Practical Software Analysis

Exercises-week 35

Mahsa Varshosaz

Autumn 2025

Follow the instructions in the `Gradle Guide` https://docs.gradle.org/current/userguide/installation.html to install `Gradle`. Then use the provided code in the gradle project in the `exercises_code` folder so you can write and run unit tests.

We will start by practicing basic concepts and unit testing with unit testing in JUnit, designing, implementing, and automating tests (including parameterized tests). Subsequent exercises will focus on specification-based testing, introducing key concepts and deriving tests from written specifications.

Exercises

Exercise 1. (Ex. 1.6. [1]) Consider this requirement: "A web shop runs a batch job, once a day, to deliver all orders that have been paid. It also sets the delivery date according to whether the order is from an international customer. Orders are retrieved from an external database. Orders that have been paid are then sent to an external web service."

As a tester, you have to decide which test level (unit, integration, or system) to apply. Which of the following statements is true?

1. Integration tests, although more complicated (in terms of automation) than unit tests, would provide more help in finding bugs in the communication with the web service and/or the communication with the database.

2. Given that unit tests could be easily written (by using mocks) and would cover as much as integration tests would, unit tests are the best option for any situation.

3. The most effective way to find bugs in this code is through system tests. In this case, the tester should run the entire system and exercise the batch process. Because this code can easily be mocked, system tests would also be cheap.

4. While all the test levels can be used for this problem, testers are more likely to find more bugs if they choose one level and explore all the possibilities and corner cases there.

Exercise 2. Consider the method `boolean isPalindrome(String s)` in `Palindrome.java`. It returns whether `s` is a palindrome after normalization. A palindrome is a finite string that reads the same forward and backward. To normalize, the method removes all characters that are not ASCII alphanumeric (`A-Z`, `a-z`, `0-9`) and converts the remaining letters to lowercase; the check is performed only on this normalized string. If `s` is `null`, the method throws `IllegalArgumentException`. The method returns `true` if the normalized string is empty or has length one, or if the normalized string equals its reverse; otherwise it returns `false`. The provided implementation is intentionally incorrect.

1. What is the input space for testing this program?
2. What is the oracle?
3. Design a set of unit tests by selecting inputs and expected outputs based on the specification.
4. Implement the tests in JUnit (a template is provided in the accompanying exercises code). Ensure at least one test fails on the buggy implementation.
5. After your tests expose the bug(s), can you fix the code?

Exercise 3. Consider the method `String[] splitCsv(String line)` in `CsvUtil.java`. This method splits a comma-separated line into fields while preserving empty entries (including trailing columns). It help you can safely ingest raw CSV from logs or exports without silently losing data at the end of a row. It returns the fields obtained by splitting `line` on commas. There is no quoting or escaping in this exercise: commas are always separators, and whitespace is not trimmed. The method must preserve empty fields, including leading, middle, and trailing empties; for example, `"a,b,"` → `["a","b",""]`, `","` → `["",""]`, and `""` → `[""]`. If `line` is `null`, the method throws `IllegalArgumentException`. The provided implementation is intentionally incorrect.

1. What is the input space for testing this program?
2. What is the oracle?
3. Design a set of unit tests by selecting inputs and expected outputs based on the specification of the method.
4. Implement these tests in JUnit as parameterized tests (e.g., using `@MethodSource`)

Exercise 4. This exercise is inspired by a real bug that once appeared in widely used code (historically affecting `Arrays.binarySearch` in the JDK). You are given `BinarySearch.java`. The helper method `int midpoint(int low, int high)` returns the integer index in the middle of the inclusive range `[low, high]` under the precondition $0 \leq \texttt{low} \leq \texttt{high}$. For even-length ranges we use the lower middle. For example, `midpoint(3, 7)` returns `5`, and `midpoint(2, 7)` returns `4`.

The `BinarySearch.binarySearch(int[] a, int key)` method calls `midpoint` internally and returns the index of `key` if present, otherwise `-(insertion point+1)`. The current implementation may contain a bug; do not change the code until you have a failing test.

