Teaching Design-by-Contract with Formal Specifications

Svetlana V. Drachova-Strang
Clemson University
School of Computing, Clemson, SC,
29634
sdracho@cs.clemson.edu

ABSTRACT

Benefits of design-by-contract are enhanced when contracts are formal. This paper summarizes our experience in teaching computer science students to use formal contract specifications in team software development, while designing participating components independently. Developed correctly, these components can be assembled together into a software system with minimal integration difficulties. Designed according to formal specifications, software becomes more reliable, fails less often, and costs less at the integration and maintenance stages. The novelty of this approach is in the use of both external and internal formal contracts in team development. Initial experiments at the graduate and undergraduate levels and evaluations show that teaching formal contracts is both possible and effective.

Categories and Subject Descriptors

K.3.2 **[Computers and Education]**Computer and Information Science Education – *computer science education*; Also:
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *modules and interfaces, object-oriented design methods, user interfaces.*

General Terms

Design, Experimentation, Languages.

Keywords

Components, formal specifications, design by contract, internal and external contracts, dependencies, compiler.

1. INTRODUCTION

This paper motivates and introduces an approach for teaching formal contract specifications to software engineering (SE) students in a team environment. The idea of team development is not new in Computer Science. Every student who takes an SE class usually participates in some type of group project. In our classrooms, we take it a step further by creating an exciting hands-on experience that illustrates the important role of formal contracts in modular software development. After students learn the basics of formal specifications in the classroom, they apply their knowledge in practice by developing software components. If components are designed correctly following formal contract specifications, they can be assembled into a software system without integration difficulties. Formal contracts not only minimize integration difficulties in combining the components written by different developers, but also lead to more reliable and maintainable software. Modern object-oriented, modular, reusable software development relies on principles of information hiding, and a number of developers will potentially be designing, implementing, modifying, and maintaining software during its life cycle. Therefore, it is critical to ensure that components are documented with formal internal and external contracts.

Any modern object oriented language combined with a suitable specification language to express contracts would work for our purposes, though one with an integrated formal specification support would be the ideal choice.

We experimented with our approach in both graduate and undergraduate classes using a software development project. The intention of the project was to determine how well students assimilated the knowledge of formal contracts and their ability to apply it in practice. This type of project, with varying levels of difficulty, can be adapted at other institutions, across the undergraduate and graduate curricula. To fully realize the benefits of contracts we used both external and internal contracts. The paper describes the experimental settings, analyzes results, and demonstrates that the results from initial experimentation and evaluation have been positive.

The paper is organized into the following sections. Section 2 presents a discussion on external and internal contracts. Section 3 describes the task at hand, experimental setup, results, and error analysis. Section 4 contains a discussion of the academic value of this approach, related work, and our conclusions.

2. FORMAL CONTRACTS

Contracts can be expressed using formal specifications. Formal specifications are mathematical descriptions of what an operation or a software component is supposed to do. They help to organize communication between various software components through precisely specifying their obligations in a non-ambiguous manner. Furthermore, given formal specifications, it becomes possible to prove that software behavior is correct in respect to these specifications using formal verification methods [Harton 08].

2.1 External Contracts

The term of "design by contract", coined by Bertram Meyer in relation to his design of Eiffel programming language, explains how software components should collaborate [4][5][7]. We define external contracts as contracts between separate software components. Some programming languages such as Eiffel[7] and Spec#[6] developed by Microsoft, have a built-in support for specifications, and others have to rely on a separate specification language. For example, Java has interfaces and implementations, and all the defined operations in an interface should be implemented in a class for that interface. But the language does not have a mechanism to specify behavior. Therefore, in order to use Java with formal contracts, it has to be used in conjunction

with a specification language such as JML [8].

Contracts specify mathematical models for objects, behavior of operations using these models, and are expressed using well-known mathematical theories and notations. Mathematical modeling of objects, preconditions and postconditions of operations, and external invariants constitute the basic elements of external contracts. To illustrate the idea, the formal specification of an Enqueue operation is given below in the RESOLVE notation [10]. It is based on a mathematical modeling of Queue as a (mathematical) string of entries.

