# Predicting bugs and issues

## with automated code reviews

**Master Thesis**

Master Software Engineering, University of Amsterdam


*Author*

ing. Henk Bosman

henk@ferion.nl

*Supervisor*

Dr. Jurgen Vinju

jurgen.vinju@cwi.nl


29 August 2013

UNIVERSITEIT VAN AMSTERDAM

CWI

# Abstract

*To speed up development of existing or new software third party libraries are often used. Basic third party library application programming interface (API) calls are normally documented. More advanced API calls and combinations with API calls are often not documented which leads to trial and error programming. This research implements an automated code review system employing different algorithms to find anomalies in API usage. These anomalies are expected to correspond to bugs or issues. Using 180 software projects with a combined total of 236000 lines of code the effectiveness of combining multiple algorithms is demonstrated.*

# Preface

Being a professional programmer for 15 years, I have reviewed and discussed many source code solutions with other programmers. The Master Software Engineering at the University of Amsterdam allowed me to see everything I learned to date in a different perspective. This new insight has led to the topic of this Master's thesis and the opportunity to give something back to my employer who made it all possible. I want to thank my teachers Jan van Eijck, Tijs van der Storm, Hans Dekkers, Hans van Vliet and Paul Klint for all the lessons learned. I am especially grateful to have Jurgen Vinju as my teacher, my inspiration and my guidance for this Master's thesis. My wife and two daughters, now almost five and seven, were my greatest support. They have given much of their life to allow me to follow my dream. Special thanks to Chris Burtt-Jones for proofreading this thesis and giving suggestions for improvement.

# Table of Contents

# 1 Introduction

### 1.1 Motivation

Bugs in software are so detrimental that the costs for the U.S. economy are estimated at $59.5 billion annually [13]. It is likely that some of these bugs are linked to wrong library usage. To find bugs, code reviews are introduced within development teams. Dunsmore et al. [10] found that during code review a bug is initially found every 10 minutes, dropping steeply after 60 minutes. Finding one bug every 10 minutes seems productive but study shows [11] [12] that between 15% and 30% of these found bugs are false positives making code reviews time intensive. This research focuses on less time-consuming automating code reviews. Found anomalies in framework API usage are expected to correspond to bugs or issues. Programmers can focus on areas pointed out by the automated code reviewing tool.

### 1.2 Anomalies, bugs and API usage

During testing the difference between expected behavior and actual behavior is called an anomaly. The reason for this anomaly can be an external factor or a bug in the software This research focuses on finding anomalies within source code, deviations between expected and found lines of code; specifically lines of code containing Application Programming Interface (API) statements. The goal is to limit the number of false positives as much as possible but it is still up to the programmer to determine if the anomaly really relates to a bug or issue.

### 1.3 Approach and contribution

To be able to identity API usage anomalies the norm needs to be determined. Analyzing existing source code makes this possible. Zhenmin Li and Yuanyuan Zhou developed PR-Miner [3] which analysed a target project to gather data about normal API usage to find anomalies within the same project. Zeller et al. [19] mined existing source code from other software to find anomalies in selected development projects. This research will also mine existing source code from other software to find anomalies in a selected project. A prototype will be developed to verify the theory.

PR-Miner and other research showed that using existing source code to find anomalies in selected projects can be rewarding. False positives are pruned using a combination of violation rules and ranking every anomaly. This research combines the output of the algorithms used in different related research in order to reduce the number of false positives. Specifically the algorithms used in PR-Miner and the research done by Engler et al. [8]. Engler analysed source code for common pairs based on the principle "*function A is always followed by function B*". If *function A* is found and *function B* is missing it is most likely an anomaly. PR-Miner searches for recurring sequences and matches these with other source code to find deviations. A third algorithm is introduced matching complete API usage sequences to limit false negatives. The output of these three algorithms are combined. Only anomalies found by at least two of the three algorithms are reported to the user in order to reduce the number of false positives. Where needed the algorithms are adapted in order to deal with the object oriented language C#.

### 1.5 Research question and method

A lot of source code already exists and contains solutions for problems the programmers faced. Sometimes the problems still exists in the source code leading to unexpected behavior. "*Can anomalies in source code be found automatically and with high accuracy using other source code?*" is the research question for this master's thesis. Is it possible to for the solutions the programmers applied in the source code to be used to identify problems in other source code? Research in this area is already done but focusing on a single approach. One problem is the amount of false positives that these approaches generate. "*Can the amount of false positives be reduced when combining the results of different approaches?*" is the question that will be researched in this thesis. Reducing the amount of false positives is important to reduce the time investment needed to verify the proposed anomalies.

In order to verify the theory and answer the research questions a prototype is built. The projects developed within the company where this research is done will be used to find anomalies. Open source projects are mined using the prototype. The prototype will use the mined source code to find anomalies in the selected projects. The output of all three algorithms combined together with the final output are written to disk. All outputs are manually verified for false and true positives checking every marked anomaly to see if it is related to any bug or issue. Comparing the results of the individual output with the combined output will show how effective combining the output of multiple algorithms is at reducing the number of false positives.

# 2 Problem domain

### 2.1 False positives

A good way to describe a false positive is as a false alarm. Emails incorrectly marked as spam or a virus scanner reporting a non-existing virus are some examples. Davis [14] researched the relation between usefulness and usage of software. His research showed a strong relationship between usefulness and usage. Software that generates many false positives is quickly perceived as not usable and time consuming, effectively leading to reduced usage. As this research focuses on finding anomalies, false positives are bound to happen. It is important to reduce the number of false positives in order for programmers to find the software based on this research useful.

### 2.2 False negatives

Finding the anomalies connected to a bug is a difficult task. If anomalies are found it is still unknown whether more anomalies still exist. Undiscovered anomalies in the source code are called false negatives. Every framework API has its own unique possible anomalies making it important to use techniques that can find anomalies for different situations.

```
        public void ChangeRegionalSettings(string url, uint culture, bool time24){
            SPSite spSite = new SPSite(url);
            SPWeb spWeb = spSite.RootWeb;
            spWeb.ChangeRegionalSettings(culture, time24);
            spWeb.Update();
            spSite.Dispose();
        }

        public void ChangeRegionalSettings(uint culture, bool time24){
            SPSite spSite = SPContext.Current.Site;
            SPWeb spWeb = spSite.RootWeb;
            spWeb.ChangeRegionalSettings(culture, time24);
            spWeb.Update();
            spSite.Dispose();
        }
```

**Figure 2.1 First method correctly disposes the *SPSite* object, the second method should not dispose the *SPSite* object**

### 2.3 API Usage

The SharePoint API [21] has specific dos and don'ts, some of which are documented while others are only found by trial and error. For example a *SPSite* object needs to be explicitly disposed after it is used in order for the SharePoint COM object to close the associated SQL connections. Yet not all *SPSite* objects should be disposed. When the object is also used within the framework, disposing will result in unexpected behavior. Figure 2.1 shows two methods where the first method correctly disposes the *SPSite* object. The second method tries to dispose a *SPSite* object belonging to the framework which will result in unexpected behavior. While Visual Studio tools exist to scan for undisposed *SPSite* objects they will accept the second method resulting in a false positive.

```
        public void AddCulture(SPWeb spWeb, string culture){
            spWeb.AddSupportedUICulture(new System.Globalization.CultureInfo(culture));
            spWeb.AddProperty(culture, "Supported");
            spWeb.Update();
        }
```

**Figure 2.2 This method has a mix of static and dynamic API usage sequence**

### 2.4 Execution ordering

Some API calls need to be executed in a fixed order. For example an object needs to be created first before it can be disposed. Other API calls can be executed without a fixed sequence. Most API calls are executed relative to another API call. The methods in figure 2.1 contain API calls which need to be executed in a fixed order. Figure 2.2 shows a method which executes two API calls before executing the update function. The order in which *spWeb.AddSupportedUICulture* and *spWeb.AddProperty* are executed is not important, but both must be executed before *spWeb.Update.* This semantic relation and restrictions between statements can only be inferred from the API documentation.

```
public bool ChangeRegionalSettings(string url, uint culture, bool time24){
     SPSite spSite = new SPSite(url);
     SPWeb spWeb = spSite.RootWeb;
     if (spWeb.UserIsWebAdmin){
         spWeb.ChangeRegionalSettings(culture, time24);
         spWeb.Update();
         return true;
     }
     spSite.Dispose();
     return false;
 }
```

**Figure 2.3. Method has two execution paths of which one is not disposing the *SPSite* object**

### 2.5 Control flow

A control flow graph contains the order in which statements are executed. A control flow statement is a statement that can change the control flow based on an input. For example an *if-then-else* statement has two execution paths based on the outcome of the *if* statement. When the outcome is *true* the *then* is followed whereas with *false* the *then* is skipped and the *else* is followed. Figure 2.3 shows a method with two execution paths. When *spWeb.UserIsWebAdmin* is *true* the regional settings get changed and the method returns *true* without disposing the *SPSite* first. When *spWeb.UserIsWebAdmin* is false the *SPSite* object gets disposed and the method returns *false*. Every control flow statement generates a possible extra execution path. Every execution path needs to be analysed individually to find anomalies in the API usage.

```
public UserPrincipal GetUserPrincipal(){
     SPUser spUser = SPContext.Current.Web.CurrentUser;
     PrincipalContext pc = new PrincipalContext(ContextType.Domain);
     UserPrincipal user = UserPrincipal.FindByIdentity(pc, spUser.LoginName);
     pc.Dispose();
     return user;
 }

 public UserPrincipal GetUserPrincipal(){
     SPUser spUser = SPContext.Current.Web.CurrentUser;
     PrincipalContext pc = new PrincipalContext(ContextType.Domain);
     UserPrincipal user = UserPrincipal.FindByIdentity(pc, spUser.Name);
     pc.Dispose();
     return user;
 }
```

