Test-Driven Approach for Safety-Critical Software Development

Onur Özçelik [1,2*], D. Turgay Altilar[2]

[1] Scientific and Technological Research Council of Turkey, 41470 Kocaeli, Turkey.
[2] Department of Computer Engineering, Istanbul Technical University, 34469 Istanbul, Turkey.

* Corresponding author. Tel.: (90)0262-6753209; email: onur.ozcelik@tubitak.gov.tr
Manuscript submitted January 8, 2015; accepted June 14, 2015.
doi: 10.17706/jsw.10.7.904-911

Abstract: Nowadays software control large majority of systems that humankind use. Systems that software is used widely, such as transportation, military, medicine and avionics must be safe during operation. Fail in these critical systems may cause catastrophic results (i.e. loss of human life, loss or severe damage to environment or equipment etc.). In order to avoid failure on safety critical systems or at least mitigate risks detailed testing is required. Many safety critical systems are developed with sequential phases and tested with test last approach. But test last approach is not sufficient when requirements are unclear or changed. To overcome weaknesses of test last approach we propose test driven approach for safety critical software development, regulated by IEC 61508 standard. A systematic analysis of IEC 61508 software requirements against proposed approach showed that the approach directly supports some objectives and partially supports some of them. Supported objectives are safety requirement identification, simple and testable software design, verification and validation. In general introduced approach suits regulated software development well and this paper outlines these details.

Key words: Acceptance test-driven development, IEC 61508, safety-critical software, test-driven development.

1. Introduction

Increasing usage of software in safety-critical systems makes software more complex. Complex software brings more bugs and increases testing load. Although effective testing helps correct functioning of software systems the goal of testing is not to prove that no errors exist. The goal of testing is to prove that software may have some bugs.

No matter how much time and resources allocated for testing, it is impossible to test all inputs and corresponding outputs of software because of time and budget constraints of the project. These general testing principles are also valid for safety-critical software.

In this paper our goal is to define a test-driven approach for safety-critical software development and analyze the approach against the requirements of functional safety standard IEC 61508 [1].

IEC 61508 is a European standard that sets out a generic approach for all safety lifecycle activities for system comprised of electrical and/or electronic and/or programmable electronic (E/EE/PE) elements that are used to perform safety functions. The standard has seven parts. Parts 1-3 contains requirements of the standard and parts 4-7 are guidelines and examples for development.

The rest of the paper is organized as follows: In Section 2 we give some background and related works. In

Section 3 we introduce test-driven approach for safety-critical software development. In Section 4 we define research method of this paper. In Section 5 we compare our approach with regulations defined in IEC 61508 Part 3 and present the result of the analysis. Conclusions and future work can be found in Section 6.

2. Background and Related Work

2.1. Safety-Critical System

Safety Critical System (SCS) is a system whose failure or malfunction may result in death or permanent injury to people, loss of serious damage to equipment or harm to environment. Engineers design Safety-Critical Systems to lose less than one life per billion (109) hours of operation [2]. Several kinds of SCS exist.

- *Fail-operational systems* continue to operate when their control system fail. Elevators are an example of fail-operational system.
- *Fail-safe systems* become safe when they cannot operate. Railway signaling systems are examples of fail-safe system.
- *Fail-secure systems* secure the *system* which it belongs to when they cannot operate. Fail secure doors will lock during power failures.
- *Fail-passive systems* continue to operate when system fails. Aircraft autopilot is an example of fail-passive system.
- *Fault-tolerant systems avoid* service failure when faults occur. Nuclear reactors are examples of fault-tolerant system.

2.2. Safety-Critical Software

Before defining Safety-Critical Software we must give some definitions found in the literature. According to NASA Software Safety Guidebook [2],

Mishap is an unplanned event or series of events that result in death, injury, occupational illness, or damage to or loss of equipment, property, or damage to the environment.

Hazard is the presence of a potential risk situation caused that can result in or contribute to a mishap.

Safety-Critical Software is software that runs in Safety-Critical System and includes hazardous software (which can directly contribute to, or control a hazard). Software is also considered safety-critical if it controls or monitors hazardous or safety-critical hardware or software.

