DPDX - A Common Exchange Format for Design Pattern Detection Tools

Günter Kniesel*, Alexander Binun*, Péter Hegedus†, Lajos Jeno 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-szeged.hu
‡University of Macedonia, Thessaloniki, Greece; Email: nikos@java.uom.gr, achat@uom.gr
§École Polytechnique de Montréal, Québec, Canada; Email:yann-gael.gueheneuc@polymtl.ca

***Abstract*—Tools for design pattern detection (DPD) can significantly ease program comprehension, helping programmers understand the design and intention of certain 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. In the process of building the format, we also clarify some central notions of design patterns that lacked a common, generally accepted definitions, and thus provide a sound common foundation and terminology for DPD.**

I. Introduction

Object-oriented design patterns are an important part of current design knowledge. They offer design motifs, solutions to recurring design problems. Understanding the design patterns and motifs 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[1] (DPD) is a helpful technique for program comprehension. Building on the rich set of DPD tools available today ([1], [2]), Kniesel and Binun [3] showed that the precision and recall of the outputs of DPD tools can be improved by fusing these outputs, i.e., by combining the outputs of different tools to complement results and–or balance conflicting results.

However, fusing also revealed several limitations of the current outputs of the DPD tools, in forms and contents: some output formats (1) do not report either their own identity or the name and version of the program that they analysed; (2) do not report all roles relevant to a given motif; (3) do not identify reported roles unambiguously; (4) do not identify detected motif candidates unambiguously; (5) do not report their conceptual schema of the identified motif; (6) do not justify their results; and (7) use ad hoc (generally textual) output formats. Point 1 makes it difficult to reproduce the DPD tool results; point 2 makes it hard to combine results from different tools; points 3 and 4 make results ambiguous; point 5 renders comparison of results difficult; point 6 leads to problems when understanding and verifying the results; and, point 7 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-defined and extensible metamodel. This format would ease the comparison, fusion, visualisation, and validation of the outputs of different DPD tools. In the process of building this format, we also clarify some central notions of design patterns that lacked common, generally accepted definitions, and thus provide a sound common foundation and terminology for design pattern detection.

Consequently, the contributions of this paper are twofold: first, we provide a sound common foundation and terminology for DPD; second, we propose a common exchange format for DPD tools that fosters their synergetic use and supports automated processing of their results. In the long term, the foundation and terminology might be extended and–or complemented to accommodate other tools, for example design pattern code generators.

Section II defines a common terminology for design pattern detection, motivates the need for a common exchange format, and describes a set of requirements for the format. Section III describes the current DPD tools reported in the literature and their output formats. Section IV presents the concepts on which the proposed exchange format is built. Sections V and VI describe our common exchange format. Section VII reports an evaluation of the common exchange format while Section VIII concludes and presents future work.

II. Background

This section introduces a common terminology, motivates the need for a common exchange format, and describes a set of general requirements that an acceptable exchange format must fulfill.

A. Terminology

A *design pattern* describes a solution to a recurring design problem. A design pattern includes at least four parts: a name, a problem, a solution, and the consequences of applying the proposed solution (see Gamma et al. [4]). The solution

[1] We use the term "design pattern detection" for historical reason when we should talk about "design motif detection".

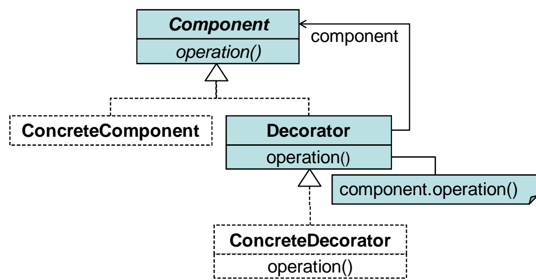


Figure 1: Motif of decorator pattern with mandatory (plain) and optional (dashed) roles

suggested by a design pattern is a *design motif* (see [5]), which describes a prototypical set of classes and–or objects collaborating to solve the design problem. A motif typically describes several *roles*, which must be fulfilled by program constituents (types, methods, fields...), their relations (inheritance, subtyping, association...), and–or their collaborations (expressed by code fragments or UML-like sequence diagrams). Roles can be *mandatory* or *optional* [6]. *Mandatory* roles (e.g., 'Composite' and 'Decorator' in the 'Decorator' motif) represent the essence of a motif. *Optional* roles might not be present in some instances (e.g., 'Concrete Decorators' may be missing). An *instance* of a design pattern *P* is a set of program constituents playing all the mandatory roles (and possibly some or all the optional roles) in the motif of P. A *candidate* of a design motif is a set of program constituents supposed to form an instance of the motif in the program and, generally, reported by a DPD tool, which typically report candidates that they deem consistent with the design motif. P[2]. Only developers can validate whether a candidate is actually an instance, i.e., is consistent with the intent, applicability, and expected consequences of *P* on the design and implementation of the program. Although textbooks typically describe explicitly just one motif per design pattern, there can be several implementation variants for each pattern, thus several design motifs to be searched for by DPD tools.

As running example in this paper, we use the 'Decorator' design pattern [4]. Fig. 1 shows the usual UML representation of its typical design motif: the name of each motif constituent (class, method, field, etc.) is not to be taken literally but reflects the role that the constituent plays in the motif. In addition, we use the convention that mandatory roles are indicated by solid lines and optional roles by dashed lines. Thus, the class-level roles 'Component' and 'Decorator', the method-level role 'operation', the field-level role 'component', and the invocation 'component.operation()' are mandatory roles in this motif. The roles 'ConcreteComponent' and 'ConcreteDecorator' are optional.

Design pattern detection, like any information retrieval task, might suffer from *false negatives* (missed instances) and *false positives* (reported candidates that are no real instances) - see [1]. When comparing tools on the same input, it is said that a tool that yields less false negatives has a better recall and one that yields less false positives has a better precision.

[2]Take this definition with a grain of salt. For a thorough definition see Sec. IV-G

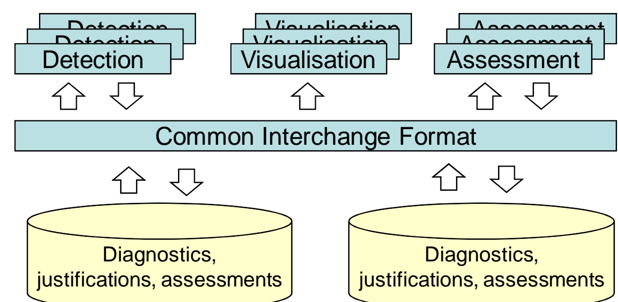


Figure 2: A federation of design pattern detection, visualisation and assessment tools cooperating via the common exchange format.

B. Motivation

A common exchange format for DPD tools would be beneficial to achieve a synergy of many different tools. Our vision is illustrated in Figure 2, where a federation of tools based on the common exchange format interact to produce new value

The figure could better emphasize the added values. You're welcome to suggest concrete ways how to improve it. . This federation and the common format is also an invitation to the program comprehension, maintenance, and reengineering research communities to contribute individual tools, including tools unforeseen in Figure 2.

For example, visualisations of DPD outputs could be built entirely using the common exchange format, instead of being implemented separately for each DPD tool. Similarly, it would be possible to automate the process of collecting, comparing, and evaluating the outputs of different tools, which is currently a manual, error-prone, and time-consuming task. Similarly, public repositories of instances of design motifs[3] would benefit greatly from a common exchange format. These repositories are important in DPD research as a reference for assesing the accuracy of tools [8]. Moreover, a common exchange format would also help in achieving an automated round-trip in DPD tools (see Albin-Amiot et al. [9]), including pattern detection, collection, fusion, visualisation, validation, storage, and generation.

C. Requirements

The common exchange format must fulfil the following core requirements to address the limitations of current DPD tools outputs and serve as the basis for a federation of tools:

1) **Specification.** The exchange format must be specified formally to allow DPD tool developers to implement appropriate generators, parsers, and–or converters.
2) **Reproducibility.** The tool and the analysed program must be explicitly reported, to allow reproducing the results.
3) **Justification.** The format must include explanations of results and scores expressing the confidence of a tool in

[3]See, for instance, PMART (http://www.ptidej.net/downloads/pmart/) and DEEBEE [7].

its diagnostics to help experts and other tools in using the reported results.

4) **Completeness.** The format must be able to represent program constituents at every level of role granularity described in design pattern literature.
5) **Identification of role players.** Each program constituent playing a role in a design motif must be identified unambiguously.
6) **Identification of candidates.** Each candidate must be identified unambiguously and reported only once.
7) **Comparability.** The format must enable reporting also the motif definitions assumed by a tool and the applied analysis methods to allow other tools to compare results.

In addition to the previous core requirements, the following two optional requirements are also desirable:

1) **Language-independence.** The common exchange format should abstract language-specific concepts so that it can be used to report candidates identified in programs written in arbitrary imperative programming languages (including in particular object-oriented languages).
2) **Standard-compliance.** The specification should be consistent with existing standards so that it can be easily adopted, maintained, and evolved.