**Figure 2.4 Interspersed API usage between the SharePoint API and the Active Directory API**

### 2.6 Interspersed API usage

SharePoint is a web-based platform and uses the ASP.NET framework. SharePoint promotes the use of external applications which include their own framework APIs. Developers frequently introduce their own framework with recurring functions to speed up and centralize development. The SharePoint API is therefore often interspersed with other framework APIs. The first method in figure 2.4 shows interspersed framework API usage between the SharePoint API and the Active Directory API. The SharePoint *SPUser* object is used as an input for the Active

Directory *UserPrincipal* object. The *spUser.LoginName* is a correct input for the *UserPrincipal.FindByIdentity* method in this case. Method two in figure 2.4 is slightly different yet is incorrect, since *spUser.Name* is not a valid input for the *UserPrincipal.FindByIdentity* method as it contains the real name instead of the login name.

```
public bool ContainsItem(SPList spList, string title){
    for (int i = 0; i < spList.Items.Count; i++){
        if (spList.Items[i].Equals(title))
            return true;
    }
    return false;
}
```

**Figure 2.5 Performance problems with SharePoint 2007 and not with SharePoint 2010**

### 2.7 Framework versions

As software gets improved, new framework versions are introduced. New API calls get introduced while existing API calls get deprecated. Some existing API calls are improved to fix problems or get new functionality. Figure 2.5 shows a method that results in performance issues in SharePoint 2007 depending on the size of the list. The SharePoint 2007 framework retrieves all items from the target list when executing *spList.Items[i]* with every pass in the loop. As lists in SharePoint can contain millions of items, the method in figure 2.5 could be a real performance issue. The SharePoint 2010 framework introduced a fix allowing a hard limit to be set on how many items are retrieved when executing *spList.Items[itemId]* along with other internal performance enhancements. This eliminates the problem shown in figure 2.5.

```
public List<string> GetTitles(SPList spList){
    List<string> returnList = new List<string>();
    foreach (SPListItem item in spList.Items){
        returnList.Add(item.Title);
    }
    return returnList;
}

public List<string> GetTitles(SPList spList){
    return (from SPListItem item in spList.Items select item.Title).ToList();
}
```

**Figure 2.6 Methods have the same outcome but uses different approaches**

### 2.8 Programmer expertise

The company where the research is performed has several in-house developers. External developers are hired when needed. Their expertise ranges from junior to senior programmers. This makes code reviews difficult within a team. While it is educating for a junior programmer to review the code of a senior developer, the chances are he will find no bugs in the code. Another problem is the difference in quality of the code produced. For example a senior programmer uses three methods with a few lines of code while a junior programmer uses one method with many lines of code. Figure 2.6 shows different approaches for the same task. While both methods are correct and accomplish the same task, they cannot be used to verify each other.

```
public SPList GetList(SPSite site, string listName){
    foreach (SPWeb web in site.AllWebs){
        SPList list = web.Lists.TryGetList(listName);
        if (list != null)
            return list;
    }
    return null;
}
```

**Figure 2.7 Generic method for retrieving a *SPList* object**

### 2.9 Generic methods

As the source code grows, so does the need for generic and reusable methods. Certain tasks are performed multiple times but in different sections of the source code. These tasks can be extracted and made generic to be reused in different sections of the source code. This is an optimization that happens a lot and effectively reduces the lines of code within a project. Figure 2.7 shows such a method. The method needs a *SPSite* object to give it context to run in. As it is an input value for the method rather than an object created within the object, it is impossible to infer from the source code if the *SPSite* should be disposed within the method or not. The SharePoint documentation dictates that every *SPSite* object needs to be explicitly disposed but disposing the object within the method shown in figure 2.7 would result in unexpected behavior in the parent method which called this method.

### 2.10 Tool usage

The dispose checker described in section 2.3 is a useful tool to detect possible resource leaks within the software. Not disposing an *SPSite* object leads to resources not being freed for other tasks. The dispose checker checks for these anomalies during the build process of the source code. In certain situations, like the one described in section 2.3 or in section 2.9, it will generate false positives. Despite its usefulness and easy usage, the dispose checker is turned off within the company where this research is done due to these false positives. For any tool to be accepted and used it is important to limit the number of false positives.

# 3 Methodology

### 3.1 Related work

Various code mining techniques and theories have been proposed to find bugs in API usage. Mining large portions of source code has become a viable option as computing power has increased and continues to increase. Engler at al. [8] introduced pair mining. Their work is based on the programmer beliefs "*A call to 'a' followed by a call to 'b' implies the programmer may believe they must be paired, but it could be a coincidence".* These programmer beliefs are inferred from analyzing the source code and are used to detect anomalies. Any source code that deviates from those beliefs is marked as an anomaly. Hovemeyer and Pugh [16] analysed different approaches for finding bugs. Their findings are that even the simplest anomaly detection method is successful in finding anomalies. It is the manual evaluation of every anomaly that is a subjective and time consuming process. Li and Zhou [3] extract frequent recurring patterns from source code and use those patterns to find anomalies. These patterns are not limited to pairs like the work of Engler et al. Their approach is useable without prior knowledge about the software and can used in conjunction with different programming languages. Inspired by the use of frequent recurring patterns in PR-Miner Wasylkowski, Zeller and Lindig[17] propose to take ordering also into account as well. They used a ranking method inspired by the clustering algorithm used by Dickinson et al. [18] to limit false positives. Zeller et al. [19] used a lightweight source code parser to mine an impressive 200 million lines of code spanning 6000 open source projects. Using a fixed threshold, patterns are identified and used to find anomalies in selected projects. They report a rate of 25% true positives using their method.

### 3.2 Selected approach

The problems described in section 2 require a generic approach. Engler et al. reported good results using pair mining dealing with the problems as described in section 2.3. As API usage often spans more than two statements, the method used in PR-Miner showed promising results. To deal with the ordering problems described in section 2.4 this research uses the approach used by Wasylkowski, Zeller and Lindig.

To limit the number of false positives I propose to combine the output of both methods and present only the anomalies found by both methods. Zeller et al. reported that 25% of their results were actually a bug resulting from the reported anomaly. It is expected that combining both methods will increase the percentage of true positives.

Combining the output of both methods does raise a new problem. As both methods employ different techniques, it is likely both techniques will find different anomalies causing limited results in the combined output. To counter this problem I propose an extra algorithm where methods are compared and the differences are registered as potential anomalies. This algorithm will resolve the problem described in section 2.2, reducing the number of false negatives but with an increase in false positives. Combining the results with the other two algorithms will effectively filter out the false positives while decreasing the false negatives.

Zeller et al. used a lightweight source code parser to enable very large corpus analysis. This research focuses on one framework which limits the corpus. While some frameworks are present in most software they are also the often the most documented and refined frameworks. This makes it more likely that bugs are to be found in API usage of less used and often specialized frameworks. Less used frameworks also means a lesser corpus available to analyse. Because the corpus is expected to be smaller than used by Zeller et al. it is important to analyse every statement in order to find most anomalies. This makes Zellers lightweight source code parser not viable for this research. As such a traditional approach is used parsing all source code allowing semantic analysis of the source code.

### 3.3 Used corpus

The algorithms selected in the previous section rely on source code to find anomalies. Engler et al. used the source code from the same software which was also selected to find anomalies in. This approach has the problem that existing bugs in the source code are used to find anomalies in other parts of the source code leading to false positives. When working with framework APIs, it is particularly likely that certain usage is perceived correct within the development team while it is actually incorrect usage. Another issue with this approach is the limited corpus available, in particular for smaller projects. Engler et al. successfully demonstrated the theory using a project with approximately 300k lines of code. Projects with considerably fewer lines of code cannot benefit from using the approach of Engler et al. Zeller et al. proposed the use of source code from other projects as input for the algorithm. This approach is scalable and results in an objective corpus not influenced by prejudices of the development team. The size of the corpus is limited to all the source code produced in the world using the selected framework API. This research uses the approach employed by Zeller et al. allowing users to add new source code to the corpus independent of the source code the user wants to find anomalies in.

# 4 Processing source code

### 4.1 Semantic analysis

Before the corpus can be filtered, traced and used as input for the algorithms, the source code needs to be parsed to enable semantic analysis. An existing solution developed by Microsoft is used to parse the source code. Microsoft introduced the Roslyn framework [22] in October 2011. In the past the Microsoft compilers acted as black boxes. Roslyn changed this and opened a public API to provide extension points in the C# and VB language services. From the Microsoft Roslyn blog: "*This opens up new opportunities for Visual Studio extenders to write powerful refactorings and language analysis tools, as well as allow anyone to incorporate our parsers, semantic engines, code generators and scripting in their own applications*.". Microsoft Roslyn allows us to focus on the actual research without the need to develop own solutions to parse and analyse source code.

### 4.2 Tracing methods

The scope of every trace is the method where the statements are found. Scoping the trace increases the speed of analyzing the source code. Section 2.8 describes how some programmers may split one task into multiple methods.

Limiting the trace to a method is needed to avoid any methods to be traced more than once. This would result in some API usage patterns incorrectly being marked as a strong pattern.

#### 4.2.1 Filtering statements

During tracing, non-related statements are filtered. Semantic analysis of every statement is performed to allow accurate filtering. Users select a framework of interest which is used as filter. The selected framework can be used to filter out any statements which are not related to the selected framework API. This operation reduces the number of statements and speeds up the process of finding anomalies.