Software engineering for safety-critical software is generally hard. Engineers must have a clear understanding of the software's role in, and interactions with, the system. Engineering activities must comply with highly regulated international standards. These standards often make the development process very rigid, unable to accommodate changes, causing late integration and increasing the cost of development.

Current functional safety standards often describe software development as strict sequential process with distinct phases for requirements, architecture, design, coding and corresponding testing at increasing levels in the end.

Safety-critical software development process is generally based on the V model, which is illustrated in Fig. 1. This model shows sequential nature of development process. The sequential nature which puts testing activities to end is required for scheduling different phases and disciplines, and for certification purposes (which is mandated by relevant safety standards). When using this model testing at the end may cause problems like;

- Difficulty to address new and changed requirements during lifecycle
- Errors *found* in late stages will be costly to fix

- Late *integration* makes system testing more dependent to subsystems.

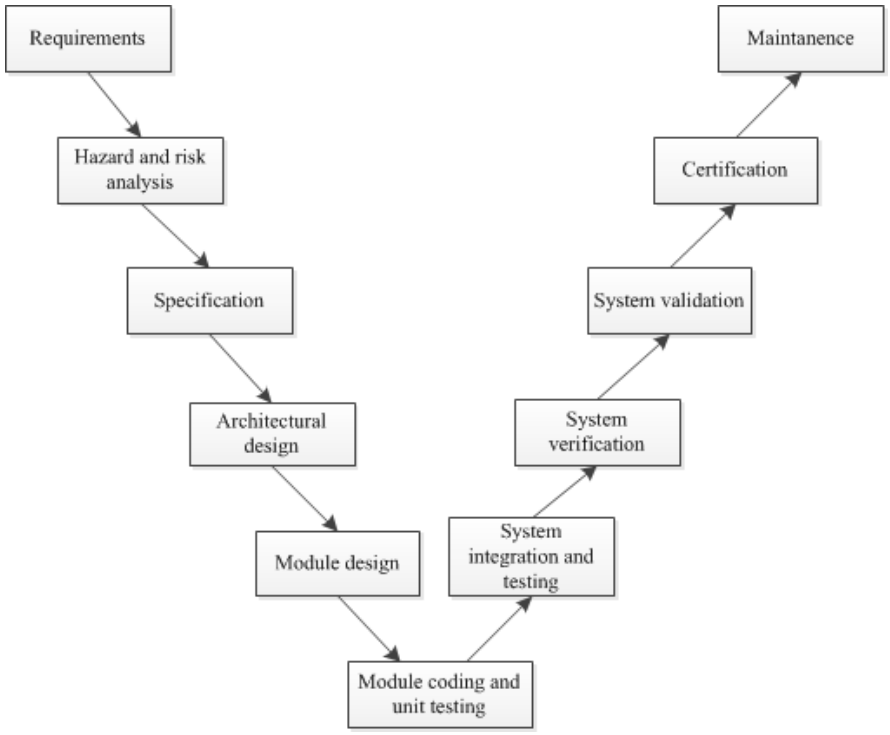

Fig. 1. V model.

2.3. Acceptance Test Driven Development

Acceptance Test Driven Development (ATDD) is a software development methodology based on collaboration between customer representatives, developers and testers [3].

ATDD's main idea is writing acceptance tests before developers begin coding. ATDD aids developers and testers to understands customer expectations and also allow customers to be able to express their needs in their own domain language.

Acceptance tests are specified in business domain terms when requirements are analyzed and before coding starts. Acceptance tests do not have predefined format but in generals tests conform to format given in Table.

Table 1. Custom Acceptance Test Format

Given (setup): A specified state of the system
When (trigger): An action or event occurs
Then (verification): The state of the system has changed or an output has been produced

2.4. Test Driven Development

Test Driven Development (TDD) is a software development methodology which follows repetitive iterations. Each iteration developer first writes an initially failing and automated test case that implements new functionality or improvement to existing functionality, then writes the minimum amount of code to pass that test, and finally refactor the code to acceptable standards [4].

Kent Beck invented TDD and stated in 2003 that TDD produces simple and effective software designs and inspires developer confidence. Although TDD gains general interest recently in software development community, it is related to test-first programming concepts of extreme programming begun in 1999.

