

DP-CORE: A Design Pattern Detection Tool for Code Reuse

Themistoklis Diamantopoulos, Antonis Noutsos and Andreas Symeonidis

Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece
thdiaman@issel.ee.auth.gr, anoutsos@auth.gr, asymeon@eng.auth.gr

Keywords: Design Pattern Detection, Static Code Analysis, Reverse Engineering, Code Reuse

Abstract: In order to maintain, extend or reuse software projects one has to primarily understand what a system does and how well it does it. And, while in some cases information on system functionality exists, information covering the non-functional aspects is usually unavailable. Thus, one has to infer such knowledge by extracting design patterns directly from the source code. Several tools have been developed to identify design patterns, however most of them are limited to compilable and in most cases executable code, they rely on complex representations, and do not offer the developer any control over the detected patterns. In this paper we present DP-CORE, a design pattern detection tool that defines a highly descriptive representation to detect known and define custom patterns. DP-CORE is flexible, identifying exact and approximate pattern versions even in non-compilable code. Our analysis indicates that DP-CORE provides an efficient alternative to existing design pattern detection tools.

1 INTRODUCTION

Developers need to understand existing projects in order to maintain, extend, or reuse them. However, understanding usually comes down to understanding the source code of a project, which is inherently difficult, especially when the original software architecture and design information is unavailable. And, although several tools extract information from source code, in cases where software projects lack proper documentation, the process of understanding the intent and design of the source code requires a lot of effort.

The design decisions taken during software development concern the non-functional aspects of the system, and are usually documented in the form of *design patterns*. Design patterns provide reusable solutions in the form of templates that developers can use to confront commonly occurring problems (Gamma et al., 1998). Inferring such non-functional knowledge from source code typically requires extracting these patterns. Lately, the problem of recovering design patterns from source code has attracted the attention of several researchers and has led to the development of several tools to detect patterns, known as *Design Pattern Detection (DPD)* tools.

Most of the tools are effective for detecting certain types of design patterns. However, they fall short in several important aspects. At first, they require the source code to be compilable, or in most cases ex-

ecutable. As a result, developers cannot exploit the source code of other systems without first resolving their dependencies and executing them correctly. Secondly, pattern representations in most tools are not intuitive, thus resulting in black box systems that do not allow the developer any control over the detected patterns. These tools also do not offer the ability to define custom patterns. Finally, several DPD tools are not up-to-date, supporting only obsolete versions of programming languages.

In this paper, we present DP-CORE, a Design Pattern detection tool for COde REUse, which is designed in order to overcome the aforementioned issues. DP-CORE uses a highly descriptive and complete representation for source code elements based on UML. Hence, the tool supports both the detection of several well known patterns and the definition of custom patterns by the developer. DP-CORE is also quite flexible, matching not only strictly defined versions of patterns but also similar versions of patterns using wildcards. Furthermore, the detection of patterns in non-executable and even non-compilable source code is fully supported, while DP-CORE also uses the latest compiler technology to support detecting patterns in current Java projects.

The rest of this paper is organized as follows. Section 2 provides background knowledge and reviews current approaches for the detection of design patterns from source code. In Section 3, we present our DPD

tool, focusing on the defined representation and the methodology used to detect patterns, while its user interface is presented in Section 4. Section 5 illustrates the usage of our tool using a case study and presents our evaluation against the DPD tool PINOT (Shi and Olsson, 2006). Finally, Section 6 summarizes work done and provides useful insight for future research.

2 BACKGROUND AND RELATED WORK

Software design patterns were first proposed in 1987 (Beck and Cunningham, 1987), shortly after the *Object Oriented Programming (OOP)* paradigm began gaining momentum. One of the first attempts to formalize the use of design patterns was a book published in 1998 by Gamma, Vlissides, Johnson, and Helm (1998), who became known as the *Gang of Four (GoF)*. The GoF defined 23 patterns, which are categorized in *creational*, *structural*, and *behavioral* patterns. Creational patterns abstract the process of creating objects, while structural patterns combine objects to form larger structures and behavioral patterns describe the communication between objects.

Current literature in DPD methods is quite broad. According to Dong et al. (2009), the defining properties of a DPD tool are (a) the type of input, which can be source code, UML diagrams or any other representation, (b) the intermediate representation, which includes all structural formats used by the DPD tool, e.g. *Abstract Syntax Tree (AST)*, *Call Dependency Graph (CDG)*, (c) the type of analysis, performed either only on source code or also on the dynamic execution trace, and (d) the type of recognition, which can be either exact or approximate. Although all of the above properties are important, the main classifying feature of DPD techniques is the type of analysis. Following a classification similar to that of current literature (Rasool and Streitferdt, 2011; Dong et al., 2007b), we distinguish among *structural analysis*, *behavioral analysis*, and *semantic analysis* techniques.

Structural analysis approaches detect patterns using information extracted from inter-class relationships (e.g. class inheritance, associations, etc.), thus they are particularly efficient for identifying creational and structural patterns. By contrast, behavioral analysis approaches employ dynamic program analysis, using runtime information to distinguish among structurally identical patterns. Semantic analysis is usually complementary to the structural and behavioral types, aspiring to reduce the false positive rates of structural and behavioral approaches by using semantic information (e.g. naming conventions).

