Assuring Dependability of Software Reuse: An Industrial Standard

Fevzi Belli(✉)

Faculty of Computer Science, Electrical Engineering and Mathematics,
University of Paderborn, Paderborn, Germany
belli@upb.de

Abstract. Whereas a software component may be perfectly suited to one application, it may prove to cause severe faults in other applications. The pre-standard IEC/PAS 62814 (*Dependability of Software Products Containing Reusable Components – Guidance for Functionality and Tests*), which has recently been released, addresses the functionality, testing, and dependability of software components to be reused and products that contain software to be used in more than one application; that is, reused by the same or by another development organization, regardless of whether it belongs to the same or another legal entity than the one that has developed this software. This paper introduces into this pre-standard and give hints how to use it. The author, who chaired its realization that started in 2006, briefly summarizes the difficult process to bring the industrial partners with controversial interests to a consensus.

Keywords: Software reuse · Dependability · Test · Industrial standardization

1 Introduction

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. The vision of software reuse is as old as software itself – it was introduced already in 1968, in the year as the term "Software Engineering" was coined during the constitutional NATO conference in Germany [9].

Many efforts to reuse software have succeeded; there is an increasingly overwhelming number of success stories available in literature. Almost all major companies and institutions that deal with information & communication technology practice software reuse and report about their success, e.g., Nippon Electronic Company, GTE Corporation, Raytheon, DEC, HP, NASA, and many more [6, 7, 10].

Nevertheless, the promises of decreased cost and increased dependability, and thus decreased risks, are not always realized. The frightening news about recent disasters definitely caused by careless soft-ware reuse are still being warningly associated with and attributed to all software reuse. The failure of Therac-25 system, in which a software component was carried over from a previous version of an X-ray system, caused the machine to malfunction, resulting in the loss of several lives in a terrible way; patients were actually burned [4].

In the Ariane project, failure of a reused software component caused the loss of a rocket costing around half a billion dollars [5].

J. Cordeiro and M. van Sinderen (Eds.): ICSOFT 2013, CCIS 457, pp. 1–12, 2014.
DOI: 10.1007/978-3-662-44920-2_5

These recent disasters as a consequence of bad reuse on the one side and success stories as a consequence of good reuse on the other side are the key factors in deciding whether or not to enhance and sustain continued provision of reuse from a lucrative business perspective.

To sum up, before reusing a software component, the context and domain it was built for should be carefully compared with the context and domain it is intended to be built in, including the hardware and physical and organizational aspects [8]. Figure 1 depicts the elements of the reuse process which is the subject of this paper. It is evident that reusability is not a single feature of a components but a "bundle" of features (Fig. 2).

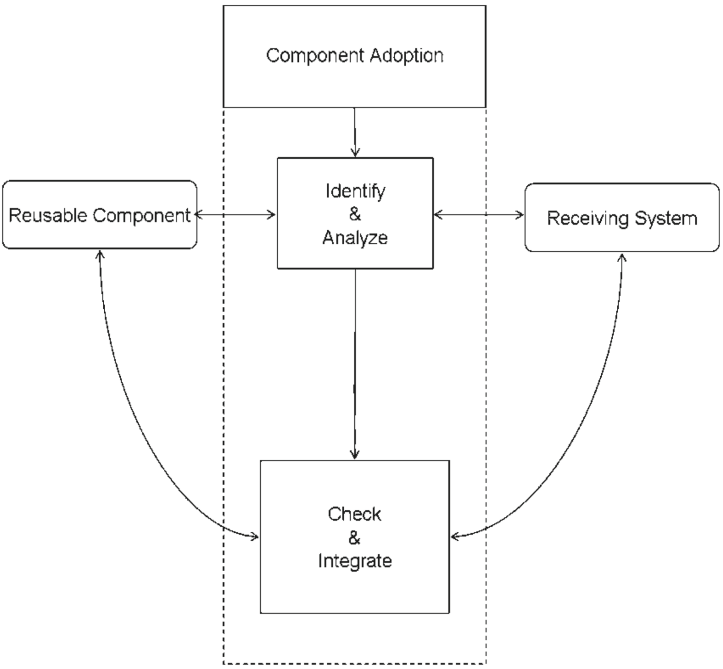

Fig. 1. Elements of the reuse process.

