1. **INTRODUCTION**

I have completed my second internship in TAI (Turkish Aerospace Industries Inc.) between the time intervals 11.06.2012 and 17.08.2012. I worked at the Software Verification Team.

In the first two days of my internship, some presentations about the company, its work fields and future expectations were done. Contents of those presentations will be explained in the “Description of the Company” part in detailed.

After the presentations, I went to my department and software verification engineer Serkan DURAL gave training about our position in the department and what kind of works Software Department does. Basically, there are two units in the Software Department which are Software Development Unit and Software Verification Unit. In the Software Development Unit, the software, which will be used in the flight program, are built to satisfy some requirements which are prepared by software development engineers according to needs of whole project.

In the Software Verification Unit, the software built by software development engineers is tested whether it satisfies all requirements or not. To do this, whole code built by software development engineer is divided into small partitions. Then these partitions are tested individually by using some test codes written this department. These codes test all possible results in the requirements. Inputs and outputs coming from other partitions are sent manually. My position in the department is to write test codes and review test codes written by authors in the Software Test Unit and verify whether these codes satisfy all requirements or not. If not, test code is changed and published in new version by author. The works that I have done in the department will be discussed in the section “Work Done” in detailed.

1. **Information about the Company**
   1. **Company Name**

TAI (Turkish Aerospace Industries INC.) (Tusas-Türk Havacılık ve Uzay Sanayi A.Ş).

* 1. **General Manager**

Muharrem Dörtkaşlı.

* 1. **Company History**

TAI, the center of technology in design, development, manufacturing, integration of aerospace systems, modernization and after sales support in Turkey, was established on 15 May 1984.

Located in Akıncı-Ankara, TAI’s modern aircraft facilities, which cover an area of 5 million square meters with an industrial facility of over 200,000 square meters under roof, is furnished with high technology machinery and equipment that provide extensive manufacturing capabilities. Furthermore, in order to develop a ground for defense-aerospace skills, and create synergy among small to medium enterprises and universities, a significant part of engineering-based activities are conducted at the Techno Park located in the Middle East Technical University. The quality system of the Company meets the stringent world standards including NATO AQAP-2110, ISO-9001:2000 and AS EN 9100. TAI has a total of 3000 highly motivated employees, 1200 of whom are engineers.

The shareholders of the Company are; the Turkish Armed Forces Foundation, the under secretariat for Defense Industries and Turkish Aeronautical Association.

TAI, which participates in global-scale design and development programs, is also engaged in the design and manufacturing of structural components with leading international aerospace companies. With its proven experience, TAI is a uniquely qualified supplier for Aermacchi, AgustaWestland, Airbus, Boeing, EADS CASA, Eurocopter, Lockheed Martin, Northrop Grumman, MDHI, Sikorsky and many more.

Since its establishment, TAI’s employees, baring in mind their target to develop not only Turkey’s national power, but also the technological capacity that will support the military capacity, have been carrying out their activities to develop the necessary capabilities and products in order to meet the aerospace requirements of the Turkish Armed Forces with “indigenous” systems. To this end, TAI, in line with its vision and mission statements, has established a modern aerospace facility, and successfully realized the co-production of F-16 fighters, CN-235 light transport/maritime patrol/surveillance aircraft, SF-260 trainers, Cougar AS-532 general purpose helicopters. With its proven experience and know-how, TAI has improved its capabilities in the fields of design, production, modernization, modification and systems integration of fixed and rotary wing air platforms, unmanned aerial vehicles and satellite.

Being the main contractor of ATAK - Attack/Tactical Reconnaissance Helicopters Program, TAI will not only customize, but also produce and provide integrated logistics support of the helicopter in accordance with the user needs. TAI, which is the prime contractor of the Turkish Unmanned Aerial Vehicle (MALE) production program, is engaged in design and development of Primary & Basic Trainer (HÜRKUŞ) Aircraft. Baring in mind its target to provide the Turkish Armed Forces with indigenous systems, TAI continues its activities regarding the design and production of Turkish Primary and Basic Trainer Aircraft (HÜRKUŞ) and Turkish Indigenous Medium Altitude Long Endurance (MALE) Unmanned Aerial Vehicle (TIHA).

TAI, which actively participates in the custom satellite development program of Turkey, will be the local integrator company for the International Satellite Acquisition Programs. To this end, a new Satellite Assembly and Integration Test Facility will be built.

In addition to indigenous programs, TAI’s core business also includes modernization, modification and systems integration programs and after sales support of both fixed and rotary wing military and commercial aircraft that are in the inventory of Turkey and friendly countries. TAI is the prime contractor of the avionic modernization programs of the C-130 transport aircraft and T-38 training aircraft that are in the inventory of the Turkish Air Force Command (TuAF). Major modernization programs include Glass Cockpit modification of Turkish Black Hawk helicopters, electronic warfare retrofit and structural modifications on TuAF F-16s, Falcon Star and Mid Life Upgrade modifications of F-16’s that are in the inventory of the Royal Jordanian Air Force, modification and modernization of Cougar AS-532, modification of CN-235 platforms for MPA/MSA missions for the Turkish Navy and Coast Guard, modification of ATR-72 platforms for the Turkish Navy as well as structural modification and systems integration activities required for the conversion of B737-700 aircraft into AEW&C aircraft.

TAI participates, as a partner, in the global scale Joint Strike Fighter (JSF/F-35) and A400M design and development programs.

Being the shareholder of Airbus Military S.L., as National Industrial Institution, TAI has been participating in the design and development activities of A400M with the leading European aerospace companies namely; Airbus, EADS and FLABEL.

TAI, which also keeps on developing its capabilities in commercial aviation, has also become a full risk-sharing partner of Airbus in the A350XWB program for the aileron work package.

Furthermore, by utilizing capabilities of the Turkish Armed Forces’ Maintenance Centers, TAI also gives maintenance, repair and overhaul services to its customers.

Determined to keep abreast of global technological developments and secure its place among the major aerospace companies, TAI aims to lead Turkey to new horizons in aviation in the 21st century.

* 1. **Organizational Structure of the Company**

****

***Figure - 1: Organizational Structure of the Company***

1. **Information about the Erciyes (C - 130) Project**



The contract of C-130 Avionics Modernization (Erciyes) Program was signed between the under secretariat for Defense Industries (SSM) and TAI in December 2006. Under the program, the avionics modernization of a total of 13 TuAF C-130 aircraft (seven C-130E and six C-130B) will be realized.

The scope of Erciyes Program:

* Making the A/C ready for international flight rules like GATM, RVSM,
* Analog to Glass Cockpit,
* Unique and expendable, Mission Computer,
* Night Vision Goggle Compatible Lighting System,
* Design, manufacture and deliver the System Integration Laboratory after prototyping,
* Design, manufacture and deliver the Ground Mission Planning system after prototyping.

TAI will complete the modernization of two prototype aircraft, and will oversee the modernization of the remaining 11 aircraft at TuAF 2nd Air Supply and Maintenance Center. TAI is also to provide Integrated Logistics Support for the new systems.

When the Erciyes Program, which will be carried out in cooperation with TuAF 2nd Air Supply and Maintenance Center, is completed, the aircraft will not only will be compatible with international flight rules, but their flight safety will be increased as well. The aircraft will also become long-term supportable with the modernized systems.

The program is very important for TAI, given the complexity of the modernization and avionics configuration, which are to be carried out completely with national capabilities. In addition to the development of the indigenous mission computer, the software development operations, which will be installed to the computer, will also be realized by TAI.

Under the program, the technical documentation and user manuals will be prepared by TAI and delivered to TuAF. Furthermore, upon delivery of the aircraft, TAI will also provide an Integrated Logistics Support for 20 years.

