# Test Driven Development (TDD) Really Works

The current Engineering interviewing process, when exploring white-board algorithm implementations, often reveals aspects of engineering not considered by either the interviewer or candidate prior to the actual interaction. The two most common ignored issues in my experience are: a lack of testing approach (e.g., *TDD*) when white boarding, and sometimes not even limited discussion of performance considerations.

I would like to use a recent interview experience to highlight both topics, and consider TDD in more detail. We will first review the design session interactions, briefly consider performance implications, and finally consider *TDD* aspects. You can jump ahead to the TDD discussion by jumping to title “TDD Description” below. We will examine the problem statement and proposed solution process discussed in the interview.

## The Initial Problem

We must count unique client IP accessing our web-site over some time period (a day let’s say). Note that we are only considering 32 bit IP address. We :

1. Are not required to persist the IP access counts outside of the session.

Need to support various query types (e.g., counts for a specific client subnets accessing our site.)

## Initial Proposed Solution

Use an In-Memory data grid to record client references ([Client-URL, Access Count] pairs.) A Data Grid platform offers a mechanism to store counts as necessary, and handles both distributing counting across monitored servers, and an aggregated view of the counts. The IMDG platforms support complex queries as well. Please review reference #1 for a background on Data Grid platforms.

## The Modified Problem

Imagine we have a single server and we must provide our own solution. For our own solution:

* What would be an appropriate data structure to store client URL and access counts?
* How might we query for subnet usage accounts?

An IP subnet is a group of 32 bit client IPs beginning with the same bit pattern. Please see reference #2 for an explanation of subnets.

## White Board Solution

We record each client URL reference in a ***HashMap*** entry that uses the 32 bit URL as a key, and keeps the accumulated access counts as the value associated with the client URL. We would obtain the subnet counts with this process.

First, define the MASK as a Java ***Integer*** with leading one bits for the subnet, and a PATTERN as the value of the subnet starting at the left end of an ***Integer***. Both MASK and Pattern have trailing zeros after the subnet part of their value. For example, subnet 1011 would use PATTERN 0xB0000000 and mask 0xF0000000. The algorithm is:

1. Extract the ***HashMap*** keys into a Collection.
2. Given a subnet MASK and PATTERN, create a search candidate ***W*** as MASK *and* PATTERN.
3. Iterate through the keys of the ***HashMap***, forming subnet identifiers ***S*** as MASK *and* KEY.
4. Increment a counter for ***C*** when ***S*** == ***W***
5. Return the cumulated count when all keys have been examined.

A proposed performance modification was to replace the iteration mechanism is step ***C*** with a binary search to find the initial subnet entry matching ***W***, followed by an early scan termination at the first key exceeding ***W***. This optimization could only be done if the keys extracted in step ***A*** were ordered. The ordering needed to be unsigned because we are using bit patterns and not signed binary integers. Java sorting required an Unsigned Comparator.

## Requirements Clarification and Algorithm Modification (Refactoring)

We had to clarify the requirement “How might we query for subnet usage accounts?” to read “How might we query for subnet access counts, computed as the total of accesses from each client URL in the subnet?” This clarification caused us to change the solution step ***D*** above to increment the counter for ***C*** differently. Instead of incrementing by one, we needed to increment by the number of accesses associated with KEY. Please see reference #3 for the Java code implementing the flow description above.

A small snippet of the code to accomplish this is:

**private** **static** **int** countMatchesInUnsortedArray(**final** Map<Integer, Integer> ipCounts, **final** **int** mask, **final** **int** pattern, **final** **int**[] keys, **int** start,

**int** length) {

**int** count = 0;

**if** (length < 1) {

**return** count;

}

**final** **int** wantedPrefix = mask & pattern;

**for** (**int** i = start; i < (start + length); i++) {

**if** (wantedPrefix == (mask & keys[i])) {

count += ipCounts.get(keys[i]);

}

}

**return** count;

}

## Performance Analysis

The performance analysis code is compares key load time (algorithm step A above) and access count execution time (algorithm steps ***B*** through ***D*** above.) The code for the performance tester is in GitHub and the URL is defined in reference #4 below.

## TDD Description

We construct components of a system and compose a complex system in layers using component interactions to achieve system functionality. Each component has two basic kinds of testing: “unit” testing and “integration” testing. Unit testing provides for isolated execution of a component independent of other components. As the component is developed, assertions about component functional behavior are added to the unit test.

Regression testing is extremely helpful when a major refactor is required (e.g., the requirement clarification for counting subnet accesses instead of hosts in algorithm step ***D*** above.) If something changes during on-going development, the accumulated functional assertions provide a level of confidence that the component continues to work as it is refactored.

The required tests, their creation time sequence, and their uses in development are outlined here:

|  |  |  |  |
| --- | --- | --- | --- |
| Phase | Test Element | Focus | Regressions |
| Solution Implementation | TestRunner.java | Orchestrate tests |  |
|  | TestUnsortedKeyExtractor.java |  |  |
|  |  |  |  |
|  | TestUnsignedComparator.java |  |  |
|  | TestUnsignedBinarySearch.java |  |  |
|  | TestSortedKeyExtractor.java |  |  |
|  |  |  |  |
|  | TestSimpleMatcher.java |  |  |
|  | TestBoundedMatcher.java |  |  |
|  |  |  |  |
| Performance Testing | IPBuilder.java |  |  |
|  | TestIPBGenerateIPAddressesAndCounts.java |  |  |
|  | TestIPBGeneratedObservedCounts.java.java |  |  |

TestRunner.java

TestUnsortedKeyExtractor.java

TestUnsignedComparator.java

TestUnsignedBinarySearch.java

TestSortedKeyExtractor.java

TestSimpleMatcher.java

TestBoundedMatcher.java

IPBuilder.java

TestIPBGeneratedObservedCounts.java.java

TestIPBGenerateIPAddressesAndCounts.java

## Resources

1. An in-memory data grid overview and examples: <https://www.predictiveanalyticstoday.com/top-memory-data-grid-applications/>.
2. Subnet of IP explanation: <https://www.pcwdld.com/subnet-mask-cheat-sheet-guide>.
3. GitHub code repository parent for this article (and others) is: <https://github.com/DonaldET/DemoDev>, and the solution implementation for this article is based at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-codingexams/dev-topics-liveramp-bitsearch>.
4. Performance data are located in GitHub at yyyyyyy.