Reverse Engineering of Object Oriented Code

Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
tonella@itc.it

ABSTRACT

During software evolution, programmers devote most of their effort to the understanding of the structure and behavior of the system. For Object-Oriented code, this might be particularly hard, when multiple, scattered objects contribute to the same function. Design views offer an invaluable help, but they are often not aligned with the code, when they are not missing at all.

This tutorial describes some of the most advanced techniques that can be employed to reverse engineer several design views from the source code. The recovered diagrams, represented in UML (Unified Modeling Language), include class, object, interaction (collaboration and sequence), state and package diagrams. A unifying static code analysis framework used by most of the involved algorithms is presented at the beginning of the tutorial. A single running example is referred all over the presentation. Trade-offs (e.g., static vs. dynamic analysis), limitations and expected benefits are also discussed.

Categories and Subject Descriptors:

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement–*Restructuring, reverse engineering, and reengineering.*

General Terms: Design.

Keywords: Diagram recovery, object oriented programming, static code analysis.

1. INTRODUCTION

Software evolution accounts for the vast majority of a program's life cycle. This phase of the software process is aimed at adding functions, correcting defects, adapting the code to a new environment or improving the internal structure. The main activities involved in such tasks, and conducted in response to a change request, are: (1) Program understanding, (2) Change location, (3) Impact analysis, (4) Change implementation, (5) Regression testing.

Reverse engineering aims at recovering design views from the source code, to offer programmers a faithful, high level representation of the program that is ensured to be consistent with the actual implementation. Reverse engineering can support the activities 1-3, by providing summary information on the main program's entities and relationships.

This tutorial deals with the main problems that are encountered when Object-Oriented code is reverse engineered. The most important UML (Unified Modeling Language) diagrams are taken into account to offer a wide overview of the techniques that can be employed to recover them from the code. The extracted views include the class diagram, the object diagram, the collaboration and sequence diagrams, the state diagrams, and the package diagram. In fact, no single diagram is able to summarize all information about an Object-Oriented system and multiple perspectives have to be adopted to cope with the different involved aspects.

When using reverse engineering techniques, it is important to be aware of some trade-offs that possibly affect the accuracy and safety of the recovered information. Examples considered in the tutorial are the choice between static and dynamic analysis, and the level of object sensitivity. As regards the presentation of the recovered information, usability issues must be also considered. Some of the available visualization options are described in the tutorial.

2. CODE ANALYSIS FRAMEWORK

Most of the static code analyses employed by the reverse engineering techniques presented in the tutorial are based on the same, unifying framework. This framework consists of a graph representation of a program, called the Object Flow Graph (OFG), which is focused on the tracing of the flow of information about objects from the object creation by allocation statements, through object assignment to variables, up until the storage of objects in class fields or their usage in method invocations.

The concrete Java syntax of the program under analysis is turned into an abstract language, for the representation of the data flows, independently of the specific control flow structure. Statements affecting the control flow are dropped, while statements modifying the data flows are maintained and translated. Conversion from the abstract language to the OFG representation is straightforward. Nodes in the OFG represent program locations, while edges represent data flows. In the construction of the OFG, program locations can be associated either with the classes (object insensitivity) or with the objects (object sensitivity) they belong to. The tutorial discusses such a trade-off.

3. CLASS DIAGRAM

The most important and most widely used structural view of a software system is the class diagram. It shows the most relevant features (attributes and methods) of the core classes and their mutual relationships.

The main problem with the recovery of the class diagram from the code is in the way relationships are inferred. A basic algorithm can be defined, taking into account the declared types. Thus, an association/aggregation is inferred when the declared type of an attribute is another class, while a dependency (a weaker relation) is inferred when the declared type is that of a local variable or a method parameter. Inheritance is inferred directly from the syntax.

However, the basic algorithm for the recovery of the inter-class relationships suffer two main limitations: (1) the actual type may be different from the declared type; (2) in the presence of weakly typed containers no type is declared at all for the contained objects. Flow propagation in the OFG can be used to tackle both problems.

