UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Urmas Tamm

Eclipse plugin for analyzing embedded SQL queries in PHP programs

Master's Thesis (30 ECTS)

Supervisor: Vesal Vojdani
Co-supervisor: Aivar Annamaa

TARTU 2015

Eclipse plugin for analyzing embedded SQL queries in PHP programs

Abstract

During code development it is crucial to get fast feedback about the correctness of our code. Various hints are given by compiler through warnings and error messages that are displayed in the IDE, e.g. Eclipse. Unfortunately, this covers only the general host-language, in which we write the code. Often we need to use another language to communicate with a specific application domain: e.g. SQL for sending queries to a database engine. By default these SQL strings are not checked statically, although it would be highly beneficial. Alvor is a tool that statically checks SQL queries in Java programs. This thesis presents an extension for Alvor to add PHP support.

The key challenge when adapting Alvor to PHP was the dynamic nature of PHP. The solution is therefore limited to operate only within the scope of a PHP function or a PHP script. We evaluated the tool on open-source software and the results showed that it would be most beneficial to use it as a tool to support beginners in learning programming.

Keywords: program analysis, static analysis, Eclipse plugins, embedded SQL, Alvor.

Eclipse *plugin* PHP programmides leiduvate SQL päringute kontrollimiseks

Lühikokkuvõte

Käesoleva magistritöö eesmärgiks oli kavandada ja implementeerida Eclipse'i põhine töövahend, mis võimaldaks kontrollida PHP-skriptides leiduvate SQL-päringute süntaktilist ning semantilist korrektsust. Aluseks sai võetud juba eksisteeriv töövahend Alvor, mis on mõeldud Java koodis leiduvate SQL lausete kontrollimiseks. Töö tulemusena valmis laiendus AlvorPHP, mis kogub kokku päringulaused neid sisaldava PHP funktsiooni või skripti skoobis ning tagastab edasiseks käitlemiseks. PHP dünaamilise eripära tõttu ei olnud võimalik realiseerida toetust kõigile temas esinevatele keelelistele konstruktsioonidele. Mõned neist jäid realiseerimata ka ajapuuduse tõttu, mis jätab võimaluse laienduse edaspidiseks täiendamiseks. Sellele vaatamata on toetatud enam levinud ning kasutatavad vahendid. Antud laienduse kasutajate sihtgrupp peaks olema ennekõike algajad programmeerijad, kes saaksid seda kasutada õppimist toetava abivahendina.

Võtmesõnad: programmianalüüs, sõnede analüüs, Eclipse'i *pluginad*, programmides leiduvad SQL päringud, Alvor

Contents

INTRODUCTION

PHP is a popular scripting language used for web application development. Web applications are usually backed by a database engine. Conventionally PHP developers interact with the database through the standard APIs that require passing them SQL queries as strings. This is referred to as *embedding* the SQL queries into the host-language. Since the standard development tools provide no feedback on the correctness of such embedded queries, they are not checked until runtime. Thus, there is no guarantee that they result in a successful call instead of an error. Such faulty database calls can be pinpointed only by testing. The fact that the queries are often built up dynamically, by using concatenation and various other methods of string manipulation, makes it even harder to track down and fix bugs. Let's consider an example:

```
function limit_query($limit=10)
{
  return mysql_query("SELECT * FROM baz LIMIT".$limit);
}
```

The function *limit_query()* has one parameter, *$limit*, that it uses to construct and return a MySQL query. The query uses the database table named "bar" and returns a number of rows equal to the integer value of the *$limit* parameter.

Here we have two types of errors in our resulting query object that are underlined in red. First, we have a semantic error, since the name of the database table has been misspelled. Second, we have a syntactic error, due to the missing space between the keyword "*LIMIT*" and the value of the "*$limit*" parameter. It would be beneficial to be aware of such errors as soon as possible, without having to run the code. This is the case not with only PHP, but with any general-purpose host language (e.g. Java, Python etc.) that embeds a domain-specific language (DSL) such as SQL.

In this master's thesis we present a tool for statically analyzing embedded SQL queries inside PHP scripts. The tool itself is implemented as a plugin for the Eclipse IDE [1] and is an extension of an existing tool named *Alvor*, implemented by Annamaa et al [2]. Alvor provides feedback by detecting the queries in code and running them against the configured database. It originally used Java as the host language and a number of SQL dialects as the embedded DSLs. Alvor performs sound SQL syntax analysis during compile-time, supporting a number of string manipulation constructs from simple concatenation of strings to more sophisticated method calls and recursion.

The scope of this work is more limited than the original tool. We have not included all the features that are supported in Alvor's program analysis. The goal was to create a proof-of-concept extension for PHP, evaluate its results and leave room for future work if it would be reasonable to further develop it.

The thesis consists of the following parts: in the first chapter we are going to give a theoretical background of how Alvor functions in terms of program analysis and more specifically, string analysis. In the second chapter we give an overview of the Eclipse platform and the lifecycle of Eclipse plugins that make up Alvor's architecture. The third chapter is dedicated to Alvor's working principles and the fourth to the extension of Alvor for PHP (*AlvorPHP*), created by the author. In the fifth chapter, we evaluate the tool, using some open source projects and finally, in the sixth, consider possibilities for future work.

