Gaining Resilience by Testing Against Abstract Data Types

Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

```
interface IMyList
{
  int Count { get; }
  void Append(int value);
  int GetFirst();
}
```

- **We can think of this interface even without a concrete class**

  New list contains no values

  Pass a number to Append( ) and it will appear in the list

  Call Append() five times and list will contain five values

- **We can try to represent these claims more formally**

  new list $\Rightarrow$ Count $= 0$

  any list $\Rightarrow$ Count ≥ 0

  new list, Append(k)
  $\Rightarrow$ GetFirst() $=$ k

  new list, N $\times$ Append()
  $\Rightarrow$ Count $=$ N

new list $\Rightarrow$ Count $= 0$

any list $\Rightarrow$ Count ≥ 0

new list, Append(k) $\Rightarrow$ GetFirst() $= k$

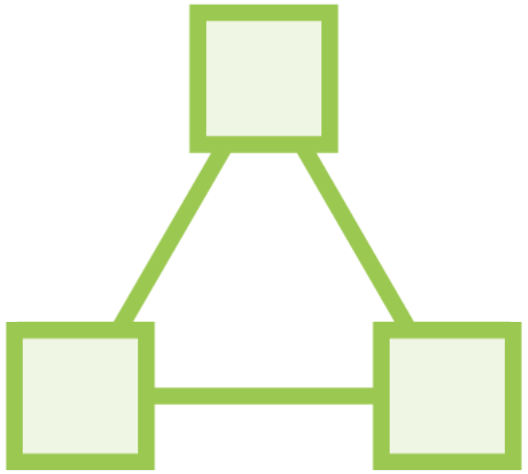
new list, N $\times$ Append() $\Rightarrow$ Count $= N$

These rules are defining an Abstract Data Type (ADT)

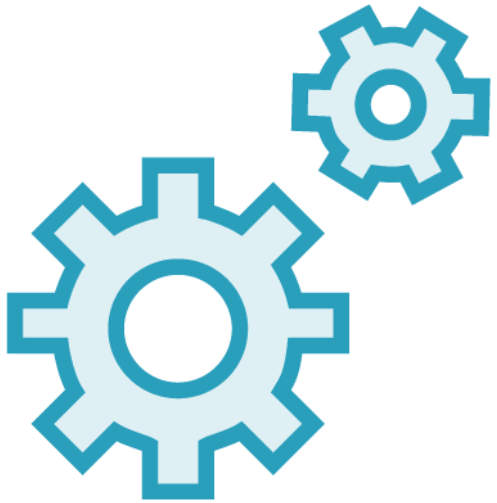
Rules of the ADT can be represented as logical implications

We can test whether a concrete type satisfies rules of its ADT

Tests Maintainability Considerations



Tests that assert behavior are easier to maintain



Tests that assert implementation are brittle

Asserting against ADT is the highest standard of testing behavior

Cyclomatic Complexity

Cyclomatic Complexity is a software metric (measurement), used to indicate the complexity of a program.
It is a quantitative measure of the number of linearly independent paths through a program's source code.

https://en.wikipedia.org/wiki/Cyclomatic_complexity

```
IMyList list = this.CreateSut();

foreach (int value in values)
  list.Append(value);

Assert.Equal(values.Length, list.Count);
```

