Extracting Generally Applicable Patterns from Object-Oriented Programs for the Purpose of Test Case Creation

Richard Torkar, Robert Feldt, and Tony Gorschek
Blekinge Institute of Technology,
School of Engineering,
372 25 Ronneby, Sweden
{rto|rfd|tgo}@bth.se

Abstract. This paper presents an experiment performed on three large open source applications. The applications were instrumented automatically with a total of 10,494 instrumentation points. The purpose of the instrumentation was to collect and store data during the execution of each application that later could be analyzed off-line. Data analysis, on the collected data, allowed for the creation of test cases (test data, test fixtures and test evaluators) in addition to finding object message patterns for object-oriented software.

1 Introduction

In order to automate software quality assurance to a high(er) extent, one needs to look at techniques that are suitable for this purpose. In this respect random testing is a likely candidate for automation due to its nature where a minimum of human intervention is needed [1]. Unfortunately, random testing of object-oriented software has not been researched widely (for an overview please see [2]). One of the reasons for this is surely the fact that random testing traditionally compares well to other techniques when it comes to dealing with scalar (pre-defined or primitive) types, while usually is seen to have weaknesses dealing with compound (programmer-defined or higher-level) types. Another reason might be that random testing, again traditionally, has been used on small and delimited example.

This paper aims to:

1. Show how object message pattern analysis, from automatically instrumented applications, can be used to automatically create test cases. (Test cases in this context equal test data, evaluators and fixtures.)
2. Point to how random testing can be performed on these test cases.
3. Examine the possible existence of object message patterns in object-oriented software in addition to the question of general applicability of said patterns.
4. Point out current possibilities for using instrumentation of software items as an underlying foundation to automated software testing.

Please use the following format when citing this chapter:

Torkar, R., Feldt, R. and Gorschek, T., 2008, in IFIP International Federation for Information Processing, Volume 275; *Open Source Development, Communities and Quality*; Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, Giancarlo Succi; (Boston: Springer), pp. 281–287.

Throughout this paper an empirically descriptive model is used, i.e. explaining a phenomenon sufficiently on 'real-life' software. Next, related work is presented. Following that, the setup of the experiment is presented (Sect. 2) while the results are covered in Sect. 3. Finally, this paper ends with conclusions and future work (Sect. 4).

1.1 Related Work

Related work for this paper can be divided mainly into four categories: random testing, type invariants, dynamic analysis and patterns in object-oriented software.

First, random testing [1], which acts a basis for our approach, is a fairly old research area which has seen several new contributions lately which directly or indirectly affect this paper.

Second, likely invariants has gained considerable attention following the work of Pacheco et al. [3] in addition to Yang et al. [4]. The concept of likely invariants is built upon the assumption that one can find *different* input values, for testing a software item, by looking at the *actual* values the software uses during a test run. Our approach does not focus on likely invariants per see but rather class interactions (not component interfaces [5]), which in its turn provides the order of the methods being executed as well as method signatures and actual input and return values and types.

Third, the concept of dynamic analysis, i.e. analysis of the software based on its execution, can in our approach best be described as a see-all-hear-all strategy. This way it resembles the omniscient debugger [6] that takes snapshots of every change in the state of the running application thus allowing the developer to move backwards and forwards when debugging the application. The difference in our approach is that every important piece of information during execution, is stored for later off-line analysis.

Finally, an overview of object-oriented software testing, and especially patterns in this area, can be found in e.g. [7]. Important to note, in this circumstance, is that the word pattern (as used by almost all of the references) has in many ways a different meaning compared to how it is used in this paper.

In order to clarify the concept of patterns, we use the name *Object Message Patterns* for our purposes. *Object* stands for the fact that the focus is set on object-oriented software. *Message* is short for the message-driven perspective as employed by object-oriented software and finally, *patterns* stands for the execution traces as found when analyzing software.

2 Experimental Setup

In this experiment the focus is set around testing intermediate representations of source code from open source software. Today, in industry, many would say that the centre of attention is mostly around the Java Virtual Machine and the Common

Language Infrastructure (CLI) with its Common Language Runtime (both inheriting from the original ideas brought forward by the UCSD P-System's developers in the late 70's [8]).

The experiment was performed on three different open source software items: Banshee (an open source media player), Beagle (an open source search tool) and the Mono C# compiler, Mcs (an open source implementation of the ECMA-334/335 standard). The selection of the software items was performed with the following in mind:

- The application should be written in a language, which can be compiled to an intermediate representation (in this case the Common Intermediate Language).
- The application should be sufficiently large and thus provide a large amount of data for later analysis.
- Separate development teams should have developed the applications.

