2010 14th European Conference on Software Maintenance and Reengineering

DPDX - Towards a Common Result Exchange Format for Design Pattern Detection Tools

Günter Kniesel*, Alexander Binun*, Péter Hegedűs†, Lajos Jenő Fülöp†,
Alexander Chatzigeorgiou‡, Yann-Gaël Guéhéneuc§ and Nikolaos Tsantalis‡
*University of Bonn, Bonn, Germany; Email: gk@iai.uni-bonn.de, binun@iai.uni-bonn.de
†University of Szeged, Szeged, Hungary; Email: hpeter@inf.u-szeged.hu, flajos@inf.u-sz ged.hu
‡University of Macedonia, Thessaloniki, Greece; Email: achat@uom.gr, nikos@java.uom.gr
§École Polytechnique de Montréal, Québec, Canada; Email:yann-gael.gueheneuc@polymtl.ca

***Abstract*—Tools for design pattern detection (DPD) can ease program comprehension, helping programmers understand the design and intention of certai**

n parts of a system's implementation. Many tools have been proposed in the past. However, the many different output formats used by the tools make it difficult to compare their results and to improve their accuracy and performance through data fusion. In addition, all the output formats have been shown to have several limitations in both their forms and contents. Consequently, we develop DPDX, a rich common exchange format for DPD tools, to overcome previous limitations. DPDX provides the basis for an open federation of tools that perform comparison, fusion, visualisation, and–or validation of DPD results.

I. Introduction

Object-oriented design patterns are an important part of current design knowledge. Understanding the design patterns employed in a program provides developers with insight into the previous developers' intentions, the structure of the program, and some of its operational aspects. Therefore, design pattern detection (DPD)[1] is a helpful technique for program comprehension.

Recently, Kniesel and Binun [1] showed that the precision and recall of DPD can be improved by combining the outputs of different tools to complement results and–or balance conflictin results. However, joint use of different DPD tools was hindered by several limitations of the outputs of the current DPD tools. These limitations make results ambiguous and render it difficul to reproduce, understand, verify, compare and combine results from different tools. In particular, the lack of a common output format hinders the automated use of the results by other tools.

We propose to address these limitations by developing a common exchange format for DPD tools, DPDX, based on a well-define and extensible metamodel. This format would ease the comparison, fusion, visualisation, and validation of the outputs of different DPD tools that interact as part of a federation to produce new value.

A. Requirements

The common exchange format encourages the reporting of internal information that is available in most tools. It acts as a contract to which tools should conform in order to be part of the aforementioned federation. A suitable exchange format must fulfil the following requirements:

Specification. The exchange format must be specifie formally to allow DPD tool developers to implement appropriate generators, parsers, and–or converters.

Reproducibility. For reproducing the results, the tool and the analysed program must be explicitly reported.

Justification. To ease result assessment, the format must include explanations of results and scores expressing the confidenc of a tool in its diagnostics.

Completeness. The format must be able to represent program constituents at every level of role granularity described in design pattern literature.

Identification of role players. Each program constituent playing a role must be identifie unambiguously.

Comparability. The format must enable reporting of the motif definition assumed by a tool and the applied analysis methods to allow other tools to compare results.

B. State of the art

We have evaluated the output formats of the DPD tools that we found to be available for practical use (DP-Miner [2], Fujaba [3], Maisa [4], SSA [5], Columbus [6], PINOT [7], Ptidej [8]) and of two DPD result repositories (DEEBEE [9] and p-MARt [10]) Table I shows that most of the above requirements for the tool output are fulfille just by very few tools. This stands in sharp contrast to the fact that much of the related information is internally available in most tools. Thus we conclude that the specificatio of a common output format and the motivation of its utility for third-party tools will encourage DPD tool authors to provide the available information in the future. General purpose exchange formats such as GXL [11]are too heavyweight for this purpose and have not been adopted by any of the existing DPD tools.

