Teaching Software Testing using Data Structures

Ingrid A. Buckley
Department of Software Engineering
Florida Gulf Coast University
Fort Myers, FL, USA

Winston S. Buckley
Department of Mathematical Sciences
Bentley University
Waltham, MA, USA

***Abstract*—Software testing is typically a rushed and neglected activity that is done at the final stages of software development. In particular, most students tend to test their programs manually and very seldom perform adequate testing. In this paper, two basic data structures are utilized to highlight the importance of writing effective test cases by testing their fundamental properties. The paper also includes performance testing at the unit level, of a classic recursive problem called the Towers of Hanoi. This teaching approach accomplishes two important pedagogical objectives: (1) it allows students to think about how to find hidden bugs and defects in their programs and (2) it encourages them to test more effectively by leveraging data structures that are already familiar to them.**

Keywords*—*Software Testing; Data Structures; Abstract Data Type (ADT); Unit Testing; Performance Testing; Stacks; Binary Search Tree; Towers of Hanoi

I. Introduction

In general, software testing is a hugely neglected area in the software development life cycle. This is evident in students' approach to testing. Students often perform very little testing to find bugs or defects in their software projects. Even though these projects are group oriented, consisting of at least two members, testing is rarely ever an automated, planned or systematic activity. Inadequate testing is a major issue in the software development field and bugs and defects account for huge losses and rework when testing is neglected. At the course level, it is important to motivate students to take a responsible approach to software development by integrating proper testing with the aim of finding and correcting bugs and errors.

Data Structures [8] is a common course that is offered in most Computer Science, Computer Engineering and Software Engineering degree programs. However, software testing is not always a required course. The idea behind testing may seem simple to most students. However, in general, students only manually test their programs using inputs they know will always produce a correct output, instead of trying to break, find bugs or flaws in the logic of their programs [1,2,3,4,]. This phenomenon is known as confirmation bias [3] in software testing. It may be due to the fact that, finding bugs or defects, means that they will need to spend more time to optimize their programs, and time is something students are typically short of.

One of the goals of this paper is to share a relatable teaching approach that will enable students to write automated tests by considering the fundamental properties and constraints of a problem. It introduces a straight forward approach to unit testing by utilizing common data structures that are often used in programming and software development. By using data structures, along with well-known problems that are introduced earlier in the curriculum or in a prerequisite course, students can seamlessly learn the principles and application of software testing without the added burden of learning new unfamiliar content. The rest of the paper is organized as follows. Section 2 presents a fundamental overview of Stacks and Binary Trees. Section 3 explains how to test the fundamental properties and constraints of a stack, and binary search tree. Section 4 presents the Towers of Hanoi which is a classic recursive problem to illustrate performance testing at the unit level. Section 5 concludes, after discussing future work.

II. Using Data Structures for Software Testing

As stated earlier, software testing is not always a required course in most degree programs. However, it is a fundamental aspect of software development and is typically introduced briefly in the later stages of most Software Engineering courses.

Often times, students become overwhelmed with the software testing tools they need to learn to conduct automated testing. They often struggle with the concept of testing to find bugs rather than just testing to show that their software is operating perfect on a given set of inputs. To address this issue, a wide variety of software testing problems are given to students, and it becomes immediately apparent that they do not quite understand the fundamental properties or dynamics of testing to find bugs. A natural approach is to utilize Abstract Data Types (ADT) to teach them this type of testing [8].

Abstract Data Types [8] are taught in Data Structures, and most students learn about ADTs in the previous semesters to aid and develop their programming skillset and knowledge. It, therefore, makes perfect sense to utilize ADTs in teaching software testing, because doing so provides continuity and allows students to concentrate more on learning and applying testing principles.

Stacks are a last-in-first-out (LIFO) data structure. This fundamental property is easy for students to understand and test. In a stack, the element which is placed (inserted or added) last, is accessed (removed) first. Similarly, Binary Search Trees (BST) have a fundamental property which states that all elements in the left sub-tree must be less than the root, and all the nodes in the right sub-tree must be greater than or equal the root node. The Towers of Hanoi is a recursive problem that students are introduced to in Data Structures. This

This material is based upon work supported by the National Science Foundation under Grant number DUE-1562773

problem involves moving a given number of disks from peg A to peg C using peg B as auxiliary, where the disks can be moved successfully from one peg to another in a minimum number of steps. Because of the fundamental constraints of these two data structures, and the nature of the Towers of Hanoi problem, in particular, they allow students to better understand how to effectively create test cases, and to ensure that their constraints are enforced. This exercise will be explained further in Section 3.