4. OBJECT DIAGRAM

The object diagram shows the set of objects created by a given program and the relationships holding among them.

A flow propagation in the OFG can be exploited to reverse engineer information about the objects allocated in a program and the inter-object relationships mediated by the object attributes. The allocation points in the code are used to approximate the set of objects created by a program, while the results accumulated in the OFG nodes after propagation are used to determine the inter-object relationships.

An alternative technique that can be used to produce the object diagram is based on the execution of the program on a set of test cases. Each test case is associated with an object diagram depicting the objects and the relationships that are instantiated when the test case is run. The diagram can be obtained as a post-processing of the program traces generated during each execution.

The tutorial discusses the pros and cons of the two approaches. The static technique is safe with respect to the objects and relationships it represents, but it cannot provide precise information on the actual multiplicity of the allocated objects nor on the actual layout of the relationships associated with the allocated objects.

5. INTERACTION DIAGRAMS

Interaction diagrams augment the object diagrams with information about the messages that are exchanged among the objects over time. Construction of the interaction diagrams require the ability to resolve the method calls into the target objects. This can be achieved statically, by means of the flow information determined at the OFG nodes, or dynamically, by tracing the actual method calls. In the presence of incomplete systems, the output of the static method remains safe (i.e., valid for any possible execution) only if external data flows are properly modeled.

Given an Object-Oriented system under analysis, it makes no sense to produce one overall interaction diagram that describes all possible computations, since this is likely to exceed the cognitive abilities of human beings for any non trivial program. A solution is to apply focusing, by restricting the message exchanges being considered to those triggered by a computation of interest.

6. STATE DIAGRAMS

State diagrams show the possible states an object can be in and the transitions from state to state, as triggered by the method invocations received by the object.

Reverse engineering of the state diagrams from the code is a difficult task, that cannot be fully automated. However, it is possible to *partially* automate it by means of abstract interpretation. Class attributes are associated with symbolic, abstract values. Similarly, the effect that statements may have on such abstract values is represented in the form of their abstract semantics, by providing an abstract interpretation table for them. Recovery of the state diagrams is then achieved by running an abstract interpretation of the constructors, to determine the initial states, and of the methods, to determine the possible state transitions.

7. PACKAGE DIAGRAM

Packages are a general grouping mechanism that can be used to decompose a given system into components (and sub-components) that are relatively independent of each other. Key to a good decomposition of a system into packages is the definition of highly cohesive and loosely coupled modularization units. Automated recovery of a package structure for a given program might be interesting in at least three cases: (1) when a flat sequence of classes is to be organized into packages (no pre-existing package structure); (2) when the existing package structure is known to be inadequate; (3) when the existing package structure is being assessed against alternative ones.

Reverse engineering of the package diagram is based on the discovery of similarities among entities (classes), to be grouped together, and on the minimization of the relationships that cross the boundaries of the packages (low coupling). This tutorial describes how two widely used methods, *clustering* and *concept analysis*, can be adopted to determine heuristic solutions to this problem.

8. CONCLUSIONS

In the last part of the tutorial, some software evolution scenarios that could possibly benefit from the reverse engineered diagrams are presented, with reference to the running example. The architecture of a tool that implements the described reverse engineering techniques is considered and the presenter's experience in the reverse engineering of some large C++ systems developed at CERN (Conseil Européen pour la Recherche Nucléaire) is briefly reported.

Finally, some perspectives on the future role of reverse engineering in software engineering are given. Modern programming languages make the source code increasingly expressive, for example by supporting annotations and reflection. This opens to the possibility of recovering extremely meaningful and informative views from the source code. Agile development processes (such as XP, Extreme Programming), that are centered around the source code (*"the source code is the design"* is one of XP's guiding principles), can be integrated with reverse engineering in a very natural way.

9. REFERENCES

[1] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code.* Springer-Verlag, Berlin, Heidelberg, New York, 2005.