Appendix C

Notes on Quality Assurance and Control

In general, the Software Quality Assurance Plan is created before any quality assurance activities are carried out. In fact, the description of these activities and how are they deployed, is precisely the type of information that is needed in advance. Quality Assurance then refers to the process of measuring, analyzing and reporting back these control activities. We will assume for the sake of simplicity that quality will deal generically with "defects". These defects are related with three other components. An "error" is a human action that derives into an incorrect result. A "fault" is an incorrect step in a software product. A "failure" is the inability of a system to perform what is expected. As a consequence, we can establish a relation among these four concepts stating that "errors" are the cause of "faults" appearing in software products, and the combination of one or more failures may result in a "failure" in a software product. All three entities are contained in the concept of "defect". Software quality engineering thus relates to the procedures to address "failures".

1. Prevention, reduction and containment

There are numerous QA measures that can be conceived and deployed. But a good way to grasp how are they organized is to divide them depending on the type of action that they address. QA measures are typically divided in the following three categories:

- Defect prevention: These measures are designed to avoid potential sources of errors. When removing the source of errors, as a consequence the number of faults is reduced, and as consequence the number of failures. What kind of error sources are identified? Although each project is different, there are some conventional sources that are typically addressed by these measures. For example, make sure the development team have a unique view of the architecture of the product, they agree on the vocabulary of the application, they identify and correct ambiguities in the design architecture, etc.

  These types of measures are assuming that certain procedures decrease the probability of a person making an error, and therefore, by detecting the problem at its root cause, the cost of the defect is significantly decreased.

- Defect reduction: This is the most intuitive QA type of measures. A defect is detected, and a procedure is put into place to remove it. The main difference with the previous one is that the faults are removed from the software product *only after being detected*. This

implicitly means that a system needs to be conceived to make sure the search for faults is as exhaustive as possible. Detecting faults can be done automatically, but also manually through conventional inspection procedures. These inspections should include code, architecture and design documents.

- Defect containment: These techniques are designed to react *after* a failure has been detected and to reduce its effect to a minimum. Some software products include functionality specifically to treat failures and restore the product functionality as much as possible. The containment of failures, in some systems, may require even the use of physical objects. For example a product that controls traffic lights in the street, upon failure may defect to all lights being red to minimize damage.

When combining these three categories with the waterfall scheme for product development, we could assign focus preventing actions in the initial phases of software requirement and specification, Design and partially in coding. Defect remove would mostly focus on the coding and testing areas. Finally, defect containment would focus solely on the release and support phases of the project.

2. Defect prevention

The measures to prevent defects try to solve the problem at its origin. If humans behave in erroneous ways, we should provide measures so that they identify those situations and avoid them. The key observation for these measures is that the source of the errors are known in advance. How to know the sources of error? There are two main sources. The first one is based on the wealth of knowledge already existing on how to develop software products. There are typical situations that already occurred in so many products that can be considered perennial to a project. The second source is derived from the constant review and refinement steps within SQA. If in a project a new source of errors is detected, the adequate measures to avoid those errors need to be put in place and evaluated. A continuous refinement strategy is the best guarantee to maximize defect prevention.

The first measure to reduce defect prevention is *Education and training*. Education provides people with the right tools to detect error and avoid them. This training can be done at various levels:

- Product and domain specific level: People developing a product for a specific area should be as familiar as possible with that area. There are design decisions that can be heavily influenced by how the product is suppose to behave in a domain. There is a possibility that these details are not captured by any of the software requirements, thus, training within the domain is a valuable asset.
- Software development expertise: This is probably the most intuitive of all. The objective is to develop a software product, thus, expertise on development techniques are fundamental.
- Knowledge about methodology, technology and tools: Using the right tools and methodology have an impact on the quality of a product. But some of these topics are not trivial and require formal training for people to use them effectively.
- Development process knowledge: The development process should be understood in detail by all team members. Misunderstanding in some of the procedures will sure increase the possibility of errors occurring and deriving into failures.