III. State of the Art

Without aiming for completeness, we evaluate in this section the output formats of existing DPD tools with respect to the requirements collected in Section II-C. The conclusions presented below are based on intensive practical evaluations of all tools presented in [2], [6] and thorough literature review. Table I lists the analysed tools and the design patterns detected by each DPD tool.

To make similarities and differences stand out clearly all formats are presented on the common example of the 'Decorator' instance from Java IO, illustrated in Figure **??**. This is also done for tools that cannot analyse Java programs (like Columbus) or do not detect instances of the 'Decorator' pattern (like Fujaba). In such cases, we extrapolated how the format would look like if the tool supported the 'Decorator' pattern mining from Java source code. Examples containing extrapolated elements are distinguished in the section. In addition, we also reviewed the result representation format used by two DPD result repositories, PMART www.ptidej.net/downloads/pmart/ and DEEBEE [7].

a) SPQR: SPQR (System for Pattern Query and Recognition - see [10]), detects design patterns in C++ source code. The output of SPQR is presented on Figure 3.

```
1<pattern name="Decorator">
2  <role name="Component">"Writer"</role>
3  <role name="Decorator">"BufferedWriter"</role>
4  <role name="ConcreteComponent">"OutputStreamWriter"</role>
5  <role name="ConcreteDecorator">"CharCountBufferedWriter"</role>
6  <role name="operation">"close"</role>
7</pattern>
```

Figure 3: Output format of SPQR

	Abstract Factory	Adapter	Builder	Bridge	Chain of Responsibility	Command	Composite	Decorator	Facade	Factory Method	Flyweight	Mediator	Observer	Prototype	Proxy	Singleton	Strategy	State	Template Method	Visitor
SPQR								✓								✓				
DP-Miner		✓		✓			✓										✓			
Fujaba				✓		✓	✓						✓				✓	✓		✓
Maisa	✓	✓	✓							✓	✓	✓	✓	✓	✓	✓				✓
SSA		✓				✓	✓	✓		✓			✓	✓	✓	✓	✓	✓	✓	✓
Columbus	✓	✓	✓	✓	✓			✓		✓				✓	✓	✓	✓	✓	✓	✓
PINOT	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
Ptidej	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓

Table I: Design patterns detected by each tool evaluated by us

INSTANCE	Component	ConcreteComponent	Decorator	ConcreteDecorator
Decorator[0]	Writer	OutputStreamWriter	BufferedWriter	CharCountBufferedWriter
Decorator[0]	Writer	OutputStreamWriter	BufferedWriter	BlackListBufferedWriter

Table II: DP-Miner result

Due to the lack of a publicly available version of SPQR we base our evaluation on the paper presenting it [10] and on all technical reports about SPQR published at http://www.cs.unc.edu/~stotts/techreports/. Without practical evaluation our claims refer to the example used in the paper (but probably generally true for all patterns). The format fulfills *Completeness* since assignments for all mandatory roles are reported. However, it fails to fulfill *Identification of Role Players* since it does neither provide full class paths nor complete method signatures. It does not mention any information about the techniques being applied and the tool itself, therefore *Reproducibility* and *Comparability* are not fulfilled. Moreover, it does not contain information like hit probability or background information about the tool's decision so *Justification* is not fulfilled. The format is based on XML so it is *standard-compliant*.

b) DP-Miner: **DP-Miner** (see [11]) is able to present its results in a textual table format (CSV). The information provided by the tool for our 'Decorator' example is presented in Table II):

One of the main advantages of the CSV-format is that it is a quasi-standard that Microsoft Excel and many other tools can read and display. This makes it very easy for experts to interpret and evaluate design pattern results. On the other hand it fulfills only two of our requirements: *Language Independence* and *Standard Compliance*. The format contains only basic information about a design pattern instance. Clearly, it is a format for human examinations rather than for serving as a common exchange format.

c) Fujaba : Fujaba [12], [13] detects design patterns in Java source code. An example output is provided on Figure 4 (we constructed it manually since Fujaba does not detect 'Decorator' candidates).

We believe that the format fulfills *Completeness* since it reports assignments of all mandatory roles for the patterns that it supports. *Justification* is fulfilled partly since the format contains a confidence score for each pattern (but without explanations why the score is less than 100%). Explanations

```
1<StructuralAnnotation name="Decorator"
2      fuzzyBelief="56.6666666666664">
3  <BoundObject key="Component" name="java.io.Writer"/>
4  <BoundObject key="Decorator" name="java.io.BufferedWriter"/>
5  <BoundObject key="operation" name="write(char[],int,int)"/>
6</StructuralAnnotation>
```

Figure 4: Output format of Fujaba

in the form of a derivation history are available at GUI level but currently not in the textual output format. *Identification of Role Players* is fulfilled only for class role players. Complete method signatures are reported but not which method belongs to which class. This could only be deduced if we knew the schema of the respective pattern. However, schemata are not reported. The output format is based on XML, so it is *standard-compliant*. Finally, since the output does not contain information about the tool that generated it, *Reproducibility* and *Accountability* are not fulfilled.

d) Maisa: Maisa (see [14] and http://www.cs.helsinki.fi/group/maisa/) discovers design patterns based on logical constraints that are expressed by Prolog clauses. The output of the 'Decorator' pattern is presented on Figure 5. [4]:

```
Solution 0
Component = Writer
Component.Operation() = Writer.close()
Decorator = BufferedWriter
Decorator.Operation() = BufferedWriter.close()
Decorator.component = BufferedWriter.out
ConcreteComponent = OutputStreamWriter
ConcreteComponent.Operation() = OutputStreamWriter.close()
ConcreteDecorator = CharCounterBufferedWriter
ConcreteDecorator.Operation() =
CharCounterBufferedWriter.close()
ConcreteDecorator.AddedBehavior() =
CharCounterBufferedWriter.extendStream()
```

Figure 5: Output format of Maisa

Completeness is fulfilled since the format reports all role players (classes and methods) needed to reason about the 'Decorator' pattern. It fails to fulfill *Identification of Role Players* for the same reasons as SPQR. The format does not contain any background information that can be used by fusion or visualization tools, breaking the requirements of *Justification*, *Reproducibility* and *Comparability*. The format is not based on XML but on an own syntax so *Standard Compliance* is not fulfilled. Nevertheless clear role assignment technique helps the experts to evaluate the results manually.

e) Similarity Scoring (SSA): SSA (see [15]) can present identified pattern candidates on a graphical user interface but can also export the results into an XML file. So *Standard Compliance* is fulfilled. The XML file uses the following format for representing the 'Decorator' design pattern instance which is presented on Figure 6.

SSA always reports only class level roles. Therefore *Completeness* is not fulfilled by SSA. SSA reports full paths to class role players (including nested classes) therefore *Identification of Role Players* is fulfilled. SSA does not report

[4] *Maisa* can not recover *Decorator* pattern so this format is prepared by hand

```
1<pattern name="Decorator">
2 <instance>
3   <role name="Component" class="java.io.Writer"/>
4   <role name="Decorator" class="java.io.BufferedWriter"/>
5 </instance>
6</pattern>
```

Figure 6: Output format of SSA

information about the underlying techniques (or tool) therefore *Reproducibility* and *Comparability* are not fulfilled. Finally, since SSA does not report the reasons behind its findings, *Justification* is not fulfilled.

f) Columbus: The Columbus [16] framework's built-in tool for design pattern detection provides a textual output. The example on Figure 7 presents such an output.

```
Source class(es) for pattern class Component:
/src/Writer.java(50): pattern class Component =
source class Writer
/src/Writer.java(323): pattern operation Operation =
source operation close

Source class(es) for pattern class ConcreteComponent:
...
Source class(es) for pattern class Decorator:
...
Source class(es) for pattern class ConcreteDecorator:
```

Figure 7: Output format of Columbus

The output is not based on standard formats like XML thus breaking *Standard Compliance*. It fulfills the requirement of *Completeness* since classes and operations which are needed to reason about the 'Decorator' pattern are reported . The format fulfills the most important requirement, *Identification of Role Players* only partly: full file names are reported but it is unclear how nested classes can be identified this way; method signatures are not reported (just names). Note that using line numbers for identification of the pattern elements fails for two reasons. First, it is no help if the tools processing the output have no access to source code (many work on byte code only). In addition, it can be unstable: if some formatting changes that do not change the structure of the program happen in the source code the line numbers will change. This behavior is not desirable in cases when someone tries to use this identification scheme for pattern evolution. However, if source is available are a sufficient identification in one version of a software. Moreover, this additional information can be very useful for tools which visualize or evaluate design pattern instances.

As the format does not contain any background information about the reasons behind the tool's decision, the analyzed software system or the benchmark being analyzed it fails to fulfill *Justification*, *Reproducibility* and *Comparability* .

g) PINOT: PINOT [17] detects all design patterns in Java source code. An example of the PINOT output is provided on Figure 8.

