Deprogramming Large Software Systems

Yohann Coppel and George Candea
School of Computer & Communication Sciences
EPFL (Lausanne, Switzerland)

Abstract

Developers turn ideas, designs and patterns into source code, then compile the source code into executables. Decompiling turns executables back into source code, and *deprogramming* turns code back into designs and patterns. In this paper we introduce DeP, a tool for deprogramming software systems. DeP abstracts code into a dependency graph and mines this graph for patterns. It also gives programmers visual means for manipulating the program. We describe DeP's use in several software engineering tasks: design pattern identification, source code refactoring, copy-paste detection, automated code documentation, and programmer fingerprint recognition.

1 Introduction

Programming is the process of turning concepts, patterns, and designs into source code; a compiler then turns the source code into executables. By analogy to decompiling—the reverting of executables back into source code—we define *deprogramming* as the reverting of source code back into concepts, patterns, and designs.

Such reverse processes are powerful tools for manipulating programs and systems. The fact that decompilers are not used much as an engineering tool, but mostly as a means for reverse-engineering, is mainly because those who need to maintain a software system typically have access to its source code. The same cannot be said of programming: the thoughts that programmers had when writing the code often become unavailable, either because the programmers leave the company, are forgetful, or simply because the code has evolved and now embodies a different set of patterns than originally intended.

Lack of access to the patterns and designs behind a body of code makes it difficult to maintain large code bases; having a global understanding of a system is often crucial to being able to transform it in a reliable and productive manner. When developers understand the software structure, they are better able to identify design problems and fix them, as well as to extend the software without introducing new problems.

Some common developer tasks that require good understanding of the system are code reviews and refactoring. During a code review, programmers will often identify code patterns that can lead, for instance, to poor performance or race conditions. They may also try to recognize the design patterns [10] used, in order to grasp what the code does. Without tool support and good code documentation, identifying such patterns can be time-consuming. Refactoring [9]—the restructuring of an existing body of code without changing its external behavior—is also common, especially when code bases increase or APIs change. Tools like Eclipse [8] provide good support for simple refactoring, like renaming variables or changing method signatures, but little in the way of conceptual transformations, such as replacing the use of an iterator pattern with a visitor pattern [10].

Some more exotic tasks include detecting copy-pasted code and recognizing programming "fingerprints." Copy-pasting code is widely practiced and very tempting, in order to save time; unfortunately, in the long run it leads to code that is hard to maintain and may even have legal implications, depending on which code base the code was copied from. Detecting copy-pasted code by searching for code snippets fails when layout or identifier names are modified. Plagiarism detection tools use code fingerprinting techniques to determine similarity [15]. We believe fingerprinting can be applied at higher semantic levels: a programmer often develops a certain style of programming, leading to a fingerprint that is found in all code (s)he produces. Recognizing such fingerprints automatically can help identify code by origin, with the purpose of rewriting it (e.g., to avoid licensing problems) or double-checking it (e.g., because the originator is suspected of having coded a backdoor into the system).

The difficulty of such maintenance operations, however, tends to increase substantially as the number of contributors and the size of the code base increases [3]. As the cost of maintenance goes up, the quality of maintenance tends to drop, leading to a deterioration of software quality. It is therefore necessary to develop tools that can aid programmers in rapidly understanding large bodies of software at a global level.

Appears in Proceedings of the 4th Workshop on Hot Topics in System Dependability (HotDep), December 2008

2 The Deprogramming Approach

In this paper, we describe preliminary steps toward DeP, a tool that helps developers deprogram software. Our approach aims to take static analysis to a higher conceptual level: whereas traditional static analyzers identify low-level patterns, like uninitialized variables and potential race conditions, in DeP we perform static analysis at the level of design patterns embodied within the code.

DeP operates on Java source code or bytecode and turns it into an intermediate representation. Of the different possible representations, we chose the dependency graph, as it is a highly expressive abstraction of the code. After statically extracting dependencies, DeP utilizes dynamic analysis to augment the graph with additional information. §3 describes dependency graphs in detail.

The graph is used for various types of static analysis, primarily pattern-matching to recognize design constructs and structures (see §4). DeP also provides custom views of the dependency graph, with the ability to visually manipulate the underlying source code (see §5).

Finally, DeP can generate new artifacts based on these analyses, such as code documentation (see §6).

3 Dependency Graphs

The dependency graph is a directed graph that represents relations (e.g., inheritance or membership) as edges between nodes representing Java classes, packages, interfaces, methods, and fields. Figure 1 shows a small fraction of the graph representation of the JBoss application server [11], a system with over 350 thousand lines of code (KLOC); we only show one type of edges.