Some statement types are used more often than others. When statement types do not occur often it is likely that the corpus is not sufficient to accurately find anomalies. This is why the following statement types are selected and others are filtered out:

- Invocations
- Variable modifiers
- Null checks

Examples are of these statement types:

- *Object.Dispose()* (Invocation)
- *Object.Visible = true* (Variable modifier)
- *Object == null* (Null check)

```
public string GetTitle(){
    Uri url = HttpContext.Current.Request.Url;
    string returnValue = "";
    using (SPSite site = new SPSite(url.ToString())){
        returnValue = site.RootWeb.Title;
    }
    return returnValue;
}
```

**Figure 4.1 Method containing statements using the SharePoint framework API**

To demonstrate the filtering of statements, consider figure 4.1 which shows a method where the statements are marked which are using the SharePoint framework API. After the method is traced the control graph contains the following statements:

*using (SPSite site = new SPSite(url.ToString()))*
*returnValue = site.RootWeb.Title*

The *using* statement is a special statement that automatically disposes the *SPSite* object after the statements are executed between the brackets. The *SPSite* object *site* is not available outside the *using* brackets. As development tools catch these errors there is no need to register the brackets in order to check if the *site* object is used outside the *using* brackets.

### 4.2.2 Method control flow

Section 2.3 pointed out that methods can have multiple execution paths. Some statements can alter the execution path within a method depending on the input for the statement. The following statements, used often within C#, can change the execution path:

- *If-then-else*
- *Switch-case-default*
- *For*
- *Foreach*
- *While*
- *Try-catch-finally*

Notice how the *for*, *foreach* and *while* statements are also listed as statements which can change the execution path. All three statements create a loop within a method. The statements all use an input to determine if the loop should be repeated. If the input is false before the loop is executed then the loop is skipped. Skipping the loop effectively creates two different execution paths: one execution path executing the statements in the loop, and the second execution path skipping the statements in the loop. The statements within loops are traced once and are treated like normal statements. Tracing the statements more than once within loops holds no added value in finding anomalies. Also it is, in most cases, impossible to infer from code how often the statements in the loop are iterated.

```
public string UpdateSite(string url, SPUser newAuthor){
    if (string.IsNullOrEmpty(url))
        return string.Empty;
    SPSite spSite = new SPSite(url);
    SPWeb spWeb = spSite.RootWeb;
    string currentVersion = (string)spWeb.GetProperty("Version");
    if (currentVersion.Equals("1")){
        spWeb.SetProperty("Version","2");
        spWeb.Author = newAuthor;
        currentVersion = "2";
    }
    spWeb.Update();
    spSite.Dispose();
    return currentVersion;
}
```

**Figure 4.2 Example method containing multiple execution paths with statements using the SharePoint API**

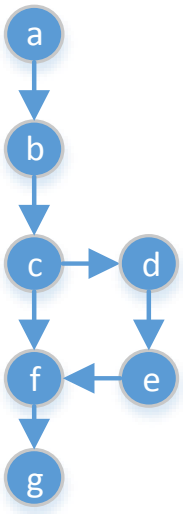


**Figure 4.3 Control flow graph**

To illustrate the conversion from a method to a control flow graph, consider the method shown in figure 4.2. The statements which use the SharePoint API are underlined. All the other statements are disregarded during tracing. Figure 4.3 shows the control flow graph after the method is traced. Two execution paths can be identified *{a,b,c,f,g}* and *{a,b,c,d,e,f,g}* which translate into:

```
SPSite spSite = new SPSite(url);
SPWeb spWeb = spSite.RootWeb;
string currentVersion = (string)spWeb.GetProperty("Version");
spWeb.Update();
spSite.Dispose();
```

```
SPSite spSite = new SPSite(url);
SPWeb spWeb = spSite.RootWeb;
string currentVersion = (string)spWeb.GetProperty("Version");
spWeb.SetProperty("Version","2");
spWeb.Author = newAuthor;
spWeb.Update();
spSite.Dispose();
```

### 4.3 Normalizing statements

This research relies on mining existing source code to use the data to find anomalies in selected projects. Lines of code need to be compared to achieve this. Invocations like *Update()* or *Dispose()* are easily matched as their name is fixed but variables and objects have unique names. The *SPSite* object is called *spSite* in figure 4.1 whereas it is called *site* in figure 4.2 yet they are of the same type. The problem grows when there are more of the same variables and objects in the same method; they all have to have a unique name. To resolve the problem the identifier names are renamed using a combination of their type and an incremental number. In the above example the *SPSite* objects in both methods are renamed into *SPSite1*. A second *SPSite* object within a method would become *SPSite2* and so on.

Static values like the text *"Version"* used in the method in figure 4.2 do not have an identifier name. It is likely that the content is linked to the environment where the software is used. This makes the content not interesting for finding anomalies yet it is important to register that some content was used in the statement. As *Literal* is the term used to describe these values a combination of the type of the content and the text "Literal" is used to describe the content. In the case of *"Version"* it will become *StringLiteral*. For example:

*spWeb.SetProperty("Version","2");*
will become
*SPWeb1.SetProperty(StringLiteral, StringLiteral)*

### 4.4 Hashing statements

When dealing with a large number of sequences and statements performance and memory becomes a problem. Especially with an algorithm like Apriori which is described in section 5. To reduce memory usage and to speed up the algorithms, every statement is hashed into a unique number. This allows statements to be compared quickly and efficient as only numbers have to be compared. Theoretically it is possible for two different statements to have the same hash. A hash is a signed int32 which ranges from $-(2^{31})$ to $(2^{31})-1$. The odds that two different statements have the same hash and are compared is very small and does not outweigh the performance hit from not using hashing.

# 5 Algorithms

### 5.1 Combining the output

As described in section 3.2 three different algorithms are to be used to find anomalies. To limit false positives and false negatives the output of the three algorithms is combined. Every algorithm gives every anomaly found a score as described in the following sections. In related research, thresholds are employed to prune false positives. Use of thresholds is based on the theory that the anomaly score has a correlation with the likelihood of the anomaly being related to an issue or a bug. Whether or not this theory is correct, a question arises how many true positives are ignored as a result of having a score lower than the threshold. While the related research does not answer this question it is still important to limit the number of false negatives. This research combines the outcome of all three algorithms and only presents anomalies found by two or all three algorithms. An anomaly which would normally be ignored due to a threshold is still presented to the user when it is also found by another algorithm. This allows for lower thresholds while retaining the ability to prune false positives.

### 5.2 Control flow graph and sequences

In order for the algorithms to find anomalies they need data. This data is stored in control flow graphs after the methods are traced as explained in section 4. This is a resource-effective way to handle and storing the data. The algorithms on the other hand use sequences, an ordered list of statements, and not control flow graphs. When a method is analysed using an algorithm the control flow graph containing the traced data is transformed into one or more sequences. The number of sequences has a direct relation with the number of execution paths possible in the control flow graph. These sequences are used to find patterns or anomalies depending on the origin of the control flow graph. Source code gathered for mining is used to determine the norm. Source code originating from a selected project is checked for anomalies using the mined source code.

| **Pair combination** | **Template type** |
|---|---|
| Invocation -> Invocation | Invocation <*a*> is followed by invocation <*b*> |
| Invocation -> Variable modifier | Invocation <*a*> is followed by variable modifier <b> |
| Invocation -> Null check | Invocation <*a*> is followed by a null check |
| Variable modifier -> Invocation | Variable modifier <a> is followed by invocation <b> |
| Variable modifier -> Variable modifier | Variable modifier <a> is followed by variable modifier <b> |
| Variable modifier -> Null check | Variable modifier <a> is followed by a null check |
| Null check -> Invocation | A null check is performed before invocation <a> |
| Null check -> Variable modifier | A null check is performed before variable modifier <a> |

**Figure 5.1 Pair templates**

### 5.3 Pair comparison

As previous mentioned, the work of Engler et al. is based on the beliefs of programmers. The source code is checked for violations of those belief. Generic rule templates are used to restrict the scope of the beliefs. An example mentioned in the paper is the template "<*a*> *must be paired with* <*b*>". Possible values for <*a*> and <*b*> could be creating an object and disposing the object afterwards.

The templates used in this research can be inferred from the filters applied during tracing. This is where this research differs from Engler et al. as he specifically created templates for a purpose. Nine different combinations are possible using the three statement types mentioned in section 4.2.1. One combination, namely two *null* checks on the same object, is redundant but violating the rule does not result in an anomaly. Figure 5.1 shows a table containing the possible combinations and their templates.

```
public string GetLogin(){
    SPWeb spWeb = SPContext.Current.Web;
    string returnValue = "";
    if (spWeb != null){
        SPUser user = spWeb.CurrentUser;
        returnValue = user.LoginName;
    }
    return returnValue;
}
```

**Figure 5.2 Example method**

### 5.3.1 Finding pairs

Data gathered from the mined source code is used to create pairs. Pairing every statement would result in many false positives therefore only statements with a data dependency are paired. When the method in figure 5.2 is traced and filtered the following sequences are created from the control flow graph:

**Sequence 1:**
*SPWeb SPWeb1 = SPContext.Current.Web;*
*SPWeb1 != null*
*SPUser SPUser1 = SPWeb1.CurrentUser;*
*String1 = SPUser1.LoginName;*

**Sequence 2:**
*SPWeb SPWeb1 = SPContext.Current.Web;*
*SPWeb1 != null*

The following pairs can be created using the data dependency restriction:

| First statement | Second statement |
|---|---|
| *SPWeb SPWeb1 = SPContext.Current.Web;* | *SPWeb1 != null* |
| *SPWeb SPWeb1 = SPContext.Current.Web;* | *SPUser SPUser1 = SPWeb1.CurrentUser;* |
| *SPWeb1 != null* | *SPUser SPUser1 = SPWeb1.CurrentUser;* |
| *SPUser SPUser1 = SPWeb1.CurrentUser;* | *String1 = SPUser1.LoginName;* |
| *SPWeb SPWeb1 = SPContext.Current.Web;* | *SPWeb1 != null* |

