Software Engineering for Geoinformatics

June 4, 2020

Software Test Document

Web-Based Geospatial Data Analysis Application on Environmental Noise Pollution Survey in Pune, India

Version: 0.1

Authors:

R. Cedeno, D. Cumming, H. Mahmoudi, A. Zacchera

Table of Contents

1. Introduction

Testing is the last important step before the role out of software. Its purpose is to verify that the code performs nominally, that is it successfully passes a set of test cases, and to validate the software against the original requirements and specifications contained in the Requirement Analysis and Specification Document. The testing phase terminates with the user acceptance test by the client, which usually also coincides with the time when the software enters production stage.

Testing can take place at different levels of abstraction, ranging from the code review or unit testing, which usually involves inspecting single lines of code, to integration testing and, eventually, system testing. Furthermore, testing shall be scaled on the size and complexity of software and the required level of reliability, with large, highly complex and mission-critical software requiring more sophisticated and ramified testing procedures.

A common approach to testing is based on the identification of test cases. A test case is a collection of inputs (e.g. function input parameters), assumptions about the internal state of the system at the time when inputs are provided and a set of expected outputs (e.g. returned values by a function). If the results produced by a piece of code match the expected outputs, then the test case is successful. Two or more test cases can be grouped in a test set. A test set is successful if and if all the test cases in it are successful.

There are several methods in which one can generate test cases but, in general, they can be classified in two main categories: random testing and systematic testing. Random testing is based on statistical methods that can generate and execute hundreds of thousands of pseudo-random test cases, require little a priori knowledge but are “blind” in the sense that they do not look for bugs. Systematically testing, on the contrary, requires stronger a priori knowledge of the problem but it allows to focus on those parts of the software that more likely to hide bugs or unwanted behaviors. Systematic testing methods can be further grouped into: “white-box” testing methods based on use characteristics/structure of the software

artifacts (e.g. code), or “black-box” testing methods which exploit information about the behavior of the system as whole (e.g. specifications

In this document we briefly discuss a possible and simplified version of the V-model (see Figure 1) and show some examples of how it can be applied to the software we developed. However, the reader should be aware that, while providing some examples of test cases, this document is neither an exhaustive discussion about software verification and validation in general or a complete testing process of our web application.