The format fulfils the *Completeness* requirement since it represents all information relevant to the detected patterns - including roles at the class, field and method levels. Reasons behind the results (scores, explanations) are not provided.

```
Decorator Pattern.
Writer is the Decoratee class.
BufferedWriter is the Decorator class.
Concrete Decorator classes:
CharCounterBufferedWriter BlackListBufferedWriter
flush() is the decorate operation
Files Location:
java/io/Writer.java, java/io/BufferedWriter.java
```

Figure 8: Output format of PINOT

Therefore *Justification* is not fulfilled. *Identification of individual role players* is unambiguous only for outer classes since PINOT reports a set of files that contain the outer classes. However PINOT does not report signatures or full paths to elements below class level. The format consists of raw text strings and therefore does not conform to any standard. *Reproducibility* and *Comparability* are neither fulfilled since the output does not contain information about the tool or the techniques applied by PINOT.

h) Ptidej: Ptidej [18] detects design patterns in Java source code, but requiring it to be compiled into the byte-code. An example of Ptidej output is provided on Figure 9.

```
Decorator : 100%
component as java.io.Writer ,
concretecomponent-1 as java.io.FileWriter ,
concretecomponent-2 as java.io.ObjectWriter ,
decorator as java.io.BufferedWriter ,
concretedecorator-1 as java.io.CharCounterBufferedWriter ,
concretedecorator-2 as java.io.BlackListBufferedWriter
```

Figure 9: Output format of Ptidej

Completeness is not fulfilled since only class roles are reported (while method roles are also needed to reason about the 'Decorator' pattern). *Justification* is fulfilled since Ptidej reports not only scores but also *explanations* how a pattern instance was recognized. A pattern is expressed as a CSP problem and explanations are formulated as constraint violations/satisfactions (see [19]). *Identification of Role Players* is fulfilled since all reported role players (classes only) are identified with their full paths. The format represents raw textual phrases therefore it is not *standard-compliant*. Since the output contains justification information about the constraints being satisfied/violated, human users can guess that constraint satisfaction might be used. However, this is not made explicit in a form that a tool could use. So *Reproducibility* and *Comparability* is not fulfilled.

i) DEEBEE: DEEBEE (see [7], [2]) is a benchmark environment intended to analyze and compare the results of several DPD tools. The results of DPD tools are kept in CSV files (therefore *Standard Compliance* is fulfilled) and include classes, methods and attributes. An excerpt of such a CSV file (taken from [7]) is provided on Figure 9.

Completeness is fulfilled since all roles needed to reason about the 'Decorator' pattern (classes, methods) are reported. No information about the reasons behind decisions of DPD tools being compared therefore *Justification* is not fulfilled . *Identification of Role Players* is not fulfilled since methods

```
Decorator
class AbstractClass,Writer,Writer.java:1:6
operation flush,Writer,Writer.java:3:15
class ConcreteClass,BufferedWriter,BufferedWriter.java:8:12
operation flush,BufferedWriter,BufferedWriter.java:9:9
```

Figure 10: Input format of DEEBEE

and fields are reported without signatures. DEEBEE assigns unique IDs (so-called *tickets*) to each pattern candidate and merges several different candidates with the same mandatory role assignments (marking these candidates as *siblings*). Therefore we assume that I*dentification of Candidates* is fulfilled. DEEBEE does not keep any information about the techniques applied by DPD tools or design motifs being recognized so *Comparability* is not fulfilled. *Reproducibility*, however, is fulfilled since DEEBEE maintains the information about the tool including its version) and the code repository (including its version) for each pattern candidate.

j) P-Mart: P-Mart is a repository that keeps the results of manual DP identification from several Java repositories. An example 'Decorator' instance from P-Mart is presented in Fig. 11:

```
01<microArchitectures>
02 <microArchitecture number="76">
03 <roles>
04    <components>
05      <component roleKind="AbstractClass">
06        <entity>java.io.Writer</entity>
07      </component>
08    </components>
09    <concreteComponents>
10      <concreteComponent roleKind="Class">
11        <entity>java.io.StringWriter</entity>
12      </concreteComponent>
13    </concreteComponents>
14    <decorators>
15      <decorator roleKind="AbstractClass">
16        <entity>java.io.BufferedWriter</entity>
17      </decorator>
18    </decorators>
19    <concreteDecorators>
20      <concreteDecorator roleKind="Class">
21        <entity>./CharCountBufferedWriter</entity>
22      </concreteDecorator>
23      <concreteDecorator roleKind="Class">
24        <entity>./BlackListBufferedWriter</entity>
25      </concreteDecorator>
26    </concreteDecorators>
27   </roles>
28 </microArchitecture>
29</microArchitectures>
```

Figure 11: Format of PMART

P-Mart keeps results in XML format, therefore fulfilling *Standard Compliance*. Only class role assignments are kept therefore *Completeness* is not fulfilled. P-Mart does not keep explanations why a given DP instance is a true positive therefore *Justification* is not fulfilled. *Identification of Role Players* is fulfilled since class role players are reported with full paths (even including local classes). P-Mart does not include information about the techniques applied by DPD tools being analyzed since the DPD detection is done manually - therefore *Comparability* is not fulfilled. *Reproducibility* is fulfilled since P-Mart identifies pattern instances using repository name and

version (e.g.: JHotDraw-v.5.1, JRefactory-v.2.6)

k) Summary: Each of the reviewed output formats for describing design pattern candidates contains some elements that are worth to use in a general exchange format. In particular, each fulfills the *Language Independence* requirement, containing no language specific information. Despite that, there is no format that would fulfill all of our requirements.

Table III shows that three formats (of DP-Miner, SSA and PTidej) do not fulfill *Completeness* by not reporting all the relevant roles. Half of the tools analyzed by us use ad hoc formats, failing to fulfill *Standard Compliance*. The formats of DP-Miner and DEEBEE exchange are based on a quasi standard, CSV, but unfortunately one that fails unambiguity since even classes are not identified uniquely (only by their name). However, it is worth noting that the other standard compliant output formats structure information using XML syntax, which supports nesting and therfore recommends itself as a good basis for a general exchange format.

All formats (except the output format of SSA) either do not support *Identification of Role Players* at all or just for a limited set of program elements, mostly outer classes. Fujaba is the only one that supports classes and method / field signatures. No format supports unambiguous identification of elements at finer grained levels (individual statements). We noted that line numbers are not a satisfactory identification scheme.

Obviously, all the reviewed output formats are mainly targeted at satisfying a human expert. They assume much implicit knowledge about programming languages and design patterns that a software engineer typically has but which an automated tool has not. Typically, none of the tools provide explicit schemata of the searched design motifs, that would help another tools to understand their conceptual model of a pattern, or information about employed analysis methods. Therefore the *Comparability* requirement is not fulfilled by any tool.

DP-Miner does not fulfill *Identification of Candidates* since it can repeat the same candidate with the same role assignments multiply. *Identification of Candidates* is not fulfilled by Maisa, PINOT and Fujaba since when a candidate has several method/fields playing the same role these tools report multiple candidates (one for each method/field). SSA does not report multiple candidates in such cases. It reports only mandatory class roles, a candidate is identified unambiguously therefore *Identification of Candidates* is fulfilled.

Ptidej, DEEBEE and Columbus fulfill *Identification of Candidates* since they merge different candidates that have the same mandatory role assignments.

Reproducibility is fulfilled partly by P-Mart (only repository names and versions are included).

We should note that P-Mart reports role kinds for class roles (Class, Abstract Class etc). Therefore we could claim that *Specification* is partly fulfilled (only by P-Mart).

Last but not least, only two tools (Ptidej and Fujaba) report confidence scores and a single tool (Ptidej) provides explanations. Ptidej, which provides explanations about violated and fulfilled constraints implicitly hints at constraint satisfaction as the employed analysis technique.

	DP-Miner	Maisa	SSA	SPQR	Columbus	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 Signatures	Classes Methods fields	Classes
Identification of Candidates	—	—	✓	✓	✓	—	✓	—	Unique IDs	—
Justification	—	—	—	—	—	—	Scores explanations	Scores	—	—
Comparability	—	—	—	—	—	—	—	—	—	—
Reproducibility	—	—	—	—	—	—	—	—	—	Repository info
Specification	—	—	—	—	—	—	—	—	—	Role types

Table III: Tools and requirements fulfilled by their output formats

IV. DPDX Concepts

In this section, we develop the concepts on which our proposed exchange format, DPDX, is based. We show how DPDX adresses each of the requirements stated in Section II-C, overcomming the limitations of existing output formats identified in the previous section.

A. Specification

> *"For enabling DPD tool developers to implement appropriate exchange format file generators parsers or converters, the exchange format must be specified formally."*

The common exchange format will be specified by a set of extensible metamodels that capture the strucutral properties of the relevant concepts, e.g., candidates, roles and their relations. Metamodels that reflect the decisions explained in this section are presented in Sec.V. They significantly extend previous similar proposals, for example, the PADL metamodel of Albin-Amiot et al. [20]. However, they intentionally do not capture certain details, since a too detailed metamodel would be too rigid. In particular, we do not model the constraints that define a certain relation between roles but only the names of relations, together with a predefined but extensible set of relation names with an informally stated meaning. Similarly,we do not model the precise structure of object-oriented programming languages but just the concepts that are essential for unique identification of program elements (see Sec. IV-E). The possible kinds of program constituents and the related abstract syntax tree are no first-class elements of the metamodel but are captured by a set of predefined 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. The set of defined terms can be seen as a simple ontology. Ontologies have already been used in the domain of design pattern detection. For example, Kampffmeyer et al. [21] showed that an ontology can be

used to model the intents of design patterns. Their proposed ontology is useful to relate automatically design patterns with one another.

