UDC 004.415.5(075.8)

I. SLIZOVSKAYA

National Aerospace University Named After M.E. Zhukovsky, Ukraine

METHODS FOR IMPROVING QUALITY OF SOFTWARE PRODUCTS

This paper studies the principles of improving the quality of software, which were first suggested by David L. Parnas. The paper contains a description of the key methods for creating software architecture. The methods described allow simplifying procedures of controlling the quality of a software product. The paper also contains an analysis of the described methods, which details the concept of creating quality software for the purpose of its further development.

quality control, testing, decomposition, information hiding, inspection, tabular construction

Introduction

At every stage of IT technology development, a special attention has been paid to issues related to improving software quality.

Quality is understood to be subjective opinions of end users and is formed according to the extent to which software satisfies their demand. These subjective opinions relate to the functionality, usability, reliability, performance, and scalability of any piece of software, as well as its other characteristics. To satisfy customers' demand, software makers have to care about the quality of their products before the development actually starts. For this reason, quality assurance, error detection and correction are major players in the process of software development, without preventing a project from being on time and on budget.

The industry experience in creating software products demonstrated that the most serious problems in software development are related to incompleteness and contradictions in project documentation as well as project requirement management. Research shows that troubles with requirements create more risks than other problems in the software development process. Errors in requirements cause approximately one-third of all detected defects.

The process of managing the requirements of a software development project has been defined as a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the syste.

Requirements management is a relatively new term. It used to be called “requirements engineering [1]. The term “software engineering” was coined to suggest that those who design and build software should work with the professional knowledge and discipline that is expected of engineers in other fields. David L. Parnas is one of the grand masters of software engineering, whose academic research and industrial collaborations have exerted far-reaching influence on software design and development. His ground-breaking writings capture the essence of the innovations, controversies, challenges, and solutions of the software industry. Together, they constitute the foundation for modern software theory and practice. David L. Parnas has been developed principles and methods that are both of academic interest and applicable to real-world problems. Software Fundamentals – Collected Papers by David L. Parnas is a practical guide to key software engineering concepts. It introduces and explains such key topics as [2]:

– decomposition of software into components;

– information hiding as the basis for a modular program;

– abstract interfaces that provide services without revealing implementation;

– relational and tabular documentation construction;

– documentation-based software testing;

– software inspections.

Results of researches

Decomposition of software into components. Decomposition is a process of dividing software products into components that can be developed, analyzed, tested, and so on separately. Components interact with each other in accordance with previously defined procedures, specified by the interfaces. The advantages of this approach are the following.

- Software architecture allows changing different components and testing the components separately (provided that the interfaces remain unchanged). It is useful for lengthy development cycles as well as for projects with iterational life cycles (several releases of new versions with added new features).
- Clear-cut definitions of interfaces allow using automatic "black-box" unit-testing, which decreases manpower expenses in the area of software quality control.
- Since tasks are sent to developers as small components, requirements for skills, knowledge, and responsibility of every developer are less onerous; thus, the quality of a product can be expected to improve.
- Ready components can be used in new projects, thus making their repeated testing redundant.
- Decomposition required sufficient documentation and often allows detecting incompleteness and contradictions in the requirements that might result in a conflict when the components are assembled.

The other side of the above:

- Performing decomposition prior to starting development takes a qualified analysts, architects, and a great deal of time.
- The project depends on the right architecture having been selected; selecting the wrong architecture may result in a complete failure of the project, for example, when the architecture offers no possibilities for scalability or extension.
- New requirements, when made in the middle of the life-cycle, call for a new analysis and mapping requirement changes to changes in modules.
- Changes in interfaces result in a repeated architecture definition, which, in its turn, require additional expenditure and repeated testing.

Information hiding as the basis for a modular program. Information hiding is a consequence of decomposition [3]. The logic of a module remains closed within this module and there is no need for it to be known outside the module, since the interface interaction is defined beforehand. Special features of the method are following.

- A module's logic can be easily modified (for example, a more efficient algorithm can be implemented), without changing other interacting modules and tests.
- Module calls need additional, certain time-critical parts of a program cannot be consigned to a module (for example, in embedded systems, when resources are limited).
- Minimizing information exchange in the processes of writing requires intensive information exchanges when the application is running, and vice versa.
- Higher constructive flexibility requires more resources, and vice versa.

Abstract interfaces that provide services without revealing implementation. An abstract interface is an interface that represents many possible actual interfaces equally well, an interface that models some, but not all, of the properties of an actual interface, a proper subset of the set of assumptions in the actual interface.

If all the properties of the abstract system correspond to the properties of the real system, a great deal of information about the real system can be obtained through studying the abstraction. The abstraction provides less information, but it may appear more complex because it is described using unfamiliar notation. The results of the abstraction may be "reused." They apply in many situations. These situations share the abstraction and differ only in the things that are abstracted from. Abstractions can introduce restrictions but do so consciously.

Relational and tabular documentation construction. There are two aspects to abstract documentation:

- better design, which is easier to document;
- using mathematics, which is
  - more compact;
  - less ambiguous;
  - more useful than natural language.

The use of mathematics enables checking for completeness, checking for consistency, having a precise description, having a reviewable document, often simulating the system, basing the design on the document, and making optimizations to simplify the system.

The main purpose of using mathematics is automatic testing since a table record can be processed by specialized software. Below are the mathematics-based methods suggested by Parnas for recording requirements.