III. The Software Testing Approach

Students are first introduced to testing at the unit level [6] using Eclipse[1] and JUnit[2]. These tools allow them to develop automated test methods and test classes [7, 8]. Unit testing is a software development process in which the smallest testable parts of a program are individually and independently analyzed for proper operation. Unit testing focuses more on finding bugs in objects, functions and classes. In particular, students are taught how to test Stacks and Binary Search Trees to ensure that their fundamental properties are not violated. They are also introduced to performance testing at the unit level.

A. Stacks

The dynamics of a stack are relatively simple [8]. Stack operations may involve initializing the stack, using it, and then de-initializing it. A stack has two basic primary operations: (a) **push()** − pushing (storing) an element on the stack; and (b) **pop()** − removing (accessing) an element from the stack. Additionally, other supporting operations that must be defined to efficiently use a stack are:

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if the stack is full.
- **isEmpty()** − check if the stack is empty.

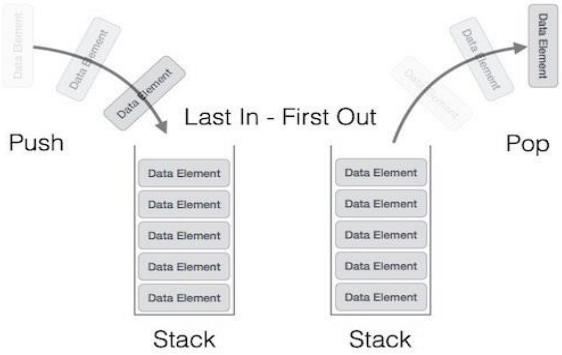

Fig. 1. Example of stack dynamics

Fig. 1 shows the basic idea behind a stack. A new element is always added at the top of the stack using the push() operation. The element at the top of the stack is always removed with the pop() operation.

B. The Stack Test

Students are asked to create a test that will effectively test the properties of a stack. This is simple to test; it involves adding a bunch of elements on a stack, and ensuring that they are removed in the correct order.

1http://www.eclipse.org

2http://junit.org/junit4/

For example, if 1, 2, 3 and 4 are pushed unto a stack one at a time, and if the stack is popped (the element at the top of the stack, is removed first, one at a time) until it is empty, then this means the stack is adhering to its fundamental LIFO property. This process is illustrated in Fig. 2.

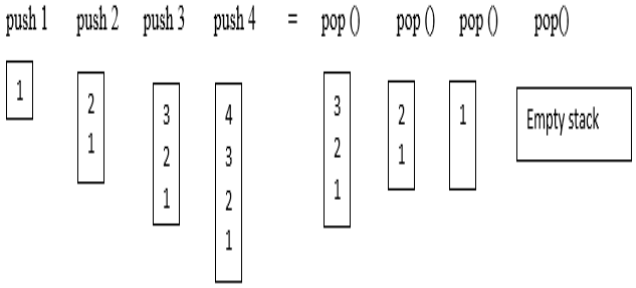

Fig. 2. Stack Unit Test

In this example 1 was pushed on the stack first; this means that 1 will be the last item to be popped from the stack. Similarly, 3 was the third element to be pushed on the stack. Therefore, 3 must be the second element to be popped from the stack. The last element that was added to the stack is 4. Thus, the first pop operation should remove an element with the value 4. In other words, the sequence and value of elements added must adhere to the LIFO constraint. In the example given, notice that each element holds a unique value to better illustrate the basic dynamics of this test. If the first pop operation removed an element with a different value, then clearly the stack is not adhering to its fundamental LIFO constraint.

C. Binary Search Tree (BST)

A Binary Search Tree [8] is a finite set of elements that is either empty or partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary search trees, called left and right sub-trees of the original tree. A left or right sub-tree can be empty; each element of a binary tree is called a node. Fig. 3 illustrates a binary search tree.