B. Reproducibility

> *"The tool and the analysed program must be explicitly reported, to allow reproducing the results."*

A DPD result file 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 field is optional, since the analysed program might not be publicly accessible. The other fields are mandatory.

C. Justification

> *"The format must include explanations of results and scores expressing the confidence of a tool in its diagnostics to help experts and other tools in using the reported results."*

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

D. Completeness

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

To identify a candidate unambiguously each program constituent that can possibly play a mandatory role must be reported (see IV-G). Therefore, DPDX allows reporting each of the following constituents: *nested and top-level types* (interfaces, concrete and abstract classes); *fields* and *methods*; *any statements (including in particular field accesses* and *method invocations*). *Expressions* are not distinguished from statements since many languages (notably Java and C++) let expressions contain method invocations with side-effects, thus making them only technically but not semantically different from statements[5]. Reporting role mappings at all possible granularity levels improves the presentation of the results and ease their verification by experts and use by other tools. Reporting roles played by statements different from invocations and field accesses is important because they are essential for disambiguating certain motifs. An example is given in Section VII.

[5] *Only local variables and parameters are not reported. We did not find any motif in which they would play a role. However, if the need occured, the identification scheme proposed below would apply without modifications - only the set of keywords would need to be extended.*

E. Identification of role players

> *"Each program constituent playing a role in a design motif must be identified unambiguously."*

The main part of a DPD result file consists of role mappings, i.e., assignments of program constituents to the roles that they play in a motif. Given a particular program version and program consitutent description, it must be possible to identify the constituent precisely and unambiguously in the program. In addition, it would be beneficial if the identification scheme were *stable*, i.e. if it were not affected by changes in the source code that are mere formatting issues or reordering of elements whose order has no semantic meaning. For instance, after inserting a blank line or changing the order of declarations, each program element should still have the same identifier as before. This is necessary to compare DPD results across different program versions, when analysing the evolution of design pattern implementations over time. The stability requirement rules out identification based on line numbers, as provided by some of the reviewed tools (Sec. III).

Identifying named elements: According to Sec. IV-D we must unambiguously identify program elements down to the granularity level of individual statements.

Stable identification is easy for type and field declarations, which are typically named. Chaining names from outer to inner scopes is sufficient for identifying declarations of classes and fields. For instance, in the example below `myApp.A` identifies the class A and `myApp.B.b` identifies the field 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 *** cite: Java language specificatioin, C++ Annotated Reference Manual *** , unique identification requires including the types of method arguments in the identifier of a method.

Identifying unnamed elements: Unfortunately, nested naming is inapplicable to fine-grained elements (statements and expressions), which may occur multiply in the same scope, e.g. in the same method body or field initializer expression. Cases like the two invocations of method f() within the body of method b() above can neither be disambiguated by additionally reporting the static type of invocation arguments (which is the same in both cases) nor by adding line number information (which anyway fails the stability requirement).

However, every element can be identified uniquely by a *path* in a 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. We call this the *model-based identification scheme* since it assumes a standardized model of an abstract syntax tree and standardized names for its parts. For instance, the if statement

in the above code example can be identified by *ifPath* = myApp.B.b(myApp.A).body.2. This illustrates how child elements of an already identified element are identified either by their unique, standardized name within the enclosing element (e.g."body as the name of the block representing a method body) or by their unique position inside the enclosing element (e.g. 6 a the position of the if-statement within the block). Similarly, we can distinguish the expression that represents the target of a method invocation (target) by the invocation itself (call). In a conditional statement we can distinguish elements nested inside the condition (if), the first alternative (then) and the second alternative (else). Accordingly, we can denote the invocation of f() in the first alternative by *ifpath*.then.1.call distinguishing it from the one in the second alternative, denoted *ifpath*.else.1.call .

Serving all needs.: To satisfy the diverging needs of fusion tools, visualisation tools and humans, precise hierarchical identification information is complemented with information about source code positions, if available. Source code positions contain a file path in Unix syntax (relative to the base directory indicated by the URI of the analysed program), a start position and an end position in the file, each indicated by a line and column number.

In addition, field aceesses and method invocations may be complemented by information about the accessed field or called method. For instance, the invocation of f() in the then part could also be reported as "*ifpath* . then.1.call=myApp.A.f(int,int)" Since model-based identification is unambiguous the additional information "=myApp.A.f(int,int)" is just an optional courtesy to programmers and tools who use the DPD results. It lets them know which element is referenced by the field access or method invocation without to analyse themselves the code of the source program. The class 'ReferencingStatement' in the metamodel (Sec.V-B) reflects the option to provide such additional referencing information.

DPD tools are required to support at least hierarchical naming of types, fields methods and argument types in method signatures. The source code position and the model-based identification of statements is optional, since tools based on byte code analysis will not always be able to provide it.

F. Language independence

> *The common exchange format should abstract language-specific concepts so that it can be used to report candidates identified in programs written in arbitrary imperative programming languages (including in particular object-oriented languages).*

The standardized model of an abstract syntax tree that underlies the above program element identification approach is reflected by the program element identifier metamodel in Sec.V-B and the standardized element names listed in the Appendix cover the abtract syntax of a wide range of strongly typed imperative and object-oriented languages with a name based type system (e.g. Beta, C, C++, Eiffel and Java) and dynamically typed languages (e.g. Smalltalk).

To be generally applicable, the metamodel necessarily abstracts from details that are not relevant for unique identification. Types are subsumed as named elements. This includes class and interface types but also primitive types (e.g. "int, char") and built-in types (e.g. array types in Java and pointer types in C++). Language-specific syntactic indicators for built-in types are regarded as part of the repective type name (e.g. "MyClass[][]" or "char**").

The metamodel also captures nested type definitions (e.g. nested and local classes in Java). Java's anonymous nested classes and Smalltalk's closures can simply be treated like unnamed blocks, referencing them by their position within the enclosing element. The same is true for blocks nested directly in other blocks.

G. Identification of Candidates

> *"Each candidate must be identified unambiguously and reported only once."*

Several of the reviewed tools (e.g., PINOT and DP-Miner) report multiple candidates for the same instance of a motif whose mandatory roles are played by different program constituents. For example, in the case illustrated in Figure **??**, PINOT outputs a separate Decorator candidate for each forwarding method. Reporting "related" candidates multiple times

- can confuse developers and automated tools that would use the results and leads to
- erroneous precision and recall and
- false diagnostics that could be otherwise avoided (see example in Sec. "Evaluation").

Avoiding multiple reporting of "related" candidates requires in the first place a well-defined notion of identity for candidates. Most tools do not explicitly define such a notion. Some define the conceptual identity of a candidate to be the set of values that it assigns to mandatory roles (Columbus, Ptidej and DEEBEE). However, this definition is insufficient, since it implies that two Decorator candidates that only differ in the player of the (mandatory) "Operation" role are considered to be different. However, a decorator instance may have many methods playing the "Operation" role and all the players must be reported as being part of the same instance (or candidate).

In this context the contribution of this section it a precise definition of candidates and candidate identiy and a clarification of its implications for DPD tools, the exchange format and DPD result fusion.

l) From Motifs to Schemata. : Usually, motifs are illustrated by UML templates in which role names are used like variables to be substituted by concrete names of elements from a program. For instance, Fig. 1 illustrates the design motif of the decorator pattern. This notation is intuitive enough for humans, who understand, for example, that the "operation" role is not played just by a single method and that the case illustrated in Fig. **??** is a legal instance. A tool, however, needs a more precise specification that makes *all* cardinalities and relationships explicit. In addition, the motif notation often cannot name roles properly, being constrained by the rules of

UML. For instance, an "operation" role appears in the "Component" and the "Decorator" to express the overriding relation between these methods. So the motif fails to express that the "operation" in the "Decorator" actually plays a different role in the pattern that would more accurately be called "forwarding method". For these reasons, we introduce the concept of design pattern schema as a precise and unambiguous means to specify motifs.

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 legaly play that role (e.g. a class, method, etc.), a set of contained roles and a specification of the role cardinality, which determines how many elements that play the role may occur within the encolsing entity. Mandatory roles have cardinality greater than zero. *A relationship* has a name and cardinalities specifying how many program elements that play a particular role can be related on either end of the relationship.

m) From Schemata to Candidates. : 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 fulfill the cardinality constraints stated in the schema. The essential task of DPD tools is suggesting role mappings. The set of all role mappings identified by a DPD tool for a particular schema and analysed program defines a graph with nodes being the program elements playing roles and arcs being the relations between these elements. Each relation between elements reflects a relation between the roles that the elements play. We call this graph the *projection graph*, since it represents the projection of the schema on the analysed program. A *candidate* is the set of nodes in a connected component of the projection graph. Each proper subset of a candidate is called a candidate fragment or simply *fragment*.

