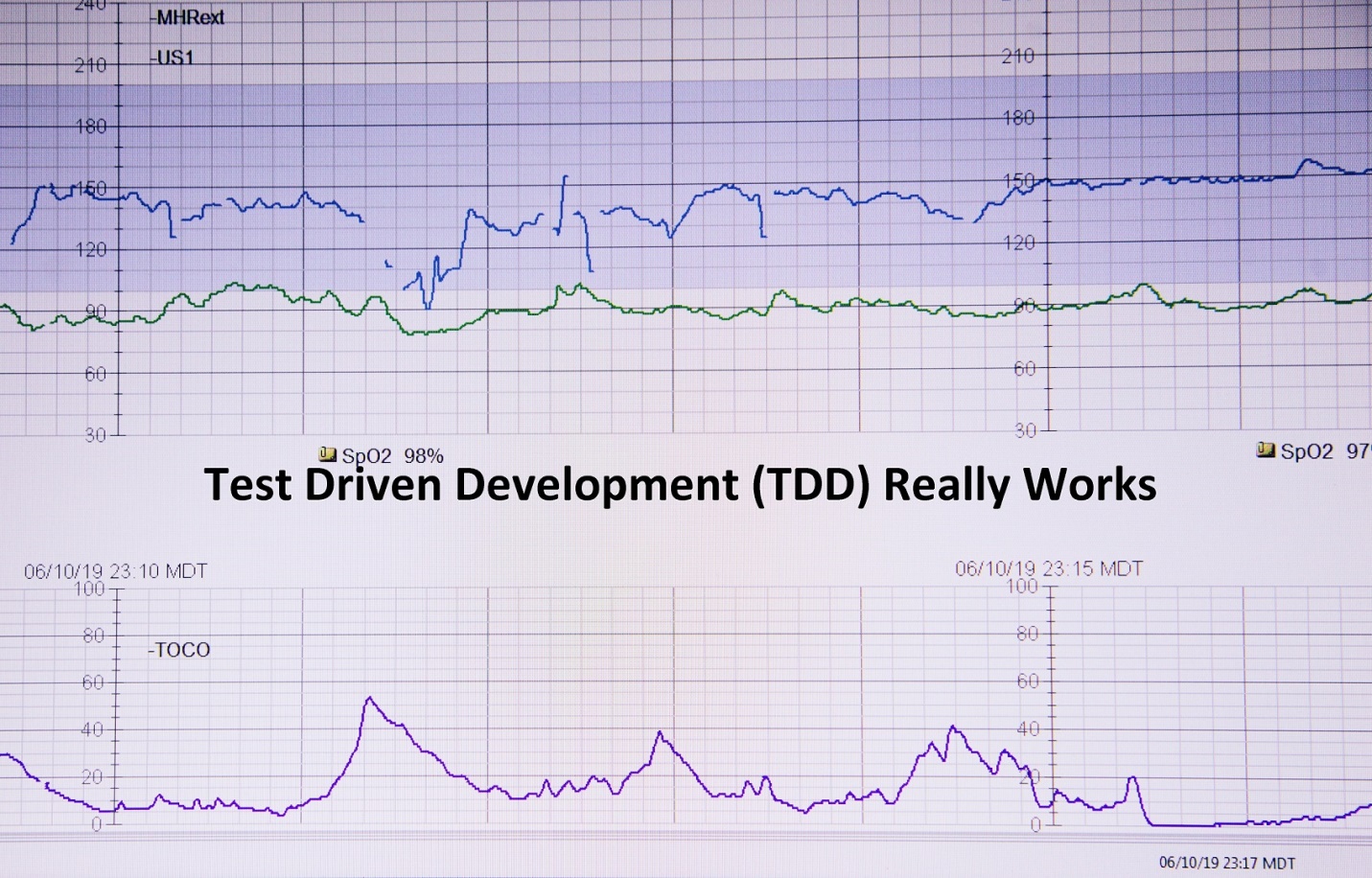
# Test Driven Development (TDD) Really Works



## Executive Summary

Constructing, modifying, and understanding software is enhanced by a suite of tests. Tests enable constructing features incrementally. Tests offer a debugging observation point into individual software components. A body of working tests allow a developer to modify code with confidence that disaster has not occurred. Keeping the tests in a regression context gives confidence in making modifications. Reviewing tests gives the maintainer a sense of how the software is used and components orchestrated. This article focuses on the regression aspect of tests in the contest of performance improvement. The article TDD And Algorithm Development focuses on the construction phase of TDD.

The current Engineering interviewing process, when exploring white-board algorithm implementations, often reveals aspects of software engineering not considered by either the interviewer or candidate prior to the interview. The two most common under examined issues in my experience are: a limited discussion of performance considerations and a lack of testing approach (e.g., *TDD*). Here we offer a more complete example with performance and testing.

I will use a recent interview experience to highlight both topics, and to 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 locating the title “TDD Description” below. Hopefully, given your interest, we will first examine the problem statement and proposed solution process discussed in the interview.

## The Initial Problem

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

1. Are not required to persist the IP access counts outside of the session (~5 million.)
2. Need to support various query types (e.g., counts for a specific client subnets accessing our site.)

## Initial Proposed Solution

We would use an In-Memory Data Grid (IMDG) implementation to record client references ([Client-URL, Access Count] pairs.) An IMDG platform offers a mechanism to store counts as necessary, and handles distributing counting across monitored servers, and offers an aggregated view of the counts. It also supports complex queries of the collected data. Please review reference #1 below for a background on In-memory Data Grid platforms.

