Reverse Engineering State and Strategy Design Patterns using Static Code Analysis

Khaled Abdelsalam Mohamed, Amr Kamel
Faculty of Computers and Information, Cairo University
Giza, Postal Code: 12613, Egypt

Abstract—**This paper presents an approach to detect behavioral design patterns from source code using static analysis techniques. It depends on the concept of Code Property Graph and enriching graph with relationships and properties specific to Design Patterns, to simplify the process of Design Pattern detection. This approach used NoSQL graph database (Neo4j) and uses graph traversal language (Gremlin) for doing graph matching. Our approach, converts the tasks of design pattern detection to a graph matching task by representing Design Patterns in form of graph queries and running it on graph database.**

Keywords—Reverse engineering; source code analysis; design patterns; static analysis; graph matching; Gremlin; Joern; Neo4j

I. Introduction

Software as an artifact is not static. Software is in continuous change. During the software lifetime, there are a number of sources of change that affects it, e.g., bug fixing, new features added, requirements changes or technology changes. To make such changes, the assigned developers should have a good understanding of the software internals s/he is going to change. Typically, a team of developers implements applications. Sometimes, the developer who is assigned to change the application is not a member of the original development team, even if the developer was one of the team, it is unlikely that he knows every little detail of the software implementation. Here comes the importance of having a complete documentation of the software, so all development team have required insight of the software.

The documentation always has many problems. An extreme problem is that documentation may be lost, so the developer will need to start understanding the software from scratch, although this not always the case, the most probable problem of documentation is not being synchronized with the application. If a developer depends on this outdated documentation s/he will get wrong understanding of the software at hand, which will be an obstacle for the developer to accomplish the task.

As the documentation goes out of sync, it will be a source of problems. If a developer starts from outdated document and makes his changes without reflecting changes in the documentation continuously, the significance of the documentation will diminish over time, eventually the documentation will be useless. One reason of such problem is that the job of updating documentation is a tedious task for the developers.

A lot of research effort has been done in the field of gaining insight of legacy software and knowing the intentions of software code. In addition, it is considered one of the important reverse engineering research fields. One of the different approaches for gaining understanding of legacy software is to extract design patterns out of the source code; design patterns [1] describe high quality practical solutions to recurring programming problems.

Design patterns are a toolbox of reusable solutions and best practices that have been refined over many years to a compact format. Design patterns do not describe specific algorithms or data structures like linked list or variable length arrays, which are traditionally implemented in individual classes. As each design pattern has a specific intention, detecting them out of source code can lead to understanding the usage of different parts of the software, design patterns provide a coherent map that leads the developers through the design of the software analyzed.

This document is divided into four sections. In Section II, The different approaches used for detecting design patterns are presented. In Section III, structural similarities and behavioral differences between State design pattern and Strategy design pattern are presented. In Section IV, The different techniques and frameworks used for doing static analysis to source code are presented. In Section V, Our approach for detecting design patterns in source code using graph enrichment and static analysis techniques is presented. Finally, Section VI, offers the conclusions and future work.

II. Related Work

The architecture design of software highly affects its quality. The high quality software follows design patterns. The mining of design patterns can be helpful in understanding and knowing design decisions in legacy systems [2]-[5].

The design pattern recovery is considered one of the hot topics in reverse engineering research field [2], [6], [7]. There are many approaches used in literature to recover design patterns from source code to facilitate software maintenance [8], [9] and program comprehension [10]-[12]. The techniques used in literature can be classified based on two factors [13], the type of analysis and the search methodology.

A. Analysis Type

Based on the analysis type, the pattern recovery approaches can be classified [13] into structural analysis, behavioral analysis and semantic analysis.

Structural analysis [14] are based on recovering inter-class relationships such as class inheritance, association, composition, modifiers of classes and methods, method parameters, etc. They focus on recovering structural design patterns such as Proxy, Decorator and Adapter, but they completely miss the behavioral aspects of design patterns.

Behavioral analysis [15] focuses on the execution behavior of the program. These approaches are based on dynamic analysis, machine learning and static analysis techniques to extract behavioral aspects of the pattern. Supplementing behavioral analysis by structural analysis techniques helps in recovery of identical or weak-structure patterns where structural analysis fails.

Semantic analysis approaches supplements both structural and behavioral analysis approaches to reduce the false positive rate of recognition of design patterns. The semantic analysis approach uses the naming convention of classes and methods in recovering different roles inside design patterns.

B. Searching Techniques

Based on the searching techniques, the pattern recovery approaches can be summarized as follows:

1) Database queries

In this approach, the source code is first transformed to an intermediate representation such as (ASG, AST, XMI, meta-data and UML structures etc.) then SQL queries are used to extract information from a specific representation.

2) Constraint Resolver

The approach [14] used by The PTIDEJ team is a multilayered approach, where design motifs are described as constraint systems where each role is represented as a variable. Relationships among roles are represented as constraints among these variables.

