .NET Framework (all versions up to 4.0) was encountered when implementing parsing the server's response:

The response we received from the server always stated “Unauthorized” and the login cookie was not present, even though the login information was correct.

It took almost a week to pinpoint the problem, because it was located deep within the BCL. The issue arose because there is a problem in Microsoft's implementation of the class used to make HTTP web requests. The class handles cookies incorrectly when there is a “Location” HTTP header present. Consider the following scenario:

- Make a POST HTTP request to the login URL containing **valid credentials**

- The server verifies the login information and issues a response containing **a cookie definition and a redirection directive in a Location header** to a URL that requires login

- The .NET Framework parses the Location header, ends the response parsing and immediately redirects to the specified page **without parsing potential cookie headers**

- The protected web page **checks for the authentication cookie, fails to find it and redirects to the login page with the “Unauthorized” response**

We were forced to create a workaround for this issue that involved parsing cookies “by hand”. Later on we discovered that

an undocumented API exists that allows for simple retrieval of blog statistics and incorporated this API into the plugin.

3.2. Blogger Plugin

This chapter discusses the important aspects of our implementation of a plugin for the Blogger platform.

3.2.1. Blogger Data API

Blogger (just like its competitor, WordPress) also provides its own API. The entire API uses HTTP and XML technologies, although it is not XML-RPC based. Instead, the XML data are formatted according to the *Atom Publishing Protocol* (AtomPub).

The complete specification of the Blogger Data API are available from Google[19] . The AtomPub specifications are available from the Internet Engineering Task Force[20]

3.2.2. Authorization

Before the plugin can access the user's blog data, it must first prove to the server that it is authorized to do so. Blogger Data API support several authorization schemes:

- OAuth 2.0[21]
- OAuth 1.0[22]
- AuthSub[23]
- ClientLogin[24]

The OAuth schemes are the most advanced and secure - they both allow the user to keep his/her login credentials hidden from the client application. However, they were designed to be used mainly in web applications - pages trying to connect with other pages. It is still possible to use OAuth schemes from client-side (or “installed”) applications, but the implementation is not entirely straight-forward.

The AuthSub authentication scheme can only be used to authenticate web applications and is not suited for client applications.

ClientLogin is a very simple scheme, but it is much less secure then the OAuth schemes - the user needs to enter his/her credentials directly into the application, which presents a potential security issue.

Since we decided to write the Blogger plugin mainly to demonstrate the modularity of UBM, we decided to use the ClientLogin method, because of its simplicity. In case we decided to release the plugin publicly, we would have to rewrite the authentication code to use the OAuth 2.0 scheme to provide the best possible protection to the users.

4. Implementation

This chapter covers the overall organization of the application and describes its components.

4.1. Assemblies

The current version of UBM consists of six assemblies:

1. **Main executable** (`UniversalBlogManager.exe`)

   Project name: *UniversalBlogManager*

   This is the main executable of the whole thesis. It loads plugins and provides the user with user interface that he/she can interact with.

2. **Plugin proxy library** (`PluginBase.dll`)

   Project name: *PluginBase*

   This assembly contains all the classes and interfaces that need to be visible from both the main executable and the plugins. Main application, plugins and the plugin manager reference this library.

3. **Modular UI components library** (`ModularUIControls.dll`)

   Project name: *ModularUIControls*

   This assembly contains all the currently implemented custom user interface components used to work with extensible data structures and blog objects. The main executable references this assembly.

4. **WordPress plugin** (`Plugins\WordPressNet.dll`)

   Project name: *WordPressNet*

   This assembly contains the implementation of the UBM plugin for the WordPress platform. The main application loads this assembly at run-time.

5. **Blogger plugin** (`Plugins\BloggerNet.dll`)

   Project name: *BloggerNet*

   This assembly contains the implementation of the UBM plugin for the Blogger platform. The main application loads this assembly at run-time.

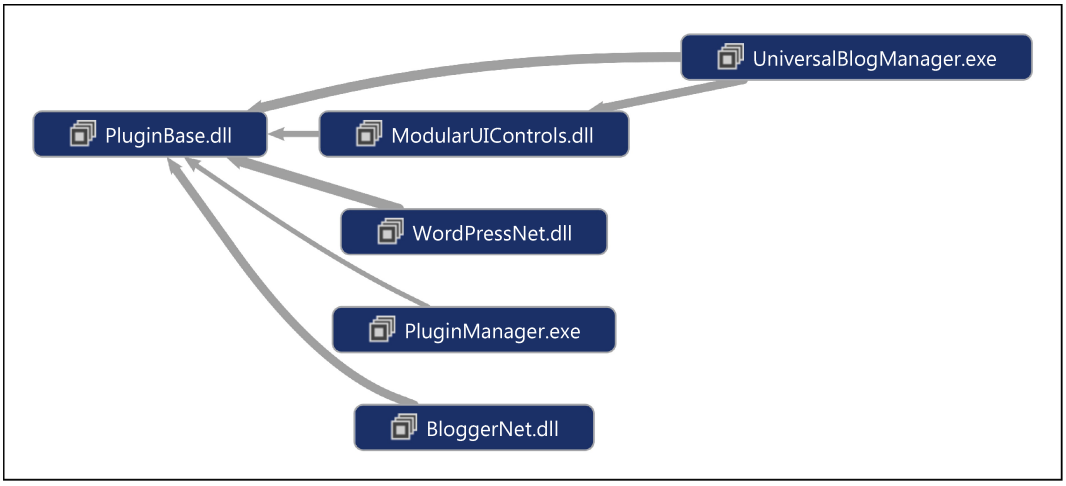
6. **Plugin manager** (`PluginManager.exe`)

   Project name: *PluginInstaller*

   This is a separate executable that allows end users to install and uninstall plugins easily and comfortably.

The following picture desribes relationships between the project assemblies. Grey arrow from assembly A to assembly B indicates that A uses objects stored in B. It is apparent that the main executable stands at the top of the hierarchy (right in the picture; it only consumes entities from other assemblies), while the library containing plugin-related entities is at the bottom (left in the picture; it only provides classes and interfaces to other assemblies).

Figure 4.1. Assembly dependencies



All six assemblies will be discussed in detail in the following sections.

There are two third-party components referenced by the main application - Agility HTML Framework[11] and MSHTML[8]. These are beyond the scope of this thesis and will not be discussed here.

4.2. Main executable

The main executable is the central spot of the whole thesis and as such references or uses all the other assemblies of the project. This assembly contains virtually all the user interface available to the end user (with the exception of modular user interface components stored in the `ModularUIComponents.dll` assembly and the UI components that might be implemented by plugins).

User interface components are implemented using Windows Presentation Foundation and every window/usercontrol is composed of two files - `[classname].xaml` containing the UI definition (in the form of eXtensible Application Markup Language) and `[classname].xaml.cs` containg the interaction logic (a “code-behind” file). By usercontrol we mean a separate reusable custom-written user interface component - a piece of UI that was created by us specifically for our application.

The main window (`MainWindow` class) contains only very little functionality - it only hosts UI controls for accessing registered blog accounts and the main menu. One of the menu items is capable of starting the plugin manager application.

Probably the most important class in the whole assembly is the `AccountControl` class (and usercontrol). This is the UI component that provides the user with access to all the controls for managing blog accounts and interacting with a single plugin (each `AccountControl` represents a single account; therefore a single plugin). It utilizes modular UI extensively, as well as plugin manipulation using plugin interfaces. Its components also use several usercontrols for displaying blog data, such as `DropDownButton`, `[Post/Page]List`, `[Post/Page]Item` and others. These are simply minor UI components designed to improve the ease of use of the application.

Other application window classes all begin with the letter 'w' (e.g. `wManageAccounts`) and can be divided into two categories: major and minor. Major windows implement the core application functionality, such as publishing blog posts or creating blog categories. Windows are usually only used to create or edit blog data - ability to delete blog objects is in most cases available directly in the `AccountControl` object.

Several of the major windows contain WYSIWYG editor implementation to allow the user to compose HTML documents comfortably. This WYSIWYG editor is currently not encapsulated in a single class/usercontrol, even though it would make its usage simpler. Major windows all manipulate plugins using the interfaces and classes from the `PluginBase.dll` assembly. They also usually use modular user interface components.