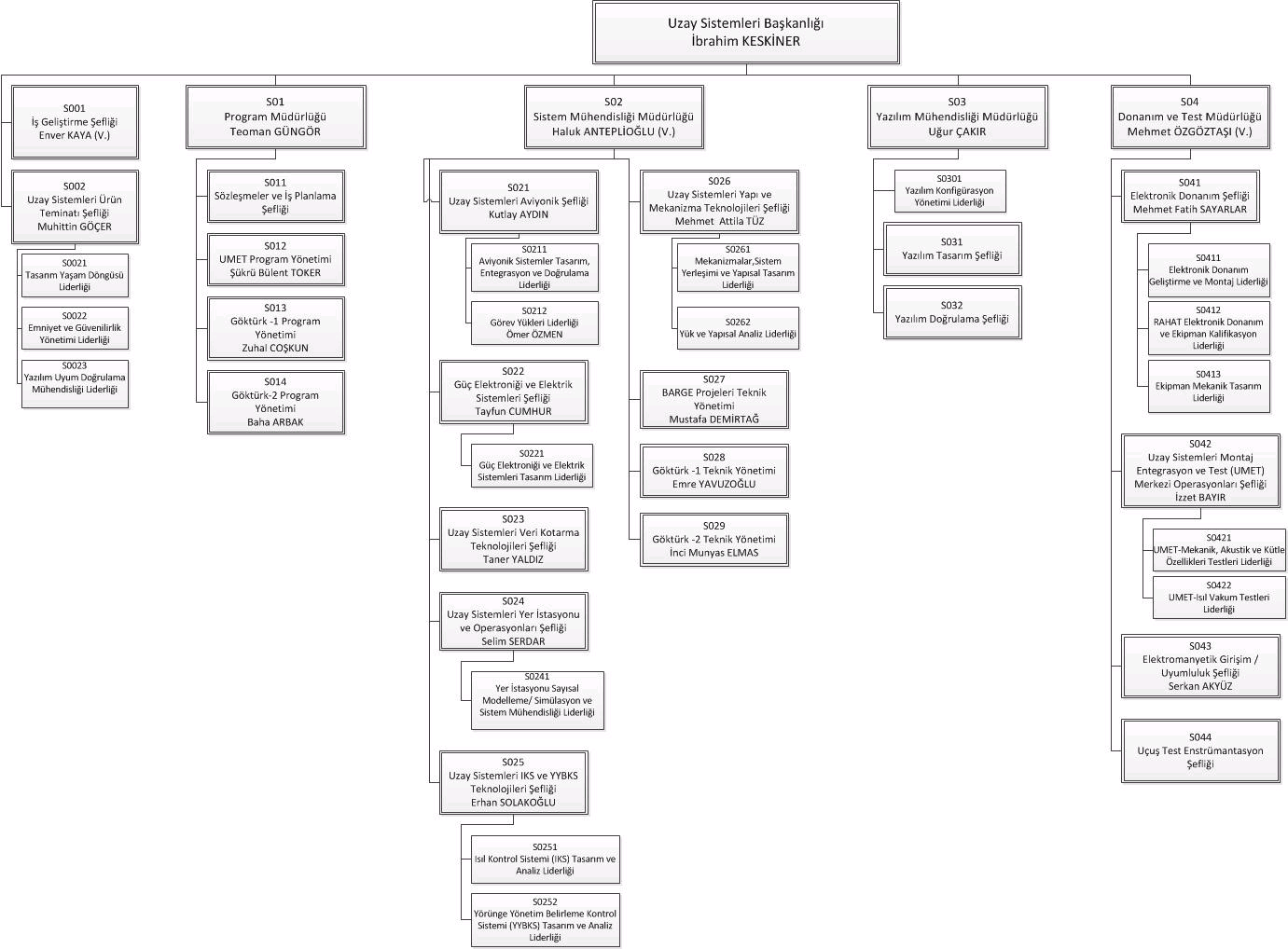
With the realization of the Erciyes Program, TAI will gain the capability to meet the modernization requirements of all countries that have C-130 aircraft in their inventories.

The first Turkish Air Force (TuAF) C-130 aircraft, the avionics modernization of which will be realized by TAI, arrived at TAI’s facilities on December 26, 2007. The aircraft, which took off from the 12th Transmission Base Commandership based in Kayseri, landed at TAI’s facilities, where it will go through series of avionics modernization activities.

With the arrival of the first C-130, the prototype period, which is planned to last 30-months, and comprises the modernization of the first two aircraft by TAI, was initiated. Under the Erciyes program, expected to last 56 months, TAI will oversee the modernization of the remaining 11 aircraft at TuAF 2nd Air Supply and Maintenance Center.

1. **Information about the Department**

During the internship that I have done at TAI, I worked at the Department of Software Engineering which is under the Department of Space Systems, in the Erciyes (C-130) Project as a software verification engineer.

******

***Figure - 2: Organizational Structure of the Department of Space Systems***

Department of Software Engineering has different sections that have different responsibilities.

|  |  |
| --- | --- |
| ***Organization*** | ***Responsibilities*** |
| Software Project Responsible | Responsible for the planning of the software |
| Systems Engineering Group | Responsible for the operational requirements |
| Avionics System Engineering Group | Responsible for the functional requirements |
| Development Team | Responsible for the development of the software (Writing the requirements, design, coding, integration) |
| Integration Team | Responsible for the integration and building the scripts |
| Verification Team | Responsible for the verification of the software |
| Software Configuration Management | Responsible for the configuration of the software |
| Software Quality Assurance | Responsible for the quality assurance of the software |
| AS and CVE | Responsible for the certification |

***Table - 1: Sections of the Department of Software Engineering***

* 1. **The Role of Software Development Team**

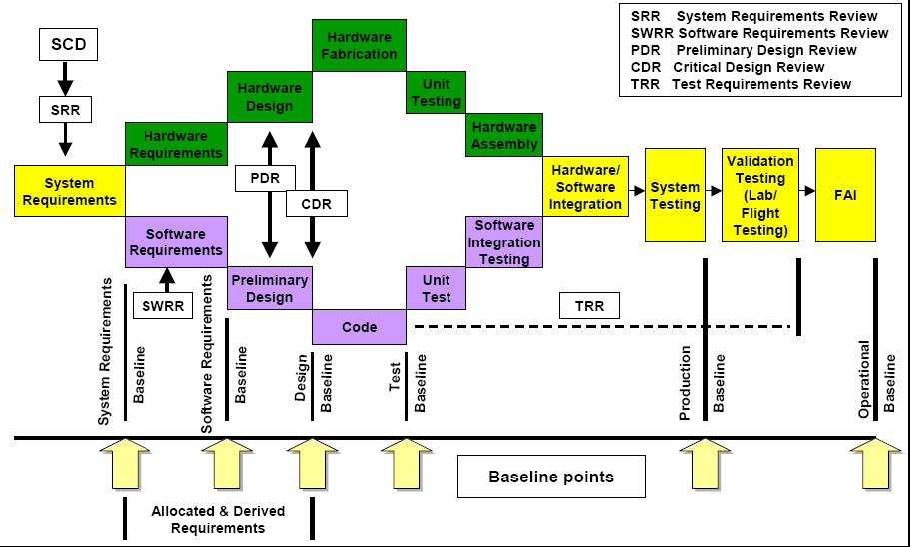
This part is responsible for the development of software project. Engineers in this department write codes according to some requirements. These requirements are prepared by software development engineers to satisfy the needs of whole project. However, the problem is that there may be some errors in codes which causes catastrophic

* 1. **The Role of Software Verification Team**

In this department, they first divide whole codes and design into small modules. Then, they divide them into small partitions. Then, they write test codes for each partition. Test codes must cover all conditions and results of requirements. The connection between partitions is enabled manually to one test code. Assume that one partition has 5 input ports and 5 output ports. Input ports come from other partitions and output ports go to other partitions. Since partitions are tested differently, software test engineers must assume that all other partitions work correctly. This means that input data come from other partitions must be given manually in test codes. Therefore, in the output ports, results are expected.