Additionally from training and education, there are other techniques that hold some promise at preventing errors. They are normally known as “formal method” techniques. The approach consists

on treating a system as a collection of unambiguous specifications, capture its behavior with a model, and then *prove* that the system behavior is formally derived from the specification and its behavior. There are numerous tools used in software contexts to perform formal verification of software. The main obstacle is it high complexity and requirements. The tools need the description of a product in a very special format that is not trivial to derive. Additionally, for highly complex systems, these tools can only prove some basic properties and struggle to verify some more complex ones.

Other methods for defect prevention tackled directly the software development and provide support to detect ambiguous constructions, scarce documentation, good programming practices, etc.

3. Defect reduction

Defect reduction is a task that aims at covering 100% of all possible behaviour, knowing that reaching that mark is impossible. Thus, the strategy is to try to explore as much as possible of the entire behaviour space and hope that most of the defects are detected. The difficulty of these techniques is that to cover all aspects of a product, sometimes an equally complex testing infrastructure is needed.

The defect prevention techniques can be further sub-divided as follows:

- Inspections: Formal procedures in which a set of people plan and execute the inspection of portions of code to detect faults. There are detailed descriptions on how to proceed with these inspections.
- Testing: Perhaps the most intuitive of the defect reduction techniques. It requires the execution of software and sometimes the design of special additional modules. These tests are not exhaustive and have serious limitations. The main aspects to take into account when designing software testing is to know when testing is required, the type of faults that are found and how much testing is required.

4. Defect containment

Defect containment refers to the ability of a software product to detect a failure and maintain its effect within some reasonable bounds that still allow the product to proceed. The approach in this stage is different. Not all faults can be detected, thus, some additional measures need to be put in place for when a failure occurs, to minimize its impact. The techniques to contain the effect of a failure are usually referred also a s“fault tolerance”. There is quite extensive literature on how to design these measures and the effect that can achieve. For example, repeated executions are used for those blocks in which a different result can occur. N-Version programming relies on the use of parallel redundancy units, where N copies each with a different version of a program are executed. A local failure in one of these units is then compensated by the other versions.

5. How to handle defects

As an important part of QA is handling defects, one of the most important tasks is how defects are resolved. Whenever a failure is detected and corrected, there needs to be the right mechanisms to guarantee that indeed the failure no longer exists, and will never appear again in the product.

Defects are complex entities. The way to resolve a defect can be to rule it *not a defect*. Even though this may seem contradictory, it happens often in products in which some new behavior appears that was not expected, it is first ruled as a failure, but upon a closer inspection, it is ruled as not a defect.

The two most important activities related to defect resolution are: logging and tracking. The first task refers to report how the defect was discovered, under which conditions, in which environment, the steps taken to activate the failure, etc. Good defect documentation increases the chance of fully understanding the derivations and finding a solution faster.

Defect tracking, on the other hand, refers to how the actions *after* the defect has been detected. Which changes have been introduced? Why? Are they effective? If a set of multiple changes are required, this task becomes more complex as it may involve various members of the development team and they have to coordinate their efforts.

6. Verification and Validation

Software quality engineering is about the process to guarantee that a software product does the right things, but also to guarantee that the product does things right. These two innocuously similar statement in fact reflect two significantly different stages in SQA. By validation we refer to those tasks that make sure that a software product offers the functionality expected by the customers. On the other hand, by verification we understand those actions to prove that the tasks the product is suppose to execute, are done so correctly.

Typical validation activities are:

- System testing: The overall product provides what users expect.
- Acceptance testing and beta testing: the software and its performance is accepted by users.
- Usage-based statistical testing: The operating environment is simulated before product release.
- Software fault tolerance: The system is subject to conditions to provoke failures and check that the service is continued.
- Software safety assurance: The software is tested to prove that is accident free.

Verification tasks, on the other hand are more related to lower level details typically referring to simpler system tests, integration tests, component tests, and even coding and unit testing. There are some QA activities that deal with both, validation and verification. For example, system level test can be considered as verifying that a function expected by the customer is there, and is correctly implemented. Thus, these tests would be both for validation and for verification.