International Journal of Engineering, Science and Mathematics
Vol. 6Issue 5, September 2017,
ISSN: 2320-0294 Impact Factor: 6.765
Journal Homepage: http://www.ijesm.co.in, Email: ijesmj@gmail.com Double-Blind
Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

STUDY ON SOFTWARE DESIGN ISSUES

Vijay Kumar Sharma

Abstract

Adherence to a defined process or standards is necessary to achieve satisfactory software quality. However, in order to judge whether practices are effective at achieving the required integrity of a software product, a measurement-based approach to the correctness of the software development is required. A defined and measurable process is a requirement for producing safe software productively. In this study the contribution of quality assurance to the software development process, and in particular the contribution that software inspections make to produce satisfactory software products, is addressed.

To be more effective detecting software defects, not only should defect detection techniques be studied and compared but also the entire software detection process should be studied to give us a better idea of how it can be conducted, controlled, evaluated and improved.

This study addresses the requirement for an independently repeatable, scalable and substantially automated method for yielding semantics from computer code in a complete, unambiguous and consistent manner in order to facilitate, and make repeatable, verification based code inspection.
Algorithmic and heuristic techniques for investigating loop progress and termination are also introduced. Some of these techniques have been automated in supporting tools, and hence, the resulting defects can be repeatable identified. Throughout this study a strong highlighting is placed on describing implementable algorithms to realise the derivation techniques discussed. A number of these algorithms are implemented in a tool to support the application of the verification methods presented.
The techniques and tools presented in this study are well suited, but not limited to, supporting rigorous methods of defect detection as well as formal and semi-formal reasoning of correctness. The automation of these techniques in tools to support practical, formal code reading and correctness argument will assist in addressing the needs of trusted component technologies and the general requirement for quality in software.
The main part of this study is a systematic reconstruction of the B-method, using Generalized Substitution Language (GSL) and finishing with the Abstract Machine Notation (AMN).

Introduction to Software Fault Detection

Software fault detection is an central part of software development. The quality, the schedule, and the cost of a software product based heavily on the software fault detection process. In the development of software systems, 40% or more of the project time is spent on fault detection activities, such as, inspection, testing, and maintenance. In this thesis, maintenance means the fault detection activities after software releases, which include trouble shooting and debugging.
Software fault detection has proposed new inspection and testing techniques, and has studied and compared different inspection and testing techniques. However, most of the research has focused on a single inspection or testing technique. At most, different inspection or testing techniques were compared to determine which one detected more faults. To be more efficient in this area, not only the study of a fault detection technique itself is necessary, but also more importance should be put on the fault detection process in which these techniques are applied. How can we get more from the fault detection process by a significant selection and combination of the available fault detection

techniques? How can we calculate and improve the software's fault detection process? Largely, these questions are still open. Since there is no general advice on how to conduct the software fault detection process?

The initial aim of this work was to review software quality assurance within the development process and more specifically the area of software inspections, with a view to establishing areas of strengths and weakness and to identify areas of work, which would benefit from further research. A review of current study shows that software inspections have been successful in identifying errors within software products close to the point of their introduction, and therefore improving software productivity. However, software inspections are still very variable in application and effectiveness, depending greatly on the ability and experience of the individual inspector.

In proper verification and inspections, to find functional faults, the behavioural stipulation exhibited by a software artefact must be extracted from that artefact and compared to its intended stipulation. In this study, we present techniques for deriving semantic assertions from a software artefact. These semantics represent the abstracted behavioural stipulation required to support proper verification and inspection activities on that artefact. The repeatable techniques presented form a basis for reasoning about functional correctness and for assisting in the detection of functional faults.

The deduced semantics serve different purposes depending on the formality of the stipulation given. Although the semantic derivation techniques are manually applied to examples throughout this study, we place an emphasis on the definition of algorithms for extracting semantics that are agreeable to automation.

Hypothesis and Contribution

1. Generally, Current techniques of verification and fault detection are not repeatable.
2. All algorithms for the derivation of trusted semantic representations from program code exist and can be built-up.
3. The derivation of semantic information, including constants, from program code can support and improve the repeatability of verification and inspection tasks.

In this study addresses the issue of repeatability and practicality in verification and software inspection activities by proposing a proper and repeatable method of paraphrasing code into its semantics, and by defining techniques for using the extracted semantics in support of these activities.

Generally, the use of proper techniques, however insignificant, does have a positive effect on the reliability of the software in question, if other software engineering practices are not abandoned. Proper techniques research has also helped main stream software engineering to effect many changes for the better.