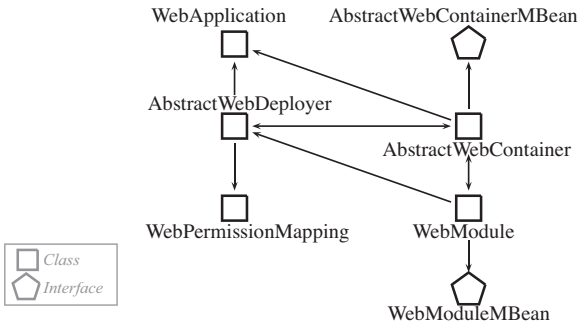

Figure 1: Part of the dependency graph for the org.jboss.web package in the JBoss application server.

We build the dependency graph of a system in two steps: one static, one dynamic. The static extraction focuses on the code's internal structure and identifies several types of dependencies between graph nodes: calls, read access, write access, extends, implements, method overload, method overwrite, anonymous type, inner type, member method, member type, member field, contained in file, contained in directory, contained in package, etc.

In the dynamic step, we use an improved version of AFPI [4] to obtain additional information for the graph: calls that are missed by the static step, member methods and member types, and, most importantly, error and exception propagation. AFPI uses fault injection to exercise and observe propagation paths, relying on dynamic stack traces to find inter-component dependencies; we added the ability to record the context in which faults occur. This dynamic analysis adds important information, such as indirect relationships arising from the use of reflection followed by invocations of the reflected API, which escapes static analysis.

The two steps generate two different graphs, which DeP needs to merge into a single representation. There is usually an exact overlap between the two graphs in terms of nodes, we just need to merge nodes that represent the same data structure. The AFPI-based graph has a new type of edges—error propagation—which can simply be added to the statically-obtained graph. For call, class, and exception handling edges, we take the union of edges found in the two graphs, while the member method and member type edges are equivalent to each other. Edges of different types between two identical nodes become parallel edges in the final graph.

The final dependency graph includes a large fraction of the code's constructs, but omits implementation details, such as variable names or sequencing of different calls. While such details are crucial to the proper functioning of the program, they are generally unnecessary for understanding the structure of the code. Removing these details helps programmers (and DeP) to focus only on those elements that matters to the analysis. The graph is essentially an abstraction of the underlying source code.

Of course, the choice of which relations to capture in the graph and which ones to ignore can make a significant difference in the quality of the subsequent analyses. In our current version of DeP, deprogramming focuses on matching design patterns, and these are usually described in terms of nodes and their inter-dependencies. Therefore, the type of representation described above is well suited. The more types of dependencies are included in the graph, the richer the kinds of patterns that can be searched for. As will become evident in the next section, extra edges (relationships) between a given pair of nodes do not slow down the pattern matcher.

An important advantage of the two-step approach to building the dependency graph is that it is entirely automatic and can be easily repeated whenever the code changes. This allows the abstract representation of the program to closely track the evolution of its source code.

4 Finding Design Patterns

We define graph patterns using a small, domain-specific language. There can be three types of expressions:

- variable definitions of node/edge types to match (class, method, etc.)
- value definitions, i.e., constraints on each variable to be used during matching, and
- specification of edges connecting the above nodes.

Consider a pattern, called "sibling envy," in which a method, said to be "envious," calls a static method in another branch of the inheritance hierarchy, as shown in Figure 2. This is usually bad coding style, because class inheritance layers are violated. One fix is to make the common ancestor class provide this method to all its subclasses, or to introduce another layer that provides the method. This way, new classes introduced in the hierarchy are aware of the method and can use it accordingly. DeP can be used to find instances of such bad coding and fix them.

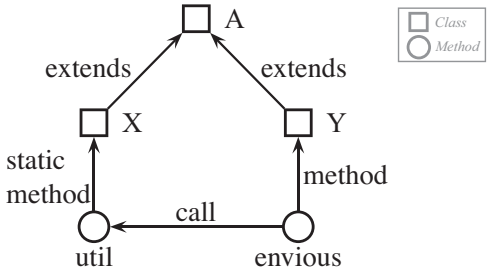

Figure 2: Graph representation of "sibling envy."

