Implementations and Invariants

BY EVAN OLDS

CPT S 422

Priorities

1. Above all else – solid knowledge of how code works
   1. Know all the features of language you're using
   2. Know what's going on beneath your code – things that are done automatically for you
      1. Standard library stuff
      2. Compiler-hidden stuff like destructor calls and deep vs. shallow copies
   3. Most of this stuff should be review from prerequisite courses
2. Design the software itself as something that's testable
   1. If it's not flexible enough, you may not even be able to test the necessary components
   2. Much more on this later

Priorities

3. Understanding of testing
   - strategies/methods
   - tools
   - frameworks
   - flexibility
   - much much more

First…

- Some generic code review stuff follows on the next few slides
- Invariants and implementations come after

Var++ vs ++Var

- What’s the difference in output between the two pieces of code?

```
// 1.
for (int x = 0; x < 5; x++) {
    Console.WriteLine(x);
}

// 2.
for (int x = 0; x < 5; ++x) {
    Console.WriteLine(x);
}
```

Var++ vs ++Var

- What's the difference in output between the two pieces of code?

```
// 3.
for (int x = 0; x < 5; ) {
   Console.WriteLine(x++);
}

// 4.
for (int x = 0; x < 5; ) {
   Console.WriteLine(++x);
}
```

Invariant Defined

- In general it’s a Boolean property that holds true during the execution within some particular chunk of code.
- Wikipedia: “In computer science, an **invariant** is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution.”

Invariants

- Note that what we're concerned about with the design of software is defining invariants for these scopes and using them to both build and test the software.
- Recall function preconditions and postconditions from prerequisite courses
  - Precondition – what must hold true before execution of the function
  - Postcondition – what must hold true after execution of the function
- These are function-level invariants

Importance of Invariants

- If your invariants are holding everywhere that they should, then your application's state should be valid.
- Does this mean it's bug-free?
  - No, this depends heavily on how rigorously and accurately you've defined the invariants.
  - Obviously if you have an incorrect set of preconditions and/or postconditions in your software specification then having code that accurately adheres to the invariants doesn't do you much good
- So invariants aren't the whole design + testing battle, but good identification and testing of invariants is a good start.

Loop Invariants

- Simple but effective way to address correct implementation and testing of a portion of code (a loop).
- Identify the loop invariant. That is, identify what must hold true for each loop iteration.
- Fairly easy (when you have access to the code and you're not just writing external testing tools) to insert a validation to test this loop invariant.
- Think of insertion sort. This often has nested loops but the "main" (outermost) loop – **what's the invariant?**

Insertion Sort Invariants

- Remember how insertion sort works: loops through with an index i, works under the assumption that the array before position i is sorted, then moves back as needed to insert the element at i into the correct spot.
- Preconditions: ?
- Postconditions: ?
- (look at code in class and see if we got all the right pre-post conditions)

Insertion Sort Invariants

- Remember how insertion sort works: loops through with an index i, works under the assumption that the array before position i is sorted, then moves back as needed to insert the element at i into the correct spot.
- Preconditions:
  - Non-null and non-empty array with comparable data items
- Postconditions
  - All n elements in the array are sorted in ascending order
  - The n elements in the array match the original n elements
  - For insertion sort (but not all other sorting algorithms): order of equal elements has been preserved
- Loop invariant: Same as postconditions, but only applies to the subset of the array before index i.

Binary Search Tree Invariants

- What are the invariants for a binary search tree?
- How do we test these invariants?
- First let's discuss basic implementation. What's wrong with the BST class implementation (if anything) on the next slide?

BST Class

```
public class BST
{
        private class Node
        {
                public int Data;
                public Node Left, Right;
                public Node(int dataValue)
                {
                        Data = dataValue;
                        Left = Right = null;
                }
        }

        private Node m_root = null;

        public bool Add(int dataValue)
        {
                return Add(dataValue, m_root);
        }

        private bool Add(int dataValue, Node node)
        {
                if (node == null)
                {
                        node = new Node(dataValue);
                        return (node != null);
                }

                if (dataValue == node.Data) { return false; }
                else if (dataValue > node.Data)
                {
                        return Add(dataValue, node.Right);
                }
                return Add(dataValue, node.Left);
        }

}
// What's wrong with this implementation?
// How to fix without increasing the number of statements at all?
```

Binary Search Tree Invariants

- Back to testing the BST
  - What are the invariants for a binary search tree?
  - How do we test these invariants?
  - (code demo and implementation in class)

Binary Search Tree Visitor

- As seen in the code demo, the visitor allows us to plug in testing components.
- We write tree in order traversal logic once and only once, but can use it to do a variety of things (based on the visitor that's passed in):
  - ○ Print out node values on screen
  - ○ Count number of nodes in tree
  - ○ Validate BST rules for each node
  - ○ Sum up all values in tree
  - ○ See if a specific value or set of values exist in the tree
  - ○ Much more…

Unit Tests for Loop Invariants

- You want to insert calls to verification/testing functions within loops, but don’t want those calls to be in your production code when you release your product.
- What’s a good way to ensure this?

Unit Tests for Loop Invariants

- You *could* have an abstraction around some verification object and pass a “NullVerifier” in to the function for the release build.
- But if you really don’t want the verification code to be there, it’s better to just use preprocessor:

```
while (some_condition_for_loop)
{
    // Do loop stuff
    #if DEBUG
    // Verification calls here
    #endif
}
```

Final Notes

- Identifying invariants is a big part of what test engineering all about
- Abstraction is useful
    - Verification / validation objects
    - System disrupting objects
    - (notice how this comes up again and again in this class?)
- Can pass such objects in to provide testing code of major functionality