Minor windows are used to perform minor tasks, such as registering an account, verifying supplied user credentials or setting a master password for the application. Several of these windows use objects stored in the `PluginBase.dll` and `ModularUIControls.dll`

assemblies, but most of them have no connection to the plugins or modular UI components.

There are several support classes located in the main executable - classes representing event arguments (all classes with names ending with “Args”) or exceptions (classes ending with “Exception”) or classes containing application settings and assembly-wide methods (`Globals` class, with methods `SaveAccounts`, `DecryptAccounts` etc.).

4.3. Plugin proxy library

This assembly contains all the classes and interfaces required in both the main application and the plugins. The main executable and plugins reference this library, as it contains all the classes and interfaces used in plugin contracts.

Every entity in this library belongs to one of the following categories (every category will be discussed in detail later on):

- **Plugin interfaces**

  All of the available plugin interfaces are in this group. These are used to implement plugins and used by the main application to manipulate and call plugin methods without knowing the exact type of the plugin. The plugin manager application also utilizes these interfaces when loading and analyzing plugins to install.

- **Blog entities**

  This category contains all the classes representing blog entities, such as articles, comments, categories etc. These classes are used by plugins to return platform-specific plugin objects to the main application in a unified way. Every time the main application requests blog data, objects of this category are used to pass it.

- **Plugin exceptions**

  This group contains all the classes plugin developers can use to signal problems to the main application. These are usually raised by plugins and caught in the main application wherever plugin operations are performed (data retrieval and sending).

- **Custom message box implementation**

  Two classes for displaying custom message boxes (that are graphically consistent with the main application's UI) are in this

category. Message boxes are used primarily by the main application to display messages to the end user, but may be used by plugins as well. The Blogger plugin currently uses objects in this category to display messages related to logging into user accounts.

- **Miscellaneous**

  The last group contains several classes that could not be fitted into the previous categories.

4.3.1. Plugin interfaces

These are all the interfaces that define plugin capabilities. Several of the interfaces inherit from other interfaces (such as `IPosts`). The main application determines capabilities of a plugin by evaluating what interfaces are implemented in it and modifies its user interface accordingly.

These interfaces are the “building blocks” from which plugins are constructed. They are usually entirely independent on other interfaces, with the exceptions of the `IProvider-IPosts/IPages` inheritance and the `IComments-ITotalComments` inheritance. The following two figures illustrate the “hierarchy” of plugin interfaces.

The first figure (4.2) depicts the main plugin interfaces - `IPosts` and `IPlugins`. These interfaces are considered “main” because they are responsible for the most basic plugin functionality - retrieving posts and pages respectively. They inherit from the `IProvider` interface because we wanted to express the fact that when a plugin implements one of them, it is automatically a valid plugin.

The second figure (4.3) illustrates the additional interfaces. These are responsible for implementing additional functionality in plugins. `ITotalComments` inherits from `IComments`, because for a blogging platform to support retrieval of latest comments independent of their parent articles, it must also support retrieving comments for specific articles.

Figure 4.2. Inheritance of the `IProvider` interface

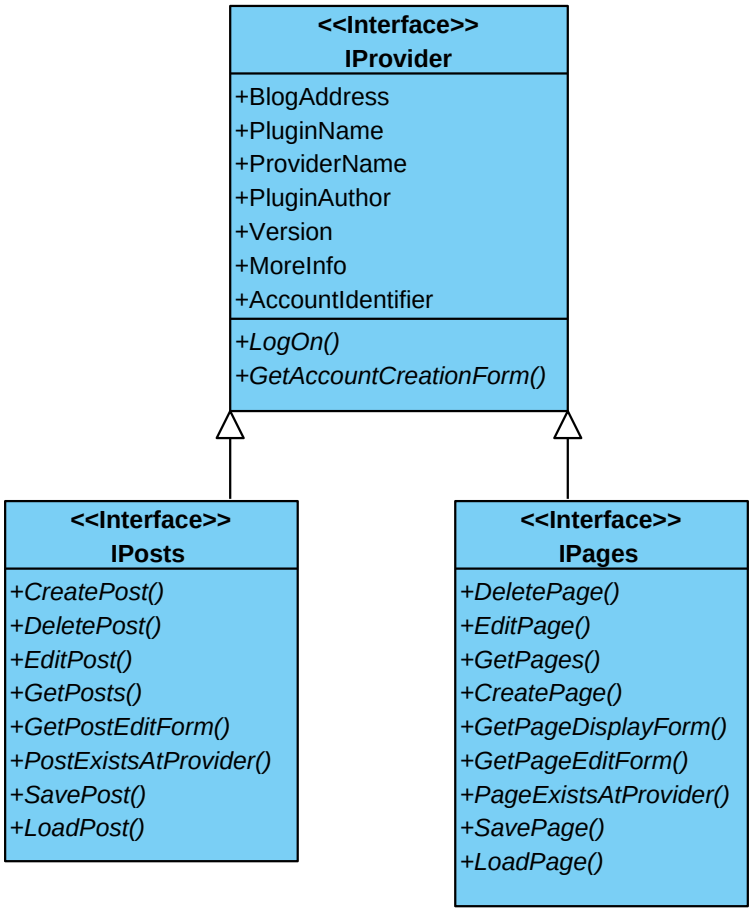
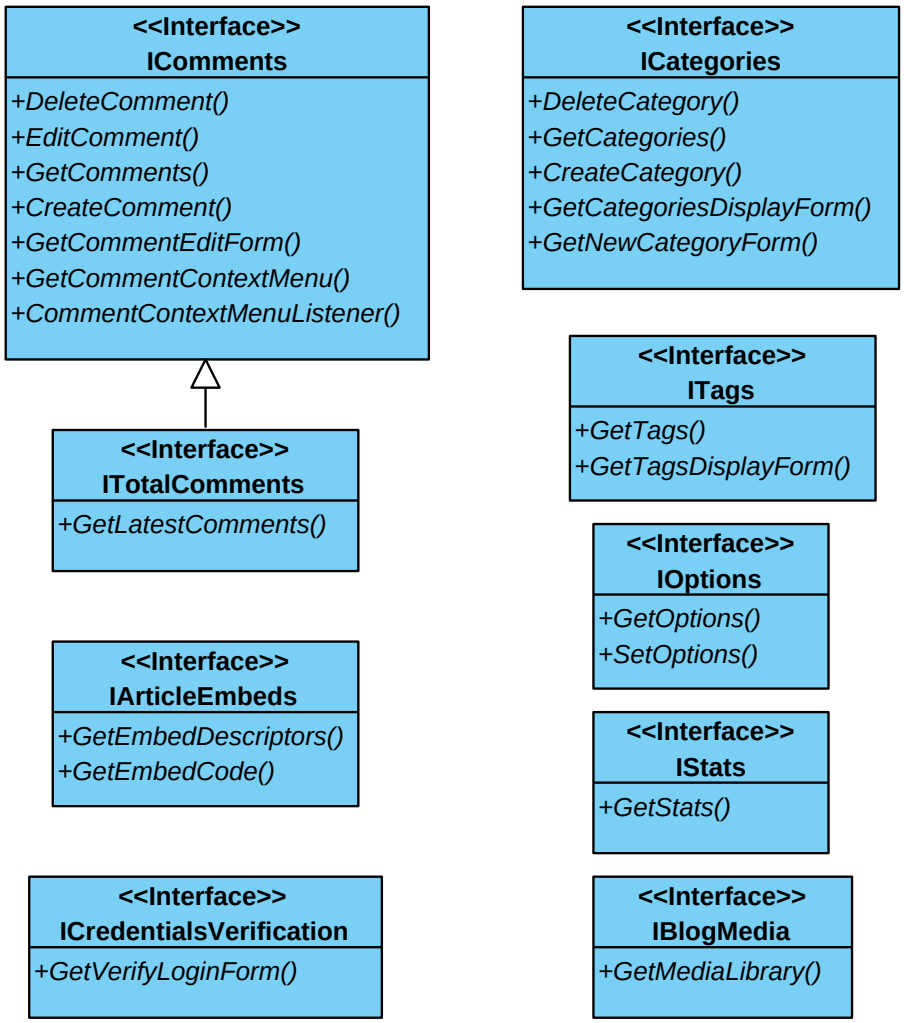


Figure 4.3. Other plugin interfaces



4.3.1.1. Interface description

The following list briefly describes the responsibilities of the respective interfaces (for full details see the programmer's documentation).