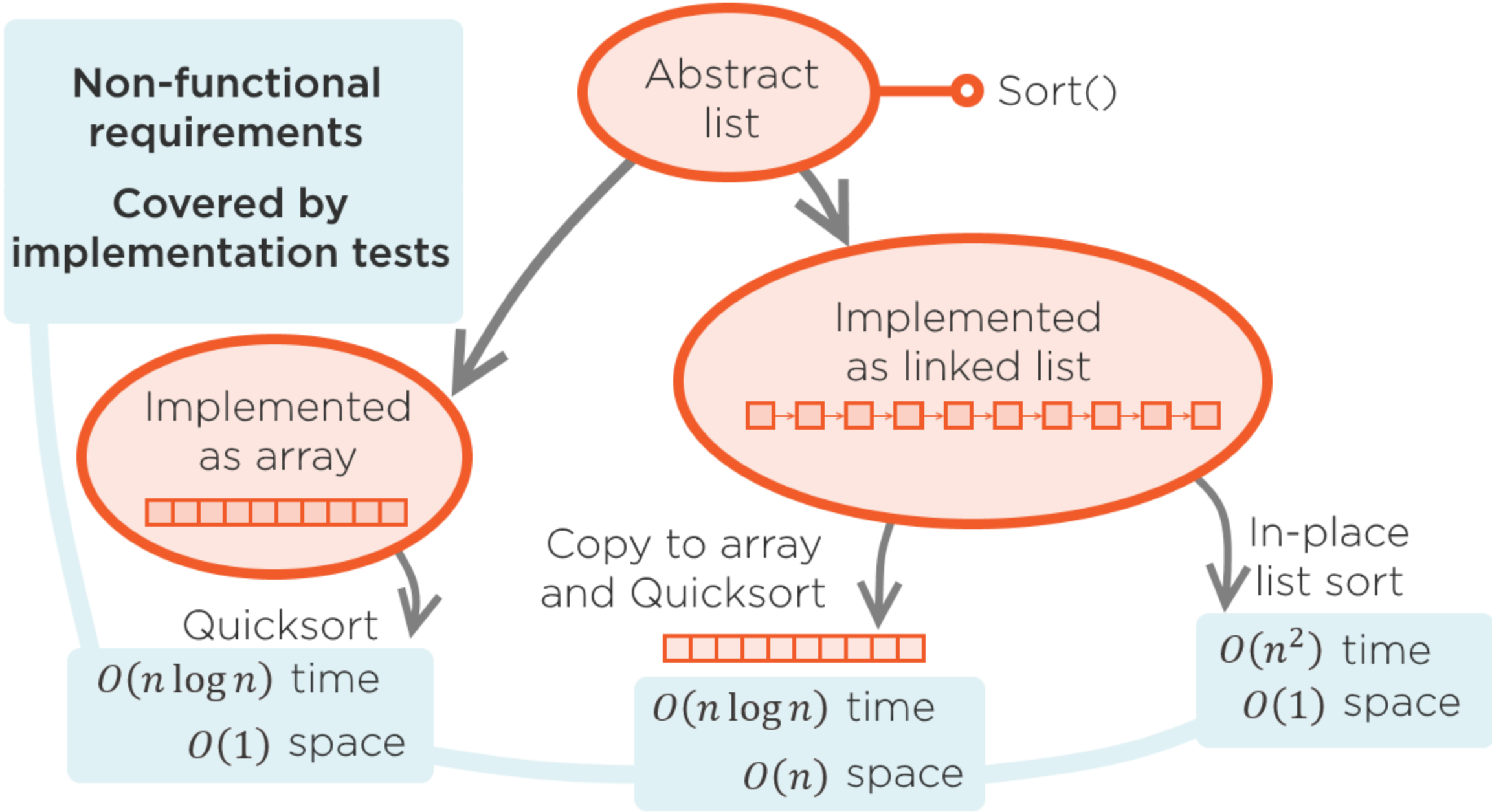
◀ **Cyclomatic complexity = 2**

```
IMyList list = this.CreateSut();
list.Append(value);
Assert.Equal(value, list.GetFirst());
```

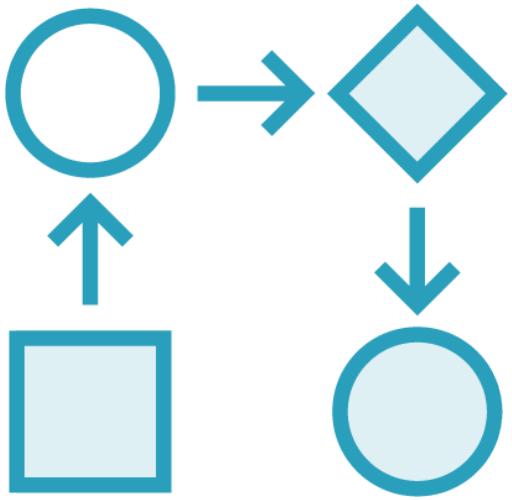
◀ **Cyclomatic complexity = 1**

Straight line of instructions is very easy to get right

Loops and branching instructions are progressively hard to get right

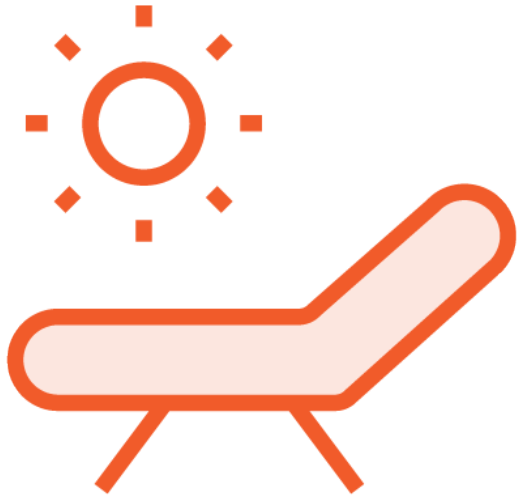

Non-functional requirements
Covered by implementation tests
Abstract list
Sort()
Implemented as array
Implemented as linked list
Quicksort
$O(n \log n)$ time
$O(1)$ space
Copy to array and Quicksort
$O(n \log n)$ time
$O(n)$ space
In-place list sort
$O(n^2)$ time
$O(1)$ space

