DPDX - A Common 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-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 II-D describes the current DPD tools reported in the literature and their output formats. Section III presents the concepts on which the proposed exchange format is built. Sections IV and V describe our common exchange format. Section VI reports an evaluation of the common exchange format while Section VII 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".

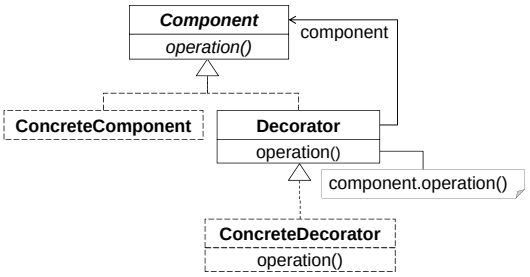


Figure 1: Motif of decorator pattern with mandatory (plain) and optional (dashed) roles

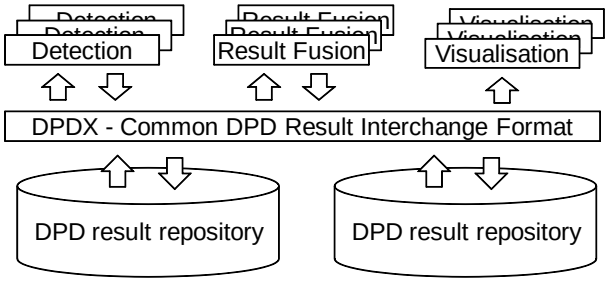


Figure 2: A federation of design pattern detection, visualisation and assessment tools cooperating via the common exchange format.

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.

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.

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. 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 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.

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

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

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.

D. State of the art

We have evaluated the output formats of several existing DPD tools (SPQR [20], DP-Miner [11], Fujaba [12], [13], Maisa [14], [15], SSA [16], Columbus [17], PINOT [18], Ptidej [19]) and of two DPD result repositories (DEEBEE [7], [2] and p-MARt [21]) with respect to the requirements collected in Section II-C. The conclusions presented below are based not just on thorough literature review but also intensive practical evaluations [2], [6], [10] of all tools except SPQR, which is not publicly available.

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 I shows that three formats (of DP-Miner, SSA, PTidej and P-Mart) 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 therefore 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	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 fulfilled by their output formats

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. By reporting only mandatory class roles, a candidate is identified unambiguously therefore *Identification of Candidates* is fulfilled.

Ptidej, DEEBEE, Columbus and SPQR 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) and DEEBEE. 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.

III. 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

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.IV. They significantly extend previous

similar proposals, for example, the PADL metamodel of Albin-Amiot et al. [22]. 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. [23] 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

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

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. IV-C).

D. Completeness

To identify a candidate unambiguously each program constituent that can possibly play a mandatory role must be reported (see III-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*). 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 VI.

E. Identification of role players

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.

Identifying named elements: According to Sec. III-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 , 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). 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.IV-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 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.IV-B and a set of standardized element names (see [10]) 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). The metamodel abstracts from details that are not relevant for unique identification. Types are subsumed as named elements.

G. Identification of Candidates

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, PINOT outputs a separate Decorator candidate for each forwarding method if multiple methods play the 'Operation' role. 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.

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.

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. 3. 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. IV-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.

The umambiguous candidate identification requirement is fulfilled if a tool reports (complete) candidates only, not

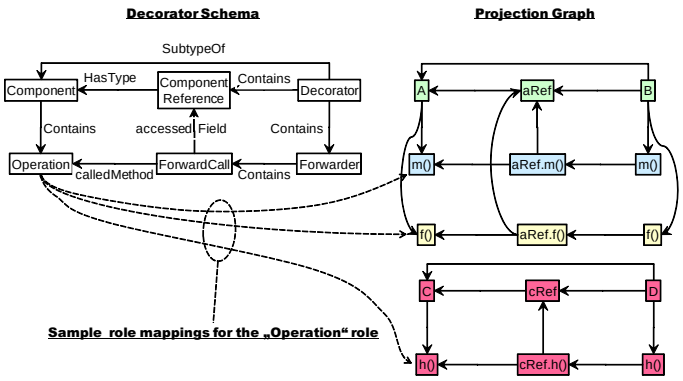


Figure 3: Illustration of candidates

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. 3. Since XML is well-suited to represent hierarchical nesting and also fulfils our standard compliance requirement, DPDX is based on XML.