Notice how only *String1 = SPUser1.LoginName* is only paired with *SPUser SPUser1 = SPWeb1.CurrentUser*. Both sequences use the same object *SPUser1* and as such they are paired. Also notice how one pair is found twice. This means that the pair has a stronger connection and any deviation from that pair is more likely to be an anomaly. Every pair is scored based on the frequency with which the pair is found within the mined source code. The score is called "support" in this research to normalize the terminology between the other two algorithms.

| First statement | Second statement | Support |
|---|---|---|
| *SPWeb SPWeb1 = SPContext.Current.Web;* | *SPWeb1 != null* | 2 |
| *SPWeb SPWeb1 = SPContext.Current.Web;* | *SPUser SPUser1 = SPWeb1.CurrentUser;* | 1 |
| *SPWeb1 != null* | *SPUser SPUser1 = SPWeb1.CurrentUser;* | 1 |
| *SPUser SPUser1 = SPWeb1.CurrentUser;* | *String1 = SPUser1.LoginName;* | 1 |

**Figure 5.3 Example pairs with their support**

Figure 5.3 shows the support of each unique pair. As the pair *SPWeb SPWeb1 = SPContext.Current.Web -> SPWeb1 != null* was found twice the support becomes 2.

```
public string GetUrl(){
    SPWeb web = SPContext.Current.Web;
    string returnValue = web.Url;
    return returnValue;
}
```

**Figure 5.4 Example method**

### 5.3.2 Violations and anomalies

When all pairs are found in the mined data the next step is to find violations within the selected project. Figure 5.4 shows a method which will be used to illustrate the algorithm. Before any anomaly can be found, the method in figure 5.4 is processed as described in section 4. The control flow graph is transformed into one sequence as the method has only one execution path:

*SPWeb SPWeb1 = SPContext.Current.Web*
*String1 = SPWeb1.Url*

For this example the pairs from the previous section shown in figure 5.3 are used. Every statement in the sequence is matched against the first statement in each pair. Any pair where the first statement matches the second statement in the pair is searched for in the sequence. If the second statement in the pair is not found in the sequence the violation is registered. Using the pairs and the sequence from the previous section the following violations can be found:

*SPWeb1 != null*
*SPUser SPUser1 = SPWeb1.CurrentUser*

Each violation is scored based on the support of the pair it violated. *SPWeb1 != null* will get a score of 2 as the pair has a support of 2. The score of *SPUser SPUser1 = SPWeb1.CurrentUser* will become 1.

### 5.3.3 Filtering and selecting anomalies

After a method is analysed and all violations are registered, a threshold is used to determine whether a violation will be marked as an anomaly. In case of a threshold of 2 and the example of the previous section, only *SPWeb1 != null* would be marked as an anomaly. Using a threshold of 1 and the example of the previous section, both violations would be marked as an anomaly. To prune false positives, all anomalies with no complete data dependency with the method are filtered out. One anomaly in the previous example does not have complete data dependency with the method: *SPUser SPUser1 = SPWeb1.CurrentUser*. The method does have an *SPWeb* object but lacks an *SPUser* object and therefore *SPUser SPUser1 = SPWeb1.CurrentUser* is filtered out. The data dependency rule allows for a lower threshold without increasing false positives, which means that false negatives are limited. Violations which have a score the same as or same or higher than the threshold and which also fit the data dependency rule are added to the output as anomalies.

```
public string UpdateSite(string url, SPUser newAuthor){
    if (string.IsNullOrEmpty(url))
        return string.Empty;
    SPSite spSite = new SPSite(url);
    SPWeb spWeb = spSite.RootWeb;
    string currentVersion = (string)spWeb.GetProperty("Version");
    if (currentVersion.Equals("1")){
        spWeb.SetProperty("Version","2");
        spWeb.Author = newAuthor;
        currentVersion = "2";
    }
    spWeb.Update();
    spSite.Dispose();
    return currentVersion;
}

public void SetProperty(string siteUrl, string key, string value){
    SPSite site = new SPSite(siteUrl);
    SPWeb web = site.RootWeb;
    if (web.Properties.ContainsKey(key)){
        web.SetProperty(key, value);
        web.Update();
    }
    site.Dispose();
}
```

**Figure 5.5 Example methods containing multiple control flows with static and relative API usage sequences**

### 5.4 Recurring sequences

PR-Miner employs recurring sequences and shows promising results in finding anomalies. The mined sequences are analysed for recurring sequences and are used to find anomalies in target sequences.

#### 5.4.1 Finding recurring sequences

To explain the algorithm, the methods shown in figure 5.5 are used. These methods are processed as described in section 4. Every unique statement in all execution paths in the control flows is retrieved. In order to improve the performance of this algorithm, the statements are hashed as described in section 4.4. Figure 5.6 shows all unique statements and a simplified hash using the methods in figure 5.5.

| Statement | Hash |
|---|---|
| *SPSite SPSite1 = new SPSite(string1);* | 1 |
| *SPWeb SPWeb1 = SPSite1.RootWeb;* | 2 |
| *string string2 = (string)SPWeb1.GetProperty(StringLiteral);* | 3 |
| *SPWeb1.SetProperty(StringLiteral, StringLiteral);* | 4 |
| *SPWeb1.SetProperty(string2,string3);* | 5 |
| *SPWeb1.Author = SPUser1;* | 6 |
| *SPWeb1.Update();* | 7 |
| *SPSite1.Dispose();* | 8 |
| *SPWeb1.Properties.ContainsKey(string2)* | 9 |

**Figure 5.6 Methods in figure 5.5 hashed**

Notice that *SPSite spSite = new SPSite(url)* and *SPSite site = new SPSite(siteUrl)* are the same statements. The variables *url* and *siteUrl* have different names yet they are of the same type: *string*. The same holds for the *spSite* and *site* objects. Both statements are normalized into *SPSite SPSite1 = new SPSite(string1);* as described in section 4.3.

The control flow of the methods in figure 5.5 reveals the following API usage sequences using the hash in figure 5.6: *{1, 2, 3, 4, 6, 7, 8}, {1, 2, 3, 7, 8}, {1, 2, 9, 5, 8}, {1, 2, 9, 8}*. Using the classic algorithm *Apriori* [4] it is possible to find frequent itemsets where an itemset is a set of items. An itemset is considered frequent if a sub-itemset, a subset of an itemset, is contained in more than a specified threshold of itemsets. For this example a threshold of 4 is used on the list of sequences L:

L = *{{1, 2, 3, 4, 6, 7, 8}, {1, 2, 3, 7, 8}, {1, 2, 9, 5, 8}, {1, 2, 9, 8}}*

The smallest itemsets with their support to be found are the following:

| **Itemset** | **Support** |
|---|---|
| {1} | 4 |
| {2} | 4 |
| {3} | 2 |
| {4} | 1 |
| {5} | 1 |
| {6} | 1 |
| {7} | 2 |
| {8} | 4 |
| {9} | 2 |

Using the threshold of 4 the following itemsets are discarded: *{3}, {4}, {5}, {6}, {7}, {9}*. Any larger itemsets containing any of these sets are also discarded as their support will never be higher than the support of these itemsets. Using the previous itemsets with a support higher than the threshold, the following itemsets can be found with their support.

| **Itemset** | **Support** |
|---|---|
| {1, 2} | 4 |
| {1, 8} | 4 |
| {2 ,8} | 4 |

All three itemsets have a support of 4, which means none is discarded as their support is the same or higher than the threshold. The next step is to find even bigger itemsets.

| **Itemset** | **Support** |
|---|---|
| {1, 2, 8} | 4 |

Only one itemset can be found using the previous itemsets. Combining the previous results the output using the Apriori algorithm on the list L is as follows.

| Itemset | Support |
|---|---|
| {1} | 4 |
| {2} | 4 |
| {8} | 4 |
| {1, 2} | 4 |
| {1 ,8} | 4 |
| {2, 8} | 4 |
| {1, 2, 8} | 4 |

**Figure 5.7 Recurring sequences**

Looking at the output of the *Apriori* algorithm shown in figure 5.7 a problem emerges. In the example *{1, 2}* and *{1, 2, 8}* are contradictory. *{1, 2}* suggests that this is a correct sequences whereas *{1, 2, 8}* suggests that *{1, 2}* must contain *8* implying that *{1, 2}* is an anomaly. Z. Li and Y. Zhou propose [3] the use a FP-tree-based mining algorithm called *FPclose* [3][5] to solve the frequent itemset problem [9]. Instead of using all found frequent sub-itemsets *FPclose* only uses the closed sub-itemsets. "*A closed sub-itemset is the sub-itemset whose support is different from that of its super-itemsets*" [3]. Considering the recurring sub-itemsets found in the example *{1}, {2}, {8}, {1, 2}, {1, 8}, {2, 8}* are not closed as their support is the same as their super-itemset *{1, 2, 8}.* Using the *FPclose* algorithm only *{1, 2, 8}* is regarded closed. *{1, 2, 8}* translates back into:

*SPSite SPSite1 = new SPSite(string1);*
*SPWeb SPWeb1 = SPSite1.RootWeb;*
*SPSite1.Dispose();*

The closed recurring sequences hold no information about the order in which statements are executed. The result *{1, 2, 8}* shows a list of items without information about of those items. It makes no sense to execute *SPSite1.Dispose()* before *SPSite SPSite1 = new SPSite(string1)*. The sequence order can be extracted from the identified statements and their ordering within the method. Analyzing the methods containing *{1, 2, 8}* reveals that only *{1}* ⇒ *{2}* and *{2}* ⇒ *{8}* occur implying that only *{1}* ⇒ *{2}* ⇒ *{8}* is a valid recurring sequence. Before recurring sequences are used their ordering is determined by analyzing the methods where these recurring sequences are used.