Operation Enqueue(alters E: Entry; updates Q: Queue);
requires |Q| < Max_Length;
ensures Q = #Q o <#E>;

Here, the **requires** clause indicates a pre-condition that must be met before the routine is called. Thus, in this case the length of the queue Q should be less than the Max_Length variable in order for a new item to be added. It is the responsibility of the client to make sure that this condition is met. The **ensures** clause is the post-condition, and a correct implementation guarantees that the condition will be true after routine has executed. Syntactically, the *#Q* in the ensures clause of the above operation indicates the initial value of the *Q* before the operation. Here, it is guaranteed that the contents of the new queue Q will be the same as the contents of the initial value of Q with the initial value of the item E appended. Also, <#E> denotes the string containing the Entry value #E and “o” denotes string concatenation. It is the responsibility of the implementer to ensure that the post-condition is true. If neither condition is violated – the operation will function correctly and will produce expected results.

To make sure that students actually understand specifications, as opposed to merely guessing what they do using the names, we use “unhelpful” names in the initial exercises. In [9], our approach for teaching formal interface specifications using a *Test Case Reasoning Assistant* is discussed. This tool requires that students construct test cases based on their understanding of a given set of method specifications. It guides them through their exercises and provide real-time feedback as they work. Results in [9] indicate the effectiveness of this approach.

2.2 Internal Contracts

Internal contracts reflect the relationship between internal routines in a component, and they specifically have to do with the consistency of the internal representation of the data abstraction itself. For example, if we are working with queues – every time we enqueue an item and then dequeue one, we should get back the correct item. To ensure this, both the operations should be consistent in their internal implementation, which becomes clear by studying an example realization.

Suppose that we use a *Stack* to represent a *Queue*, and use *Stack* operations to implement *Queue* operations. Two distinct implementations are possible. We can push an item onto the top of the stack to *Enqueue* and have a *Dequeue* operation retrieve the bottom item; or we can place an item on the bottom of the stack while enqueueing and have *Dequeue* remove the item from the top. How do we make sure that *Dequeue* and *Enqueue* work consistently even if they are developed independently?. This question motivates the need for a very important kind of internal contract - *correspondence*. Correspondence is simply an abstraction function or abstraction relation. For both the *Enqueue* and *Dequeue* to work correctly they have to agree to a common correspondence. In this case, two possible correspondences are shown below:

Correspondence
Conc.Q = Reverse(Q.Contents);

Correspondence
Conc.Q = Q.Contents;

The choice of correspondence will dictate how the content of a *Queue* represented by a *Stack* will correspond to our mathematical model of a string of entries, which in it’s turn will dictate how these operations are to be implemented. In the first case the conceptual queue will be the reversal of the contents of the stack, in the second case it will directly correspond to the stack contents.

Another example of internal contract violation would be incorrect handling of the underlying structure. If we implemented a queue using an array, for example, we could use a Length variable to keep the correct index, and then increment it by 1 in *Enqueue* before adding the enqueued entry to the array. Another way of doing it is to increment the Length after adding the entry to the array. Clearly, the code for *Dequeue* should correspond. To know which way is correct we need a second piece of the internal contract: *convention*. Convention is representation invariant. If we represented the above queue with an array, we can either choose to start our queue at index 0, or at index 1. This convention will dictate our treatment of the queue and its operations. For example, in the two different conventions below:

Convention
0 <= Q.Length <= Max_Length;

Convention
1<= Q.Length <= Max_Length + 1;

the first queue starts with Length of 0, in the second case – starts with 1. To correctly implement all the queue operations we therefore need both the correspondence and the convention. There may be several correspondences and several conventions, which creates a number of possible combinations and room for error. But no matter what correspondence and convention we use, our implementation has to be consistent. This is especially true for large software items. The value of this lays in the fact that during the life cycle of software various modules are modified by different developers, and implementations may change many times as well. But if the representation remains consistent, software will still function correctly.

2.3. Enhancements

All modern languages allow components to be extended with new operations. For example, the basic *Queue* operations *Enqueue*, *Dequeue*, and *Length* constitute a basic or primary set of operations to achieve all types of queue manipulations. However, often it is convenient to extend it with other operations, such as an operation to reverse or rotate a queue. For example, a rotating search capability to a queue will add an operation that searches for a specific item while rotating the queue.. This is accomplished by using a only primary queue operatons.