We are summarizing few definitions:

Software Product

A software product is any artefact created as part of creating and maintaining software, including computer programs, plans, procedures, and associated documentation and data.

Software Process

A software process is a set of activities, techniques, practices, and transformations that people use to develop and maintain software product.

Software Faults

A software fault is any flaw or imperfection in a software product, including both code and documentation.

Maintenance

Software maintenance is the activity required to keep the software system functioning properly or to add enhancements after software release.

Software Fault Detection

Software fault detection is the process of discovering software faults. It includes these activities: inspection, testing, and maintenance.

Software Development Processes

a) What is a process?

A software development process is a discipline by which control can be imposed on the design and development of a software product. A process defined as a set of partially ordered steps intended to reach a goal, this definition leads to an assumption that a process is a sequence of steps, with the components being called the process elements. A process step is defined as an atomic action of a process that has no externally visible substructure.

ISO12207 [International Organization for Standardization and International Electro technical Commission] defines a process as a set of interrelated activities, which transform inputs into outputs. It also notes that the term "activities" in this definition also covers the use of resources. The ISO definition appears to be wider in that it implies that processes can contain concurrent elements.

To control the development of software, states a process may contain the following elements:

• Prescriptive, requiring that the process should be performed in a particular way

• Proscriptive, which requires that a process should not be performed in a particular way

• Descriptive in that it describes the way in which development is actually conducted.

b) Who uses the process?

There are a number of motivations for controlling the process.

The project manager wishes to have improved estimates of costs and time-scales to prepare bids, to have a structure by which he/she can plan to make most efficient use of resources and subsequently monitor the process.

The software developer is looking for appropriate tools, techniques and environment to support the current activity and may need guidance on the activity and the context of that activity, together with other developers.

The customer (often represented by the quality assurance activity) needs to know that the project's development is meeting functional, as well as cost and time-scale requirements.

Considering all these different perspectives, we can see that there are many constraints and interactions associated with a process.

Software errors

The wear-out mechanisms that occur with hardware cannot occur in software, therefore all software errors are systematic errors. An important mechanism for this type of error introduction is human error, which results both from our nature as individuals and in the way in which engineers act and communicate in groups.

Software errors can also be introduced during any of the subsequent stages of development. Software errors, when detected, lead to re-work especially when detection occurs later in the process, i.e. testing. Then the re-work of the previous development stages is often at considerable expense and consequent re-testing.

A software error will only be manifested as a failure of the system when a particular input sequence, exercising the portion of the software code containing the fault, is presented to the system. This set of inputs may never occur during the operation of the software and the fault remains latent for the life of the system. Therefore, it is not possible to consider software in conventional reliability

terms, and this is confirmed by considering the profile of software integrity against time. They estimated the number of errors in a project would be:

$$Number\ of\ errors\ remaining = total\ errors - number\ of\ errors\ found\ by\ reviews\ and\ inspections - number\ of\ errors\ found\ by\ testing$$

The problem with this equation is that if we wish to know the number of errors remaining then we also need to know the total number of errors. They assume that the total number of errors in a project will be the same as that of other similar projects.

It should also be noted that the absence of errors found during quality and testing processes does not indicate that the product is free of error.

Software Quality Assurance Processes

Our concepts of integrity and reliability of a product result from the assurance of the quality of the product to a standard. Quality has been described as "The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs". Similarly, Grady describes quality as "Fitness for use, satisfying customer needs, and absence of faults". ISO9001 [International Organization for Standardization] provides a model for such a quality assurance standard, ISO9000-3 [International Organization for Standardization] and the Tick IT guide provide guidance for the application of the standard to software.

Alternative methods to the ISO standard are concerned with continuous improvement or capability measurement, i.e. as in the SEI Capability Maturity Model. This model grades the maturity of an organization's software development process into five levels:

Level 1: Initial - At this level the organization is ad hoc and often chaotic with few formalized procedures existing, and where they do exist, there is no management mechanism to ensure that they are used. Indeed, management may not understand problems and issues for a given project, such as lax change control, software installation and maintenance problems, or the need to integrate software tools.

Level 2: Repeatable - At this level a new manager has no orderly basis for understanding the organization's development projects. New team members have to 'learn the ropes' informally from other team members by observation of actual practice and so on.

Level 3: Defined - At this level measurement is focused on specific tasks to indicate the effectiveness of project organization.