1. Write parameterized tests for `midpoint` using JUnit 5 `@CsvSource`. Start with small cases (e.g., `(0,0)`→`0`, `(0,1)`→`0`, `(1,2)`→`1`, `(2,7)`→`4`, `(3,7)`→`5`). If you have not used `@CsvSource` before, see the JUnit 5 User Guide: https://bit.ly/3HOnbMI.

2. Now instead write a parameterized test using `@CsvFileSource`. A set of initial test cases are provided in the file `src/test/resources/midpoint_cases.csv`. The file has three semicolon-separated columns: `low;high;expected`.

3. Extend the CSV by appending additional valid ranges, including very large bounds, until at least one test fails. Do not modify the code before you obtain a failing test.

4. Fix `midpoint`, re-run the suite, and keep your added rows as regression tests.

5. Write parameterized tests for `BinarySearch.binarySearch` over a fixed sorted array (e.g., `A = {0,1,2,3,4,5,6,7,8,9}`) using `@CsvSource`. Confirm the tests pass once `Midpoint` is correct.

Exercise 5. (Ex. 2.1 [1]) Which statement is false about applying the specification-based testing method on the following Java method?

```
/**
 * Puts the supplied value into the Map, mapped by the supplied key.
 * If the key is already in the map, its value will be replaced by the new value.
 * NOTE: Nulls are not accepted as keys; a RuntimeException is thrown when key is
    null.
 * @param key the key used to locate the value
 * @param value the value to be stored in the HashMap
 * @return the prior mapping of the key, or null if there was none.
 */
public V put(K key, V value) {
    // implementation here
}
```

A. The specification does not specify any details about the *value* input parameter, and thus, experience should be used to partition it (for example, value being null or not null).

B. The number of tests generated by the category/partition method can grow quickly, as the chosen partitions for each category are later combined one by one. This is not a practical problem for the `put()` method because the number of categories and partitions is small.

C. In an object-oriented language, in addition to using the method's input parameters to explore partitions, we should also consider the object's internal state (the class's attributes), as it can also affect the method's behavior.

D. With the available information, it is not possible to perform the category/partition method, as the source code is required for the last step (adding constraints).

Exercise 6. (Ex. 2.5 [1]) A game has the following condition:

$$\texttt{numberOfPoints} \leq 570$$

Perform boundary analysis on the condition. What are the on- and off-points?

1. On point = 570, off point = 571
2. On point = 571, off point = 570
3. On point = 570, off point = 569
4. On point = 569, off point = 570

Exercise 7. (Ex. 2.3 [1]) Zip codes in country X are always composed of 4 numbers + 2 letters, e.g., `2628CD`. Numbers are in the range $[1000, 4000]$. Letters are in the range $[C, M]$. Consider a program that receives two inputs: an integer (for the 4 numbers) and a string (for the 2 letters), and returns `true` (valid zip code) or `false` (invalid zip code). The boundaries for this program appear to be straightforward:

- Anything below 1000 $\rightarrow$ invalid
- $[1000, 4000] \rightarrow$ valid
- Anything above 4000 $\rightarrow$ invalid
- $[A, B] \rightarrow$ invalid

- $[C, M] \rightarrow$ valid
- $[N, Z] \rightarrow$ invalid

An implementation `ZipCodeValidation.java` (with possibly some bugs) is provided in the exercises folder. Implement a set of unit tests (e.g., JUnit 5; parameterized tests are encouraged) that exercise these boundaries and your chosen invalid cases. How many bugs are there in the implementation?