### 5.4.2 Finding and filtering anomalies

The process of finding and filtering anomalies is the same as with the pair comparison described in section 5.3. All closed recurring sequences are used to find violations in the selected project. The score of each violation is the support of the recurring sequence which was violated. Semantic analysis is used to test if all the used objects in the missing statements have a data dependency with the method. Every violation which has a data dependency with the method and a score which is the same or higher than the threshold is marked as anomaly and added to the output of this algorithm.

## 5.5 Sequence comparison

This algorithm is not based on existing work like the other two algorithms. It has the potential to generate a lot of false positives but also to find many false negatives. Hovemeyer and Pugh [16] findings are that even the simplest algorithm manages to find anomalies. The other two algorithms focus on specific areas to find anomalies and the combined output only shows anomalies which are found by at least two algorithms. Because the two algorithms focus on specific areas it is likely that the combined output will contain few or no results. Also, due to the focus of the other two algorithms, false negatives are likely to increase. To counter these problems this algorithm is used to limit the false negatives and increase the number of results in the combined output.

```
public bool HasTitle(){
    SPWeb web = SPContext.Current.Web;
    return string.IsNullOrEmpty(web.Title);
}

public void AddUser(string url, string login){
    SPSite site = new SPSite(url);
    SPWeb web = site.RootWeb;
    web.EnsureUser(login);
    web.Update();
    site.Dispose();
}

public void RenameWeb(string url, string title){
    SPSite spSite = new SPSite(url);
    spSite.RootWeb.Title = title;
    spSite.RootWeb.Update();
    spSite.Dispose();
}
```

**Figure 5.8 Example methods to be mined**

```
public bool HasAdmins(string url){
    SPSite site = new SPSite(url);
    SPWeb web = site.RootWeb;
    return web.SiteAdministrators.Count > 0;
}
```

**Figure 5.9 Example method to be analysed**

### 5.5.1 Finding and ranking anomalies

This algorithm has no prior tasks to fulfill in order to find anomalies. In essence this algorithm compares two sequences and registers all differences. To illustrate this algorithm figure 5.8 shows three example methods to be mined and figure 5.9 a method to be analysed. Like the two other algorithms all data is processed as described in section 4. Every method is transformed into a control flow graph and statements not related to the selected API are discarded. Converting the control flows of the methods the following sequences emerge:

**Mined data:**
*SPWeb SPWeb1 = SPContext.Current.Web*
*return string.IsNullOrEmpty(SPWeb1.Title)*

*SPSite SPSite1 = new SPSite(string1)*
*SPWeb SPWeb1 = SPSite1.RootWeb*
*SPWeb1.EnsureUser(string2)*
*SPWeb1.Update()*
*SPSite1.Dispose()*

*SPSite SPSite1 = new SPSite(string1)*
*SPSite1.RootWeb.Title = string2*
*SPSite1.RootWeb.Update()*
*SPSite1.Dispose()*

**Sequence to be analysed:**
*SPSite SPSite1 = new SPSite(string1)*
*SPWeb SPWeb1 = SPSite1.RootWeb*
*return SPWeb1.SiteAdministrators.Count > 0*

This algorithms compares every sequence from the mined data with the sequence to be analysed and registers the differences. At least one statement should match between the two sequences otherwise all differences between the two sequences are ignored. Looking at the first sequence in the mined data no statement matches the statements in the sequence to be analysed therefore no differences are registered. The other two sequences have at least have one statement which matches the statements in the sequence to be analysed: *SPSite SPSite1 = new SPSite(string1)*. Comparing the two sequences from the mined data with the sequence to be analysed reveals the following differences:

*SPWeb1.EnsureUser(string2)*
*SPWeb1.Update()*
*SPSite1.Dispose()*

*SPSite1.RootWeb.Title = string2*
*SPSite1.RootWeb.Update()*
*SPSite1.Dispose()*

During the comparison every differentiation is matched against the already-found differentiations. Matching differentiations are counted and the result used as a score. For this example the differences and their scores is as follows:

| **Statement** | **Score** |
|---|---|
| *SPWeb1.EnsureUser(string2)* | 1 |
| *SPWeb1.Update()* | 1 |
| *SPSite1.Dispose()* | 2 |
| *SPSite1.RootWeb.Title = string2* | 1 |
| *SPSite1.RootWeb.Update()* | 1 |

To filter out valid differentiations from the anomalies, a threshold is used to limit the false positives. If a threshold of 2 is used for the example, one differentiation is registered as anomaly: *SPSite1.Dispose()*. As with the other two algorithms, a data dependency rule is used. If no data dependency exist between the method and the missing statement, the anomaly is discarded. All remaining anomalies are used for the output of this algorithm.

**5.6 Presenting the anomalies**

The goal of this research is not only to find anomalies but also to limit the number of false positives and false negatives. Using the sequence comparison algorithm the number of false negatives is likely to be reduced. In order to reduce the false positives the results of all three algorithms are combined. Any anomaly found by two or all three algorithms are selected for the combined output. This output is presented to the user to verify the anomalies and fix any bugs or issues related to these anomalies.

Reducing the false positives and false negatives is a good way to deal with the problems described in section 2.10. Adler and Borys [15] found that enabling users also increases the usage. They presented an example where Xerox deviated from the normal belief that tasks should be simplified and user choices should be limited. Xerox changed the interface of their photocopiers giving users more and increasingly complex tasks with every layer in the interface. This enabled users to resolve complex errors such paper jams themselves. While paper jams occurred often, users were more forgiving, increasing the positive feel about the photocopiers. Inspired by the examples of Adler and Borys the output of every individual algorithm is also presented to the users giving them the option to check for more anomalies if they desire. In doing so the users may be forgiving of false positives in the main output when they have the option to check anomalies in the individual output files.

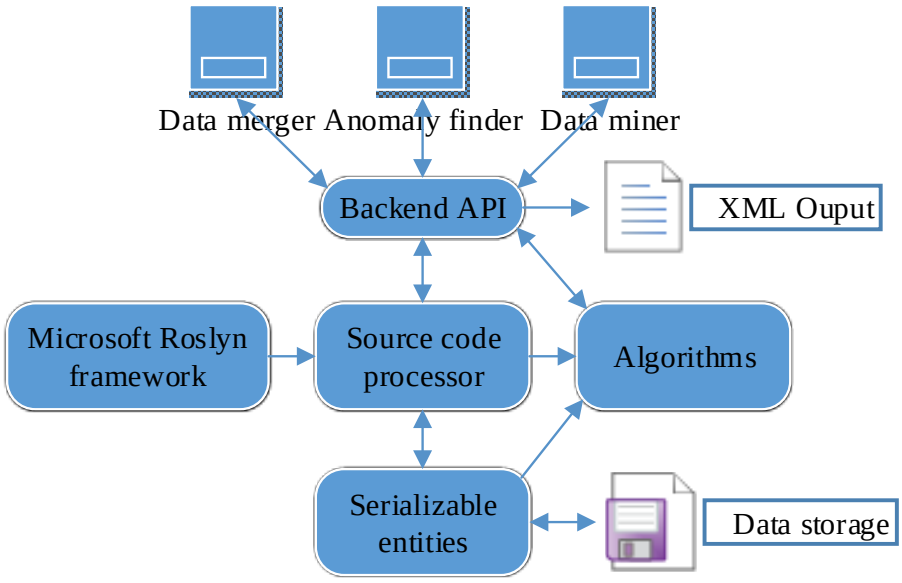


**Figure 6.1 Prototype design**

# 6 Prototype

To test the theory of this research and show its effectiveness, a prototype is developed employing the theory of this research. The prototype is written in C# using ASP.NET 4.5 which supports advanced multitasking statements. These multitasking statements are used to speed up the prototype considerably. The September 2012 version of Microsoft Roslyn is used to retrieve data from the source code. An API is developed containing all logic needed to support all tasks. The data within the prototype is serializable to enable storage on disk. To decrease the size, the serializable data is compressed, and decompressed when accessed. The output for the user is written in an XML format allowing it to be opened by various applications such as Microsoft Excel.

### 6.1 Design

Figure 6.1 shows a high level design of the prototype. All interfaces communicate with a backend API to allow extensibility. New interfaces are easily added or existing ones changed without the need to change the backend. The source code from the mined projects or selected project is processed by the source code processor. Parsing and semantic analysis of the source code is performed using the Microsoft Roslyn framework. Data which is generated is stored within serializable entities which allow for storage on disks. This enables the retrieval of the data when needed. Source code filtering and tracing is performed by the source code processor and stored as a control flow graph entity. These control flow graph entities hold the ability to generate the sequences needed for the algorithms. The algorithms process the data and report the anomalies found back to the backend API. The backend API writes the output to XML files to a user-defined location.

### 6.2 Interfaces

The graphical user interface is split into different forms specializing in certain tasks. This allows for easy customization without interfering with the other forms.

#### 6.2.1 Data miner

Using the user-provided directory the miner processes any Visual Studio solution found within all sub directories. The data miner builds a list of all frameworks used in the found Visual Studio solutions. For every framework it retrieves the namespace identifiers. For example *Microsoft.SharePoint, Version=14.0.0.0, Culture=neutral, PublicKeyToken=71e9bce111e9429c* is one of the identifiers for the SharePoint 2010 framework API. The miner

selects all child namespaces based on the root namespace selected by the user and processes the statements as described in section 4. All control flow graphs found are serialized, compressed and stored on disk based on the location and filename provided by the user.

#### 6.2.2 Data merger

Mined data from other solutions can be merged into one big data file using the data merger tool. It holds the ability to preselect framework API namespaces to filter out any unwanted framework APIs. The data merger decompresses and deserializes the selected data files. It merges the lists leaving out unwanted namespaces. The new list is also serialized, compressed and stored on disk.