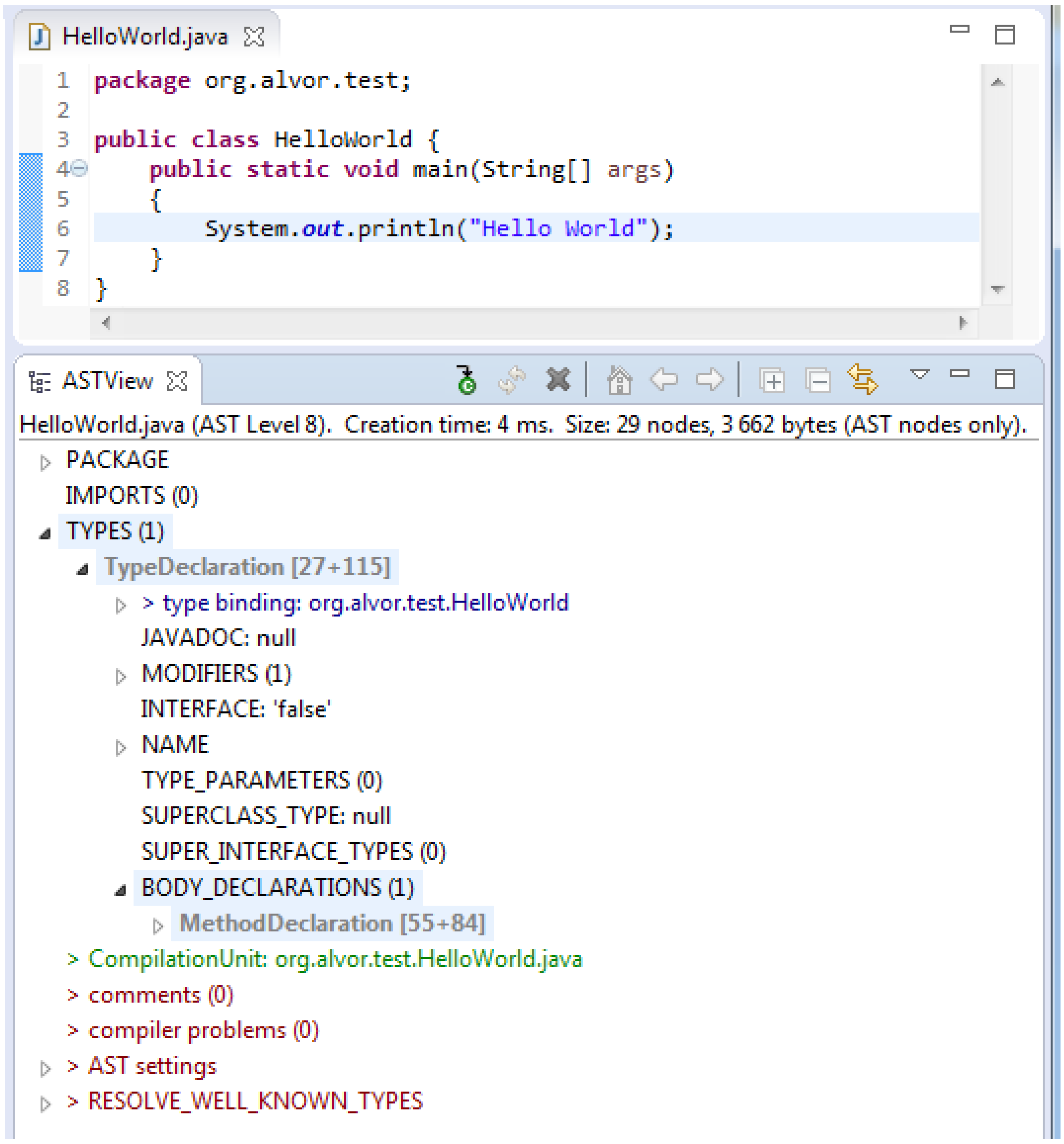
1. PROGRAM ANALYSIS

Program analysis is a technique for analyzing programs in terms of a specific property, such as the absence of certain types of run-time errors. Program analysis may take place during the execution of the program, called *dynamic program analysis*, or without executing the program at all, called *static program analysis*. In this thesis we are going to focus on the latter, since our goal is to give feedback to the programmer during the phase of code writing. The type of program analysis that Alvor uses is called *string analysis*. String analysis determines values of string expressions at specific points of program. We use that information to determine all the possible values of our embedded SQL queries.

1.1 Abstract Syntax Tree

To perform our analysis we make use of the program's *abstract syntax tree* (AST). The AST is a tree representation of the source code, where each node of the tree represents a language construct occurring in the program's source. The AST is similar to the DOM model inside an XML document. That kind of tree model is easier to analyze than the source text. Most of the code-related tools and features in the Eclipse IDE make use of the AST. The Eclipse *Java Development Tools* (JDT) [3] framework considers every source file as a tree of AST nodes. Each node in the AST is a subclass of the abstract *ASTNode* class, defined in the JDT. Every such subclass is a specialization for an element in the programming language being used.

Let us consider an example AST in Figure 1.1 that has been created using the Eclipse's built-in *ASTView* plugin for Java [4]. Here we have a simple "Hello World" program and its AST representation. As we can see, we have a separate node for our *main* method declaration (*MethodDeclaration*). There are specific nodes for other language constructs as well, such as assignments (*Assignment*), variable declarations (*VariableDeclarationFragment*), etc.


HelloWorld.java

```java
package org.alvor.test;

public class HelloWorld {
    public static void main(String[] args)
    {
        System.out.println("Hello World");
    }
}
```

ASTView

HelloWorld.java (AST Level 8). Creation time: 4 ms. Size: 29 nodes, 3 662 bytes (AST nodes only).

```
▷ PACKAGE
  IMPORTS (0)
◢ TYPES (1)
    ◢ TypeDeclaration [27+115]
        ▷ > type binding: org.alvor.test.HelloWorld
          JAVADOC: null
        ▷ MODIFIERS (1)
          INTERFACE: 'false'
        ▷ NAME
          TYPE_PARAMETERS (0)
          SUPERCLASS_TYPE: null
          SUPER_INTERFACE_TYPES (0)
        ◢ BODY_DECLARATIONS (1)
            ▷ MethodDeclaration [55+84]
  > CompilationUnit: org.alvor.test.HelloWorld.java
  > comments (0)
  > compiler problems (0)
▷ > AST settings
▷ > RESOLVE_WELL_KNOWN_TYPES
```

Figure 1.1: AST of a “Hello World” program in Java

1.2 Visitor pattern

As we can see in Figure 1.1, even a simple “Hello World” program can produce quite complex AST. Say, we want to reach the *MethodInvocation* object that represents our *System.out.println(“Hello World”)* call. One possible solution for that would be scanning all the levels in the entire AST, but is definitely not very convenient. Fortunately there exists a better solution for that: the *visitor design pattern*. It allows us to query for the child nodes of each *ASTNode* in our program AST. Covering all the aspects of the visitor pattern is out of the scope of this thesis. Gamma et. al. have given a good overview of it in their book about design patterns [5].

In the JDT, the *ASTVisitor* class represents our *visitor* object, which defines the *visit()* method for every *ASTNode* class object. In the *ASTNode* class we have the corresponding *accept()* method that accepts the visitor as it gets passed to it when we step through the AST.

The visitor pattern is somewhat cumbersome: each time we need to search for a specific node, we have to implement the needed *visit()* method that takes the node as its argument. This creates a certain amount of boilerplate code that is used nowhere else. In the JDT there is an alternative solution for that, which is being used in Alvor: a specific *SearchEngine* class allows us to conveniently search for Java elements inside the AST. We can define our own search patterns that allow us to search for a node by its *structural properties*. E.g. a *MethodDeclaration* contains information about its name, return type, modifiers etc. that we can use to identify the node of interest. Unfortunately, the *SearchEngine* class is available only in the JDT and our PHP extension for Alvor still has to rely on the general visitor pattern.

1.3 Bindings

An important feature in the JDT that we use in our program analysis are *bindings*. A binding defines a named entity in the Java language, e.g. packages, types, fields, methods, constructors and local variables [6]. Bindings give the picture about the structure of the program the way the compiler sees it.