## The Modified Problem

Imagine we are not going to use an Enterprise product and we have a single server for which we must provide our own custom-coded solution. For our own custom solution:

1. What would be an appropriate data structure to store client URL and access counts?
2. 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 on the left and the differing right-hand bits identify a host. 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 the map keeps the accumulated access counts as the value associated with the key. Each client access either adds a new URL key and an access count of one, or increments the access count associated with the existing URL key. We next describe how to obtain the subnet usage counts.

First, define a MASK as a Java ***Integer*** with sufficient 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 they define. For example, binary subnet 1011 would use PATTERN 0xB0000000 and mask 0xF0000000. The Simple Search algorithm implementing the subnet counts collection from the usage counts ***HashMap*** is:

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

I proposed a performance modification 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, named Bounded Search here, could only be done if the keys extracted in step ***A*** were ordered. Java bit-pattern sorting requires an Unsigned ***Comparator*** because the default ***Comparator*** is signed. The Java Collections binary search requires the same ***Comparator***.

## Added Requirements Clarification and Algorithm Modification (Refactoring)

We had to clarify the requirement #4: “How might we query for subnet usage accounts?” It changed to: “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 the counter by one, we needed to increment it by the number of accesses associated with URL key. Please see reference #3 for the Java code implementing the flow description above. A small snippet of Java code to accomplish the simplest search 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 (i.e., **PerformanceRunner.java**) compares these algorithm actions over an input range of interest (less than five million unique URLs access the site in a day):

* Key load time (algorithm step ***A*** above, the key extraction from the ***Map***), and
* Query of Access Count, or search execution time (algorithm steps ***B*** through ***D*** above.)

Please note that the complete code for the performance tester is in GitHub (see reference #4 below.) The population load time analysis graph is:

The load for the later Bounded Search (**BS**) key extraction is within 175 nanoseconds of the key extraction for the un-randomized Simple Search (**not-rand SS**). To insure a random distribution of ***HashMap*** keys in the following search testing, we perform an extra key shuffle step in key extraction for randomized simple search (**SS**); we note that randomization adds significant execution time. Still, at worst, this added time is less than 400 NS.

Once the load step (key extraction) is completed, we use the extracted keys to search for six MASK/PATTERN pairs and collect counts. We do this using both Simple Search (**SS**) and Bounded Search (**BS**) algorithms. The search performance graph results are:

We see that the initial binary search and early termination of the Bounded Search (**uBS**) algorithm does indeed run faster than Simple Search (**uSS**). However, for the range of interest, **uBS** is only 48 NS faster than **uSS**). We would need four searches to make up for the extra load time of sorted keys. We were able to make good use the of the **Arrays.parallelSort** method to accelerate sorting speed.

*As suggested during the interview, modern CPU machine-level operations are so fast that sometimes simpler algorithms offer acceptable performance and lower complexity for the range of interest.*

## TDD Description

For me, the real opportunity of solving this kind of problem is to showcase the role of TDD!

Engineers construct components of a system and compose complex systems 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 a component is developed, assertions about component’s functional behavior are added to the unit test incrementally. Integration testing covers interactions between multiple components, and has assertions added incrementally as well. TDD combines incremental testing during development and ongoing regression testing to insure code stability.

Regression testing is extremely helpful when a major refactoring is required (e.g., here, 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 being refactored.

The required tests for this example problem, their creation time sequence, and their uses in development and regression stability are outlined here:

|  |  |  |  |
| --- | --- | --- | --- |
| **Phase** | **Test Element** | **Responsibility** | **Regression Tests** |
| *Solution Components* | ***TestRunner.java*** | Orchestrate tests | Rerun all after performance modifications |
|  | ***TestUnsortedKeyExtractor.java*** | Map keys into int[] (speed-space optimization) | Test different collections |
|  | | |
| ***TestUnsignedComparator.java*** | Required for bit strings | Replace custom comparison with built-in **compareUnsigned** in **Integer** class |
| ***TestUnsignedBinarySearch.java*** | Search ordered list and indication a point of insertion key not in list | Handle key not found and comparison indicator |
| ***TestSortedKeyExtractor.java*** | Map keys into ordered int[] | Substitute **Arrays.parallelSort** for plain sort |
|  | | |
| ***TestSimpleMatcher.java*** | Sum counts of client access for clients on the same subnet | Check modified ***Comparator*** |
| ***TestBoundedMatcher.java*** |
|  | | |
| *Performance Components* | ***IPBuilder.java*** | Construct mock data | Multiple tuning runs |
|  | ***TestIPBGenerateIPAddressesAndCounts.java*** | Create counts for a single subnet |
| ***TestIPBGeneratedObservedCounts.java.java*** | Use single generator to create a consistent group of subnets |

The above table demonstrates how the tests both aid in developing components, and in regression testing enabling confident refactoring. Another value not shown here is the ability to use regression testing to aid in debugging. *This is the way to go!*

## References

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 URL 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 analysis data for this article are located in GitHub at <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-codingexams/dev-topics-liveramp-bitsearch/analysis>.
5. A TDD overview: <https://www.guru99.com/test-driven-development.html>.
6. A similar article by this author on performance and algorithms: <https://www.linkedin.com/pulse/lies-damn-algorithm-analysis-donald-trummell/>.

Ω