The pattern would be specified in DeP as follows:

```
def Class = elements.TypeElement
def Call = Relation.CALL

val A,X,Y : Node = Class
val util,envious : Node = Method
val call  : Edge = Call

X--[static_method]-->util
X--[extends]-->A
Y--[extends]-->A
Y--[method]-->envious
envious--[call]-->util
```

Using DeP to search for this pattern in an application might find a code snippet like the one in Figure 3. Finding all instances of sibling envy can give a programmer a quick way to fix all such problems in a large code base. A better way to write this code would be to place the *checkIfValid()* method in the *SimpleBrowser* class, since it is to be used in multiple subclasses.

```
class SimpleBrowser {}
class CompliantBrowser extends SimpleBrowser {
  static boolean checkIfValid(String url) {
    //...
  }
}
class MobileBrowser extends SimpleBrowser {
  private void openUrl(String url) {
    if (!CompliantBrowser.checkIfValid(url))
      //...
  }
}
```

Figure 3: Sample code illustrating "sibling envy."

Finding design patterns in a dependency graph is similar to the subgraph isomorphism problem [17]: we need to find a correspondence between nodes and edges of a graph (the pattern) and nodes and edges of an unspecified subgraph in another graph (the dependency graph).

There exist various algorithms to find subgraph isomorphisms, and we chose VF2 [6]. It is based on a depth-first search strategy, with a set of efficient pruning rules to restrict the search to a set of feasible solutions. In other words, any time a node is added to the solution, if the new solution does not satisfy a set of feasability rules, the search tree is pruned immediately. Such aggressive pruning ensures the search is efficient in large graphs, while preserving completeness.

DeP's pattern-matching is a slight variation of the isomorphism problem, in that we can accept a solution if an edge between two nodes is present in the dependency graph, but not in the pattern. Therefore, in our implementation of the VF2 algorithm, we removed two pruning rules. Also, in the original isomorphism problem, nodes and edges are not typed, whereas in DeP nodes and edges often have different types. To take advantage of this specificity, we added two early matching rules to the algorithm: (a) during the search for a node to be included in the solution, we immediately eliminate any node from the graph that does not match the node in the pattern definition; this restriction helps reduce considerably the scope of the search; and (b) same restriction applied to edges. In the VF2 algorithm, when the set of pair candidates to be included in the mapping is computed, none of the pairs in this set include a non-matching node.

The modified matching algorithm is complete, i.e., it finds all instances of the specified pattern; in other words, there are no false negatives. We can get, however, false positives. In practice, we found the rate of false positives to be low—in all our tests on real systems (JBoss, JEdit,

Azureus, and Jython) we encountered a maximum false positive rate of 4%. All false positives resulted from the pattern definition, and we were able to eliminate all false positives by suitably respecifying the pattern.

Although the graph isomorphism problem is NP-complete, our implementation performs well for large dependency graphs. For example, completing a search for the 5-node "sibling envy" pattern in JBoss (>350 KLOC) takes on the order of minutes on a regular workstation. There are plenty of opportunities left for optimization, especially given that the search is parallelizable, thus being a good fit for multi-core architectures.

Figure 4 shows a screenshot of DeP's pattern matching dialog window. In the upper part, users gives the pattern definition using the language described earlier; it is possible to define several patterns at a time, all of which are to be searched for. Then the scope of the search is specified (either the entire graph, or a subgraph opened in another window). The number of solutions can be limited to some upper bound; 0 indicates "all." Search results are shown in the lower portion, as they are found. By clicking on them, the user can open a view containing the matched subgraph.

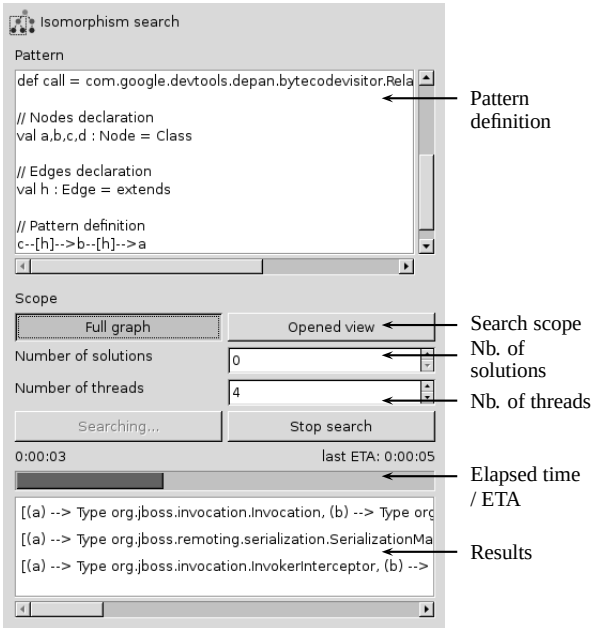

Figure 4: The pattern definition dialog in DeP.