3) XPG formalism and parsing

This approach [6] used a technique where SVG (scalable vector graphics) format is used as an intermediate representation of source code and design patterns are represented using a visual language. Patterns are recovered using a visual language parsing technique by mapping visual language grammar of the patterns with the graph representation. The advantages of these approaches are the visualization and good precision, but are limited only to structural design patterns.

4) UML structures and matrices techniques

Metrics techniques [16]-[18] compute program metrics such as generalization, aggregation, association, etc. from different representations of source code and then a number of techniques are used to compare metric values of each design pattern definition with source code metrics. These techniques are computationally efficient because of search space reduction through filtration.

III. State vs Strategy Design Patterns

State and Strategy design patterns are two interesting patterns, as both of them have the same structure although each of them have a different behavior.

Balanyi and Rudolf [19] stated that during their process of pattern formalization, they found an interesting problem, which is that both State and Strategy patterns have identical structure, and the differences between them are in motivation and intention that they could not formalize.

Aikaterini et al. [20] proposed a method to automatically transform/refactor source code to comply with the Strategy design pattern. Their method complements JDeodorant [21] that focuses mainly on the State pattern, by taking into account behavioral properties of the Strategy design pattern during candidate selection phase.

Von Detten and Platenius [22] used dynamic analysis to analyze the runtime behavior of the system. First static analysis is used to detect the structure of a design pattern, the detected classes, methods are annotated, then during the dynamic analysis phase the behavior of annotated classes, and methods are traced during the software execution. For each pattern candidate, a number of traces are generated. A behavioral analysis algorithm assess if traces of each pattern candidate conform to the corresponding behavioral pattern. If most of the traces of a candidate match the behavioral pattern, the candidate pattern is accepted and if the most of traces do not match then the candidate pattern is rejected.

Hummel and Burger [23] mentioned that the class diagram of the strategy and state patterns are identical from the class diagram perspective. In addition, the main difference between them resides in who controls the change of state or strategy. The state implementations have control over state changes themselves, but for strategy pattern, the client is responsible for the changes of the applied strategy.

Uchiyama et al. [24] used an approach of metrics and machine learning technique to detect design patterns. This work was interested in distinguishing between State patterns from Strategy pattern. They firstly using various metrics and their machine learning identify the roles and secondly detect patterns as structure of those roles.

The below class diagram (Fig. 1) shows the structure of the Strategy Design Pattern.

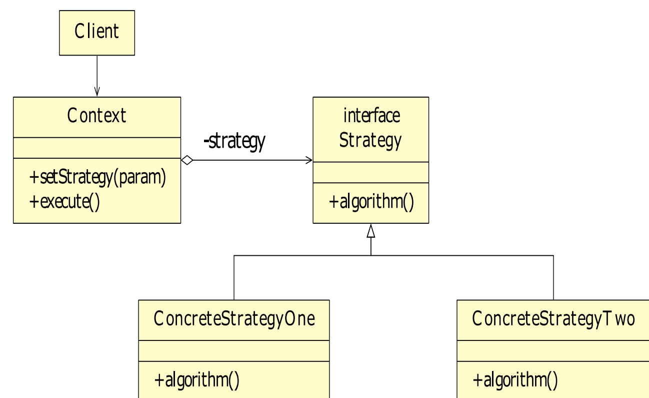

Fig. 1. Class diagram of strategy design pattern.

A. Structural Characteristics of Strategy Design Pattern

1) Classes: Client, Context, Strategy, ConcreteStrategy.

2) Use: Client uses Context

3) Aggregation: Context aggregates Strategy.

4) Inheritance: More than one ConcreteStrategy inherits Strategy.

5) Abstract Method: Strategy contains an abstract method.

6) Method Overriding: ConcreteStrategy(ies) override Strategy Abstract Method "algorithm".

The below sequence diagram (Fig. 2) show the behavior of Strategy pattern:

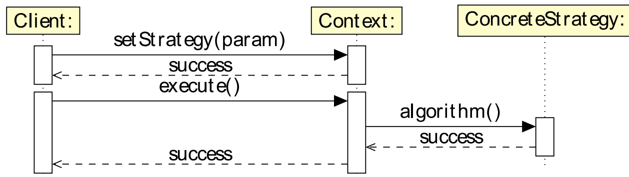

Fig. 2. Sequence diagram of strategy design pattern.

B. Behavioral Characteristics of Strategy Design Pattern

1) Call A: A call from "Client" to "Context", to set the required strategy.

2) Object Creation: "Context" created "ConcreteStrategy", based on value sent by "Context".

3) Call B: A call from "Context" to the abstract method implementation in "ConcreteStrategy".

4) Call C: A call from "Client" to "Context", to start execution.

5) Call D: A call from "Context" to "ConcreteStrategy", to perform the algorithm.

IV. Techniques and Frameworks

Static analysis [25] is the most frequently used approach for code analysis, dynamic analysis needs that the source code to be runnable; however, static analysis can be used for incomplete source code.