After sending data to the ports, output ports are read. If expected data in the output port does not occur, this means that there is an error in the code written by developer engineer. Since test code author must consider all possibilities, they test same ports from different input data.

Number of trials or number of different possible inputs depends on the level of importance of code. All partitions have level of importance and these are determined by system engineers. According to level of importance, there are five levels which are Level A, Level B, Level C, Level D and Level E. Level A is the most important level. This means that if an error occurs in the codes which are Level A, the result will be catastrophic. Therefore, software test engineer should consider all different possibilities here.Division of codes into these levels enables software engineers to write test codes in an efficient and clear way. If they would consider all partitions as Level A, they must have tested codes for all possible results. Then even not important and not possible results in real life would be tested. This scenario gives more complex, long and difficult to implement codes. Even the duration of the project work will increase.

1. **Information about Software Verification Process**

***Figure - 3: Software Development and Verification***

Verification is a technical assessment of the results of both the software development processes and the software verification process. Verification is not simply testing. Testing, in general, cannot show the absence of errors.

* 1. **Software Verification Process Objectives**

The purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes. Removal of the errors is an activity of the software development processes.

The general objectives of the software verification process are to verify that:

1. The system requirements allocated to software have been developed into software high-level requirements that satisfy those system requirements.
2. The high-level requirements have been developed into software architecture and low-level requirements that satisfy the high-level requirements. If one or more levels of software requirements are developed between high-level requirements and low-level requirements, the successive levels of requirements are developed such that each successively lower level satisfies its higher level requirements. If code is generated directly from high-level requirements, this objective does not apply.
3. The software architecture and low-level requirements have been developed into Source Code that satisfies the low-level requirements and software architecture.
4. The Executable Object Code satisfies the software requirements.
5. The means used to satisfy these objectives are technically correct and complete for the software level.
   1. **Software Verification Process Activities**

Software verification process objectives are satisfied through a combination of reviews, analyses, the development of test cases and procedures, and the subsequent execution of those test procedures. Reviews and analyses provide an assessment of the accuracy, completeness, and verifiability of the software requirements, software architecture, and Source Code. The development of test cases may provide further assessment of the internal consistency and completeness of the requirements. The execution of the test procedures provides a demonstration of compliance with the requirements.

The inputs to the software verification process include the system requirements, the software requirements and architecture, traceability data, Source Code, Executable Object Code, and the Software Verification Plan.

The outputs of the software verification process are recorded in Software Verification Cases and Procedures and Software Verification Results. The need for the requirements to be verifiable once they have been implemented in the software may itself impose additional requirements or constraints on the software development processes.

The verification process provides traceability between the implementation of the software requirements and verification of those software requirements:

The traceability between the software requirements and the test cases is accomplished by the requirements-based coverage analysis.

The traceability between the code structure and the test cases is accomplished by the structural coverage analysis.

Guidance for the software verification activities includes:

1. High-level requirements and traceability to those high-level requirements should be verified.
2. The results of the traceability analyses and requirements-based and structural coverage analyses should show that each software requirement is traceable to the code that implements it and to the review, analysis, or test case that verifies it.
3. If the code tested is not identical to the airborne software, those differences should be specified and justified.
4. When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means should be provided and their justification for satisfying the software verification process objectives defined in the Software Verification Plan or Software Verification Results.
5. Deficiencies and errors discovered during the software verification process should be reported to the software development processes for clarification and correction.
   1. **Software Reviews and Analyses**

Reviews and analyses are applied to the results of the software development processes and software verification process. One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness. A review may consist of an inspection of an output of a process guided by a checklist or similar aid. An analysis may examine in detail the functionality, performance, traceability and safety implications of a software component, and its relationship to other components within the airborne system or equipment.

* + 1. **Reviews and Analyses of the High-Level Requirements**

The objective of these reviews and analyses is to detect and report requirements errors that may have been introduced during the software requirements process. These reviews and analyses confirm that the high-level requirements satisfy these objectives:

1. Compliance with system requirements: The objective is to ensure that the system functions to be performed by the software are defined, that the functional, performance, and safety-related requirements of the system are satisfied by the software high-level requirements, and that derived requirements and the reason for their existence are correctly defined.
2. Accuracy and consistency: The objective is to ensure that each high-level requirement is accurate, unambiguous and sufficiently detailed and that the requirements do not conflict with each other.
3. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially, system response times and input/output hardware.
4. Verifiability: The objective is to ensure that each high-level requirement can be verified.
5. Conformance to standards: The objective is to ensure that the Software Requirements Standards were followed during the software requirements process and that deviations from the standards are justified.
6. Traceability: The objective is to ensure that the functional, performance, and safety related requirements of the system that are allocated to software were developed into the software high-level requirements.
7. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.
   * 1. **Reviews and Analyses of the Low-Level Requirements**

The objective of these reviews and analyses is to detect and report requirements errors that may have been introduced during the software design process.

These reviews and analyses confirm that the software low-level requirements satisfy these objectives:

1. Compliance with high-level requirements: The objective is to ensure that the software low-level requirements satisfy the software high-level requirements and that derived requirements and the design basis for their existence are correctly defined.
2. Accuracy and consistency: The objective is to ensure that each low-level requirement is accurate and unambiguous and that the low-level requirements do not conflict with each other.
3. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the software requirements and the hardware/software features of the target computer, especially, the use of resources (such as bus loading), system response times, and input/output hardware.
4. Verifiability: The objective is to ensure that each low-level requirement can be verified.
5. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process, and that deviations from the standards are justified.
6. Traceability: The objective is to ensure that the high-level requirements and derived requirements were developed into the low-level requirements.
7. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.
   * 1. **Reviews and Analyses of the Software Architecture**

The objective of these reviews and analyses is to detect and report errors that may have been introduced during the development of the software architecture.

These reviews and analyses confirm that the software architecture satisfies these objectives:

1. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes.
2. Consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow.
3. Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization and interrupts, between the software architecture and the hardware/software features of the target computer.
4. Verifiability: The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms.
5. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, especially complexity restrictions and design constructs that would not comply with the system safety objectives.
6. Partitioning integrity: The objective is to ensure that partitioning breaches are prevented or isolated.
   * 1. **Reviews and Analyses of the Source Code**

The objective is to detect and report errors that may have been introduced during the software coding process. These reviews and analyses confirm that the outputs of the software coding process are accurate, complete and can be verified. Primary concerns include correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards. These reviews and analyses are usually confined to the Source Code.

The topics should include:

1. Compliance with the low-level requirements: The objective is to ensure that the Source Code is accurate and complete with respect to the software low-level requirements, and that no Source Code implements an undocumented function.
2. Compliance with the software architecture: The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture.
3. Verifiability: The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.
4. Conformance to standards: The objective is to ensure that the Software Code Standards were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified.
5. Traceability: The objective is to ensure that the software low-level requirements were developed into Source Code.
6. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, fixed point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of uninitialized variables or constants, unused variables or constants, and data corruption due to task or interrupt conflicts.
   * 1. **Reviews and Analyses of the Outputs of the Integration Process**

The objective is to ensure that the results of the integration process are complete and correct. This could be performed by a detailed examination of the linking and loading data and memory map.