TDD has some concepts like stubs and mocks. In the TDD terminology:

A stub object is an entity which unit test developer uses to break a dependency to another entity. Stubs are used to make unit tests run faster. Stubs do not used to verify unit test results [5].

A mock object is an entity which unit test developer uses to break a dependency to another entity and record some data to verify unit test results. Mocks are similar to stubs but they differ in unit test verification [5].

A literature survey on test driven approach for SCS development showed that most publications do not directly related with SCS development. Mostly publications have concerned SCS development and TDD separately. Some publications for SCS have concerned agile practices in SCS development and indirectly related with TDD.

Johnson and et al., made an analysis of agile practices in the context of software development for the European railway regulated by EN 50128 standard [6]. They concluded that agile practices support some of the objectives and requirements of EN 50128 but most practices must be tailored to fit in a regulated development environment.

In another study Ge and et al., propose an iterative approach for developing safety-critical software [7]. First they address the notion of up-front design in safety critical software development, and describe the characteristics of an up-front design that is minimal from perspective of achieving safety objectives. Then they present a way to develop both a software system and safety arguments iteratively.

The study by Fitzgerald and et al., focuses on a case study which an agile approach was implemented successfully in a regulated environment [8]. They concluded that the agile process as it has been adopted and augmented in a case study has worked very well in the regulated environment.

In another study Wolff describes a way to add the use of formal methods to the agile development process Scrum [9]. Wolff concluded that formal specifications can be integrated to an agile process.

Shrivastava and Jain study unit test case design metrics in TDD [10]. Although their work does not target safety-critical software development they propose useful metrics for TDD in general.

3. Test-Driven Approach for Safety-Critical Software Development

Our approach to safety-critical software development consists three phases. Widely used V model development life cycle is followed but some test supporting modifications to some phases are proposed also. Object oriented analysis/design (OOA/OOD) is used as a software design methodology.

3.1. Classification of Requirements

In phase 1, proposed approach starts with fault tree analysis (FTA) to analyze undesired states of system [2]. After performing FTA we try to classify software requirements as safety-critical or non-safety-critical. A technique described by Martins and de Oliveira is applied at this point [11]. They adapt a protocol which uses FTA to classify requirements as safety-critical. After safety-critical requirements are identified, the rest of requirements are classified as non-safety-critical.

3.2. Writing Acceptance Test Cases

In phase 2, acceptance test cases to all possible safety-critical requirements are written. Possible safety-critical requirements should be mapped to corresponding acceptance test cases. Proposed approach does not limit writing acceptance test cases to only safety-critical requirements. Also writing acceptance test cases to rest of the requirements improves testability of system.

3.3. Test-Driven Development and Test-Driven Design

In phase 3, for each acceptance test case, classes required to simulate that test case are identified. Then for each class that is identified unit tests and corresponding functionality implementations are written.

Therefore, general TDD workflow is followed and practices given in a guide by Hevery and et al., are applied [12]. While applying TDD, test-driven design is also taken into account. SOLID principles are used in order to produce more testable design [13]. SOLID principles that named by Robert C. Martin are five basic principles of object-oriented programming and design. The principles, when applied together will create a system that is easy to test, maintain and extend over time. Each principle and its relation to testability are elaborated in the following paragraphs.

Single Responsibility Principle states that a class should have only one reason to change. This principle forces us to give only one responsibility to a class. Instance of a class having single responsibility should do exactly one task, and there should be only one object in codebase that does the corresponding task. This can be further detailed as a method belongs to single responsible class should have one purpose and should be the only method in codebase that does this need. By adhering to this principle, the testability of design is increased by decreasing the number of tests that have to be written for testing same functionality on different objects. Moreover software design exposes smaller pieces of functionality that are easier to test in isolation and test cases become more cohesive and less coupled to other classes in software design.

Open / Closed Principle states that software entities (class, module, function) should be open for extension but closed for modifications. This principle forces us to design our objects in unchangeable manner. There should be no need to change that object to add new functionality. Instead, the object should be extended using polymorphism. Additional functionality can be introduced as needed, without changing the behavior of the object as it is already used elsewhere. By adhering to this principle, software concordance to mocking increases. Also rewriting tests is avoided. Existing tests for an object will be still valid for derived implementation.