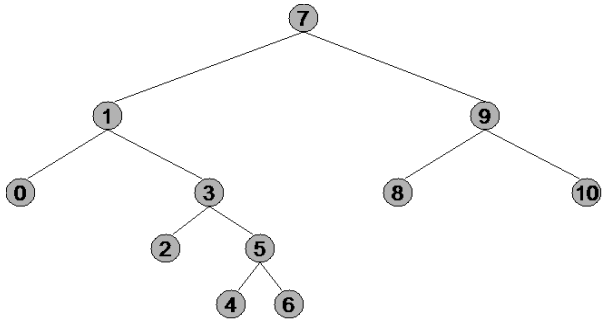

Fig. 3. Binary Search Tree

The fundamental properties of a binary search tree are: (i) all the nodes in the left sub-tree must be less than the value of the root node; and (ii) all the node values in the right sub-tree must be greater than or equal to the value of the root node.

Traversal [8] is a process that visits all the nodes in the BST in a particular order. Note that any node can be a root of the entire tree or a sub-tree. There are three (3) ways to traverse a BST; they are:

- Preorder traversal algorithm:
  - Visit the root
  - Traverse the left sub-tree in preorder
  - Traverse the right sub-tree in preorder
- Postorder traversal algorithm:
  - Traverse the left sub-tree in postorder
  - Traverse the right sub-tree in postorder
  - Visit the root
- Inorder traversal algorithm:
  - Traverse the left sub-tree in inorder
  - Visit the root
  - Traverse the right sub-tree in inorder

D. The Binary Search Tree Test

Students are asked to create a test that will effectively test the properties of a BST. A simple way to test the BST property (where all nodes in the left sub-tree must be less than the root; and all nodes in the right sub-tree must be greater than or equal to the root node) is to perform an inorder traversal on the binary search tree.

For example, given the BST in Fig. 3, an in order traversal would visit each node as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Notice that all the node values (0, 1, 2, 3, 4, 5, 6,) in the left sub-tree are all less than the value (7) in the root node. Similarly, all the node values (8, 9, 10) in the right sub-tree are greater than or equal to the root node value (7). Therefore, doing an in order traversal is a simple and effective way to test that a given tree is actually a Binary Search Tree.

The stack and binary search tree examples are just two of many ADTs that can be used to teach the fundamentals of testing at the unit level to uphold fundamental constraints.

IV. Performance Testing at the Unit Level

In unit testing [5], sometimes the performance of a given method or class is tested to determine its efficiency in solving a problem. Exhaustive testing is expensive (and time consuming). Therefore, evaluating the efficiency of a solution can be used as a performance test at the unit level. Recursion [8] is a topic that is covered in Data Structures. Essentially, recursion is used where a large problem can be broken down into smaller repetitive "sub-problems". A recursive method calls itself to perform those sub-problems, and eventually the method will come across a sub-problem so trivial, that it can handle it without recalling itself. This is known as a base case, and it is required to prevent the method from calling itself repeatedly without ever stopping.

The Towers of Hanoi [8] is a classic problem that is solved using recursion. The basic problem is as follows. Given three pegs and a stack of N disks, where each disk is a little smaller than the one beneath it, the goal is to transfer all N disks from one of the three pegs to another, while adhering to two important constraints:

- You can only move one disk at a time
- You can never place a larger disk on top of a smaller one

Fig. 4 provides an example that illustrates this problem. There are 3 disks on peg A and the goal is to move all of them to peg B while adhering to the two important Towers of Hanoi constraints mentioned above.

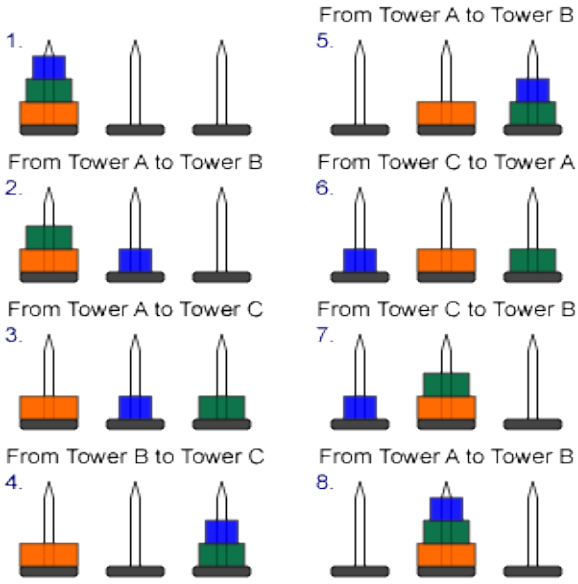

Fig. 4. Binary Search Tree

A. The Tower of Hanoi Test

Given the nature of the Towers of Hanoi problem, its performance can be evaluated, since N disks can be moved from one peg to another peg, using a minimum number of moves. Given N disks, one can mathematical find the least number of moves to achieve this goal. Students were asked to write a test that will verify that the least number of moves are used to move a stack of 3 disks from peg A to peg B using peg C as auxiliary.