Several DPD tools in current literature employ static code analysis techniques to identify patterns. One of the first tools in this category is the DPD tool by Tsantalis et al. (2006), which extracts static information about the software project including inheritance, method signatures, etc., and represents the project as a graph. Relationships among objects are represented using matrices, while the tool uses similarity algorithms to detect patterns. A slightly different approach is followed by Lucia et al. (2009) for the DPRE tool, which is implemented as an Eclipse plugin. The tool constructs UML diagrams and uses visual language parsing techniques to identify patterns.






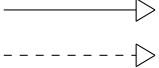
PINOT, designed by Shi and Olsson (2006), is another popular tool that combines both structural and behavioral analysis. It extracts information from the AST of the source code, and detects patterns using structural and behavioral (data flow) template matching. Several tools also use machine learning methods. Arcelli and Christina (2007) developed MARPLE, an Eclipse plugin that uses neural networks to classify source code representations to behavioral patterns. The authors also extended their work, introducing JADEPT (Arcelli et al., 2008), a tool that represents patterns as combinations of rules.

FUJABA (Nickel et al., 2000) is another Eclipse plugin, which, among other functions, supports detecting design patterns. The tool defines patterns using UML class diagrams and expresses their behavioral aspects using story-diagrams, i.e. a combination of activity and interaction diagrams. Metamodel-based approaches are also popular. Guéhéneuc and Antoniol (2008) introduced DeMIMA, which extracts classes, methods, etc. from source code to instantiate a metamodel that is used to specify objects and their relationships. The tool employs constraint programming techniques to identify patterns. A similar approach is followed by Ptidej, designed by Kaczor et al. (2006), which uses a constraints solver to detect sets of objects that are similar to design patterns.

Another notable tool is D-CUBED, by Stencil and Wegrzynowicz (2008), which formulates design patterns in first order logic and inserts source code into a database where patterns can be identified via queries. Finally, DP-Miner by Dong et al. (2007a) is another quite interesting tool that also uses semantics to distinguish among certain patterns that may have similar structural and behavioral aspects. The tool represents source code entities and relationships in XMI and analyzes class and method names to identify patterns.

The tools analyzed in the previous paragraphs are quite effective for detecting different types of patterns. However, their applicability is limited to compilable and executable projects. In specific, tools us-

Table 1: DP-CORE Connections

| Connection Type | Description | UML Relation | UML Symbol |
|-----------------|--|-----------------------------|---|
| A calls B | A method of class A calls a method of class B | Dependency |  |
| A creates B | Class A creates an object of type class B | Composition |  |
| A uses B | A method of class A returns an object of type B | Dependency/ Multiplicity |  |
| A has B | Class A has one or more objects of type B | Aggregation |  |
| A references B | A method of class A has as parameter an object of type B | Association |  |
| A inherits B | Class A inherits or implements class B or class A realizes interface B | Inheritance/ Realization |  |

ing dynamic analysis techniques, such as D-CUBED [17] or DP-Miner [5], require the code to be executable. Moreover, static analysis tools, such as PINOT [16] or DPD [18], also operate on .class files, therefore require that the developer first resolves any dependencies to compile the examined project. Furthermore, tools that rely on compilers, such as PINOT [16], cannot always compile all projects not only because of dependencies but only because of obsolete compiler versions. Finally, several tools such as DeMIMA [9] or Ptdiej [10], use complex representations that do not allow the developer any control over the detected patterns. As a result, these tools do not offer the ability to define custom patterns or the ability to detect incomplete or similar patterns. To the best of our knowledge, no tool presented in this Section and no other tool offers all of the aforementioned features. In the following Section, we present an up-to-date customizable DPD tool that effectively abstracts the methodology of pattern detection even in uncompileable or incomplete source code.

3 DP-CORE: A DESIGN PATTERN DETECTION TOOL

In this Section, we describe our DPD tool, DP-CORE, in detail. Subsection 3.1 describes the representation used for source code objects and relationships, while subsection 3.2 provides details about the representation of design patterns by our system. The extraction of entities and relationships from source code is presented in subsection 3.3 and the algorithm for detecting patterns is described in subsection 3.4. Subsection 3.5 illustrates how the results are grouped in order to form combined patterns.

3.1 Representing Objects and Relationships

As already mentioned, an effective structural representation for source code and design patterns should be complete as well as intuitive. Given that most developers are more or less familiar with UML, the representation of DP-CORE is related to UML entities and relationships. Given intuition from (Birkner, 2007), we define two concepts for our representation: the abstraction of each class and the connection between two classes. The abstraction types that we defined for each class, including their description, are shown in Table 2.

Table 2: DP-CORE Abstraction Types

| Abstraction Type | Description |
|------------------|-----------------------------------|
| Normal | a non-abstracted class |
| Abstract | a Java abstract class |
| Interface | a Java interface |
| Abstracted | an abstract class or an interface |
| Any | any of the above class types |

The type `Normal` refers to a simple non-abstracted class, while types `Abstract` and `Interface` correspond to the known Java abstract classes and interfaces. Additionally, we define the type `Abstracted` as either one of types `Abstract` and `Interface`, while the type `Any` denotes any of the above abstraction types and functions as a wildcard. Apart from their abstraction types, the classes of the examined source code connect to each other with directional relationships. We define 6 types of connections that are summarized in Table 1, including their description, the corresponding UML relation and the corresponding UML symbol for each connection.