#### 6.2.3 Anomaly finder

The anomaly finder gives the ability to select a Visual Studio solution and a data file containing the mined data. It opens the solution and performs the same step as the data miner except it does not store the data on disk. The selected project and the mined data are used as input for the algorithms described in section 5. The output from every algorithm is stored individually on disk as XML along with the final output.

### 6.3 Using the prototype

To find anomalies certain tasks need to be performed which are described in the following sections.

#### 6.3.1 Selecting a framework

First step is selecting a target framework. This framework is used to find anomalies in API usage within selected projects. The APIs of many frameworks are split into certain tasks and areas. The SharePoint framework has an API for server tasks, an API for client tasks, etc. C# uses namespaces to identify these frameworks. These namespaces include, in addition to the name, information such as the version of the framework. Frameworks often use a root namespace; Microsoft.SharePoint in the case of the SharePoint framework. For this step a root namespace is selected which is used for the subsequent steps. Using these namespaces resolves the problem described in section 2.7 as they contain version numbers.

#### 6.3.2 Mining source code

When a framework is selected for finding API usage anomalies, source code needs to be searched for to mine data from. It is important to select source code that uses the same framework otherwise no data can be extracted from the source code. Besides external source code the source code to be analysed can also be used to mine. During mining control flow graphs are created holding the normalized statements as described in section 4. These control flow graphs are stored into a data file which can be used for the next step.

#### 6.3.3 Finding anomalies

When a framework is selected and the data is mined the last step is to find anomalies. A project is selected to find anomalies together with the data file containing the mined data. Control flow graphs are created from the selected project as described in section 4. These control flow graphs together with the mined data are used as input for the algorithms. Every algorithm searches for anomalies and creates an output as described in section 5.

#### 6.3.4 Presenting the data

When all algorithms are finished the outcome of every algorithm is combined. Anomalies found by two or all three algorithms are added to the combined output. When each algorithm is finished the combined output and the individual outputs of the algorithms are presented to the user in an XML file. This enables the user to open the file in different programs such as Excel or, for example, to develop a Visual Studio add-on which can process the XML file and point out the anomalies.

### 6.4 Performance

At the moment the prototype uses the latest beta of the Roslyn framework to process the statements. Like the prototype itself the Roslyn framework is not optimized for performance. It takes a minute to mine 236000 lines of code divided between 180 solutions. Finding anomalies for a solution of 36000 lines of code using the mined data takes about 5 minutes. These figures are based on a desktop computer using a 3.4 GHz quad core CPU and 8GB internal memory. The mined data takes about 3MB to store on disk.

# 7 Evaluation

A real scenario is used to demonstrate the effectiveness of the algorithms and the use of a combined output. Appendix B shows the list of all external solutions which are used to mine data from. The mined data is used to find anomalies in the solutions listed in Appendix A. All found anomalies are manually checked to verify for false and true positives allowing precision be to determined.

### 7.1 Results

The solutions listed in Appendix A and B are both mined for data using the miner in the prototype. The solutions listed in Appendix A are selected to find anomalies using the mined data. Cloc [20] counted 236.000 lines of code in the mined data. The solutions to be searched for anomalies have 36000 lines of code. Using the SharePoint framework API the following namespaces were found within the mined data:

| Namespace | Version |
|---|---|
| Microsoft.SharePoint | 14.0.0.0 |
| Microsoft.SharePoint.Publishing | 14.0.0.0 |
| Microsoft.SharePoint.Taxonomy | 14.0.0.0 |
| Microsoft.SharePoint.Portal | 14.0.0.0 |
| Microsoft.SharePoint.Search | 14.0.0.0 |
| Microsoft.SharePoint.Client | 14.0.0.0 |
| Microsoft.SharePoint.Client.Runtime | 14.0.0.0 |
| Microsoft.SharePoint.Client.ServerRuntime | 14.0.0.0 |
| Microsoft.SharePoint.IdentityModel | 14.0.0.0 |
| Microsoft.SharePoint.WorkflowActions | 14.0.0.0 |

**Figure 7.1 List of all namespaces**

#### 7.1.1 Results: algorithms

The algorithms all employ thresholds to prune false positives. Two thresholds are shown in the results to show the effectiveness of changing the thresholds. The optimal threshold was determined after identifying the true and false positives using the lowest and highest thresholds. Using the output of the algorithms and manually verifying every anomaly, these are the results.

| Threshold | False positives | True positives | Precision |
|---|---|---|---|
| **Minimum** | 1028 | 134 | 0.12 |
| **Optimal** | 50 | 19 | 0.28 |

**Figure 7.2 Result of the pair comparison algorithm**

| Threshold | False positives | True positives | Precision |
|---|---|---|---|
| **Minimum** | 108 | 112 | 0.51 |
| **Optimal** | 108 | 112 | 0.51 |

**Figure 7.3 Result of the recurring sequences algorithm**

| Threshold | False positives | True positives | Precision |
|---|---|---|---|
| **Minimum** | 461 | 84 | 0.15 |
| **Optimal** | 303 | 60 | 0.17 |

**Figure 7.4 Result of the sequence comparison algorithm**

The closed recurring sequence algorithm attained the highest precision. The high precision is due to one statement often missing within the source code. The value *SPWeb.AllowUnsafeUpdates* must be set to true before a *SPWeb.Update()* and set to false afterwards. This was missing 118 times in the source code. If the *SPWeb.AllowUnsafeUpdates* is filtered out of the results the precision drops to 0.15 which is in line with the other algorithms.

### 7.1.2 Results: combined output

To limit the false positives the results of all algorithms are combined. The following results show the results of the outputs combined.

| Threshold | False positives | True positives | Precision |
|---|---|---|---|
| **Minimum** | 307 | 96 | 0.24 |
| **Optimal** | 45 | 23 | 0.34 |

**Figure 7.5 Combined results when anomalies are found by two out of three algorithm**

| Threshold | False positives | True positives | Precision |
|---|---|---|---|
| **Minimum** | 12 | 11 | 0.48 |
| **Optimal** | 0 | 3 | 1.00 |

**Figure 7.6 Combined results when anomalies are found by all three algorithm**

Combining the outputs of the algorithms does increase the precision but in the process the recall decreases. Because the individual outputs are also available for the user, the decreased recall is not problematic. The user is likely to be inclined to check the individual outputs when the precision of the combined output is high.

### 7.2 Analysis

While a large number of true positives have been found the number of false positives is much higher. To determine where the false positives come from, the source code is analysed and a theory is formed.

```
public int CountTitle(string title)
{
    int returnValue = 0;
    foreach (SPList list in SPContext.Current.Web.Lists){
        foreach (SPListItem item in list.Items){
            if (item.Title.Equals(title)) returnValue++;
        }
    }
    return returnValue;
}

public int CountTitle(string title){
      return (from SPList list in SPContext.Current.Web.Lists from SPListItem item in
                list.Items select item).Count(item => item.Title.Equals(title));

}
```

**Figure 7.7 Example method rewritten using Linq**

#### 7.2.1 Language evolution

During code review of the selected source code it was apparent that ASP.NET has evolved a lot. New extensions are introduced like Linq which allow for more efficient source code. Complex nested loops can be rewritten with one single Linq expression. ReSharper is a development tool used by the programmers who developed the analysed solutions. ReSharper gives suggestions as to which loops can be rewritten using Linq and automatically rewrites normal and nested loops without any help from the programmer. Figure 7.7 shows a method which is rewritten using the ReSharper tool. No anomalies were found using these Linq expressions. As it is unlikely no bugs or issues exist when using Linq expressions the approach used in this research is likely not suited for dealing with these expressions.

```
public void CreateList(string name){
    SPWeb web = SPContext.Current.Web;
    CreateList(web, name);
}

public void CreateList(string url, string name){
    using (SPSite site = new SPSite(url)){
        SPWeb web = site.RootWeb;
        CreateList(web, name);
    }
}

public void CreateList(SPWeb web, string name){
    web.AllowUnsafeUpdates = true;
    web.Lists.Add(name, name, SPListTemplateType.Tasks);
    web.Update();
    web.AllowUnsafeUpdates = false;
}
```

**Figure 7.8 Example of function overloading**

### 7.2.2 Function overloading

Some programmers used function overloading to allow the same task with different input values. Figure 7.8 shows three methods which are overloaded. The reason behind it is to give programmers more freedom in other sections of the source code. The algorithms however report an anomaly in the first two methods that the following is missing:

- *SPWeb.AllowUnsafeUpdates = true;*
- *SPWeb.Update()*
- *SPWeb.AllowUnsafeUpdates = false;*

The suggestion is correct as the *SPWeb* object is changed but those statements are executed in the last method. Therefore the presented anomalies are a false positive.

### 7.2.3 Object abstraction

The SharePoint framework offers objects which can be used to interact with the data in SharePoint. The objects do not always offer ways to add custom data. Object abstraction layers are introduced in the code to resolve the problem. While it is an elegant solution for the problem, it makes the source code more difficult to automatically scan for anomalies. In the case of the analysed source code, the *SPUser* object is wrapped in a custom class *WndUsr* which offers specific data on top of the normal *SPUser* data. Using object abstraction makes it impossible for the selected algorithms to find anomalies. The mined data does not contain any *WndUsr* objects and the *WndUsr* object is not part of the SharePoint framework which means that statements using the *WndUsr* object are filtered out.

