İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

EMNİYET KRİTİK YAZILIM TEST EDİLEBİLİRLİĞİNİN İYİLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Onur ÖZÇELİK

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

HAZİRAN 2015

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

EMNİYET KRİTİK YAZILIM TEST EDİLEBİLİRLİĞİNİN İYİLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Onur ÖZÇELİK
504101512

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Doç. Dr. D. Turgay ALTILAR

HAZİRAN 2015

İTÜ, Fen Bilimleri Enstitüsü'nün 504101512 numaralı Yüksek Lisans Öğrencisi **Onur ÖZÇELİK**, ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"EMNİYET KRİTİK YAZILIM TEST EDİLEBİLİRLİĞİNİN İYİLEŞTİRİLMESİ"** başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı :	**Doç. Dr. D. Turgay ALTILAR** İstanbul Teknik Üniversitesi	.....
Jüri Üyeleri :	**Yrd. Doç. Ayşe Tosun MISIRLI** İstanbul Teknik Üniversitesi	.....
	Doç. Dr. Güray YILMAZ Hava Harp Okulu	.....

Teslim Tarihi : **30 Nisan 2015**
Savunma Tarihi : **09 Haziran 2015**

Aileme,

ÖNSÖZ

Tez çalışmalarım süresince yardımları ve katkıları ile her zaman yol gösteren tez danışmanım Doç. Dr. D. Turgay Altılar başta olmak üzere, fikirlerinden her zaman yararlandığım ve desteğini hiç esirgemeyen çalışma arkadaşım Özdemir Kavak'a tez ve bildiri yazım sürecinde düzeltmeleriyle ve desteğiyle büyük emeği geçen İlksen Özcan'a, TÜBİTAK BİLGEM'deki yöneticilerime ve hayatımın her döneminde olduğu gibi yüksek lisans eğitimim boyunca destek olan aileme sonsuz teşekkür eder, saygılarımı sunarım.

Nisan 2015

Onur Özçelik
Bilgisayar Mühendisi

İÇİNDEKİLER

Sayfa

KISALTMALAR

UDSP : Ulusal Demiryolu Sinyalizasyon Projesi
OOP : Object Oriented Programming
LCOM : Lack of Cohesion of Methods
RFC : Response for a Class
WMC : Weighted Methods Per Class
NOC : Number of Childs
CBO : Coupling Between Objects
IEC : International Electrotechnical Commission
ISO : International Organization for Standardization
LOC : Line of Code
DIT : Depth of Inheritance Tree
EN : European Norm
UML : Unified Modelling Language
İTÜ : İstanbul Teknik Üniversitesi
KM : Kontrol Merkezi
TCDD : Türkiye Cumhuriyeti Devlet Demiryolları
XML : Extensible Markup Language
MOOD : Metrics for Object Oriented Design
QMOOD : Quality Model for Object Oriented Design

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

EMNİYET KRİTİK YAZILIM TEST EDİLEBİLİRLİĞİNİN İYİLEŞTİRİLMESİ

ÖZET

Yazılım, günden güne hayatımıza daha çok girmektedir. Sıklıkla kullanılan bazı sistemlerde yazılım donanıma göre daha çok tercih edilir hale gelmiştir. Yazılımın yoğun olarak kullanıldığı askeri sistemler, medikal sistemler, havacılık ve ulaşım sistemleri çalışırken emniyetli olmak zorundadırlar. Bu tip emniyet kritik sistemlerde yazılımın çalışması esnasında oluşabilecek olan hatalar felaketle (ölüme sebebiyet verme, sakat bırakma, çevreye veya kullanılan donanıma yıkıcı zararlar verme) sonuçlanabilir. Bu tip sistemlerde yazılımından veya yazılımın kontrol ettiği donanımdan kaynaklanabilecek hataların engellenmesi veya en azından sonuçlarının kontrol edilebilmesi için bu sistemler için geliştirilen yazılımın, sistem devreye alınmadan önce detaylı test edilmesi gereklidir. Emniyet kritik sistem yazılımları genellikle biribirini takip eden evrelerle geliştirilirler ve testleri gerçekleme (kodlama) evreleri tamamlandıktan sonra yapılır. Gerçekleme evresi tamamlandıktan sonra yapılan testler müşteri isterlerinin sıklıkla değiştiği, tam anlaşılmadığı ve yeni isterlerin ortaya çıktığı durumlarda yetersiz kalabilirler. Bu durumun önüne geçebilmek için emniyet kritik yazılım sistemleri planlama evresinden itibaren test edilebilir olarak geliştirilmelidir. Bir yazılım sisteminin test edilebilirliğini yazılım kalitesi ile ilişkilidir. Bir yazılım sistemin kalitesinin doğru olarak ölçülmesi yazılımda test edilebilirlik açısından iyi veya kötü kısımların ortaya çıkmasını sağlayabilir.

Ölçme, varlıkların niteliklerinin sayılaştırılması olarak tanımlanır. Bu sayılaştırma işlemi yapılmadan önce varlığa uygun metriklerin tanımlanması gereklidir. Yazılım sistemlerinin kalitesinin ölçülebilmesi için yazılım tasarımına ve gerçekleme yöntemine uygun çeşitli metrikler bulunmaktadır. Yazılım metrikleri kullanılarak yazılım test edilebilirliğini analiz etmek için sayısal değerler elde etmek mümkündür.

Bu çalışma kapsamında emniyet kritik yazılım sistemlerinin test edilebilirliği ile nesneye yönelik yazılım tasarım ve geliştirme yöntem ve ilkeleri arasındaki ilişki araştırılmıştır. Sonuçların çıkarılması için Tübitak ve İstanbul Teknik Üniversitesi işbirliği ile geliştirilen Ulusal Demiryolu Sinyalizasyonu Projesi (UDSP)'nin emniyet kritik yazılım bileşenlerinden demiryolu anklaşman yönetim yazılım bileşeni incelenmiştir. Mevcut yazılımın kalitesi, nesneye yönelik ve nesneye yönelik olmayan çeşitli yazılım metriklerine göre ölçülmüş ve yazılım test edilebilirliği değerlendirilmiştir. Aynı yazılımın ikinci bir sürümü test yönelimli yazılım tasarım ve geliştirme yöntem ve ilkeleri dikkate alınarak gerçeklenmiştir. İkinci sürüme ait yazılım kaliteside benzer şekilde ölçülmüş ve yazılım test edilebilirliği değerlendirilmiştir. Test yönelimli yazılım tasarımı ve geliştirme yöntem ve ilkeleri kullanılarak gerçeklenen sürümün yazılım test edilebilirliğinin ilk sürüme göre iyileştirildiği gösterilmiştir.

IMPROVING TESTABILITY OF SAFETY CRITICAL SOFTWARE

SUMMARY

Nowadays software controls large majority of the systems that humankind use. Systems that software is used widely, such as transportation, military, medicine and avionics must be safe during the operation. Failure 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 software or at least mitigate the risks, elaborated testing is required. Safety critical software systems present more testing challenges as compared to general-purpose systems. Safety critical software's dependency on hardware, the limited user interface and lack of tools makes the testability of this software more difficult. Quality in safety critical software is generally tied to platform-specific testing tools geared towards debugging.

Engineers usually develop safety critical software systems with sequential phases and test with test at the end approach for certification purposes. Although effective testing helps to correct the functioning of software systems, the goal of testing is not proving 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 domain. However, designing for testability through hardware abstraction in conjunction with low complexity, few lines of code and simplicity of the code help improve testability.

Test at the end approach is not sufficient for any software system when requirements are unclear or change frequently. To overcome weaknesses of test at the end, software systems can be tested starting from the early stages. To test in the early stages**,** software must be testable. Testability of software is defined as whether or not software supports testing activities. Software testability is considered under software quality. Software quality is defined as degree to which the software product satisfies stated and implied needs when used under specified condition. To evaluate software quality first we need a way to measure it. Correct measurement of software quality can help project team to understand which parts of software are testable and which are not.

Measurement is the assignment of numbers to object or event properties. Suitable metrics should be defined for object or event before the measurement. Measurement in software development plays an important role in every phase. For measuring software quality, suitable software metrics should be chosen. With the help of software metrics it is possible to obtain some clues about software quality. The important point here is that software metrics should be analyzed together. If metrics are analyzed on their own, they will not tell too much.

This thesis investigates testability from the perspective of metrics used in an object-oriented system. The main idea is giving an overview of software metrics with the prioritization of testability as the overall goal. Several metrics have been proposed to identify testability weaknesses. However, it is sometimes difficult to be convinced that those metrics are really related to testability.

For the purpose of this study, testability of safety critical software and its relation to object oriented patterns, principles and practices were investigated. In this work testability is understood as unit level testability. A case study was conducted for railway interlocking management software component. Studied software component was first developed under National Railway Signalization Project by the collaboration of The Scientific and Technological Research Council of Turkey and Istanbul Technical University. The original railway interlocking management software was developed using object oriented design and programming. The component was analyzed to measure its existing testability. Testability of software component was determined by using object oriented and non-object oriented software metrics. Well known Chidamber and Kemerer (CK) metrics suite was chosen for object oriented metrics. With the widespread use of object-oriented technologies, CK metrics have proved to be very useful. Complexity and lines of code (LOC) metrics were chosen for non-object oriented metrics.

The measurement results showed that most of the classes inside railway interlocking management software are very complex according to cyclomatic complexity, weighted methods per class and response for a class metrics. Their cohesion is low according to lack of cohesion of methods metric and couplings between them are high according to coupling between object classes metric. Complexity, cohesion and coupling are three important factors that affect testability of software. Testing software becomes a tedious task as the size and complexity of software increases. In addition, lack of cohesion inside software system is directly related with complexity. Likewise software system that has huge numbers of couplings is hard to test. Because these couplings must be supplied by, providing real or fake objects before, testing begins. In terms of complexity, cohesion and couplings existing testability of railway interlocking management software is low.

To prove the correlation between good design and software testability, second version of the interlocking management software was developed. For second version, test driven software design and development approach was used. The primary benefit of using test driven development is that it improves the design of the code. In addition, some well-known object oriented principles and patterns were applied to increase the software testability. After completion of development, the second version is analyzed using same metrics and results were compared with previous results. In addition, some code coverage metrics like statement, function and branch coverage were measured. The measurement results showed that testability of second version is considerably improved in terms of complexity, cohesion and coupling. This argument is supported by code coverage metrics results. However, several threats to validity of study were also identified.

First, this study's sample size was quite small. This was due primarily safety critical software systems are usually developed by using closed source software technologies. Therefore, the study sample is chosen from a closed sourced safety critical software project that author previously worked. A small size of study makes

it easy to argue that the numbers in this study may not reach a suitable level of statistical significance.

In addition, it is easy to argue that metric analysis is insufficient for software testability improvement. Various researchers took different approaches to the software testability measurement in their studies. Generally, software testability is not measured by using software metrics in those studies. However, in this study the chosen metrics are used for measuring complexity, cohesion and coupling in software systems. Therefore, improvement in that trio should affect software testability in positive way.

In addition, one can argue that validation of test-driven development for improving software testability is insufficient. However, test-driven development effort in this study is supported by code coverage metrics like statement, function and branch coverage.