- IProvider

  Specifies methods and properties required for every valid plugin.

- IPosts

  Specifies methods required to work with blog posts.

- IPages

  Specifies methods required to interact with blog pages.

- IComments

  Specifies methods necessary for a plugin to be able to manage blog comments. Does not a contain method for retrieving the list of latest comments.

- ITotalComments

  Extends the `IComments` interface with a method to retrieve article-independent list of latest comments.

- IArticleEmbeds

  Specifies methods required to enable insrting of plugin-specific embed codes into blog articles.

- ICredentialsVerification

  Specifies methods necessary for credentials verifications.

- ICategories

  Specifies methods necessary for interacting with blog categories.

- ITags

  Specifies methods necessary for interacting with blog tags.

- IOptions

  Specifies methods necessary for displaying and editing blog options.

- IStats

  Specifies methods necessary for the retrieval of blog statistics.

- IBlogMedia

  Specifies methods necessary for the retrieval of blog media.

4.3.2. Blog entities

These classes are used to represent blog entities, such as posts, pages or comments. They provide a layer of abstraction over the actual blog data - it is not necessary for the main application to know how the actual object data looks like (e.g. whether it is XML or JSON

serialized), the plugin just needs to create the proper object from it and pass it to the main application. Classes in this group are used in the main application, plugins and the modular UI controls library.

Where metadata is expected, the classes contain special storage for it (see the Extensible Data Structures in Analysis). This storage is implemeted using a private `Dictionary<string, object>` with a getter and setter in the form of an indexer (`this[key]`).

It is important to note that both blog posts and pages are represented by the `Article` class, because these blog objects are usually very similar in structure.

Contained classes: `Article`, `BlogOption`, `Category`, `Comment`, `CustomField`, `MediaItem`, `StatItem`, `Tag`. Names of the classes are self-explanatory.

4.3.3. Plugin exceptions

Classes contained in this group are designed to allow plugin developers to throw exceptions into the main application safely (to learn more about exceptions, see Exceptions in Analysis and the Plugin Development Guide). The `abstract PluginException` class is used as a common ancestor for all the other exception classes.

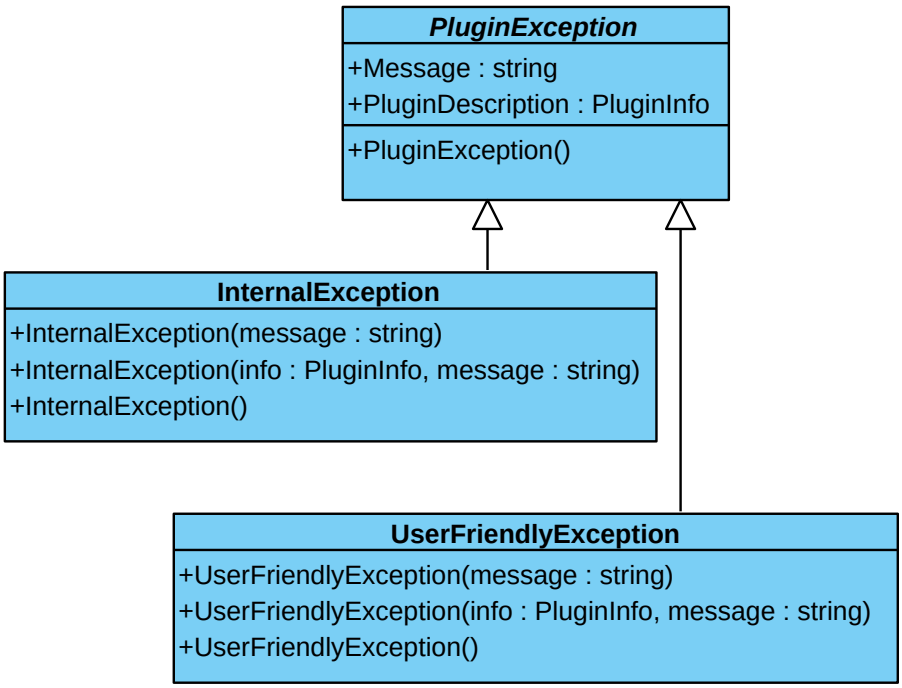
Throwing a `UserFriendlyException` will cause the main application to display a message box to the user containing the content of the exception message. This exception should be thrown in situations, where the plugin wants to notify the user about an “expected” exceptions - an exception that the user might be able to resolve. The exception message should be clear and easy to understand to the average end user.

When a plugin throws an `InternalException`, only a generic message about an unexpected problem is displayed to the user (unless the plugin debugging mode is active, see Debugging in Plugin Development Guide for details). This exception type should be used in cases where the exception is truly “unexpected” and it is not possible for the end user to rectify the problem.

Contained classes: `PluginException`, `UserFriendlyException`, `InternalException`.

The following figure illustrates the inheritance tree of plugin exceptions described above.

Figure 4.4. Exception class inheritance



4.3.4. Custom message box implementation

These two classes implement a custom message box used throughout the application. This message box has been designed specifically to display messages in a way that is visually compatible with the main application's user interface.

The actual message box window is marked as `internal` to prevent other assemblies from instantiating it directly. The `public static CustomMessageBox` class must be used to display a new message box (this behavior is very similar to the one of the native .NET's `MessageBox` class).

Classes: `CustomMessageBox` and `CustomMessageBoxWindow`.

4.3.5. Miscellaneous

These classes are used for various minor tasks mainly by the main application.

Classes: `CommentEventArgs`, `FormComponent`, `LoginInfo`, `MenuDescriptor`, `PluginInfo`, `Support`.

The `CommentEventArgs` class represent event arguments for comment-related events (from the `IComments` interface). Events are raised by plugins and handled by the main application.

The `FormComponent` class is used to describe user interface components in the UI-related plugin methods (see Modular User Interface in Analysis). Plugins return collections of `FormComponents` to the main application whenever plugin-specific user interface is required.

The `LoginInfo` class stores information necessary to log into a plugin account. This class is binary-serializable. Instances of this class are created by the main application (in the account management window) and passed to plugins when the user attempts to log into an account.

The `MenuDescriptor` class is similar in usage to the `FormComponent` class - it is also used to describe user interface. Context menus for working with comments are described using this class.

The `PluginInfo` class is lightweight object used to pass information about plugins in situations where only the basic information is necessary, not the whole plugin (such as when throwing an exception). `PluginInfo` objects are currently only used in conjunction with throwing plugin exceptions, which means the class is only used in plugins.

The `static Support` class contains methods that might be useful to plugin developers, but are not necessary. Only plugins currently use its methods.

4.4. Modular UI components library

This assembly contains all the currently implemented modular UI controls (to learn more about modular UI, see chapter Modular User Interface in Analysis). These controls are used by the main application to display and edit blog object properties and metadata. No other projects/assemblies use objects stored in this assembly.

Mudular UI components are implemented as WPF usercontrols - stand-alone and reusable UI components. To be considered modular, they must also implement the `IModularUIControl` interface. This allows the main application to intaract with different controls in the same manner.

Usage of these controls is simple - whenever the user starts a task that requires dynamic UI, the main application first queries the plugin

for a description of the UI for this task. Then the application parses the response and creates new instances of the required UI components, sets their value and displays them to the user.

When information needs to be passed from modular UI to plugins, the main application collects all the rendered modular UI components, extracts their keys and values and creates a list of key-value pairs. This list is then passed to the plugin. Since the plugin is the one responsible for specifying keys for the UI components, it knows what keys to expect.