Not all nodes contain binding information. The subclasses of the *ASTNode* class that do are always identifiable by their binding information. Usually they have more than one type of binding information available. For example, an instance of the *MethodInvocation* class has a binding to its return type, declaring class, types of exceptions it might throw etc.

All binding information is of great use to us when we investigate the nodes of our AST. It allows us to gather the needed information when we try to determine the possible values of our embedded SQL queries. Bindings are also being used in a number of other language specific development frameworks, not just the JDT.

1.4 String analysis

Static string analysis is a type of program analysis that allows us to compute run-time values of relevant strings occurring in the program. In our case these are the strings that are used as parameters in the SQL query functions. Because we cannot compute precise values of dynamically constructed strings at compile-time, string expressions are evaluated *abstractly* by over-approximating the set of possible string values. String analysis is *sound* if every string that may occur during the execution of the program is represented in the set of abstract strings, computed by the analyzer.

Let us consider an example:

```
String result = "SELECT * FROM ";
for (i = 0..3) {
   if (i % 2 == 0) result = result + "bar";
   else result = result + "baz";
}
```

As the result of executing the previous code we would have the string *"SELECT * FROM barbazbar"* stored in the variable *result*. If we evaluate the result abstractly it would have the following form: "*SELECT * FROM (bar|baz)**". The concrete string is included in the set of strings represented by the regular expression, thus the abstract string is a sound over-approximation of the concrete behavior of the program. As we can see, the sets of strings that may occur as the result of conditional branching and iteration have a neat and compact representation as regular expressions. We do not have to turn to more complex analysis techniques, such as computing the least fix-points, since the abstract domain has a natural way of representing iteration via the Kleene star operation on regular expressions.

Next, we can check, whether the abstract string itself is correctly formed by using a technique called abstract parsing. If the abstract string conforms to the SQL syntax, it means that every string represented by the regular expression is a syntactically correct SQL statement. Since all concrete strings are represented by this set of abstract strings, we have statically verified the correctness of the possible outcome. On the other hand, if the abstract string is not a valid SQL statement, this means that we may have identified a bug. Yet, this could also be just a false alarm, because not all elements in our abstract set are possible in real execution.

2. ECLIPSE PLATFORM

2.1 Overview of Eclipse

The Eclipse Project is an open-source project written in the Java programming language and is developed by the Eclipse Foundation [1]. Although originally Eclipse was meant to be a Java development IDE (*integrated development environment*), it can be used to build a variety of tools. The Eclipse Project tries to provide a universal platform to be “an open extensible IDE for anything and yet nothing in particular” [7].

When speaking about Eclipse we mean the Eclipse Software Development Kit (SDK), which is an IDE and a platform for building Eclipse based tools at the same time. Eclipse itself is not just one big Java program, but rather a collection of modules, called *plugins*. In the core of the Eclipse SDK is the Eclipse Platform, which is composed of the following layers that can be seen in Figure 2.1:

- Platform Runtime – the kernel of the Eclipse Platform. It discovers the needed plugins on startup and runs them using an in-memory registry. Platform Runtime is the only part of the Eclipse Platform that is not implemented as a plugin.
- Rich Client Platform – a set of tools for building Rich Client Applications, applications that make use of the underlying Eclipse Platform framework.
- Workbench IDE UI – a set of plugins that make up the basic graphical interface of the Eclipse IDE.

The Eclipse SDK also contains the before mentioned JDT (Java Development Tools) framework, plugins that make up the Eclipse Java IDE, and the *Plug-in Development Environment* (PDE): a set of plugins that enable the development of plugins for Eclipse itself. The PDE allows the programmer to conveniently develop and run plugins. It also includes wizards for creating plug-in projects, templates for generating code and specialized editors for configuring plugin’s properties. For quick testing, a new instance of Eclipse can be launched.

There is a number of materials available on the web for beginners to start writing their own Eclipse plugins, such as [8], which is a bit deprecated, but still a good starting platform for total newcomers. More thorough overview with tutorials and code samples can be found in the book by Blewitt [9].

We can always install additional components to our Eclipse installation that allow us to fulfill a variety of tasks. The Eclipse PDT (PHP Development Tools) [10] framework provides all the necessary components needed to develop PHP-based web applications. Our PHP extension for Alvor makes heavy use of the features provided by the PDT. The PDT framework can be used to perform similar program analysis as was covered for JDT in the previous chapter. There is also a number of other open-source projects and commercial products that are built on top of or integrated with the PDT framework, such as the popular Zend Studio IDE for PHP [11].

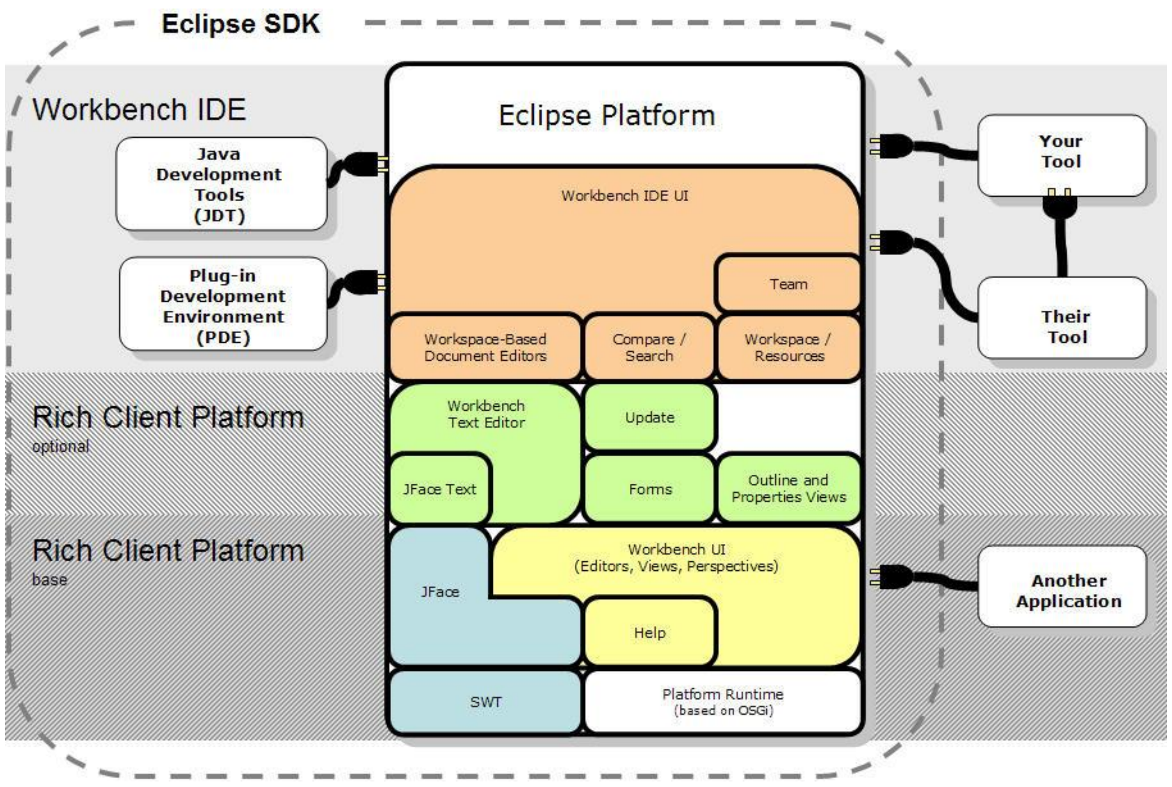


Figure 2.1: Overview of the Eclipse architecture [12]