The topics should include:

1. Incorrect hardware addresses.
2. Memory overlaps.
3. Missing software components.
   * 1. **Reviews and Analyses of the Test Cases, Procedures and Results**

The objective of these reviews and analyses is to ensure that the testing of the code was developed and performed accurately and completely.

The topics should include:

1. Test cases: The verification of test cases is presented in paragraph 5.4.4.
2. Test procedures: The objective is to verify that the test cases were accurately developed into test procedures and expected results.
3. Test results: The objective is to ensure that the test results are correct and that discrepancies between actual and expected results are explained.
   1. **Software Testing Process**



***Figure - 4: Software Testing Process***

Testing of airborne software has two complementary objectives. One objective is to demonstrate that the software satisfies its requirements. The second objective is to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed.

The objectives of the three types of testing in the figure are:

• Hardware/software integration testing: To verify correct operation of the software in the target computer environment.

• Software integration testing: To verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.

• Low-level testing: To verify the implementation of software low-level requirements.

If a test case and its corresponding test procedure are developed and executed for hardware/software integration testing or software integration testing and satisfy the requirements-based coverage and structural coverage, it is not necessary to duplicate the test for low-level testing. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested.

To satisfy the software testing objectives:

1. Test cases should be based primarily on the software requirements.
2. Test cases should be developed to verify correct functionality and to establish conditions that reveal potential errors.
3. Software requirements coverage analysis should determine what software requirements were not tested.
4. Structural coverage analysis should determine what software structures were not exercised.
   * 1. **Test Environment**

More than one test environment may be needed to satisfy the objectives for software testing. An excellent test environment includes the software loaded into the target computer and tested in a high fidelity simulation of the target computer environment.

In many cases, the requirements-based coverage and structural coverage necessary can be achieved only with more precise control and monitoring of the test inputs and code execution than generally possible in a fully integrated environment. Such testing may need to be performed on a small software component that is functionally isolated from other software components.

Certification credit may be given for testing done using a target computer emulator or a host computer simulator.

Guidance for the test environment includes:

1. Selected tests should be performed in the integrated target computer environment, since some errors are only detected in this environment.
   * 1. **Requirements-Based Test Case Selection**

Requirements-based testing is emphasized because this strategy has been found to be the most effective at revealing errors. Guidance for requirements-based test case selection includes:

1. To implement the software testing objectives, two categories of test cases should be included: normal range test cases and robustness (abnormal range) test cases.
2. The specific test cases should be developed from the software requirements and the error sources inherent in the software development processes.

**Normal Range Test Cases:**

The objective of normal range test cases is to demonstrate the ability of the software to respond to normal inputs and conditions. Normal range test cases include:

1. Real and integer input variables should be exercised using valid equivalence classes and boundary values.
2. For time-related functions, such as filters, integrators and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.
3. For state transitions, test cases should be developed to exercise the transitions possible during normal operation.
4. For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.

One method is to test all combinations of the variables. For complex expressions, this method is impractical due to the large number of test cases required. A different strategy that ensures the required coverage could be developed. For example, for Level A, the Boolean operators could be verified by analysis or review, and to complement this activity, test cases could be established to provide modified condition/decision coverage.

**Robustness Test Cases:**

The objective of robustness test cases is to demonstrate the ability of the software to respond to abnormal inputs and conditions. Robustness test cases include:

1. Real and integer variables should be exercised using equivalence class selection of invalid values.
2. System initialization should be exercised during abnormal conditions.
3. The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.
4. For loops where the loop count is a computed value, test cases should be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.
5. A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.
6. For time-related functions, such as filters, integrators and delays, test cases should be developed for arithmetic overflow protection mechanisms.
7. For state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.
   * 1. **Requirements-Based Testing Methods**

Requirements-based testing methods consist of methods for requirements-based hardware/software integration testing, requirements-based software integration testing, and requirements-based low-level testing. With the exception of hardware/software integration testing, these methods do not prescribe a specific test environment or strategy.

Guidance includes:

1. Requirements-Based Hardware/Software Integration Testing: This testing method should concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. The objective of requirements-based hardware/software integration testing is to ensure that the software in the target computer will satisfy the high-level requirements. Typical errors revealed by this testing method include:

* Incorrect interrupt handling.
* Failure to satisfy execution time requirements.
* Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads and input power transients.
* Data bus and other resource contention problems, for example, memory mapping.
* Inability of built-in test to detect failures.
* Errors in hardware/software interfaces.
* Incorrect behavior of feedback loops.
* Incorrect control of memory management hardware or other hardware devices under software control.
* Stack overflow.
* Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
* Violations of software partitioning.

1. Requirements-Based Software Integration Testing: This testing method should concentrate on the inter-relationships between the software requirements, and on the implementation of requirements by the software architecture. The objective of requirements-based software integration testing is to ensure that the software components interact correctly with each other and satisfy the software requirements and software architecture. This method may be performed by expanding the scope of requirements through successive integration of code components with a corresponding expansion of the scope of the test cases. Typical errors revealed by this testing method include:

* Incorrect initialization of variables and constants.
* Parameter passing errors.
* Data corruption, especially global data.
* Inadequate end-to-end numerical resolution.
* Incorrect sequencing of events and operations.

1. Requirements-Based Low-Level Testing: This testing method should concentrate on demonstrating that each software component complies with its low-level requirements. The objective of requirements-based low-level testing is to ensure that the software components satisfy their low-level requirements.

Typical errors revealed by this testing method include:

* Failure of an algorithm to satisfy a software requirement.
* Incorrect loop operations.
* Incorrect logic decisions.
* Failure to process correctly legitimate combinations of input conditions.
* Incorrect responses to missing or corrupted input data.
* Incorrect handling of exceptions, such as arithmetic faults or violations of array limits.
* Incorrect computation sequence.
* Inadequate algorithm precision, accuracy or performance.
  + 1. **Test Coverage Analysis**

Test coverage analysis is a two steps process, involving requirements-based coverage analysis and structural coverage analysis. The first step analyzes the test cases in relation to the software requirements to confirm that the selected test cases satisfy the specified criteria. The second step confirms that the requirements-based test procedures exercised the code structure. Structural coverage analysis may not satisfy the specified criteria. Additional guidelines are provided for resolution of such situations as dead code.

**Requirements-Based Test Coverage Analysis:**

The objective of this analysis is to determine how well the requirements-based testing verified the implementation of the software requirements. This analysis may reveal the need for additional requirements-based test cases. The requirements-based test coverage analysis should show that:

* + - * 1. Test cases exist for each software requirement.
        2. Test cases satisfy the criteria of normal and robustness testing

**Structural Coverage Analysis:**

The objective of this analysis is to determine which code structure was not exercised by the requirements-based test procedures. The requirements-based test cases may not have completely exercised the code structure, so structural coverage analysis is performed and additional verification produced to provide structural coverage.

Guidance includes:

1. The analysis should confirm the degree of structural coverage appropriate to the software level.
2. The structural coverage analysis may be performed on the Source Code, unless the software level is A and the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences. A compiler-generated array-bound check in the object code is an example of object code that is not directly traceable to the Source Code.
3. The analysis should confirm the data coupling and control coupling between the code components.

**Structural Coverage Analysis Resolution:**

Structural coverage analysis may reveal code structure that was not exercised during testing. Resolution would require additional software verification process activity.

This unexecuted code structure may be the result of:

1. Shortcomings in requirements-based test cases or procedures: The test cases should be supplemented or test procedures changed to provide the missing coverage. The method(s) used to perform the requirements-based coverage analysis may need to be reviewed.
2. Inadequacies in software requirements: The software requirements should be modified and additional test cases developed and test procedures executed.
3. Dead code: The code should be removed and an analysis performed to assess the effect and the need for reverification.
4. Deactivated code: For deactivated code which is not intended to be executed in any configuration used within an aircraft or engine, a combination of analysis and testing should show that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated. For deactivated code which is only executed in certain configurations of the target computer environment, the operational configuration needed for normal execution of this code should be established and additional test cases and test procedures developed to satisfy the required coverage objectives.
5. **Work Done**

In my internship at Software Verification Team in TAI, I have done these jobs:

* Writing test codes.
* Test code review.
* SCR verification.