Standardization is the most efficient means to bring research, industrial, commercial, and consuming parties with different roles, but participating on the same objects and ethic objectives, e.g., to protect environment, to save resources, etc. Standardization helps with understanding and unifying the quality notion, also for reusing previously used products. Standardization helps also prevent legal problems that arise because reuse will be already practiced tentatively and insecurely.

This paper is on standardization of software reuse concerning its quality, test criteria etc., depending on the purpose of the software that will be reused.

The publicly available specification (PAS) *IEC/PAS 62814/Ed. 1: Dependability of Software Products Containing Reusable Components – Guidance for Functionality and Tests* is a pre-standard and addresses the functionality, testing, and dependability of

software components to be reused and products that contain software to be used in more than one application; that is, reused by the same or by another development organization, regardless of whether it belongs to the same or another legal entity than the one that has developed this software. IEC is the acronym of "International Electrotechnical Commission" that is the world's leading organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

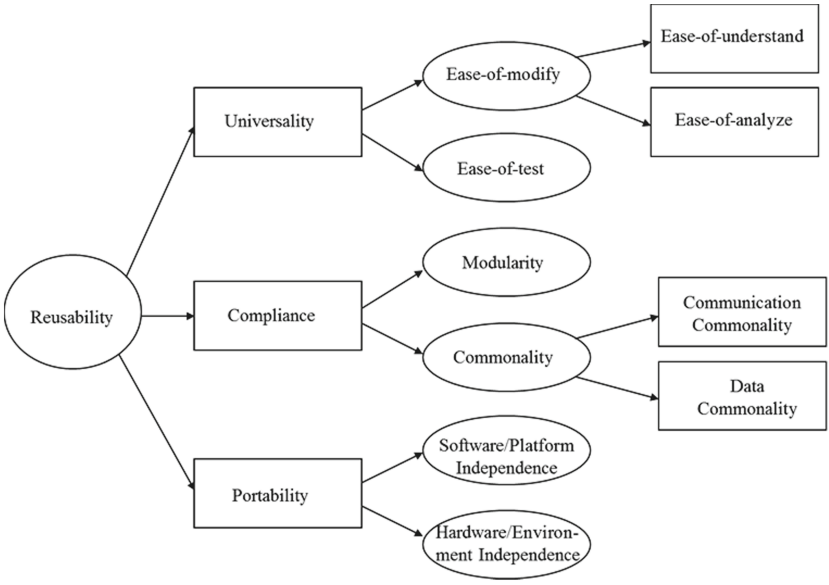

Fig. 2. Reusability characteristics.

The present paper gives an introduction into the PAS, which has been released in December 2012. The author chaired its realization that started in 2006 62814, and will give hint how to use it.

Next section clarifies terminology and discusses notions used in the practice. Section 3 introduces one of the most notable aspects of the PAS, that is, Reusability- & Dependability-Driven Software Development Technique. Recommended methods of validation, revalidation, and reliability of software reuse are summarizes in Sect. 4. Section 5 sketches the structure of the PAS, and explains and discusses its scope, objectives, and usage. Concluding remarks and future work are included in Sect. 6. AQ1

2 Notions and Practices of Reuse

Not each "copy and paste" action, which programmers do daily when they construct their programs, forms a software reuse that PAS 62814 has in mind. Also calling an internal or external function and even a remote-procedure call is not necessarily a reuse this PAS would regulate. All these examples suggest that the context and domain of the

called software does not change. Therefore, there is no need for them to consider this PAS and, for example, perform pre-store and pre-use activities that are described in PAS 62814 in detail.