Step 1. Record the requirement as a

$$\left(\left(\exists i, B[i] = {}'x\right) \wedge \left(B[j'] = x\right) \wedge \left(present' = true\right) \vee \\ \forall i, \left(\left(1 \le i \le N\right) \Rightarrow B[i] \ne x\right) \wedge \left(present' = false\right)\right) \wedge \\ \wedge \left({}'x = x' \wedge {}'B = B'\right).$$

Step 2. Provide an expanded view of the requirement recorded as a mathematical formula. This method is not more formal or more difficult than the programming language. This means: “Set i to indicate the place in the array B where *x* can be found and set present to be true. Otherwise set present to be false.”

But there are some unclear points:

- What need be done if the array is of zero length?
- What need be done if *x* is present more than once?
- Is it allowed to change *B* or *x*?
- What does the “otherwise” mean: Does it mean that if something that need be done is not done, or if there is no place in the array where *x* can be found, or if there are many places where *x* can be found?

Step 3. Record the requirement as a table:

	x can be found in B	*x* can not be found in B
j =	place where *x* can be found in *B*	any number at all
present =	true	false

Step 4. Modify the table so that all the expressions conform to the rules of computer algebra:

	$(\exists i, B[i] = x)$	$(\forall i, \neg(B[i] = x))$
j’ |	$B[j'] = x$	true
present’=	true	false

The first method can be implemented in mathematics-based tools, but requirements recorded using this method are difficult for people to interpret.

The second method appears to be clearer but does not answer key questions.

The third method is clearer but does not answer one key question and cannot be implemented in reliable tools.

The fourth method is complete and could be processed by tools. It is, in theory, equivalent to the first, but in practice much better.

Table records are successfully used for documenting processes that can be expressed as a table of states and transitions [4], which allows considering a module as a finite state machine and applying the principles of finite state machine theory to it. One table may best serve one purpose, for example, communicating between people involved in specifying the program in question, while a different but equivalent table may best serve some other purpose, for example, communicating between other people or planning and designing the program. It is, therefore, often desirable and useful in practice to construct a table representing the specification for the program and then to transform that table into other equivalent tables. To create and use a mathematical record, it is necessary to:

– review existing Computer Algebra Systems and select the combination of tools that is most suited for our purposes;

– develop an automated method of checking that each tabular expression satisfies a specified restriction for all possible assignments of values to its variables.

Documentation-Based Software Testing. Software must be tested according to appropriate documentation. The documentation is assumed to be ready and complete by the time testing begins. David Parnas suggests that the following widely-known testing methods be used: Black Box Testing, Clear Box Testing, and Grey Box Testing [5].

The following method of calculating test volumes, sufficient to achieve the required degree of trustworthiness, is suggested:

1. Assume that the right input distribution is available. Tests selected randomly from this distribution will be used.
2. Let 1/h be the required reliability.
3. The probability of passing N properly selected tests if each test would fail with probability of 1/h will be: M = (1 - 1/h)N
4. M is the probability that a marginal product would pass a test of length N.

This method works reliably if the number of test cases is unlimited from the start.

Software Inspections. The procedure for inspecting software [6] consistently finds subtle errors in software. The procedure is based on four key principles:

- All reviewers actively use the code.
- Reviewers exploit the hierarchical structure of the code rather than proceeding sequentially through the code.
- Reviewers focus on small sections of the code, producing precise summaries that are used when inspecting other sections. The summaries provide the "links" between the sections.
- Reviewers proceed systematically so that no case, and no section of the program, gets overlooked.

David Parnas suggests the following inspection procedure:

1. Begin by identifying and listing desired properties.
2. Prepare questionnaires for the reviewers.
3. Prepare a precise specification of what the code should do.
4. Decompose the program hierarchically into parts.
5. Produce the descriptions required for the "display approach"/
6. Compare the "top-level" display description with the requirement specification.

Inspections are carried out by a sufficient number of experts specializing in the subject area. The experts must be familiar with the purposes and tasks of the project. Their responsibilities must be clearly defined and assigned.

Inspections have the following peculiarities:

- constant control of the development process at all of its stages;
- all deviations in the process are reported and corrected;
- external critical reviews lead to better results than internal reviews within a group of developers;
- a sufficient number of subject area experts are required;
- sufficient time is required to conduct inspections.

Conclusion

The approaches and methods, suggested by David Parnas, allow systemizing requirements, create architecture, and organize the software development process, so that the end product is of high quality. Using computer algebra allows automating the process the process of managing requirements for software products; this approach results in lower development costs and a shorter development time.

References

1. Peters, D.K., Parnas, D.L. Requirements-Based Monitors for Real-Time Systems // IEEE Transactions on Software Engineering. – February 2002. – Vol. 28, No.2. – P. 146-158.

2. David Lorge Parnas, P.Eng. Decomposition of Software Into Components // Faculty of Informatics and Electronics. University of Limerick. 2003.

3. Parnas, D.L. The Secret History of Information Hiding // Software Pioneers: Contributions to Software Engineering, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, Berlin – Heidelberg, 2002. – P. 399-409, ISBN 3-540-43081-4.

4. Baber, Robert L. Practical Guidelines for Constructing and Simplifying Tables for Finite State Machines (FSMs) and Trace Functions (TFs) // SQRL Paper, June 14. 2005.

5. David Lorge Parnas, P. Eng. Documentation Based Software Testing // Faculty of Informatics and Electronics. University of Limerick. 2003.

6. Parnas, D.L, Lawford, M. The Role of Inspection in Software Quality Assurance // IEEE Transactions on Software Engineering. – Guest Editor's Introduction. – August 2003 – Vol. 29, No. 8. – P. 674-676.

Поступила в редакцию 22.02.2007

Рецензент: д-р техн. наук, проф. В.М. Илюшко, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Харьков.