A number of techniques are used for doing static analysis for source code. Next, a number of most common static analysis techniques are presented.

A. Techniques

1) Call Graph

Gall Graph [26] represents the possible callers at each call site in each function. Call Graph is a directed that represents the relationships between the program's functions. There is a wide range of algorithms for call graph construction [26], e.g. RTA, 0-CFA and SCS.

2) Control Flow Graph

Control flow [27] graph, is a directed graph where nodes represent program statements, and edges represent flow paths from one program statements to another.

Constructing CFG can be for source code or byte codes, for example JavaPDG [28], constructed the Control Flow Graph from byte code using the following steps:

a) Get all instructions/statements of the method.

b) Create a node that represent method entry.

c) Make a link between entry node and first instruction/statement.

d) Create a node that represent method exit.

e) Get reference to last instruction/statement.

*f)*Make a link between last instruction and method exit node, if last instruction/statement is not "Return".

g) Make a reference to previous and current instruction and Loop all instructions.

h) If previous instruction type is not ("CP" or "JU" or "Return"), make link between pre and cur instructions.

*i)*If current instruction type is ("CP" or "JU"), make a link between cur instruction and all jump labels.

*j)*If current instruction of type "Return", make a link between cur instruction and method exit node.

3) Dominator Tree

A dominator tree [29] is a graph $G = (V, E, r)$, Where V is the set of vertices, E is the set of edges and r is the root node of the graph. Every node except root node in the graph has a unique immediate dominator. If two nodes "v" and "w" in the dominator tree, and "v" is the ancestor of "w", then "v" dominates "w". Node "v" dominates "w" if all paths from the entry node to "w" contains "v". In addition, "w" post-dominates "v" if all paths from "v" to exit node contains "w".

The dominator tree is computed from Control Flow Graph. By having CFG and DT, control dependence graph can be derived.

4) Control Dependence Graph

Control Dependence Graph [30] is a merge between Control Flow Graph and Dominator Tree, It can be defined as a directed graph "G", it has two unique entries, entry node "START" and exit node "STOP". For any node "N" there exists a path from "START" to "N" and from "N" to "STOP". So, node "Y" control dependent on "X" iff:

- There is a directed path "P" from "X" to "Y", which contains node "Z", where "Y" post-dominate "Z".
- "Y" doesn't post-dominate "X".
- Node "V" is post-dominated by "W", if every directed path from "V" to "STOP" contains "W".

5) Data Dependence Graph

A data dependence graph (DDG) for every method is calculated by tracking data flows on its CFG. A definition-use chain, i.e., one instruction assigns a value to an abstract variable, usually represents a data flow and the other instruction uses the value. Reaching-definition and upward-exposed-uses analyses are conducted following the steps:

Analyze the effect of each instruction in terms of its variable definition and use sets.

Iteratively propagate the information over the CFG;

During each iteration, inspect whether there is any unknown definer/assigner of the variable(s) used in each instruction, and update its information sets accordingly.

Once the information propagation ends (no changes are found), the data dependences between instructions is calculated by the definition-use chain analysis.

6) Program Dependence Graph

A PDG [30] is defined as a labeled, directed graph that maps out control dependences and data dependences between elements in a program.

7) System Dependence Graph

A system dependence graph (SDG) [31] is a generalization of PDG and contains one procedure dependence graph (pDG) for each method.

8) Code Property Graph

A single representation alone to represent the source code in insufficient. Fabian et al. [32] combines three representations into a unified data structure. In [32], author introduced a new concept of Code Property Graph which models ASTs, CFGs and PDGs as property graphs.

Fabian et al. [32] showed that common types of vulnerabilities can be modeled as a traversal of code property graph, also by importing code property graph into a graph database, makes traversals can be executed efficiently on large code base.

A code property graph is a property graph $G = (V, E, \lambda, \mu)$ constructed from AST, CFG and PDF of source code:

$V = V_A$,

$E = E_A \cup E_C \cup E_P$,

$E = \lambda_A \cup \lambda_C \cup \lambda_P$ And

$E = \mu_A \cup \mu_C$,

B. Frameworks

Here, a number of frameworks that provides implementations, for different static analysis techniques, first works on binary level, and the second works on source code level.

1) JavaPDG

JavaPDG [28] implements static dependence analysis for Java Virtual Machine (JVM) bytecode. The tool parses the bytecode of a Java program, computes the SDG and related graphs, and stores the data for each program in a database. JavaPDG includes tools for visualizing the graphs it produces and for exporting the data in the JSON format. Additionally, users are able to query the output using SQL by utilizing Apache Derby. The analysis process takes as input the compiled class files of a Java program, and yields a SDG and related graphs as the final output.