### 7.2.4 Functional intention

Every method is written with a functional intention often inferred from the name of the method. A method with the name *CreateList* will likely to be able to create lists. The algorithms do not have any knowledge about these intentions which leads to many false positives. For example the object *SPList* has several variables which can be set to change the specifications for a list. These specifications have an impact on how the user interacts with the list. *SPList.EnableFolderCreation* is a variable which disables or enables the ability to create folders effectively allowing or disallowing for users to create folders in the list. During mining of the source code *SPList.EnableFolderCreation* is often observed in conjunction with *SPList.Update()*. When the solutions are analysed for anomalies *SPList.EnableFolderCreation* is registered as an anomaly when *SPList.Update()* is found in a method. *SPList.Update()* is executed when changes are made to the *SPList* object or its content. Setting the threshold to a higher level to filter out these anomalies also filters out true positives. Another solution would be to give programmers the option to select statements to ignore, although this approach is sensitive to the programmers' subjectivity leading to wrongly ignored statements.

```
public SPList GetList(SPSite site, string listName){
    foreach (SPWeb web in site.AllWebs){
        SPList list = web.Lists.TryGetList(listName);
        if (list != null)
            return list;
    }
    return null;
}
```

**Figure 7.9 Generic method for retrieving a SPList object**

### 7.2.5 Generic methods

It is common practice to optimize source code and to reuse methods. In the analysed source code optimizations have been performed and methods created for generic usage. As SharePoint is a CMS and a document management system many tasks are done within a context. This is the reason why the reusable methods are developed to perform a task in different contexts. The context is often an input for the method to perform the task. Figure 7.9 shows a generic method to retrieve an *SPList* while it is unknown where the lists resides. The method uses an *SPSite* object as an input for the context. It return an *SPList* object if an *SPList* is found otherwise it will return *null*. When the algorithms analyse the method they will suggest that *site.Dispose()* is missing as every *SPSite* object must explicitly be disposed. In this case disposing the *SPSite* object results in unexpected behavior as the *SPSite* object is no longer available for the method which called the generic method. *Site.Dispose()* is not an anomaly as the algorithms suggest.

### 7.2.6 Verifying anomalies

For this research every anomaly was manually verified for related bugs or issues. While this was a time consuming process it was considerably faster than code reviewing 36000 lines of code. There is however room for improvement. To help programmers with verifying anomalies an extension for the development tools can be created. The list of anomalies is presented sorted by method. When the anomalies are verified and fixes are applied the programmer can move on to the next method. This would speed up the process even further, increasing the likeliness programmers will use the software based on this research.

### 7.2.7 Mined data quality

The algorithms use the mined data to determine the norm and register deviations from that norm as an anomaly. It is highly likely that the mined data also contains bugs and issues. If a bug or issue exists in a piece of source code it will not be a problem as the algorithm focusses on using the norm. A problem arises if a bug or issue is systematically spread throughout the mined data. This problem can be resolved using more different projects to mine data from. In some cases a bug is regarded as the norm in the community using the framework not knowing that in reality it is a bug. The algorithms will falsely imply from the mined source code that the bug is the norm and that correct code represents a bug. These inverted results will not be recognized by the programmer as the framework community regards it as the norm. The only solution for this problem is to refresh the source code which is mined replacing old source code with recent source code. When the community using the framework recognizes their mistake they will update their source code. If the source code of the mined data is regularly refreshed the outcome of the algorithms will reflect the new insights of the community.

### 7.2.8 Threat to validity

Subjectivity is a threat to validity for the results in this research. The anomalies presented by the prototype are manually checked for related bug or issues. Bugs causing crashes are easily marked as true positive as nobody will argue that a crash is normal behavior. For other anomalies related to bugs or issues, it may be debatable whether these bugs or issues are really bugs or issues. This is where the subjectivity of a programmer plays an important role. As mentioned in section 3.2 it is the manual evaluation of every anomaly that is a subjective process. While the results in this research are discussed within the development team it is no guarantee of complete objectivity. Some anomalies may incorrectly be marked as false positive or true positive due to this subjectivity.

# 8 Conclusion

Mining source code to find anomalies in selected projects is a sensible approach. However human verification is still needed to check all anomalies for related bugs or issues. The time investment in using an automated approach as used in this research is much lower than manual code reviewing but results and analysis do show that automated code reviews are still not a viable replacement for manual code reviews.

Combining the results of two existing algorithms with the help of a simpler algorithm showed promising results. With the lowest possible thresholds the combined output manages to achieve 24% true positives. After determining the optimal thresholds in conjunction with the mined data the true positives went up to 34%. Zeller et al. reported a true positive percentage of 25%. Combining multiple algorithms to find anomalies therefore warrants further research. If development teams were to consider changing their coding standards to allow better automatic analysis this percentage would likely go up even more.

In conclusion the U.S. economy would be able to reduce the $59.5 billion it now loses due to bugs with relative ease using automated code reviews.

# 9 Future work

Like this research, other related research uses existing source code to determine the norm and register any deviations from the norm. Different types of algorithms are introduced and used to find the norm from different perspectives. Various sizes of source code are used to mine data up to 200 million lines of code spanning 6000 open source projects. Zeller reported a rate of 25% true positives. Would he also achieve a rate of 25% true positives with a corpus of 100 million lines of code? Or even with a corpus of 1 million lines of code? What is the impact of the corpus size on the found anomalies and rate of true positives?

During the manual process of checking every anomaly for related bugs or issues it became clear that many false positives and false negatives were due to different programming styles. Some programmers used one line of code while others used multiple lines of code for the same task. Analyzing existing code to determine the norm seems to have its limitations where comparing statements is impossible due to programming styles. To counter this problem it is possible to transform the source code to a default programming style. Would code transformation limit the false positives and false negatives?

At the moment this and other related research lack a feedback system. While programmers have to manually check every anomaly for possible related bugs or issues the results are not used to refine the data. This data is also usable to refine what is and what is not an anomaly. Can a feedback system where programmers can report their findings reduce the number of false positives?

Often a development team uses a repository to enable teamwork allowing multiple programmers to work on the same section of source code. Changes, bugs and issues are registered in the repository to improve development processes. Programmers select a task to work on and check in source code related to that task when they are done. Is it possible to mine these repositories for reported bugs and issues to decrease the false negatives?

# Bibliography


[1] Y.M. Mileva, V. Dallmeier, M. Burger and A. Zeller, *Mining trends of library usage*. IWPSE-Evol'09, August 24–25, 2009, Amsterdam

[2] Y.M. Mileva, V. Dallmeier and A. Zeller, *Mining API popularity*. TAIC PART 2010, LNCS 6303, pp. 173–180, 2010

[3] Z. Li and Y. Zhou, *PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code*. ESEC-FSE'05, September 5-9, 2005, Lisbon, Portugal

[4] R. Agrawal and R. Srikant. *Fast algorithms for mining association rules*. In Proc. 20th Int. Conf. Very Large Data Bases, 1994

[5] G. Grahne and J. Zhu. *Efficiently using prefix-trees in mining frequent itemsets*. In Proc. of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations, 2003

[6] J. Han, J. Pei, and Y. Yin. *Mining frequent patterns with-out candidate generation: A Frequent-Pattern Tree Approach*. In Proceedings of ACM SIG-MOD'00, pages 1–12, May 2000.

[7] M. Acharya, T. Xie, J. Pei, J. Xu, *Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications.* ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.Copyright 2007 ACM 978-1-59593-811-4/07/0009

[8] D. Engler, D. Y. Chen, and A. Chou. *Bugs as deviant behavior: A general approach to inferring errors in systems code*. In Proc. of the 18th ACM Symp. on Operating Systems Principles, 2001.

[9] P. Purdom, D van Gucht, D. Groth. *Average Case Performance of the Apriori Algorithm*, SIAM J. COMPUT., Vol. 33, No. 5, pp. 1223-1260, 2004

[10] Dunsmore, A., M. Roper, and M. Wood. 2000. *Object-Oriented Inspection in the Face of Delocalisation*. Proceedings of the 22nd International Conference on Software Engineering (ICSE) 2000: 467-476

[11] Votta, L. 1993. *Does every inspection need a meeting?* Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, December 8-10: 107-114

[12] Kelly, D., and T. Shepard. 2003. *An experiment to investigate interacting versus nominal groups in software inspection*. Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Centre for Advanced Studies Conference: 122-134

[13] Tassey, Gregory., *The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute of Standards and Technology, May 2002.

[14] Fred D. Davis., *Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology*. MIS Quarterly , Vol. 13, No. 3 (Sep., 1989), pp. 319-340

[15] Paul S. Adler, Bryan Borys., *Two Types of Bureaucracy: Enabling and Coercive*. Administrative Science Quarterly, Vol. 41, No. 1 (Mar., 1996), pp. 61-8

[16] David Hovemeyer and William Pugh., *Finding Bugs is Easy*. OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

[17] A. Wasylkowski, A. Zeller and C. Lindig., *Detecting Object Usage Anomalies*. ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

[18] W. Dickinson, D. Leon, and A. Podgurski. *Finding failures by cluster analysis of execution profiles*. In ICSE '01: Proc. of the 23rd International Conference on Software Engineering, pages 339–348. IEEE Computer Society, 2001.

[19] N. Gruska, A. Wasylkowski and A. Zeller., *Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection*. ISSTA'10, July 12–16, 2010, Trento, Italy.