In the end, Banshee, Beagle and Mcs, were considered to fit the profile for the case study. For each application, one use case was selected which would be executed after the application had been instrumented:

- *Banshee:* Start application, select media file, play media file, stop playback, shut down application.
- *Beagle:* Start search daemon in console (background process), perform query in console, close the GUI which presents the search results, stop search daemon.
- *Mcs:* Compile a traditional 'Hello World' application.

Table 1. An overview of the experiment showing the number of LOC (lines including comments, white spaces, declarations and macro calls), the size of the assemblies (in KB), the number of classes that were instrumented and the number of instrumentation points (IP), for each application. In addition the time to instrument (TTI) and execute (TTE) with and without instrumentation for each application is presented (in seconds).

App.	LOC	IL (KB)	#Classes	#IP	TTI	TTE w. instr.	TTE w/o instr.
Banshee	53,038	609	414	2,770	28	424	63
Beagle	146,021	1,268	1,045	5,084	71	74	6
Msc	56,196	816	585	2,640	166	34	0.8

After the selection of the candidate applications was accomplished the actual instrumentation took place (see Table 1 for some descriptive statistics). To begin with, each valid class in the Software Under Test (SUT), disregarding abstract, extern and interface annotated signatures, in every assembly (exe and dlls), had instructions inserted in each method which would collect runtime input and return value(s) and type(s), as well as the caller (i.e. what invoked the method). All this data, together with a time stamp, was then stored during runtime in an object

database while the SUT was executed following one of the before mentioned use cases. That is to say, each time a method was executed, during the execution of a use case, an object containing all the values necessary to recreate that state (with its connections), was stored in the object database (i.e. a form of deep copy was saved). Having to serialize the data beforehand would be too resource intensive for obvious reasons not to mention posing some difficulties from a technical perspective.

Next, the content of the object database was analyzed looking for patterns and discovering likely critical regions. The selected paths could then be used for creating a test case (using the actual runtime values as test data and test evaluators). The execution of the use cases, as well as the instrumentation of the assemblies, was performed on Linux 2.6.15, Mac OS X 10.4.4 and Windows XP SP2 using Cecil 0.3 (open source assembly manipulator), AspectDNG 0.47 (open source tool to support aspect oriented programming) and the open source (in-memory) object database engine db4o 5.2.

3 Results

The intention of performing Object Message Pattern Analysis (OMPA) on data in this experiment was to find and generalize patterns that later could be used for testing the SUT. In addition to this the hypothesis is that patterns, if found, could be generally applicable to most, if not all, object-oriented software. Since the analysis was performed manually a limitation on the number of analyzed objects was needed. Thus, 300 objects (in a call sequence) from each application were analyzed from an arbitrary point in the object database (selected by a pseudo-random generator as found on pp. 283–284 in [9]). In the end, eight object message patterns were found during the analysis of the data stored in the object database (Tables 2–3, respectively).

The patterns found are of two different categories. Four patterns belong to, what has by us been defined as object unit patterns. Object unit patterns constitute of a sequence of method invocations on *one* object, i.e. methods in an object has been executed in a certain order. Object trace patterns, on the other hand, are slightly different. They cover the path of execution through *more than* one object.

Object Unit Patterns. Four object unit patterns were found during the OMPA (Table 2); these patterns exercised only one object consistently over time and were found in all three applications (needless to say, the names of the objects and classes differed in all three applications, but the pattern can nevertheless generally be applied on all three applications).

The first pattern, the Vault pattern (Table 2), is a straightforward pattern which is executed by first invoking a constructor then invoking a setter and, finally, multiple times, a getter (before a destructor is invoked). This can be seen as a very rudimentary pattern for storing data in an object which then is fetched by one or many objects, and as such is suitable to always execute in a testing scenario, i.e. data is stored in a simple vault. During the analysis the intermediate representation was

used for examining if a method was defined as a getter or setter (property) by searching for the keyword `.property`. There is of course a possibility that a method is *acting* as getter or setter while not being defined as such, but in this analysis these types of methods are disregarded and an emphasize is put on the proper definition of a getter or setter according to the CIL.

Next, the Storage pattern is an evolved Vault pattern and the combinations of setter and getter invocations can be many (Table 2). Hence, the Storage pattern can be constructed in different ways and a combinatorial approach might be suitable in a testing scenario (compared to the Vault pattern which is very straightforward), i.e. data is stored in storage and the storage has (many) different ways of adding or offering content. The reason for distinguishing between Vault and Storage is that a Vault was common during OMPA and as such should always be used when testing object-oriented software, while Storage, on the other hand, is a more complex pattern (more steps performed) and as such needs additional analysis.

The Worker pattern at first glance looks like bad design. An object gets instantiated, and immediately filled with data. A method is next invoked which manipulates the data, returns the manipulated data and, finally, a destructor is invoked. The reason for this design might be to make sure that the method's implementation is accessible by different objects (extended) since it is declared `public`. If one had opted for a method declared as `private` or even `protected`, which could be invoked when the getter is invoked, then there would be no simple way to reuse the implementation.