The steps for building SDG are as follows:

a) Preprocessing

In the SDG, one PDG vertex represents each instruction. Artificial entry and exit vertices for every method are added to the graph to represent the start and end of the method, respectively. A vertex is added for every call-site as its actual-output parameter if the callee method has any return value.

b) Inter-procedural Analysis

An SDG is a collection of interconnected pDGs, each of which is composed of the CDG and DDG for a method. The static call graph of a program is used to investigate communications between methods. Based on the call graph, three types of inter-procedural control and data dependences are computed.

The output SDG is a labeled, directed graph consisting of multiple PDGs. Besides the SDG, JavaPDG outputs some additional information, including:

- Static structure of a program that describes classes, fields, methods, and relationships among them.
- Variable information that contains the name, type and scope of every class field, object field and local variable (including formal input parameter).
- Control flow graphs and dominance trees that are constructed during dependence analysis and share the same vertices as in the SDG.
- A static call graph whose vertices correspond to Java methods and whose edges represent potential caller-callee relationships indicated in the program.

2) Joern

Joern [33] is a platform for robust analysis of C/C++ code. It generates code property graphs; code property graph [32] consists of code's syntax, control-flow, data-flow and type information. These graphs are then stored in Neo4J database. By this, it is possible to do code mining through running search queries formulated in the graph traversal language Gremlin.

Joern platform [33] consists of three components joern(-core), python-joern and joern-tools. Joern(-core) is the main component, it takes the source code and parses it, creates a code property graphs [32] and finally, import the graphs into Neo4j database. Python-joern is a python interface to Joern database. It provides a number of utilities for the common operations of traversing code property graphs. Joern-tools is a collection of command line tools that makes using python-joern utilities possible form the shell.

3) Gremlin

Gremlin [34] is the graph traversal language of Apache TinkerPop. Gremlin is a functional, data-flow language that enables users to succinctly express complex traversals on (or queries of) their application's property graph.

Gremlin recently appeared in a number of works such as Model-to-Model transformation [35], modeling and discovering vulnerabilities in source code [32].

V. Approach

In our approach, detecting behavioral design patterns from source code using static analysis techniques is chosen. Program

Dependence Analysis, Control Dependence Analysis and Data Dependence Analysis are applied on source code to be able to capture the behavioral characteristics of design patterns.

In our approach, The detection problem is represented as a graph matching problem, the source code graphs is stored in a graph database e.g. Neo4j, and the design pattern features are extracted by running graph matching queries against the database where the source code graphs are saved.

In our approach, Joern platform [33] is used to analyze the source code and save the analyzed source code in graph database.

Joern platform is designed mainly for the detection of vulnerabilities in code. Therefore, Joern is mainly interested in C++ code at functions level and not interested in Object Oriented interactions between classes.

As inheritance between classes forms an important information that is required during the process of detecting design patterns, a minor change to Joern platform is made to store the parent class of each class during the parsing step of the analysis process.

The following figure (Fig. 3) shows a high-level view of the steps of our approach:

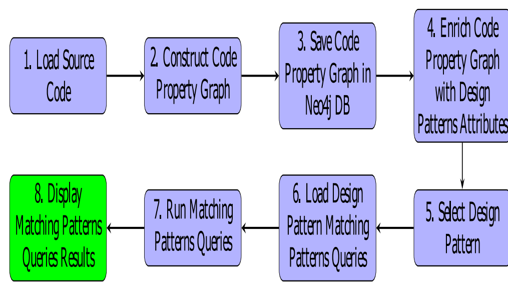

Fig. 3. Approach steps.

The steps of our approach is as follows:

1) Load Source Code

2) Generate Code Property Graphs [32] of the source code under investigation

3) Insert the Code Property Graph into a graph database.

4) Enrich Generated Code Property Graph with properties and relations between nodes to simplify the graph matching steps.

5) Decide the design pattern(s) to detect.

6) Load the list of features (structural and behavioral) that represent the design pattern.

7) Load the corresponding graph matching query for each design pattern features

8) For each design pattern, run features detection queries using Gremlin language [34] against the Enriched Code Property Graphs in the Neo4j.

9) Inspect detected features and decide if design pattern instance is found or not.

10) Display results.

Our approach enriches the Code Property Graph with a number of properties and relations between vertices to make the phase of detecting State and Strategy patterns straight forward. Once these relations and properties are constructed, the pattern detection graph matching algorithms for State and Strategy patterns are used to detect State and Strategy patterns from the Enriched Code Property Graph.

To express the capabilities of our approach, differentiating between State and Strategy design patterns is selected, as they are identical from the structural perspective but differs from the behavioral and run time perspective. Our approach show that differentiating between these patterns is possible, while still using static analysis and no dynamic analysis is needed.

The enrichments required for differentiating between State and Strategy Patterns are listed:

1) Methods to Classes: C++ class methods can be defined outside its class, in such case Joern tool does not link between the class and its member method, so a link between methods and their classes is created.

The steps are as follows:

a) List all methods that their names contains symbol "::".

b) Split the method name into two parts, class name and method name.

c) Search for class with the same name of first part of full method name.

d) Make a link of type "IS_CLASS_OF" between the class and method.

