A Survey of Tool Support for Working with Design Decisions in Code Supplementary Materials

Sahar Mehrpour, Thomas D. LaToza

1 Inferring Design Rules from Frequently Co-Occurring Code Fragments

Many techniques detect frequent patterns in code, particularly code clones. Following prior surveys [19, 20], we categorize techniques into those using text, data mining, graph mining, metrics, and machine learning.

Text-based Techniques

Using the source code text is one of the simplest approaches to detecting frequent patterns. In naive approaches, two pieces of code are compared, and in the case of an exact match, a pattern is detected. Later techniques improved on this by incorporating lexical approaches which first parse code into tokens. For example, CCFinder [13] parses code into statements, calculates the longest common substrings, and identifies frequent patterns based on set criteria, such as a number of shared tokens.

Data Mining Techniques

Data mining techniques may be used to process source code into data tables and extract information. Two well-known techniques are mining frequent itemsets and frequent subsequence mining. In frequent itemset mining, patterns emerge as co-occurring sets of items (itemsets). These techniques process the source code as transactions, with lower-level elements as items. Transactions may be classes or components, and items may be individual code statements (e.g., [24]). Frequent itemsets may need to be manually rechecked to assess the similarity to items. In CloneMiner [6], items are code clones found by string-matching algorithms, and patterns identify higher-level code clones across directories and packages.

Frequent subsequence mining also identifies co-occurring items, but takes the order of items into account. For example, CP-Miner [15] detects identical code. Mined frequent subsequences may be syntactically similar but not duplicated code, again requiring manual inspection by the developer.

Graph Mining Techniques

Graph mining techniques process source code into a graph representation to identify frequent and meaningful subgraphs. One common graph representation is the abstract syntax tree (AST). Using an AST, mined subgraphs preserve the code's structure, which is fundamental to frequent pattern mining techniques. For example, CloneDR [7] compares graph subtrees and calculates their similarity. A calculated similarity above a predefined threshold marks subgraphs as belonging to the same clone. FREQTALS [17] creates fewer clusters by adding constraints on each mined cluster, including a minimum number of occurrences, elements, and types of elements.

Another common representation is the dependency graph, or equivalently, the dependency matrix. One technique first computes a procedure dependence graph, representing method calls with callers and callees, extracting patterns which developers then review, edit, and approve to record as design rules [8]. The last step may be partially automated for rules specifying the validity of method arguments, validity and processing of method call outputs, and method call chains [22].

Metric-Based Techniques

In metric-based approaches, a metric is first computed for code fragments, which are then compared based on their values [19]. For example, in DECKARD [12] the structural information of the AST is approximated by counting the occurrence of important code elements in the AST and stored as 'characteristic vectors'. The tool then clusters code fragments based on the similarity scores between their vectors to extract patterns.

Machine Learning Approaches

Many techniques have applied machine-learning approaches and probabilistic models of the code to mine frequent patterns in code. For example, HAGGIGS assesses the quality of detected idioms by creating models and analyzes their measured values [3].

Related Surveys. Prior work has surveyed techniques for detecting code clones (e.g., [19, 20]), introducing a classification we apply in this survey, as well as pattern mining techniques employing probabilistic models of source code [2].

2 Mining Design Patterns

Design pattern mining tools identify design pattern implementations in code by identifying matches to a predefined template describing code's structure and behavior. These tools differ from tools which infer design rules by looking for matches to a predefined template rather than inferring patterns from examples. Design pattern detection tools vary along several dimensions [10, 1]. To search for patterns, tools often transform source code into intermediate representations, such as an AST or dependency graph. Design patterns may be differentiated through specific behavioral, structural, and semantic properties. Tools vary in their choice of which subset of properties to match and their resulting accuracy. Approaches include database queries, metric-based, graph mining, and miscellaneous approaches [1].

Database Query Approaches

Database query approaches transform the source code into an intermediate representation, such as an AST. As discussed above, compared to using source code, intermediate representations simplify search. For example, database queries may be used to extract relationships among elements and annotations to describe the semantics of code [18].

Metric-based Approaches

Metric-based approaches calculate metrics from the source code and compare these to metric-based definitions of design patterns. Metrics may include the number of public, private and protected attributes; the number of public, private and protected methods; and the number of relations in which the class is involved (associations, aggregations, and inheritance) [4].