H. Comparability

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.

IV. 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.

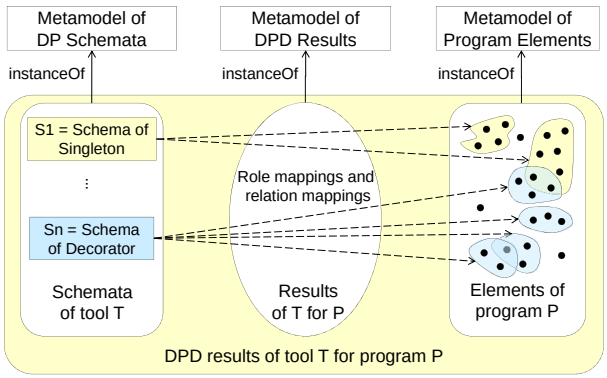


Figure 4: Relation of schemata, diagnostics and instances

Figure 4 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.III-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 4.

A. Schema Metamodel

The metamodel of design pattern schemata is illustrated in Figure 5a. An instance of the metamodel that represents the schema of the ‘Decorator’ motif is shown in Figure 5b.

The metamodel reflects the definition of schemata in Sec. III-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.5b). 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.

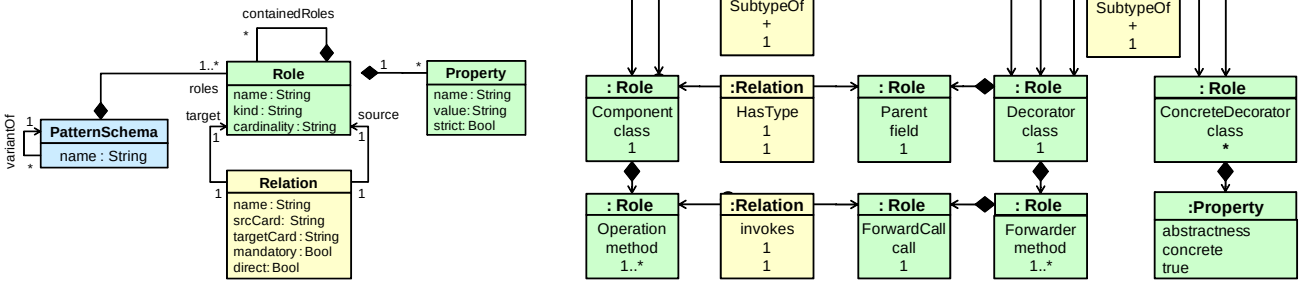
Note that the representation can accomodate arbitrary languages and the evolution of existing languages without any change in the metamodel because language level concepts (e.g. classes, methods, statements) are not first class entitites of the metamodel but just values of the ‘kind’ field of the Role 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 [10].

B. Program Element Metamodel

The identification scheme elaborated in Sec. III-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.

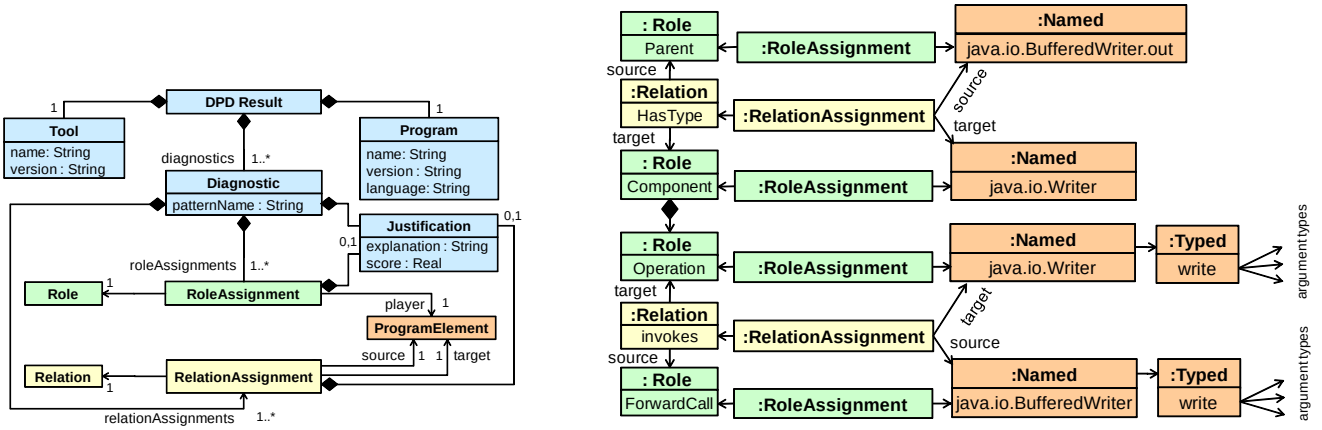
Each of these elements can be nested inside each another element. That is general enough to accomodate even exotic languages. Although blocks and named elements look similar (both contain just a name), there is a significant distinction. The names of named elements stem from the analysed program whereas those of blocks belong to a fixed vocabulary (see [10]). Each block is named after its role in the program element in which it occurs (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. III-E). Fig. 7b illustrates the object representation of the