Benefits of Writing Tests Against an ADT



We can substitute entire concrete class

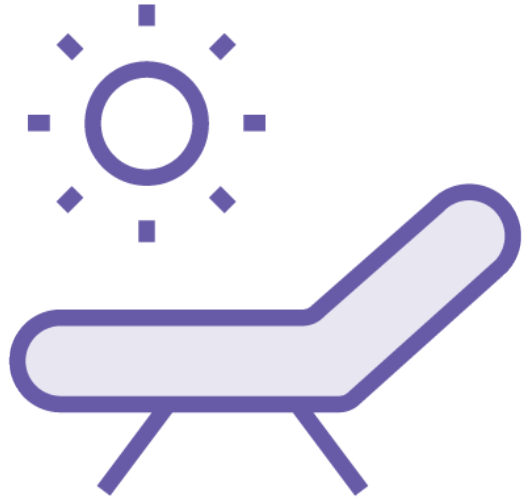
Tests must pass as they did before



No need to maintain tests written against an ADT

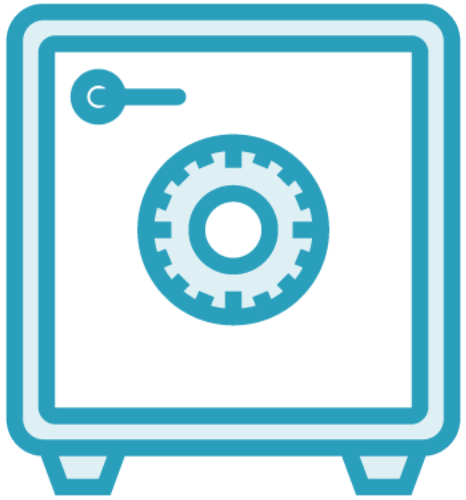
Benefits of Writing Tests Against an ADT

Rewrite entire feature
of a concrete class
when needed



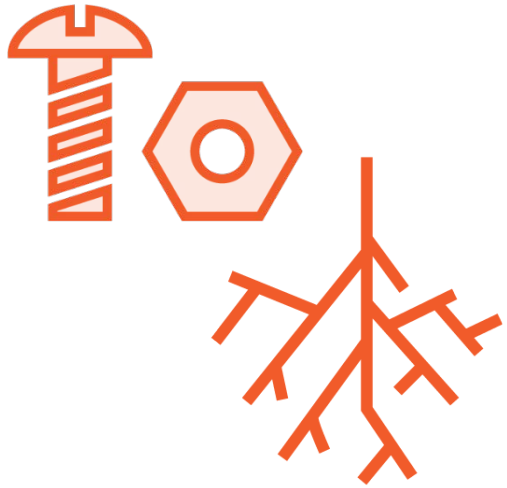
No need to make
any changes in unit tests
written against the ADT

Benefits of Writing Tests Against an ADT



Black-box testing

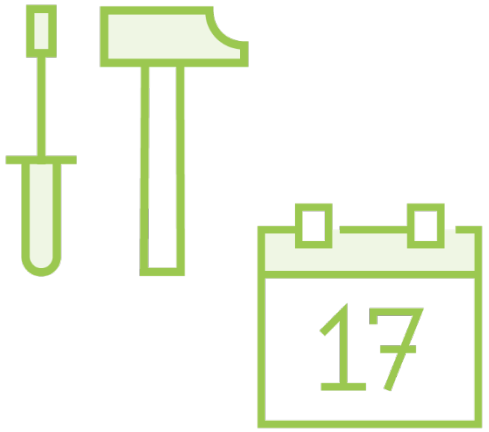
Writing unit tests against a public interface



White-box testing

Writing unit tests against concrete class text

E.g. branch coverage



Tests independent from concrete implementation are easier to maintain

Tests support future implementations

new list	$\Rightarrow \text{Count} = 0$
any list	$\Rightarrow \text{Count} \geq 0$
new list, Append(k)	$\Rightarrow \text{GetFirst}() = k$
new list, N × Append()	$\Rightarrow \text{Count} = N$
new list, N × Append(k_i)	$\Rightarrow \text{list}[i] = k_i$

Method precondition

- Condition which must be satisfied *before* the method is invoked

Method postcondition

- Condition which must be satisfied by the invoked method *after* it executes

More on preconditions and postconditions in the last module of the course

Summary

Abstract Data Type (ADT)

- Used to design classes better
- Used to discover unit tests for a class

Starting from ADT

- ADT can be turned to an interface
- Concrete class can be provided as interface implementation

Tests written against an ADT

- More resilient to implementation changes
- Change in a concrete class requires no changes in unit tests

Summary

ADT can be used to develop a class

- Missing requirements are visible in ADT rules
- That immediately indicates which additional unit tests are missing

Method applicability

- Use preconditions to terminate cases to which method is not applicable
- It is possible to avoid writing unit tests for logic in preconditions

Next module:

Testing abstract interfaces