

A Static Reference Flow Analysis to Understand Design Pattern Behavior

Chanjin Park¹, Yoohoon Kang¹, Chisu Wu¹ and Kwangkeun Yi²

School of Computer Science and Engineering

Seoul National University, Seoul, Korea

¹ {cjpark, rmaker, wuchisu}@selab.snu.ac.kr, ² kwang@cse.snu.ac.kr

1. Introduction

Design patterns are actively used by developers expecting that they provide the design with good quality such as flexibility and reusability. However, according to industrial reports on the use of design patterns, the expectation is not always realized [1, 2]. Especially, [2] points out two causes of inappropriately applied patterns from a case study on a large commercial project: developers inexperienced in design patterns and no connection with project requirement. Wrong decisions on the use of design patterns make the program difficult to understand, and refactoring the program to improve the underlying structure, especially without documentation, can be very tricky.

To eliminate wrongly applied patterns or document important decisions automatically, design pattern recovery is important for not only the development phase but also the maintenance phase. Many design pattern recovery approaches focus on structural characteristics and do not touch set-up behavior that configures links between participants and precedes pattern behavior [3, 4].

To detect design patterns implemented in program code more precisely and to show their behavior, we analyze program at expression level. Our approach is based on statically approximating run time behavior among pattern participants. For this, a static program analysis technique is used. Many static analysis techniques for object-oriented languages exist mainly for optimizing compiler in program analysis area [5, 6].

2. Describing Design Pattern Behavior

To identify the code implementing a design pattern, we represent the program with a graph $P = (V, E_{CG} \cup E_{RFG})$, where V is set of vertices consisting of methods and fields, and E_{CG} and E_{RFG} are the set of directed edges and correspond respectively to edge set in call

graph and edge set in reference flow graph. An edge in E_{CG} is method-to-method call relation and represents that one method implementation has call expression to the other. In addition, an edge in E_{RFG} represents that a reference flows among methods and fields. Three types of reference flow edges are possible: method-to-field (field write), method-to-method (parameter passing and return), field-to-method (field access).

We propose pattern role graph (PRG) describing design pattern behavior to detect it from program. In the graph, a vertex represents a role in design pattern, and an edge does call or reference flow relation between roles. A role can be assigned for method and field. Figure 1 shows the graph consisting of roles and their relations in command design pattern. The solution part of command design pattern defines as participants Command, ConcreteCommand, Client, Invoker, and Receiver [7].

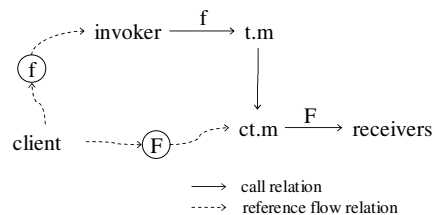


Figure 1. PRG for command pattern

In the figure, $t.m$ represents execute method of Command class, $ct.m$ does overriding method of $t.m$, and call relation from $t.m$ to $ct.m$ is a supplementary relation for dynamic binding. The role of $invoker$ is assigned for methods that read the field f and use it as a target reference of call to $t.m$. The role of $receivers$ is assigned for methods that are called with F when executing a command. F represents a set of field variables used in calls to $receivers$. Methods playing client role create ConcreteCommand objects and set their references to f and references of Receiver's to F .

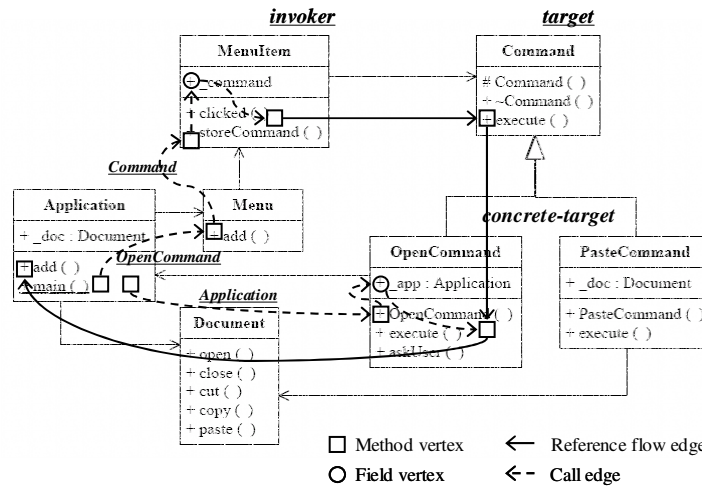


Figure 2. An Annotated Class Diagram with Call and Reference Flow Relations

3. An Example of Command Design Pattern

Figure 2 shows the annotated class diagram with call and reference flow relations that are obtained by statically analyzing program at expression level. The relations are sliced according to the command PRG in Fig. 1. We can see from the diagram that how the directed association from MenuItem to Command is set and how the reference used when executing OpenCommand is passed.

Following show the methods corresponding to each role in the command PRG. To make the execution of command to be complete, invoker's command reference and the reference to call command's receivers should be set. Reference flow information shows what operations are required to execute command at method and field level.

- invoker : MenuItem.clicked
- receivers : Application.add
- t.m: Command.execute
- ct.m: OpenCommand.execute
- client: Application.main, OpenCommand.OpenCommand, Menu.add, MenuItem.storeCommand

4. Conclusion

We implemented a static reference flow analysis for Java in which the reference flow graph is constructed by collecting flow information from program code and navigating the graph to obtain reference flow sequences for a given PRG.

We made an experiment on Command Pattern with JHotDraw [9], and our analysis detected more pattern

behavior information such as how the commands are configured to call receivers of command execution. This information can help refactor or restructure the pattern structure implemented in program code.

References

- [1] J.M. Bieman, D. Jain, and H.J. Yang, "OO Design Patterns, Design Structure, and Program Changes: An Industrial Case Study", *Proc. of IEEE International Conf. on Software Maintenance (ICSM '01)*, 2001.
- [2] P. Wendorff, "Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project", *Proc. of Fifth European Conf. on Software Maintenance and Reengineering (CSMR'01)*, 2001.
- [3] U. P. Schultz, J. L. Lawall, C. Consel, "Specialization Patterns", In *The Fifteenth IEEE International Conf. on Automated Software Engineering (ASE'00)*, 2000.
- [4] J. Niere, W. Schäfer, J.P. Wadsack, and L. Wendehals, "Towards Pattern-Based Design Recovery", *Proc. of the 24th International Conf. on Software Engineering (ICSE '02)*, 2002.
- [5] C. Krämer and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software", *Proc. of 3rd Working Conf. on Reverse Engineering (WCRE '96)*, 1996.
- [6] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis", *Proc. of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, 1995.
- [7] F. Tip and J. Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms", *Proc. of the 15th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, 2000.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison Wesley, 1994.
- [9] JHotDraw, <http://www.jhotdraw.org>.