2.2 Plugin architecture

A plugin is the smallest component of the Eclipse Platform that provides a service. It is a small Java program that extends the functionality of Eclipse in its own way. It consumes the services of other plugins or it provides its own functionalities for others to consume. The Eclipse platform loads its plugins dynamically, only on demand. This ensures that Eclipse is able to operate fast, loading and using only the components which it currently needs.

A plugin is a self-contained software component in the sense that it includes all the resources it needs to run: code, image files, resource bundles, etc. Some plugins do not contain code at all [13]. An example of this kind of a plugin is the one that provides online help in the form of HTML-pages.

Each plugin has a manifest file, *MANIFEST.MF*, which describes the plugin's dependencies and its contributions to the outside world. By contributions we mean the services and functionalities for other plugins to use. An extension management system, called *Extension Registry,* can be used to create *extension points*, which define contracts for other plugins to implement. All extension points and extensions are defined in a separate *plugin.xml* file. Using these pre-defined contracts, we can, for example, create an XML editor by extending the functionalities of a plain text editor via its publically exposed extension points. This way we create a new implementation that is completely separate of the base plugin, similar to the concept of inheritance. Doing so, we can flexibly create our tool on top of the Eclipse Platform that consists of loosely coupled, but still well integrated modules.

3. ALVOR

Alvor is a tool developed by Annamaa et. al. that statically analyzes SQL statements embedded in Java programs. Alvor is implemented as a plugin for the Eclipse IDE, using the Eclipse JDT framework. It can perform sound analysis on strings constructed using different programming features and provides feedback on both syntactic mistakes as well as semantic errors.

3.1 Alvor's architecture

In this chapter we are going to describe the technical working principles of Alvor. For more details one should consult the original article [2].

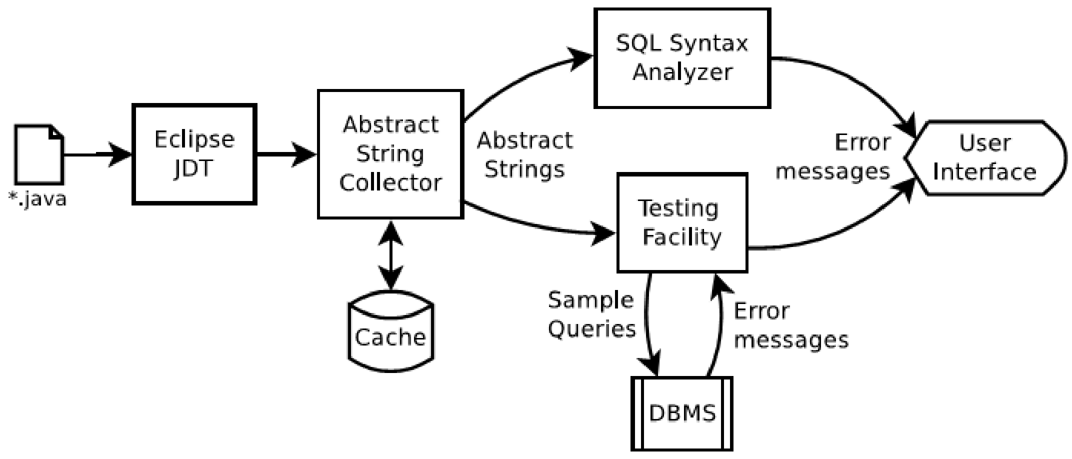


Figure 3.1: Alvor's architecture [2]

The stages of Alvor's workflow are displayed in Figure 3.1. Alvor operates within the scope of a Java project, assigned by the user. It goes through all the **.java* files located within the project's scope and its dependencies. It uses the capabilities of the Eclipse JDT to acquire the AST from the source code of each file. The AST gets then analyzed by the *Abstract String*

Collector to locate the Java expressions representing SQL statements; such expressions are referred to as *hotspots*. For each hotspot the collector evaluates the *abstract string* representation, which is a *regular expression*, representing all the possible values the hotspot can evaluate to. The entire process of locating the hotspots and evaluating their values is referred to as *crawling.*

After crawling has finished, the collected values are checked for errors, which is done in two parts. The *SQL Syntax Analyzer* parses the abstract strings for syntax errors. The *Testing Facility* generates sample queries and runs them against the database engine. If the test fails and the database returns an error, e.g. due to a misspelled or missing field name, an error is reported. Error messages from both of the analysis are displayed in the Eclipse UI similar to the Java compiler errors.

3.2 Abstract String Collector

Abstract String Collector collects the locations of hotspots and evaluates their approximate values as abstract strings. These abstract strings are in form of regular expressions, which may have the following forms: *concatenation*, *repetition*, *choice* and *parenthesized* value. The simplest form of an abstract string is a plain string or character value.

The String Collector crawls the code for certain API calls that accept SQL statements as arguments. The names of methods to search for can be defined by the user in Alvor's preferences. The corresponding abstract string values for each hotspot are evaluated through inter-procedural constant propagation analysis. To guarantee termination on recursive programs, the depth of analysis is limited.

Conditional appending of string values is represented with a choice ($a \mid b$) between two abstract strings a and b, where one of them stands for the *then* and the other for the *else* branch of the *if-else* statement. Loop-statements, such as *while* and *for,* are assumed to execute zero or more times. If we use concatenation inside the body of a loop, we can represent the resulting abstract string as a^* – an empty string or a number of repetitions of the abstract string a.

If a method parameter is being used in a hotspot expression, the resulting abstract string is going to be constructed as a choice of all possible parameter values that are gathered at the method’s call sites. If the value of the hotspot is computed using a method call, then all implementations of this method are considered. The resulting hotspot value will be a choice, built from the results of all the gathered method calls.

There exist a number of features of the host language that are not supported by the string collection algorithm, such as field variables in Java. If such an unsupported feature is encountered by the algorithm, the hotspot is considered as unsupported and a corresponding marker will be displayed in the Eclipse IDE. In case of supported hotspots, the string collection algorithm is sound and collects all the possible string values the hotspot can evaluate to.