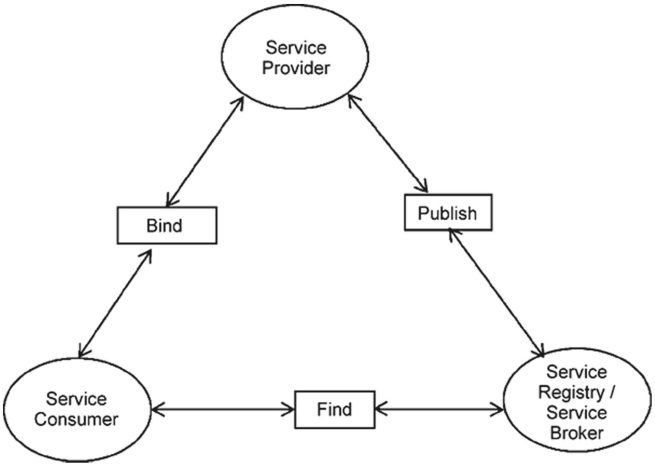

Fig. 3. Typical reuse by service-oriented architecture.

2.1 What Reuse Really Is

Using a service in a service-oriented (SO) landscape or in "Common Object Request Broker Architecture (CORBA)" is of more interest to this PAS because the context and domain of the software that delivers a service might change. Indeed, SO and CORBA are typical reuse constellations concerning constructing, offering, selecting, and validating services repeatedly. A service has to be registered and "published" before it will be offered. Infrastructural services are offered to realize a broker, etc. (Fig. 3) [1].

2.2 Where Reuse Will Be Practiced

Examples given above clarify that software reuse is not limited to the source or object code; it has, moreover, to consider all of the information that is related to the product generating processes, including also requirements, analysis, design, documents, and test cases apart from the code. Examples of well-known, widely accepted practices of software reuse are (Fig. 4) [11]:

- Component-based development (CBD): Building systems by integrating components that conform to system's specification.
- COTS integration: CBD using commercial components.
- Service-oriented systems: Building systems by linking shared services.
- Program generators: Embedding knowledge of a particular type of application to produce component(s) in that domain.

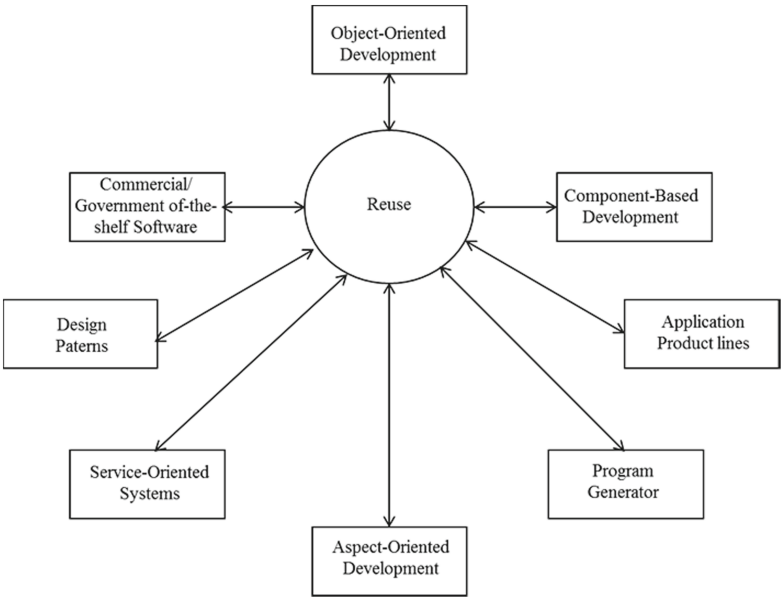

Fig. 4. Approaches to the reuse.

- Application product lines: Generalization of an application around a common architecture so that it can be used to produce different applications in different domains for different customers.
- Object-oriented programming: Implementing applications using "objects" that consist of data structures, methods (algorithms) and their interactions and computer programs
- Aspect-oriented software development: Weaving shared components into an application at different places when the program is compiled, if separation of concerns is feasible.

2.3 Software Reuse Has Many Faces

There is a great variety of reusing software, from ad hoc, unplanned to systematic. Following list attempts to structure this variety.

1. *Accidental* (ad hoc or *opportunistic*) reuse denotes reuse without strategy, typically reusing software components not designed for reuse.
2. *Systematic* (*planned*) reuse requires developing software components intended for reuse and/or building new applications from those reusable components, following a formal plan of product line.
3. *Adaptive* reuse uses previously developed software that is modified only for portability, e.g., a new application on a different operating system.
4. *Black-box* reuse uses unmodified software components, incorporating existing software components into a new application without modification.