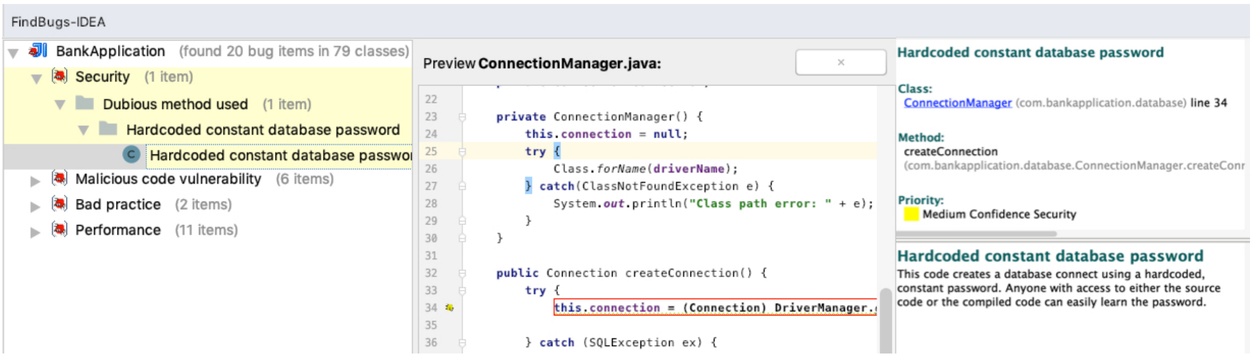
Graph Mining Approaches

In graph mining approaches, the source code and design patterns are transformed into a graph representation such as an Abstract Semantic Graph [5, 16] or dependency matrix [23, 9] to represent their structural and behavioral properties. For example, DP-Miner [9] builds a matrix to represent the relationship between each pair of classes. The value of matrix cells corresponds to the structural properties of the classes, such as generalization or aggregation. The tool matches the built matrix with design pattern matrices to find candidate design patterns. DP-Miner then checks the behavior of candidate patterns using control flow graphs and checks the semantics of the candidate patterns by analyzing the naming conventions, comments, and documentation. Another technique considers design patterns as directed dependency graphs between classes, merging sub-patterns and checking candidates by analyzing method signatures against a design pattern template [26].

Miscellaneous Approaches

Several other approaches have also explored techniques for reverse engineering design rules. One example is Pat [14], which identifies five structural design patterns by mapping C++ header files and the patterns to a Prolog representation as ‘facts’ and ‘rules’. It then executes queries to find matches. PINOT [21] statically analyzes inter-class relationships to identify structural design patterns, such as Bridge and Proxy, and applies inter-class and data flow analysis to detect behavioral design patterns, such as Singleton and Factory.

Related Surveys. Dong et al. [10] categorize design pattern detection tools along six dimensions, including the characteristics of patterns checked by tools (structural, behavioral, or semantic), the intermediate representation, and the accuracy of matching the pattern to code (exact or approximate). Our survey focuses on different dimensions. Other work surveyed 34 papers published from 1996 to 2015 to create a classification of design pattern detection tools [1], which we adopt in this survey.


FindBugs-IDEA
BankApplication (found 20 bug items in 79 classes)
Security (1 item)
Dubious method used (1 item)
Hardcoded constant database password
Hardcoded constant database passwor
Malicious code vulnerability (6 items)
Bad practice (2 items)
Performance (11 items)
Preview ConnectionManager.java:

```
    private ConnectionManager() {
        this.connection = null;
        try {
            Class.forName(driverName);
        } catch(ClassNotFoundException e) {
            System.out.println("Class path error: " + e);
        }
    }

    public Connection createConnection() {
        try {
            this.connection = (Connection) DriverManager.

        } catch (SQLException ex) {
```

Hardcoded constant database password

Class:
ConnectionManager (com.bankapplication.database) line 34

Method:
createConnection
(com.bankapplication.database.ConnectionManager.createConn