Exercise 8. (based on Ex. 2.4 [1]) We have a program called `FizzBuzz`. It does the following: Given an integer `n`, return the string form of the number followed by `"!"`. If the number is divisible by 3 use `"Fizz"` instead of the number, and if the number is divisible by 5 use `"Buzz"` instead of the number, and if the number is divisible by both 3 and 5, use `"FizzBuzz"`. For example:

- The integer `3` yields `"Fizz!"`
- The integer `4` yields `"4!"`
- The integer `5` yields `"Buzz!"`
- The integer `15` yields `"FizzBuzz"`

A novice tester is trying hard to devise as many tests as she can for the `fizzBuzz` method and has designed the following tests:

$$\text{T1} = 15, \quad \text{T2} = 30, \quad \text{T3} = 8, \quad \text{T4} = 6, \quad \text{T5} = 25$$

1. Based on the specification (without seeing the code), what are the input *equivalence partitions* over the integers (i.e., cases that should yield the same behavior)?
2. Based on your answer, which of the above tests can be removed while keeping a good test suite?
3. Add *boundary* tests around the transitions between partitions (e.g., values immediately below, at, and above the points where behavior changes).

An implementation is provided in the exercises folder. There are bugs in the implementation. Implement a set of tests based on the specification; do not modify the implementation until you have at least one failing test.

Exercise 9. This exercise is inspired by a common real-world bug class: overflowing 32-bit integers while reversing digits. You are given `Numbers.java` with the method `int reverse(int x)` in the exercises folder. Given a 32-bit signed integer `x`, the method returns the integer formed by reversing its decimal digits. The sign is preserved; leading zeros in the reversed form are discarded. If the reversed value would fall outside the `int` range $[-2^{31}, 2^{31} - 1]$, the method returns `0`. For example:

- `reverse(123)` $\rightarrow$ `321`
- `reverse(-123)` $\rightarrow$ `-321`
- `reverse(120)` $\rightarrow$ `21`
- `reverse(0)` $\rightarrow$ `0`
- `reverse(1534236469)` $\rightarrow$ `0` (would overflow)

1. From the specification, identify input *partitions* over integers that should yield the same observable behavior. *Hint: consider zero vs. non-zero, positive vs. negative, single-digit vs. multi-digit, ...*
2. For each partition, choose one representative input and write a JUnit test. Keep tests small and descriptive.
3. Add *boundary* tests around transitions between partitions e.g., near-overflow pairs, extremes, and trailing zero cases
4. Run your suite. If a test fails, record which partition/boundary revealed the bug. Then fix `reverse` according to the specification and re-run the suite.

Exercise 10. Consider the `ListNode partition(ListNode head, int x)` method. The method should rearrange the nodes in a linked list so that all nodes with values *less than* `x` appear before nodes with values *greater than or equal to* `x`. The relative order of nodes within each partition must be preserved. For example:

- `partition([1,4,3,2,5,2], x=3)` $\rightarrow$ `[1,2,2,4,3,5]`
- `partition([2,1], x=2)` $\rightarrow$ `[1,2]`
- `partition([1,2,2], x=3)` $\rightarrow$ `[1,2,2]`
- `partition([], x=1)` $\rightarrow$ `[]`

1. From the specification, identify input *partitions* over linked lists that should yield the same observable behavior. *Hint: consider empty vs. non-empty, all nodes* $< x$ *vs. all nodes* $\geq x$*, mixed cases,...*

2. For each partition, choose one representative input and write a JUnit test. Keep tests small and descriptive.

3. An implementation of the method is provided (cf. `PartitionList.java` in the exercises folder).Run your test suite. If a test fails, record which partition revealed the defect. Then fix `partition` according to the specification and re-run the suite.

Exercise 11. (From [2]) We have the following method:

```
public String sameEnds(String string) {
  int length = string.length();
  int half = length / 2;

  String left = "";
  String right = "";

  int size = 0;
  for(int i = 0; i < half; i++) {
    left += string.charAt(i);
    right = string.charAt(length - 1 - i) + right;

    if (left.equals(right))
      size = left.length();
  }

  return string.substring(0, size);
}
```

Perform **boundary analysis** on the condition in the `for`-loop: `i < half`, i.e., what are the on- and off-point and the in- and out-points? You can give the points in terms of the variables used in the method. You may express the points in terms of the variables used in the method.

References

[1] Maurício Aniche. *Effective Software Testing: A Developer's Guide*. Manning Publications, 2022.

[2] Maurício Aniche. *Software Testing: From Theory to Practice*. 2022.