Additionally, in order to define a recursive solution, students must find the base case and the recursive call [8]. The base case specifies when the function ends to avoid an infinite loop. The recursive call is defined to break the problem into smaller, yet, identical steps, to solve the bigger problem. Students would have to define the following in order to solve the Towers of Hanoi problem recursively:

1) Base case:
- When the number of disks N, equals 1.

2) Recursive call:

- Move N-1 disks from peg A to peg C, using peg B as auxiliary

In Fig 4, two disks are moved from peg A to Peg B, as shown in steps 2, 3, and 4.

- Move the remaining disk from peg A to peg B

In Fig 4, we move the remaining disk from peg A to peg B, as shown in step 5.

- Move the N-1 disks from peg C to peg B, using peg A as auxiliary.

In Fig 4, two disks are moved from peg C to Peg B, as shown in steps 6, 7, and 8.

By means of mathematical induction, the minimum number of moves required to solve the Towers of Hanoi problem is $2^n - 1$, where n is the number of disks.

This means that students would have to figure out that the minimum number of moves to transfer 3 disks from peg A to peg B is 7. This test presents a practical example of testing the performance/efficiency of a method or class by using the Towers of Hanoi example, which is a classic problem that is taught in most Data Structures course.

V. Future Work and Conclusion

Future work entails identifying, and developing, additional relatable examples that can be used to teach software testing at other testing levels-including at the integration, and system testing levels. Additionally, finding techniques and relatable exercises that help students understand code coverage in terms of data path, and input partition coverage, are also important.

Software testing is a very important activity that requires more relatable teaching strategies to help students learn how to effectively test their programs. Testing does not get enough attention in the software development life cycle and so, naturally, students do not spend enough time to fully understand the problems they are solving at a fundamental level. As a result, this negligence propagates into how they test their code.

Using the three examples in Sections 3, it was demonstrated that effective testing can be achieved by utilizing some of the basic topics covered in a typical Data Structures course. This approach focuses on understanding constraints and the fundamental properties associated with solving a particular problem. The aim is to encourage students to invest the minimum time to fully understand a problem in order to create test cases that will effectively find bugs and defects, which are the primary goals of software testing. Additionally, we extended the scope of unit testing to include performance testing of a recursive method, which was applied to Towers of Hanoi problem.

By using data structures, along with well-known problems that were introduced to students earlier in the curriculum or in a prerequisite course, they can seamlessly learn the principles and application of software testing without focusing on learning new unfamiliar content. Furthermore, students often utilize the same data structures to implement software programs in other upper level courses and internship projects. Therefore, teaching automated software testing with ADTs, provides students with a second opportunity to master their skills and knowledge in software development and testing.

References

[1] K. Muşlu, B. Soran, and J. Wuttke, “Finding bugs by isolating unit tests”, In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). 2011 ACM, New York, NY, USA, 496-499. DOI=http://dx.doi.org/10.1145/2025113.2025202

[2] K. Buffardi, S. H. Edwards, “Exploring influences on student adherence to test-driven development”, In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (ITiCSE '12), 2012, ACM, New York, NY, USA, 105-110. DOI=10.1145/2325296.2325324

[3] G. Calikli, B. Arslan, A. Bener, “Confirmation bias in software development and testing: An analysis of the effects of company size, experience and reasoning skills”, In Proceedings of the 22nd annual psychology of programming interest group workshop, 2010.

[4] G. Calikli, A. Bener, “Empirical analyses factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In Proceedings of 5th international workshop on predictor models in software engineering, 2010.

[5] L. Copeland, “A Practitioner's Guide to Software Test Design”, Artech House Publishers, ISBN: 158053791X.

[6] A. Hunt, D. Thomas, “The Pragmatic Programmer From Journeyman to Master”, Addison-Wesley, ISBN: 978-0-2016-1622-4

[7] Y. Langsam , M. Augenstein, A. M.Tenenbaum, "Data Structures using Java", Pearson Prentice Hall, ISBN: 0-13-047721-4.

[8] J. D. McGregor, D. A. Sykes. “*A Practical Guide to Testing Object-Oriented Software”,* Addison-Wesley Longman Publishing Co., Inc., 2001, Boston, MA, USA.