2) Inheritance: An inheritance relation between each super class and their subclasses (Fig. 4).

The steps used to construct the inheritance relationship:

- List all classes that their base class name not equals to "<unnamed>".
- For each class in the list, get the class's base class name.
- Search for a class that its name equals to child's base class name.
- Make a link of type "INHERITS" between the class and its base class.

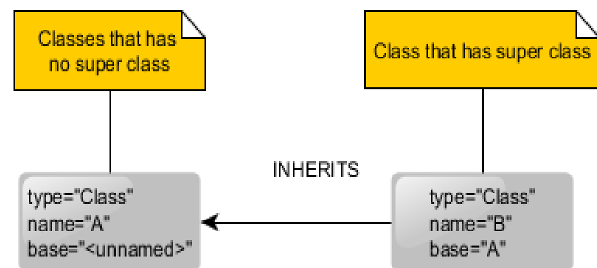

Fig. 4. Inheritance relation.

3) Abstract (Virtual) Methods: Abstract methods can have two types, one that has no body definition (Pure Virtual), second that *has* body definition and declared with virtual keyword as modifier (Fig. 5).

The steps to mark a method declaration as virtual are as follows:

a) Get list of node that are of type “Decl”

b) Extract nodes that contain brackets, as indication that they are methods declaration.

c) Extract nodes that do not have body definition.

d) Mark extracted nodes as abstract.

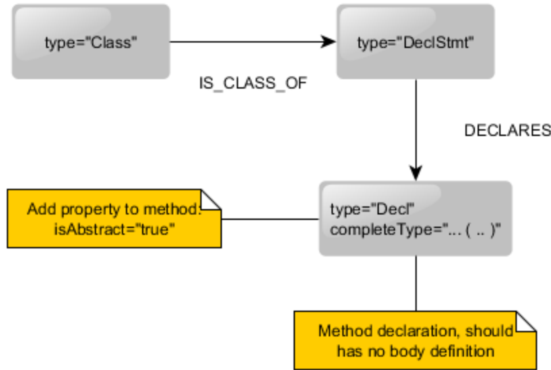

Fig. 5. Pure abstract method.

The steps to mark a method with body definition as virtual are as follows (Fig. 6):

a) Get list of node that are of type “Function”.

b) Traverse to the return type of the function.

c) If return type contains keyword “virtual”, then this function is marked as abstract.

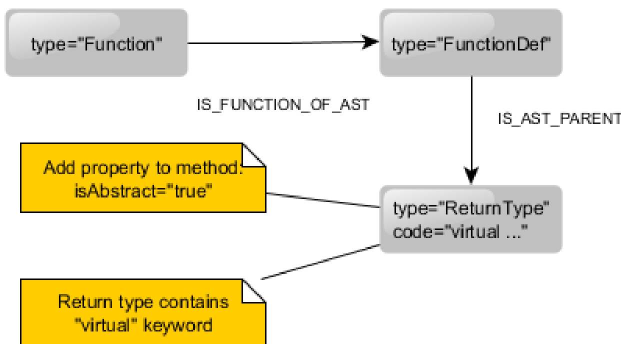

Fig. 6. Abstract method.

4) Class Aggregates Class: A new relation between two classes are created if one aggregates the other (Fig. 7).

The steps to construct aggregation relation between two classes are:

a) Get all declaration statements for each class e.g. “Class A”.

b) For each declaration statement extract class name from its “baseType“.

c) Search for the class with same name extracted in previous step “Class B”.

d) Create a link between that represents the aggregation between “Class A” and “Class B”.

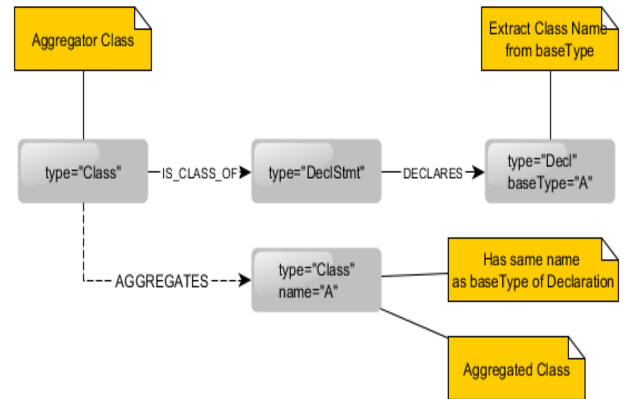

Fig. 7. Class aggregates class.

5) Method Creates Class: A relation between a method and a class is created, if a method creates a class (Fig. 8).

The steps to create relation between class and the method that creates it are:

a) Get all method statements that contains “new” statement.

b) Extract class name from the new statement.

c) Search for a class that has the same name of step b.

d) Create a link of type “Create” between the Method and the Class.

