nMARPLE: .NET Reverse Engineering with MARPLE

Francesca Arcelli, Luca Cristina, Daniele Franzosi
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca
{arcelli|luca.cristina}@disco.unimib.it

Abstract

Research on reverse engineering and automated design pattern detection currently focuses most on some programming languages, i.e. Java and C/C++, with little interest in the .NET area. In this paper we present a tool for analyzing .NET executables to produce output compatible with the design pattern detection tool we are developing, MARPLE (Metrics and Architecture Recognition PLug-in for Eclipse).

1. Introduction

There are a lot of works in the literature and many tools have been developed in the context of reverse engineering for programming languages as *Java and C/C++*. Tools for program comprehension, design pattern detection (DPD), software architecture reconstruction. We experimented ([1]) several of these tools as Ptidej [9], Fujaba [12], DPD-Tool [16], Pinot [13], Columbus, Codecrawler [10] and we observed the lack of such kind of support for the .Net environment. We started working towards the development of a tool, called Marple [2], for DPD for Java language described in Section 3. In Marple we faced design pattern detection starting form the detection of some basic elements [3], useful in the design patterns recognition. We considered in particular the Elemental Design Patterns (EDP) [14] and the Design Patterns Clues [11] and we developed a module called *Basic Element Detector* able to recognize these kinds of components from any Java system given as input. The results provided by the Basic Elements Detector are stored in XML files, thus providing some sort of system language independence. Exploiting this facility we provided an implementation of the basic elements detector component to extend MARPLE capabilities to the .NET platform. The new engine module for .NET is called nMARPLE. The basic elements extracted from the Abstract Syntax Tree (AST) are saved into an XML file that is later processed and validated by MARPLE.

Even though nMARPLE is a porting of MARPLE on .NET platform, it has been completely re-designed to improve overall capabilities for language independence and performance. We followed an approach not based on source code but on already compiled assemblies instead, since at that level nMARPLE is able to process virtually any language designed for the .NET platform[1]. Rumors claim that today there are almost 40 languages that run on .NET platform, so ideally we would have 40 different kinds of AST to work with. It is obviously impracticable to write application logic for such a wide number of languages, hence we followed an approach based on AST generation from .NET assemblies[2].

The paper is organized trough the following sections: in Section 2 we describe some features of .NET reverse engineering, in Section 3 we describe Marple's overall architec-

[1] nMARPLE supports .NET 2.0 version and language independence is achieved working with IL language

[2] Different languages compiled to .NET assemblies share the same MSIL language (Microsoft Intermediate Language), the lowest-level human-readable programming language in the Common Language Infrastructure and in the .NET Framework

ture and in Section 4 we introduce the architecture and some integration issues of nMarple.

2. NET Reverse Engineering

Reverse engineering of .NET software must necessarily start with its conversion into a structure allowing easy handling and analysis. Processing source code directly is unfeasible, also because there are potentially dozens of languages which can be compiled into .NET-compatible executables.
The software must be converted into abstract syntax trees (ASTs), representations of code in the form of typed graphs; however, this conversion does not, by itself, solve the problem of supporting several languages. But .NET compilers do not generate machine level executable code; instead, they generate MSIL[3] code, which is then converted into machine code and executed by the .NET interpreter. MSIL is the equivalent of Java bytecode in the .NET framework, and since it is the format all .NET languages are compiled to, it could be used for reverse engineering to avoid the multi-language problem.
However, libraries supporting the loading of an AST from MSIL are not so easy to find; besides, they are usually research projects. The only one distributed under GPL is Cecil, a library allowing manipulation of ASTs and assemblies written in IL. Others, developed by Microsoft, are FxCop [7], currently included in Visual Studio; Abstract IL, still in an early phase of development, and Phoenix, which is a software optimization and analysis tool which is the basis for all future Microsoft compiler technologies. Although integration with Phoenix would grant language independence, in our tool we use the ASTs provided by FxCop, which is more widely used.

[3] MicroSoft Intermediate Language, shortly IL

3. MARPLE

MARPLE is an ongoing research project at our laboratory [15]. Its aim is to provide Design Pattern detection and metrics collection for Java software. For now we are focusing on the design pattern detection feature. We now describe the principal modules of MARPLE, whose structure is depicted in Figure 1.

3.1. Information Detector Engine

This component is responsible for analyzing source code and retrieve useful information to be used in the next phases of analysis. It is made up of two elements: the Basic Element Detector (BED) and the Metrics Collector.
The BED analyzes Java software in order to retrieve elements which, like design patterns, capture some design practice and could be useful in the identification of higher-level structures, such as design patterns. We call these structures *basic elements*. These are for now used only for DP detection, but could be employed for other purposes in the future, as for software architecture reconstruction and for collecting metrics.

The Metrics Collector, currently under development, will flank the BED in the retrieval of basic information in the form of metrics, either generic or specific for a particular purpose.