1. GİRİŞ

Yazılım her geçen gün daha çok sistemde kullanılmaktadır. Yazılımın bu kadar çok yaygın hale gelemesi yazılım karmaşıklığını artırmaktadır. Karmaşık yazılımlarda hataya daha açık yazılımlardır ve test edilmeleri daha zordur. Doğru yaklaşım ve yöntemle yapılan yazılım testleri her ne kadar yazılımdaki hataların tespit edilmesine yardımcı olsa da yazılım testlerin amacı yazılımda hata bulunmadığını kanıtlamak değildir. Yazılım testlerinin amacı yazılımda hatalar bulunabileceğini göstermektir. Yazılım testleri için ne kadar zaman ve kaynak ayrılırsa ayrılsın yazılımı tüm giriş ve buna karşılık gelen tüm çıkışları ile test etmek projedeki zaman ve bütçe kısıtları nedeniyle olası değildir [16].

Yazılım, esnekliği sebebiyle emniyet kritik sistemlerde her geçen gün donanıma göre daha çok tercih edilmektedir. Emniyet kritik sistem yapmakla sorumlu olduğu işi yaparken hata yapması durumunda çeşitli felaket senaryolarının (ölüm, kalıcı sakatlık, ciddi çevre veya donanım tahribatı) gerçekleşmesine neden olabilecek sistem olarak tanımlanır. Günümüzde yoğun olarak kullanılan ulaşım sistemleri, askeri sistemler ve medikal sistemler emniyet kritik olarak tanımlanmaktadır. Emniyet kritik bir sistem içinde çalışan yazılım, sistemin üstlendiği sorumluluk sebebiyle diğer tür sistem içinde çalışan yazılıma göre çok daha iyi test edilmesi gerekir [16].

Yazılım içeren sistemler tümleştikten sonra yapılan kabul testleri tüm olası durumları test etmede yetersiz kalabilmektedirler. Bu sebeple yazılım sistemlerini oluşturan yazılım bileşenleri kendi içlerinde çok iyi test edilmelidirler. Bir yazılım bileşeninin çok iyi test edilebilmesi için bileşenin test edilebilirliğinin yüksek olması gereklidir. Yazılım test edilebilirliği basitçe yazılımın test faaliyetlerini ne kadar desteklediğidir. Yazılım test edilebilirliği doğrudan ölçülebilen bir yazılım niteliği değildir. Test edilebilirlik, yazılım kalitesi altında değerlendirilir. Kaliteli yazılımda, test edilebilirlik yüksek olacağı için hata çıkma olasılığı düşüktür ve yazılım isterlerin değişmesi veya yeni isterlerin eklenmesi gibi durumlar test edilebilirliği çok etkilemez. Bunun aksine kalite nitelikleri düşük olan yazılım hatalara açıktır, test

edilebilirliği düşüktür ve gelen değişiklik talepleri yazılımı sürdürülebilir olmaktan çıkarabilir. Yazılım kalitesinin ölçülmesi amacıyla yazılım metrikleri kullanılır. Metrikler basitçe bir varlığın niteliklerinin sayısal olarak ifade edilmesidir. Yazılım metrikleri kullanılarak yapılan ölçümlerde dikkat edilmesi gereken en önemli nokta tek bir metriğin değerinin dikkate alınmasının hatalı olacağıdır. Bunun yerine metriklerin değerleri birlikte değerlendirilmelidir. Çünkü metrikler tek başlarına yeterince anlam ifade etmezler.

Bu çalışma kapsamında emniyet kritik yazılım sistemlerinin test edilebilirliği ile nesneye yönelik yazılım tasarım ve geliştirme yöntem ve ilkeleri arasındaki ilişki araştırılmıştır. Sonuçların çıkarılması için emniyet kritik yazılım kategorisinde değerlendirilen demiryolu anklaşman yönetim yazılımı incelenmiştir. Mevcut yazılımın kalitesi, nesneye yönelik ve nesneye yönelik olmayan çeşitli yazılım metriklerine göre ölçülmüş ve yazılım test edilebilirliği değerlendirilmiştir. Aynı yazılımın ikinci bir sürümü test yönelimli yazılım tasarım ve geliştirme yöntem ve ilkeleri dikkate alınarak gerçeklenmiştir. Gerçeklenen ikinci sürümün kalitesi aynı metrikler kullanılarak ölçülmüş ve yazılım test edilebilirliği değerlendirilmiştir. Sonuç olarak ikinci sürümün test edilebilirliğinin ilk sürüme göre oldukça iyileştirildiği gösterilmiştir.

1.1 Tezin Amacı

Bu çalışmanın amacı nesneye yönelik programlama (OOP) kullanılarak gerçeklenen emniyet kritik yazılım test edilebilirliği ile nesneye yönelik yazılım tasarım ve geliştirme yöntem ilkeleri arasındaki ilişkinin incelenmesidir. Tez kapsamında nesneye yönelik tasarım ve programlama ilkeleri kullanılarak gerçeklenen emniyet kritik bir yazılım bileşeninin birim test edilebilirliği incelenecektir.

1.2 Tezin Organizasyonu

Bölüm 2'de literatür araştırmasına yer verilmiştir. Bölüm 3'te emniyet kritik yazılım sistemleri detaylandırılmıştır. Bölüm 4'te yazılım kalitesi ve yazılım test edilebilirliği anlatılmıştır. Bölüm 5'te demiryolu anklaşman yönetim sistemi yazılımın test edilebilirliği üzerine bir vaka çalışması yapılmıştır. Bölüm 6'da elde edilen sonuçlar sunulmuş ve ileriki çalışmalar hakkında bilgi verilmiştir.

2. LİTERATÜR ARAŞTIRMASI

Emniyet kritik yazılım sistemlerinin test edilebilirliğinin iyileştirilmesi alanında yapılan literatür araştırmasında bir çok araştırmanın yazılım test edilebilirliği, test yönelimli geliştirme ve emniyet kritik yazılım sistemi geliştirme konularıyla ayrı ayrı ilgilendiğini göstermiştir. Bazı araştırmalar ise emniyet kritik yazılım sistemlerinin çevik ilkelerle geliştirmesi konu başlığında test yönelimli geliştirme ile dolaylı olarak ilgilenmiştir.

Jonhson ve diğerleri Avrupa raylı ulaşım sistem yazılımı geliştirme standardı olan EN 50128 belgesindeki yönergelere uyması gereken emniyet kritik raylı ulaşım yazılım sistemlerinin çevik ilkelerle geliştirilmesinin EN 50128 belgesinde tavsiye edilen yazılım isterlerini destekleyip/desteklemediğini analiz etmişlerdir [1]. Araştırmacılar bazı yazılım isterlerinin çevik ilkelerle desteklendiğini bazılarının ise çevik ilkelerle desteklenebilmesi için çevik ilkenin yönergelere uydurulmasının gerektiği sonucuna varmışlardır [2].

Bir başka araştırmada Ge ve diğerleri emniyet kritik yazılım sistemi geliştirmek için artımlı bir yaklaşım sunmuşlardır. Çalışmalarında ilk olarak emniyet kritik yazılım sistemi geliştirmek için yukarıdan aşağıya yazılım tasarım yöntemini incelemişler ve bu yönetimin yazılım emniyet hedeflerine erişmek için sağlaması gereken en karakteristik özelliklerini çıkarmışlardır. Ardıdan hem yazılım sisteminin hemde emniyet hedeflerinin geliştirilmesi için artımlı bir yöntem sunmuşlardır [3].

Fitzgerald ve diğerleri tarafından yapılan bir diğer çalışmada yönergelere uyulmuş bir ortamda çevik yazılım geliştirme yaklaşımın başarı ile uygulandığı bir örnek vaka incelenmiştir. Araşırmacılar yönergelere uyulan bir ortamda sundukları çevik yazılım geliştirme yaklaşımının oldukça iyi sonuçlar verebileceği sonucuna varmışlardır [4].

Wolff çalışmasında formal yöntemlerin çevik yazılım geliştirme yaklaşımlarından Scrum yöntemine uygulanabilirliğini araştırmıştır. Araştırmacı formal yöntemlerin Scrum sürecine uyarlanabileceği sonucuna varmıştır [5].

Shristava ve Jain tarafından yapılan bir araştırmada test yönelimli yazılım geliştirme yaklaşımı için kullanılabilecek birim test tasarım metrikleri incelenmiştir. Araştırmacılar emniyet kritik sistemleri dikkate almamış olmasına rağmen önerdikleri metriklerden bazıları emniyet kritik yazılım sistemleri geliştirmek amacıyla kullanılabilir [6].

Bruntink ve Deursen tarafından yapılan bir araştırmada yazılım test edilebilirği ile literatürde tanımlı olan nesneye yönelik yazılım metrikleri arasındaki ilişki incelenmiştir. Araştırmacılar sınıfın tetiklediği metot sayısı, çocuk sayısı ve sınıf kod satır sayısı gibi metriklerin yazılım test edilebilirliği üzerinde etkili olduğu sonucuna varmışlardır [7].

Khanna araştırmasında nesneye yönelik bir sistemde yazılım test edilebilirliğini metrikler açısından incelemiştir. Araştırmacı, Analitik Hiyerarşi Süreci (Analytic Hierarchy Process) yöntemini kullandığı araştırmasının sonucunda daha az bağımlılık ve iyi bir nesneye yönelik tasarıma sahip olan sınıfların test edilebilirliğinin arttığı sonucuna varmıştır [8].

Alwardt ve diğerleri tarafından yapılan nesneye yönelik sistemler üzerinde yapılan bir araştırmada test edilebilirlik için yazılım tasarımı, otomatik regresyon testi yaklaşımı ve bu yaklaşımın Uç Programlama (Extreme Programming) ile ilişkisi ve Lean 123 yaklaşımının yararları ve zararları incelenmiştir. Araştırmacılar araştırmanın test edilebilirlik için yazılım tasarımı bölümünde sınıf içi uyumun yüksek olmasının ve sınıf bağımlılıklarının az olmasının test edilebilirliği artıran etkenler olduğunu belirtmişlerdir. Ayrıca aynı araştırmanın birim test geliştirmek için en iyi yaklaşımlar bölümünde test edilebilirliği yüksek olmayan sınıflar için geliştirilen birim testlerin etkinliğinin az olacağını belirtmişlerdir [9].

Sahay araştırmasında gömülü yazılım projelerinde test edilebilirlik için yazılım tasarımını irdelemiştir. Araştırmacı, araştırmasının sonucunda gömülü yazılım projeleri için yazılım test edilebilirliğini artıran iki önemli etkenin kontrol edilebilirlik ve gözlenebilirlik olduğu çıkarımında bulunmuştur [10].