The identifiers of *any* program element that is part of a candidate uniquely identifies that candidate since the same element cannot occur in any other (complete) candidate. However, having possibly different identifiers for the same candidate is unsatisfactory, since it makes it hard to compare candidates based on their identifiers. Therefore, we require that every pattern schema specifies exactly one of its mandatory roles as the identifying role. The identifier of the element that plays that role in a particular candidate will identify that candidate.

These notions are illustrated in Fig. 12. Its left-hand side shows a graph that represents the core structure of the Decorator schema (only the roles and relationships are shown without their attributes - for a detailed representation see Sec. V-A). The right-hand-side illustrates that projection graph induced in some hypotetical program by a possible set of role and relation mappings. It has two connected components, corresponding to two candidates. If we assume that "Cmponent" is the identifying role, then the two candidates are uniquely identified by the classes A and C. The different colors in the A candidate illustrate possible fragments. The multiple candidates erroneously reported by PINOT and DP-Miner for one instance correspond to such fragments.

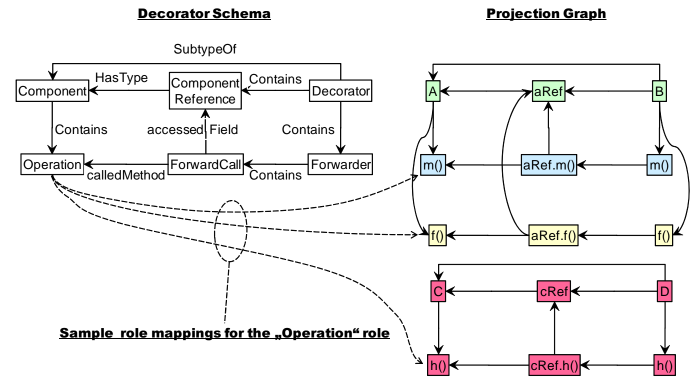


Figure 12: Illustration of candidates

The umambiguous candidate identification requirement is fulfilled if a tool reports (complete) candidates only, not fragments. This requires the exchange format to provide means of expressing mapping of a particular role to multiple players within the same candidate such as. the methods m() and f() playing both the "Operation" role in the upper candidate shown in Fig. 12. Since XML is well-suited to represent hierarchical nesting and also fulfils our standard compliance requirement, DPDX is based on XML.

H. Comparability

> *"For comparing results with those of other tools, the format must enable reporting also the design motif definitions assumed by a tool and the applied analysis methods."*

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

V. DPDX Meta-Models

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

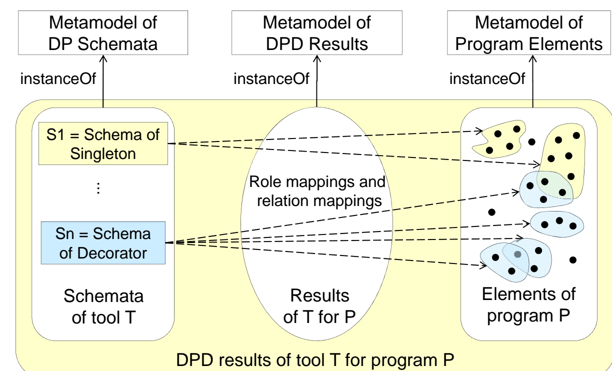


Figure 13: Relation of schemata, diagnostics and instances

Figure 13 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 (see Sec.IV-G). 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 13.

A. Schema Metamodel

The metamodel of design pattern schemata is illustrated in Figure 14a. An instance of the metamodel that represents the schema of the 'Decorator' motif is shown in Figure 14b.

The metamodel reflects the definition of schemata in Sec. IV-G and supplements it with the definition of properties as triples consisting of a name, a value and a boolean that indicates whether the property must be met exactly or might be relaxed. In the first case it represents a core characteristic (e.g. the 'ConcreteDecorator' role must be played by a class whose 'abstractness' property has the value concrete – see Fig.14b). Otherwise, it is is ignored if not fulfilled but increases the confidence in the diagnostic if fullfilled (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.

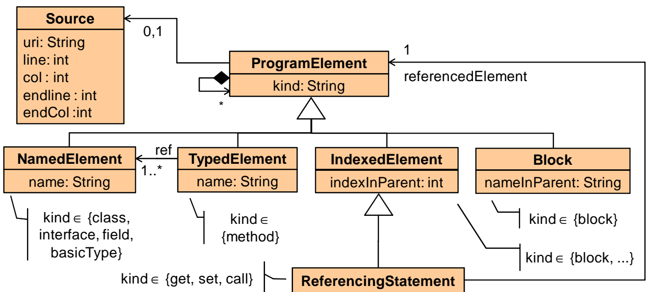
Note that the representation can accomodate arbitrary languages and the evolution of existing languages without any change in the metamodel because language level concepts (e.g. classes, methods, statements) are not first class entitites of the metamodel but just values of the 'kind' field of the Role class. In order to enable different tools understand each other, it is sufficient to agree on a common vocabulary, that is, a set of 'kind' values with a fixed meaning. For instance, the 'kind' class generally represents an object type and the distinction between interfaces, abstract classes and concrete classes is represented by the property, 'abstractness', with predefined values interface, abstract and concrete. A suggested common vocabulary is presented in the appendix.

B. Program Element Metamodel

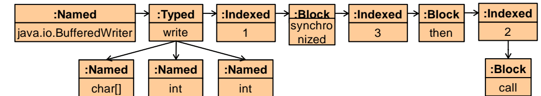
The identification scheme elaborated in Sec. IV-E distinguishes

- 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 each another element. That is general enough to accomodate even exotic languages. Although blocks and named elements look similar (both contain just a name), there is a significant distinction. The names of named elements stem from the analysed program whereas those of blocks belong to a fixed vocabulary specified in the Appendix. Each block is named after its role in the program element in which it occurs



(a) Metamodel of program element identifiers and optional source locations



(b) Representation of the marked invocation from Fig. 17

Figure 16: Metamodel of program element identifiers and sample instance

```
public void write(char cbuf[], int off, int len) ... {
  synchronized (lock) {              // 1st statement
    ensureOpen();
    if (...) ... else ...
    if (...) {                       // 3rd statement
      flushBuffer();
—>    out.write(cbuf, off, len);    // 2nd statement
      ...
    }
    ...
  }
}
```

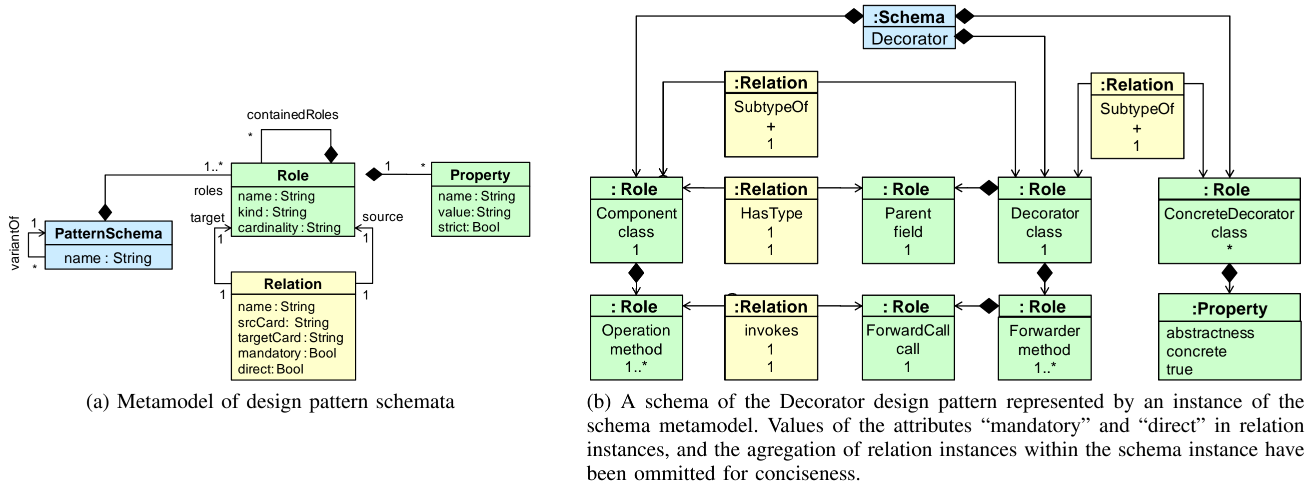
Figure 17: An excerpt of method write(char[],int,int) from class java.io.BufferedWriter

(e.g. ifCondition, then, else, whileCondition, whileBody). The 'kind' field corresponds to the one in the schema metamodel and can have the same values. Indexed elements whose kind is get, set or call can be optionally treated as referencing statements, allowing to add information about the referenced element (see Sec. IV-E). Fig. 16b illustrates the object representation of the marked invocation from Fig. 17. In the textual notation from Sec. IV it would be denoted as 'java.io.BufferedWriter.write(char[],int,int).1.synchronized.3.then.2.call'.

C. Result Metamodel

Figure 15a 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 the 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 justifications can be added to diagnostics and each of their role and relation assignments. Schemata, roles, relations and program elements are defined according to Fig. 14a and Fig. 16a.