Finally, let us consider an example SQL query embedded in Java and show how we evaluate the value of a hotspot:

```
public Statement nameFilter(String name) throws SQLException
{
  String query = "SELECT * FROM tbl WHERE name = "+name;
  return getConnection().prepareStatement(query);
}
```

We have configured Alvor to search for the JDBC API method *prepareStatement()* calls that represent hotspots. The method call has a single parameter, which is a query string, named “query”. To find its value, we start moving up the AST from the method’s call site. We see that the variable is declared in an assignment that consists of a concatenation operation. The first part of the concatenation is a string object that we do not need to evaluate any further. The second part is a variable, called *name* that we now start to evaluate. Since the only place, where we encounter the *name* variable is the parameter of the method, we need to move outside of the scope of the method and consider all the places, where the *nameFilter()* method is called. The resulting abstract string is as follows:

```
‶SELECT * FROM tbl WHERE name = ″ ‶[.*]″
```

We expect the value of the variable to be a plain string, therefore we can represent it here as the regular expression that accepts any character.

4. ALVOR FOR PHP

AlvorPHP is an extension of the original Alvor. It is an Eclipse plugin that allows us to gather SQL statements embedded in PHP programs and forward them to Alvor to check for syntactic and semantic errors. For that, it uses the Eclipse PDT framework to obtain the needed ASTs from the source code.

4.1 Modifications

Compared to the original tool, there exist a number of principal differences in our extension's architecture. The main reason for that is using PHP as the host language. Instead of **.java* files inside Java projects we are now dealing with **.php* scripts and projects with the PHP nature.

The analysis itself is now limited to the scope of the PHP program or the containing function of the hotspot. If the hotspot is located inside a PHP function, we crawl the AST and try to gather the information available until reaching the header of the function. Should we encounter a call to a function on the way, we do not follow it and stop our analysis. This is the case even if the function is the same one that we are currently operating in, i.e. recursion is also not supported.

If the hotspot is located outside a PHP function or class definition, then the scope is the containing PHP script. In that case, we climb the AST until we reach the start of the program. If we encounter any resource inclusions, we attempt to build an AST from the contents of the included file and analyze it the same way as the original file. The reason for such limitations was to keep the analysis as simple as possible.

The forms of abstract strings that we gather are the same. We have only modified the value that a loop statement may have. Loop statements are now handled as conditionals, where the *then* part consists of the body of the loop and the *else* part is simply left empty. I.e.

we expect the body of the loop to be executed one or zero times. The resulting value is a choice of abstract strings, where the *else* part is represented by the empty string. All the rest of the architecture is more or less the same as described in Chapter 3.

4.2 Implementation

Creating Alvor's PHP extension required re-factoring the existing code. Previously Alvor supported only Java as the host language and had just a single implementation that depended on the JDT. The new solution had to contain an interface instead that each language-specific extension needed to implement. The logic of AlvoPHP in terms of program analysis mostly stayed the same. Only the language constructs that we encounter during crawling are different, because now we are using PDT.

The PDT framework is closely related to the *Dynamic Languages Toolkit* (DLTK) framework [14] and uses some of its features. The DLTK framework is a set of other extensible frameworks that provide JDT-alike features for building development environments oriented for dynamic languages.

A notable problem during the implementation was the lack of proper documentation and code examples. Compared to the JDT, it was really hard to get access to the needed materials concerning DLTK and PDT.

A number of problems related to the host language had to be considered during the implementation. The biggest challenge was handling the dynamic nature of PHP. As PHP is a weakly typed language, there exists no type safety. Therefore we cannot statically check the correctness of the code before we start our analysis.

There are certain features in PHP that we are not able to properly check with our algorithm. Because of that, a number of limitations had to be introduced, although some of them already existed in the original Alvor tool as well. In the next two sub-sections we give an overview of language constructs in PHP that are supported by AlvorPHP and those that are not. The constructs and names we use to identify them are defined in the PDT.

4.3 Supported features

4.3.1 Scalar

Represents a general scalar value: *‘string’*, *1*, *1.2*, etc. When we encounter a scalar value as the parameter of our hotspot, we do not have to perform any further crawling. The value of the *Scalar* is handled as a plain string value and gets returned as the value of the hotspot.

An example: *mysql_query(“SELECT * FROM tbl”)*, where the *mysql_query()* API call is our hotspot function. The argument value is given by a *Scalar* class object and is the value of the hotspot.

4.3.2 Variables and Assignments

The *Variable* class object refers to a variable, e.g. *$a*. Variables are the constructs that we encounter the most in our code. For the variable to have any use to us it has to hold a value. A value is given to a variable in an *Assignment* class object, where the value on the right side is assigned to the variable on the left. We support two types of operands in assignments, the plain “equals” operand, e.g. *$a = “foo”,* and the “concat equals”, e.g. *$a .= “foo”*. The second one is often used when we need to build our queries dynamically.

Tracking down variables to evaluate their possible values can be tricky: a variable can exist in a wide range of expressions, not just inside assignments. Therefore, the evaluation of variables is done in a number of separate phases. We move up the AST, starting from the first encounter of the variable and use its unique *variable binding* to track its usage in the code. A variable binding can identify either a field of a class or an interface or a global/local variable declaration. If the variable is not present in the node being investigated, we skip it and move on. Finding an occurrence, we evaluate it and add its value to the resulting set of values. When we have reached the root of our search scope, the resulting set of values gets returned.

We have to look out for unassigned variables: variables that are present within the search scope, but with no value assigned or the value has been assigned in a different scope. For example, we do not have access to instance field variables. We are going to cover this later when we talk about unsupported features. This means that we can sometimes come up with nothing, when trying to track down the usage of our variable. This is the case with the widely used *superglobals*: *$_GET* and *$_POST* that are used to collect form data: we have no access to their runtime values. A common practice is using the gathered form data directly in query strings, e.g.:

```
"SELECT * FROM tbl WHERE name=".$_POST['name']
```

4.3.3 Quote

Quote is a special type of language construct available in PHP. It is a kind of hybrid, consisting of *Variable* and *Scalar* class objects. An example of such kind of construct would be the following:

```
"SELECT * FROM tbl where foo=$foo and bar=$bar"
```

Here *$foo* and *$bar* represent variables that are being used in the context of a query. To get the full value of the *Quote* construct, we have to track down the values of the variables. The resulting abstract string will be a sequence of values that we are able to gather.