Joshi ve Sardana tarafından yapılan bir araştırmada nesneye yönelik yazılım sistemlerin tasarım ve kodlama zamanı test edilebilirlik analizi yapılmıştır. Araştırmacılar, yazılımda eksik isterler bulunması, tasarım kalıbı kullanmama, çözülemeyen nesne ağına sahip olma gibi özelliklerin tasarım zamanı test

edilebilirliğini etkilediğini ifade etmişlerdir. Aynı çalışmada istisna yönetme mekanizması olmama, tekrarlamalı (recursive) kodlamaya sahip olma, yapısal olmayan ve erişilemez kodlama içerme gibi özelliklerin kodlama zamanı test edilebilirliği ilişkili olduğu belirtilmiştir. Dairesel bağımlılıklar, uyumsuz arayüzlere sahip olma, takip edilebilir olmama, kontrol edilebilir olmama ve izlenebilir olmama gibi özelliklerin ise hem tasarım hemde kodlama zamanı test edilebilirliği ile ilişkili olduğu belirtilmiştir [11].

Tahir ve diğerleri tarafından yapılan bir araştırmada nesneye yönelik yazılım sistemlerindeki sınıfların test edilebilirliğinin, yazılım çalışma zamanı nitelikleri ile ilişkisi araştırılmıştır. Araştırmacılar, statik kod analizi yerine dinamik yazılım çalışma zamanı analizi yaptıkları çalışmalarında sınıfların çalışma zamanı bağımlılıklarının ve çalışma zamanı çağrılma frekanslarının sınıf test edilebilirliğini etkileyen iki önemli etken olduğu sonucuna varmıştır [12].

Kanstren çalışmasında bileşen temelli gömülü yazılım sistemlerinde test edilebilirlik için yazılım tasarımını incelemiştir. Araştırmacı, yazılım için test edilebilirliğin mümkün olan en erken safhada ele alınması gerektiği, donanımda koşan işletim sistemi ile donanımın benzetiminin yapıldığı işletim sisteminin aynı olmasının ve sistemde test işlevselliğini sağlayacak düzenlemelerin yapılmasının yararlı olduğu sonuçlarına varmıştır. Bunun yanında test durumlarının gerçekleştirmeden soyutlanmasının gerekli olduğunu ve sistemin test edilebilirliğini iyileştirecek çalışmaların yapılabilmesi için proje yönetiminin desteğinin oldukça önemli olduğunu belirtmiştir [13].

Alanen ve Ungar çalışmalarında test edilebilirlik için yazılım tasarımını, test edilebilirlik için donanım tasarımı ve donanınım sistemlerinde bulunan gömülü öz testlerle karşılaştırmışlardır. Araştırmacılar test edilebilirlik için yazılım ve donanım tasarımının oldukça benzer yönleri olduğunu ve donanım sistemlerinde kullanılan gömülü öz testlerin ve test edilebilirlik için donanım tasarım yöntemlerinin yazılım sistemlerine uyarlanması için gerekli çabanın olumlu sonuçlar vereceğini belirtmişlerdir [14].

Mulo çalışmasında yazılım sistemlerinde test edilebilirlik için yazılım tasarımını incelemiştir. Çalışmada test edilebilirlik iki ayrı bakış açısıyla incelenmiştir. Bu bakış açılarından ilkinde test edilebilirlik incelenen sisteme ait içsel bir nitelik olarak

kabul edilmektedir. Diğer bakış açısında ise yazılım geliştirme faaliyetleri sırasında önem verilmesi gereken ve yazılım testlerinin yapılmasını kolaylaştıran faaliyetler olarak kabul edilmektedir. İçsel nitelik olarak test edilebilirlik yazılımın kontrol edilebilirlik ve gözlenebilirlik niteliklerine sahip olması olarak tanımlanmaktadır. Yazılım geliştirme faaliyetleri sırasında önem verilmesi gerekli faaliyetler açısından test edilebilirlik yazılımın bazı yapısal ve davranışsal ve veri akış metriklerine sahip olup/olmamasına göre değerlendirilmiştir [15].

3. EMNİYET KRİTİK YAZILIM SİSTEMLERİ

3.1 Yazılım Emniyeti

Yazılım tek başına emniyetli ya da emniyetsiz olarak tanımlanamaz. Yazılım emniyetinden bahsederken aslında yazılımın bir parçası olduğu sistemin emniyetinden bahsediyoruz demektir. Sistem sorumlu olduğu işi yaparken hayat kaybına veya sakatlığa, hayati bir ekipmanın kaybına veya ciddi hasarına veya çevreye verilecek ciddi hasara neden olabilecekse sistem emniyeti olmazsa olmazdır. Sistem emniyeti tehlikeler ve bu tehlikeli durumların engellenmesi ile ilgilidir. Yazılım yada donanım tehlikeli bir duruma neden olabilecekse, tehlikeyi kontrol ediyorsa veya tehlikenin oluşması durumunda etkilerinin azaltılmasından sorumlu ise sistem emniyeti içerisinde ele alınmalıdır. Yazılım bir çok sistemde olduğu gibi emniyetin öncelikli olduğu sistemlerde de hayati ve ana parçalardan birisi haline gelmiştir. Bunun nedeni aynı işi yapan bir donanıma göre daha çok hızlı yanıt vermesi ve en olumsuz koşullarda bile güncellenebilmesidir.

3.2 Emniyet Kritik Yazılım

Emniyet kritik yazılım tanımını yapabilmek için öncelikle bu tip yazılımların çalıştığı sistemlerde kullanılan iki adet terimi tanımlamak gerekir. NASA Güvenlik El Kitabına göre [16]:

Kaza (Mishap): Gerçekleşmesi durumunda hayat kaybına, kalıcı sakatlığa veya iş hastalıklarına, hayati bir ekipmanın kaybına veya ciddi hasarına, çevreye verilecek ciddi hasara neden olan planlanmamış olaylar bütünüdür.

Tehlike (Hazard): Kaza ile sonuçlanabilecek veya kazanın gerçekleşmesine katkıda bulunabilecek olası risk durumudur. Her tehlike durumunun bir oluşma nedeni ve oluştuğunda bir veya birden fazla olumsuz etkisi bulunur. (hasar, hastalık vb.)

Yazılım tek başına tehlikeli değildir fakat tek başına var olması da mümkün değildir. Genellikle bir elektronik sistemin (bilgisayar vb.) içinde çalışır ve başka donanımları kontrol eder.

Yazılım eğer tehlike durumuna doğrudan neden olabilecekse veya tehlike durumunu kontrol ediyorsa tehlikeli olarak değerlendirilir.

Emniyet kritik yazılım tehlikeli olarak değerlendirilen yazılımı içeren yazılımdır. Yazılım ayrıca eğer tehlikeli yada emniyet kritik başka bir yazılımı veya donanımı yönetiyorsa ve izlenmesinden sorumlu ise emniyet kritik olarak sınıflandırılır. Emniyet kritik yazılım ile aynı fiziksel ortamda çalışan yazılım emniyet kritik bölümden tamamen izole değilse emniyet kritik olarak değerlendirilir [16].

Özet olarak yazılım aşağıdaki özelliklere sahipse emniyet kritiktir:

- Tehlikeli veya emniyet kritik yazılım veya donanımı kontrol ediyorsa
- Emniyet kritik yazılımın veya donanımın neden olabileceği tehlikelerin izlenmesinden sorumluysa
- Emniyet kritik bir işlemin gerçekleşmesi için veri sağlıyorsa
- Emniyet kritik bir işlemi etkileyen için bir hesaplama yapıyorsa
- Yazılım ya da donanım tehlike kontrollerini doğrulanmasından sorumluysa
- Emniyet kritik yazılımın ya da donanımın doğru çalışmasını engelleyebilecek bir sorumluluğu bulunuyorsa

3.3 Yazılım Emniyet İsterleri

Yazılım emniyeti genel olarak uluslararası standartların, yönetmeliklerin dayattığı veya proje ve çalışma ortamına özel olan isterler çevresinde şekillenir. Yönetmeliklerde veya standartlarda geçen isterler belirlendiğinde projeye özel emniyet isterlerinin belirlenmesi için elde mevcut sisteme özel veri değerlendirilir.

Bu tip değerlendirmeler yapılırken yaygın olarak kullanılan yöntem Ön Tehlike Analizi (Preliminary Hazard Analysis) ‘dir. Ön Tehlike Analiz raporunda tehlikelerin oluşma nedenleri ve sistem emniyet isterlerine uygun tehlike kontrollerine yer verilir. Her bir tehlike kontrolüne karşılık en az bir tane yazılım isteri oluşturulur. Her oluşturulan yazılım isteri Yazılım İsterleri Belirtimi (Software Requirements Specification) belgesine emniyet kritik yazılım isteri olarak eklenir.

Emniyet kritik yazılım sistemlerinde emniyet kritik işlevlerin neden olabilecekleri kazaların sonuçları ölümcül ve çok pahalı olacağından emniyet kritik yazılım istelerinin doğrulanması oldukça önemlidir. En çok karmaşık, tamamlanmamış veya eksik istelerin emniyet zaafiyetlerine neden olduğu unutulmamalıdır.

3.4 Ön Tehlike Analizi (Preliminary Hazard Analysis)

Tehlikeli faaliyetler yürütülmesi amacıyla kullanılan ve yazılım içeren herhangi bir sistemin veya bu sistemin çalıştığı ortamın analiz edilebilmesi için Ön Tehlike Analizi yapılması zorunludur. Başlangıçta yapılan Ön Tehlike Analizinin sonuçları belirlendiğinde emniyet isterleri belirlenebilir ve bu isterler yazılım veya donanım ile ilişkilendirilebilir. Ön Tehlike Analizinin yapılmasındaki en önemli nokta yazılımın, sistemin geri kalanı ile nasıl etkileşim kurduğunu tespit etmektir [16].

3.5 Yazılım Alt Sistem Tehlike Analizi (Software Subsystem Hazard Analysis)

Ön Tehlike Analizi, tehlikeleri sistem seviyesinde değerlendirir. Bu tehlikelerin bazıları yazılım ile ilişkili olabilir. Yazılım Alt Sistem Tehlike Analizi, hangi tehlikelerin doğrudan veya dolaylı olarak yazılımla ilişkili olduğunu belirlemeye yarayan yöntemdir [16].

Yazılım Alt Sistem Tehlike Analizi için gerekli izlek oldukça basittir. Ön Tehlike Analizinde yazılım ile doğrudan veya dolaylı olarak ilişkili olabilecek olan tüm tehlikeler yazılım tehlikeleri listesine eklenir.

Yazılım bir tehlike durumunu çeşitli şekillerde tetikleyebilir. Yazılımın tetikleyebileceği tehlike durumları arasında:

- Yazılımın çökmesi sonucu oluşabilecek tehlike durumları
- Yazılımın tehlike kontrolünün çalışmaması
- Yazılım emniyet geçişi (tehlikeli durumdan tehlikesiz duruma) sırasında oluşabilecek tehlike durumları
- Kaza sonuçlarının kontrol edilmesini sağlayan yazılımın hata yapması
- Donanım tehlike durumlarının doğrulanması için kullanılan yazılımın hata yapması

bulunur.