(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 5: Design pattern schemata: metamodel and sample schema



(a) Metamodel of design pattern detection results

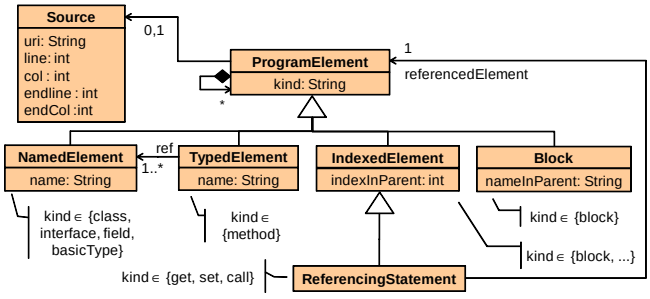
(b) A partial instance of the result metamodel. It shows role and relation assignments for a decorator candidate that conforms to the schema from Figure 5b)

Figure 6: Design pattern diagnostics: metamodel (a) and sample role and relation assignments (b)

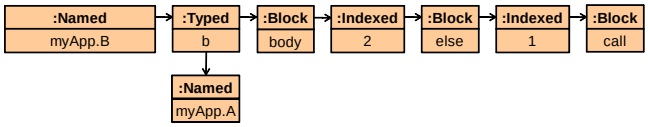
invocation a.f(d,c) from the code example given is Sec.III-E. In the textual notation used there it has been denoted as 'myApp.B.b(myApp.A).body.2.else.1.call'.

C. Result Metamodel

Figure 6a 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. 5a and Fig. 7a. Figure 6b shows a few role and relation assignments for an instance of the Decorator pattern consisting of the program elements java.io.Writer, java.io.BufferedWriter, java.io.Writer.write(), java.io.BufferedWriter.write(), etc.



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



(b) Representation of the invocation a.f(d,c) from the code example in Sec.III-E.

Figure 7: Metamodel of program element identifiers (a) and sample instance (b)

V. DPDX Implementation

The DPDX meta-models are a common framework of reference for developing and implementing textual output of DPD tools and parsing / interpretation of this output by users of DPD results (developers, the fusion tool, benchmarks and visualization tools). For long-term maintainability, the implementations of the meta-models 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.

```
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">
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 8: Implementation of schema metamodel

A. Implementation details

The implementation of DPDX consists of realization of the three meta-models introduced in Section IV. Implementation of the Schema meta-model (see Subsection IV-A) allows the tools to report the schema of the patterns they search for, the Program Element meta-model (see Subsection IV-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 IV-C) implementation describes the detected pattern instances themselves.

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, and the element that wants to refer to this element has an attribute to refer it. The referencing attribute is named as the meta-model association that it implements
- an association with target cardinality greater than 1 is represented by a contained group element with individual referencing elements. For instance, the TypedElement’ nodes contain a ‘ref’ element which collects further ‘ref’ elements to refer to the types of the parameters of the ‘TypedElement’t (Figure 9, line 17-21).