[20] Cloc. Available at: http://cloc.sourceforge.net/ (Accessed: 2 Augustus 2013)

[21] SharePoint API documentation. Available at: http://msdn.microsoft.com/sharepoint (Accessed: 2 Augustus 2013)

[22] Microsoft Roslyn. Available at: http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx (Accessed: 2 Augustus 2013)

# Appendix A

| Project Name | Project Name |
|---|---|
| Windesheim.Wise.ProvisionWebparts | Windesheim.Internet |
| Windesheim.Wise.Startpagina.Webparts | Windesheim.SN.ExtraTools |
| Windesheim.Wise.Tools | Windesheim.SN.Mappings.UpdateMappingsContent |
| Windesheim.Wise.PowerShell2 | Windesheim.SN.Startpagina.Definitions |
| Windesheim.Wise.Onderwijs.Webparts | Windesheim.SN.Startpagina.Stapler |
| Windesheim.Wise.Organisatie.PageLayout | Windesheim.SN.Startpagina.Webparts |
| Windesheim.Wise.Organisatie.Webparts | Windesheim.SN.TimerJob.CommOverviewPerformance |
| Windesheim.Wise.PowerShell | Windesheim.SN.TimerJob.GsaPassportPhoto |
| Windesheim.Wise.Branding.MySite.Stapler | Windesheim.SN.TimerJob.MoveUserToAdGroup |
| Windesheim.Wise.Branding.MySite | Windesheim.SN.TimerJob.ProcessMysites |
| Windesheim.Wise.Branding.MySiteHost | Windesheim.SN.TimerJob.ProcessUserProfiles |
| Windesheim.Wise.Branding.Onderwijs | Windesheim.SN.TimerJob.SetQuota |
| Windesheim.Wise.Branding.Organisatie | Windesheim.SN.UserProfiles.WebService |
| Windesheim.Wise.CentraleCommunities.TimerJob | Windesheim.SN.Webparts |
| Windesheim.Wise.Comakership.PageLayout | Windesheim.Tools |
| Windesheim.Wise.Communities.CommConfig | Windesheim.Vacatures |
| Windesheim.Wise.Communities.SiteDefinition | SharePoint 2010 Search Adapters |
| Windesheim.Wise.Communities.Stapler | CreateCommunities |
| Windesheim.Wise.Communities.Webparts | DeleteCommunities |
| Windesheim.Wise.Configuration.Development | DeleteMySites |
| Windesheim.Wise.Configuration.Production | ImportUserProfileSettings |
| Windesheim.Wise.Configuration.Test | MigrateFileNames |
| Windesheim.Wise.ContentTypes | MysiteErrorReport |
| Windesheim.Wise.Helper | MysiteMigrationReport |
| Windesheim.Wise.MySite.Updater | MysiteSetNeeded |
| Windesheim.Wise.Onderwijs.CourseConfig | SharenetCreateUsers |
| Windesheim.Wise.Onderwijs.Stapler | SharenetFillPersonaUserprofile |
| Windesheim.Wise | Windesheim.SN.Configuration.Development |
| Windesheim.Wise.Branding.Comakership | Windesheim.SN.Configuration.DevelopTest |
| Windesheim.Wise.Branding.Communities | Windesheim.SN.Configuration.Production |
| Windesheim.Wise.Branding.General | Windesheim.SN.Configuration.Test |
| Windesheim.ApplicationPages | Windesheim.SN.Configuration |
| Windesheim.Controls | Windesheim.SN.ContentTypes |
| Windesheim.DocumentTemplates | Windesheim.SN.Controls |
| Windesheim.ExternalAccess | Windesheim.SN.CorporateNewsGatherer |
| Windesheim.Functiegebouw | Windesheim.SN.Departments.RVD |
| Windesheim.Helper | Windesheim.SN.Departments.RVE |
| Windesheim.Presentation | Windesheim.SN.Departments.Team |
| Windesheim.SearchBox | Windesheim.SN.Departments.Webparts |
| Windesheim.SiteMapProviders | Windesheim.SN.DepartmentsHost.Definitions |
| Windesheim.Webparts | Windesheim.SN.GSA |
| Windesheim.Wise.Generiek | Windesheim.SN.Infosite.Definitions |
| Windesheim.Internet.TwitterHandler | Windesheim.SN.MySite.Config |
| Windesheim.Internet.VacaturesPart | Windesheim.SN.News.Definitions |
| Windesheim.Internet.WebParts | Windesheim.SN.Search.ActionRedirect |
| Windesheim.Internet.TagCloud | Windesheim.SN.SearchBox.Infosite |
| TelerikRadControlsSP | Windesheim.SN |
| LiveAuth | Windesheim.SN.Announcements.Webparts |
| Windesheim.Internet.ResizeImages | Windesheim.SN.Branding |
| Windesheim.Internet.SiteMapControl | Windesheim.SN.Communities.Config |
| Windesheim.Internet.StaticSiteMapControl | Windesheim.SN.Communities.Domain |
| Windesheim.Internet.Studiekeuzetest | Windesheim.SN.Communities.Project |
| Windesheim.Internet.ChatPart | Windesheim.SN.Communities.ReusableContent |
| Windesheim.Internet.Configuration | Windesheim.SN.Communities.SiteDefinitions |
| Windesheim.Internet.ContactForm | Windesheim.SN.Communities.Study |
| Windesheim.Internet.ErrorPages | Windesheim.SN.Communities.WebServices |
| FormulierComakership | Windesheim.SN.CommunitiesHost.Definitions |
| FormulierMeeloopdagen | Windesheim.Inschrijvingen.Branding |
| Windesheim.Internet.FormulierTraining | Windesheim.Inschrijvingen.Data |
| Windesheim.Internet.Harmonica | Windesheim.Inschrijvingen.Framework |
| Windesheim.Internet.InschrijvenMeetingPoint | Windesheim.Inschrijvingen.Lijsten |
| Windesheim.Internet.MediaPart | Windesheim.Inschrijvingen.Mijn |
| Windesheim.Internet.MijnWF | Windesheim.Inschrijvingen.WebParts |
| Windesheim.Internet.NewsOverzicht | Windesheim.Lockdown |
| Windesheim.Internet.Opendagplanner | Windesheim.Formulieren.Conformation |
| Windesheim.Internet.Branding.Utils | Windesheim.Formulieren.Forms |
| Windesheim.Internet.BrochureAanvraag | Windesheim.Formulieren.TimerJobs |
| Windesheim.Internet.Branding | Windesheim.ShareNet.DesignPrototype |

# Appendix B

| Name | Location |
|---|---|
| 3D TagCloud for SharePoint 2010 | http://www.codeplex.com/ |
| Adventures with the SharePoint REST API | http://www.codeplex.com/ |
| Automated Metadata management for SharePoint | http://www.codeplex.com/ |
| Community Kit for SharePoint | http://www.codeplex.com/ |
| Export Version History Of SharePoint 2010 List Items to Microsoft Excel | http://www.codeplex.com/ |
| Multi Value Refiner for SharePoint 2010 | http://www.codeplex.com/ |
| People Picker Plus for SharePoint 2010 | http://www.codeplex.com/ |
| Pivot Viewer for SharePoint | http://www.codeplex.com/ |
| Send Documents as attachments with SharePoint 2010 | http://www.codeplex.com/ |
| SharePoint 2010 Activity Feed WebPart | http://www.codeplex.com/ |
| SharePoint 2010 Audience Membership Workflow Activity | http://www.codeplex.com/ |
| SharePoint 2010 automatic sign-in with mixed authentication | http://www.codeplex.com/ |
| SharePoint 2010 Batch Edit | http://www.codeplex.com/ |
| SharePoint 2010 Bulk Document Importer | http://www.codeplex.com/ |
| SharePoint 2010 CSV Bulk Taxonomy TermSet Importer and Exporter | http://www.codeplex.com/ |
| SharePoint 2010 FBA Pack | http://www.codeplex.com/ |
| SharePoint 2010 Foundation User Profile Service | http://www.codeplex.com/ |
| SharePoint 2010 Tasks Delegations Manager | http://www.codeplex.com/ |
| SharePoint 2010 The Genesis Framework | http://www.codeplex.com/ |
| SharePoint 2010 Unlock SP Files | http://www.codeplex.com/ |
| Sharepoint 2010 User Redirect | http://www.codeplex.com/ |
| SharePoint Bulk Uploader | http://www.codeplex.com/ |
| SharePoint Client Browser for SharePoint 2010 | http://www.codeplex.com/ |
| SharePoint Commander | http://www.codeplex.com/ |
| SharePoint Components3 | http://www.codeplex.com/ |
| SharePoint Correlation ID View Webpart | http://www.codeplex.com/ |
| SharePoint Cross Site Collection Security Trimmed Navigation | http://www.codeplex.com/ |
| SharePoint Dynamics Forms | http://www.codeplex.com/ |
| SharePoint Event Receiver Manager | http://www.codeplex.com/ |
| SharePoint Extensions | http://www.codeplex.com/ |
| SharePoint List Association Manager | http://www.codeplex.com/ |
| SharePoint List Field Manager | http://www.codeplex.com/ |
| SharePoint List View Filter | http://www.codeplex.com/ |
| Sharepoint manager 2013 | http://www.codeplex.com/ |
| SharePoint Messenger | http://www.codeplex.com/ |
| SharePoint Outlook Connector | http://www.codeplex.com/ |
| SharePoint PLM | http://www.codeplex.com/ |
| SharePoint Site Map Generator | http://www.codeplex.com/ |
| SharePoint SlideShow | http://www.codeplex.com/ |
| SharePoint Social Networking | http://www.codeplex.com/ |
| SharePoint Taxonomy and TermStore Utilities | http://www.codeplex.com/ |
| SharePoint Topology Data Collection | http://www.codeplex.com/ |
| Simple Recruitment solution for SharePoint 2010 | http://www.codeplex.com/ |
| Site Directory for SharePoint 2010 | http://www.codeplex.com/ |
| SkyDrive Connector for SharePoint | http://www.codeplex.com/ |
| SP List Kit for SharePoint 2010 | http://www.codeplex.com/ |
| SPManager | http://www.codeplex.com/ |
| Useful SharePoint Site Workflow Utilities | http://www.codeplex.com/ |
| User Profile Synchronization for SharePoint Foundation | http://www.codeplex.com/ |
| WebParts for SharePoint | http://www.codeplex.com/ |
| DotnetNuke | http://www.dotnetnuke.com/ |
| Umbraco CMS | http://www.umbraco.com/ |