While doing these jobs, I have used these technologies:

* Tortoise SVN
* Crucible Dashboard
* Telelogic DOORS Module
* JIRA
* Jenkins
* Beyond Compare
  1. **Technologies Used**
     1. **Tortoise SVN**

Tortoise SVN is a free open-source client for the subversion control system. Engineers put the codes that they have written to the SVN Repository, and then they can read, change, or delete these codes. Tortoise SVN saves the all versions and makes seeing the all changes possible. By using Tortoise SVN; many people can reach to a file at the same time, people can reach to the old versions of a file and information about a file like writer and time of a version commit can be looked easily.

* + 1. **Crucible Dashboard**

Crucible Dashboard is used to do reviews. This platform reaches to the files in SVN repository. Test code writers create a review for their codes that are not mature, and select a moderator and any number of reviewers for their codes by using Crucible Dashboard.

* + 1. **Telelogic DOORS Module**

Telelogic DOORS Module is a database for the documents of the project. Requirements of the project like; SRS (Software Requirements Specification), IRS, OFP, FRS; are saved in this module. Engineers write their codes according to the requirements in DOORS Module. Furthermore, previous baselines of the documents can be reached by this module.

* + 1. **JIRA**

If it is needed to change a code that has been released, or there is mistake in SRS, an SCR (Software Change Request) must be opened. JIRA is the tool that is used to open these SCRs.

* + 1. **Jenkins Dashboard**

In order to check that test codes work properly and all requirements have been covered, tests are sent to Jenkins Dashboard and run there. This independent machine runs all tests and gives test results.

* + 1. **Beyond Compare**

This tool helps to see the differences between two documents. DOORS and SVN documents can be compared easily by using this tool.

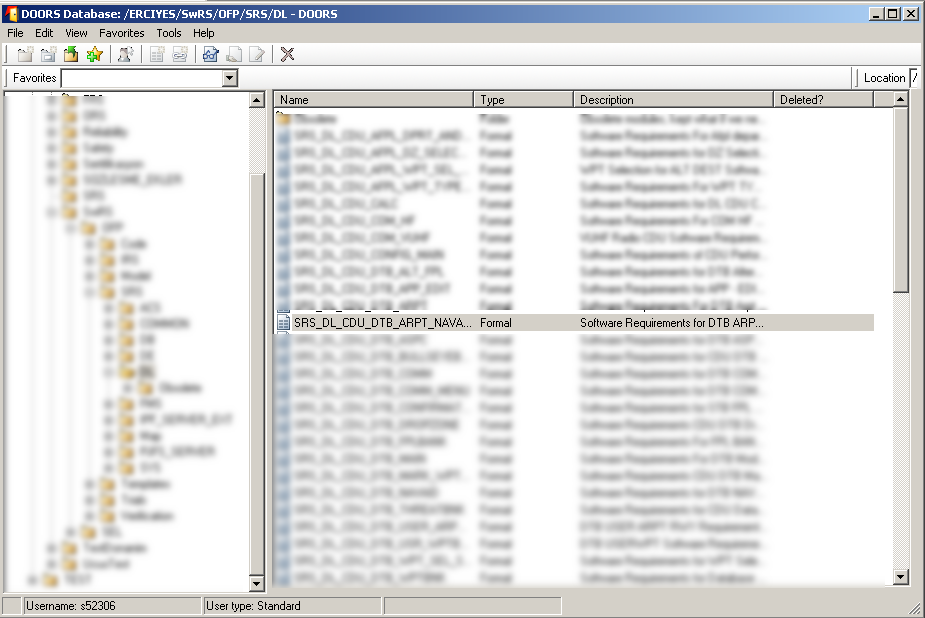
* 1. **Writing Test Codes**

During my internship, I mostly wrote test codes and the purpose of most of my test codes was testing some pages of CDU (Cockpit Display Unit). CDU is the one of the tools in the aircraft through that flight team can do data entrance, flight management function control, sub-system control and system information watching.



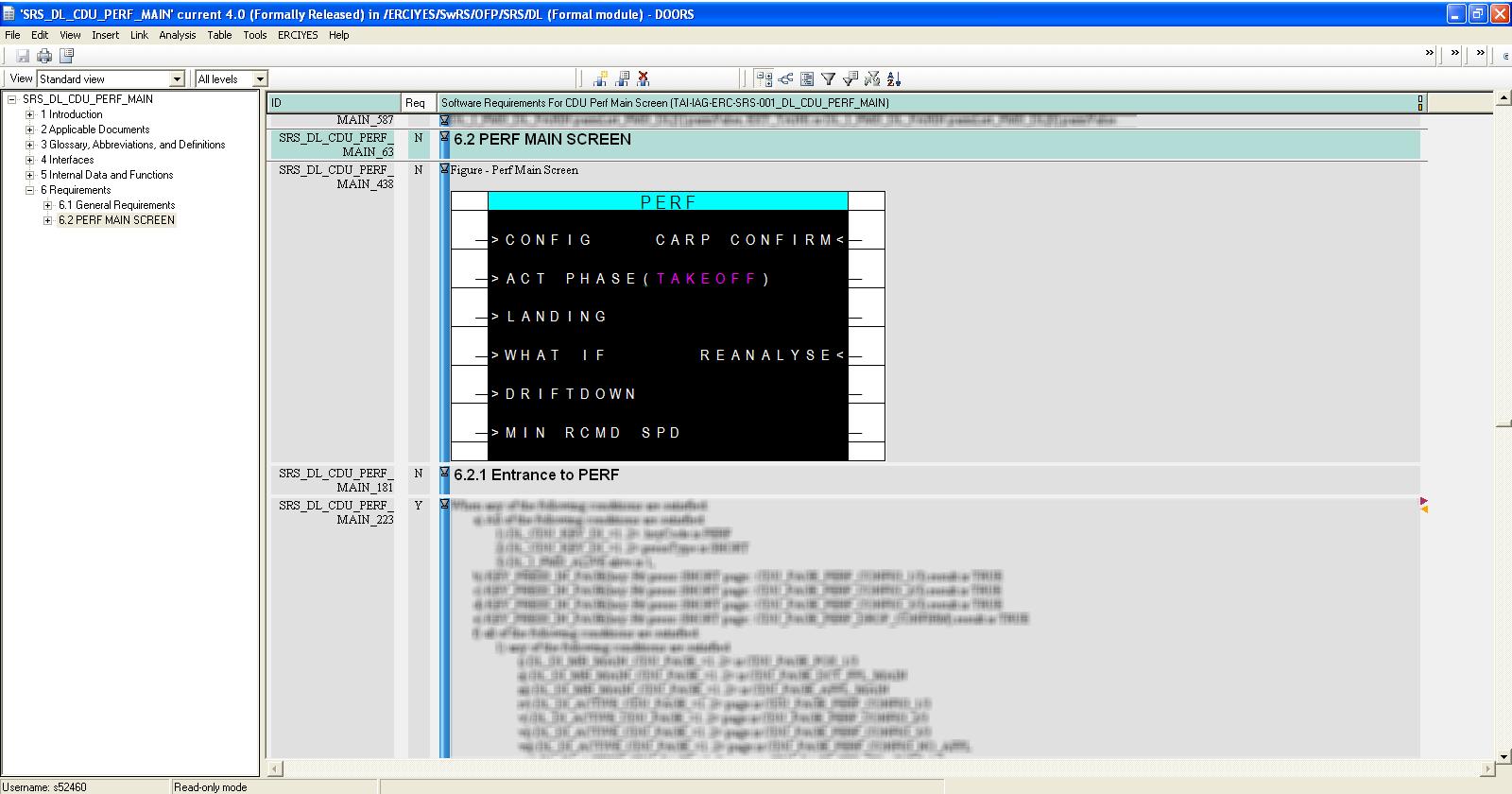
***Figure - 5: CDU***

Test codes are written according to requirements; therefore, to write a test code, the first thing to be done is to open the SRS of the package that is tested through Telelogic DOORS Module.