```
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 namedElement="PE13"/>
07    <ref namedElement="PE14"/>
08    <ref namedElement="PE14"/>
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 namedElement="PE13"/>
19    <ref namedElement="PE14"/>
20    <ref namedElement="PE14"/>
21   </ref>
22   <IndexedElement id="PE6" indexInParent="1" kind="block">
23    <Block nameInParent="synchronized">
24     <IndexedElement id="PE7" indexInParent="3" kind="conditional">
25      <Block nameInParent="then">
26       <ReferencingStatement id="PE8" indexInParent="2"
27            kind="call" referencedElement="PE2" source="P6"/>
28      </Block>
29     </IndexedElement>
30    </Block>
31   </IndexedElement>
32  </TypedElement>
33  <TypedElement id="PE9" name="flush" kind="method" source="P7">
34   <IndexedElement id="PE10" indexInParent="1" kind="block">
35    <Block nameInParent="synchronized">
36     <ReferencingStatement id="PE11" indexInParent="2"
37          kind="call" referencedElement="PE3" source="P8"/>
38    </Block>
39   </IndexedElement>
40  </TypedElement>
41  <NamedElement id="PE12" name="out" kind="field" source="P9"/>
42 </NamedElement>
43
44 <NamedElement id="PE13" kind="basicType" name="char[]"/>
45 <NamedElement id="PE14" kind="basicType" name="int"/>
46
47 <Sources>
48  <Source id="P1" URI="/java/io/Writer.java" line="33" col="1"
49       endLine="308" endCol="1"/>
50  <Source id="P2" URI="/java/io/Writer.java" line="128" col="5"
51       endLine="128" endCol="81"/>
52  <Source id="P3" URI="/java/io/Writer.java" line="293" col="5"
53       endLine="293" endCol="52"/>
54  <Source id="P4" URI="/java/io/BufferedWriter.java" line="47"
55       col="1" endLine="253" endCol="1"/>
56  <Source id="P5" URI="/java/io/BufferedWriter.java" line="154"
57       col="5" endLine="183" endCol="5"/>
58  <Source id="P6" URI="/java/io/BufferedWriter.java" line="169"
59       col="3" endLine="169" endCol="28"/>
60  <Source id="P7" URI="/java/io/BufferedWriter.java" line="232"
61       col="5" endLine="237" endCol="5"/>
62  <Source id="P8" URI="/java/io/BufferedWriter.java" line="235"
63       col="6" endLine="235" endCol="17"/>
64  <Source id="P9" URI="/java/io/BufferedWriter.java" line="49"
65       col="5" endLine="49" endCol="23"/>
66 </Sources>
67</ProgramElements>
```

Figure 9: Implementation of program metamodel

To avoid any ambiguities we require that intentionally missing values be made explicit by special reserved values (enclosed