Yazılım Alt Sistem Tehlike Analizi sırasında çeşitli hata durumlarını hesaba katmak gerekir. Hesaba katılması gerekli hata durumları arasında:

- Sensörlerin ve tetikleyecilerin belirli değerlerde takılı kalmaları
- Kabul edilen aralığın altında veya üstünde değer okunması
- Kabul edilen aralığa uygun fakat hatalı veri okunması

- Fiziksel birimlerin hatalı çalışması
- Hatalı veri tipi veya büyüklüğü
- Operatör hatalı komut, veri girişi
- Hesaplama yapılırken veri taşması (overflow), alt taşması (underflow)
- Algoritmanın hatalı çalışması
- Sıraya uygun gerçekleşmesi gereken olayların rastgele gerçekleşmesi
- Zamanlama isterlerinin karşılanmaması
- Bellek kullanım problemleri
- Uygun olmayan veri paketi gelmesi veya yazılımın algılayamacağı hızda veri gelmesi
- Veri örnekleme hızının değişimleri algılamak için yeterli olmaması
- Ortak kaynakların bir görev tarafından kilitlenmesi
- Çoklu görev sisteminde ölümcül kilitlenme
- Sistem veya donanım hatalarının yazılıma etki etmesi

bulunur [16].

3.6 Emniyet Kritik Yazılım Geliştirme

Yazılım emniyetinin sağlanması için belli başlı kurallar bulunmaktadır. Bunlar:

- Olası tehlikeli bir olayın herhangi bir olay veya işlem sonucunda tetiklenmemesi
- Emniyetsiz bir durum veya komut tespit edildiğinde
- Olası tehlikelerin engellenmesi
- Daha önceden belirlenmiş emniyetli bir duruma geçiş için gerekli işlemlerin başlatılması

olarak belirlenmiştir [16]. Daha emniyetli yazılım geliştirmek için uyulması gereken kurallar arasında:

- Doğru iletişim
- Doğru yazılım mühendisliği ilkelerinin ve uygulamalarının takip edilmesi
- Emniyet ve yazılım geliştirme analizlerinin yapılması
- Uygun yazılım geliştirme yöntemlerinin, tekniklerinin ve yazılım tasarım yöntemlerinin kullanılması

- Sorumluluğun her zaman satın alan kişiye ait olduğunun unutulmaması

bulunur [16]. Doğru iletişim yazılımın emniyetinin ve geliştirilen sistemin emniyetinin iyileştirilmesi için paha biçilemez bir araçtır. Doğru iletişim sayesinde:

- Yanlış ve/veya eksik anlaşılmalar engellenir
- Riskler, soruna dönüşmeden önce belirlenir
- Tasarım kararlarının ardındaki sebepler açığa çıkar
- Takım üyeleri anormalliklerin, sorunların veya diğer durumların farkına varır
- Yönetim üyeleri, projenin son durumunu daha iyi anlar
- Mühendislerin bilgi ve tecrübeleri artar

Emniyet kritik yazılım sistemleri geliştirilmeden önce kullanılabilecek doğru yazılım mühendisliği ilkelerine ve usullerine karar verilmelidir. Bu karar verme sürecinde en önemli nokta hiç bir usulün yada yöntemin gümüş kurşun olmadığını bilmektir. İyi yazılım zanaatı, emniyetli yazılım geliştirmek için ön koşullardan sadece birisidir. Bunun yanında yapılacak emniyet ve geliştirme analizleri yazılımın emniyet sorunlarını idare edebildiğini doğrulamak için kullanılır.

Yazılım geliştirmek için bir çok yöntem, teknik, ve tasarım karakteristikleri bulunur. Yapılması gereken sisteme uyumlu olanların seçilmesidir. Yazılım emniyetinin sağlanmasında "satın alan sorumludur" prensibine dikkat edilmelidir. Yazılım geliştirmek için satın alınan derleyici, editör, hata bulucu ve işletim sistemi gibi hazır ürünlerde emniyet söz konusu olduğunda hesaba katılmalıdır.

3.7 Emniyet Kritik Yazılım Geliştirme Süreci

Herhangi bir yazılım geliştirme sürecinde, yazılım mühendisleri genel olarak benzer faaliyetleri yaparlar. Bu faaliyetler aşağıda detaylandırılmıştır.

- Geliştirilecek projenin/ürünün içerisinde yazılımın rolünü anlamak için sistem, emniyet ve kalite mühendisleri ile işbirliği yapmak
- Yazılım yönetim ve geliştirme planlarını hazırlamak
- İster analizi yapmak
- İsterlere uygun yazılım tasarımı yapmak
- Tasarıma uygun yazılım geliştirmek
- Yazılımı test etmek

3.7.1 Yazılım yaşam döngüleri

Yazılım yaşam döngüsü, yazılım geliştirme evreleri arasındaki ilişkileri tanımlar. Yazılım geliştirme evreleri genel olarak isterlerin belirlenmesi, yazılım tasarımı, yazılım geliştirme, yazılım testleri ve yazılım doğrulama aktivitelerinden oluşur. Bu aktiviteler seçilen yaşam döngüsüne bağlı olarak birbirini takip edecek veya birbirinin içine girecek şekilde gerçekleştirilebilir. Emniyet kritik yazılım sistemleri geliştirilirken yasal mevzuatlara uyum sağlamak, sertifikasyon ve dokümantasyon faaliyetlerini desteklemek amacıyla ile genellikle birbirini takip eden evrelere sahip yazılım yaşam döngüleri seçilir. Emniyet kritik sistemlerde fonksiyonel güvenlikle ilgilenen uluslararası standartta emniyet kritik sistemlerin V modeli yazılım yaşam döngüsüne uygun geliştirilmesi önerilmektedir [17].

3.7.1.1 V modeli

V modeli doğrulama ve geçerleme modeli olarak tanımlanır. Birbirini takip eden aktivitelerden oluşur. Bir evrenin başlayabilmesi için bir önceki evrenin bitmiş olması gereklidir. Her bir evrenin doğrulama faaliyetleri evreye paralel olarak gerçekleştirilir.

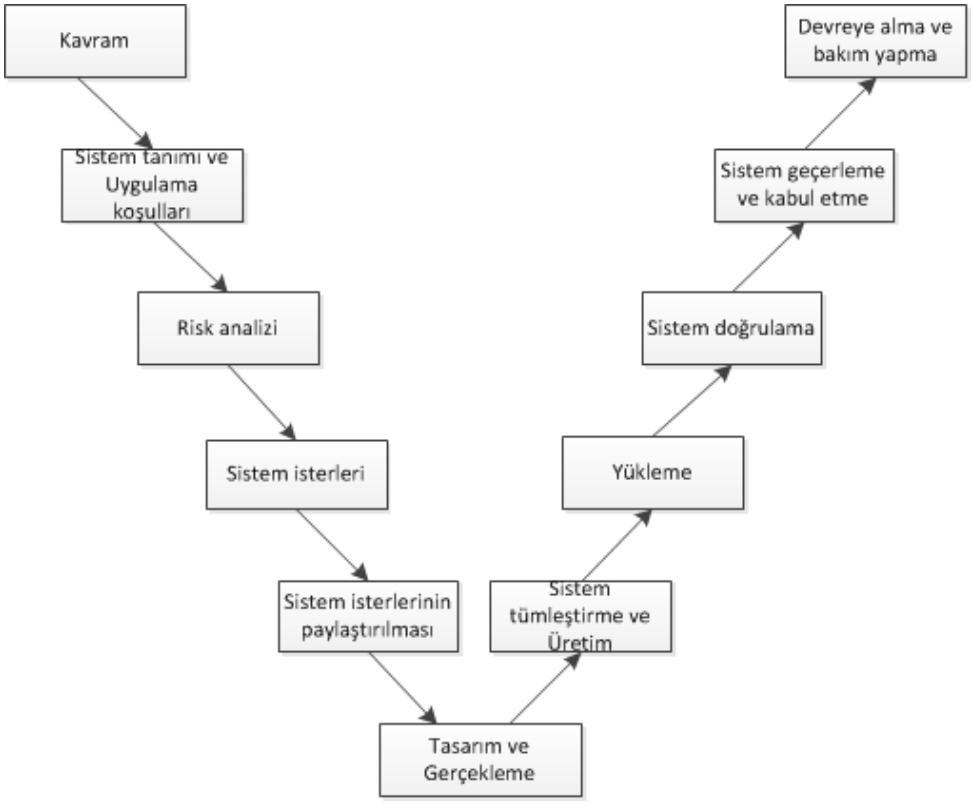
V modelindeki evreler aşağıda detaylandırılmıştır.

İsterler, sistem ve iş isterleri olmak üzere modelin başlangıcında belirlenir. V modelinde tasarım ve geliştirme faaliyetleri başlamadan önce sistem test planı hazırlanır. Sistem test planı hazırlanırken belirlenen isterlerin beklenilen işlevselliği sağlayıp/sağlamadığı dikkate alınmalıdır.

Tasarım, üst seviye ve alt seviye tasarım olmak üzere ikiye ayrılır. Üst seviye tasarım sistem mimarisi ve tasarımı ile ilgilidir. Üst seviye tasarımda kullanılacak hazır ürünler, geliştirme ortamı, servisler ve süreçler belirlenir. Tümleştirme test planı üst seviye tasarımda hazırlanır. Alt seviye tasarımda yazılım bileşenlerine ait tasarım yapılır. Her bir yazılım bileşeninde çalışacak olan algoritmalara ait detaylı tasarım bu evrede yapılır. Yazılım bileşenlerine ait testler bu evrede oluşturulur.

Gerçekleştirme, yazılım kodlama faaliyetlerinin yer aldığı evredir. Kodlama faaliyetleri, birim test geliştirme faaliyetlerini de içerir. Kodlama faaliyetleri tamamlandığında daha önceden hazırlanan test planlarına göre test faaliyetlerine başlanır.

V modeline ait evreler Şekil 3.1'de gösterilmiştir.



Şekil 3.1 : V modeli evreleri.

V modelini kullanmanın avantajları arasında:

- Basit ve kullanımının kolay olması
- Test planlama ve tasarım faaliyetleri kodlama faaliyetleri başlamadan yapılır. Bu şelale modeline göre büyük zaman tasarrufu sağlar
- Önetkin hata takibi – hatalar erken evrelerde bulunur
- Hataların ilerleyen fazlara doğru ilerlemesini engeller
- İsterlerin çok iyi anlaşıldığı küçük projelerde başarılı sonuçlar verir

bulunur.

V modelini kullanmanın dezavantajları arasında:

- Oldukça katı ve esnek olmayan evrelere sahip olması
- Yazılım geliştirmenin sadece gerçekleştirme evresinde yapılması
- Sürecin ortasında herhangi bir değişiklik olması gerekirse, tüm test dokümanlarının ve ister dokümanlarının güncellenmesinin gerekmesi

bulunur.

3.7.2 Yazılım tasarım yöntemleri

Tasarım, inşa edilecek proje yada ürünün mühendislik açısından anlamlı gösterimidir. İyi bir yazılım tasarımı geriye yönelik müşteri isterlerine kadar izlenebilmelidir. Aşağıda listelenen yazılım tasarım yöntemleri en yaygın kullanılan yöntemlerdir.