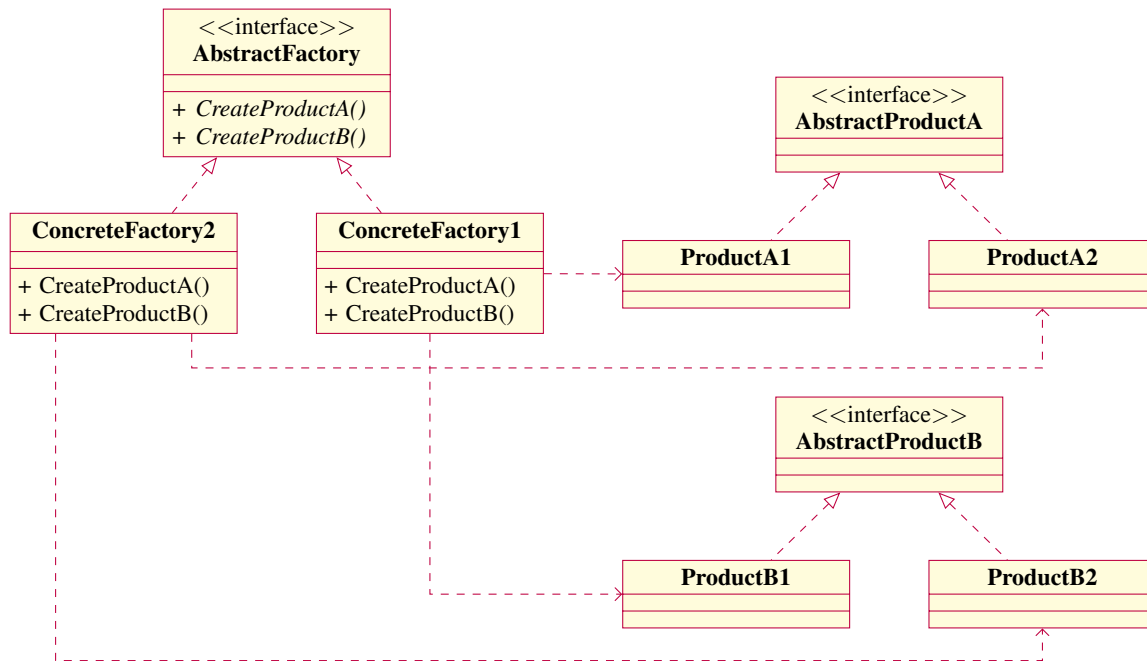


Figure 1: Abstract Factory Pattern

The connections cover all possible relations that can exist in a source code project. UML dependencies and associations are handled by connections calls/uses and references respectively, while compositions and aggregations correspond to the creates and has connections. Inheritance and realization relations are handled by the inherits connection. Finally, we define a relates connection that covers all possible connection types, and thus is used as a wildcard.

3.2 Representing Design Patterns

Upon having presented how source code entities and relationships are represented in our system, we illustrate how well known (or custom) design patterns can be represented by our system. For each pattern, one must define the abstraction of its member classes and the connections among them. In this subsection, we illustrate how the Abstract Factory pattern is defined. According to the GoF (Gamma et al., 1998), the purpose of the Abstract Factory pattern is to “provide an interface for creating families of related or dependent objects without specifying their concrete classes”.

Figure 1 depicts the interface `AbstractFactory` which defines the methods that are implemented by `AbstractProductA` and `AbstractProductB`, while `ConcreteFactory1` and `ConcreteFactory2` define the methods of the `Product` objects. Note that this is one possible illustration, since it is possible to have different number of `ConcreteFactory`,

`AbstractProduct` and/or `Product` classes. By definition, an Abstract Factory pattern has to include instances of `AbstractFactory`, `AbstractProduct`, `ConcreteFactory`, and `Product`. Using the representation of subsection 3.1, we define the 4 members of the pattern and their connections in Figure 2.

```

Abstract Factory
A Normal ConcreteFactory
B Abstracted AbstractFactory
C Normal Product
D Abstracted AbstractProduct
End_Members
A inherits B
C inherits D
A creates C
A uses D
End_Connections

```

Figure 2: Abstract Factory Pattern File

For each pattern member, A, B, C, and D, we can see its abstraction type and its ability. The ability of a member is a description field that can be set by the developer so that the detected pattern instances are more comprehensive. The pattern has 4 connections, 2 inherits, one from A (`ConcreteFactory`) to B (`AbstractFactory`) and one from C (`Product`) to D (`AbstractProduct`), 1 creates from A (`ConcreteFactory`) to C (`Product`), and 1 uses from A (`ConcreteFactory`) to D (`AbstractProduct`).

```

public class Car extends Vehicle{----- Car inherits Vehicle

    private Model model;----- Car has Model
    private Fuel fuel;----- Car has Fuel

    public Car (Model model,----- Car references Model
                float fuelCapacity){
        this.model = model;
        fuel = new Fuel(fuelCapacity);----- Car creates Fuel
    }

    public Model getModel(){----- Car uses Model
        return model;
    }

    public void addFuel(float fuelQuantity){
        fuel.add(fuelQuantity);----- Car calls Fuel
    }
}