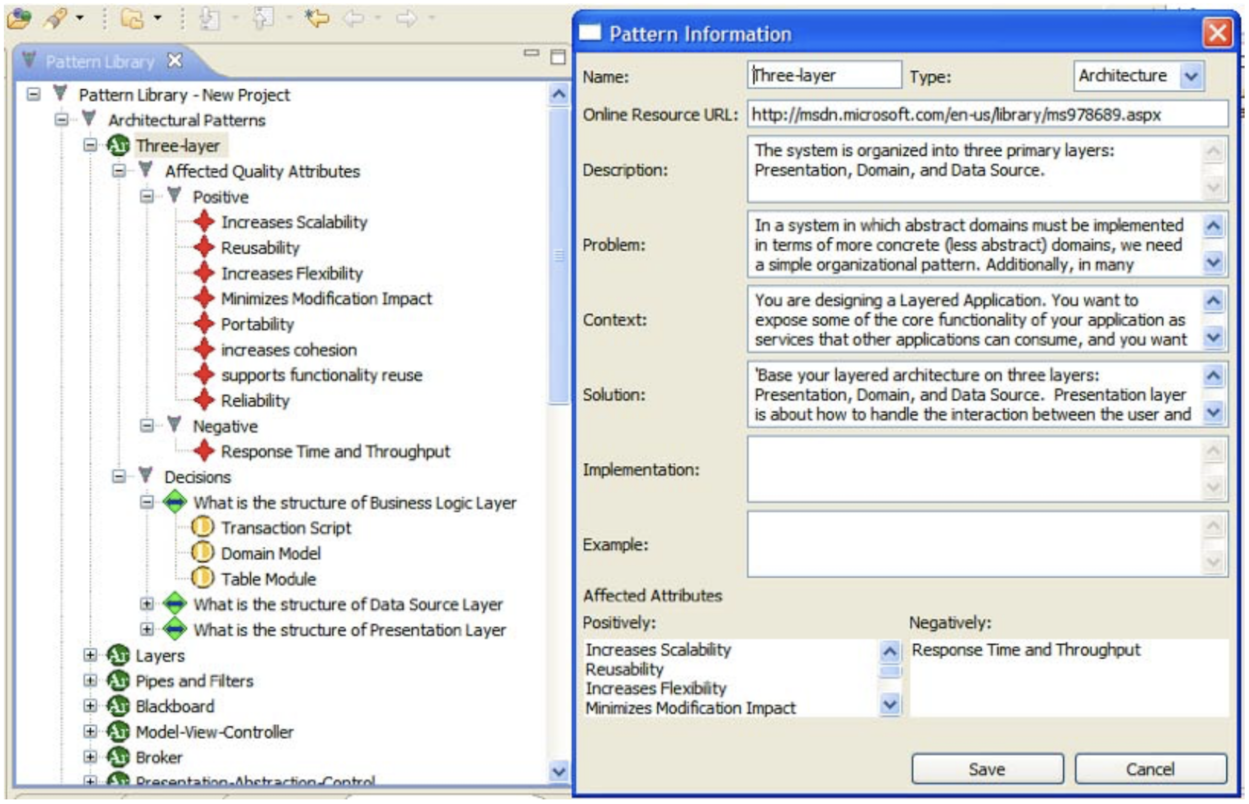
Priority:
🟨 Medium Confidence Security

Hardcoded constant database password
This code creates a database connect using a hardcoded, constant password. Anyone with access to either the source code or the compiled code can easily learn the password.

Figure 1: Static analysis tools, such as FindBugs [11], identify statements violating rules, and may also offer text explaining the rationale for the rule (bottom right).

```
<ruleset name="Custom Rules"
         xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 https://pmd.sourceforge.io/ruleset_2_0_0.xsd">
    <description>
        Custom rules for PMD
    </description>
    <rule name="BankApplication_methods"
          language="java"
          message="BankApplication must contain the following five methods: store, update, deposit, withdraw, and destroy"
          class="net.sourceforge.pmd.lang.rule.XPathRule">
        <description>
            The void functionalities in the application should be limited to a required bank functions,
            i.e., store, update, deposit, withdraw, and destroy.
        </description>
        <priority>3</priority>
        <properties>
            <property name="xpath">
                <value>
                    <![CDATA[
                    //ClassOrInterfaceBodyDeclaration/MethodDeclaration[ResultType/@Void="true" and MethodDeclarator[not(@Image="store"
                    or @Image="update" or @Image="deposite" or @Image="withdraw" or @Image="destroy")]]
                    ]]>
                </value>
            </property>
        </properties>
    </rule>
</ruleset>
```