- Yapısal Analiz ve Yapısal Tasarım (Structured Analysis and Structured Design)
- Nesneye Yönelik Analiz ve Nesneye Yönelik Tasarım (Object Oriented Analysis and Object Oriented Design)
- Formal Yöntemler (Formal Methods)
- Model Tabanlı Geliştirme (Model-based Development)

3.7.2.1 Yapısal analiz ve yapısal tasarım

Sistemi işlevler hiyerarşisi olarak tanımlayan analiz ve tasarım yöntemlerinin hepsine verilen ortak addır. Kullanılan çeşitli yapısal analiz ve tasarım yöntemleri arasında en popüler olan üç tanesi İşlevsel ayrıştırma (Functional decomposition), Yapısal analiz (Data flow) ve Bilgi modelleme (Information modelling)' dir.

İşlevsel ayrıştırma, sistemin yerine getirmesi gereken işlevler, alt işlevler ve bu işlevlerin arasındaki arayüzlerle ilgilenir. Problemi, daha kolay ele alınabilecek alt problemlere bölme ve bu alt problemlerin çözümlerinden esas problemi çözmeye yarayan tekniktir. İşlevsel ayrıştırma yukarıdan aşağıya geliştirme yöntemidir. Soyut ürünle başlayıp, somut ürüne doğru çalışır. Süreç, işlevin alt adımlara bölünmesiyle başlar.

Daha sonra her bir alt adım kendi içinde alt adımlara bölünerek devam edilir. Alt adımlara bölme daha fazla alt adıma bölünemeyecek bir alt adıma ulaşıldığında son bulur.

Bu yöntemin en çok eleştirilen yönleri arasında:

- Tasarım yaşam döngüsü boyunca işlevsel yeteneklerin sık sık değişmesi
- Teklif edilen sistem işlevleri ile sistemi oluşturan işlevler arasındaki bağlantının görülmesinin zor olması
- Yapılacak faaliyetlerin desteklenmesinde verinin ikinci planda olması

bulunur.

Yapısal analiz, 1980'li yıllarda popüler olmuştur ve hala yoğun olarak kullanılmaktadır. Analiz sistem kapsamını (gerçek dünya) veri ve verinin kontrol terminolojisi (veri akış diyagramları ile gösterilir) olarak ele alır. Veri sözlükleri veriyi ve verinin nerede ve nasıl kullanıldığını tanımlar.

Yapısal analiz yapmak için işlem adımları:

- Veri akış diyagramı ile başla
- Giriş biriminden çıkış birimine ana akışı belirle
- Giriş, merkezi dönüştürme ve çıkış süreçlerine ayır
- Yüksek seviye yapı çizelgesine dönüştür
- İyileştir
- Bağımlılık ve uyumluluk açısından doğrula

Bilgi modelleme, varlık-ilişki diyagramlarını kullanır ve nesneye yönelik analizin atasıdır. Analiz, problem uzayında öncelikli olarak nesneleri bulur. Bulduğu nesnelerin niteliklerini, nesnelerin birbiriyle ilişkilerini tanımlar, nesneleri alt ve üst tipler olarak düzenler ve son olarak nesne birliklerini tanımlar. Bu aşamadan sonra genel olarak bir normalleştirme işlemi yapılır.

Yöntemin en çok eleştirilen yönleri arasında:

- Her bir nesnenin işlem ve servis isterleri ele alınmaması
- Miras alma (inheritance) belirgin olarak tanımlanmaması
- Nesneler arasında zayıf arayüz (mesajlaşma) yapılarının bulunması
- Nesnelerinin belirlenmesinde, sınıflandırma ve birleştirme yapılarının öncelikli kullanılmaması

bulunur.

3.7.2.2 Nesneye yönelik analiz ve nesneye yönelik tasarım

Nesneye yönelik paradigmaya dayanır. Bu paradigmaya göre:

- Problem birbirinden bağımsız nesneler olarak ifade edilmelidir
- İlgi alanı nesnelere göre ayrılmalı ve her nesnenin belirli nitelikleri ve davranışları olmalıdır
- Nesneler
    - Birbirlerine mesajlar göndererek, birbirleri üzerinde belirli olayları tetikleyebilmelidir

- Hiyerarşik bir yapıda tasarlanmalı ve alt seviyedeki nesneler, üst seviyedeki nesnelerin tüm nitelik ve davranışlarını miras olarak almalıdırlar
- Alt seviyedeki nesneler miras olarak aldıkları nitelik ve davranışları değiştirebilmeli ve yeni davranışlar ekleyebilmelidirler

Gerçek dünyayı nesnelere göre modellemenin bazı faydaları bulunmaktadır. Bu faydalardan en belirgini bu yöntemin insan düşünme şekline oldukça yakın olmasıdır.

Bunun yanında doğru olarak belirlendikleri takdirde nesneler gerçek dünya problem uzayında oldukça kararlı varlıklar olmaktadırlar. Bunun nedeni nesnenin işlevlerinin, verilerinin, ve komut/mesajlarının sistemin tümünden izole olmasıdır.

Nesneye yönelik analiz soyutlama, bilgi saklama ve miras alma ilkelerinin üzerine kuruludur. Bu ilkelerden soyutlama karmaşık bir problemi, sistemi, fikri yada durumu daha iyi anlayabilmek belirli yönleri ile ele almayı ifade eder. Analistin bakış açısı, sistem nesnelerinin en belirgin olan benzer nitelikleri üzerine odaklanır. Sistem böylece basit bir şekilde ifade edilebilmektedir

Bilgi saklama, değişme ihtimali bulunan gereksinimlerin saklanması anlamına gelir. Aynı zamanda karmaşıklığın yönetimini kolaylaştırır.

Miras alma, bir sistemdeki nesneler arasındaki ilişkiyi tanımlar. Bu ilişkide bir nesne, diğer nesne veya nesnelerin yapısını yada davranışlarını miras alır.

Nesneye yönelik analiz ve tasarım yöntemi ile ilgili eleştiriler aşağıda detaylandırılmıştır.

- Her problem ilgi alanı nesneye yönelik analiz ve tasarım için iyi bir aday değildir
- Soyut varlıklar için nesneler belirlemek zordur
- Büyük oranda yeniden kullanım ve tümleştirme için güçsüzdür
- Sistemin ayrıştırılması için güçsüzdür
- Birden çok takım ve dağıtık geliştirme için güçsüzdür
- Nesneye yönelik test etme yöntemleri halen yeterince gelişmemiştir
- Gelişen sistemlerdeki nesneye yönelik modeli değiştirmek ifade edildiği kadar basit değildir

3.7.2.3 Biçimsel yöntemler (Formal methods)

Biçimsel Yöntemler isterlerin, bilgisayar sistem ve yazılımlarının belirlenmesi ve doğrulanması için matematiksel modelleme temeline dayanan bir takım teknikler ve araçlar bütünüdür [18]. Yöntemlerin belirleme ve doğrulama olmak üzere iki kısmı bulunur.

Yazılım ve sistem isterleri genellikle insanlar tarafından okunabilen ve anlaşılabilen bir dilde yazılırlar. Fakat iki ayrı insan aynı cümleyi farklı yorumlayabilir ve buda karmaşıklığa yol açabilir. Bu karmaşıklığın önlenmesi için isterler biçimsel, matematiksel bir dille yazılabilir. Bu tekniğe biçimsel belirleme denir.

Biçimsel doğrulama ise biçimsel belirleme tekniği ile belirlenen yazılım isterlerini doğrulanması için bir ispat tekniğidir. Doğrulama sürekli ilerleyen bir aktivitedir. Her bir aşamada ortaya çıkan yeni ürün eski ürünle tutarlı olması için biçimsel olarak doğrulanır. Örneğin ayrıntılı tasarım, başlangıçtaki tasarımla karşılaştırılır ve doğrulanır.

Emniyet kritik veya yüksek güvence gerektiren sistemlerin üretim aşamasında biçimsel yöntemler güvenilir yazılım sistemi için en yüksek güvenceyi verir.

Biçimsel yöntemler:

- Olay gerçekleştikten sonra yazılım güvencesi sağlama amaçlı
- Paralel olarak yazılım güvencesi sağlama amaçlı
- Yazılım geliştirme yöntemi olarak

kullanılırlar.

3.7.2.4 Model tabanlı yazılım geliştirme (Model-based development)

Model tabanlı yazılım geliştirme yazılım sisteminin eksiksiz bir modelini geliştirmek üzerine odaklanır. Modeller, sistemin soyut ve yüksek seviyeli bir tanımıdır. Modeller belirli bir modelleme dilinde cümleler veya modelleme aracında ifade edilirler. Standart tasarım dokümanlarının aksine modeller çalıştırılabilirler (Sistem içindeki süreç akışınının benzetilebilmesi amaçlı).

Bu yöntemde standart olan ister belirleme, tasarım, kodlama, birim, tümleşim, ve sistem testi döngüsü, ister belirleme, model, doğrulama (test) ve hata ayıklama, kod üretme ve sistem testi halinde takip edilir.

3.7.2.5 Tasarım kalıpları (Design patterns)

Yazılım mühendisliğinde, teker düzenli olarak yeniden icat edilir. Yeniden kullanılır yazılım bileşenleri oluşturmak bu tekrar icat etme sürecinin engellenmesi için bir yoldur. Tasarım kalıpları ise diğer bir yoldur. Yeniden kullanılabilir yazılımın tersine tasarım kalıpları birer yazılım bileşeni değildirler. Daha çok fikirleri ifade ederler. Yazılım mühendisliğinde tekrarlayan problemler için geliştirilmiş kanıtlanmış, çözümlerdir.

Tasarım kalıpları fikri birçok kaynaktan beslenmiştir. Bunlar arasında Christopher Alexander tarafından başlatılan mimari tasarım hareketi, yazınsal programlama (literate programming) ve her meslek dalında öğrenilen dersler ve en başarılı uygulamaların dokümantasyonu vardır. Belirli bir sistem için geliştirilen yazılım mühendisliği çözümleri genellikle o sistem kapsamına özeldir. Tasarım kalıpları ise özel çözümlerin problemin çözümüne kattığı püf noktaların genelleştirilmiş halidir.

Yazılım tasarım kalıplarının 4 temel elemanı bulunur. Bunlar:

- Kalıp ismi
- Problem tanımı. Bu tanımda problem ve problemin bağlamı, uyulması gereken şartlar ve tasarım kalıbının ne zaman uygulanması gerektiği tanımlanır.
- Çözüm. Tasarımı oluşturan elemanlar, elemanların birbirleriyle ilişkileri, sorumlulukları ve işbirlikleri tanımlanır.
- Sonuçlar. Tasarım kalıbını uygulamanın avantajları ve dezavantajları tanımlanır. Dezavantajlar genelde uygulama uzayı ve çalışma kısıtları ile ilgilidir.

3.7.3 Yazılım isterleri

İster analizi ve yönetimi başarılı ve emniyet kritik bir yazılım geliştirme sürecinin olmazsa olmaz aktivitesidir. Yazılımın sorumlu olduğu belirlenen bir çok maliyetli ve kritik sistem arızalarının hatalı, tam anlaşılmamış veya yetersiz isterlerden kaynaklandığı belirlenmiştir.

Yazılım isterlerlerinin belirlenmesi için bir çok kaynak vardır. Bunlar arasında:

- Sistem isterleri
- Emniyet ve güvenlik standartları

- Tehlike ve risk analizleri
- Sistem kısıtları
- Müşteri talepleri
- Tecrübeye dayalı olarak edinilen yazılım emniyeti uygulamaları

bulunur.