5. *White-box* reuse modifies and integrates software (function) blocks into new applications.
6. *Vertical* reuse uses components in the same domain.
7. *Horizontal* reuse uses components in different domains.
8. *Internal* (*in-house*) reuse uses components developed within the company, or government unit.
9. *External* reuse uses components of another company, or government unit.

2.4 Software Reuse Has also Many Facets

The above discussion has identified practical and relevant kinds of reuse. A general taxonomy of software reuse is included in Table 1, which uses the following six aspects for a thorough, exemplary classification [2, 3]. Numbers in parentheses refer to the numbering used in the listing in Sect. 2.3.

- Reuse *assets* and *entities* can be product-oriented and, thus, concrete, such as components; they can also be ideal, such as concepts, ideas, algorithms, etc.
- *Domain scope* refers to application area (6 and 7).
- *Development scope* refers to origin of the component (8 and 9).
- Additional work required prior to reuse is referred to by *modification* (3, 4, and 5).
- Whether and which kind of work is to be done in performing reuse is a managerial aspect (1 and 2).
- Reuse *approach* is *compositional* if existing components are reused (such as the Unix™ shell); *generative* reuse requires application or code generators (such as Refine and Meta tool).
- *Direct* reuse approach requires no "glue code" that intermediates between the reusable component and the receiving system, *indirect* reuse necessitates an intermediate entity (Fig. 5).

Table 1. Summary of reuse classification.

Reuse asset	Reuse entity	Domain scope	Development scope	Modification	Management	Approach
Ideas, concepts	Architectures	Vertical	Internal	Adaptive	Accidental	Compositional
Artefacts, components	Requirements	Horizontal	External	Black box	Systematic	Generative
Procedures, skills	Designs			White box		Indirect
	Specifications					Direct
	Source code					
	Object code					
	Test cases					

Note that Table 1 shows the summary of the classification. It is possible to add further issues, for example, the issue of "Information to Reuse." It means that reused-based software development can be required for the complete specification of the reusable component.

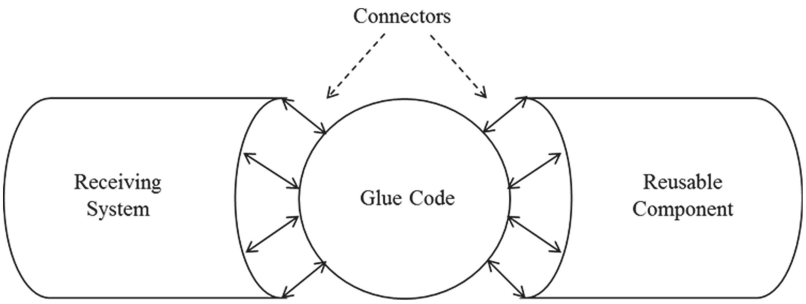

Fig. 5. Integration of reusable components.

3 Software Development Driven by Reusability and Dependability Aspects

Architecture is the key to software reuse. The architecture of a system commits its structure to combine the elements it is comprised of and their features, and relations among those elements.

Typical structures are hierarchical, centralized (star form), or decentralized (network form); relations are defined as consists-of or neighbored. Architectural elements can be event, state, or service-oriented.

It is important for reuse that the software architecture should allow a precise design and specification of interfaces and their dependability-critical features so that it enables evaluation, selection, acquisition, and integration of reusable components into the receiving system.

While planning substantial reuse of their software components, software engineers are often overly optimistic concerning how much reusable functionality can be achieved. Reuse is not a ultimate saver of costs, schedule, or dependability. Even COTS deployment often satisfies only less than 40 % of the functionality of an industrial application.

Also important is the addressing of the critical non-functional requirements, that is, dependability and quality, which certainly result in schedule and cost impacts, and, caused by poor dependability and reliability, maybe invoke severe safety and security risks.

Note that if the functional and interface requirements are not fulfilled, glue code and wrappers are to be planned, specified, designed, implemented, and carefully tested.

Dependability methodologies include application aspects and the organization of the reuse. Pre-store and pre-use characteristics should be met and the cases build-for reuse or build-by-reuse should be distinguished.