Figure 2: Many static analysis tools, such as PMD, are extensible and support writing project-specific rules. In PMD, design rules can be written as XPath queries with a description describing the rationale.



```
Pattern Library - New Project
  Architectural Patterns
    Three-layer
      Affected Quality Attributes
        Positive
          Increases Scalability
          Reusability
          Increases Flexibility
          Minimizes Modification Impact
          Portability
          increases cohesion
          supports functionality reuse
          Reliability
        Negative
          Response Time and Throughput
      Decisions
        What is the structure of Business Logic Layer
          Transaction Script
          Domain Model
          Table Module
        What is the structure of Data Source Layer
        What is the structure of Presentation Layer
    Layers
    Pipes and Filters
    Blackboard
    Model-View-Controller
    Broker
    Presentation-Abstraction-Control
```

Name: Three-layer Type: Architecture

Online Resource URL: http://msdn.microsoft.com/en-us/library/ms978689.aspx

Description: The system is organized into three primary layers: Presentation, Domain, and Data Source.

Problem: In a system in which abstract domains must be implemented in terms of more concrete (less abstract) domains, we need a simple organizational pattern. Additionally, in many

Context: You are designing a Layered Application. You want to expose some of the core functionality of your application as services that other applications can consume, and you want

Solution: 'Base your layered architecture on three layers: Presentation, Domain, and Data Source. Presentation layer is about how to handle the interaction between the user and

Implementation:

Example:

Affected Attributes

Positively:	Negatively:
Increases Scalability	Response Time and Throughput
Reusability	
Increases Flexibility	
Minimizes Modification Impact	

Figure 3: SEURAT_Architecture [25] enables creating a catalog of design patterns.

References

[1] Mohammed Ghazi Al-Obeidallah, Miltos Petridis, and Stelios Kapetanakis. A survey on design pattern detection approaches. *International Journal of Software Engineering (IJSE)*, 7(3):41–59, 2016.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *Computing Surveys (CSUR)*, 51(4):81:1–81:37, 2018.

[3] Miltiadis Allamanis and Charles A. Sutton. Mining idioms from source code. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 472–483, 2014.

[4] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. Design pattern recovery in object-oriented software. In *International Workshop on Program Comprehension (IWPC)*, pages 153–160, 1998.

[5] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from c++ source code. In *International Conference on Software Maintenance (ICSM)*, pages 305–314, 2003.

[6] Hamid Abdul Basit and Stan Jarzabek. A data mining approach for detecting higher-level clones in software. *Transactions on Software Engineering*, 35(4):497–514, 2009.

[7] Ira D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM)*, pages 368–377, 1998.

[8] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. Discovering neglected conditions in software by mining dependence graphs. *Transactions on Software Engineering*, 34:579–596, 2008.

[9] Jing Dong, Dushyant S. Lad, and Yajing Zhao. Dp-miner: Design pattern discovery using matrix. In *International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 371–380, 2007.

[10] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855, 2009.

[11] David Hovemeyer and William Pugh. Finding bugs is easy. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 132–136, 2004.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[13] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *Transactions on Software Engineering*, 28(7):654–670, 2002.

[14] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering (WCRE)*, pages 208–215, 1996.

[15] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *Transactions on Software Engineering*, 32:176–192, 2006.

[16] Murat Oruc, Fuat Akal, and Hayri Sever. Detecting design patterns in object-oriented design models by using a graph mining approach. In *International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 115–121, 2016.

[17] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. Mining patterns in source code using tree mining algorithms. In *International Conference on Discovery Science*, pages 471–480, 2019.

[18] Ghulam Rasool, Ilka Philippow, and Patrick Mäder. Design pattern recovery based on annotations. *Advances in Engineering Software*, 41(4):519–526, 2010.

[19] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Technical Report 115, 2007.

[20] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[21] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *International Conference on Automated Software Engineering (ASE)*, pages 123–134, 2006.

[22] Boya Sun, Gang Shu, Andy Podgurski, and Brian Robinson. Extending static analysis by mining project-specific rules. In *International Conference on Software Engineering (ICSE)*, pages 1054–1063, 2012.

[23] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32:896–909, 2006.

[24] Vera Wahler, Dietmar Seipel, J. Wolff, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *International Workshop on Source Code Analysis and Manipulation*, pages 128–135, 2004.

[25] Wei Wang and Janet E Burge. Using rationale to support pattern-based architectural design. In *ICSE Workshop on Sharing and Reusing Architectural Knowledge*, pages 1–8, 2010.

[26] Dongjin Yu, Yanyan Zhang, and Zhenli Chen. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *Journal of Systems and Software*, 103:1–16, 2015.