Finally, the Cohesion pattern is a pattern that executes one or more methods in one object. It does this without *a priori* setting any values and the order of executing the methods is not always important, i.e. each and every method was found to be (by analyzing objects in the object database) an atomic unit with no dependency on other methods in the class and as such the word cohesion (united whole) was found to be appropriate to use.

Table 2. Different object unit patterns found in all three applications. The first column shows the name selected for the pattern and the second column the actual pattern. Abbreviations used: ctor and ~ctor is short for constructor and destructor respectively, while setter and getter is a method which sets or gets data stored in the object.

Name	Pattern
Vault	ctor → setter → 1…*n* getter → □ctor
Storage	ctor → setter → 1…*n* getter → 1…*n* setter → … → □ctor
Worker	ctor → setter → method invocation → □ctor
Cohesion	ctor → 1…*n* method invocation → □ctor

Object Trace Patterns. Looking at the object trace patterns, one can see four patterns that can be generally applicable (see Table 3 on next page); these patterns exercise several objects and constitutes sequences of object:method invocations.

A fairly common pattern that seems to come up on a regular basis is the Cascading pattern. This pattern instantiates object after object, which all can be of different or same types. The approach seems to be quite common when object-oriented applications are starting up, but in addition shows up in several phases of an application's lifetime (from start up to shutdown).

Next, the Storing pattern and the Fetching pattern showed up many times as well. These patterns are directly connected to the object unit Storage and Vault patterns, and as such can be combined in many ways.

The final pattern is the Dispatch pattern. The Dispatch pattern simply invokes one method after another (not a constructor though). In most cases the Dispatch patterns ends up with executing the Storing or Fetching pattern as a final step.

Table 3. Different object trace patterns found in all three applications. Abbreviations used: ctor and ~ctor is short for constructor and destructor respectively, while setter and getter is a method which sets or gets data stored in the object. An alphabetical character in front of the abbreviation, i.e. A:ctor, indicates that a type A object's constructor is invoked.

Name	Pattern
Cascading	A:ctor → B:ctor → C:ctor → …
Storing	A:ctor → B:ctor; A:method → B:setter
Fetching	A:method → B:getter
Dispatch	A:method → B:method → C:method → …

3.1 Test Case Creation

Applying test case creation on the example (and on data in the experiment) is fairly straightforward when all entities needed can be found in the object database, i.e. the following data is available: methods' name, input type and values, and return type(s) and value(s).

In addition, information regarding the caller can be extracted from the object database by simply examining the caller value in the current method being executed (a value stored in the database) or by examining the time stamp for each stored entity in the database.

The above information provides us with an opportunity to automatically create test cases and, in the end, have a simple smoke test or regression test mechanism, depending on the aim of our testing efforts. In short, the software item is executed, test cases are created from the database, patterns are extracted from the database and the SUT is validated. Obviously, the end goal is to make this a fully automatic process.

4 Conclusions and Future Work

This paper presented eight software object message patterns, for testing object-oriented software. As such it indicates in our opinion that the first steps have been taken on the road to extract generally applicable object message patterns for the purpose of testing object-oriented software. In this case study, the patterns are accompanied with automatically generated test cases whose entities (test data, test fixture and test oracle) are retrieved from a database which stores runtime values that are collected when executing a use case on the SUT. These test cases are then stored and used as a simple regression testing mechanism, hence avoiding manual, error-prone and tedious test case creation.

In the future a replication of this study should be made were the focus would be put on large applications and extensive data collection (i.e. collecting more or all use cases during the system testing phase). In addition, a number of issues with respect to optimization must be solved and the analysis must be performed fully automatically for this approach to be beneficial.

References

1. Hamlet, D.: Random testing. In: Marciniak, J.J. (ed.): Encyclopedia of software engineering. John Wiley & Sons, New York (1994) 970-978
2. Binder, R.V.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc. (1999)
3. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-Directed Random Test Generation. Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society (2007)
4. Yang, J., Evans, D.: Dynamically inferring temporal properties. Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, Washington DC, USA (2004)
5. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. SIGSOFT Softw. Eng. Notes **27** (2002) 218-228
6. Lewis, B., Ducasse, M.: Using events to debug Java programs backwards in time. Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, Anaheim, CA, USA (2003)
7. Lange, M.: It's Testing Time! Proceedings of the 6th European Conference on Pattern Languages of Programming and Computing, Irsee, Germany, UVK Universitatsverlag Konstanz GmbH (2001)
8. Institute for Information Systems, UCSD, University of California: UCSD PASCAL System II.0 User's Manual. (1979)
9. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: Numerical recipes in C: the art of scientific computing. Cambridge University Press (1988)