En başarılı yazılım emniyeti uygulamaları (genel isterler) önceki tecrübelerden edinilmesi sebebiyle yazılım isterlerinin belirlenmesinde dikkate alınmalıdır. Örneğin hata idare mekanizması, arıza tespit ve kurtarma mekanizması, veya kritik hatada sistemi emniyetli bir duruma getirme mekanizması genel isterlere birer örnektir.

Her bir ister beklenen işlevleri ve performans beklentilerini karşılaması için bütünlük, açıklık ve doğrulanabilirlik açısından belirlenmeli ve analiz edilmelidir. Bunun yanında sistem, emniyet kritikliği açısından değerlendirilmelidir. Yukarıdan aşağı analiz yöntemlerinden Yazılım Hata Ağacı Analizi (Software Fault Tree Analysis) emniyet kritik yazılım isterlerinin belirlenmesi için sıkça başvurulan bir yöntemdir. Belirlenen her bir ister iyi yönetilmeli, geliştirme sürecinden son test durumlarına kadar takibi yapılmalıdır. İsterler doğru olarak belirlendiği takdirde sistem ve kabul testi planlarını yapmak kolay olacaktır.

3.7.4 Yazılım tasarımı

Yazılım tasarımı, isterlerin yazılım kodlarına dönüştürülmesi için bir yapı tanımlar. Bu yapı yazılımı geliştirecek kişiye isterleri alt seviye yazılım kodlarına dönüştürmeden önce yüksek seviyeli olarak düşünme fırsatı verir.

Emniyet kritik sistemler için yazılım tasarımı yapma süreci:

- Tasarım özelliklerinin ve işlevlerin belirlenmesi
- Her bir istere tasarımda yer verilmesi
- Emniyet kritik yazılım bileşenlerinin tespit edilmesi
- Tutarlılık ve işlevselliğin sağlanması için tasarım analizlerinin yapılması
- Yazılım tasarımının emniyet analizinin yapılması
- Yazılım tümleştirme planlarının geliştirilmesi, gözden geçirilmesi, sistem ve kabul testi planlarının yapılması

faaliyetlerini içerir.

3.7.4.1 Yazılım emniyeti için yazılım tasarımı

Emniyet kritik yazılım için en önemli nokta en az risk içerecek şekilde tasarımın yapılmasıdır. En az risk, tehlike riskleri, yazılım hatarı riskleri, kullanıcı hataları riskleri ve diğer tüm riskleri içermelidir. Mümkün olduğunda tehlike riskleri ve diğer tüm riskler tasarım sırasında elenmelidir.

Risklerin azaltılması için:

- Yazılım ve yazılım arayüzlerinin karmaşıklığının azaltılması
- Kullanıcı dostu kriteri yerine kullanıcı emniyeti düşünülerek yazılım tasarımı yapılması
- Geliştirme ve tümleştirme süresince test edilebilirlik için tasarım yapılması
- Tasarıma daha fazla kaynak (zaman, efor vb.) harcanması

gereklidir.

3.7.4.2 Yazılım sürdürülebilirliği (maintainability) için yazılım tasarımı

Sürdürülebilir yazılım geliştirmek için çeşitli yazılım metriklerinden yararlanılabilir. Bu amaçla yazılım tasarım kalitesi ile ilgili metrikler sürdürülebilirlik bakış açısıyla değerlendirilmelidir. Nesneye yönelik yazılım tasarım paradigmasından önce geliştirilen metrikler arasında çevrimsel karmaşıklık, kod satır sayısı ve yorum satır sayısı vb. bulunmaktadır. Nesneye yönelik yazılım paradigması ile geliştirilen metrikler arasında sınıf ağırlıklı metot sayısı, sınıfın tetiklediği metot sayısı, sınıf metotları arasındaki uyumsuzluk, nesneler arasındaki bağımlılıklar, kalıtım ağacı derinliği, ve alt sınıf sayısı verilebilir. Bunun yanında Software Engineering Institute (SEI) tarafından geliştirilen Sürdürülebilirlik Indeks metrik değeri çeşitli metriklerin bir araya gelmesiyle hesaplanır ve yaygın olarak kullanılmaktadır [19].

Yazılım metriklerinin yanında bakımı kolay yazılım geliştirmek için yapılabilecek faaliyetler arasında:

- Erken planlama – yazılımın nasıl ve ne şekilde değişebileceğini ön görme
- Modüler tasarım – alt kümeler tanımlama, işlevselliği basitleştirme
- Nesneye yönelik tasarım
- Aynı geleneklerin kullanılması
- İsimlerde standardının kullanılması
- Kodlama standardının kullanılması

- Dokümantasyon standardının kullanılması
- Ortak araçların kullanılması
- Konfigürasyon yönetiminin kullanılması

sayılabilir.

3.7.5 Yazılım gerçekleme (Kodlama)

Yazılım gerçekleme, yazılım tasarımının alınıp belirli bir programlama dilinde kodlanması faaliyetleridir. Yazılım gerçekleme (kodlamaI programcının kafasının içinde saklı, yapısal olmayan adımlardan oluşur. Bu nedenle kritik tasarım kararlarının alt seviye programcıya bırakılması tavsiye edilmez. Emniyet kritik yazılımlarda gerçekleme faaliyetleri yapılırken takip edilmesinde yarar olan bazı aktiviteler bulunmaktadır. Bunlar arasında:

- Kodlama kontrol listeleri
- Kodlama standartları
- Birim seviye testler
- Yapılal iyileştirme (refactoring)

bulunmaktadır.

3.7.5.1 Kodlama kontrol listeleri ve standartları

Yazılım geliştiren kişiler yazılım gerçekleme için kodlama kontrol listelerinden yararlanmalıdır. Kodlama standartları, yazılım geliştiren kişinin sakınması gereken emniyetsiz işlevlerin ve prosedürlerin belirlendiği belgelerdir.

Bu işlevlere ve prosedülere ek olarak programlama stil bilgileri, isimlendirme ve yorum biçimleri, ve yazılım bileşenleri arasında tutarlılık sağlayacak yöntemleri içeririrler. Kodlama standartları, yazılım tasarım evresinde belirlenmeli ve gerçekleme (kodlama) evresinde kullanılmalıdır.

Sık kullanılan kodlama kontrol listeleri aşağıda detalandırılmıştır.

- Emniyet kontrol listesi (Projedeki emniyet isterlerini içerir)
- Koruyucu programlama (Defensive programming) kontrol listesi
- Kod gözden geçirme kuralları kontrol listesi
- İster kontrol listesi

3.7.5.2 Birim seviye test etme

Birim seviye testler detaylı tasarım evresinde, birim ile ilgili metotlar tamamlandığında planlanır. Kod derlenebilir duruma geldiğinde ise çalıştırılabilir. Birim testler için temel kriter birimin hatasız derlenebilir olmasıdır. Her bir birim hatasız işlevselliğin doğrulanması için test edilir. Birim seviye testler yazılım tümleştirildiğinde ulaşılabilir olmayabilecek yazılım detaylarına erişebildiği için önemlidir. Bunun yanında gerçekleme hatalarını yada performans problemlerini ortaya çıkarma açısından yararlıdırlar. Emniyet kritik olan tanımlanan her birim için özel birim emniyet testleri tasarlanmalıdır. Bu testler için emniyet test isterleri kapsama analizi yapılmalıdır.

Birim testlerin iki tipi vardır. Bunlar beyaz kutu, kara kutu testleridir. Beyaz kutu testleri bileşenin iç yapısı ile ilgili detayların test edilmesi için kullanılırlar. Kara kutu testleriyse bileşenin verilen bir giriş değerine göre nasıl bir çıkış değeri ürettiği ile ilgilenirler. Kara kutu testlerinde bileşenin içinde gerçekleşenler ile ilgilenilmez. Beyaz kutu testlerinde dallanma, döngü gerçeklemesi vb. detaylar test edilir. Kara kutu testlerinde giriş değer aralıkları, çıkış değer aralıkları ve hata idaresi test edilir.

Nesneye yönelik yazılım testleri sınıflara ait işlevlerin testlerini kapsar. Bu kapsamda test etme yapısal programlardaki prosedür ve işlevlerin test edilmesiyle bezerdir. Nesneye yönelik yazılımlarda yapıcı, yıkıcı ve nesne kopyalama işlevlerininde test edilmesi gereklidir.

3.7.5.3 Yapısal iyileştirme (Refactoring)

Yapısal iyileştirme nesneye yönelik yazılan kodun belirli bir disiplinle yeniden yapılandırılmasıdır. Yazılımın daha esnek ve yeniden kullanılabilir hale getirilmesi faaliyetleridir.

Bu faaliyetlerin yürütülmesinin en önemli amacı yazılımı sürdürülebilir kılmaktır. Yapısal iyileştirmenin faydaları arasında:

- Kopya kodun tekleştirilmesi
- Kodun okunabilirliğinin artırılması
- Kullanılmayan dolaylamanın kaldırılması
- Halihazırda bulunan algoritmaların değişme ihtiyacı bulunan kısımlardan izole edilmesi

- Algoritmalar arasında kolay geçiş sağlanması
- Uygulamanın daha hızlı çalışmasının sağlanması

sayılabilir.

3.7.6 Yazılım testleri

Test etme, yazılım bileşeninim gerçek ve benzetilmiş bir ortamda kullanıma hazır olarak çalıştırılmasıdır. Test etme çeşitli amaçlara hizmet eder.

Bunlar arasında:

- Hataların tespit edilmesi
- Sistemin veya sistem bileşenlerinin doğrulanması
- İşlevsellik, performans ve emniyet isterlerinin doğrulanması

bulunur.

Test etme, genellikle doğrulama ve geçerleme üzerine odaklanır. Ancak hata tespiti test etmenin en önemli yararıdır. Testler sayesinde yazılım içindeki kalite test edilemez ama mümkün olduğunca çok hata tespit edilip; giderilebilir.

Yazılım üzerinde çeşitli tipte testler yapılabilir. Birim testler yazılım bileşenlerinin izole olarak test edilmesini sağlar. Tümleştirme testleri bileşenlerin birleştirilmesi esnasında koşturulur ve arayüz ve bileşen etkileşimi üzerine odaklanır.

Sistem testleri bileşenler tam anlanıyla birleştirildikten sonra yapılan testlerdir. İşlevsellik, performans, yük, stres, emniyet, ve kabul testleri diğer test tiplerindendir.

Test etmenin temel ilkeleri aşağıda detaylandırılmıştır.

- Her test durumu geri dönük olarak isterlere kadar izlenilebilmelidir
- Test durumları, test etme faaliyetleri başlamadan planlanmalıdır
- Hataların %80’ si yazılım bileşenlerinin %20’ sinde bulunur prensibi ile hareket edilmelidir
- Test etmeye küçük birimlerle başlanmalı ve testi tamamlanan küçük birimler sisteme dahil edilerek devam edilmelidir
- Her durum test edilemez prensibi ile hareket edilmelidir
- Mümkünse testler bağımsız bir ekip tarafından yapılmalıdır

4. YAZILIM KALİTESİ VE TEST EDİLEBİLİRLİĞİ

Yazılım kalitesi genel olarak yazılımın kendisinden beklenenleri ne ölçüde karşıladığıdır [20]. Yazılım yaşam döngüsü boyunca yazılım kalitesinin sağlanması en zorlayıcı aktivitelerden birisidir. Zaman içinde yazılım kalitesine etki eden karakteristikleri tanımlamak için bazı kalite modelleri önerilmiştir.