Classes: `CheckBox`, `DateSelector`, `IntegerBox`, `Label`, `LinkLabel`, `MultilineLabel`, `MultilineTextBox`, `TimeSelector`, `TextBox`, `PasswordBox`, `CategoryDisplayTree`, `CategorySelectionTree`, `SelectionBox`, `StatsList`. Names of the classes should be self-explanatory.

4.5. WordPress plugin

The `WordPressNet.dll` assembly contains only a single public class - `WordPressApi`, which contains the implementation of the WordPress API. Since WordPress API is fairly complex (and powerful), this class implements virtually all the available plugin interfaces. To improve readability, the code has been split into several files using the `partial` keyword. This class is used by the main application and the plugin manager application through the plugin interfaces abstraction (the actual `WordPressApi` type is never used).

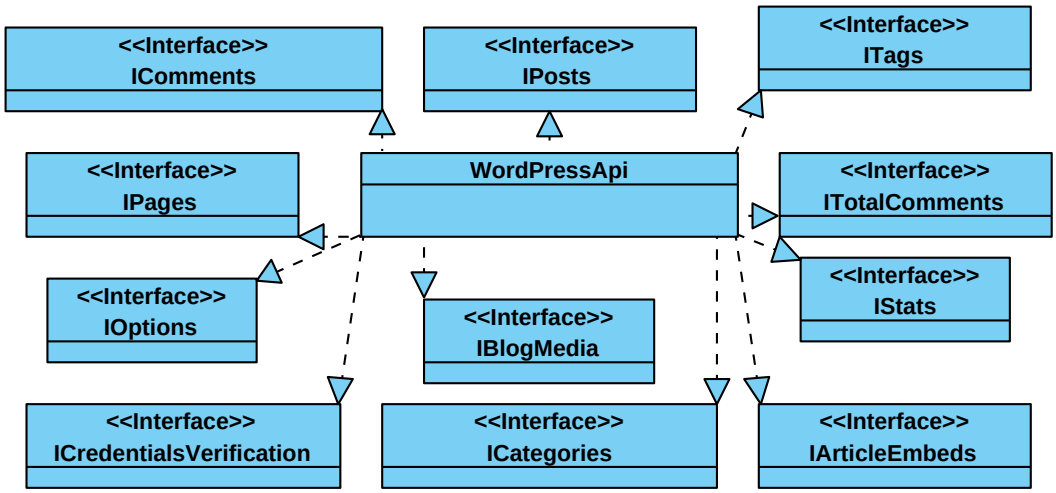
The assembly also contains several internal classes (e.g. `Author`, `CommentCount` or a static class `Support`) and an internal enum (`ArticleStatus`). These classes are usually simple helper classes used for preventing code repetition and extraction of often-used code.

There are also four internal UI windows declared in the assembly. These are displayed whenever a user attempts to embed a special object (such as a video or a music file) into a post/page. When the main application sends a request to embed such an object (using the plugin API), the required window is displayed to the user.

The `WordPressApi` class implements basically the whole WordPress XML-RPC API and its visibility is set to `public`, therefore it is theoretically possible to reuse this class in other projects that require interaction with WordPress blogs. In such a scenario, however, it is recommended to acquire the source code of the assembly and remove the reference to the `PluginBase.dll` library, thus eliminating the need to ship it with the `WordPressNet.dll` assembly.

The following figure illustrates the relationship between the `WordPressApi` class and the plugin interfaces.

Figure 4.5. `WordPressApi` class - plugin interface inheritance



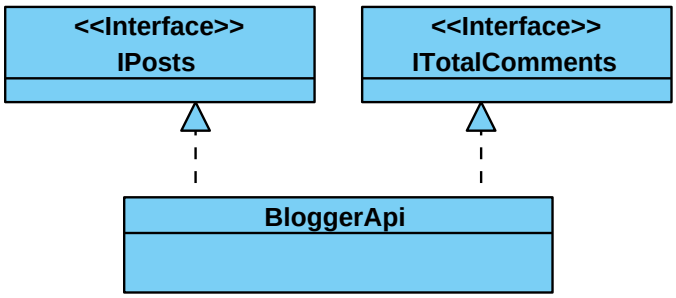
4.6. Blogger plugin

This section discusses the implementation of the Blogger UBM plugin.

The Blogger UBM plugin consists of a single class named `BloggerApi`. Since the Blogger API is currently very simple, this class implements only the `IPosts` and `ITotalComments` interfaces. This class is used by the main application and the plugin manager application through the plugin interfaces abstraction (the actual `BloggerApi` type is never used).

The following figure illustrates the relationship between the `BloggerApi` class and the plugin interfaces.

Figure 4.6. `BloggerApi` class - plugin interface inheritance



In addition to the interface-enforced methods and properties, it also contains several private helper methods.

You can find more technical details in the programmer's documentation.

4.7. Plugin manager

The plugin manager application basically contains only a single window (the main application window - `MainWindow`) and two classes for retrieving plugin information from assemblies.

The application cannot be started without a command line parameter specifying the location of the UBM executable. If this parameter is not present, an error message is displayed and the application is terminated.

The two classes for loading plugin information are called `PluginDescriptor` and `AppDomainProxy`. `PluginDescriptor` class is used to describe the plugin, including displaying the plugin information in the user interface using XAML bindings. To retrieve information from the actual plugin assemblies, it uses application domains in combination with the `AppDomainProxy` class. The `AppDomainProxy` extends the `MarshalByRefObject` class, which means it can be used safely to perform cross-domain tasks (to learn more about application domains and dynamic assembly unloading, see Plugin management in Analysis). The process of retrieving plugin information is as follows: first, the `PluginDescriptor` creates a separate temporary application domain. Then, the `AppDomainProxy` is instantiated in this temporary domain, loads the given assembly and attempts to read its properties. Finally, the `PluginDescriptor` reads the retrieved values and unloads the temporary domain, thus removing the loaded assembly from memory.

This application manipulates plugins using the abstraction provided by the plugin interfaces located in the `PluginBase.dll` assembly.

5. Comparison with similar applications

This chapter discusses application functionally similar to this thesis, how they differ from it and how Universal Blog Manager (UBM) addresses their shortcomings.

5.1. Microsoft Live Writer 2011

Microsoft Live Writer (MLSW) is a tool designed for home users - its user interface is simple, but only has one goal: to allow users to create new content quickly. The application's user interface is basically one large post editor. This is very comfortable, because is allows users to start writing immediately after the application is started.

MSLW supports several blogging platforms out-of-the-box, such as WordPress, Blogger or SharePoint, but the list is not extensible.

5.1.1. How UBM differs

It is very difficult to compare UBM to MSLW, mainly because the latter is not a true blog “manager” - while it is able to load an existing post or page, this ability is hidden behind menus and additional dialogs. Articles can be deleted using MSLW, but with the same limitation - the ability is not readily available.

Working with other blog objects (such as comments) is omitted entirely, as are advanced features like displaying blog statistics or modifying blog settings. The application is only capable of working with one article at a time - it is impossible to edit multiple posts or pages at the same time.

Another advantage UBM has over MSLW is its ability to extend its list of supported blogging platforms using easy-to-create plugins. MSLW also supports plugins, but these are plugins of a different sort - they extend the application with additional functions, such as screen capturing and image upload.

There are several MSLW features that are not available in UBM yet, for example the ability to preview blog posts including the blog's visual theme or the ability to use spell check while writing an article. These are mainly “cosmetic” features and have no direct impact on the main application's functions. Spell-checking is a very handy

feature for a blogging application and has been placed on Universal Blog Manager's roadmap (see Future development in the following chapter).

Overall, we would describe MSLW as a tool useful for smaller and simpler blogs, mainly because of its simplicity. Managing a larger blog (or a customized CMS) is virtually impossible - these blogs usually employ heavily modified or custom-written blogging platforms. MSLW has no control over any potential customizations and is uncapable of interacting with custom platforms. Many basic functions (such as editing an existing older post) are not readily available. This makes for a cumbersome user experience in case of larger and more popular blogs.

5.2. BlogJet 2

BlogJet (BJ) belongs into the same category as the previously mentioned MSLW - its primary function is quick post creation, but its user interface is far less user-friendly. It uses the classic windows application UI composed of toolbars and menus (as opposed to Live Writer's Ribbon UI), which complicates navigation and orientation.