Enhancements or extensions provide such special capabilities. The value of an enhancement, therefore, is that it makes it

unnecessary to rewrite the entire data structure, thus supporting the principle of reusability, but provides the ability to enhance an existing data structure with custom functionality. Like a concept, an enhancement operation contains formal specifications, and may be implemented in a variety of ways.

3. EXPERIMENTATION AND ANALYSIS

The purpose of our experimentation was three-fold: to demonstrate the practical importance of strictly adhering to the internal and external contracts while developing modules in teams, to show that strictly adhering to formal specifications produces more reliable software modules that can be easily integrated, and to highlight the effects of violating correspondence and conventions.

We conducted the experiments in both graduate and undergraduate classes. Students built several modules using RESOLVE and a translator to Java. All projects were executed, and data was collected and evaluated later on a laptop with 1.6MHz processor, 250GB hard drive, 2GB RAM, and Ubuntu Linux (Hardy Heron version).

In the graduate class, students developed modules following external and internal contracts that were provided in class, with correspondences and conventions already written by their professor. These students worked individually, and because their task was more difficult than that of the undergraduate students, they only had to develop three modules – a concept implementation, an enhancement implementation, and a test driver.

In the undergraduate class, students worked in small groups of their choice. Each group had to develop several modules. The task was the same for both groups of students– develop modules that will work when combined with modules from members of the same or different team. Relationships among modules used in the undergraduiate class are demonstrated in Figure 1. Below is the legend that explains the items in Figure1.

- C1I1 – Implementation 1 of Concept 1;
- E1 Enhancement;
- E1I1 – Implementation 1 of Enhancement 1;

3.1 Experimental Results

3.1.1 Experimental Results and Error Analysis

To test the modules for external contract violations modules developed by different students were assembled in one software item, compiled, run, and the results were tested. One of the students submitted an incomplete project, which was also incorporated into the experiment. The few that did not compile were excluded.

A table of permutations of student components was created and 159 graduate and 196 undergraduate combinations were tested. To test for internal contract violations, code for various primary operations within the same component, but from different developers were combined, compiled, and executed. Another permutations table was created and 142 randomly chosen combinations were tested. The table below indicates the percentage of the combinations that worked, and did not, if any.

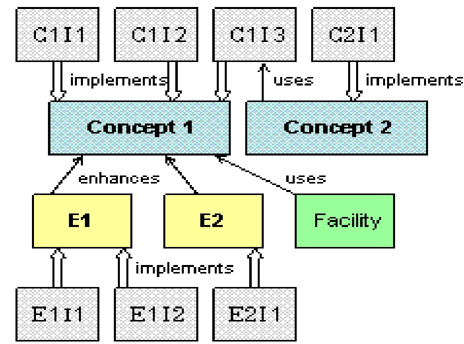

Figure 1. Component Relationship diagram

The only internal contract violations for the graduate project stemmed from violating conceptual representation of the queue data structure. The queue specifies that *Conc.Q = Q.Contents*, which puts the top of the internal stack to the front of the queue,

Table 1. Results of the experiment.

	Total External	External failed	Success	Total Internal	Internal failed	Success
Grad	159	0	100%	142	18	81.4%
Undergrad	196	27	86%	164	8	95.1%

but the erroneous implementation corresponds to the *Conc.Q = Reverse(Q.Contents)* convention. Therefore, all combinations of queue operations that satisfy different conceptual representations produced incorrect results, giving us a rate of success of only 81.6%. External contract violations in undergraduate projects resulted from not implementing operations required by a concept, or using the wrong variable name. Their internal contract violations were the result of incorrect implementation of basic operations.

An interesting question here is to try to understand how use of formal contracts impacts reliability of software. Specifically, how good are the numbers and the success rates in the table? For example, suppose that we do not use internal contracts and use a stack to represent a queue data structure. As we discussed above, we would have two possible ways of doing it because we have two different correspondences. In this case the probability of a correct implementation would be 50%.