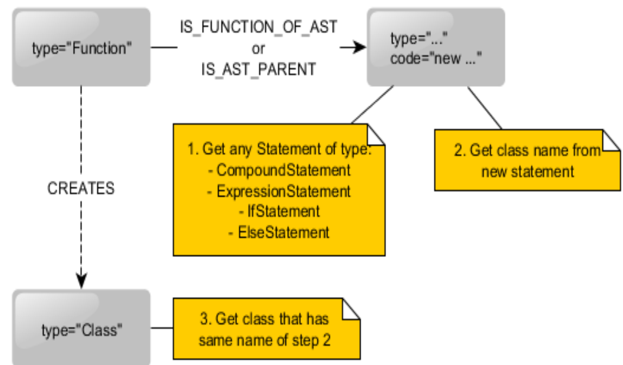

Fig. 8. Method creates class.

6) Method Overrides Method: A new relation between two methods, if one method overrides the other method (Fig. 9).

a) Get list of all functions and declaration statements.

b) Get list of classes of step “a”.

c) Get list of classes that are super class of classes in step “b”.

d) Get list of all functions and declaration statements that are abstract of classes in step “c”.

e) Filter list of step “d”, which Subclass method name should be equals to Superclass method name of step “c”.

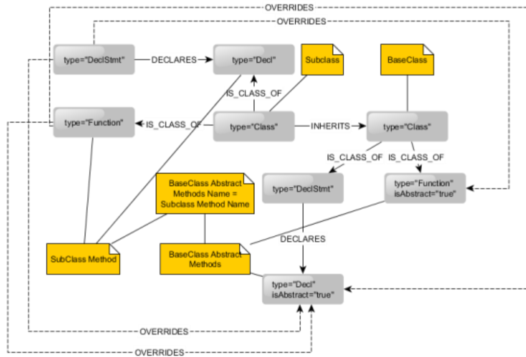

Fig. 9. Method overrides method.

7) Method Calls Method: A relation between two methods are created if one *method* creates the other.

The steps to construct these new relations (Fig. 10) are:

a) Get list of all method "Caller Methods".

b) Get list of classes of step "a", and keep method that are not related to classes e.g. main method.

c) Get list of all statements of type "Callee" of step "a" "Call Sites".

d) Get list of nodes of types "PtrMemberAccess", "MemberAccess", or "Identifier" that are linked to step "c".

e) Get list of nodes that are linked to nodes of type "PtrMemberAccess" or "MemberAccess" of step "d" with in-edge of type "USE".

*f)*Get list of nodes of type "Parameter", "Decl", or "IdentifierDeclStatement" and having in-edge of type "DEF" from step "e".

g) Keep nodes of type "Identifier" or "Symbol" that are not in step "f" but have declaration in classes of caller methods of step "a".

h) Loop lists of steps "f" & "g".

*i)*Get callee method name.

*j)*Get callee class name.

k) Get node represented by class and method names (Callee Method).

*l)*Create a link between Caller Method and Callee Method (Step "h.iii").

*m)*Create a link between Call Site (Step "c") and Callee Method (Step "h.iii").

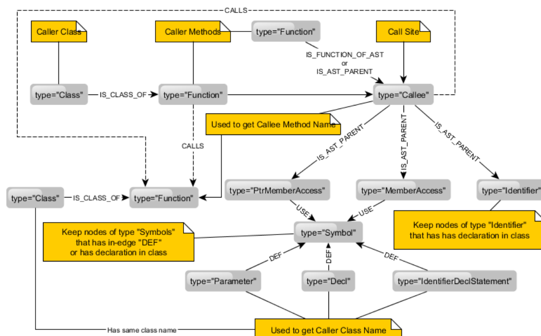

Fig. 10. Method calls method.

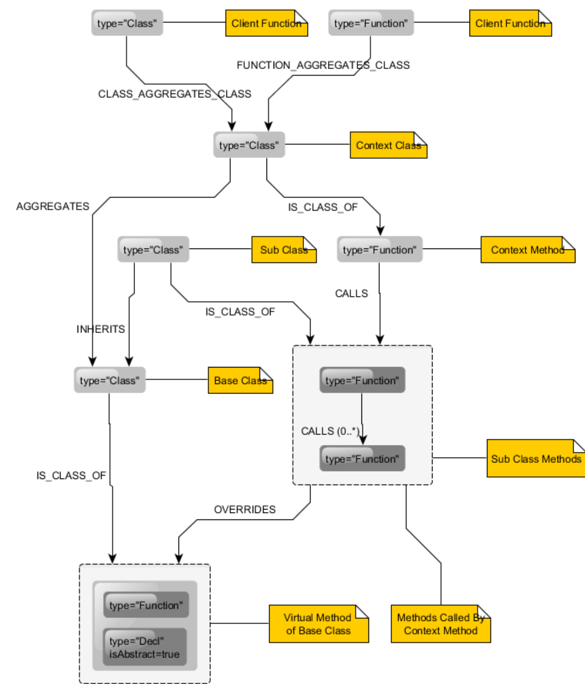

Fig. 11. State and strategy design pattern candidates.