DeP provides a library of well-known design patterns that the programmer can reuse directly or to define custom patterns. Patterns still have to be specified in the language described above, but it is straightforward to provide a graphical interface to make pattern construction and manipulation more intuitive. One can also imagine a command-line interface that could be used by automated commit hooks in source control systems to reject new code that has bad coding patterns. Programmers doing code reviews can add such patterns to a pattern database, enabling other programmers to check their code for these patterns before submitting for review.

5 Refactoring

Refactoring [9] is a frequent task in the lifetime of large software projects. Although refactoring is well known and understood, it is often difficult to safely perform structural changes in production code, and decisions on what to refactor require global knowledge of the system. Techniques for automatically finding candidate code for refactoring is still an open area of research [13, 16], as current approaches yield high rates of false positives.

DeP helps developers find and judge refactoring targets in two ways: First, it can be used to search for good and bad patterns (§4). Second, DeP helps developers understand the code and its interactions at a higher level of cognition, thus aiding in more quickly deciding whether a refactoring target should be pursued or not. Figure 5 shows a screenshot of DeP being used for refactoring.

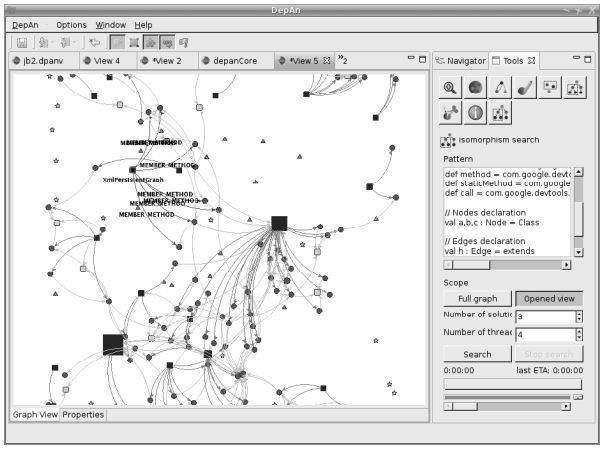

Figure 5: DeP used in refactoring DepAn [5]: a searched pattern is highlighted by indicating the node names.

A typical way of using DeP is to select a set of design patterns to be searched for; results can be visually inspected, and the programmer can decide whether to pursue refactoring or not. Nodes and edges can be dragged around to perform what-if analyses. Once the programmer has decided how to refactor, the source code is modified. We used DeP to refactor DepAn [5], a Google tool into which we are currently incorporating DeP; we found several design flaws, corrected them, and submitted patches that were accepted into the development tree.

6 More Deprogramming Tasks

DeP enables developers to see programs as aggregations of code patterns, making a number of additional deprogramming tasks possible. Here we describe three such tasks; their implementation in DeP is still pending.

Copy-Paste Detection: Despite all warnings, programmers often use copy-paste when they perceive a refactoring as too expensive. Copy-paste detection tools are used most frequently to detect illegal copying of source code [1, 7, 14]. These tools use approximation methods to handle obfuscation of the copy-paste, such as identifier renaming or inversion of independent statements.

Deprogramming can complement existing tools by helping detect copy-paste at higher semantic levels. We rely on the notion of a "pattern fingerprint" of a source code artifact: we conjecture that each program has an identifying fingerprint determined by the patterns it contains and the ways in which these patterns are combined, just like minutiae in a human fingerprint.

If this conjecture holds, it then becomes possible to use DeP to check whether code from program *A* was copy-pasted into program *B*: deprogram the two programs in question, compare their dependency graphs to find code portions that have identical structure, and then inspect those code areas. Alternatively, one might generate a "pattern profile" of *A* (i.e., a list of all the patterns that appear in *A*) and compare it to *B*'s profile. The pattern profiles can be augmented with statistics, such as how many times a particular pattern is used. Pattern profiles can also be compared hierarchically: does a subset of *B* contain the same number and types of patterns as some subset of *A*?

In some sense, DeP distills the essence of code, allowing the essences of two pieces of code to be compared. However, the more we distill and abstract two programs, the higher the likelihood that they will match. There is therefore a tradeoff, which we have yet to study.

Design Documentation: Good programmers are expected to document their code, such that other programmers can better understand it and make use of it. Tools have been developed to ease documentation of APIs, such as Doxygen [18] and Javadoc [2]. These rely on annotations in the code, and good programmers are expected to make liberal use of these annotations. The challenge with both internal documentation and annotations is that humans must keep them synchronized with the actual code. History and experience suggests this is a battle that cannot be won without better tool support.