II. DPDX Concepts

In this section, we develop the concepts on which our proposed exchange format, DPDX, is based. We show how DPDX addresses each of the requirements stated in Section I-A, overcoming the limitations of existing output formats.

[1]For an introduction to the used DPD terminology please refer to [1]

1534-5351/10 $26.00 © 2010 IEEE
DOI 10.1109/CSMR.2010.40

	DP-Miner	Maisa	SSA	SPQR	Colum-bus	Pinot	Ptidej	Fujaba	DEEBEE	P-Mart
Language-independence	✓	✓	✓	✓	✓	✓	✓	✓	✓	---
Completeness	---	✓	---	✓	✓	✓	---	✓	✓	---
Standard compliance	CSV	---	XML	XML	---	---	---	XML	CSV	XML
Identification of role players	---	---	Nested classes	---	Outer classes	Outer classes	Outer classes	Classes Signa-tures	Classes Methods fields	Clas-ses
Identification of Candidates	---	---	✓	✓	✓	---	✓	---	Unique IDs	---
Justification	---	---	---	---	---	---	Scores explan ations	Scores	---	---
Comparability	---	---	---	---	---	---	---	---	---	---
Reproducibility	---	---	---	---	---	---	---	---	Tool and reposito-ry info	Repos itory info
Specification	---	---	---	---	---	---	---	---	---	Role types

Table I: Tools and requirements fulfille by their output formats. For a detailed explanation of the finding see [1].

Specification.The common exchange format will be specifie by a set of extensible metamodels that capture the structural properties of the relevant concepts, e.g., candidates, roles and their relations. Metamodels that reflec the decisions explained in this section are presented in Sec.III. They significantl extend previous similar proposals, for example, the PADL metamodel of Albin-Amiot et al. [12]. The possible kinds of program constituents and the related abstract syntax tree are no first-clas elements of the metamodel but are captured by a set of predefine values for certain attributes in the metamodel. This ensures easy extensibility since only the set of values must be extended to capture new relations or language constructs whereas the metamodel and the related exchange format remain stable.

Reproducibility.A DPD result fil must contain the diagnostics of a particular DPD tool for a particular program. To enable reproducing the results, it must include the name and version of the *originating tool* and the name, version, and the URI of the *analysed program*. Names and versions may be arbitrary strings. The URI(s) must reference the root directory(ies) of the analysed program. The URI fiel is optional, since the analysed program might not be publicly accessible. The other field are mandatory.

Justification. Justificatio of diagnostics consists of confi dence scores, reported as real numbers between 0 and 1, and textual explanations. Justificatio can be added at every level of granularity: for an entire candidate, individual role assignments and individual relation assignments (see Sec. III-C).

Completeness. For completeness, the output format must support at least reporting each program constituent that can possibly play a mandatory role. Therefore, DPDX allows reporting *nested and top-level types* (interfaces, concrete and abstract classes), *fields*, *methods* and *statements (*including *field accesses* and *method invocations*). Reporting role mappings at all possible granularity levels improves the presentation of the results and eases their verificatio by experts and use by other tools. Reporting roles played by statements different from invocations and fiel accesses is important because they are essential for disambiguating certain otherwise hard to distinguish motifs. An example is given in [1].

Identification of role players. The program elements suspected to play certain roles must be identifie unambiguously in the DPD result. The identificatio scheme should be *stable*, i.e. not affected by changes in the source code that are mere formatting issues or reordering of elements whose order has no semantic meaning.[2]

Stable identificatio is easy for type and fiel declarations, which are typically named. Chaining names from outer to inner scopes is sufficien for identifying declarations of classes and fields For instance, in the example below `myApp.A` identifie the class A and `myApp.B.b` identifie the fiel b in class B:

```
package myApp;
class A {public void f(int a, int b){...}}
class B {
   int b;
   public void b(B b) {...}
   public void b(A a) {
      int c, d;

      if(...) a.f(c,d) else a.f(d,c);
   }
}
```

Because in many object-oriented languages methods can be overloaded , unique identificatio requires including the types of method arguments in the identifie of a method.