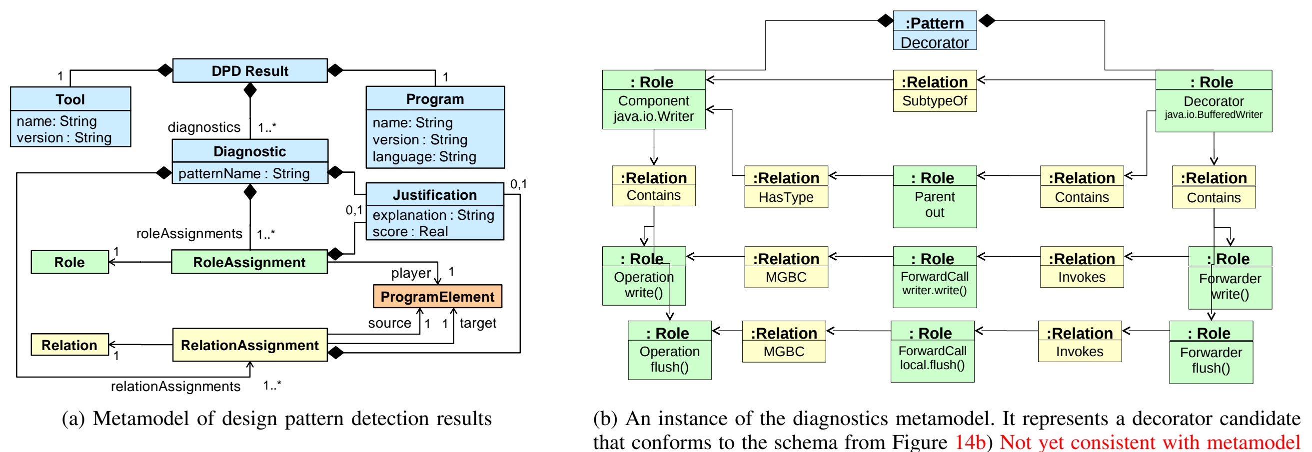
Figure 15b shows a diagnostic that is an instance of this



(a) Metamodel of design pattern schemata

(b) A schema of the Decorator design pattern represented by an instance of the schema metamodel. Values of the attributes "mandatory" and "direct" in relation instances, and the agregation of relation instances within the schema instance have been ommitted for conciseness.

Figure 14: Design pattern schemata: metamodel and sample schema



(a) Metamodel of design pattern detection results

(b) An instance of the diagnostics metamodel. It represents a decorator candidate that conforms to the schema from Figure 14b) Not yet consistent with metamodel

Figure 15: Design pattern diagnostics: metamodel and sample pattern instance

metamodel. It represents an instance of Decorator consisting of the program elements java.io.Writer, java.io.FilterWriter, java.io.Writer.write(), java.io.FilterWriter.write(), etc.

VI. DPDX Implementation

The DPDX metamodels are a common framework of reference for developing implementingtextual output of DPD tools and parsing / interpretation of this output on by users of DPD results (developers, the fusion tool, benchmarks and visualization tools) For long-term maintenability, the implementations of the meta-model should rely as much as possible on emerging or de-facto standards. Therefore we base our common exchange output format on XML. Thus XSLT can be used to transform results into a readable format or the proprietary format of individual tools. Furthermore, the rules of the format can be easily defined by XSD.

The implementation of DPDX consists of realization of the three meta-models introduced in Section V. Implementation of the Schema meta-model (see Subsection V-A) allows the tools to report the schema of the patterns they search for, the Program Element meta-model (see Subsection V-B) implementation is for identifying the program elements of the source code playing some role in the pattern instance and the Result Meta-model (see Subsection V-C) implementation describes the detected pattern instances themselves.

A. Schema Meta-model implementation

In order to support **Comparability**, DPDX must contain definition of the design pattern's schema that a tool searches for. This definition is embedded into DPDX in the 'PatternSchema' XML tag. The structure of the 'PatternSchema' must reflect the meta-model presented in 14a. For our running example, this is the XML representation of the UML diagram shown in Figure 14b.

The 'PatternSchema' tag has two children: 'Roles' and 'Relations'. These nodes describe the roles and relations between them, that a tool searches for. A 'PatternSchema' can refer to another schema by its 'variantOf' attribute, meaning that this schema is a variation of the other one (or contains the value %NONE% if the schema is not a variation). Each 'Role' tag describes a design pattern role with an unique 'id',

```
01<PatternSchema name="Decorator" variantOf="%NON%">
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">
08      <Role id="R4" name="Forwarder" kind="Method" cardinality="+">
09        <Role id="R5" name="ForwardCall" kind="Call" cardinality="1"/>
10      </Role>
11      <Role id="R6" name="Parent" kind="Field" cardinality="1"/>
12    </Role>
13    <Role id="R7" name="ConcreteDecorator" kind="Class" cardinality="*">
14      <Property name="abstractness" value="concrete" strict="true"/>
15    </Role>
16  </Roles>
17
18  <Relations>
19    <Relation id="RE1" name="subTypeOf" source="R3" srcCard="1"
20          target="R1" targetCard="1" mandatory="true" direct="false"/>
21    <Relation id="RE2" name="subTypeOf" source="R7" srcCard="1"
22          target="R3" targetCard="1" mandatory="false" direct="false"/>
23    <Relation id="RE3" name="invokes" source="R5" srcCard="1"
24          target="R2" targetCard="1" mandatory="true" direct="true"/>
25    <Relation id="RE4" name="hasType" source="R6" srcCard="1"
26          target="R1" targetCard="1" mandatory="true" direct="false"/>
27  </Relations>
28</PatternSchema>
```

Figure 18: Implementation of schema metamodel

a ‘name’, a ‘kind’ and a ‘cardinality’ attribute. The ‘kind’ attribute holds the kind of the program element that can play this role (e.g.: Class, Method, Field, etc. – see Appendix A), while the ‘cardinality’ attribute defines the legal number of this role in the instance. Optionally the ‘Role’ nodes can contain ‘Property’ elements which describe a property of a ‘Role’. An sample ‘Property’ tag can be seen in the example at the role with ‘id’ R1 (Fig. 18, line 4). A ‘Property’ has a ‘name’ (property name), ‘value’ (value of the property) and ‘strict’ (whether it is strict property or can be relaxed) attribute.

The aggregation between ‘Role’ elements is represented by XML nesting. The possible number of instances of a certain ‘Role’ is represented by the ‘cardinality’ attribute of the ‘Role’.

The relations between the roles are expressed by ‘Relation’ tags. The ‘Relation’ tag describing relations, have a ‘name’ describing the relation (e.g. invokes, subtypeOf, hasType, etc. for a complete list of possible values see Appendix A), ‘srcCard’ and ‘targetCard’ attributes describing the source and target cardinality and ‘source’ and ‘target’ attributes referring to the roles that are the sources and targets of the relation. It also contains a ‘mandatory’ and a ‘direct’ attribute. The first one determines if the relation is mandatory, the second one tells if the relation needs to be direct between the source and target or it can be a transitive relation.

B. Program Element Meta-model implementation

In this subsection we present the implementation of the Program Element Meta-model. The root node of the format is ‘ProgramElements’. ‘ProgramElements’(Figure 19, line 1) can contain any special kind of ‘ProgramElement’ (e.g. ‘NamedElement’, ‘TypedElement’, and so on) that is defined in the Program Element Meta-model. Furthermore, ‘ProgramElements’ can also contain a ‘Sources’ element which contains ‘Source’ elements representing position information in the source code. A ‘Source’ element has the following attributes to represent source code positions: ‘URI’, ‘line’, ‘col’, ‘endLine’, ‘endCol’.

```
01<ProgramElements>
02  <NamedElement id ="PE1" name="java.io.Writer" kind="class"
03        source="P1">
04    <TypedElement id="PE2" name="write" kind="method" source="P2">
05      <ref>
06        <ref ref="PE11"/>
07        <ref ref="PE12"/>
08        <ref ref="PE12"/>
09      </ref>
10    </TypedElement>
11    <TypedElement id="PE3" name="flush" kind="method" source="P3"/>
12  </NamedElement>
13
14  <NamedElement id ="PE4" name="java.io.BufferedWriter" kind="class"
15        source="P4">
16    <TypedElement id="PE5" name="write" kind="method" source="P5">
17      <ref>
18        <ref ref="PE11"/>
19        <ref ref="PE12"/>
20        <ref ref="PE12"/>
21      </ref>
22      <IndexedElement id="PE6" indexInParent="7" kind="if">
23        <Block nameInParent="if">
24          <ReferencingStatement id="PE7" indexInParent="2"
25                kind="call" referencedElement="PE2" source="P6"/>
26        </Block>
27      </IndexedElement>
28    </TypedElement>
29    <TypedElement id="PE8" name="flush" kind="method" source="P7">
30      <ReferencingStatement id="PE9" indexInParent="2" kind="call"
31            referencedElement="PE3" source="P8"/>
32    </TypedElement>
33    <NamedElement id="PE10" name="out" kind="field" source="P9"/>
34  </NamedElement>
35
36  <NamedElement id="PE11" kind="basicType" name="char[]"/>
37  <NamedElement id="PE12" kind="basicType" name="int"/>
38
39  <Sources>
40    <Source id="P1" URI="/java/io/Writer.java" line="33" col="1"
41          endLine="308" endCol="1"/>
42    <Source id="P2" URI="/java/io/Writer.java" line="128" col="5"
43          endLine="128" endCol="81"/>
44    <Source id="P3" URI="/java/io/Writer.java" line="293" col="5"
45          endLine="293" endCol="52"/>
46    <Source id="P4" URI="/java/io/BufferedWriter.java" line="47"
47          col="1" endLine="253" endCol="1"/>
48    <Source id="P5" URI="/java/io/BufferedWriter.java" line="154"
49          col="5" endLine="183" endCol="5"/>
50    <Source id="P6" URI="/java/io/BufferedWriter.java" line="169"
51          col="3" endLine="169" endCol="28"/>
52    <Source id="P7" URI="/java/io/BufferedWriter.java" line="232"
53          col="5" endLine="237" endCol="5"/>
54    <Source id="P8" URI="/java/io/BufferedWriter.java" line="235"
55          col="6" endLine="235" endCol="17"/>
56    <Source id="P9" URI="/java/io/BufferedWriter.java" line="49"
57          col="5" endLine="49" endCol="23"/>
58  </Sources>
59</ProgramElements>
```

