# Effective and Systematic Testing

## Meta Activity: Group vs Team

Throughout the course, you will need to examine and process information, ask and answer questions, construct your own understanding, and develop new problem-solving skills.

**Questions (8 min)** **Start time:**

**1**. What are some advantages to working in groups/teams?

You get to know other people and make new friends. Different people have different backgrounds and skills. The responsibility is shared.

**2**. What are some disadvantages to working in groups/teams?

Some group members may decide not to contribute. One or two students may be absent. People may not always get along with each other.

**3**. Based on the images above, what is the difference between a group and a team? Come up with a precise answer.

A group is students who just sit by each other and turn in the same assignment. A team actually works together toward a common goal, drawing on each other's strengths.

**4**. How can working as a team help you accomplish the tasks described above? Give at least two specific examples.

Working as a team makes it easier to examine and process information, because different people have different perspectives. We can also develop new problem-solving skills by observing how each other approaches the problems.

# Model 1 Testing Principles

The first chapter in the book defines effective and systematic testing and gives an overview of the book's content. It also introduces a list of principles of testing that are meant to capture why testing is difficult and how to handle it. A copy of these principles can be found at the end of the packet.

## Questions (10-15 mins) Start time:

**5**. In your own words, explain what systematic testing is and how it is different from non-systematic testing.

Answers may vary; The goal of systematic testing, as its name says, is to engineer test cases in a more systematic way, rather than simply following gut feelings. Systematic testing gives engineers sound techniques to engineer test cases out of requirements, boundaries, and source code.

**6**. Kelly, a very experienced software tester, visits *Books!*, a social network that matches people based on the books they read. Users do not report bugs often, as the *Books!* developers have strong testing practices in place. However, users say that the software is not delivering what it promises. What testing principle applies here and why?

The absence-of-errors fallacy (or verification is not validation principle). While the software does not have many bugs, it does not give users what they want. In this case, the verification is good, but the developers need to work on the validation.

**7**. Suzanne, a junior software tester, has just joined a very large online payment company. As her first task, Suzanne analyzes the past two years' worth of bug reports. She observes that more than 50% of the bugs happen in the international payments module. Suzanne promises her manager that she will design test cases that completely cover the international payments module and thus find all the bugs. Which of the testing principles may explain why this is not possible and why?

Exhaustive testing is impossible in most cases.

**8**. John strongly believes in unit testing. In fact, this is the only type of testing he does for any project he's part of. Which testing principles is John disregarding?

Pesticide paradox (or variability is important), tests are context-dependent, and absence-of-errors fallacy (or verification is not validation) principles.

**9**. Sally just started working as a consultant for a company that develops a mobile app to help people keep up with their daily exercises. The development team members are fans of automated software testing and, more specifically, unit tests. They have high unit test code coverage (>95% branch coverage), but users still report a significant number of bugs. Sally, who is well-versed in software testing, explains a testing principle to the team. Which testing principles did she most likely talk about and why?

The pesticide paradox (or variability is important) fits the discussion best. The development team has high code coverage, but they need to apply different techniques.

# Model 2 Unit Testing (Junit)

Designing testcases is different from their execution. Today, we focus on designing testcases for methods that take in lists or arrays. The book categorizes the testcases for this problem as *invalid input tests* (which are often language or problem dependent), *good weather tests*, and **boundary tests**. Note that for valid testcases certain properties are also important, for example, the ordering of the elements or whether the sequence contains duplicates. First, let's recall the testcases for the problem presented in the reading, and then apply the same technique to a couple of other problems.

**Chapter Example: Planning Poker**

**Problem specification:** In a planning poker session, developers estimate the effort required to build a specific feature of the backlog. After the team discusses the feature, each developer gives an estimate: a number ranging from one to any number the team defines. Higher numbers mean more effort to implement the feature. For example, a developer who estimates that a feature is worth eight points expects it to take four times more effort than a developer who estimates the feature to be worth two points. Given a list of estimates, the program has to return a list with the developer with the smallest estimate and the developer with the highest estimate.
**Method header:** `public List<String> identifyExtremes(List<Estimate> estimates)`

## Questions (10-15 mins) Start time:

**10**. What are the invalid input tests?

Null list
Empty list
One element list

**11**. What are the boundary tests?

One element list (which is already included)
Two elements list

**12**. What are the good-weather tests?

Three elements lists
Lists with duplicate estimates
All the permutations of a list with multiple elements (3 or more)

## Apache Commons Lang: isSorted

Apache Commons Lang is a common Java library. Below is the code and documentation for one of its methods:

```java
/**
 * This method checks whether the provided array is sorted according
 * to natural ordering.
 *
 * @param array
 *            the array to check
 * @return whether the array is sorted according to natural ordering
 * @since 3.4
 */
public static boolean isSorted(final int[] array) {
    int previous = array[0];
    final int n = array.length;
    for (int i = 1; i < n; i++) {
        final int current = array[i];
        if (previous > current) {
            return false;
        }
        previous = current;
    }
    return true;
}
```

## Questions (10-15 mins) Start time:

**13.** What are the invalid input tests?

Null array -> true
Empty array -> true
Single element array -> true

**14.** What are the boundary tests?

Two elements array: [2, 1] -> false
Single element array has been already specified

**15**. What are the good-weather tests?

Unsorted: [1, 3, 2] -> false
Sorted: [1, 2, 3] -> true
Sorted with doubles: [1, 1, 2, 2, 3] -> true

**16**. Explain how you would modify or what you would add to the code in order to correct it?

Add at the beginnig of the method:

```
1 if (array == null || array.length < 2) {
2     return true;
3 }
```

## Coding Bat: CountClumps

Coding Bat is a popular website for coding practice. CountClumps is one of the Coding Bat problems. This exercise requires you to come up with testcases at different levels and to fix the code.