We believe DeP can be used to automatically document certain aspects of the code. Good code documentation usually contains the general idea behind the code, the role of each components, expected interactions, etc. Therefore, design patterns are an important part of code documentation and design specifications.

Deprogramming can be used to turn the source code into a dependency graph and identify all known patterns, good or bad, that appear in the code. Thinking of the code as an aggregate of patterns, we can automatically generate text documentation based on the dependency graph. For example, if Model-View-Controller patterns are found, the documentation could read "Classes *A*, *B*, and *C* are part of a MVC pattern: *A* is the model, *B* the view, and *C* the controller." Design patterns can also be turned into UML diagrams. Such analyses may be particularly useful after the code has undergone major restructuring or after a large volume of contributions have been incorporated in the code base. DeP can also serve as a learning tool for new employees, who need to understand the design of a large software product.

Another potential use is to extract relatively formal design specifications from the code and verify them against the prescribed design that was handed by architects to the programmers. Such design specifications can also serve as models for proving properties about the code.

Programmer Fingerprint Recognition: Identifying the author of a body of work is always a challenging detective's task. For instance, detecting forged paintings or identifying paintings by the same artist has already entered the computational domain: fake paintings can be detected by software using features extracted from a database of digital representations of paintings [12].

We believe a similar approach might be applicable to dependency graphs: a future version of DeP could use multiple programs written by the same person to generate a fingerprint of the author's style. Unlike copy-paste detection, where we aimed to capture a specific fingerprint of an individual program, here we would use multiple programs to find identifying features of a programmer's style: preferences for certain patterns or combinations thereof, specific approaches to modularization, etc. Assuming that every programmer has a unique fingerprint, extracting it and comparing it to a database of fingerprints might serve in identifying the author(s) of a program. Such uses of DeP could be used when authorship is unknown or is the subject of a dispute.

Whether this is possible or not is still an open question—unlike painting, programming is considerably more canonical, thus limiting the programmer's freedom of expression. Yet, the constraints of rhyme, meter, and form in poetry do not seem to have limited the freedom that makes so many poets easily recognizable from their verses. Programs, though, often have many authors, each of whom leaves their fingerprint on the code.

7 Conclusion

In this paper we described DeP, a deprogramming tool that helps view and analyze programs as combinations of code patterns; it abstracts code into a dependency graph and mines this graph for patterns. We presented ideas on how DeP can be used in several software engineering tasks: design pattern identification, source code refactoring, copy-paste detection, automated code documentation, and programmer fingerprint recognition. We believe DeP's results can be used by other tools commonly employed in managing code bases (e.g., source code repositories) or that perform optimizations or suggest refactoring targets. We used the current DeP prototype to understand previously-unknown programs, refactor large code bases, and find common design patterns as well as patterns of bad coding.

8 Acknowledgments

We would like to thank Lee Carver from Google, who gave us valuable feedback in early stages of this work, and the anonymous reviewers, who greatly helped us improve this paper.

References

[1] Moss: A system for detecting software plagiarism (unpublished). http://www.cs.berkeley.edu/ aiken/-moss.html.

[2] G. Aitken. Automatically generating Java documentation: javadoc and the doc comment. *Dr. Dobb's Journal of Software Tools*, 21(7), July 1996.

[3] B. Boehm. *Software engineering economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[4] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for software systems. In *IEEE Workshop on Internet Applications*, 2003.

[5] L. Carver and Y. Coppel. DepAn: A direct manipulation tool for visualization, analysis, and refactoring of dependencies in large applications. http://code.google.com/p/google-depan.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(10), 2004.

[7] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato. A plagiarism detection system. In *Proc. 12th SIGCSE Technical Symposium on Computer Science Education*, 1981.

[8] Eclipse. http://www.eclipse.org, 2008.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[11] JBoss. http://jboss.org.

[12] J. Li and J. Z. Wang. Studying digital imagery of ancient paintings by mixtures of stochastic models. *IEEE Transactions on Image Processing*, 13(3), 2004.

[13] H. Melton and E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *Proc. 29th Australasian Computer Science Conference*, 2006.

[14] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Universal Computer Science*, 8(11), November 2002.

[15] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. 2003 ACM International Conference on Management of Data*, 2003.

[16] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. 7th European Conference on Software Maintenance and Reengineering*, 2003.

[17] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1), 1976.

[18] D. van Heesch. Doxygen manual. http://www.doxygen.org, 2001.