Figure 19: Implementation of program metamodel

Most of the elements of the Program Element Meta-model are implemented directly. First of all, every ‘ProgramElement’ node can contain other ‘ProgramElement’ nodes by the nesting technique of XML. In the followings we examine the special cases of ‘ProgramElement’. ‘NamedElement’ has a ‘name’ attribute which represents it’s name, it also has a ‘source’ attribute which refers the corresponding ‘Source’ element, and it has an ‘id’ attribute which can be used to refer this ‘NamedElement’. ‘TypedElement’ has the same attributes as the ‘NamedElement’, the only difference between them that ‘TypedElement’ can contain a ‘ref’ element which refers

to the types of it's parameters ('TypedElement' represent a method). 'IndexedElement' (Figure 19, line 22) can have a 'name' attribute for naming and an 'id' attribute for referring this element. 'ReferencingStatement', the specialization of 'IndexedElement', has a 'kind' attribute to describe it's kind (get, set or call), it has a 'referencedElement' attribute to refer the referenced element (e.g. the called element) and it can has a 'source' attribute to refer the appropriate 'Source' element optionally. 'Block' element has a 'nameInParent' attribute for representing the parent statement of a the given block.

C. Result Meta-model implementation

The XML format of the result implementation reflects the structure of the pattern shown in Figure 15b as an instance of the Diagnostics meta-model (see Figure 20).

```
01<DPDResult creationDate="2009-09-28 16:04:00">
02 <Tool name="NotNamed" version="1.0"/>
03 <Program name="JDK" version="1.6" language="Java"/>
04
05 <Diagnostic id="PI1" patternName="Decorator">
06   <RoleAssignments>
07     <RoleAssignment id="RA1" role="R1" player="PE1">
08       <RoleAssignment id="RA2" role="R2" player="PE2"/>
09       <RoleAssignment id="RA3" role="R2" player="PE3"/>
10     </RoleAssignment>
11     <RoleAssignment id="RA4" role="R3" player="PE4">
12       <RoleAssignment id="RA5" role="R4" player="PE5">
13         <RoleAssignment id="RA8" role="R5" player="PE7"/>
14       </RoleAssignment>
15       <RoleAssignment id="RA6" role="R4" player="PE8">
16         <RoleAssignment id="RA9" role="R5" player="PE9"/>
17       </RoleAssignment>
18       <RoleAssignment id="RA7" role="R6" player="PE10"/>
19     </RoleAssignment>
20     <RoleAssignment id="RA10" role="R7" player="%MISSING%"/>
21   </RoleAssignments>
22
23   <RelationAssignments>
24     <RelationAssignment relation="RE1" source="PE4" target="PE1"/>
25     <RelationAssignment relation="RE3" source="PE8" target="PE2"/>
26     <RelationAssignment relation="RE3" source="PE10" target="PE3"/>
27     <RelationAssignment relation="RE4" source="PE11" target="PE1"/>
28   </RelationAssignments>
29
30   <Justifications>
31     <Justification for="RA5" score="80%" explanation=""/>
32     <Justification for="RA8" score="80%" explanation="conditional
33           forward"/>
34     <Justification for="RA10" score="" explanation="missing subclass"/>
35     <Justification for="PI1" score="95%" explanation=""/>
36   </Justifications>
37 </Diagnostic>
38</DPDResult>
```

Figure 20: Implementation of result metamodel

The root node of the DPDX format is the 'DPDResult', which contains the creation date as an attribute. The children nodes of the root are: 'Tool', 'Program', and 'Diagnostic'. By the 'Tool' tag the tool can set basic information in form of attributes about itself, for example it's 'name' and 'version'. The 'Program' element holds information about the analyzed program: 'name', 'version', and 'language'. The 'DPDResult' can contain one or more 'Diagnostic' children nodes. 'Diagnostic' nodes represent a possible occurrence of a design pattern with the name of the pattern (attribute 'patternName', 'Decorator' in our example). 'Diagnostic' nodes also have a unique 'id'.

'Diagnostic' tags can contain the following children nodes: 'RoleAssignments', 'RelationAssignments' and 'Justifications'. The first two are mandatory tags, the last one is not. The children of 'RoleAssignments' are the 'RoleAssignment' nodes, which describe the discovered roles of the candidate with the appropriate mappings to program element(s). The role mapping is based on the design pattern meta-model (for the 'Decorator' pattern model, see Figure 14b). The attributes of the 'RoleAssignment' node are the 'id', which is a unique id within the results, 'role' is a reference to a 'Role' defined in 'PatternSchema', and 'player' is the unique identifier of the program element (defined in 'ProgramElements' node) that is assigned to this role. The 'player' can hold a special value, %MISSING% (for example in Figure 20, line 20), meaning that the tool could not assign this role to any program element (despite the fact it searched the role).

The 'RelationAssignments' node collects the 'RelationAssignment' entries. These 'RelationAssignment' nodes represent the discovered relations between the found roles. Each 'RelationAssignment' tag has a 'relation' attribute referring to a 'Relation' element defined in 'PatternSchema', a 'source' and a 'target' attribute holding a reference to program elements which are related. Finally a set of 'Justification' elements are defined inside a 'Justifications' node. These optional tags can hold additional justification information for each 'RoleAssignment' (attribute 'for' refers to a 'RoleAssignment' or to a 'RelationAssignment') in terms of a confidence 'score' and a low level textual 'explanation'. If there is no 'Justification' node referring to a 'RoleAssignment' or to a 'RelationAssignment' than the assignment has a 100% confidence 'score' with no extra 'explanation' by default.

The exact rules of the XML format is defined by an XSD schema file which is available online [?].

D. Integration and visualization

The implementation of the three meta-model discussed above could have been placed into three different XML files. However, we chose to integrate them into one, since it eases the process and the visualization of DPDX. In the implementation it means, that the 'ProgramElements' and the 'PatternSchema' tags are inserted into DPDX as the children of 'DPDResult' tag.

For supporting the human readability of the format, an XSLT transformation file is also provided. It transforms DPDX files into nicely formatted HTML tables, whereby the source code of the pattern candidates can be loaded immediately. The HTML representation of the example DPDX presented in this section is available online at the DPDX homepage [22].

VII. Evaluation

In this section we show how the DPDX format eases data fusion of the outputs of different DPD tools and enables to achieve results that were not possible before. In particular, we show how our solutions to the requirements from Section II-C improve the accuracy of data fusion.

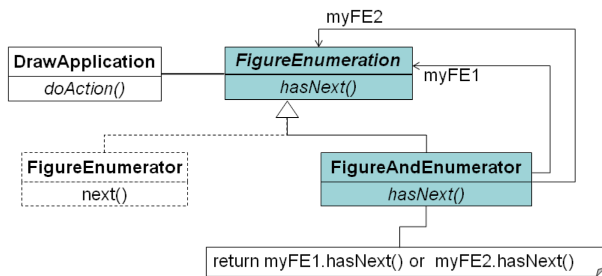


Figure 21: Improper Decorator (JHotDraw 6.0)

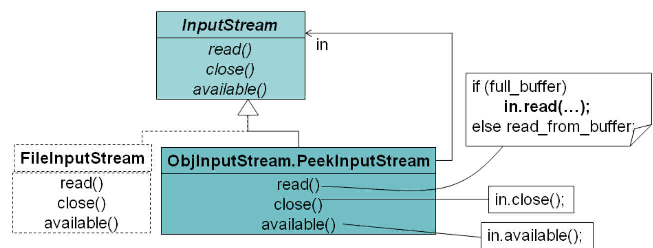


Figure 22: Decorator and Chain of Responsibility (Java IO)

A. Identification of Candidates

Figure 21 presents an example from JHotDraw 6.0. In class org.jhotdraw.standard.FigureAndEnumerator the method next() forwards to two different objects referenced by different fields of type FigureEnumeration: myFE1 and myFE2. In this case, PINOT reports two false 'Decorator' candidates, one for each field. In each candidate FigureAndEnumerator is reported as Decorator, FigureEnumeration as Component, and next() as Operation. This violates the *Identification of Candidates* requirement

Our definition of candiates (Sec. IV-G) and program element identification scheme enables a data fusion tool to detect such cases. Whenever overlapping candidate fragments are reported (either by the same or different tools) they can be joined into a single candiate. In our example this yields a 'Decorator' candidate with multiple fields playing the 'componentField' role. This helps to discover that the assumed 'Decorator' candidate is a false positive because decorators never forward to multiple immediate parents. GoF [4, , page 178] notes that 'Decorator forwards requests to its Component object. That is, forwarding to multiple parents indicates the absence of 'Decorator'.

