Refactoring of Assertions in Design by Contract

D. Rodríguez[1], M. Satpathy[1], J. Covas[1], J.J.Cuadrado[2]

[1]Dept. of Computer Science
The University of Reading
PO Box 225, Whiteknights
Reading RG6 6AY
United Kingdom

Dept. of Computer Science
University of Alcalá
Ctra. Barcelona km 33.6 - 28871
Alcalá de Henares, Madrid
Spain

drg@ieee.org

Abstract

Assertions are formal constraints over the state variables of a source program which are inserted as annotations in the program text. When some code has been annotated with assertions and is then subjected to refactoring, original assertions would no longer be consistent with the refactored code. The main focus of this paper is to specify how assertions could be made consistent across the refactoring process considering design-by-contract.

1 Introduction

Maintenance has become the most expensive part of the the software development life-cycle. As a result programming languages and Integrated Development Environments (IDEs) have evolved in order to facilitate this process. Tools in this evolution include assertions, design by contract and refactoring.

Assertions [5, 9] are formal constraints on software system behaviour which are inserted as annotations in a source program text. Assertions as Boolean expressions evaluate to *true* when the rogram state satisfies the desired constraints. If an assertion evaluates to *false* then it means that the program has entered into an inconsistent state and hereafter the program behaviour cannot be relied upon even if the output was correct. Assertions are widely used in the software industry today, primarily to detect, diagnose and classify programming errors during testing.

Refactoring, generally described as patterns, help developers to rebuild the structure of a source code in order to improve or clean it, to make its maintenance easier and quicker. In this work, we define a set of steps and rules in order to adapt the assertions during the refactoring process. Since there is not a complete definition of all kind of refactorings (and this is impossible, because new kind refactorings are being continuously developing to solve new problems) the goal of this project is not to define a set of rules for each refactoring, but to define some of the most important refactorings and make an abstract classification in order to make it easier to understand what we can do to adapt the assertions in the new refactorings that can appear in time.

Meyer has developed the notion of Design by Contract in the context of object oriented software construction [20]. Every method has a precondition and a postcondition which are expected to be satisfied respectively at function entry and function exit. Assertions can check such conditions.

In this paper we discuss the refactoring of assertions in the context of object oriented (OO) software; especially, when using design by contract. We extend the refactoring steps taking into account the boolean expressions of assertions.

2 An Assertion Maintenance Refactoring

We can make a first classification of refactorings depending on how they affect assertions:

Trivial Assertion Maintenance Since in many languages assertions are expressions like any other, they may be treated as such during the refactoring process. Following this criteria a lot of the Refactorings described in Fowler's book maintain the assertions consistency without doing anything. We give the names of some refactorings (using Fowler's nomenclature which we could include in this category: *Inline Temp, Replace Temp with Query, Introduce Explaining Variable* and *Remove Assignments to Parameters*.

Automatable Refactorings In the examples above, we saw refactorings like *Encapsulate Field* which the assertions are not inconsistent after the refactoring, but we can not guarantee they are correct. We should guarantee the new changes are consistent with design-by-contract in order to demonstrate their correctness. For instance, when we add new public functions it is easy to add the *Class Invariant* (*CI*) checking before the end of the method, or when we create a new subclass inheriting and checking the *CI* of the Base Class in each public method and in the constructor.

Manual Refactorings At the end we found refactorings which stops us to change the assertions automatically after applying them. Sometimes because we need to know the meaning of what the developer is trying to check with some assertion in order to modify it properly, and sometimes because the functionality of some assertion has changed, we cannot change that automatically and we have to inform the programmer. One possible solution is giving a kind of warning (like the ones given during the compilation process) to the developer.

2.1 Some Examples

To maintain assertions through the refactoring process, we are adding some extra steps, called *Augmented Refactoring Steps* (ARS) in Fowler's refactoring steps [6].

Pull Up Method The goal of the *Pull Up Method* refactoring is to move one method from one or more subclasses (in the second case the function must be the same in each subclass) to its superclass.