BJ currently supports 12 blogging platforms, such as WordPress, Blogger, LiveJournal and all platforms supporting MetaWeblog or MovableType APIs, but the list cannot be extended further.

5.2.1. How UBM differs

Since BJ is very similar to MSLW, it suffers from the same drawbacks - unintuitive user interface, inability to browse recent articles quickly and the overall inability to support platform-specific features. Additionally, it is a paid application with 30 day trial period.

BJ does not support any kind of plugins - all the features are either present in the application or simply not available. This is a great drawback for a paid application aiming at the quickly evolving area of blogging. The application is, just like MSLW, incapable of editing multiple articles at the same time.

BlogJet also contains a very interesting feature not available in UBM: the ability to create new articles cross-platform - to publish posts to several accounts at once. This feature has been implemented in previous versions of UBM, but had to be discontinued because of technical limitation when we decided to make the application plugin-extensible - since blog posts from different plugins have different

metadata, it would be impossible to cross-post anything but the post's title and body, which we considered virtually useless. This feature has been placed on the roadmap.

BJ is more suitable for experienced bloggers (its UI is not so intuitive as the MSLW's Ribbon UI), but managing a larger or custom system with it would range from very difficult to impossible (in case of custom platforms - BJ would simply not support them). Article deletion is hidden behind menus and features like post comments and statistics are omitted.

5.3. BlogDesk

BlogDesk (BD) is also similar in concepts to the two previously mentioned applications. The application is also focused on rapid content publishing and it only provides basic blog management functions not available directly from the main window. The user interface is more intuitive and less distracting.

BD is only capable of working with a single article at a time, does not handle diacritics in categories and was unable to retrieve the list of published posts for our WordPress hosted blog.

BD currently supports all blogging platforms using the MovableType and MetaWeblog APIs. This greatly limits the application's features, since many blogging platforms provide their own custom API with extended functionality.

5.3.1. How UBM differs

UBMs advantages over BD are virtually the same as with the previously mentioned applications - clearer user interface, extensibility and full support for true blog management (e.g. editing several posts/posts from different accounts at once, deleting posts in a single click etc.). BD also supports spell checking, although not as-you-write.

5.4. Conclusion

After evaluating several of the most popular and highly-regarded blogging tools, we concluded that UBM is a unique piece of blogging software, mainly because its focus is evenly distributed between content creation and content management. In our search we have not found a free Windows application with capabilities similar to those of UBM.

The inability to create, edit and moderate comments struck us as the most critical shortcoming of the mentioned application - blogging is about sharing opinions and the resulting discussion is one of the most interesting parts of it. A blogging tool that does not provide its users with the ability to interact with their readers fulfills only half of its responsibilities.

6. Conclusion

This chapter summarizes the thesis and describes to what extent the thesis goals were met. It also outlines potential for future development of the application discussed in this thesis.

6.1. Fulfillment of thesis goals

We attempted to solve three problems of the blogging world with this thesis: we wanted to create an application that is a) universal (supports working with any blogging platform), b) extensible (its features must be extensible using plugins) and c) user-friendly (its user interface must be as simple as possible).

After evaluating the result of our work and comparing it to similar applications, we can safely say that the first two thesis goals have been successfully met. Our plugin architecture is flexible enough to handle virtually any blogging platform back-end and present it to the user in a unified way. The main application is capable of handling plugins and extend its features.

By keeping the user interface as simple as possible, we managed to create an application where every important function is no more than three clicks away. By minimizing the number of various buttons and toolbars we also allow bloggers to write their articles in an environment that does not distract them.

We would estimate the degree to which the third thesis goal was met to 90%. There are several improvements to the UI that would make for an even greater user experience that did not make it into the current application version. These are highlighted in the following section.

6.2. Future development

There are several areas for improvement in the current version of UBM, ranging from plugin architecture through HTML rendering to user interface:

Since our implementation of plugin architecture turned out to be viable and powerful, future versions of UBM could incorporate plugins not only to communicate with blogging platforms, but also to provide users with additional functions that are plugin independent, such as searching Wikipedia or taking screenshots.

The Internet Explorer rendering core used in the current version is not entirely stable - it leaks memory over time and it is not very fast nor web standards-compliant. The logical next step would be to replace this rendering core with another one, such as Gecko or WebKit, both of which are considerably faster and more compliant.

To simplify plugin management, a central plugin repository could be constructed. This would make searching for a specific plugin much more comfortable. It would also allow for automatic updating of plugins.

Spell checking should be added to the application, allowing the users to spot and correct typos and misspellings before publishing their articles.

At last, plugins for more blogging platforms should be implemented, allowing UBM to reach a wider audience of users.

Appendix A. Table of links

Table A.1. Table of links

[1]	Blog: Merriam-Webster dictionary	http://www.merriam-webster.com/dictionary/blog
[2]	Wordpress home page	http://www.wordpress.com/
[3]	Tumblr home page	https://www.tumblr.com/
[4]	Blogger home page	http://www.blogger.com/
[5]	Posterous home page	https://posterous.com/
[6]	MovableType home page	http://www.movabletype.org/
[7]	Plugin: Merriam-Webster dictionary	http://www.merriam-webster.com/dictionary/plugin
[8]	MSHTML home page	http://msdn.microsoft.com/en-us/library/aa741317%28v=vs.85%29.aspx
[9]	Gecko Rendering Engine home page	https://developer.mozilla.org/en/Gecko
[10]	WebKit home page	http://www.webkit.org/
[11]	HTML Agility Pack home page	http://htmlagilitypack.codeplex.com/
[12]	XML-RPC specifications	http://xmlrpc.scripting.com/
[13]	WordPress XML-RPC specifications	http://codex.wordpress.org/XML-RPC_wp
[14]	Blogger Developer Center	http://code.blogger.com/
[15]	MetaWeblog API documentation	http://xmlrpc.scripting.com/metaWeblogApi.html
[16]	MovableType API documentation	http://developer.typepad.com/
[17]	LINQ-to-XML documentation	http://msdn.microsoft.com/en-us/library/bb387098.aspx
[18]	XML DOM documentation	http://www.w3.org/TR/DOM-Level-3-Core/introduction.html
[19]	Blogger API documentation	http://code.google.com/apis/blogger/docs/2.0/reference.html
[20]	Blogger API - AtomPub specifications	http://www.ietf.org/rfc/rfc5023.txt

[21]Blogger - OAuth 2.0 specifications	http://code.google.com/apis/accounts/docs/OAuth2.html
[22]Blogger - OAuth 1.0 specifications	http://code.google.com/apis/accounts/docs/OAuth.html
[23]Blogger - AuthSub specifications	http://code.google.com/apis/accounts/docs/AuthSub.html
[24]Blogger - ClientLogin	http://code.google.com/apis/accounts/docs/AuthForInstalledApps.html
[25]MSDN	http://msdn.microsoft.com/
[26]MSDN - Application domains	http://msdn.microsoft.com/en-us/library/cxk374d9%28v=vs.90%29.aspx
[27]MSDN - Application domains	http://msdn.microsoft.com/en-us/library/system.appdomain%28v=VS.90%29.aspx

Appendix B. Plugin Development Guide

B.1. Introduction

This document is part of the Universal Blog Manager bachelor thesis. The goal of this document is to provide programmers with guidelines concerning plugin development for the Universal Blog Manager (UBM) application.

Before you proceed reading this document, we strongly recommend to read the bachelor thesis, which will give you a clearer understanding of how UBM is implemented and what the main concerns are when writing plugins for the application.

B.2. Creating a simple plugin

Since UBM is a .NET application and uses .NET reflection to load plugins, all plugins must be valid .NET assemblies. This means that any new plugin must be written in a .NET language, such as C#, Visual Basic.NET or F#.

The process of creating a plugin starts by creating an empty **Class Library** project. It is necessary to create a class library project, because UBM only accepts plugins contained within a DLL file.

The second step in creating a plugin is to reference the library containing plugin interfaces. This library is called `PluginBase.dll` and is part of the UBM distribution. This library is crucial, because it contains all the classes and interfaces necessary to correctly declare and implement a plugin.