```
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"/>
07      <RoleAssignment id="RA2" role="R2" player="PE2"/>
08      <RoleAssignment id="RA3" role="R2" player="PE3"/>
09      <RoleAssignment id="RA4" role="R3" player="PE4"/>
10      <RoleAssignment id="RA5" role="R4" player="PE5"/>
11      <RoleAssignment id="RA6" role="R4" player="PE9"/>
12      <RoleAssignment id="RA7" role="R6" player="PE12"/>
13      <RoleAssignment id="RA8" role="R5" player="PE8"/>
14      <RoleAssignment id="RA9" role="R5" player="PE11"/>
15      <RoleAssignment id="RA10" role="R7" player="%MISSING%"/>
16    </RoleAssignments>
17
18    <RelationAssignments>
19      <RelationAssignment relation="RE1" source="PE4" target="PE1"/>
20      <RelationAssignment relation="RE3" source="PE8" target="PE2"/>
21      <RelationAssignment relation="RE3" source="PE11" target="PE3"/>
22      <RelationAssignment relation="RE4" source="PE12" target="PE1"/>
23    </RelationAssignments>
24
25    <Justifications>
26      <Justification for="RA5" score="80%" explanation=""/>
27      <Justification for="RA8" score="80%" explanation="conditional
28          forward"/>
29      <Justification for="RA10" score="" explanation="missing subclass"/>
30      <Justification for="PI1" score="95%" explanation=""/>
31    </Justifications>
32  </Diagnostic>
33</DPDResult>
```

Figure 10: Implementation of result metamodel

in % signs) instead of simply providing empty attributes. In particular,

- the‘variantOf’ attribute of a ‘PatternSchema’ element must have the value %NONE% if the respective schema is not a variant of another schema (Fig. 8, line 1);
- the ‘player’ attribute of a ‘Role’ element must have the value, %MISSING% if no program element that plays this role could be found (see Fig. 10, line 15).

We chose to slightly deviate from these general principles when we felt it would make the implementation clearer and easier to understand:

- To avoid cluttering the format with redundant imformation, the ‘kind’ attribute of program elements is not set if it is implied by the program element type (for example ‘Block’ node in Figure 9, line 23).
- The result meta model has defined an aggregation between ‘Justification’-‘RoleAssignment’ and ‘Justification’-‘RelationAssignment’ elements. However, in the implementation we represent ‘Justification’ elements separately and not as the children of the assignment nodes. ‘Justification’ tags can refer to a ‘RoleAssignment’ or to a ‘RelationAssignment’ with the ‘for’ attribute (Figure 10, line 26).
- In the paper, we do not present justifications for every assignment node because space limitations.

B. Integration and visualization

The implementation of the three meta-models could have been placed into three different XML files. However, we chose to integrate them into one, since it eases processing and visualization. This means, that the ‘ProgramElements’ and the ‘PatternSchema’ tags are inserted into DPDX files as the children of the ‘DPDResult’ tag.

For supporting 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 [24]. Futhermore, the exact rules of the XML format are defined by an XSD schema file which is also available online [24].

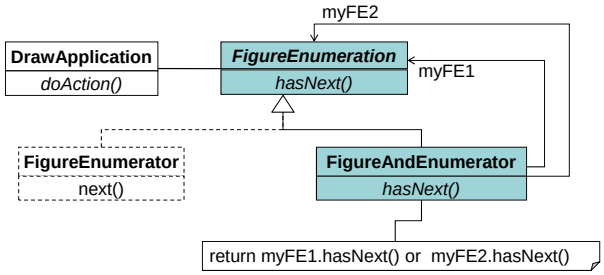


Figure 11: Improper Decorator (JHotDraw 6.0)

VI. Evaluation

In this section we evaluate the DPDX format and its conceptual unerpinnings, in particular the unique identification of program elements and candidates. For brevity, we focus on some of our solutions to the requirements from Section II-C, showing how they ease data fusion of the outputs of different DPD tools and improve its accuracy. For more details see [10].

A. Identification of Candidates

Figure 11 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 candidates (Sec. III-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 candidate. 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 Comparability

Figure 12 also illustrates an instance of the ‘Chain of Responsibility’ (CoR) pattern. The class

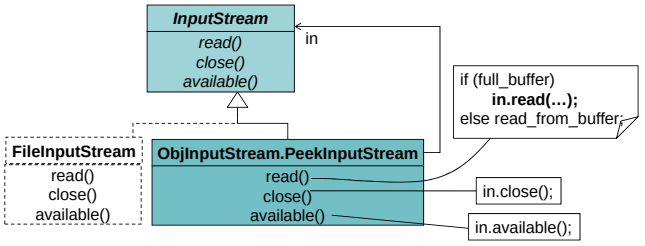


Figure 12: Decorator and Chain of Responsibility (Java IO)

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. The conditional forwarding in read() which characterizes the 'Request' role, distinguishes it from the Decorator pattern's '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 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 contradict but confirms the diagnosis of PINOT, since the CoR and 'Decorator' motifs are the same except for conditional versus unconditional forwarding.

VII. 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.

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. [Online]. Available: 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.

[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] 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/research/dpd/dpdx/

[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] Maisa homepage. [Online]. Available: http://www.cs.helsinki.fi/group/maisa/

[16] 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.

[17] 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.

[18] 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.

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

[20] J. Smith and D. Stotts, "Spqr: Flexible automated design pattern extraction from source code," in *ASE'03*. IEEE, 2003.

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

[22] 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.

[23] 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.

[24] DPDX homepage. [Online]. Available: https://sewiki.iai.uni-bonn.de/dpdx/