Liskov Substitution Principle states that subtypes must be suitable for their base types. This principle ensures that any dependency can be easily replaceable by mocks and stubs.

Interface Segregation Principle emphasizes that clients should not be forced to depend on methods they do not use. If properly used, this principle assures that class interfaces will be small. Small interfaces make design more robust to change and improve testability of system under test by reducing complexity of one large interface. With this principle mock objects can be easily injected.

Dependency Inversion Principle (DIP*)* states that software entities (class, module) should depend on abstraction rather than concretion. DIP also states that abstractions should not depend on details instead details should depend on abstractions. This principle is used to make software entities loosely coupled to other entities. Loose coupling allows us to swap implementations without having to change the usage. For testing, this principle has a huge impact. It allows, once again, to easily injecting mock implementation instead of production implementation into object being tested.

4. Research Method

Since our approach is mostly related with requirements classification, software design, and development and testing we compare our approach with the regulations in IEC 61508 Part 3. Part 3 contains software requirements for safety-lifecycle phases and activities which shall be applied during design and development of safety-related software. We ask the following questions to compare our approach with IEC 61508 Part 3 software requirements.

1) Are there any requirements in IEC 61508 Part 3 that is supported by our approach?
2) Are there any requirements in IEC 61508 Part 3 that is partially supported by our approach?

The analysis was done for software safety requirements specification (sub-clause 7.2), software design and development (sub-clause 7.4), software aspects of system safety validation (sub-clause 7.7), software verification (sub-clause 7.9). For each requirement in selected subclasses we first analyzed whether the requirement was related/not related with our approach. Then for each related requirement, we analyzed

whether our approach supports or partially supports the objective of the requirement. Results are given in Table 2.

Table 2. Supported and Partially Supported Software Requirements of IEC 61508 Part III

	Supported Requirements	Partially Supported Requirements
Software safety requirements specification (7.2)	7.2.2.2, 7.2.2.3, 7.2.2.13	
Software design and development (7.4)	7.4.2.2, 7.4.2.3, 7.4.2.4, 7.4.2.6, 7.4.2.7, 7.4.5.3,7.4.7.1, 7.4.7.2	7.4.3.2, 7.4.4.13, 7.4.5.5, 7.4.8.1
Software aspects of system safety validation (7.7)	7.7.2.9, 7.9.2.9	7.7.2.7, 7.9.2.7, 7.9.2.10, 7.9.2.11

5. Result and Analysis

This section presents the results of this work, summarized in [14] Our approach employs a technique to derive software safety requirements from fault tree analysis of system (req. 7.2.2.2-3 are supported). Our approach pays importance to software testing (req. 7.2.2.13 is supported) and employs testable software design methodology (req. 7.4.2.2-7 (excluding req. 7.4.2.5) are supported).

We do not mention about software architecture in our approach but testable software design methodology assures testable software architecture (req. 7.4.3.2 is partially supported). We do not use a coding standard but test-driven development assures good programming practices like verification and testing (req. 7.4.4.13 is partially supported). We follow SOLID principles and test-driven development life cycle to make modifications to software (req. 7.4.5.3 is supported).

We do not mention about software integration testing in our approach but acceptance test cases may be used as a basis to integration test cases (req. 7.4.5.5 and req. 7.4.8.1 are partially supported). We use test driven development to verify (req. 7.4.7.1-2 are supported and req. 7.9.2.11 is partially supported) and acceptance test driven development to validate the software (req. 7.7.2.9 is supported and req. 7.7.2.7 and req. 7.9.2.10 are partially supported).

6. Conclusions and Future Work

This analysis shows that test-driven approach for safety-critical software development directly supports some software requirements and partially supports some software requirements of IEC 61508 Part 3. Although our aim is not to show that our approach is fully compatible with IEC 61508 Part 3 requirements, we show that our approach does not have serious conflicts with the standard.