This kind of variable embedding is distinctive to PHP and allows the developer to conveniently use the variables he needs, without turning to concatenation. Had we gone the other way around, this is how our query string would have looked like:

```
"SELECT * FROM tbl where foo=".$foo." and bar=".$bar
```

4.3.4 InfixExpression

Represents an infix expression, such as: *$a+1, foo()*2.* An infix expression is made up of *n* expressions and *n-1* infix operands that can be of various kinds. We support only the concatenation of strings, since this is the only operation that gives us reasonable results and which we can properly track. That is, we cannot evaluate the result of multiplication for example, due to the fact that it does not take string values as arguments. When encountering an infix expression, it gets split into parts and each sub-expression is evaluated separately. The resulting value is a sequence of strings, made up of gathered results.

4.3.5 ParenthesisExpression

Represents an expression inside (round) parenthesis, e.g. *($a = 1)*. Occasionally we have to use parenthesis to separate our logic and control the flow of the algorithm. When encountering a parenthesized expression, we discard the parenthesis and evaluate the expressions inside one after another.

4.3.6 ConditionalExpression

Represents a type of conditional expression, holding the condition, the *true* expression and the *false* expression, e.g. *$a > 0 ? $a : -$a*. Instead, we could have defined and called the following function:

```
function foo($a)
{
  if($a > 0)
     return $a;
  else
     return -$a;
}
```

Since we are dealing with a conditional here, the resulting value is a choice of values, evaluated from expressions inside the *true* and *false* branches. The *ConditionalEpression* can be directly used as the parameter in the tracked API calls, e.g. *foo($a > 0 ? $a : -$a),* because it is an expression. That does not hold for statements that represent conditionals: the *if-else* and *switch*.

4.3.7 IfStatement

Represents an *if-else* statement, e.g.:

```
if ($a > $b) {
  $sql .= "value > ".$b;
 } elseif ($a == $b) {
  $sql .= "value = ".$b;
 } else {
  $sql .= "value < ".$b;
 }
```

The previous example can also be written using alternative syntax, both of them are supported by our tool:

```
if ($a > $b) :
  $sql .= "value > ".$b;
elseif ($a == $b) :
  $sql .= "value = ".$b;
else :
  $sql .= "value < ".$b;
endif;
```

Each branch of the *if-else* statement defines a block of code, which itself is an object defined in the PDT, named *Block*. The body of the *Block* can be another *if-else* statement, i.e. the *elseif* construct.

We are unaware, which of the branches is going to be taken during runtime, therefore we have to evaluate them all. The value of an *if-else* statement is a choice of all the possible abstract string values the statement may evaluate to.

4.3.8 SwitchStatement

The *SwitchStatement* is yet another type of conditional available in PHP that may have the following form:

```
switch ($i)
{
  case 0 :
        $sql .= "value = 0";
        break;
  case 1 :
        $sql .= "value = 1";
        break;
  default :
        $sql .= "value IS NULL";
        break;
}
```

Once again we have no idea of the branch that is going to be taken, thus we have to investigate all the paths available and return their values as a choice of abstract strings.

A good thing about the API of the *SwitchStatement* defined in the PDT is that each branch is represented as a separate *SwitchCase* class object. Each *SwitchCase* has an array of statements that are defined inside its body. At the same time, the *SwitchStatement*'s counterpart in the JDT framework has rounded up all the expressions defined in the entire switch into a single array, which makes finding and using them a bit cumbersome.

4.3.9 Loop statements

Loop statements allow us to execute a block of code for a number of times. This way we can conveniently build our queries, e.g. if we need to append a number of conditions to our query string.

The following are the loop statements defined in the PDT and supported by our tool:

- *WhileStatement*
- *ForStatement*
- *DoStatement*
- *ForEachStatement*

The *while* and *for* statements are in principle the same as in any other programming language. The *DoStatement* is a special case of the *WhileStatement*, e.g.:

```
do {
   $sql .= ",".$i;
} while ($i > 0);
```

The *ForEachStatement* is the one that is distinctive to PHP. The *foreach* loop goes through a block of code for each element in an array, e.g.:

```
foreach ($arr as $i => $value)
{
   $sql .= ",".$value;
}
```

As we mentioned already, the support for loop statements is somewhat limited in AlvorPHP. We consider them as *if-else* statements, where the body of the loop represents the *then* part of an *if-else* statement. The abstract string value of a loop statement is a choice of abstract strings, consisting of an empty string and the value that we are able to gather inside the body of the loop. The reason for such solution was the fact that creating support for loops was technically quite complicated and was left for future work. Although the current solution does not provide us too much information, we still tried to give at least a limited support for the loop statements that are being used quite widely.

4.3.10 FunctionDeclaration

An object of the *FuncionDeclaration* class represents a block of code that defines the declaration of a function in our code. Should we find a hotspot inside a *FuncionDeclaration* object, then this declaration object is going to be our search scope.

During the analysis we move up the AST, starting from the location of our hotspot until we have reached the start of the function. The last thing we do is evaluate the function’s formal parameters. If there are none, we stop our analysis and return whatever we have been able to gather.

PHP has a nice feature that allows assigning default values to function parameters, as shown in the following code snippet. If we haven’t found any useful information in the function body, we could be able to gather at least something from the function signature. The problem is that when crawling the function’s parameters, we cannot always be sure that we have found what we are looking for. Consider the following code:

```
function foo($bar, $bar=”bar”)
{
  echo $bar;
}
```

Defining a function with such signature and calling it in the following manner: *foo(“bar”, “foo”)* is perfectly fine in PHP and the value “foo” gets printed as the result. Function’s formal parameters’ default values are not used very often, because most of the times we have argument values passed to our function calls.

4.3.11 Resource inclusion

A distinctive feature of PHP is the possibility to include the contents of one PHP file in another, using the specific *include()* and *require()* functions. This is a convenient way to reuse a piece of code in multiple places.

In the PDT we use the *Program* class to define a single PHP file. In terms of program analysis, if we include one *Program* class object in another, we’d have to consider the contents of the included file in our analysis as if they were present in the original file from the point of the inclusion.

The following is a brief code example that describes the principles of resource inclusions in PHP and how we handle them:

```
inc2.php
<?php
  $foo = 'inc2';
  $bar = 'inc2_val';
?>

inc1.php
<?php
  $table = 'inc1';
  $value = 'inc1_val';
  require_once 'inc2.php';
?>

test.php
<?php
  include 'inc1.php';
  mysql_query("SELECT * FROM ".$table." WHERE value = ".$value);
?>
```

Our main script here is the "test.php", which contains a *mysql_query()* function call that represents a hotspot. The possible value of the hotspot depends on the values of the *$table* and *$value* variables. We start the evaluation by looking for the usage of the *$table* variable, because we encounter it first.

On the line preceding the hotspot function is an inclusion statement that refers to a file named "inc1.php". We obtain its contents and construct the corresponding AST that we start crawling from bottom up. We are going to carry on until we are able to evaluate and return a value of our variable. We do not have to look any further, because the inclusion overwrites everything else that precedes it.