```

Figure 3: Example of Extracting Connections for a Car Class

3.3 Extracting Objects and Relationships from Source Code

The main building blocks of our methodology are objects and relationships between them. DP-CORE uses the Java Compiler Tree API (Oracle, 2015) to extract these elements. Upon extracting the AST for each file, all class objects are extracted from the code, including their abstraction type. After that, DP-CORE extracts the connections. A uses connection from A to B is extracted if the return type of any method of A is an object of type B. Connections of type `inherits` are extracted if class A extends or implements class B. For the `has` connection from A to B, all variables of A are checked to find if any of them is of type B. For the `calls` connection, all method invocations of class A are checked whether the method invoked is a method of class B. Connections of type `creates` are extracted if class A constructs a new object of type B. Finally, a `references` connection from A to B is declared if any method of A has as parameter an object of type B.

Figure 3 depicts an example of extracting the connections of a `Car` class which interacts with three classes. It inherits the `Vehicle` class and has two objects of type `Model` and `Fuel`. Additionally, `Car` references the `Model` in its constructor, where the `Fuel` object is also created. Finally, the getter function of `Car` also implies that it uses the `Model` class, while `Car` also calls a method of `Fuel` to add fuel to its tank. Hence, we can define 7 connections among the classes in this example, which are shown in the annotations on the right of Figure 3.

3.4 Design Pattern Detection Algorithm

Upon having extracted the objects and relationships of the examined software project, the next step is to detect patterns in the extracted structure. Although this problem could be solved by iterating over all possible permutations of classes, this brute force method would be computationally inefficient. For instance, given a project with 50 classes and a pattern with 4 members, this method would check more than 5 million permutations. Thus, we designed an algorithm that prunes permutations as it recursively finds pattern candidates.

Input: Objects, Members
Output: Candidates

```

Detect(Objects, Members, Candidate, d):
    if d < Members.length():
        Member = Members[d]
        for Object in Objects:
            if abstraction(Object, Member)
            and connections(Object, Member):
                Detect(Objects \ {Object},
                    Candidate ∪ {Object},
                    Candidate, d + 1)
    else:
        Add Candidate to Candidates

```

Figure 4: Design Pattern Detection Algorithm

Our algorithm is shown in Figure 4. It receives as input the objects extracted from the examined project (and their connections), as well as the pattern to be

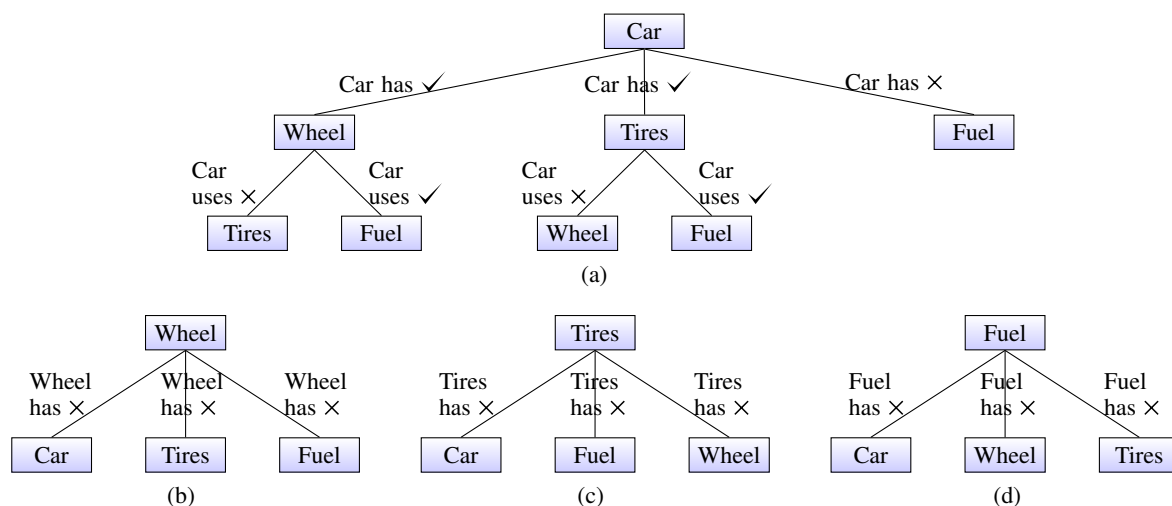


Figure 5: Combinations of objects and their relationships, where the object that matches the first pattern member is (a) Car, (b) Wheel, (c) Tires, and (d) Fuel

detected in the format defined in the previous subsections. It iterates over the Objects and checks whether the current Object can be matched to the current pattern member. This is performed by recursively calling the Detect function and providing the index to current pattern member as the depth parameter d . At first, the algorithm is initialized with depth equal to 0 (and Candidate is the empty set). Iterating over the first Object, it is checked whether its abstraction and its connections are the same with pattern member 0. If the Object matches this pattern member, then the Detect function is called again given as parameters the Objects without the already matched Object and the updated Candidate so that it includes the Object, while the depth parameter is also incremented. If at any time the current Object does not match the current pattern member, then the recursion stops. When all pattern members are matched, then the Candidate is added to the detected pattern Candidates.