In summary, we believe that our approach has a potential to make safety-critical software development more comfortable for software engineers by paying importance to test first development. . By using our approach software engineers will introduce fewer bugs to the software and/or will be more confident when working in safety-critical software projects. As a criticism of our approach we may say that simultaneous software design and development conflicts with sequential nature of safety-critical software development and makes our approach closer to agile practices. Since we do not support the results in this analysis with empirical evidence, these results can only serve as an initial guidance for a team of software engineers considering similar approach to safety-critical software development.

As a future work we plan to use our approach in safety-critical software project for Turkish State Railways. That project will be good case study to evaluate our approach in a real project.

Acknowledgment

This work has been partially funded by the Scientific and Technological Research Council of Turkey under YERLISINYAL09 project [14].

References

[1] European Standard BS EN 61508-3:2010, *IEC 61508*.

[2] NASA Software Safety Guide Book, NASA Technical Standard, NASA-GB-8719.13.

[3] Wikipedia Contributors, Acceptance Test-Driven Development. Retrieved December 2014, from http://en.wikipedia.org/w/index.php?title=Acceptance_test-driven_development&oldid=619209362.

[4] Wikipedia Contributors, Test-Driven Development, Wikipedia, The Free Encyclopedia. Retrieved December 2014, from http://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=636208361

[5] Osherove , R. (2009). *The Art of Unit Testing: with Examples in NET*, Manning Publications.

[6] Jonsson, H., Larsson, S., & Punnekkat, S. (2012). Agile practices in regulated railway software development. *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops* (pp. 355-360).

[7] Ge, X., F. Paige, R., & A. McDermid, J. (2010). An iterative approach for development of safety-critical software and safety arguments.

[8] Fitzgerald, B., Stol, K., O'Sullivan, R., & O'Brien, D. (2013). Scaling agile methods to regulated environments: An industry case study. *Proceedings of the 35th International Conference on Software Engineering* (pp. 863-872).

[9] Wolff, S. (2012). Scrum goes formal: Agile methods for safety-critical systems. *Software Engineering: Rigorous and Agile Approaches (FormSERA)*.

[10] Shrivastava, D. P., & Jain, R. C. (2011). Unit test case design metrics in test driven development. *Communications, Computing and Control Applications*, 1-6.

[11] Martins, L. E. G., & Oliveira, T. (2014). A case study using a protocol to derive safety functional requirements from fault tree analysis.

[12] Hevery, M., Wolter, J., & Ruffer, R. (2014). Guide to Writing Testable Code. Retrieved December 2014, from http://misko.hevery.com/2008/11/24/guide-to-writing-testable-code.

[13] Martin, R. C., & Martin, M. (2006). *Agile Practices, Patterns, and Practices in C#*, 1st ed., Prentice Hall.

[14] Scientific and Technological Research Council of Turkey. Retrieved December 2014, http://www.tubitak.gov.tr

D. Turgay Altilar received his Ph.D degree from Queen Mary, University of London, in 2002 He was involved with several EU projects related to parallel multimedia processing during his Ph.D research.

He is currently serving as an associate professor at the Computer Engineering Department of Istanbul Technical University. He is also a R&D coordinator of AGRIMONIS, a real-time data acquisition, management, decision support, prediction system, in AgroInformatics. Dr. Altilar's research interests are related to wireless sensor networks, cognitive radio, real-time systems, pervasive computing, parallel, distributed and grid computing. The current research interests are real-time software development, vehicular networks, routing protocol design in cognitive radio networks, data dissemination on multimedia sensor networks, MAC and routing protocol design for multichannel sensor networks, resource management grid computing and naturally tolerant parallel algorithm design and heterogeneous network based GPGPU computing.

Dr. Altilar has served as a technical program committee chair, technical program committee member, session and symposium organizer, and workshop chair in several conferences. Dr. Altilar is a member of IEEE.

Onur Özçelik was graduated from Computer Engineering Department of Gebze Technical University, Kocaeli, Turkey in the year 2007.

He has total 6 years of working experience as a software engineer since 2007. Currently, he works as a researcher at the Scientific and Technological Research Council of Turkey, Gebze, Kocaeli, Turkey. His current research interests are safety-critical software development, test-driven development and agile software design practices.