# Testing

Mason Vail
Boise State University Computer Science

## Is my class a good implementation of my ADT?

For any non-trivial ADT, the number of possible sequences of method calls that could interact with and alter the ADT is effectively infinite. It is not possible to exhaustively test every imaginable scenario, let alone scenarios you didn’t imagine. Yet you need to be confident that your implementation works as required before you trust it in a production environment.

So how can you be confident an implementation of an ADT works correctly under all circumstances if you can’t exhaustively test it?

## Test Planning - Consistent results: Reducing risk

By carefully planning and developing a robust suite of tests for change scenarios (also called test cases) that represent changes to the ADT using its interface-defined methods, we can **reduce** the risk that there are flaws in the implementation.

For each individual change scenario, we need to test every interface method (with both valid and invalid inputs, if applicable) to confirm that every method produces the expected results according to its documentation and the expected state of the ADT after the change.

## Test Planning - Start simple and build up

We start with the simplest change scenarios and only move to more complex scenarios after we have tested the functionality on which those more complex scenarios depend.

For example, you cannot trust tests for a change to an existing, non-empty list before you have confirmed that you can correctly initialize a brand new list and that you can correctly build up to the list you want to change.

It is the combined results of all methods returning expected results for all change scenarios that builds confidence that the implementation is working correctly under all circumstances. No one change scenario, and certainly no one test, is trustworthy on its own.

## Test Planning - Boundary and Equivalence Cases

A good test suite includes all high-risk test cases that mark significant transitions, such as the initialization of a new data structure, the adding of the first element, removal of the last remaining element, changes occurring at data structure boundaries or involving significant values, if applicable. These cases are collectively referred to as “boundary cases.”

Test cases that serve as representative scenarios for many possible similar scenarios are referred to as “equivalence cases.” For example, in a list that can be arbitrarily long, once it has been confirmed that a modification can be made to an element in the middle of a multiple-element list, that scenario might serve as an equivalence case for similar modifications to all possible lists with more elements.

# SuperSimpleSet ADT Example

```
/** A collection of unique elements */
public interface SuperSimpleSet<T> {

   /** Adds given element if not already in the set. No effect if duplicate element.
    * @param element The element to add.  */
   public void add(T element);

   /** Returns true if element is in the set, else false.
    * @param element The element that may be in the set.
    * @return true if element is in the set, else false.  */
   public boolean contains(T element);

   /** Returns the number of elements in the set.
    * @return number of elements in the set. */
   public int size();

}
```

## SuperSimpleSet ADT Example

Even with only 3 methods, and only one of which changes the set contents, we still have a range of test cases to explore. Every change scenario (or test case) has a starting state, a single change, and an expected resulting state.

We must begin by testing a brand new SuperSimpleSet object - a critical boundary case. For this scenario, we are asking the question “If I had no set and call the constructor, do I get a correctly initialized new empty set?”

We need tests, then, for every SuperSimpleSet ADT method that begins with a newly instantiated set object and confirms that calling the tested method produces the result the ADT expects assuming a brand new empty set.

## SuperSimpleSet Example - New set

**Change scenario: No set -> constructor -> empty set**

**Test -> expected result**

- add(A) -> no Exception thrown
- contains(A) -> false
- size() -> 0

Assuming we have a correctly instantiated and initialized empty set, adding any new element should be allowed, any element passed to contains() should result in false, and size() should return 0. Note that every test is completely independent. Every test begins with a fresh set representing the change scenario.

# SuperSimpleSet Example - Adding first element

**Change scenario: empty set -> add(A) -> [A]**

**Test -> expected result**

- add(A) -> no Exception thrown
- add(B) -> no Exception thrown
- contains(A) -> true
- contains(B) -> false
- size() -> 1

This change scenario represents the boundary case of adding the first element to an empty set. Tests confirm the set contains the expected element, A, and calls to add() are allowed for duplicate and new elements.

## SuperSimpleSet Example - Adding duplicate element

**Change scenario: [A] -> add(A) -> [A]**

**Test -> expected result**

- add(A) -> no Exception thrown
- add(B) -> no Exception thrown
- contains(A) -> true
- contains(B) -> false
- size() -> 1

Adding a duplicate A is expected to have no effect on the set. These tests serve to confirm that the set still contains only one element, A, and calls to add() with duplicate or new elements are allowed.

## SuperSimpleSet Example - Adding second element

**Change scenario: [A] -> add(B) -> [A, B]**

**Test -> expected result**

- add(A) -> no Exception thrown
- add(B) -> no Exception thrown
- add(C) -> no Exception thrown
- contains(A) -> true
- contains(B) -> true
- contains(C) -> false
- size() -> 2

## When can I stop?

Every novel new test case and test reduces the risk that you have missed something. No novel test is worthless.

All boundary cases need to be tested. Equivalence cases for all anticipated general categories of changes (e.g. adding to the middle of a multiple-element list) need to be tested.

As new scenarios are added that conceptually overlap existing equivalence cases, however, you begin to experience diminishing returns. Less risk is reduced with each similar scenario.

At some point, you will have to decide that you can accept the remaining risk and additional test cases are unlikely to tell you something significantly new.

## Test-driven Development (TDD)

Develop a test suite for an ADT **FIRST** - *before* you write any implementation code.

- Forces you to understand the ADT and its methods before you start hacking at code.
- Places the burden on your code to prove itself against tests, rather than writing tests that make your code look good.
- Reduces bugs, as your code is tested from the beginning and flaws are revealed early. The value of this cannot be overstated.
- Prevents testing from getting crowded out at the end of a project.

# Testing

Mason Vail
Boise State University Computer Science