Following Fowler's steps [6] to carry out the Pull Up Method refactoring, we add the steps in bold:

1. Inspect the methods to ensure they are identical.
2. If the methods look like they do the same thing but are not identical, use algorithm substitution on one of them to make them identical.
3. If the methods have different signatures, change the signatures to the one you want to use in the superclass.
4. Create a new method in the superclass, copy the body of one of the methods to it, adjust, and compile.
   (a) If you are in a strongly typed language and the method calls another method that is present on both subclasses but not the superclass, declare an abstract method on the superclass.
      i **If we are in a language that allows the definition of preconditions and postconditions of abstract methods (e.g. Eiffel), we should define the precondition of that method as the AND of the precondition of the original methods, and the postcondition as the OR of the postcondition of the original methods**.
   (b) If the method uses a subclass field, use Pull Up Field, or Self Encapsulate Field and declare and use an abstract getting method.

   (c) **Ensure the CI called is the one of the superclass, otherwise do the respective modifications**.

5. Delete one subclass method.

6. Compile and test

7. Keep deleting subclass methods and testing until only the superclass method remains.

8. Take a look at the callers of this method to see whether you can change a required type to the superclass.

This refactoring can be assertion-maintained automatically. To automatise it we have to consider two parts:

- Automatise the part of defining the Pre/Post of each abstract method created in the superclass as a method that is needed for the method we are going to move. Therefore we can classify this step in *Automatable Refactorings*, but if the language does not allow the definition of Pre/Post in abstract methods (as Java) we can classify it in *Trivial Assertion Maintenance*.

- Check that the *CI* checked after moving the method is the one of the superclass instead of the subclass. If we define some rules for the checking of the Class Invariant this step can be classified in *Trivial Assertion Maintenance*, otherwise we will have to guess the CI, and it can be easy or impossible depending in how we defined the CIs. In consequence of that, this step can be *Automatable Refactorings* or *Manual Refactorings*.

Self Encapsulate Field This refactoring consists of encapsulating a field, i.e., creating its *setter* and *getter* functions. All the expressions which before called the field directly are obliged to modify it through these new methods. Figure 5 is an schematic example of this refactoring.

Following Fowler's steps [6], we add the steps in bold:

1. Create getting and setting methods for the field

   (a) **Add a precondition to the setting method which checks whether the value we are going to assign is inside the range of allowed values for the field according to the CI. This can be extracted from the CI**

   (b) **Add a postcondition to the setting method to check whether the value of the field has been changed for the one passed in the setting method**

   (c) **Add a precondition and a postcondition to the getting method to check whether the variable we are going to return is the correspondent field**

   (d) **Add the checking of the Class Invariant before the ending of each method**

2. Find all clients outside the class that reference the field. If the client uses the value, replace the reference with a call to the getting method. If the client changes the value, replace the reference with a call to the setting method

3. If the field is an object and the client invokes a modifier on the object, that is a use. Only use the setting method to replace an assignment

4. Compile and test after each change

5. Once all clients are changed, declare the field as private

6. Compile and test

To add a precondition to the setting method so that the the design-by-contract scheme is maintained, a `True` precondition would not be correct; assuming `True` as the precondition of the setting method, we will have a tempting public method which allows us to modify the values of the field according to our will. After the refactoring, we will be able to assign a value that can violate the *CI*. In turn, it also violates the *design-by-contract* (after each execution of any public method, the CI must hold – class correctness property). Once we reached this point we should be wondering whether it is possible to extract that precondition, and whether we can automatise it. To understand the procedure of extracting the precondition from the CI, it is needed to take into account that:

- Our responsibility is to modify only one field.
- The setting method will be public and the field will be private
- The *CI* is fulfilled before executing the setting method. For the *class correctness* property, we need to ensure that before and after the execution of any public method the CI must be fulfilled.
- The *CI* can have information of the allowed range values of the field. Also before the refactoring, the field may include a check for the range of acceptable values. Furthermore, Meyer aims that a correct class should have *Implementation Invariants*[**?**] in the Class Invariant, and then, it is probable to have clauses in the CI that checks the range of the field. Otherwise, we can assume $\{True\}$ as the precondition of the method, allowing the field to take every possible values.