The third step is to create a class that implements the `IProvider` interface. Any class implementing this interface is considered a UBM plugin, which means it will be loaded into the application. Please note that a plugin implementing only this single interface has a very limited functionality - namely it provides some very basic information about the plugin itself, provides the user with the ability to add a new account using this plugin and log into it afterwards (the meaning of “log into” depends purely on the plugin itself). There will be no user interface for managing blog objects.

There are two important details that need to be implemented correctly in order for the plugin to operate: the plugin class must contain a

default parameterless constructor (this is required because of the way the main application loads plugins) and the `IProvider.LogOn` method must return a **new instance** of the plugin class. This is necessary because if the method returned "itself", it would be impossible for the user to work with two different accounts using the same plugin.

To allow the end user to actually interact with the blog account, more interfaces must by implemented be the plugin class. There are several interfaces supported by UBM, the most important of which is the `IPosts` interface. This interface allows the user to interact with blog posts (meaning anything the plugin considers a blog post). It inherits from the `IProvider` interface, therefore implementing the `IPosts` interface is enough to create a valid plugin. The same goes for the `IPages` interface, which allows the user to interact with blog pages.

There are several more interfaces available, all of which are described in detail in the programmer's documentation of the UBM bachelor thesis.

After you implement all the necessary methods and properties, your new plugin is ready to be compiled and tested. The next chapter describes how to debug plugins.

B.3. Debugging

To execute the code of your new plugin, you must first make sure that the plugin is being recognized by the UBM. Universal Blog Manager searches for plugins in the `[application executable path]\Plugins` folder. In order for your plugin to be loaded, you must copy the compiled plugin assembly into this folder and restart the main application. If everything went well (meaning the plugin was recognized and loaded), you should see your plugin listed in the About/About plugins window.

If your plugin was loaded, you can now interact with it and verify that it behaves correctly. The main application contains a special command line switch which, if present, will enable plugin debugging mode. In this mode, every encountered exception will be displayed with details, including the original error message and stack trace. Using this command line switch while debugging plugins is recommended, as it will make pinpointing potential issues much easier. To enable the plugin debugging mode, start UBM with the `/plugindebugmode` command line option.

B.4. Advanced features

There are several advanced features and interfaces available for implementation, such as UI describing methods and the `ICredentialsVerification` interface. This section discusses these advanced features.

B.4.1. UI describing methods

Several of the available plugin interfaces contain methods used to describe user interface necessary to perform certain tasks (e.g. display a blog post metadata or add a new comment).

You can identify these special methods by their name - all of them are named according to the following scheme: `Get[task]Form`, where `task` is the name of the task the method is used for (such as post display or page edit)

Most of these UI describing methods accept an argument. The value of this argument decides whether to display an empty form (meaning that the UI components will be “empty”) or to pre-fill the UI components with values taken from the argument object.

If the value of the argument is `null`, the UI components should be left empty. If the value is not `null`, the components should be initialized with values of the argument object.

To learn more about the respective methods, see the programmer's documentation.

B.4.2. Advanced interfaces

There are several advanced interfaces available to the programmers, such as the `ICredentialsVerification` interface, the `IArticleEmbeds` interface or the `IBlogMedia` interface. Implementing these interfaces is purely optional - these interfaces provide additional functionality and options to the user, but are not vital to the basic plugin operations.

An example:

Consider the `ICredentialsVerification` interface. If a plugin implements this interface, it means that the plugin is capable of verifying the stored user credentials - while registering the account, the main application allows the user to specify whether or not to

“remember” his/her credentials. If the user decides not to remember the credentials, a special window will be displayed every time the user attempts to log into this account, prompting the user to re-enter his/her credentials.

As you can see, the `ICredentialsVerification` interface is not necessary, but provides additional functionality (and in this case, security). It is recommended to implement as many interfaces as possible - more features means better user experience.

B.5. Exceptions

Exception handling should be implemented very carefully in your plugins - throwing unexpected exception types can easily crash the whole application. The `PluginBase.dll` contains three exception classes: there is the `abstract PluginException` class and two classes derived from it (`UserFriendlyException` and `InternalException`). In case you need to throw an exception **out of your plugin**, it is vital that you use one that is derived from the `PluginException` class. It is recommended not to declare your own exception types (even if they extend the `PluginException`) - use the `UserFriendlyException` class to signal problems the user should be notified about (such as network connection problems) and the `InternalException` to signal unexpected and critical problems.

Throwing a `UserFriendlyException` will cause an information box describing the problem to be displayed to the user, while throwing an `InternalException` will only display a generic error message. If plugin debugging mode is active (see above), every exception will be displayed with technical details, including error message and stack trace.

Appendix C. User Manual

C.1. Installation

C.1.1. System Requirements

Operating system: Windows 7/Vista/XP (both x86 and x64)

.NET Framework version: 3.5 SP1, including subsequent updates

Hard drive space: 15 MB

Other system requirement are the same as the requirements of the .NET Framework

C.1.2. Installation instructions

Universal Blog Manager 1.0 (UBM) is distributed in the form of a ZIP archive. To install the application, simply extract the content of the archive into a directory of your choice using 7zip or other ZIP compatible application. After the extraction is complete, you can launch the application by double-clicking the `UniversalBlogManager.exe` file.

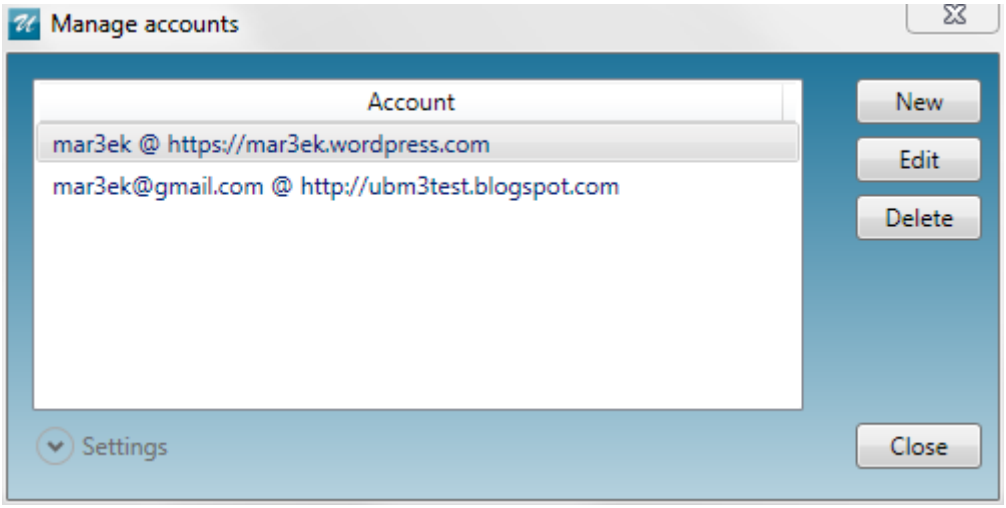
C.2. Typical usage scenarios

This chapter shows how to perform the most commonly used blogging tasks using UBM. We would like to point out that certain parts of UBMs user interface will change depending on the currently active plugin, but the general concepts remain valid for any plugin. This manual is written for the WordPress plugin, since it ships with UBM.