***Figure - 6: Telelogic DOORS Module***

In the SRS, there are 6 main sections. In “Introduction” section, there is a brief explanation of that package. In “Applicable Documents” section, connection between this package and other packages are given. In “Glossary” section, terms are explained. In “Interfaces” section, there is information about the input and output ports. In “Internal Data and Functions” section, internal data and functions that are used are explained. And finally, in “Requirements” section, requirements of the package are stated.

****

***Figure - 7: SRS***

After opening the SRS, test code is written according to requirements in this SRS. In Erciyes project, software is split into partitions and each partition is tested separately. Therefore, when testing a requirement, the wanted conditions that are stated in the requirement is set manually and then the result is checked.

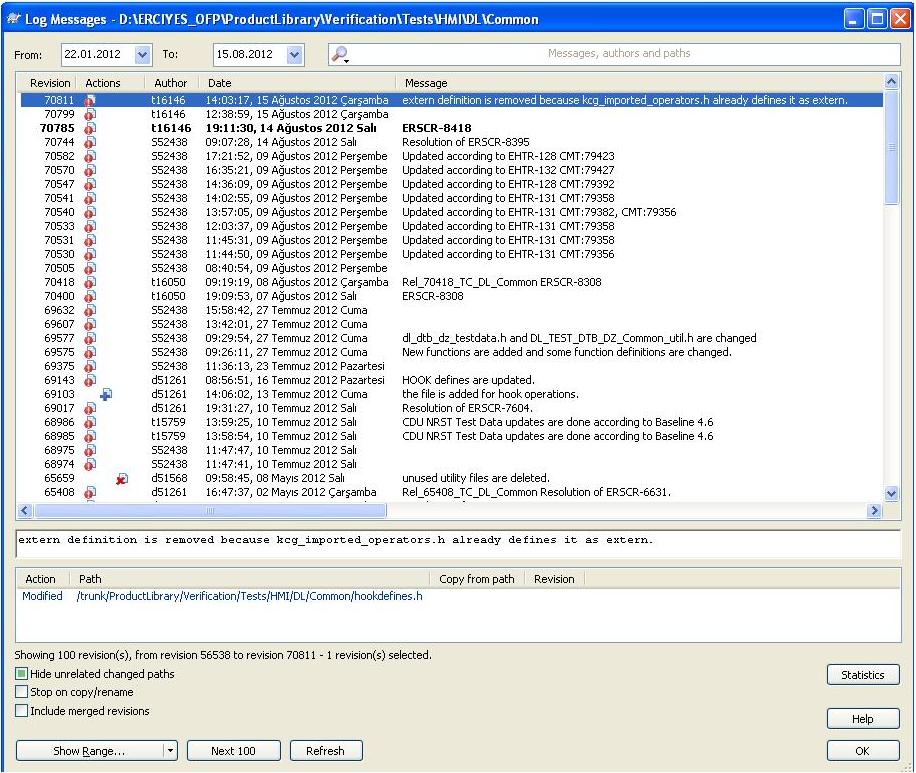
When a requirement is tested, it is traced in the test code, and then when review is being done to that code, reviewer checks these traces whether the requirement is tested completely and correctly.

When writing the test code finishes, the test code is run on the computer and a result file is created. By using this result file, pass and fail steps can be seen and the faults can be found easily.



***Figure - 8: Test Result***

After finishing to writing the test code, the code is committed through Tortoise SVN; i.e. it is sent to the SVN repository and saved there. And then, test code is sent to review in order to be released.

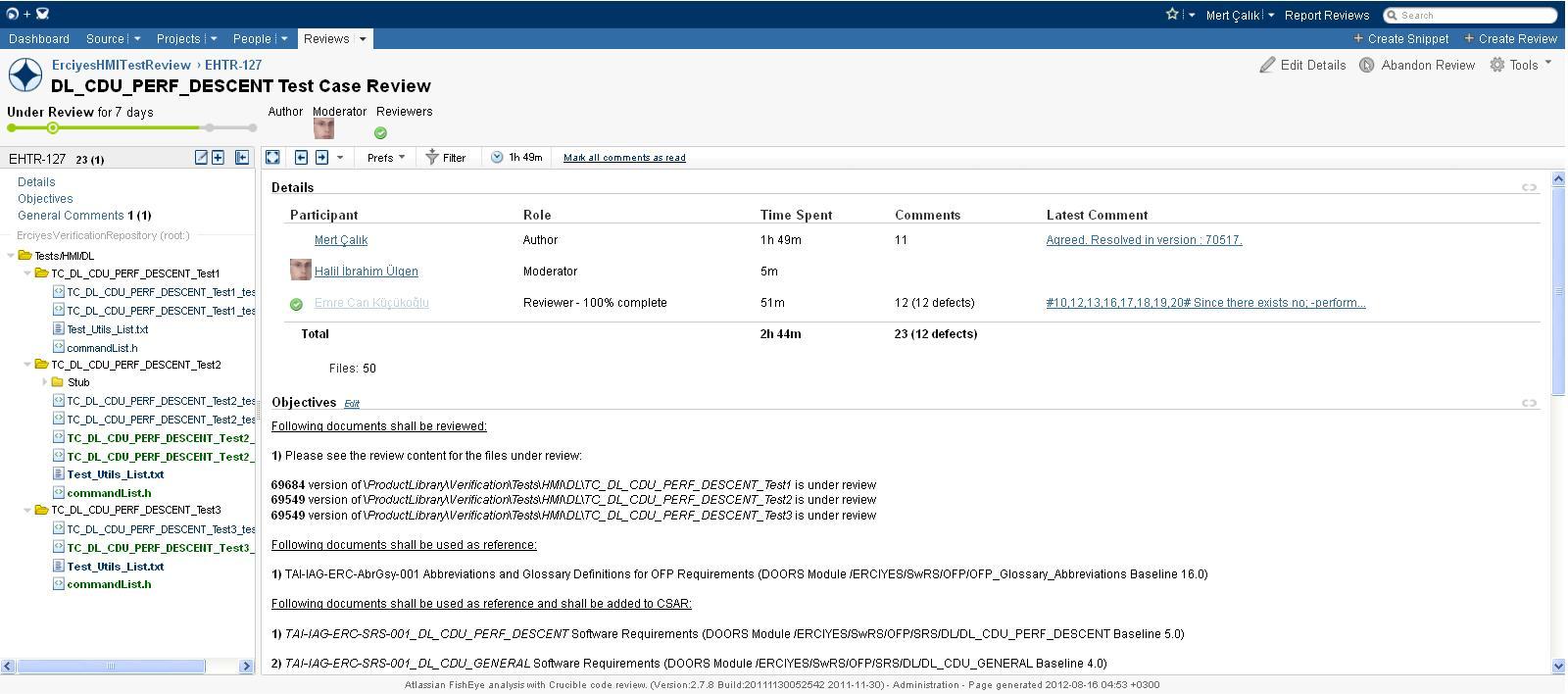
****

***Figure - 9: SVN Log Messages***

* 1. **Reviews**

The purpose of the reviews is checking the correctness of the test codes; in other words, review is testing the test code. While writing the test code, some conditions and situations can be missed, or some conditions that are not needed can be written. By reviews, these mistakes are taken to minimum. To get more effective result from the project, writing source code, writing test code and reviewing jobs are given to different people.