As a consequence, we deduce that checking the *CI* is a correct precondition for the method (it must be held before the execution and probably includes the checking of the range correctness of the field). However, the *CI* is not the best precondition because it involves some clauses that does not affect the setting method, and the client does not have to be aware of them.

In this case, it is possible to extract the precondition correctly from the CI, although such a process is not trivial. This can be done creating a binary tree from the the *CI*, and then simplifying such a tree with the required information.

3 Conclusions and Future Work

Use of assertions for the development of reliable software is an established fact, and these days, they are heavily used in software industry [10]. It is more important in Object Oriented systems since they have a tendency to hide faults and assertions provide possibly an ideal mechanism to deal with them [20]. It is also true that assertions need to be consistent with the code and code changes may make them inconsistent. The notion of an Augmented Refactoring Step (ARS) may help to improve the quality of the coding and the testing processes, especially so in the context of Extreme Programming because the latter has Refactoring as a major ingredient in its development process [3].

Future work will consist of further analysis of all refactorings from Fowler [6] that can be automated. Also there are many open source tools like Eclipse which are available to automate the Refactoring steps; future work also consist of extending such support to be consistent with assertions.

Acknowledgment

We would like to thank the Spanish Ministry of Science and Technology for supporting this research (Project CICYT TIN2004-06689-C03)..

References

[1] A.M. Baumberg. *Learning Deformable Models for Tracking Human Motion*. PhD thesis, School of Computer Studies, University of Leeds, October 1995.

[2] J. Banerjee, W. Kim, H. Kim and H.F. Korth, Semantics and Implementation of Schema Evolution in Object-Oriented Databases, ACM SIGMOD Conference, 1987.

[3] K. Beck and M. Fowler, *Planning Extreme Programming*, Addison Wesley, 2001.

[4] 11. Doherty, P., W. Lukaszewick, and A. Szalas, Computing Circumscription Revisited: A Reduction Algorithm. *Journal of Automated Reasoning*, 1997. 18: p. 297-336.

[5] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, XIX American Mathematical Society, 1967, pp. 19–32.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] P. van Gorp, H. Stenten, T. Mens and S. Demeyer, Towards Automating Source-Consistent UML Refactorings, Proc. of the Unified Modelling Language Conference, 2003.

[8] M. Hiller, Error Recovery Using Forced Validity Assisted by Executable Assertions for Error Detection: An Experimental Evaluation, DSN 2000, June 2000, pp. 24-31.

[9] C.A.R. Hoare, An Axiomatic basis for computer programming, Communications of the ACM, Vol. 12 (10), October 1969, pp. 576–580,583.

[10] C.A.R. Hoare, Assertions: a personal perspective, website: `http://research.microsoft.com/~thoare/`

[11] B.H. Liskov, Data Abstraction and Hierarchy, *SIGPLAN Notices*, Vol 23(5), 1988.

[12] B.H. Liskov and J.M. Wing, A Behavioural Notion of Subtyping, ACM TOPLAS, 16(6): 1811–1841, 1994.

[13] S. Maguire, *Writing Solid Code*, Microsoft Press, 1993.

[14] T. Mens and T. Tourwe, A Survey of Software Refactoring, *IEEE Trans. on Software Engineering*, Vol. 30(2), February 2004, pp. 126–139.

[15] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1997.

[16] W.F. Opdyke, *Refactoring Object Oriented Frameworks*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[17] D.S. Rosenblum, A Practical Approach to Programming with Assertions, IEEE Transactions on Software Engineering, Vol 21(1), 1995, pp. 19-31.

[18] D.B. Roberts, *Practical Analysis for Refactoring*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

3. Satpathy, M., N.T. Siebel, and D. Rodríguez, Using Assertions in Object Oriented Software Maintenance: Experiences and Recommendations.

4. Satpathy, M., N.T. Siebel, and D. Rodríguez, Assertions in Object Oriented Software Maintenance: Analysis and a Case Study.

[19] M. Satpathy, N.T. Siebel, D. Rodríguez, Maintenance of Object Oriented Systems through Re-engineering: A Case Study, Proc. of ICSM'02, Montréal, pp. 540–549.

[20] J. Voas, How Assertions can Increase Test Effectiveness, IEEE Software, March/April 1997, pp. 118-122.