We illustrate the execution of the algorithm using an example with 4 classes Car, Tires, Wheel, and Fuel, connected with the relationships Car has Tires, Car has Wheel, and Car uses Fuel. The pattern to be detected has 3 classes A, B, and C, connected with the relationships A has B and A uses C. To simplify our example let us assume that the objects of the examined code and the members of the pattern do not have any abstraction. Some possible combinations for this example are shown in Figure 5.

Note that given 4 objects, their possible permutations per 3 pattern members are 24. Using our algorithm, however, we examine much fewer. At first, all permutations involving Wheel, Tires, or Fuel in the position of member A (shown in Figures 5(b), 5(c), and 5(d) respectively) are discarded in the first level

of the tree, since they do not connect to any other element with the required connections. If Car is matched with pattern member A, then the other objects are checked for has and uses connections. As shown in Figure 5(a), the path of the connection Car has Fuel is pruned. The other two paths, including the connections Car has Wheel and Car has Tires, are further examined, to finally provide the two instances of the pattern $\{A = \text{Car}, B = \text{Wheel}, C = \text{Fuel}\}$ and $\{A = \text{Car}, B = \text{Tires}, C = \text{Fuel}\}$ respectively.

3.5 Grouping Design Pattern Instances

In a typical design pattern detection scenario, the examined code may have several instances of a design pattern. Since some of these instances may be part of the same design decision, we need a way to merge these candidate patterns to form a unified design pattern. For instance, consider the 4 Abstract Factory candidate instances of Table 3.

DP-CORE merges these instances in two steps. At first, the tool iterates over all candidate patterns and checks whether any of them have identical members except for one. In this case, two super-patterns would be formed, one for the ReptileFactory, i.e. $\{A = \text{ReptileFactory}, B = \text{SpeciesFactory}, C = \text{Snake|Tyrannosaurus}, D = \text{Animal}\}$, and one for the MammalFactory, i.e. $\{A = \text{MammalFactory}, B = \text{SpeciesFactory}, C = \text{Cat|Dog}, D = \text{Animal}\}$. After that, DP-CORE iterates over the super-patterns to identify again the ones having identical members except for one. Thus, the final hyper-pattern for this example is visualized in Figure 6.

Table 3: Abstract Factory Candidate Instances

| Member | Ability | Candidate 1 | Candidate 2 | Candidate 3 | Candidate 4 |
|--------|-----------------|----------------|----------------|----------------|----------------|
| A | ConcreteFactory | ReptileFactory | ReptileFactory | MammalFactory | MammalFactory |
| B | AbstractFactory | SpeciesFactory | SpeciesFactory | SpeciesFactory | SpeciesFactory |
| C | Product | Snake | Tyrannosaurus | Cat | Dog |
| D | AbstractProduct | Animal | Animal | Animal | Animal |

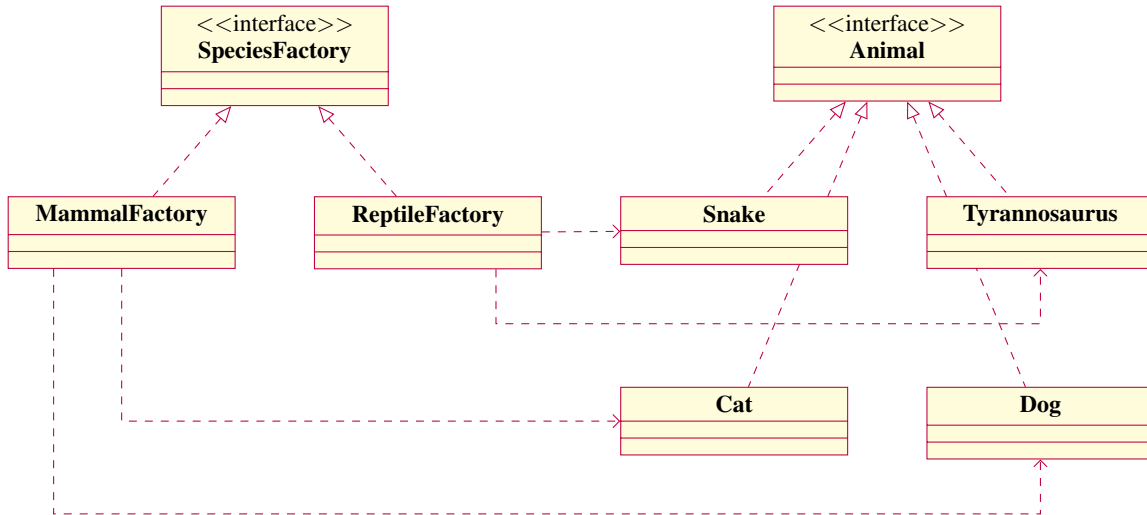


Figure 6: Hyper-Pattern for the Pattern Candidate Instances of Table 3

4 INTERFACES OF DP-CORE

DP-CORE is implemented in Java, and offers two user interfaces: a *Graphical User Interface (GUI)* and a *Command Line Interface (CLI)*. The source code and the releases are available in the repository:

<https://github.com/AuthEceSoftEng/DP-CORE>

DP-CORE receives as input project folders including .java files for pattern detection, as well as .pattern files that define the pattern to be detected. Its output is a txt file that includes the results of the detection. An example output of DP-CORE is shown in Figure 7.

```

Amount of Candidates found: 2

Candidate of Pattern Command:
A(ConcreteCommand): LightOffCommand
B(Command): Command
C(Receiver): Light
D(Invoker): RemoteControl

Candidate of Pattern Command:
A(ConcreteCommand): LightOnCommand
B(Command): Command
C(Receiver): Light
D(Invoker): RemoteControl
    
```

Figure 7: Example Output of DP-CORE

In this case, the examined project has two candidates of the Command pattern. Since they refer to the same pattern, the user could also merge them by enabling the grouping mechanism of subsection 3.5.

The GUI of DP-CORE is shown in Figure 8. The main screen of the tool, shown in Figure 8(a), includes 3 textfields, which are used to enter the folder where patterns are stored, the folder of the examined project, and the folder where the results are saved. Upon entering this information, the user can select the pattern to be detected using the drop-down menu and push the “Detect Pattern” button. The “Grouping” checkbox controls whether the pattern grouping mechanism of subsection 3.5 is activated. The user can either select a predefined pattern or define a new pattern by pushing the “Create Custom Pattern” button. Upon pressing this button, the user is presented with the screen shown in Figure 8(b), where the user names the newly defined pattern and defines the number of its members and connections. After that, the members and the connections of the pattern are defined in two subsequent screens shown in Figures 8(c) and 8(d).

Finally, the tool can be used from the command line to allow the execution of batch jobs with multiple projects and/or patterns. Similarly to the GUI, the CLI of DP-CORE receives as input the project folder as well as a .pattern file containing the pattern to be detected. The output is printed in the console.

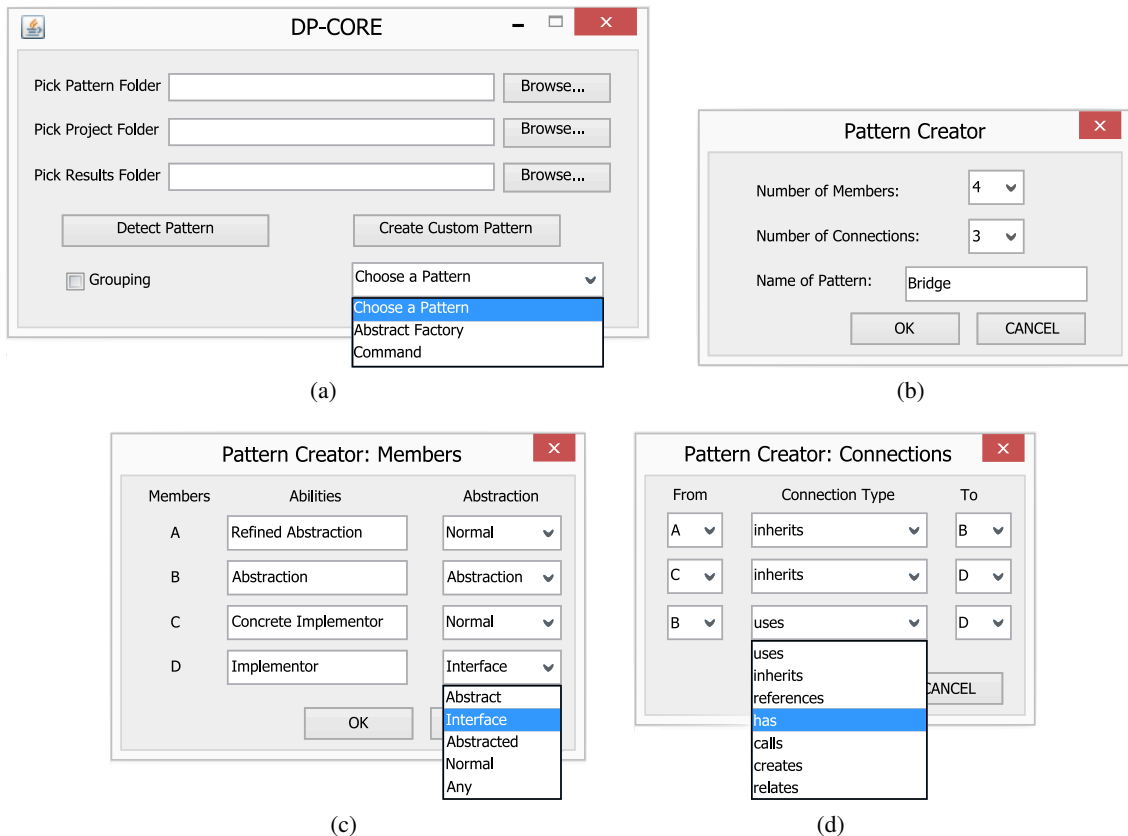


Figure 8: Screenshots of (a) the Main Screen of DP-CORE, (b) the Pattern Creator, including (c) the Members and (d) the Connections Screens of DP-CORE

5 EVALUATION

We assess the effectiveness of DP-CORE using two evaluation experiments. The first experiment involves an example project including known instances of patterns, while the second experiment involves a comparison to PINOT (Shi and Olsson, 2006) for detecting patterns in the source code of known Java libraries.