Another point covers validation and reliability aspects of the software. Also the assumptions and rules to improve software dependability are described and the hardware/software interaction is taken into account.

“Software-by-reuse” is the use of existing applications or their components to build new applications.

It is widely accepted and convenient to consider software reusability from the following viewpoints.

- Build-**for**-reuse enables planned production of reusable components.
- Build-**by**-reuse attempts planned production of systems using reusable components.

Both of these viewpoints focus on characteristics of reusability that are to be checked before storing the component and before reusing it in a new product.

Figure 7 depicts the coupling and orchestration of build-for and build-by aspects of reuse.

Following recommendations do not address only internal reuse; they can easily be adopted also for external reuse.

4 Validation, Revalidation, and Reliability of Software Reuse

Software reuse involves redesign, reimplementation, and re-testing. Redesign arises if the existing functionality does not fulfil the requirements of the new task because it requires reworking to realize the new function, and, prior to this, necessitates reverse engineering to understand its current functionality.

The design change leads to reimplementation. Exhaustive re-testing (as a kind of regression testing) is necessary to validate the functionality of the reused software in the new domain to determine whether or not redesign and reimplementation are needed.

Following undesirable events/situations, mostly caused by managerial misjudgment, negatively influence the dependability of software reuse:

- Failing to select the right component, or to favor the wrong selection criteria;
- Failing to justify and adjust the need for and/or extent of the modification of the selected component to fulfill operational or application requirements;
- Failing to justify and adjust the need for and/or extent of the maintenance of the selected component during operational stage.

To avoid such events/situations, redesign, re-implementation, and re-testing activities can be clustered in following groups:

- Redesign
  - Architectural design modification: Detection of architectural design part(s) to be modified, realization of the modification, re-validation of the entire architectural design;
  - Detailed redesign: Detection of design part(s) to be modified, realization of the modification, re-validation of the entire design;
  - Reverse engineering: Detection of the part(s) to be modified, which are not familiar to developers; understanding, modification, re-validation of the entire component;
  - Re-documentation: Detection of the part(s) to be modified, modification, re-validation of the entire document;
  - Re-implementation requires re-coding, code review, and unit testing (IEC 62628).

- Re-testing activities can be clustered in following groups:
  - Test re-planning
  - Test procedures to be altered
  - Re-integration testing
  - Re-release and re-acceptance testing
  - Test drivers/simulators to be altered
  - Test reports to be rewritten

Fundamental facts influence dependability, especially reliability when using commercially available components, e.g., COTS components for software development.

- Very often no source code is available, thus there is no way to correct a detected fault.
- This is a great restriction that prohibits application of the most widely used reliability models that require perfect correction of detected faults ("reliability growth models"; see, for example, AIAA R-013-1992, IEEE 1633-2008).
- If source code is available: Note that COTS software is no longer COTS after its source code is modified to correct a fault detected because the COTS supplier no longer maintains the documentation and source code (just as electronics equipment warranties are no longer valid after a seal is broken).
- Furthermore, the modifications can violate the original software design. From then on, modified COTS software is to be handled as an accidental reuse.

5 Structure of IEC/PAS 62814 and How to Use It

The international PAS 62814 introduces the concept of assuring reused components and their usage within new products. It provides information and criteria about the tests and analysis required for products containing such reused parts. The objective is to support the engineering requirements for functionality and tests of reusable software components and composite systems containing such components in evaluating and assuring reuse dependability (Figs. 6, 7). AQ2

Focus is on the dependability of software reuse and, thus, this document complements IEC 62309:2005-02 (Dependability of products containing reused parts – requirements for functionality and test), which exclusively considers hardware reuse. In addition to this previous, hardware-related IEC standard, the present PAS also crosses further, appropriate software-related standards to be applied in the development and qualification of software components that are intended to be reused and products that reuse existing components. In other words, this present standard encompasses the features of software components for reuse, their integration into the receiving system, and related tests. Their performance and qualification and the qualification of the receiving system is subject to existing standards, for example ISO/IEC 25000 and IEC 61508-3. The process framework of ISO/IEC 12207 on systems and software engineering and ISO/IEC 25000 on system aspects of dependability on software engineering apply to this present document.