C.2.1. Registering an account

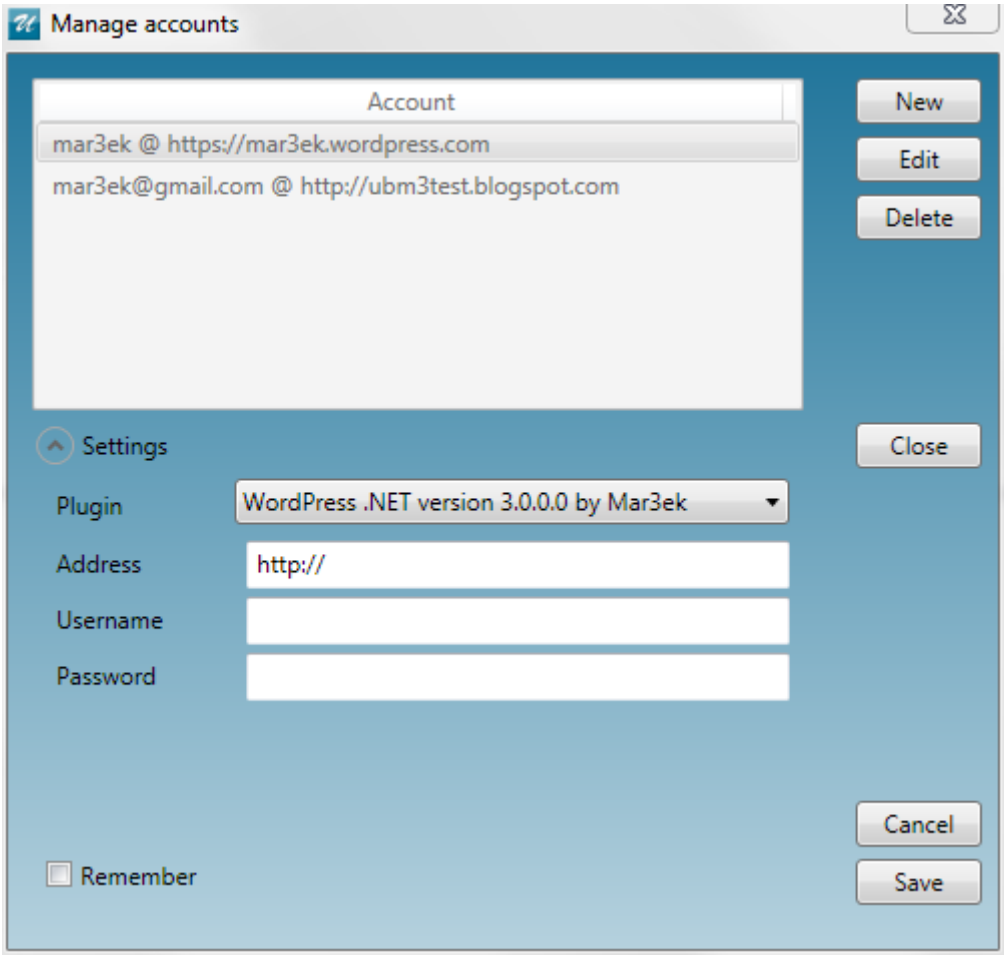
In order to interact with your blog account using UBM, you first need to register the account in the UBM. You can register any number of accounts in the Manage accounts window. This window is available from the main application window using the Accounts → Manage accounts (**Ctrl+M**) menu.

Figure C.1. Manage accounts window



The windows will list all currently stored accounts. You can add a new account using the New button or edit an existing account using the Edit button. In both cases the window will expand to display editing components.

Figure C.2. Manage accounts window - edit mode



The first step in adding an account is to choose a plugin. You need to specify a plugin that is capable of working with the blog provider you want to use (e.g. if you are adding a WordPress account, you must select a plugin for the WordPress platform). Next step is to input information necessary for the plugin to access your account. This might include blog address, username, password and/or other credentials.

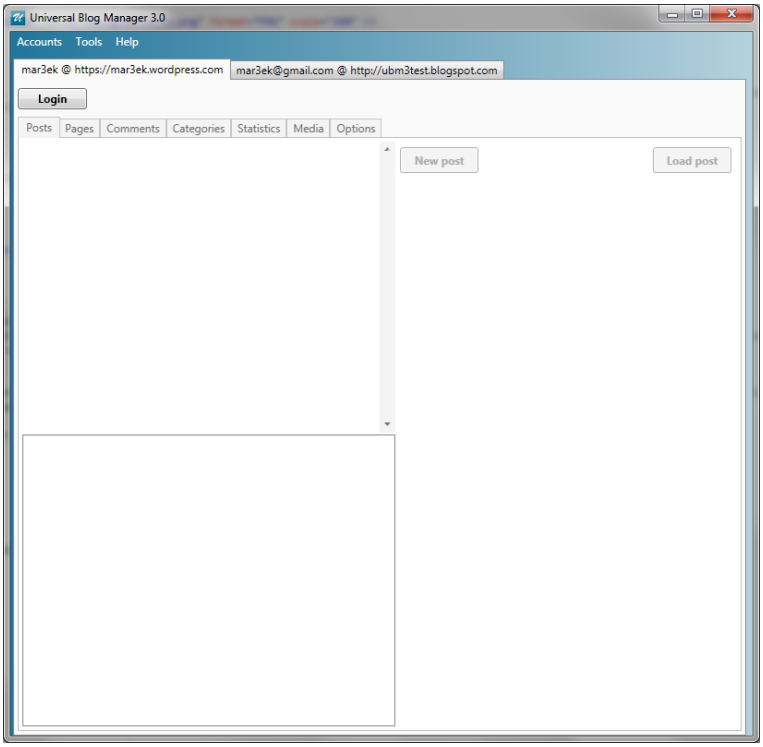
After you filled all the required fields, you can save the account using the Save button. Verification of the supplied credentials is performed at this step - if the plugin cannot access the account, you can double-check your credentials.

The same concepts apply to account editing, with a single exception - you cannot change plugin of an existing account.

C.2.2. Logging into an account

After an account has been registered, the main application will display it in a tabbed window - a tab for each account.

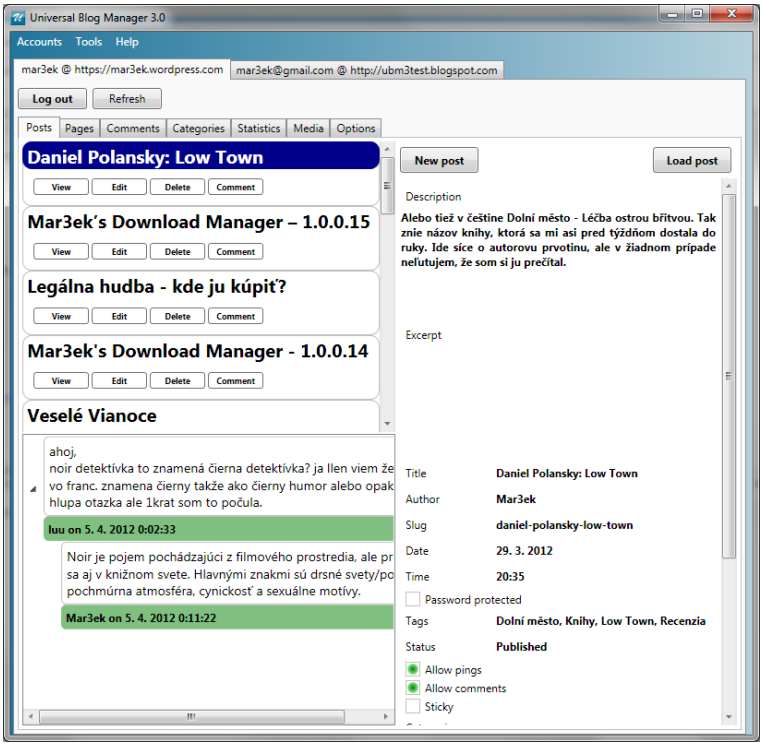
Figure C.3. Main window - account tabs



Before you can interact with your account using UBM, you must log into it. Login is performed using the Log in button located in the upper left corner of the account tab. Please note that logging into an account might take as long as several minutes, depending on your Internet connection speed and the amount of data the plugin loads.

The rest of the account tab will be unlocked after a successful login. You can see latest blog posts, pages, statistics or other blog data in the account tab. The account tab is further divided into separate tabs to simplify orientation. Each tab corresponds to a single “facet” of managing a blog - blog posts, blog pages, comments etc. You can manage (view, delete) blog data in the account tab.