The very first statement that we encounter in "inc1.php" is yet another inclusion, referring to the "inc2.php" file this time. Therefore we change our search scope and move into "inc2.php", which does not contain any information about the variable *$table*. After reaching the start of "inc2.php" we move back into "inc1.php" and continue on from the line preceding the inclusion. Eventually, we are able to collect a value from an assignment in the "inc1.php" and return it. The very same process has to be fulfilled for the other variable as well, giving us the following value for the hotspot:

```
"SELECT * FROM inc1 WHERE value = inc1_val"
```

When including a file, we have to provide its file name. This has to be a path relative to the file that the inclusion is located in. In PHP the argument for the inclusion function may be a string, value of a variable, the result of a method or a function call, etc. In our analysis we decided to support only plain strings as filename arguments.

There might exist situations where we can run into a loop when including one file in another. Consider an example:

```
inc2.php
<?php
  $table = 'inc2';
  $value = 'inc2_val';
  require_once 'inc1.php';
?>

inc1.php
<?php
  require_once 'inc2.php';
  mysql_query("SELECT * FROM ".$table." WHERE value = ".$value);
?>
```

The problem is that when we re-include “inc1.php”, we start our analysis from the bottom of the “inc1.php” script once again. This way we will be stuck in an endless loop. To avoid situations like that we maintain a list of ASTs and check it on each inclusion. Should the list already contain the AST of the file that we are about to include we halt our analysis.

4.4 Unsupported features

As we mentioned, there is a number of PHP expressions that we do not support in our analysis. The two main reasons for that are the following:

- We are unable to evaluate the resulting value, because we do not have access to the needed runtime information. Encountering such type of expression, the analysis process for the current hotspot is halted and the corresponding place in the code is going to be marked with a marker in the Eclipse IDE.
- We are able to evaluate the expression, but it does not give us any useful information that we could further use in our analysis. This kind of expressions are just skipped because the soundness of the analysis does not depend on them. Examples of such expressions include: the *EchoStatement*, which does nothing more than just outputs info, e.g. *echo $a.*

In the following subsections we cover more thoroughly a number of unsupported PHP features and explain why they were left out.

4.4.1 Exceptions

PHP 5 has an exception model similar to other programming languages [15]. An exception can be thrown (*ThrowStatement*) and caught (*CatchStatement*) inside a *try*-block (*TryStatement*). We decided to leave exceptions out from our analysis for simplicity. They are unsupported and thus handled the same way as all the rest of unsupported features. We generally do not encounter them too often in code and there are various ways to manage without them.

4.4.2 ArrayAccess

Refers to an array access that may be in a form such as *$arr[0]*. There are various ways to manipulate the contents of an array and they are often populated using runtime values. This makes it complicated to track all of the members in an array. Therefore supporting array access would make the analysis too complicated.

4.4.3 InLineHtml

In the PDT, the *InLineHtml* class refers to HTML code. Sometimes there are situations where the PHP code is mixed with HTML tags. Although this is generally considered a bad coding practice, there are still occasions where we need to use them both in the same script. Since there is nothing that the HTML markup can provide us, it just has to be discarded.

4.4.4 FunctionInvocation

The *FunctionInvocation* class represents a function call to a previously declared function, such as *foo()*. The crawled hotspots are also instances of the *FunctionInvocation* class. As already mentioned: we do not move out of the containing function scope through the call of another function. Should we encounter a function call that has effect for the possible value of our hotspot, e.g. through assigning the value of the function call to a variable that is used in one of our hotspot functions, the analysis for the given hotspot is halted. Otherwise we carry on with the analysis.

4.4.5 MethodInvocation

MethodInvocation is a *FunctionInvocation* whose function body is defined inside a class declaration, *ClassDeclaration*. The hotspot function itself can be a *MethodInvocation*, if we have written our own separate class for database interaction. The name of the method has to be declared in AlvorPHP's preferences for it to be crawled. Otherwise, we again halt the analysis, should a method invocation somehow have influence to the possible value of our hotspot, e.g. *mysql_query($a->foo())*.

4.4.6 ReflectionVariable

ReflectionVariable represents a special type of variable in PHP, consisting of an identifier and two dollar signs, e.g. *$$a*. The PHP manual defines it as a *variable variable* that takes the value of the variable and treats that as the name of a variable [16]. Let us consider a simple example:

```
$a = "hello";
$$a = "world";

$sql = "SELECT * FROM ".$hello;
```

We have now defined two variables: *$a* with the contents "hello" and *$hello* with the contents "world". Using the variable *$hello* in the concatenation operation the value "world" gets appended to the query string. By using variable variables we can dynamically set and use a variable name.

Although this kind of language construct is very convenient to use, it is quite hard to track in our analysis. When encountering a variable variable, we first have to take its value and construct a new *Variable* instance based on its contents. The problem is that the contents of the variable may include runtime values. Even if they don't, there would be no way for us to gather the needed information about the variable binding. This information is crucial, when we crawl the code for variable usages. Since it proved to be too complex technically, we decided not to support *ReflectionVariable*.

4.4.7 ClassInstanceCreation

The *ClassInstanceCreation* allows us to create a new instance of a class, e.g. *new a*. In case we create a new instance of a class in a place where it could influence the possible outcome of our analysis, e.g. assigning a value to variable that is used in our hotspot function, the analysis of the given hotspot is halted. The reason is that we'd have to move out of our current scope into a different one.

5. EVALUATION

As we already mentioned: PHP applications interact with the user during runtime and it is quite complicated to perform static code analysis for such systems. One way users could make the most of our tool would be by replacing runtime values with static ones for testing. This way we would be able to gather a larger number of hotspots. We evaluated our tool on two different systems.

First, we used a small open source project, written for a tutorial, to test the usefulness of our tool. The tutorial was about creating a blog system in PHP from scratch [17], using the capabilities of a MySQL database. The code for the tutorial is available on Github [18]. There were altogether two hotspot patterns that we crawled, namely the functions used for database interaction. We had to slightly modify the code to be able to fully test all the supported features of AlvorPHP, e.g. resource inclusions. This did not influence the number of hotspots that we were able to gather in any way. We were able to identify and evaluate all the queries in the project, i.e. there were no unsupported hotspots. All the gathered hotspot values passed Alvor's SQL checks, there were no bugs in the queries that our extension evaluated.

Next, we intentionally created a number of situations that would eventually evaluate to an unsupported hotspot. For example, calling a function to retrieve the name of the table used in the query. In return, AlvorPHP successfully pinpointed all the unsupported features that we introduced.

The second system that we used in evaluation was Moodle [19] (version 2.8.5), an environment for internet-based courses. Moodle is also running on a MySQL database and has implemented its own separate API with a number of methods for database interaction [20]. From all those methods we were able to use five in our analysis. The reason is that our analysis did not support the more complex methods, e.g. retrieving records without using the full SQL query as the argument. The results of our evaluation can be seen in Table 5.1.

