

Inspecting Object-Oriented Code from the Behavioural Perspective

Neil Walkinshaw

Department of Computer and Information Sciences, The University of Strathclyde,
Livingstone Tower, 26 Richmond Street, Glasgow G1 1XH

`neil.walkinshaw@cis.strath.ac.uk`

Code inspection [1] is a technique that involves the manual scrutiny of source code documents with the aim of reducing the number of defects. It is commonly accepted that the size and complexity of software systems demands a ‘divide-and-conquer’ approach [2]. Successful object-oriented design inspections [3] involve dividing the system into its use-cases. Dividing object-oriented source code in a similar manner is however very challenging. A system’s behaviour (encapsulated in a use-case) is the result of a series of method invocations (messages) that are sent between objects. A consequence of the object-oriented decomposition strategy (decomposition of the system into data structures as opposed to functional decomposition) is that function and code are no longer coincidental. Functionally related code is distributed over many classes that are structurally related to each other via inheritance and composition. Paradigm features such as polymorphism and dynamic dispatch make it virtually impossible to determine the run-time sequence of method invocations that corresponds to the feature that is being inspected.

Various automated approaches have been developed that attempt to automatically locate source code that corresponds to a particular feature or use-case. Many of them are based on analysis of the call graph of a system (a graph that represents methods as nodes and possible calls as edges between them). Several approaches require dynamic information (information that is captured as the system is executed), e.g. work by Eisenbarth *et al.* [4] or Egyed [5]. This is impractical for software inspections because the system may not necessarily be executable (e.g. the system may be a framework) and it is difficult to determine a complete set of test cases that are representative of the feature to be inspected. Purely static approaches on the other hand are too conservative. The subset of possible method invocations returned is too large to be of use to the inspector (e.g. work by DiLucca *et al.* [6] and Qin *et al.* [7]).

I believe that by specifying a small set of key methods which must be executed for a given feature (referred to as “landmark methods”), the amount of code to be inspected can be significantly reduced. This would provide the inspector with a more focussed code base, ignoring methods that are irrelevant. This small amount of lightweight dynamic information allows for an analysis approach that, depending on the number of methods specified by the inspector, provides a useful compromise between a sound but conservative static analysis and a precise but unsound dynamic analysis.

A two-stage approach has been developed that takes as input the call graph of the system to be inspected and a set of landmark methods specified by the inspector. Making the assumption that the inspector is only interested in other methods that may influence or be influenced by the execution of these landmark methods, the approach uses the call graph and method dependence information to identify other relevant method calls. In the first stage direct paths on the call graph are identified between landmark methods. A set of direct paths between two nodes takes the form of a hammock graph [8,9] (which is a single-entry single-exit directed graph). In the second stage the approach uses code slicing to identify further call sites that may influence the execution of methods that have been identified in the first stage. Slicing [10] is a code extraction technique that highlights code that can affect the execution of a point in the source code (in this case a call site).

Initial results [11] have been computed for a set of sample use-cases in the JHotDraw drawing editor framework. Precision and recall values for queries with respect to different combinations of landmark methods indicate that although the approach can potentially produce accurate and useful results, its success depends heavily on the selection of adequate landmark methods. A large part of this work's evaluation is based on generating a set of guidelines that can be used to assist the inspector in the selection of useful landmark methods. These guidelines will finally be validated on a larger case study.

References

1. Fagan, M.: Design and code inspections to reduce errors in program development. *IBM Systems Journal* **15:3** (1976) 182–211
2. Parnas, D., Lawford, M.: The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering* **29** (2003) 674–676
3. Thelin, T., Runeson, P., Wohlin, C.: An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering* **29** (2003) 687–703
4. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. *IEEE Transactions on Software Engineering* **29** (2003) 210–224
5. Egyed, A.: A scenario-driven approach to traceability. *IEEE Transactions on Software Engineering* **29** (2003) 123–132
6. Lucca, G.A.D., Fasolino, A.R., Carlini, U.D.: Recovering use case models from object-oriented code: A thread based approach. In: *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*. (2000) 108–117
7. Qin, T., Zhang, L., Zhou, Z., Hao, D., Sun, J.: Discovering use cases from source code using the branch-reserving call graph. In: *Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)*. (2003) 60–67
8. Kasyanov, V.N.: Distinguishing hammocks in a directed graph. *Soviet Math. Doklady* **16** (1975) 448–450
9. Kasyanov, V.N., Evstigneev, V.A.: *Graph Theory for Programmers: Algorithms for Processing Trees*. Kluwer Academic (2000)
10. Weiser, M.: Program slicing. *IEEE Transactions on Software Engineering* **SE-10** (1984) 352–357

11. Walkinshaw, N., Roper, M., Wood, M.: Understanding object-oriented source code from the behavioural perspective. In: Proceedings of the International Workshop on Program Comprehension (IWPC'05), St. Louis, IEEE (2005)