Figure C.4. Main window - logged in

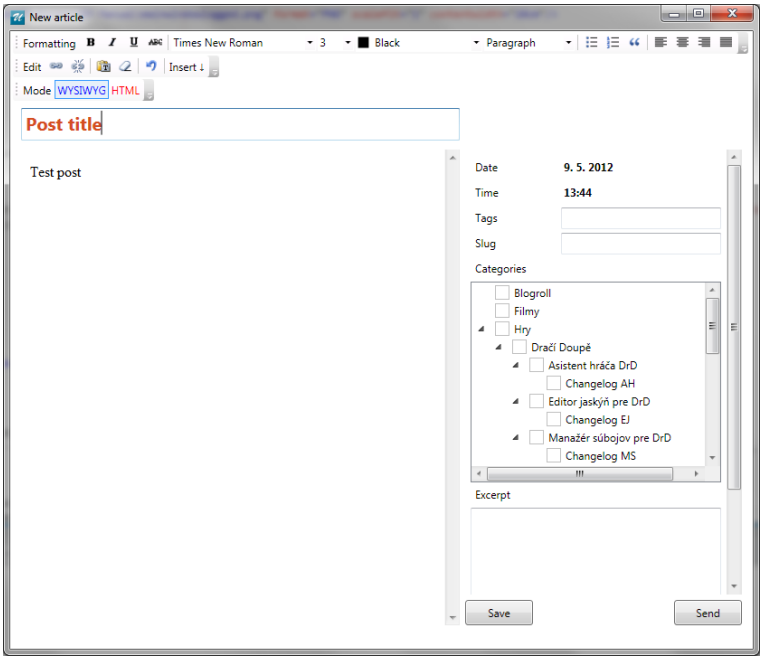


C.2.3. Creating blog content

The account tab allows you to view and delete existing blog content. Creation of new content or editing is performed in separate windows designed specifically for the purpose. We will demonstrate this on the task of creating a new blog post.

To start creating a blog post, you must first be logged into a blog account. Then selected the Posts tab in the account tab and click the New post. A new windows will open, containing a blog post editor.

Figure C.5. Blog post editor



Here you can write your new blog post. A WYSIWYG HTML editor is available in this window, so you can write rich HTML blog posts using a visual editor. Right window pane contains metadata editor - this area will change depending on the plugin and will allow you to specify various platform-specific blog post attributes, such as post categories and tags.

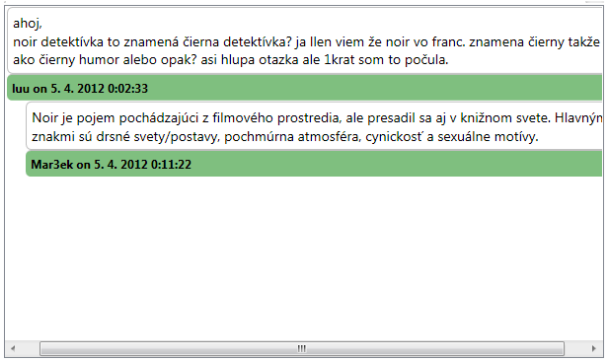
After you have written your post, you can either save it locally or send it to the blog provider (“publish it”). Local copy is saved using the Save button, the post is published using the Send button.

Post editing is performed in the same window. The same principles also apply to blog pages and comments, although the window for creating comments is much simpler.

C.2.4. Moderating comments

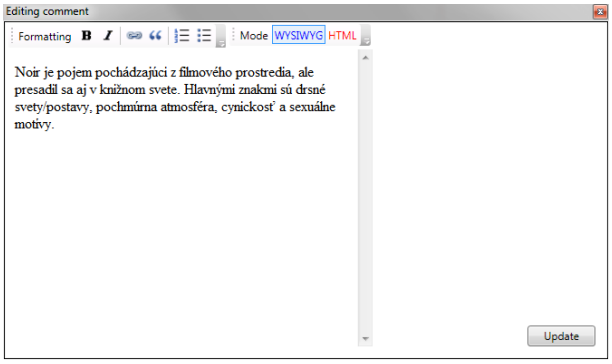
Blogs usually allow visitors to comment on articles. UBM allows these comments to be displayed in a hierarchical manner (where applicable) and add, edit, delete and moderate them.

Figure C.6. Displaying threaded post comments



You can manage a comment by right-clicking it. A context menu appears, where you can select what you want to do with that particular comment. The WordPress plugin allows for comment editing, replying, deleting and moderation (changing comment status).

Figure C.7. Comment editing window



The comment editing window contains a simplified WYSIWYG editor. Right window pane provides access to the extended properties of a comment (where applicable).

C.2.5. Managing plugins

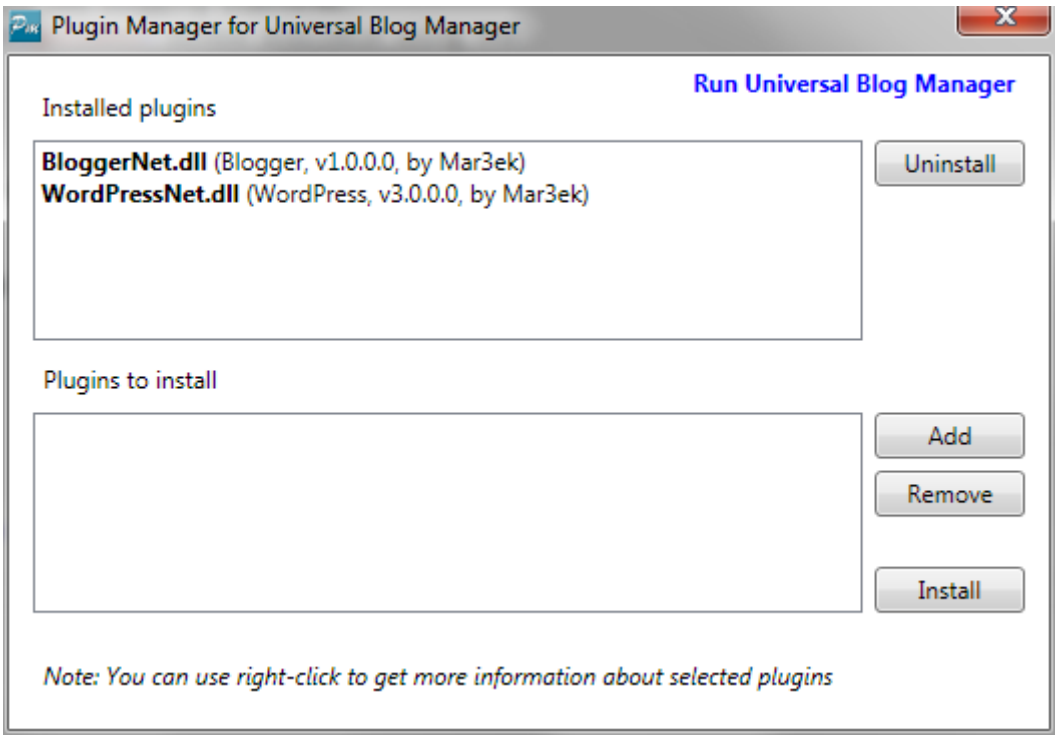
UBM is a plugin based application, which means it can be easily extended to work with new blogging platforms. You can find the list of currently installed plugins in the About plugins window available using the Help → About plugins menu. This window also displays various information about installed plugins, including author and version.

Installation of new plugins and unistallation of old plugins can be performed using the plugin manager tool that ships with UBM. You can start the plugin manager using the Tools → Plugin manager menu item.

Note

The plugin manager requires administrative right to function properly.

Figure C.8. Plugin manager



You can uninstall an installed plugin by selecting it and clicking the Uninstall button. Installation of a new plugin is performed using the Add button. You must first select a plugin to install, then the plugin manager will verify that the selected file is a valid plugin and if everything is in order, the installation is finished by clicking the Install button.

Note

Uninstalling a plugin requires all running instances of UBM to be terminated.

Note

You can install several plugins at once by first adding them one by one and then clicking the Install button.

Important

Only install plugins you trust - once a plugin is installed and an account is registered using this plugin, it will have access to your user credentials!

Appendix D. Content of the enclosed CD

The enclosed CD contains the following:

- Electronic version of the bachelor thesis (this document) in the form of a PDF document
- Distribution package - a ZIP file containing the compiled project
- Source code package - a ZIP file containing the complete project source code
- Programmer's documentation - a CHM (compiled HTML help file) containing the programmer's documentation