Unfortunately, nested naming is inapplicable to statements and expressions, such as the two invocations of method f() within the body of method b() above. However, every element can be identifie uniquely by a *path* in an abstract syntax tree (AST) representation of the respective program. This path consists of names for the child branches of each AST element and positions within statement sequences. For instance, the if statement in the above code example can be identifie by *ifPath* = myApp.B.b(myApp.A).body.2.

Comparability. DPDX supports comparability by specifying a precise metamodel of schemata, enabling tools to report their schemata. In addition, it provides means to specify used analysis methods and specifie a common vocabulary of analysis methods.

III. DPDX Meta-Models

This section presents the three meta-models that together specify the DPDX format: the meta-model of design pattern schemata, the meta-model of program element identifier and the meta-model of DPD results. These models reflec the decisions explained in the previous chapter.

Figure 1 shows how these models are related. Results are instances of the result metamodel. Their main part are mappings of roles and relations to program elements. Candidates are targets of such mappings. Note that candidates may overlap, that is, program elements can play a role in different pattern schemata, as illustrated by the overlap of one of the Singleton candidates with one of the Decorator candidates in Figure 1.

[2]This is necessary to compare DPD results across different program versions, when analysing the evolution of pattern implementations over time.

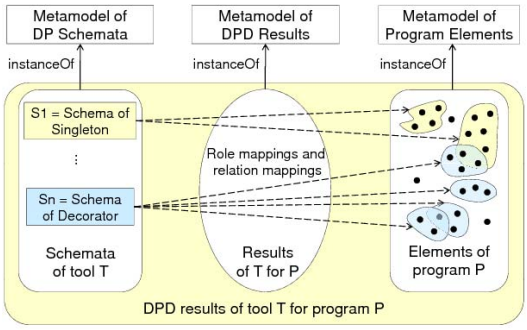

Figure 1: Relation of schemata, diagnostics and instances

A. Schema Metamodel

A *design pattern schema* is a set of named roles and named relationships between these roles. *A role* has a name, a set of associated properties, an indication of the kind of program element that may legally play that role (e.g. a class, method, etc.), a set of contained roles and a specificatio of the role cardinality, which determines how many elements that play the role may occur within the enclosing entity. Mandatory roles have cardinality greater than zero. *A relationship* has a name and cardinalities specifying how many elements that play a particular role can be related on either end of the relationship. A *role mapping* maps roles and relations of the schema to elements of a program so that the target program elements are of the required kind, have the required properties and relationships and fulfil the cardinality constraints stated in the schema.

The metamodel of design pattern schemata is illustrated in Figure 2. Apart from roles and their relations the metamodel define for each role a property as a triple consisting of a name, a value and a boolean that indicates whether the property must be met exactly or might be relaxed. In the firs case it represents a core characteristic (e.g. the 'ConcreteDecorator' role must be played by a class whose 'abstractness' property has the value concrete). Otherwise, it is ignored if not fulfille but increases the confidenc in the diagnostic if fulfille (e.g. the 'Decorator' is typically abstract but not necessarily so). The metamodel further adds the option to represent that a schema is a variant of another one, e.g. a 'Push Observer' is a variant of the 'Observer' motif.