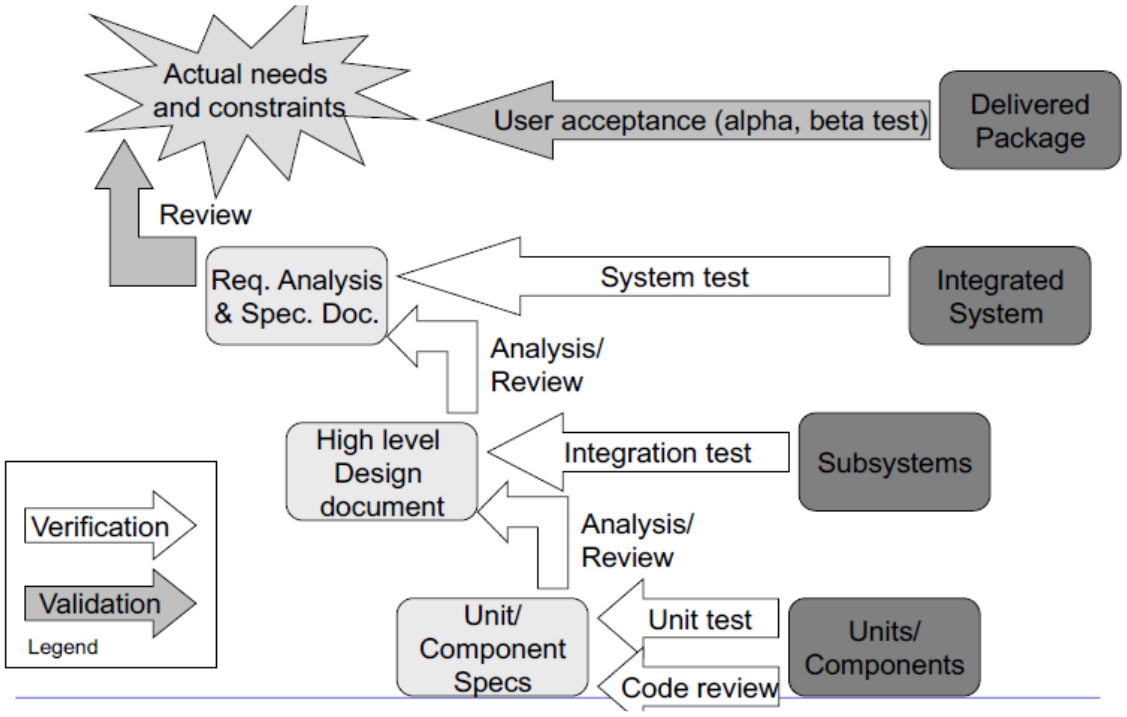

Figure 1: V Model

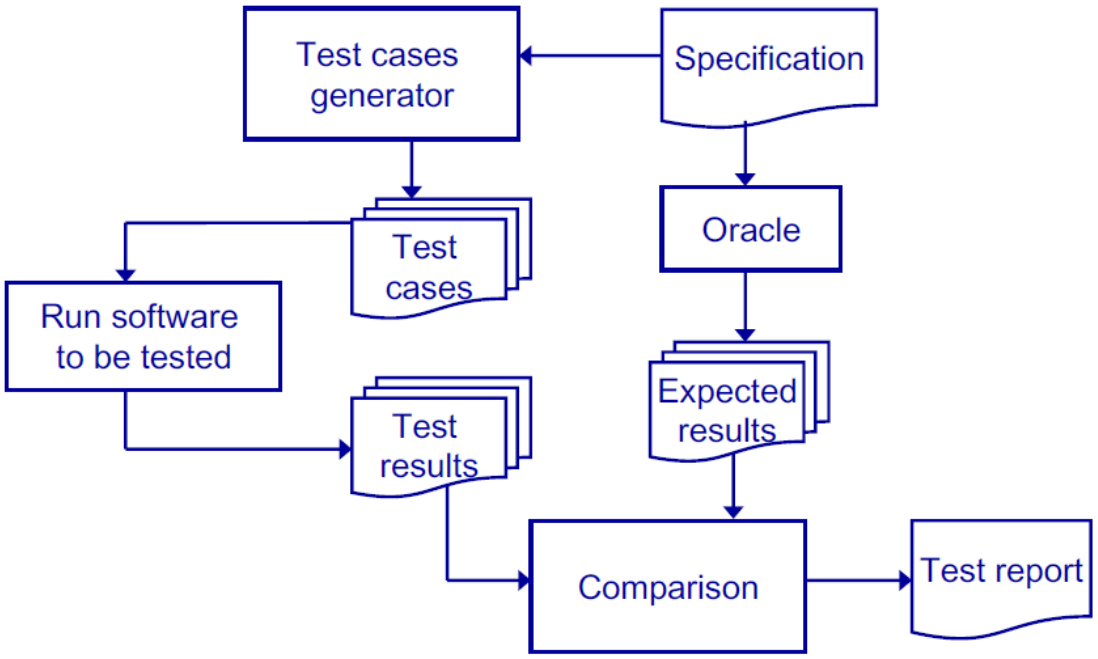

Figure 2: Test flow

2. Code review and unit test

This is the lowest level of testing as it involves inspecting single lines of code or functions. Considering that the first iteration of a code often contains some bugs which require to be fixed, such a testing can be performed by programmers themselves during development phase. This is the integrated approach that we have decided to follow in developing our project.

The standard way to test single lines of code or functions begins with the definition of a set of input values, assumptions about the internal state of the software and the expected outputs. The code is run, then the results re compared with the expected outcomes. If the results are equal to expected outcomes then the test was successful, otherwise the causes of errors must be investigated.