Kalite modelleri yazılım ekibine:

- Yazılım gereksinimlerinin anlaşılır olarak tanımlanması
- Gereksinimlerin kapsamının doğrulanması
- Yazılım tasarım hedeflerinin tanımlanması
- Yazılım test hedeflerinin tanımlanması
- Kalite kontrol koşullarının tanımlanması
- Tamamlanan yazılım ürününün kabul koşullarının tanımlanması

hususlarında yardımcı olur.

Yazılım kalite tanımı, yazılımdan açıkça belirtilen ve ima edilen beklentilerin kişiden kişiye değişebileceği için değişiklik gösterebilir. Son kullanıcı yazılım kalitesi yazılım ürününün kolay kullanılabilirliği iken yazılım mühendisi için kod okunabilirliği olabilir.

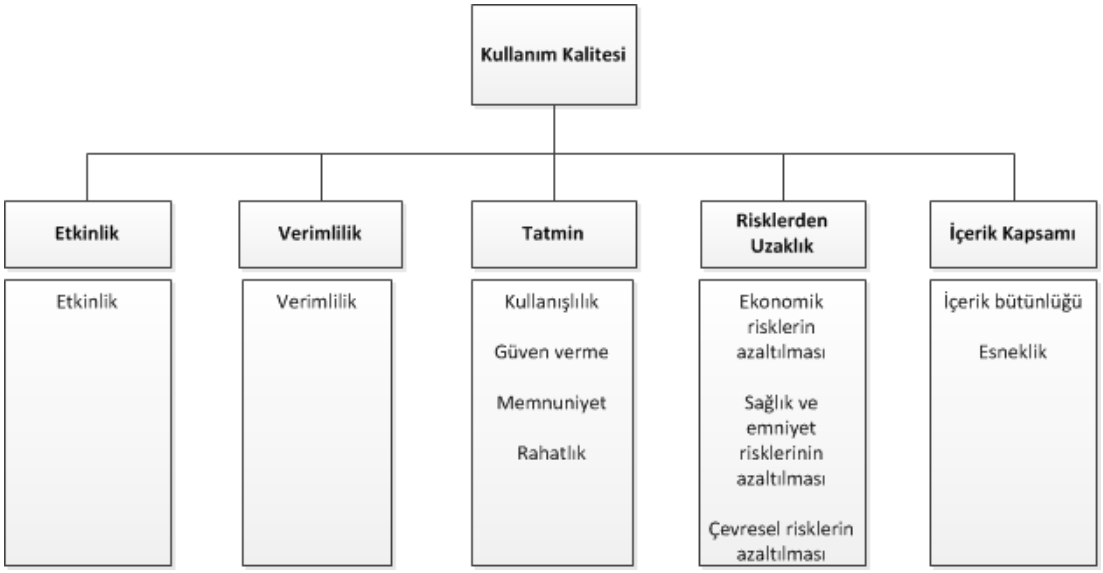
4.1 ISO/IEC 25010:2011

ISO/IEC 25010:2011 yaygın kabul görmüş bir kalite standardıdır [21]. Bu uluslararası standart "ürün kalitesi" ve "kullanım kalitesi" başlıklarında iki adet kalite modeli tanımlar. Kullanım ve ürün kalitesi modelleri tarafında tanımlanan karakteristikler herhangi bir bilgisayar sistemi ve yazılım ürününe uygundur. Bu iki modelde tanımlanan karakteristikler sistem ve yazılım ürün kalitesinin tanımlanması, ölçülmesi ve değerlendirilmesi için teknik bir dil sağlar. Kullanım kalitesi modelinde 5 adet ana karakteristik tanımlıdır. Bu karakteristikler yazılım ürünü belirli bağlamda kullanıldığı sürece geçerlidir.

Ürün kalite modelinde ise bilgisayar sisteminin dinamik nitelikleri ve yazılımın durağan nitelikleri ile ilgili 8 adet karakteristik tanımlıdır.

4.1.1 Kullanım kalite modeli (Quality in use model)

Kullanım kalite modeli, yazılımın, donanımın ve çalışma ortamının kullanım kalitesinin ölçülmesi ile ilgilenir [21]. Kullanıcıların, kullanıcı hedeflerinin ve sosyal çevrenin karakteristikleri kullanım kalitesini etkileyebilir. Modelin tanımladığı karakteristikler Şekil 4.1'de gösterilmiştir.



Şekil 4.1 : Kullanım kalite modeli.

Kullanım kalitesi modelinin 5 adet ana karakteristiği ve bu karakteristikler altında tanımlı alt karakteristikleri vardır. Bunlar:

- Etkinlik
- Verimlilik
- Tatmin
    - Kullanışlılık
    - Güven verme
    - Memnuniyet
    - Rahatlık
- Risklerden uzaklık
    - Ekomik risklerin azatılması

- Sağlık ve emniyet risklerinin azaltılması
- Çevresel risklerin azaltılması

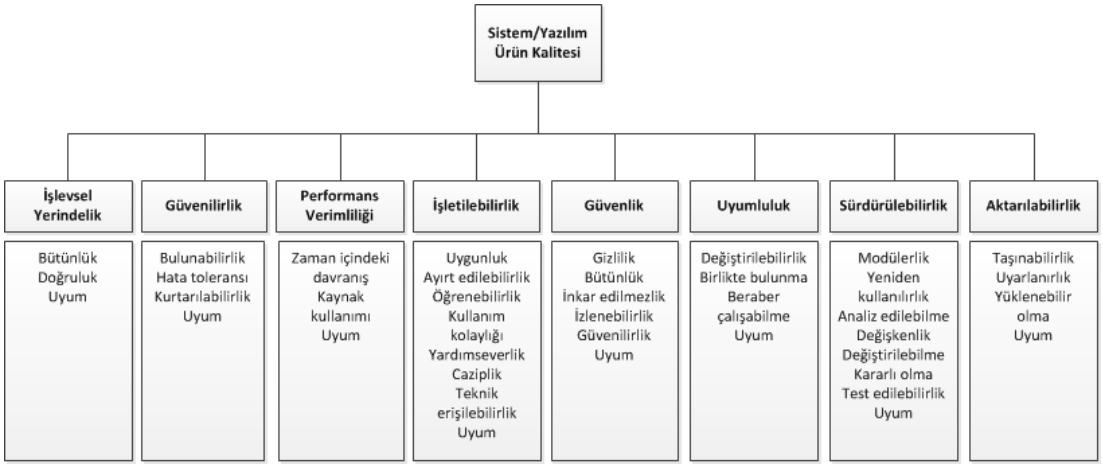
- İçerik kapsamı
  - İçerik bütünlüğü
  - Esneklik

olarak tanımlıdır.

Kullanım kalite modeli insan-bilgisayar sistemini tamamlamak ile ilgilidir. Bu modelde tanımlı bulunan karakteristikler çoğunlukla insan faktörü ile ilişkilidir.

4.1.2 Ürün kalite modeli (Product quality model)

Ürün kalite modelinde ürün kalitesi ile ilgili 8 adet ana karakteristik ve bu karakteristiklerin altında tanımlı bulunan çeşitli sayıda alt karakteristikler bulunmaktadır [21]. Ürün kalite modeli Şekil 4.2'de gösterilmiştir.



Şekil 4.2 : Ürün kalite modeli.

Ürün kalite modeli yazılım mühendisliği kalite beklentilerini karşılamaya daha uygun bir modeldir. Ürün kalite modelinde bulunan kalite karakteristikleri aşağıda detaylandırılmıştır.

İşlevsel yerindelik, sistem/yazılım ürününün belirtilen gereksinimleri karşılaması olarak tanımlanır. Bütünlük, doğruluk ve uyum işlevsel yerindelik altında tanımlı olan alt kalite karakteristiklerdir.

Performans verimliliği, sistem/yazılım ürününün performansının değerlendirilmesi ile ilgilidir. Zaman içindeki davranış, kaynak kullanımı ve uyum performans verimliliği altında tanımlı olan alt kalite karakteristikleridir.

Güvenilirlik, sistem/yazılım ürününün güvenilir olması ile ilgilidir. Bulunabilirlik, hata toleransı, kurtarılabilirlik ve uyum güvenilirlik altında tanımlı olan alt kalite karakteristikleridir.

İşletilebilirlik, sistem/yazılım ürününün işletilir olması ile ilgilidir. Uygunluk, ayırt edilebilirlik, öğrenilebilirlik, kullanım kolaylığı, yardımseverlik, caziplik, teknik erişilebilirlik ve uyum işletilebilirlik altında tanımlı alt kalite karakteristikleridir.

Güvenlik, sistem/yazılım ürününün güvenli olması ile ilgilenir. Gizlilik, bütünlük, inkar edilmezlik, izlenebilirlik, güvenilirlik ve uyum güvenlik altında tanımlı olan alt kalite karakteristikleridir.

Uyumluluk, sistem/yazılım ürününün uyumlu olması ile ilgilidir. Değiştirilebilirlik, birlikte bulunma, beraber çalışabilme ve uyum uyumluluk altında olan alt kalite karakteristikleridir.

Sürdürülebilirlik, sistem/yazılım ürününün sürdürülebilir olması ile ilgilidir. Mühendislik disiplinlerinde sürdürülebilirlik üründeki kusurların veya kusurların oluşma nedenlerinin önlenmesi veya düzeltilmesi, arızalı veya yıpranmış bileşenlerin, sorunsuz çalışmakta olan bileşenlere dokunmadan değiştirilmesi ve beklenmeyen arızaların önüne geçilmesi kolaylığı ile ilgilidir. Ayrıca ürünün kullanım ömrünün, etkinliğinin, güvenilirliğinin, emniyetinin azami süreye çıkarılması, ürünle ilgili yeni gereksinimlerin karşılanması, ürünün gelecekteki bakım faaliyetlerinin kolaylaştırılması ve ürünün değişen çevre koşullarına uyum sağlaması sürdürülebilirlik altında ele alınır. Bazı durumlarda sürdürülebilirlik ürünün sürekli iyileştirme faaliyetlerini içerir. Sürekli iyileştirme geçmişten edinilen tecrübenin ürünün sürdürülebilir ve güvenilir olması için sürekli kullanılması olarak tanımlanır. Telekomünikasyon ve bazı mühendislik dallarında sürdürülebilirlik fonksiyonel bir birimin bakım faaliyetlerinin önceden tanımlanmış gereksinimlere uygun olarak yapılması olarak tanımlanır. Modülerlik, yeniden kullanılabilme, analiz edilebilme, değişkenlik, değiştirilebilme, kararlı olma, test edilebilirlik ve uyum sürdürülebilirlik altında tanımlı olan alt kalite karakteristikleridir.