3.2. Joiner

This component [17] is responsible for analyzing the output produced by the BED and identifying sets of types which are likely to be design pattern instances. It does so by using a graph matching algorithm searching for instances of rules which correspond to the structure of the DP being searched for. The Joiner is currently under development.

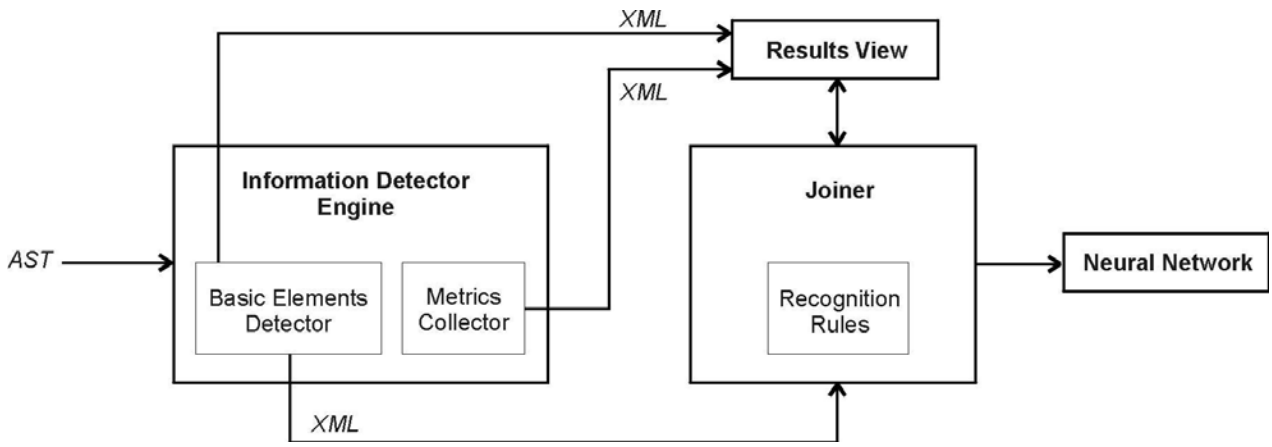

Figure 1: Architecture of MARPLE

3.3. Neural network

This module will perform the actual DP detection through supervised classification. It will receive the candidate pattern instances from the Joiner and use a previously trained Data Mining model to infer whether the candidate actually is a DP instance or not.
In [5] we reported our experiments on the results that can be obtained in classifying instances of behavioral DPs through supervised classification, using JOONE, a library for neural networks written in Java. We are currently experimenting with Weka, on creational and behavioral DPs.

4. nMARPLE

Within the MARPLE project, the role of nMARPLE [6] is to provide, for .NET languages, the same functionality as the BED, retrieving basic elements from .NET software. Here we give an example of the structures which are looked for to detect basic elements. Next we describe the integration issues identified when designing the interaction between the tool and the application using it, and briefly explain the tool's architecture.

4.1. Example of basic elements

Here we show a couple of Basic Elements, the code implementing them and the corresponding IL code. For readability IL code is reported in Figure 2.

4.1.1. Abstract Factory

Code:

```
abstract class AbstractClass{
   abstract public void fooA(){}
}
interface IMyInterface{
   void fooB();
}
```

IL:

See Figure 2a

4.1.2. Private Self Instance

Code:

```
class PrivateSelfInstance{
   private PrivateSelfInstance inst;
}
```

IL:

See Figure 2b

```
.class private auto ansi beforefieldinit AbstractClass
   extends [mscorlib]System.Object
{
}
.class private interface abstract auto ansi IMyInterface
{
}
```
(a)

```
.class private auto ansi beforefieldinit PrivateSelfInstance
   extends [mscorlib]System.Object
{
   .method public hidebysig specialname rtspecialname instance void .ctor() cil
managed
   {
   }
   .field private class PrivateSelfInstance inst
}
```
(b)

Figure 2: IL code for Abstract Interface (a) and Private Self Instance (b)

4.2. Integration issues

As said, retrieval is performed starting from .NET assemblies written in MSIL, using the FxCop library. However, due to the different technologies, .NET versus Java, integration of the tool in MARPLE is not straightforward. In an initial version, the interaction implied starting a new process, passing it the necessary parameters including source assemblies and output file, waiting for it to terminate and then retrieve the results. However this solution did not allow control of the tool from the calling application, especially for debugging purposes.

The tool was then transformed into a Web service, whose interface is described through a WSDL document. This way the service can be called through RPC.

4.3. Architecture

The tool makes use of the Visitor pattern to inspect the ASTs obtained from FxCop. In an initial version, there was a visitor for each basic element, which scanned all ASTs completely. But this approach was not efficient, since a number of full AST scans was performed, equal to the number of basic elements.