B. Program Element Metamodel

The identificatio scheme elaborated in Sec. II distinguishes

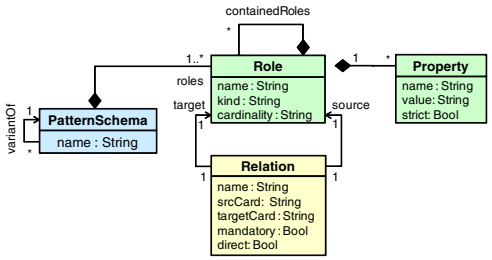

Figure 2: Metamodel of design pattern schemata

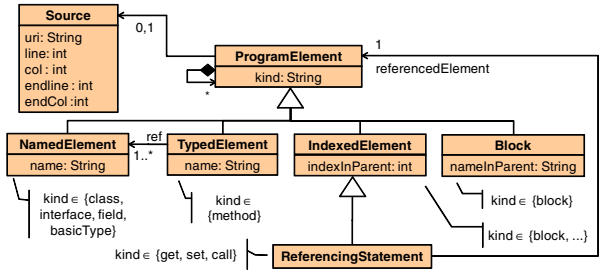

Figure 3: Metamodel of program element identifier and optional source locations

- named elements (fields classes, interfaces and primitive or built-in types),
- typed elements (method signatures),
- indexed elements (statements in a block) and
- blocks.

Each of these elements can be nested inside another element. That is general enough to accommodate even exotic languages. Although blocks and named elements look similar (both contain just a name), there is a significan distinction. The names of named elements stem from the analysed program whereas those of blocks belong to a fi ed vocabulary [13].

C. Result Metamodel

Figure 4 shows the metamodel of DPD results. A *DPD result* contains a set of diagnostics produced by a tool for a given program. Each *diagnostic* contains a set of role and relation assignments and a reference to the pattern schema whose roles and relations are mapped. Each *role assignment* references a mapped role and the program element that plays the role. A *relation assignment* references the mapped relation, a program element that serves as relation source and an element that serves as relation target. Optional justification can be added to diagnostics and each of their role and relation assignments.

IV. DPDX Implementation

For long-term maintainability, the implementations of the meta-models should rely as much as possible on emerging or

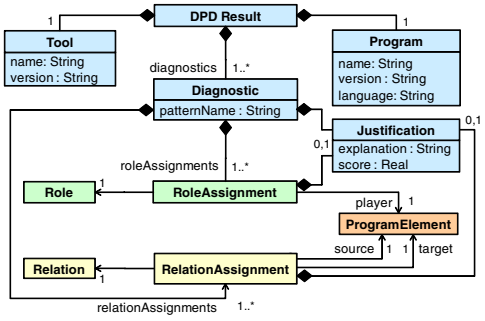

Figure 4: Metamodel of design pattern detection results

```
01<PatternSchema id ="PS1" name="Decorator" variantOf="%NONE%">
02 <Roles>
03  <Role id="R1" name="Component" kind="Class" cardinality="1">
04   <Property name="abstractness" value="abstract" strict="false"/>
05   <Role id="R2" name="Operation" kind="Method" cardinality="+"/>
06  </Role>
07  <Role id="R3" name="Decorator" kind="Class" cardinality="1">
...
16 </Roles>
17 <Relations>
18  <Relation id="RE1" name="subTypeOf" source="R3" srcCard="1"
19       target="R1" targetCard="1" mandatory="true" direct="false"/>
...
26 </Relations>
27</PatternSchema>
```

Figure 5: Implementation of schema metamodel

de-facto standards, therefore the implementation is based on XML (see Figures 5, 6 and 7).

To keep the implementation simple, we have adhered as much as possible to the following general principles for mapping meta-models to XML:

- classes of the meta-models are mapped to XML tags,
- attributes of the meta-model elements are mapped to attributes of the XML elements,
- aggregation between the elements of the meta-models are represented by the parent-children nesting technique of XML,
- an element that can be referred to by an other element has an 'id' attribute
- intentionally missing values are made explicit by special reserved values (%NONE% and %MISSING%)

```
01<ProgramElements>
02 <NamedElement id ="PE1" name="java.io.Writer" kind="class"
03      source="P1">
...
13 </NamedElement>
14 <NamedElement id ="PE4" name="java.io.BufferedWriter" kind="class"
15      source="P4">
...
42 </NamedElement>
...
47 <Sources>
48  <Source id="P1" URI="/java/io/Writer.java" line="33" col="1"
49      endLine="308" endCol="1"/>
...
54  <Source id="P4" URI="/java/io/BufferedWriter.java" line="47"
55      col="1" endLine="253" endCol="1"/>
...
66 </Sources>
67</ProgramElements>
```

Figure 6: Implementation of program metamodel

The exact rules of the XML format are define by an XML schema definition For human readability of the format, an XSLT transformation of DPDX to HTML is provided[3]. For details droped here for lack of space please check [13].