Firstly, the writer of the test code selects the files that s/he wants them to be reviewed, states which baselines of the SRSs s/he used to write that code, selects moderator and reviewers and starts the review through Crucible Dashboard.

****

***Figure - 10: Crucible Dashboard***

After that, the reviewer starts to check the test code. If s/he finds an error or unnecessary codes, s/he enters a comment according to *Reviewer Comment Table*.

|  |  |  |
| --- | --- | --- |
|  | **Description** | **Possible values** |
| **Defect** | If there should be a resolution as a result of this comment defect checkbox should be checked.  If this is selected the following attributes will be enabled. | Check or uncheck |
| **Type** | Type of the comment | Recommendation, Defect, ChecklistNA |
| **Checklist item** | Checklist item no | 1,2,3,4,5,6,7,8,9,… |
| **Status** | Status of the defect.  Open: Initial status value if there is no resolution or verification made on the defect  Verified: The resolution is verified by the reviewer  Rejected: The defect is rejected and no resolution is needed.  N/A: If this field is N/A it means that the checklist item entered is N/A and the comment entered is the explanation of this N/A checklist item. | Open, Rejected, Verified, N/A |

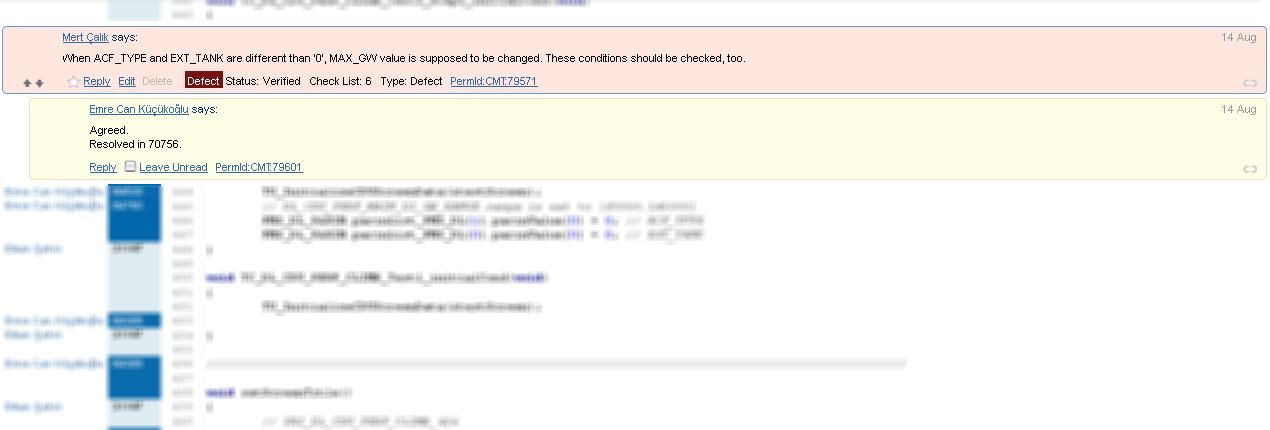
***Table - 2: Reviewer Comment***

If the reviewer finds unnecessary code or wants a piece of code to be changed, s/he enters the comment as *Recommendation*. If s/he finds an error or finds out that a requirement is not fully tested but traced in the code, s/he enters comment as *Defect*, and when entering a defect comment, selects checklist items according to *Checklist for Test Case/Procedure Review Table*.

### DELETED

***Table - 3: Checklist for Test Case/Procedure Review***

The author of the test code can accept or reject the comments of the reviewer. If s/he rejects the comment, s/he states the reason of the rejection. If s/he accepts the comment, s/he should correct the error. After correcting the error, s/he commits the new test code and while doing this, enters the review number and the number of the comment for which s/he changed the test code to SVN as a comment. And then enters a new comment in Crucible below the review’s comment and says in which revision of the code s/he corrected the error.



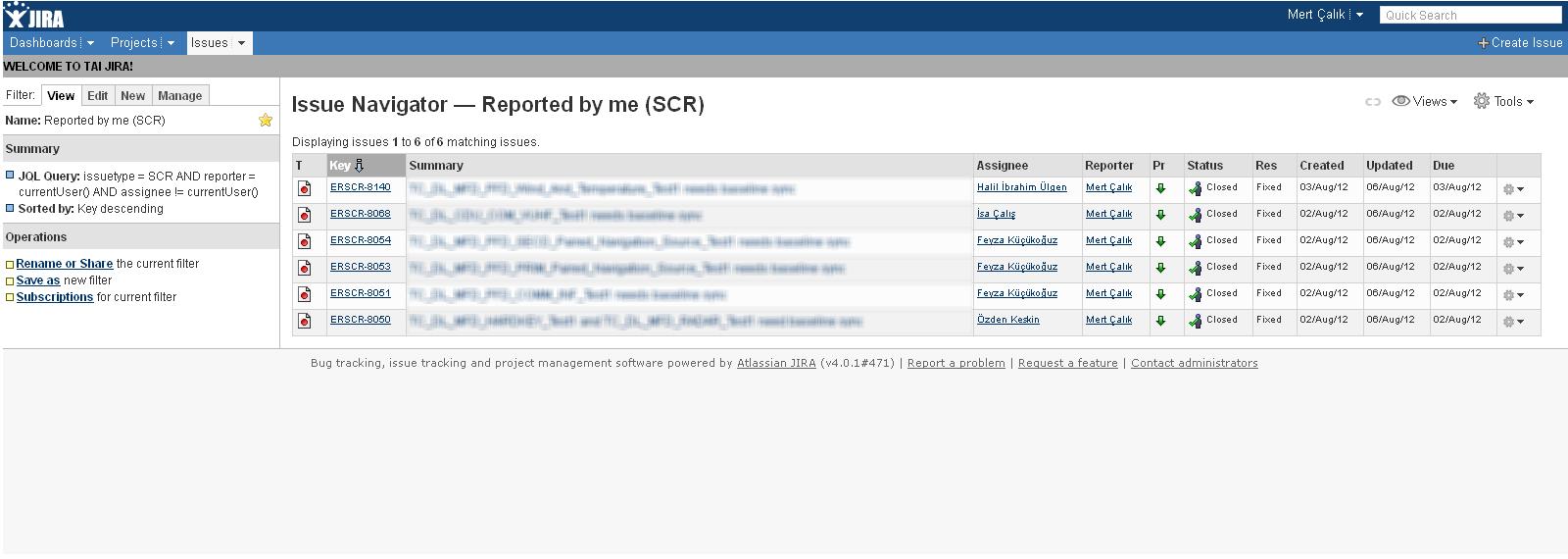
***Figure - 11: Review Comment***

After the author corrected the failure and entered the comment to Crucible, the reviewer changes his/her comment’s status to *Verified*.

After finishing the review, checking all documents, and correcting all errors, the reviewer enters a general comment about the checklist items that are not applicable. And after that s/he completes the review and when all the reviews finish, the code is released.