After enriching phase, the code graph is ready for the detection phase, the detection phase for State and Strategy design patterns is divide into two steps, first step to detect candidates that can be State or Strategy (Fig. 11), this step captures the structure of these design patterns. The second step is for differentiating between State and Strategy patterns.

The steps for deciding if a candidate is a State or a Strategy design patterns are:

a) Loop each candidate.

b) Get symbols that used by Context Class to aggregate Base Class.

c) Get methods that use the symbols from step (b).

d) Check if methods from step (c) includes sub classes methods from pattern candidates.

e) If step (d) is true then the candidate is a State design pattern.

*f)*Check of methods from step (c) includes the client method from pattern candidates.

g) If step (f) is true, then the candidate is a Strategy design pattern.

VI. Conclusion and Future Work

In this work, an approach is presented for detecting design patterns in source code, by representing the source code in form of a special graph named Code Property Graph [32], using Joern platform [33]. In addition, our approach is shown to able to differentiate between State and Strategy design patterns, which are identical from structural perspective, but differs at run time, using advanced static analysis techniques without the need to use run time dynamic analysis. The code property graph is enriched by constructing new properties and

relationships between vertices of the graph, the enrichments done by our approach presented a number of techniques to transform graphs from the functions paradigm level to the level of object oriented paradigm, so that code graph is ready for object oriented analysis and design patterns detection.

In this work, C++ code is used, because Joern platform currently supports C++, in our future work we will work on supporting Java programming language, to be able to compare our results with other approaches, as most design pattern detection benchmarks are java based [4], [11], [36], [37]. Our approach is not dependent on a specific language for enrichment and design pattern detection, as it depends on manipulating the code graph directly at run time before running the detection algorithms, which depends on the code graph also.

In future work, a catalogue of all relationships and properties of design patterns will be created, to enrich the code graph with these relationships and properties as a step before pattern detection step, so a catalogue containing a one to one mapping between a design pattern and it graph query will be available.

Design pattern can have more than one variant [38], in our future work, more than one graph definition to each design pattern will be supported, and detection algorithm will search for all different variants of design patterns to increase the true positive rate of our detection approach. Constructing graphs using design pattern concepts as relationships between vertices will make adding new design patterns or new variants of the design patterns more easily and user friendly.

REFERENCES

[1] Vlissides, John and Helm, Richard and Johnson, Ralph and Gamma, Erich, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Pub Co, 1995.

[2] Costagliola, Gennaro and De Lucia, Andrea and Deufemia, Vincenzo and Gravino, Carmine and Risi, Michele, "Design pattern recovery by visual language parsing," in Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on, 2005.

[3] Dong, Jing and Zhao, Yajing and Peng, Tu, "A review of design pattern mining techniques," International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 06, pp. 823--855, 2009.

[4] Fontana, Francesca Arcelli and Caracciolo, Andrea and Zanoni, Marco, "DPB: A benchmark for design pattern detection tools," in Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, 2012.

[5] L. Wendehals, "Improving design pattern instance recognition by dynamic analysis," in Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, 2003.

[6] De Lucia, Andrea and Deufemia, Vincenzo and Gravino, Carmine and Risi, Michele, "Behavioral pattern identification through visual language parsing and code instrumentation," in Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, IEEE, 2009.

[7] Dong, Jing and Zhao, Yajing and Sun, Yongtao, "A matrix-based approach to recovering design patterns," IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 39, no. 6, pp. 1271--1282, 2009.

[8] Ng, Janice Ka-Yee and Guéhéneuc, Yann-Gaël and Antoniol, Giuliano, "Identification of behavioural and creational design motifs through dynamic analysis," Journal of Software: Evolution and Process, vol. 22, no. 8, pp. 597--627, 2010.

[9] Fulop, Lajos Jeno and Ferenc, Rudolf and Gyimothy, Tibor, "Towards a benchmark for evaluating design pattern miner tools," in Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, 2008.

[10] Guéhéneuc, Yann-Gaël and Antoniol, Giuliano, "Demima: A multilayered approach for design pattern identification," IEEE Transactions on Software Engineering, vol. 34, no. 5, pp. 667--684, 2008.

[11] Kniesel, Gunter and Binun, Alexander and Hegedus, Peter and Fulop, Lajos Jeno and Chatzigeorgiou, Alexander and Gu{\'e}h{\'e}neuc, Yann-Ga{\"e}l and Tsantalis, Nikolaos, "DPDX--Towards a Common Result Exchange Format for Design Pattern Detection Tools," in Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, 2010.

[12] Kniesel, Gunter and Binun, Alexander, "Standing on the shoulders of giants-a data fusion approach to design pattern detection," in Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on, 2009.

[13] Rasool, Ghulam and Streitfdert, Detlef, "A survey on design pattern recovery techniques," IJCSI International Journal of Computer Science Issues, vol. 8, no. 2, pp. 251--260, 2011.