V. Conclusion

In this paper we have proposed DPDX, a common exchange format for design pattern detection tools. The proposed format is based on a well-define and extensible metamodel addressing a number of limitations of current tools. The employed

[3]See https://sewiki.iai.uni-bonn.de/dpdx/ for the XSD and XSLT code.

```
01<DPDResult >
02 <Tool name="NotNamed" version="1.0"/>
03 <Program name="JDK" version="1.6" language="Java"/>
04 <Diagnostic id="PI1" patternName="Decorator" patternSchema="PS1">
05  <RoleAssignments>
06   <RoleAssignment id="RA1" role="R1" player="PE1"/>
...
09   <RoleAssignment id="RA4" role="R3" player="PE4"/>
...
17  </RoleAssignments>
18  <RelationAssignments>
19   <RelationAssignment relation="RE1" source="PE4" target="PE1"/>
...
25  </RelationAssignments>
26  <Justifications>
27   <Justification for="PI1" score="95%" explanation=""/>
28  </Justifications>
29 </Diagnostic>
30</DPDResult>
```

Figure 7: Implementation of result metamodel

XML-based metamodel can be easily adopted by existing and future tools providing the ground for improving accuracy and recall when combining their findings

References

[1] G. Kniesel and A. Binun, "Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection," CS Department III, Uni.Bonn, Germany, Technical report IAI-TR-2009-01, ISSN 0944-8535, Jan. 2009. [Online]. Available: http://www.cs.uni-bonn.de/~gk/papers/IAI-TR-2009-01.pdf

[2] J. Dong, D. S. Lad, and Y. Zhao, "Dp-miner: Design pattern discovery using matrix," in *ECBS'07*. Washington, USA: IEEE Computer Society, 2007, pp. 371–380.

[3] L. Wendehals, "Improving design pattern instance recognition by dynamic analysis," in *WODA'03*. Portland, USA: IEEE Computer Society, 2003.

[4] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki, "Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa," *Acta Cybernetica*, vol. 15, pp. 669–682, 2002.

[5] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE TSE*, vol. 32, no. 11, pp. 896–909, 2006.

[6] Z. Balanyi and R. Ferenc, "Mining Design Patterns from C++ Source Code," in *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society, Sep. 2003, pp. 305–314.

[7] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *ASE'06*. Washington, USA: IEEE Computer Society, 2006, pp. 123–134.

[8] Y.-G. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *CASCON'04*. IBM Press, 2004, pp. 28–41.

[9] L. J. Fulop, R. Ferenc, and T. Gyimothy, "Towards a benchmark for evaluating design pattern miner tools," in *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 143–152.

[10] P-mart homepage. [Online]. Available: www.ptidej.net/downloads/pmart/

[11] A. Winter, B. Kullbach, and V. Riediger, "An overview of the GXL graph exchange language," in *Revised Lectures on Software Visualization, International Seminar*. London, UK: Springer-Verlag, 2002, pp. 324–336.

[12] H. Albin-Amiot and Y.-G. Guéhéneuc, "Meta-modeling design patterns: application to pattern detection and code synthesis," in *Proceedings of First ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[13] G. Kniesel, A. Binun, P. Hegedűs, L. J. Fülöp, N. Tsantalis, A. Chatzigeorgiou, and Y.-G. Guéhéneuc, "A common exchange format for design pattern detection tools," CS Department III, Uni.Bonn, Germany, Technical report IAI-TR-2009-03, ISSN 0944-8535, Oct. 2009. [Online]. Available: https://sewiki.iai.uni-bonn.de/dpdx/