Let's test for example `cleanNum`, a function which removes any character which is not a number or decimal separator from a text string. This function uses `sub` a routine in `re` module and is described in the documentation as:

```
sub(pattern, repl, string, count=0, flags=0)

Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl. repl can be either a string or a callable; if a string, backslash escapes in it are processed.  If it is a callable, it's passed the Match object and must return a replacement string to be used.
```

In our software we have used `cleanNum` to automatize the removal of any alphabetic character, e.g. unit of measure (dB), that EpiCollect5 users might have wrongly typed into the noise pollution level data field, thus ensuring that this attribute is always a floating number.

In the following code, after importing the module `re` and defining the function `cleanNum`, we call it twice with different arguments: in the first case the input is `'70.8'`, a string that does not include alphabetic characters while in the second call we pass the string `'70.8db'`. Here the internal state of the system is not relevant, and we should expect to get the same outcome `'70.8'`, i.e. a string that contains only numerical digits 0-9 and the dot for decimal separator. Indeed, the code performs as expected:

```
In[1]: import re

In[2]: def cleanNum(noise):

    return re.sub('[^0-9.]', '', noise)

In[3]: cleanNum('70.8')

Out[3]: '70.8'

In[4]: cleanNum('70.8db')

Out[4]: '70.8'
```

This test was successful. However, if we give `cleanNum` as input a string containing a signed number, for example `'-70.8 db'` we get an output that is different from what we could expect: instead of obtaining `'-70.8'` we get `'70.8'`, that is we lose the information about the sign. In our specific case this is not a problem because noise pollution will never take negative values; but it is something that in other context might have led to an error in code.

```
In[5]: cleanNum('-70.8db')

Out[5]: '70.8'
```

A fix for this bug is to change the `cleanNum` function in the following way:

```
In[6]: def cleanNum(noise):

    return re.sub('[^0-9.-]', '', noise)
```

Another example of unit testing could be verifying the behavior of `create_engine`, a function imported from the package `sqlalchemy` which creates a new class “Engine” that manages the connection with some specified Database Management Service, in our case PostgreSQL. The standard calling form is to send the URL as the first positional argument, usually a string that indicates database dialect and connection arguments. For us:

```
In[1]: engine =
create_engine('postgresql://postgres:password@localhost:5432/se4g?gssencmod
e=disable')
```

A test in this case mean changing the input string that indicates database dialect and connection arguments or encryption mode.

3. Integration Testing

Moving up in the testing chain the next step is to verify the integration of different blocks of code. Large and complex software is often developed in parallel by different programmers or teams and testing that their codes integrate in an effective and efficient way is crucial to ensure the correct behavior of the overall software.

Our project required the integration of several technologies and modules, as documented in the Software Design Document and shown in the figure below:

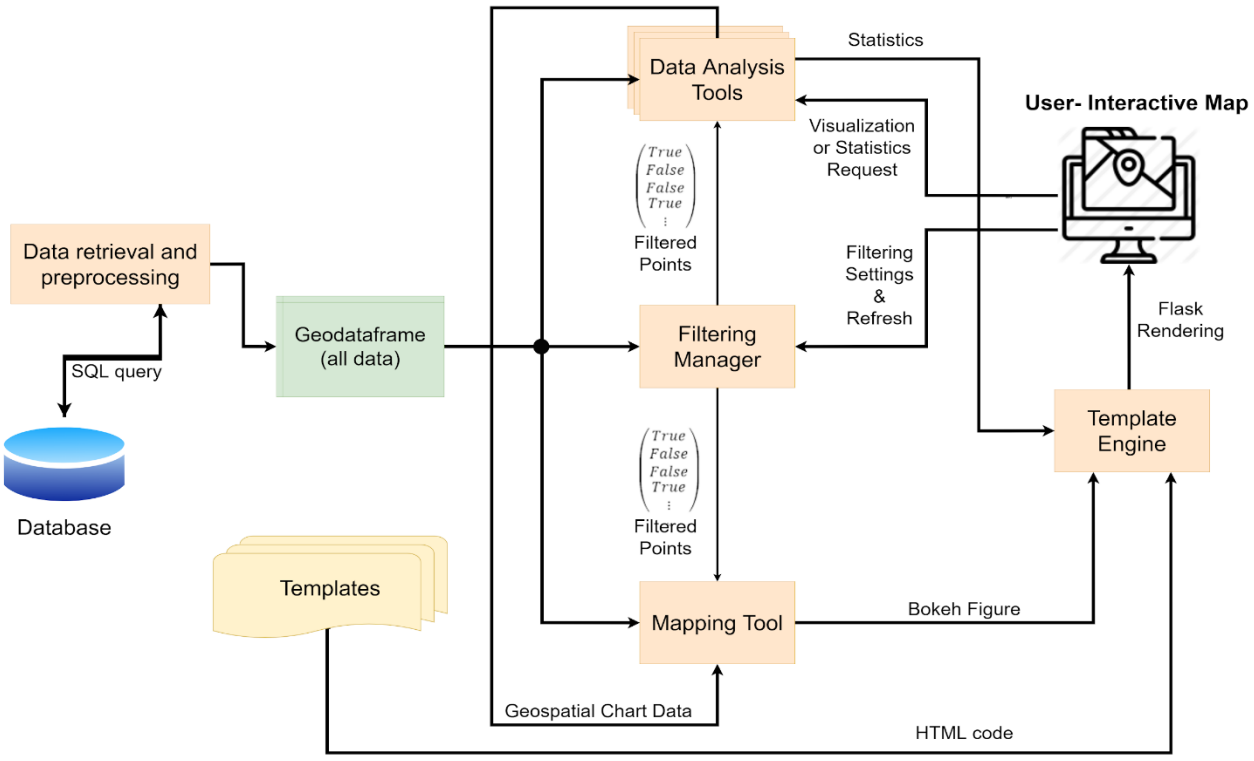

Figure 3: Summary of interaction between application server functionalities

Moreover, while every team member is familiar with the overall design of the web application and has a general understanding of how the different subsystems work, he or she is expert on one area: each of us was responsible for implementing a specific part of software, as summarized in next table. Therefore, testing that different subsystems integrate seamlessly was very important.

Name	Task/Activity	Description
Homeyra M.	Templates Engine	The Templates Engine will be developed in HTML and CSS and will be the structural backbone of the project. All of the system components will be located on top of this space and will be displaying information to the user at all times.
Alessandro Z.	DBMS	The Database Management System will be designed to work along with EpiCollect5. It will extract all the information from the API tool, pre-process it and store it in the most efficient manner. This database management system will interact with the different interfaces with a defined temporality.
Diego C.	Filtering Tools	The filtering tools will interact with the user in order to receive requests. This will work as an input-output friendly interface that will interpret the user needs and output the correct requests to be visualized with the interactive part of the software.
Rodrigo C.	Map Visualization	This task involves the implementation and development of the map visualization, which is the most important interaction tool that the web application will have. It will display all the information upon user request additional to the default visualization. It will be presented in an efficient, organized and attractive way.

Figure 4: Task and owners

Integration testing was performed directly during the development stage when different parts of software written by the four of us had to be integrated. An example of integration of different technologies is the extraction of data from EpiCollect5 – using available APIs – followed by the creation of a database table in PostgreSQL and the copy of those extracted data into the DBMS.

4. System Testing

System testing is the most abstract level of software verification. If it succeeds, then the software is passed to the client for the final acceptance test. System testing typically means to verify that the software as whole performs nominally as well as to validate it against the requirements as specified in the Requirement Analysis and Specification Document (RASD)

In our case system testing can be declinate in checking that the web application we developed fulfills its purpose of informing communities potentially affected by environmental noise pollution (ENP) and allowing users to contribute to ENP mapping. More precisely, this means to test the application against the use cases contained in the RASD, for example: [UC3] users can obtain the details about each measurement points through the interactive map. In this specific case the inputs are the noise pollution data downloaded from EpiCollect5, the state of the system is the database infrastructure, a base map and pins in the position where data were collected, while the output might be a popup window showing detail information about the point the user has selected.

5. References

E. Di Nitto, “Software Engineering for Geoinformatics A.Y. 2019-20 - Lecture 11”.