The tool was then refactored introducing the Observer pattern. Since most visitors analyzed only a small portion of the trees, they were refactored into event handlers. A new visitor was implemented, which scans the ASTs only once and fires events for each encountered node; the former visitors now register with the event generator and are notified only when it encounters nodes for which they have registered. With this performance improvement we were able to analyze the assembly mscoree.dll (Micorosft .NET Runtime Execution Engine, version 2.0) in a time the order of seconds.

5. Conclusions and future work

The development of nMARLPE has introduced two interesting innovations. The former regards the possibility to extract the basic elements from compiled assemblies allowing an operational capability also when application under analysis is not deployed with source code. nMARPLE can perform its analysis on externally referenced libraries for example with third party software components used widely in projects of medium to big complexity. The latter is about a new internal design that overcame performance issues. We have introduced a new AST parsing model based on the observer design pattern that seems offering good performances also on complex problems (for instance AST for .NET framework 2.0 it is about 770.000[4] nodes).

[4] This value is estimated using the FxCop Library

The identification of the basic elements can be used like drivers for other techniques of static analysis. Tool like MARPLE today uses EDP and Clues categories for design pattern detection, but they can be used also for the extraction of metrics or the generation of UML models. Some EDPs for instance identify important elements in UML model like create/use dependencies between classes (createObject, retrive), as well as derived association (retrive, delegates, redirect). In the class of the EDP the presence of a createObject it always implies a dependency between two classes therefore acts like a coupling metric.

Hence a possible future goal is to develop a tool for the generation of UML models at implementation level for the .NET platform, starting, again, from IL assemblies. We forecast that it will be a console application to be used in post-compilation stage in production. We would like to test it on systems under development. Initially we will disregard the graphical part of the work, and focus on a fundamental point, i.e. to generate models of software, better and more complete than those produced by other tools.

Then it would be interesting to test nMARPLE and this new tool on version 3.0 of the .NET framework, which is considerably larger than version 2.0.

We would also like to improve nMARPLE's mode of operation. First we plan to add multithreading support, to take advantage of multi-core processors.

A further improvement will be to introduce an abstraction layer for AST independence, that is, to split the analysis processo in two parts: one converting each AST to a PRG (Program Representation Graph), and another performing the actual work (basic element detection, UML diagram reconstruction) on the PRGs. This will be useful because as soon as the OMG will release the specification for ASTs, it will be enough to rewrite the mapping from AST to PRG, leaving the analysis phase unchanged.

Bibliography

[1] F. Arcelli, S. Masiero, C. Raibulet, "Elemental Design Patterns Recognition in Java", *Proc. of the 13th International Workshop on Software Technology and Engineering Practice* (STEP 2006), IEEE Computer Society, Budapest, HU, September 2006.

[2] F.Arcelli et al. The overall Marple architecture. Tech.Report University of Milano Bococca 38-10/06, December 2006.

[3] F. Arcelli, C. Raibulet, "Program Comprehension and Design Pattern Recognition: An Experience Report", *ECOOP 2006 International Workshop on Object-Oriented Reengineering* (WOOR 2006), Nantes, France, July, 4th, 2006

[4] Elliot J. Chikofsky and James H. Cross II, Reverse engineering and design recovery: a taxonomy, *IEEE Software*, 7(1):13–17, January 1990.

[5] L. Cristina, Exploiting supervised classification techniques for Design Pattern detection, Master's thesis, University of Milano – Bicocca, October 2006

[6] D. Franzosi, .NET Reverse Engineering: nMARPLE for basic elements detection, Master's thesis, University of Milano – Bicocca, 2007.

[7]FxCop,www.gotdotnet.com/Team/FxCop

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, October 1994.

[9] Yann-Gaël Guéhéneuc, Ptidej: Promoting patterns with patterns. In *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*, Springer Verlag, 2005.

[10] M.Lanza. *Object oriented Reverse Engineering*, PhD Thesis, University of Berna, 2003.

[11] S. Maggioni, Design Pattern Clues for Creational Design Patterns, *Proc, of the DPD4RE Workshop*, co-located event with

IEEE WCRE 2006 Conference, Benevento, Italy, 2006.
[12] U.A. Nickel, J.Niere, J.P. Wadsack, and A. Zündorf. Roundtrip engineering with Fujaba. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proc of 2nd Workshop on Software Reengineering* (WSR), Germany, 2000.
[13] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from Java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering*, Japan, September 2006.
[14] J. M. Smith, *An Elemental Design Pattern Catalog*, Technical Report TR02-040, Department of Computer Science, University of North Carolina at Chapel Hill, 2002.
[15] *Software Evolution and Reverse Engineering* lab at University of Milano Bicocca, www.essere.disco.unimib.it
[16] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. *In IEEE Transactions on Software Engineer*ing, volume 32, pages 896–909. IEEE, 2006.
[17] C. Tosi, M.Zanoni et al. Joiner: form subcomponents to design patterns. *Proceedings of the DPD4RE Workshop*, co-located event with IEEE WCRE 2006 Conference, Benevento, Italy, 2006.