### Level 1: Black-box testing

For this case, suppose you only know the problem specification and method header.
**Problem specification:** Say that a "clump" in an array is a series of 2 or more adjacent elements of the same value. Return the number of clumps in the given array.
**Method header:** `public static int countClumps(int[] nums)`

## Questions (10-15 mins) **Start time:**

**17**. Specify your testcases in different categories as you see fit:

Invalid input tests:
- Null array
- Empty array
Boundary tests
- One element array, for example: [2]
- Array with two equal numbers, for example, [2,2]
Good-weather tests:
- Array with multiple elements no clump: [3, 6, 2, 7, 4, 2] -> 0
- Array with one continuous clump: [42, 42, 42, 42] -> 1

**Level 1: White-box testing**

Suppose this is the documentation and code for the previous problem:

```
/**
 * Counts the number of "clumps" that are in the array.
 * A clump is a sequence of the same element with a length
 * of at least 2.
 *
 * @param nums
 *            the array to count the clumps of. The array must be
 *            non-null and len > 0; the program returns 0 in case
 *            any pre-condition is violated.
 * @return the number of clumps in the array.
 */
public static int countClumps(int[] nums) {
    int count = 0;
    int prev = nums[0];
    boolean inClump = false;
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] == prev) {
            inClump = true;
            count += 1;
        }
        else {
            prev = nums[i];
            inClump = false;
        }
    }
    return count;
}
```

## Questions (10-15 mins) Start time:

**18**. After seeing the code, can you think of additional tests?

Array with multiple clumps: [1, 1, 3, 0, 0] -> 2
Array with multiple clumps side-by-side: [1, 1, 0, 0, 3] -> 2

**19**. What are some issues with the provided code implementation?

- Missing code that deals with invalid input
- Correcting the algorithm for cases with different value clumps

**20**. Without changing the variables and structure of the code, how can the provided code be modified to correct the algorithm?

Changing the expressions in the if and else branch.

**21**. Without changing the algorithm, what are some code that can be added to improve the code?

Adding code that deals with invalid input

# Testing Principles Reference

### Exhaustive testing is impossible

We do not have the resources to completely test our programs. Testing all possible situations in a software system might be impossible even if we had unlimited resources. Imagine a software system with "only" 300 different flags or configuration settings (such as the Linux operating system). Each flag can be set to true or false (Boolean) and can be set independently from the others. The software system behaves differently according to the configured combination of flags. Having two possible values for each of the 300 flags gives $2^{300}$ combinations that need to be tested. For comparison, the number of atoms in the universe is estimated to be $10^{80}$. In other words, this software system has more possible combinations to be tested than the universe has atoms.

### Knowing when to stop testing

Prioritizing which tests to engineer is difficult. Creating too few tests may leave us with a software system that does not behave as intended (that is, it's full of bugs). On the other hand, creating test after test without proper consideration can lead to ineffective tests (and cost time and money). Our goal should always be to maximize the number of bugs found while minimizing the resources we spend on finding those bugs.

### Variability is important (the pesticide paradox)

There is no silver bullet in software testing. In other words, there is no single testing technique that you can always apply to find all possible bugs. Different testing techniques help reveal different bugs. If you use only a single technique, you may find all the bugs you can with that technique and no more. A more concrete example is a team that relies solely on unit testing techniques. The team may find all the bugs that can be captured at the unit test level, but they may miss bugs that only occur at the integration level. This is known as the pesticide paradox: every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual. Testers must use different testing strategies to minimize the number of bugs left in the software.

### Bugs happen in some places more than others

Given that exhaustive testing is impossible, software testers have to prioritize the tests they perform. When prioritizing test cases, note that bugs are not uniformly distributed. Empirically, our community has observed that some components present more bugs than others. For example, a Payment module may require more rigorous testing than a Marketing module.

### No matter what testing you do, it will never be perfect or enough

As Dijkstra used to say, "Program testing can be used to show the presence of bugs, but never to show their absence." In other words, while we may find more bugs by simply testing more, our test suites, however large they may be, will never ensure that the software system is 100% bug-free. They will only ensure that the cases we test for behave as expected. This is an important principle to understand, as it will help you set your (and your customers') expectations. Bugs

will still happen, but (hopefully) the money you pay for testing and prevention will pay off by allowing only the less impactful bugs to go through. "You cannot test everything" is something we must accept.

**Context is king**

The context plays an important role in how we devise test cases. For example, devising test cases for a mobile app is very different from devising test cases for a web application or software used in a rocket. In other words, testing is context-dependent.

**Verification is not validation**

Finally, note that a software system that works flawlessly but is of no use to its users is not a good software system. Coverage of code is easy to measure; coverage of requirements is another matter. Software testers face this absence-of-errors fallacy when they focus solely on verification and not on validation. A popular saying that may help you remember the difference is, "Verification is about having the system right; validation is about having the right system."