[14] De Lucia, Andrea and Deufemia, Vincenzo and Gravino, Carmine and Risi, Michele, "An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis," in Software Maintenance (ICSM), 2010 IEEE International Conference on, 2010.

[15] Binun, Alexander and Kniesel, Günter, "Joining forces for higher precision and recall of design pattern detection," CS Department III, Uni. Bonn, Germany, Technical report IAI-TR-2012-01, 2012.

[16] Guéhéneuc, Yann-Gaël and Guyomarc'h, Jean-Yves and Sahraoui, Houari, "Improving design-pattern identification: a new approach and an exploratory study," Software Quality Journal, vol. 18, no. 1, pp. 145--174, 2010.

[17] Antoniol, Giuliano and Fiutem, Roberto and Cristoforetti, Luca, "Design pattern recovery in object-oriented software," in Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on, 1998.

[18] von Detten, Markus and Becker, Steffen, "Combining clustering and pattern detection for the reengineering of component-based software systems," in Proceedings of the joint ACM SIGSOFT conference--QoSA and ACM SIGSOFT symposium--ISARCS on Quality of software architectures--QoSA and architecting critical systems--ISARCS, 2011.

[19] Balanyi, Zsolt and Ferenc, Rudolf, "Mining design patterns from C++ source code," in Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, 2003.

[20] Christopoulou, Aikaterini and Giakoumakis, Emmanouel A and Zafeiris, Vassilis E and Soukara, Vasiliki, "Automated refactoring to the Strategy design pattern," Information and Software Technology, vol. 54, no. 11, pp. 1202--1214, 2012.

[21] Tsantalis, Nikolaos and Chatzigeorgiou, Alexander, "Identification of refactoring opportunities introducing polymorphism," Journal of Systems and Software, vol. 83, no. 3, pp. 391--404, 2010.

[22] Von Detten, Markus and Platenius, Marie Christin, "Improving Dynamic Design Pattern Detection in Reclipse with Set Objects," in In Proceedings of the 7th International Fujaba Days, 2009.

[23] Hummel, Oliver and Burger, Stefan, "Analyzing source code for automated design pattern recommendation," in Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics, 2017.

[24] Uchiyama, Satoru and Kubo, Atsuto and Washizaki, Hironori and Fukazawa, Yoshiaki, "Detecting design patterns in object-oriented program source code by using metrics and machine learning," Journal of Software Engineering and Applications, vol. 7, no. 12, p. 983, 2014.

[25] García-Ferreira, Iván and Laorden, Carlos and Santos, Igor and Bringas, Pablo García, "A survey on static analysis and model checking," in International Joint Conference SOCO'14-CISIS'14-ICEUTE'14, 2014.

[26] Grove, David and DeFouw, Greg and Dean, Jeffrey and Chambers, Craig, "Call graph construction in object-oriented languages," ACM SIGPLAN Notices, vol. 32, no. 10, p. 108–124, 1997.

[27] F. E. Allen, "Control flow analysis," ACM Sigplan Notices, vol. 5, pp. 1--19, 1970.

[28] Shu, Gang and Sun, Boya and Henderson, Tim AD and Podgurski, Andy, "JavaPDG: A new platform for program dependence analysis," Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. IEEE, 2013.

[29] Lengauer, Thomas and Tarjan, Robert Endre, "A fast algorithm for finding dominators in a flowgraph," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 1, no. 1, pp. 121--141, 1979.

[30] Ferrante, Jeanne and Ottenstein, Karl J and Warren, Joe D, "The program dependence graph and its use in optimization," ACM Transactions on Programming Languages and Systems (TOPLAS) 9.3, pp. 319-349, 1987.

[31] Horwitz, Susan and Reps, Thomas and Binkley, David, "Interprocedural Slicing Using Dependence Graphs," ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, p. 26–61, 1990.

[32] Yamaguchi, Fabian and Golde, Nico and Arp, Daniel and Rieck, Konrad, "Modeling and discovering vulnerabilities with code property graphs," Security and Privacy (SP), 2014 IEEE Symposium on, pp. 590-604, May 2014.

[33] "Joern Website," [Online]. Available: http://www.mlsec.org/joern/.

[34] "The Gremlin Graph Traversal Machine and Language," [Online]. Available: https://tinkerpop.apache.org/gremlin.html. [Accessed 2017].

[35] G. a. S. G. a. C. J. Daniel, "Mogwaï: a framework to handle complex queries on large models," in Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on, 2016.

[36] Tsantalis, Nikolaos and Chatzigeorgiou, Alexander and Stephanides, George and Halkidis, Spyros T, "Design pattern detection using similarity scoring," IEEE transactions on software engineering, vol. 32, no. 11, 2006.

[37] Y.-G. Guéhenéuc, "P-mart: Pattern-like micro architecture repository," Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories, 2007.

[38] Bayley, Ian and Zhu, Hong, "Formal specification of the variants and behavioural features of design patterns," Journal of Systems and Software, vol. 83, no. 2, pp. 209--221, 2010.