5.1 Example Design Patterns Project

For the first experiment we used several design pattern examples merged together in a common project. Although this project is not a typical Java application, it provides an interesting case study for our tool. We created pattern files for 6 GoF patterns of all types: the creational patterns Abstract Factory and Builder, the structural pattern Bridge, and the behavioral patterns Command, Observer and Visitor. The example project and the patterns are provided with our tool as examples. The results are shown in Table 4.

DP-CORE successfully identified all the pattern instances in the project. It is notable, though, that the tool detected false positive instances, since 27.27% of

the detected instances are not design patterns. These false positives, however, are due to the non-strict definition of the patterns. For instance, the 4 falsely identified instances of the Command pattern are instances of the Builder pattern, which is expected since the definitions of these two patterns are similar. This is also the case for the Observer pattern which is similar to the Visitor pattern. In any case, DP-CORE is quite effective for identifying patterns, as it detects all instances, while false positives can be minimized by providing more precise definitions of patterns.

Additionally, upon tweaking the code of this project, we conclude that DP-CORE can detect patterns even in non-compilable code. In specific, our tool is not affected by common syntax errors, such as missing brackets, missing semicolons etc. Furthermore, most semantic errors, such as missing imports, missing variable declarations, etc., are safely ignored without influencing the detection of patterns. The errors that affect our tool are purely lexical, e.g. writing `fo` instead of `for` or having a space in a variable name. As a result, DP-CORE supports code reuse scenarios, given that reusable code is usually lexically correct, while it may have omissions in syntax or semantics.

Table 4: Pattern Detection Results for Example Project

| Design Pattern | #Instances | #Correctly Detected Instances | #Incorrectly Detected Instances | %Correctly Detected Instances | %Incorrectly Detected Instances |
|------------------|------------|-------------------------------|---------------------------------|-------------------------------|---------------------------------|
| Abstract Factory | 4 | 4 | 0 | 100% | 0% |
| Command | 2 | 2 | 4 | 100% | 66.67% |
| Bridge | 4 | 4 | 0 | 100% | 0% |
| Builder | 2 | 2 | 0 | 100% | 0% |
| Visitor | 2 | 2 | 0 | 100% | 0% |
| Observer | 2 | 2 | 2 | 100% | 50% |
| Total | 16 | 16 | 6 | 100% | 27.27% |

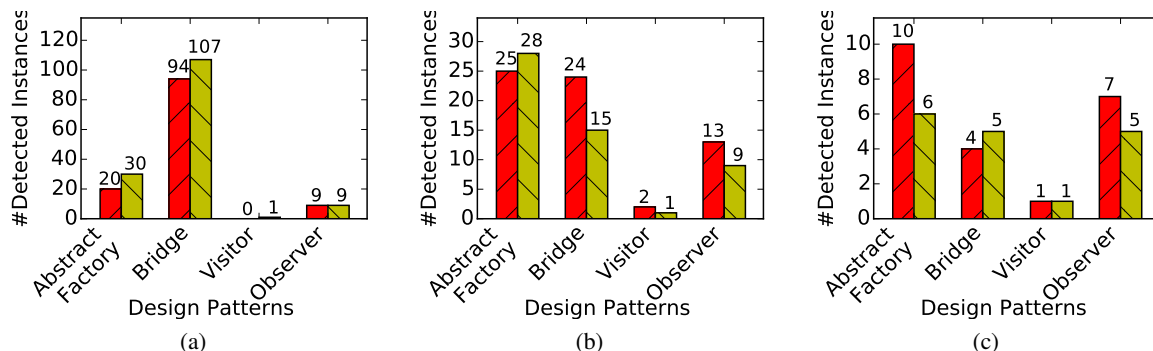


Figure 9: Diagrams of Detected Design Pattern Instances by DP-CORE (▨) and PINOT (▨) for (a) JHotDraw, (b) Java AWT, and (c) Apache Ant

5.2 Java Libraries with Design Patterns

For the second experiment we evaluated DP-CORE against PINOT (Shi and Olsson, 2006) in a dataset of 3 libraries: JHotDraw 6.0b1, Java AWT 1.3, and Apache Ant 1.6.2. PINOT was selected as it resembles DP-CORE in the way that it extracts source code objects and detects patterns. Since, however, the representation used by PINOT differs from that of DP-CORE, the comparison is quantitative, thus the results can only provide a proof-of concept for our DPD. As already noted, the scope of DP-CORE lies in providing an intuitive way of detecting custom patterns and software architectures, instead of strictly defining the known design patterns. For compatibility reasons, the comparison between the two tools includes the patterns Abstract Factory, Bridge, Visitor, and Observer.