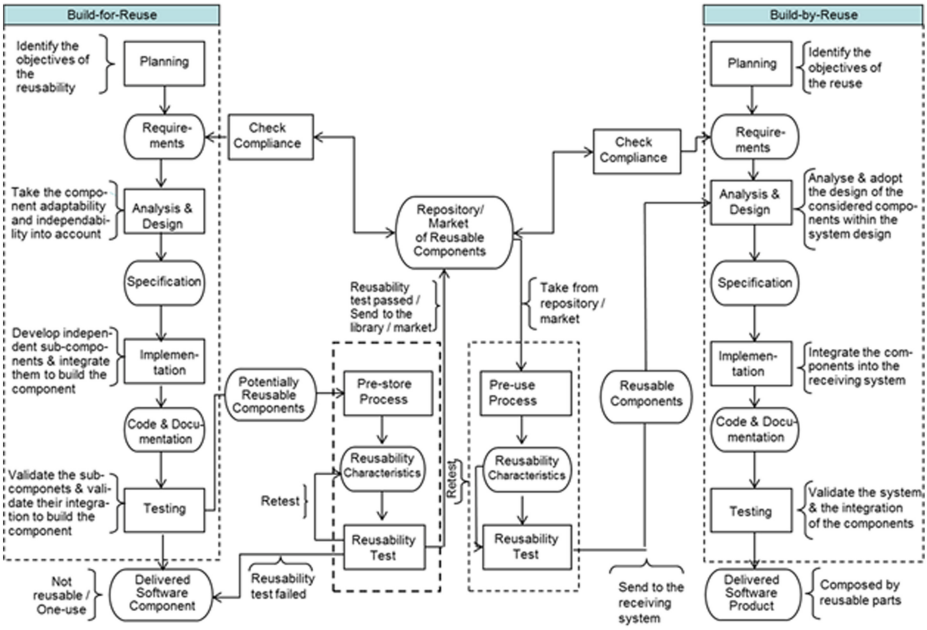

Fig. 6. Recommended framework of reuse.

NORMATIVE PART

1 Scope
2 Normative references
3 Terms and definitions and abbreviations
4 Dependability of software reuse methodology– reusability-driven software development
5 Software reuse dependability methodology applications
6 Software reuse assurance
7 Warranty and documentation

INFORMATIVE PART- ANNEXES

A General remarks on software reuse
B Qualification and integration of reusable software components
C Testing and integration of reusable software components – Issues for industrial best practice
D Example of software pre-use
E Influence of reused software to hardware components and products

Fig. 7. Structure of IEC/PAS 62814.

The purpose of IEC/PAS 62814 is to ensure through analysis and tests that the functionality, dependability and eco-friendliness of a new product containing reused software components is comparable to a product with only new components. This would justify the manufacturer providing the next customer with a warranty for the functionality and dependability of a product with reused components. As each set of hardware/software has a unique relationship and is governed by its operational scenario, the dependability determination has to consider the underlying operational background. Dependability also influences safety. Therefore, wherever it seems necessary, safety aspects have to be considered the way IEC 60300-1 addresses safety issues. This PAS can also be applied in producing product-specific standards by technical committees responsible for an application sector.

This paper could give only a brief introduction to the major aspects of IEC/PAS 62814. Due to lack of space nothing could be said about the informal part that comprehensively explains the methods and techniques for systematic reuse and its validation to assure dependability, and includes numerous examples from the practice and for the practice.

6 Concluding Remarks, Future Work

The most common form of reuse is using software developed for one-use in a new application, which is, accidental reuse. One of the major objectives of the present PAS 62814 is to warn the managers that this kind of unplanned reuse can be a potential minefield because it can cause the inheritance of all the problems of the pre-existing software in the reaping of only a few of its benefits. Many managers, while planning for software reuse, forget that both the reused component and composite system are to be tested in the new domain. Experience reports say that reusable software can cost 60 % more than one-use software, whereby a good portion of additional costs goes to testing.

This paper gave a brief introduction into IEC/PAS 62814 and which is a pre-standard, that is, it is not yet a standard. Further work and much energy are necessary to complete the work and produce a standard.