Level 4: Managed - At this level, the cost of gathering data becomes onerous.

Level 5: Optimized

These reviews and inspections are part of a quality assurance process used for all types of engineering. However, their application to software requires particular care due to the abstract and non-tangible nature of the products and the inevitable complexity of software. Three techniques have been described for software quality assurance processes: Walkthrough, Review and Inspection.

Walkthroughs

A review process in which a designer or programmer leads one or more members of the development team through a segment of design of code, that he/she has written, while the other member ask questions and make comments about techniques, style, possible errors, violation of design standards and other problems.

Reviews

1. A proper meeting at which the initial or detailed design of a system is presented to the user or any other interested parties for comment and approval.

2. The proper review of an existing or proposed design for the purpose of detection and solution of design deficiencies that could affect fitness for use and environmental aspects of the product, process or service, and/or for identification of potential improvements of performance, safety and economic aspects.

Software inspections

"A proper evaluation technique in which software requirements, design or code are examined in detail by a person or group of experts other than the author detecting faults, violations of development standards and other problems."

The standard for software inspections was described in Fagan's classic paper, which, although rather dated now, still provides the basis of much software inspection practice. He describes the need for improving techniques for ensuring quality in the production of software. He also states that inspections are proper, efficient and economical techniques of finding errors. His work follows the principals of statistical process control described by Damming to make improvements in the quality of the software produced. Software inspections are a technique where the software is examined in a proper process with a clearly defined series of operations to identify errors with the software.

Errors are properly identified during the fault-logging meeting, however these may have been found during the preparation stage. The solutions to the errors should not be discussed at the logging meeting. Errors are classified as missing, wrong or extra, with a consequence of major or minor, for example, a miss-spelt word could be classified as a minor, wrong error. Fagan describes how data from the inspection process can be used to determine the effectiveness of the reviewing process in finding errors, and to determine norms against which particular problems can be identified. In particular, he identified that the correct inspection rate was important so as not to skip detail, and not to lose productivity.

In defining error detection efficiency, he uses the formula

$$\text{Efficiency} = \frac{\text{Errors found by an inspection}}{\text{Total errors in the product before inspection}}$$

Here an important definition that a fault is an instance in which a requirement is not satisfied, linking the criteria for the successful conclusion of an inspection to meeting the requirements. He introduces the concept of feedback, where the author of the inspected item can ask questions of the inspectors.

Mixed teams of inspectors were used including representatives from systems, software, and test and product assurance. Only 16% of the inspection time was spent on the code, the remainder on inspecting the other parts of the development process. The fault density data from 203 inspections during three years experience at JPL is given below. Kelly noted that fault densities decreased exponentially as a result of correcting faults during the initial stages of a project and later development stages.

Defect Density	Major	Minor
Requirements(R1)	6.5	23.4
Architectural Design(I_0)	2.5	16.4
Detailed Design(I_1)	3.5	10.6
Source Code($I_{2)}$	1.1	11.5
Test plan	10.3	11.8
Test Procedures	6.4	13.0

Table: Software Fault density at NASA JPL

Kelly used curve-fitting techniques to develop an exponential model of faults per page as the project moves through its development stages:

$$y = 3.19\, e^{-0.61x}$$

Where x is defined by the inspection stage as:

X	1	2	3	4
Stage	R_1	I_0	I_1	I_2

Kelly did not address how applicable his model was to other organisations, so without further experimental evidence, his work is of little practical value. Weller has published inspection experience from Bull Information Systems in which he concentrates on the measurement of inspections, and notes that inspections instrument the software development process. A major problem he notes is disinclination of project managers to accept that engineers do any activity other than coding. This view, despite much evidence to the contrary, is persisting, as discovered recently by Hall and Wilson where quality was seen as “running interference to the development of the product".

REFERENCES

[1] A Discipline of Programming, E. W. Dijkstra, Prentice Hall, 1976.
[2] Software Development with Z, J. B.Wordsworth, Addison-Wesley, 1992.
[3] Software inspection: An effective verification process, A. Ackerman, L. Buchwald, and F. Leniski, IEEE Software, 6(3):31–36, May 1989.

[4] Design and code inspections to reduce errors in program development, M. Fagan, IBM Systems Journal, 15(3):182–211, 1976.

[5] Advances in software inspection, M. Fagan, IEEE Transactions on Software Engineering, 12(7), Jul 1986.

[6] Software Inspection, T. Gilb and D. Graham, Addison-Wesley, 1993.