			# Hotspots			
Benchmark	LOC	Patterns	total	unsupported	supported	%
Simple Blog	972	2	14	0	14	100
Moodle	1063551	5	1018	661	357	35

Table 5.1: Evaluation results

As the results show we were able to evaluate roughly one third of all the collected hotspots in Moodle. It isn’t a bad result, when we consider the lines of code. Most of the supported hotspots passed Alvor’s SQL checking. Though, a number of false positives was returned that turned out to be still correct after double checking them manually.

Evaluation results suggest that it would be more beneficial to use AlvorPHP on smaller programs, since the code logic tends to get too complicated in bigger applications. Thus, it could be used as a tool to support beginners when learning programming.

6. FUTURE WORK

Implemented solution as it is in its current state, represents only a small subset of the possibilities that we can consider in our program analysis. There exist a number of improvements left for future work:

- The first goal should be moving outside the boundaries of the function where our hotspot is located. I.e. if we are operating within the scope of a function, we should follow function and method calls from their call sites and try to evaluate results based on the corresponding function and method declarations.
- In AlvorPHP we only had a limited support for loop constructs and no support for recursion. The goal should be evaluating them the way it is done in the original Alvor.
- We could also add a number of other language constructs that we previously did not consider, e.g. exceptions.
- A bigger challenge would be creating support for a higher level PHP framework, such as Yii [21] and Zend [22]. The problem is that PHP frameworks use sophisticated patterns to communicate with the underlying database. Such as the active record pattern [23], used in Yii. This makes the task quite complex, if not impossible.

CONCLUSION

In this master’s thesis we gave an overview of Alvor, a tool that allows us to statically check SQL queries embedded in Java programs. Alvor is implemented as a plugin for the Eclipse IDE and allows for use in interactive real-life projects. The goal of this thesis was to create an extension that would allow Alvor to perform similar syntactic and semantic checking of SQL strings inside PHP programs.

The created extension was meant as a proof-of-concept to determine how beneficial it would be to perform static analysis on code written in a language with a dynamic nature, such as PHP. We limited our analysis to the scope of a PHP function or a script to avoid making the analysis overly complicated.

We evaluated the usefulness of our extension on the Moodle environment and a simple blog system, written for a tutorial. The results showed that in the current state AlvorPHP may be used as a tool for supporting novice programmers in their studies. To be used on bigger and more complex systems, one should further develop it in the aspects that were left for future work.

Bibliography

[1] The Eclipse Foundation, "Eclipse IDE," 2015. [Online]. Available: http://www.eclipse.org. [Accessed 2015].

[2] A. Annamaa, A. Breslav, J. Kabanov and V. Vene, "An Interactive Tool for Analyzing Embedded SQL Queries," in *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010*, Shanghai, China, 2010.

[3] The Eclipse Foundation, "Eclipse Java development tools (JDT)," 2015. [Online]. Available: https://eclipse.org/jdt/. [Accessed 2015].

[4] The Eclipse Foundation, "org.eclipse.jdt.astview - AST View," 2015. [Online]. Available: https://eclipse.org/jdt/ui/astview/. [Accessed 2015].

[5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[6] The Eclipse Foundation, "Help - Eclipse platform org.eclipse.jdt.core.dom Interface IBinding," 2015. [Online]. Available: http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/IBinding.html. [Accessed 2015].

[7] The Eclipse Foundation, "Eclipse Platform Overview," 2015. [Online]. Available: https://eclipse.org/eclipse/eclipse-charter.php. [Accessed 2015].

[8] EclipsePluginSite.com, "Eclipse Plugin Development," 2008. [Online]. Available: http://www.eclipsepluginsite.com/. [Accessed 2015].

[9] A. Blewitt, "Eclipse 4 Plug-in Development by Example : Beginner's Guide", Birmingham, UK: Packt Publishing Ltd., 2013.

[10] The Eclipse Foundation, "Eclipse PHP Development Tools," 2014. [Online]. Available: https://eclipse.org/pdt/. [Accessed 2015].

[11] Zend Technologies Ltd, "The PHP IDE for Smarter Development," 2014. [Online]. Available: http://www.zend.com/en/products/studio. [Accessed 2015].

[12] R. Ramsagar, "Eclipse IDE Primer," 13 09 2009. [Online]. Available: https://tecnoesis.wordpress.com/2009/09/13/eclipse-ide-primer/. [Accessed 2015].

[13] The Eclipse Foundation, "Eclipse Platform Technical Overview," 19 04 2006. [Online]. Available: https://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html. [Accessed 2015].

[14] The Eclipse Foundation, "Dynamic Languages Toolkit," [Online]. Available: https://eclipse.org/dltk/. [Accessed 2015].

[15] The PHP Group, "PHP: Exceptions - Manual," 2015. [Online]. Available: http://php.net/manual/en/language.exceptions.php. [Accessed 2015].

[16] The PHP Group, "PHP: Variable variables - Manual," 2015. [Online]. Available: http://php.net/manual/en/language.variables.variable.php. [Accessed 2015].

[17] D. Carr, "Creating A Blog From Scratch With PHP," 07 06 2013. [Online]. Available: https://daveismyname.com/creating-a-blog-from-scratch-with-php-bp#.VVRky_mjLNF. [Accessed 2015].

[18] D. Carr, "daveismynamecom/simple-blog-part-1-build," 11 03 2014. [Online]. Available: https://github.com/daveismynamecom/simple-blog-part-1-build. [Accessed 2015].

[19] M. Dougiamas, "Moodle - Open-source learning platform," 2015. [Online]. Available: https://moodle.org/. [Accessed 2015].

[20] M. Dougiamas, "Data manipulation API - MoodleDocs," 15 08 2014. [Online]. Available: https://docs.moodle.org/dev/Data_manipulation_API. [Accessed 2015].

[21] Yii Software LLC, "Yii PHP Framework," 2015. [Online]. Available: http://www.yiiframework.com/. [Accessed 2015].

[22] Zend Technologies Ltd., "Zend Framework," 2015. [Online]. Available: http://framework.zend.com/. [Accessed 2015].

[23] M. Fowler, "Active Record," in *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002, pp. 160-163.

Appendix

User Guide

To install Alvor use the following Eclipse update-site: https://dl.bintray.com/alvor/Alvor/
NB! You may need to uncheck "Group items by category".

Guidelines for using Alvor are available at: https://bitbucket.org/plas/alvor/overview

Source code

The source code for both the original Alvor tool and its PHP extension is publicly available on Bitbucket: https://bitbucket.org/plas/alvor

The code for the AlvorPHP extension is located in the *com.googlecode.alvor.lang.php* plugin.