References

1. Belli, F., Linschulte, M.: Event-driven modeling and testing of real-time web services. J. Serv. Orient. Comput. Appl. **4**(1), 3–15. Springer, Heidelberg (2010)
2. Frakes, W. B., Terry, C.: Software reuse: metrics and models. ACM Comput. Surv. **28**(2), 415–435 (1996). http://dl.acm.org/ft_gateway.cfm?id=234531&type=pdf&CFID=65178775&CFTOKEN=89447410
3. Frakes, W.B., Kang, K.: Software reuse research: status and future. IEEE Trans. Softw. Eng. **31**(7), 529–536 (2005). http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1492369
4. Leveson, N.: Medical devices: the Therac-25. In: Appendix A in Safeware: System Safety and Computers, pp. 1–49, Addison-Wesley, Boston (1995). http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=84A18B532CF53C4AEA6F64AA6038BFEF?doi=10.1.1.39.704&rep=rep1&type=pdf

5. Lions, J.L.: Ariane 5 Flight 501 Failure (1996). http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html
6. Mathur, A.P.: Foundations of software Testing. Addison-Wesley Professional, Boston (2008)
7. Mohagheghi, P., Ict, S., Conradi, R.: An empirical investigation of software reuse benefits in a large telecom product. ACM Trans. Softw. Eng. Methodol. **17**(3), 13:1–13:31 (2008). http://dl.acm.org/ft_gateway.cfm?id=1363104&type=pdf&CFID=82907429&CFTOKEN=24134248
8. Mohammad, M., Alagar, V.: A component-based development process for trustworthy systems. J. Softw. Maint. Evol. Res. Pract. (2010) (Wiley InterScience, Published online), doi:10.1002/smr.472. http://onlinelibrary.wiley.com/doi/10.1002/smr.472/pdf
9. Naur, P., Randell, B. (eds.): Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany (1968). http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF
10. Orrego, A., Mundy, G.: SRAE: An integrated framework for aiding in the verification and validation of legacy artifacts in NASA flight control systems. In: Proceedings of the 31st Annual Intertnational Computer Software and Applications Conference. IEEE Computer. Press, New York (2007)
11. Sommerville, I.: Software Engineering. Addison Wesley Longman, Boston (2007)

Author Query Form

Book ID :	**330392_1_En**
Chapter No.:	**5**

Springer
the language of science

Please ensure you fill out your response to the queries raised below and return this form along with your corrections

Dear Author
During the process of typesetting your chapter, the following queries have arisen. Please check your typeset proof carefully against the queries listed below and mark the necessary changes either directly on the proof/online grid or in the `Author's response' area provided below

Query Refs.	Details Required	Author's Response
AQ1	Please check and confirm that citation of Section 0 have been changed as Section 3.	
AQ2	Please check and confirm the inserted citation of Fig. 6 is correct. If not, please suggest an alternate citation.	

MARKED PROOF

Please correct and return this set

Please use the proof correction marks shown below for all alterations and corrections. If you wish to return your proof by fax you should ensure that all amendments are written clearly in dark ink and are made well within the page margins.

Instruction to printer	*Textual mark*	*Marginal mark*
Leave unchanged	··· under matter to remain	
Insert in text the matter indicated in the margin		New matter followed by or
Delete	through single character, rule or underline or through all characters to be deleted	or
Substitute character or substitute part of one or more word(s)	through letter or through characters	new character or new characters
Change to italics	under matter to be changed	
Change to capitals	under matter to be changed	
Change to small capitals	under matter to be changed	
Change to bold type	under matter to be changed	
Change to bold italic	under matter to be changed	
Change to lower case	Encircle matter to be changed	
Change italic to upright type	(As above)	
Change bold to non-bold type	(As above)	
Insert 'superior' character	through character or where required	or under character e.g. or
Insert 'inferior' character	(As above)	over character e.g.
Insert full stop	(As above)	
Insert comma	(As above)	
Insert single quotation marks	(As above)	or and/or or
Insert double quotation marks	(As above)	or and/or or
Insert hyphen	(As above)	
Start new paragraph		
No new paragraph		
Transpose		
Close up	linking characters	
Insert or substitute space between characters or words	through character or where required	
Reduce space between characters or words	between characters or words affected	