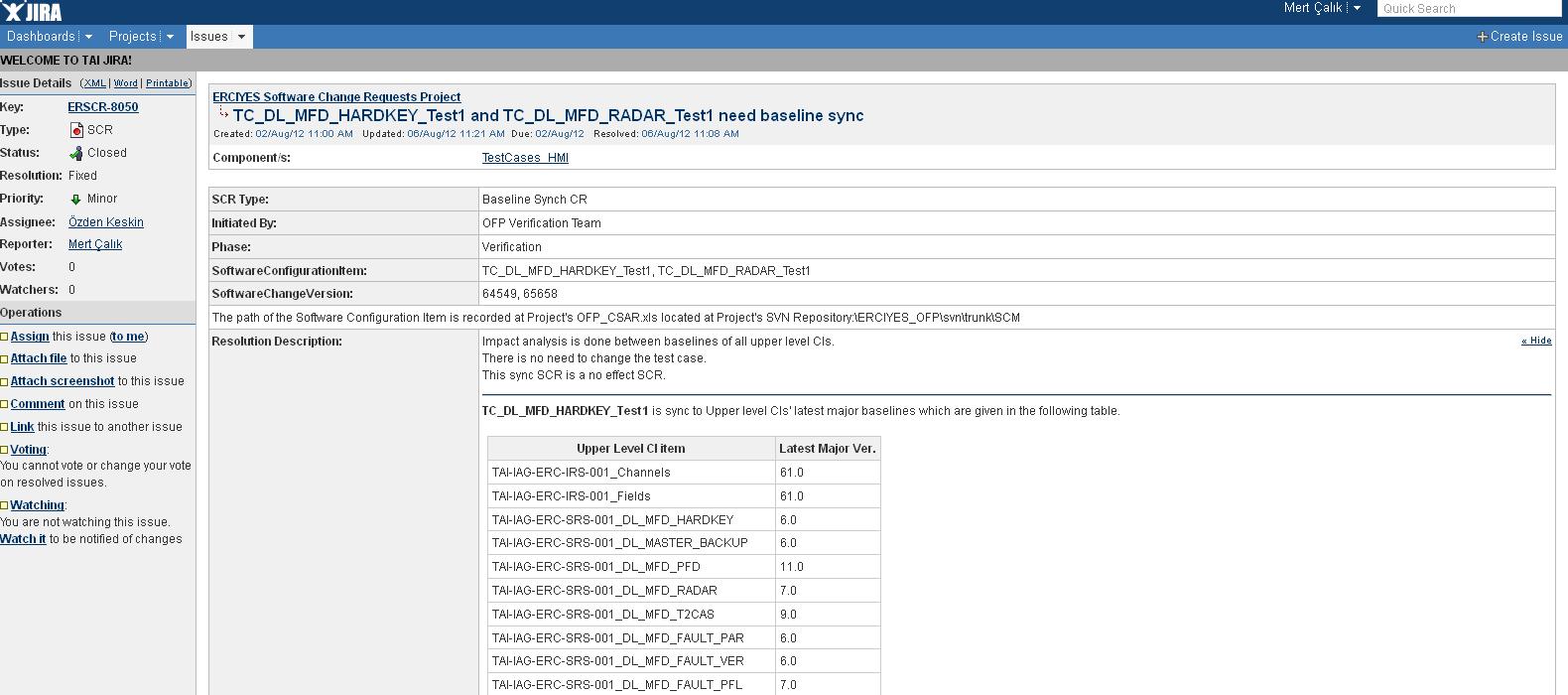
* 1. **SCR Verification**

Test codes are written according to major baselines of SRSs. But when an SRS is changed and released a new major baseline, the test code must be changed, too. However, released codes cannot be directly changed. An SCR must be opened to do this.



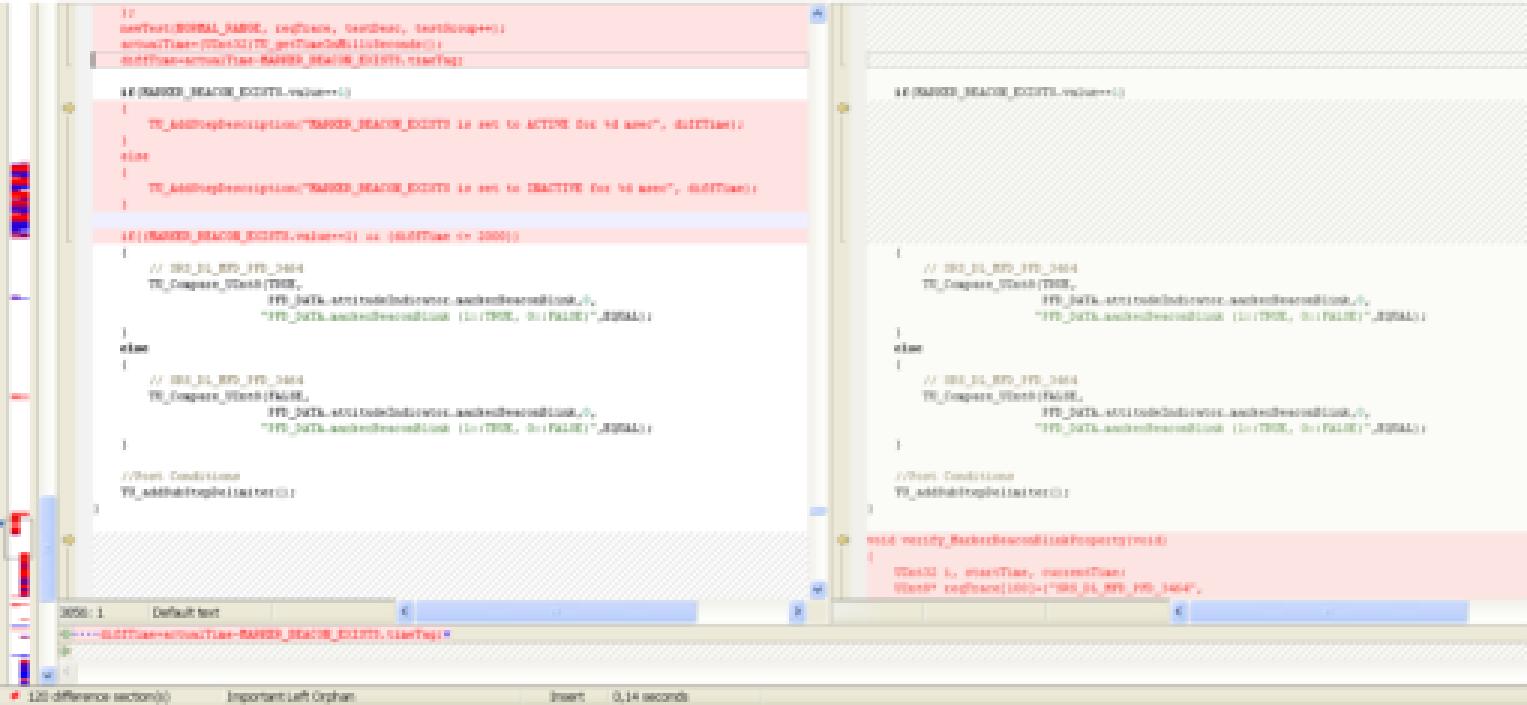
***Figure - 12: SCRs in JIRA***

When a new major baseline of an SRS is released, the differences between the two baselines are checked. According to the differences, the test code is updated. And then, an SCR is opened in JIRA and the code is committed with a comment that includes the number of the SCR. The new baselines that the test code is sync to and the new SVN revision of the test code is written in the SCR. Then, that SCR must be verified. To do this, the author assigns that SCR to someone.



***Figure - 13: Sync SCR***

In order to verify an SCR, the first thing to be done is looking the differences between the two baselines of the SRS. In order to do this, DOORS and Beyond Compare are used. And then the differences between the two code revisions are looked. In order to do this, Tortoise SVN and Beyond Compare are used.



***Figure - 14: Beyond Compare***

After these, whether the changes between the SRSs have been applied to the test code is controlled. If there is a mistake, the SCR is reopened and sent back to the author. If there is no mistake, SCR is verified.