The results for the two tools are shown in Figure 9. For JHotDraw, in Figure 9(a), there are small deviations between the detected patterns of the two tools, which are mostly due to the pattern representation of each tool. Upon manually examining the detected patterns, we conclude that both tools successfully identify the design patterns of this library. The results for the other two libraries, Java AWT in Figure 9(b) and ApacheAnt in Figure 9(c), are also similar. Note, however, that patterns are not always defined consistently in all projects. As a result, for Java AWT, we had to modify the representation of Abstract Fac-

tory, changing the connection A uses D to A uses C in Figure 2, thus allowing the ConcreteFactory to use Product instead of AbstractProduct. Apache Ant also required a modification to the Bridge pattern, where the A calls D connection was added (see Figure 8) to restrict the instances to the ones where the RefinedAbstraction calls the Implementor.

The main deviations are observed for the Abstract Factory and Bridge patterns, which is expected since the representations of these patterns are similar. Summarizing our analysis, the results of the quantitative comparison between DP-CORE and PINOT indicate that both tools can successfully detect patterns from source code. However, it is important to note that all tools are bound to the representations used to identify patterns, therefore comparing them is not trivial. DP-CORE further allows defining custom patterns and effectively recovers structural design semantics even for non-standard architectures. Finally, concerning execution time, DP-CORE is quite efficient; pattern detection in any project required less than 5 seconds.

6 CONCLUSION

Recovering non-functional design information from source code is a difficult task. Since this information is usually provided in the form of design patterns,

several DPD tools have been developed to extract it. However, most tools are limited to identifying certain known patterns in executable projects. In this work, we presented DP-CORE, a DPD tool that can recover patterns even from non-compilable source code. DP-CORE uses a flexible and intuitive representation, allowing developers to define their own patterns and even use wildcards to express ambiguity in these definitions. Our tool is up-to-date, relying on the latest compiler technology, while it offers a GUI and a CLI so that it can be used for batch tasks. The evaluation of DP-CORE has shown that it can be effective for identifying patterns in source code that match the representation provided by the developer.

Future work on our tool lies in several directions. At first, the negation of current connections could be added to the options of the representation, thus allowing the developer to define patterns more strictly. Additionally, the structural analysis of DP-CORE can be further extended using semantics on the abilities of the patterns. Finally, forthcoming versions may include graphical pattern representations by integrating with known graphical editors. In any case, pattern recovery from source code should be connected to the requirements of the developer, thus we believe DP-CORE is an efficient alternative to existing tools.

ACKNOWLEDGEMENTS

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission.

REFERENCES

- Arcelli, F. and Christina, L. (2007). Enhancing Software Evolution through Design Pattern Detection. In *Proceedings of the 2007 Third International IEEE Workshop on Software Evolvability*, pages 7–14, Paris, France.
- Arcelli, F. F., Perin, F., Raibulet, C., and Ravani, S. (2008). Behavioral Design Pattern Detection through Dynamic Analysis. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis*, pages 11–16, Antwerp, Belgium.
- Beck, K. and Cunningham, W. (1987). Using Pattern Languages for Object-Oriented Programs. In *Proceedings of the OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming*, Orlando, FL, USA.
- Birkner, M. (2007). Object-Oriented Design Pattern Detection Using Static and Dynamic Analysis of Java Software. Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg Sankt Augustin, Germany.
- Dong, J., Lad, D. S., and Zhao, Y. (2007a). DP-Miner: Design Pattern Discovery Using Matrix. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 371–380, Tucson, AZ, USA.
- Dong, J., Zhao, Y., and Peng, T. (2007b). Architecture and Design Pattern Discovery Techniques - A Review. In *Proceedings of the 2007 International Conference on Software Engineering Research & Practice*, volume 2 of *SERP 2007*, pages 621–627, Las Vegas, NV, USA.
- Dong, J., Zhao, Y., and Peng, T. (2009). A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855.
- Gamma, E., Vlissides, J., Johnson, R., and Helm, R. (1998). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Guéhéneuc, Y.-G. and Antoniol, G. (2008). DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Trans. Softw. Eng.*, 34(5):667–684.
- Kaczor, O., Guéhéneuc, Y.-G., and Hamel, S. (2006). Efficient Identification of Design Patterns with Bit-vector Algorithm. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, CSMR 2006, pages 175–184, Bari, Italy.
- Lucia, A. D., Deufemia, V., Gravino, C., and Risi, M. (2009). Design Pattern Recovery Through Visual Language Parsing and Source Code Analysis. *J. Syst. Softw.*, 82(7):1177–1193.
- Nickel, U., Niere, J., and Zündorf, A. (2000). The FUJABA Environment. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 742–745, Limerick, Ireland.
- Oracle (2015). Compiler Tree API. Avail. online: <http://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html>, [retrieved March, 2015].
- Rasool, G. and Streitferdt, D. (2011). A Survey on Design Pattern Recovery Techniques. *International Journal of Computer Science Issues*, 8(6):251–260.
- Shi, N. and Olsson, R. A. (2006). Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 123–134, Tokyo, Japan.
- Stencel, K. and Wegrzynowicz, P. (2008). Detection of Diverse Design Pattern Variants. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, APSEC '08, pages 25–32, Beijing, China.
- Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. (2006). Design Pattern Detection using Similarity Scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909.