If the queue were represented using an array, with two correspondences and two conventions, i.e. four ways of doing it, then the probability that the implementation is correct (by random accident) falls to 25%. Representing a queue using a stack is a simple task in itself, since both are simple data structures each containing several simple operations. Implementing more complex data structures (Lists, Trees, and Pointers) that are represented by other complex data structures without formal specifications will be a very challenging task, because there will containing several simple operations. Implementing more complex data structures (Lists, Trees, and Pointers) that are represented by other complex data structures without formal specifications will be a very challenging task, because there will

be a number of ways of doing it. Very few combinations will likely work because of coincidence. Our results indicate it is possible to achieve 85%-95% probability of success at integration time in classroom settings by using internal and external contracts. The ultimate experiment took place in class in front of all the students the day the projects were due: The professor randomly selected modules developed by different students and combined them in one software piece, which compiled without any errors, ran, and produced correct results. We feel it is important to mention it because this is a very exciting time for both the students and the professor, and has a great educational value. It reveals if the students were successful in learning the principle of design by contract and applying it on practice. (If the experiment in the class had failed students would have still understood the value of contracts.)

4. CONCLUSION

Design-by-contract is an important concept in software engineering. Software that adheres to specifications is more reliable and makes the later stage of maintenance easier. Teaching and learning formal specifications is not an easy task, but we have attempted to do that by involving students in some fun hands-on software development with contracts. A number of interesting projects and exercises can be used to teach formal specifications and demonstrate the principle of design-by-contract. We used RESOLVE and Java combination in our classroom, but similar project can be assigned in any object-oriented language. As shown in our experiment, the same project can be assigned on different difficulty levels, and can span across undergraduate and graduate classes of students.

As the result of our effort, both groups of students acquired hands-on experience in developing modules adhering to formal specifications, and leaned the importance of both internal and external contracts. The projects assigned were part of the SE curriculum, and fostered team work and creativity typical to SE classes. Both graduate and undergraduate groups of students successfully developed modules that were assembled together to create one correctly functioning piece of software. The ultimate test in class is a very important highlight as well, as it is a live demonstration of how well students can turn classroom theory into practice. And students can see results of their efforts right away, in “real-time”, unlike in the scenario when professor grades submitted projects in his office and returns grades to students. This also demonstrates to the professor the effectiveness of his or her teaching methods.

There were errors and lessons learned, but the most valuable thing is what every students took out of this class: No matter how many times students hear or read about a concept in class - “seeing is believing”, and nothing can substitute the hands-on experience. We believe this type of project has a high educational value and variations could be employed in all software engineering classes.

5. ACKNOWLEDGEMENTS

This research is funded in part by NSF grant DUE-0633506. The author of this paper would like to thank all the participants of the experiment for doing such an amazing job in their projects, and members of the research group who made the experiment possible. Special thanks go to the academic advisor Murali Sitaraman for his patience, invaluable suggestions, outstanding support, and numerous paper revisions.

6. REFERENCES

[1] Peter Gonn Larsen, John Fitzgerald, Tom Brooks, Applying Formal Specificatrion in Industry, IEEE Software, v.13 n3, p.48-56, May 1996.

[2] Kate Finnet, Mathematical Notation in Formal Specification: Too Difficult for the Masses?, IEEE Transactions on Software Engineering, v.22 n.2, p.158-159, February 1996.

[3] Martin D. Fraser, Kuldeep Kumar, Vijay K. Vaishnavi, Strategies for incorporating formal specifications in software development, Communications of the ACM, v.37 n.10, p.74-86, Oct.1994.

[4] Meyer, Bertrand: Design by Contract, Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986

[5] Meyer, Bertrand: Applying "Design by Contract", in Computer (IEEE), 25, 10, October 1992, p. 40-51

[6] Barnett, M., K. R. M. Leino, W. Schulte, "The Spec# Programming System: An Overview." Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS), Marseilles. Springer-Verlag, 2004.

[7] Meyer, Bertrand. Eiffel: The Language, Prentice Hall, Englewood Cliffs, NY., 1992

[8] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, ch. 12, p. 175-188. Kluwer, 1999.

[9] Dana P.Leonard, Jason O. Hallstrom, Murali Sitaraman, Injecting Rapid Feedback and Collaborative Reasoning In Teaching Specifications. SOGSE 2009

[10] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Specifying components in RESOLVE. Software Engineering Notes **19**(4) (1994), 29-39

[11] J. Kirschenbaum, H. K. Harton, and M. Sitaraman, A Case Study in Automated Verification, Technical Report RSRG-08-04, School of Computing, Clemson University, Clemson, SC 29634-0974, June 2008, 6 pages.