B. Completeness and Program Element Identification

Figure 22illustrates a 'Decorator' instance from Java IO where java.io.InputStream plays the 'Component' role, the class PeekInputStream (which is a local class in java.io.ObjectInputStream) plays the 'Decorator' role and its methods available() and close() play the 'Operation' role.

In this case, PINOT, on the one hand, reports all roles necessary to reason about this Decorator (the classes and the forwarder methods) but does not report full paths to classes, making it hard to locate the nested, local class PeekInputStream. On the other hand, SSA and Ptidej report full paths to classes but does not report forwarder methods.

If the *Completeness* and *Identification of Role Players* were fulfilled in *both* tools, they would report each role player in this Decorator instance with full paths and explicitly mention the forwarder methods. A data fusion tool could then determine that identical results had been reported by both tools, thus increasing the confidence that a true positive has been found (see Kniesel and Binun [6]).

C. Completeness and Comparability

Figure 22 also illustrates an instance of the 'Chain of Responsibility' (CoR) pattern. The class java.io.InputStream plays the 'Handler' role, the class java.io.ObjectInputStream.PeekInputStream plays the 'Concrete Handler' role and the method read() plays the 'Request' role. Note the distinction between the conditional forwarding in read() which characterized the 'Request' role distinguishes it from the 'Operation' role played by the methods available() and close().

PINOT reports the 'Chain of Responsibility' instance properly, with read() as the player of the 'Request' role. This diagnosis is apparently contradicted by SSA . SSA reports 'Decorator' instead of 'Change of Responsibility' since it does not distinguish between conditional and unconditional forwarding. The outputs of SSA includes neither information about matching techniques (violating the *Comparability* requirement) nor method level roles (violating the *Completeness* requirement).

If *Completeness* and *Comparability* were fulfilled, SSA would report all roles (including methods) and in addition, the reports would include the information that the analyses employed by SSA do *not* distinguish between conditional and unconditional forwarding. Then the data fusion tool would understand that SSA does not contracict but confirms the diagnosis of PINOT, since the CoR and 'Decorator' motifs are the same except for conditional versus unconditional forwarding.

VIII. Conclusion

Design pattern detection is a significant part of the reverse engineering process that can aid program comprehension and to this end several design pattern detection tools have been developed. However, each tool reports design pattern candidates in its own format, prohibiting the comparison, validation, fusion and vizualization of their results. Apart from this limitation, each pattern identification approach employs different terms to describe concepts that underly the pattern detection process, further inhibiting their synergetic use.

In this paper we have proposed DPDX, a common exchange format for design pattern detection tools. The proposed format is based on a well-defined and extensible metamodel addressing a number of limitations of current tools. The employed XML-based metamodel can be easily adopted by existing and future tools providing the ground for improving accuracy and recall when combining their findings. Moreover,

the paper attempts to clarify central notions in the design pattern detection process providing a common foundation and terminology.

Aknowledgements

Collaborating with researchers from four different groups in writing this paper has been a rewarding experience that we will be happy to share with others.

References

[1] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *IJSEKE*, 2008.
[2] L. J. Fülöp, A. Ilia, A. Z. Vegh, and R. Ferenc, "Comparing and evaluating design pattern mining tools," in *Proceedings of SPLST '07*, 14th June 2007.
[3] 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, http://www.cs.uni-bonn.de/~gk/papers/IAI-TR-2009-01.pdf.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1994.
[5] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," *WCRE*, vol. 0, pp. 172–181, 2004.
[6] G. Kniesel and A. Binun, "Standing on the shoulders of giants – a data fusion approach to design pattern detection," in *ICPC 2009*, A. Marcus and R. Koschke, Eds. IEEE, 2009.
[7] 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.
[8] N. Pettersson, W. Löwe, and J. Nivre, "On evaluation of accuracy in pattern detection," in *First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06)*, Oct. 2006. [Online]. Available: http://cs.msi.vxu.se/papers/PLN2006a.pdf
[9] H. Albin-Amiot and Y.-G. Guéhéneuc, "Design patterns: A round-trip," in *Proceedings of 11th ECOOP Workshop for PHD students in Object-Oriented Systems*, June 2001.
[10] J. Smith and D. Stotts, "Spqr: Flexible automated design pattern extraction from source code," in *ASE'03*. IEEE, 2003. [Online]. Available: citeseer.ist.psu.edu/article/smith03spqr.html
[11] 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.
[12] L. Wendehals, "Struktur- und verhaltensbasierte entwurfsmustererkennung," PhD thesis, Universität Paderborn, Institut für Informatik, September 2007.
[13] ——, "Improving design pattern instance recognition by dynamic analysis," in *WODA'03*. Portland, USA: IEEE Computer Society, 2003.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] Y.-G. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *CASCON'04*. IBM Press, 2004, pp. 28–41.
[19] Y.-G. Guéhéneuc and N. Jussien, "Using explanations for design-patterns identification," in *IJCAI'01*. Seattle, USA: AAAI Press, Aug. 2001, pp. 57–64. [Online]. Available: http://www.emn.fr/jussien/publications/gueheneuc-WIJCAI01.pdf
[20] 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.
[21] H. Kampffmeyer and S. Zschaler, "Finding the pattern you need: The design pattern intent ontology." in *MoDELS*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 211–225. [Online]. Available: http://dblp.uni-trier.de/db/conf/models/models2007.html#KampffmeyerZ07
[22] Dpdx homepage. [Online]. Available: http://www.sed.hu/dpdx/Decorator_Writer.dpdx
[23] Wiki page of dpdx. [Online]. Available: https://sewiki.iai.uni-bonn.de/research/dpd/

Appendix

A. DPDX attribute values

In order to make our DPDX format really beneficial, the tools (and people) using the the format must agree on a common terminology for keywords applied in DPDX. This is important because XML attributes are just strings, but we need a special meaning for some strings (e.g.: relation name can not be an arbitrary string but a discrete value from a known relation types set). So it is essential that all users of the format use the same keyword for describing the same concept. Otherwise the primary aim of the common exchange format, to make the design pattern instances comparable, would brake. For that very reason we would like to make a proposal for all the researchers interested in using (or maybe developing) DPDX: as a primary source of the existing DPDX attribute values use our publicly available WIKI page [23]. Every researcher is very welcome to extend the available keywords if there is no suitable one already. This page allows us avoiding the use of different keywords with the same meaning, moreover the introduction of new keywords would base on a common consensus.

As an example we collected a reference list of possible keywords used as attribute values in DPDX, which is presented in Table IV.

Tag name	**Attribute name**	**Possible attribute values**
Role	*kind*	'Class', 'Method', 'Field', 'Call'
Property	*name*	'abstractness', 'visibility' 'staticness'
Relation	*relationName*	'subtypeOf', 'hasType', 'invokes'
Block	*nameInParent*	'if', 'then', 'else', 'whileloop', 'dowhileloop', 'dountilloop', 'forloop', 'switch'
Block	*kind*	'block'
IndexedElement	*kind*	'block', 'if', 'then', 'else', 'whileloop', 'dowhileloop', 'dountilloop', 'forloop', 'switch'
NamedElement	*kind*	'class', 'interface', 'field', 'basicType'
TypedElement	*kind*	'method'
ReferencingStatement	*kind*	'get', 'set', 'call'

Table IV: Attribute values of DPDX

Each Property tag with different name attribute, has a different set of possible kind values. These values are collected in Table V.

The listed values can be used language independently, although the keywords used in the tables follow the Java terminology. We prefer this solution instead of creating a

Property name	Property attribute	Possible attribute values
'abstractness'	*kind*	'abstract', 'interface', 'concrete'
'visibility'	*kind*	'public', 'protected', 'private'
'staticness'	*kind*	'static', 'non-static'

Table V: Property values

whole new terminology. We chose Java for base because of its current popularity. Despite the equivalency in terminology, our keywords have a more general meaning. They can be interpreted for various different programming languages. For example the property value *abstract* (which is a Java keyword) can be used in DPDX describing C++ design pattern instances also. We have different interpretations of the keywords for different languages, e.g.: the *abstract* property of a Java class means the class is defined using the abstract Java keyword, but in case of a C++ class, it means that the class has at least one pure virtual method (but not all of them are pure virtual, since it would make the class an *interface*).

Make the following consistent with dpd-uml-metamodels.pptx page 22