Aktarılabilirlik, sistem/yazılım ürününün aktarılabilir olması ile ilgilidir. Taşınabilirlik, uyarlanabilirlik, yüklenebilir olma ve uyum aktarılabilirlik altında tanımlı olan alt kalite karakteristikleridir.

Yazılımın yukarıda açıklanan kalite karakteristiklerine göre değerlendirebilmek için bu karakteristikleri ölçülebilir hale getirmek gereklidir. Bu amaçla yazılım metrikleri kullanılır.

4.2 Yazılım Metrikleri

Herhangi bir varlığı tanımlamak için o varlığa ait niteliklere ihtiyacımız vardır. Bu nitelikler sayılar veya sembollerden oluşur [22].

Fenton tarafından verilen ölçme tanımına göre:

"*Ölçme, varlıkları açıkça belirlenmiş kurallara göre tanımlayabilmek için varlıklara sayılar veya sembollerden oluşan nitelikler verme işlemidir*."

Metrik, herhangi bir sistemin belirli bir niteliğinin ölçülmesi için gerekli özellik olarak tanımlanır. Yazılım metrikleri, yazılım mühendisliği disiplini içinde ölçme ile ilgili çeşitli aktiviteleri tanımlar. Bu aktiviteler yazılımın kodlanmasından yazılımın tasarımına kadar çeşitli özelliklerini tanımlayan sayılar olabilir. Elde edilen ölçümler mühendislere yazılıma ait kaynak verimliliği veya yazılım kalitesi hakkında tahminler yürütme olanağı vermektedir.

Yazılım metrikleri aynı zamanda yazılım mühendislerine kaynak kod okumadan yazılıma ait yapı ve tasarımı anlama olanağı vermektedir.

Yıllar içinde yazılımın boyutu, karmaşıklığı ve bağımlılığı ile ilgili birçok metrik tanımlanmıştır. Nesneye yönelik yazılım metrikleri bu metrikler arasında en fazla kabul gören ve kullanılan metrikler olmuşlardır.

4.2.1 Nesneye yönelik olmayan yazılım metrikleri

4.2.1.1 Kaynak kod satır sayısı

Kaynak kod satır sayısı (LOC) yazılım kaynak kodu içindeki tüm satırların (çalıştırılacak kod ve yorum) toplamıdır. Metrik hesaplanırken boş satırlar genellikle hesaba katılmaz.

4.2.1.2 Kaynak kod yorum satır sayısı

Kaynak kod yorum satır sayısı yazılım kaynak kodu içindeki tüm yorum satırlarının toplamıdır. Hesaplanmasının amacı kodun iyi dokümante edilip edilmediğini görmektir.

4.2.1.3 Çevrimsel karmaşıklık

Çevrimsel karmaşıklık, yazılımın karmaşıklığını ölçmek için kullanılan bir yazılım metrik değeridir. Yazılım kaynak kodu içerisindeki birbirinden bağımsız, doğrusal dal sayısına eşittir. 1976 yılında Thomas J. McCabe, Sr. tarafından bulunmuştur. Çevrimsel karmaşıklığın kullanım alanlarından birisi temel dal test yöntemidir. Bu yönteme göre yazılım kodu içindeki birbirinden bağımsız, doğrusal olan her bir dal ayrı olarak test edilmelidir. Bu yöntemle yapılan testlerde test durumu sayısı dal sayısına eşit olacaktır [23].

4.2.2 Nesneye yönelik yazılım metrikleri

4.2.2.1 Chidamber ve Kemerer (CK) metrikleri

Yazılım geliştirme süreçlerindeki iyileştirme ihtiyacı paydaşları nesneye yönelik yazılım geliştirme gibi yeni yaklaşımlara yöneltmiştir. Chidamber ve Kemerer tarafından tanımlanan 6 adet nesneye yönelik metrik bu sebeple oldukça kabul görmüştür [24]. Tanımlanan bu metrikler oldukça iyi teorik ve matematik temele oturtulmuştur. CK metriklerinden önce önerilen metrikler teorik ve matematik temelleri yeterince iyi olmadığı için eleştirilmişlerdir. Bu metrikler daha önceden önerilmiş ölçme ilkelerine göre değerlendirilmişlerdir. CK metrikleri aşağıdaki paragraflarda açıklanmıştır.

Sınıf başına ağırlıklı metot sayısı (Weighted methods per class (WMC))

Tanım: C_1 bir sınıf olsun. Bu sınıfa ait M_1, … ,M_n metot ve bu metotların karmaşıklığı sırasıyla c_1, … ,c_n olsun. Bu durumda metriğin değeri eşitlik **(4.1)**'deki gibi hesaplanır.

$$WMC = \sum_{i=1}^{n} c_i \qquad \textbf{(4.1)}$$

olarak hesaplanır. Karmaşıklık CK metriklerinde açıkça tanımlı değildir. Karmaşıklığın hesaplanma yöntemi analizcinin tercihine bırakılmıştır. Genellikle McCabe tarafında tanımlanan çevrimsel karmaşıklık hesaplama yöntemi tercih edilir. Bu yöntemde doğrusal olarak her bir çalışma yolu hesaplanır.

Eğer bir sınıfın n adet metodu varsa ve her bir metodun ölçülen çevrimsel karmaşıklığı 1 ise WMC sınıfın metot sayısı olan n değerine eşit olacaktır.

Metriğin değerlendirilmesi:

- Sınıftaki metotların karmaşıkları ve metot sayısı sınıfının geliştirilmesi ve bakımının yapılabilmesi hakkında fikir verecektir
- Çok sayıda metoda sahip olan sınıflar, bu sınıflardan türemiş olan sınıflara olumsuz etki ederler
- Çok sayıda metoda sahip olan sınıflar uygulamaya özeldir ve yeniden kullanılabilirlikleri zayıftır
- Çok sayıda metoda sahip olan sınıflar daha karmaşık ve hataya açıktırlar

Kalıtım ağacı derinliği (Depth of inheritance tree (DIT)

Tanım: Bir sınıfın kalıtım ağacının derinliği sınıfın kalıtım derinliğidir. Eğer çoklu kalıtım mevcutsa, DIT sınıfın türediği sınıflardan kalıtım ağacının kökündeki en uzak sınıfa olan uzaklıktır.

Metriğin değerlendirilmesi:

- Kalıtım ağacında derinde olan sınıflar çok sayıda metodu miras almaları nedeniyle daha karmaşıktırlar.
- Kalıtım ağacında derinde olan sınıflar hataya açıktırlar ve bu sınıfların davranışlarının tahmin edilmesi zordur
- Kalıtım ağacında derinde olan sınıfların tasarımları daha karmaşıktır
- Kalıtım ağacında derinde olan sınıflar miras alınan metotları yeniden kullanmaya eğilimlidirler

Lorenz ve Kidd tarafından önerilen bir sınıfa ait kalıtım eşik değeri 6'dır. Kalıtım ağacında bu eşik değerinden daha fazlasına sahip olan sınıflar hataya açık ve karmaşık olarak değerlendirilir [25].

Alt sınıf sayısı (Number of children (NOC))

Tanım: Bir sınıfın kalıtım ağacında doğrudan sahip olduğu alt sınıf sayıdır.

Metriğin değerlendirilmesi:

- Çok sayıda alt sınıfa sahip olan sınıfların yeniden kullanılabilirlik eşikleri yüksektir
- Çok sayıda alt sınıfa sahip olan sınıflar uygun olmayan soyutlama göstergesi olabilir
- Çok sayıda alt sınıfa sahip olan sınıflar yazılım tasarımını daha çok etkileyen sınıflardır
- Çok sayıda alt sınıfa sahip olan sınıflara ait metotlar daha çok test edilmelidirler

Sınıf nesneleri arasında bağımlılık (Coupling between object classes (CBO))

Tanım: Nesne sınıfları arasındaki bağlantı sayısı bir sınıfa bağlı olan sınıfların sayısıdır. Eğer bir C_1 sınıfı, başka bir C_2 sınıfının nitelik ya da metotlarını kullanıyorsa, C_1 sınıfı C_2 sınıfına bağlı olduğu kabul edilir. Bir sınıftan türetilen alt sınıflar türetildikleri sınıfa bağlı kabul edilir.

Metriğin değerlendirilmesi:

- Birbirine sıkı sıkıya bağımlı bulunan sınıflar modüler tasarıma engel olması ve yeniden kullanılabilirliği düşürmesi nedeniyle kaçınılmalıdır
- Sınıf nesneleri arası bağımlılıkların modüler tasarım için sınırlı sayıda tutulması gereklidir. Fazla sayıda bağımlılık içeren sınıflar yazılımı kırılgan hale getirirler. Bu tip sınıfların sürdürülebilirliği düşüktür.
- Bu metrik yazılımın farklı bölümlerinin test edilebilirliği tahmin edilmesi açısından yararlıdır. Sınıf nesneleri arasında fazla sayıda bağımlılık bulunması sınıfın test edilmesini ve hata bulunması zorlaştırır

Sınıfın tetiklediği metot sayısı (Response for a class (RFC)

Tanım: $\{M\}$ bir sınıftaki tüm metotların kümesi ve $\{R_i\}$ i. metot tarafından çağrılan metot kümesi olsun. Bu durumda metrik değerinin hesaplanma yöntemi **(4.2)** ve **(4.3)** eşitliklerinde verilmiştir.

$$RS = \{M\} \bigcup_{all\ i} R_i \quad \textbf{(4.2)}$$

$$RFC = |RS| \quad \textbf{(4.3)}$$

Metriğin değerlendirilmesi:

- Bir sınıfın nesnesine gelen bir mesaja cevap olarak çok sayıda metot çalıştırılıyorsa, test personelinin sınıfı test edebilmek ve hatadan arındırmak için sınıfın detaylarını anlaması gereklidir
- Çok sayıda metodun çalıştırılması daha karmaşık bir sınıf yapısına neden olur
- Verilebilecek en fazla cevap sayısının bilinmesi sınıfın test edilebilme süresinin tahmin edilmesine yardımcı olur

Metotlar arasındaki uyum eksikliği (Lack of cohesion of methods (LCOM))

Tanım: C_1 bir sınıf olup n adet metoda sahip olsun (M_1, … ,M_n). $\{I_i\}$ ise M_i metodunun eriştiği sınıf nitelikleri olarak tanımlansın. P, kesişimleri boş küme olan niteliklerin kümesi olarak tanımlansın. Q, kesişimleri boş küme olmayan niteliklerin kümesidir. Metriğin hesaplanmasında kullanılan P ve Q kümeleri sırasıyla eşitlikler **(4.4)** ve **(4.5)**'te verilmiştir.

$$\{I_1\}, \cdots, \{I_n\}. P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\} \quad \textbf{(4.4)}$$

$$Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\} \quad \textbf{(4.5)}$$

Eğer n adet kümenin her biri $\{I_1\}$, …, $\{I_n\}$ boş küme ise P = $\emptyset$ olarak tanımlanır. Bu durumda sınıfın metotları arasındaki uyum eksikliği eşitlikler **(4.6)** ve **(4.7)** kullanılarak hesaplanır.

$$LCOM = |P| - |Q|, eğer\ |P| > |Q| \quad \textbf{(4.6)}$$

$$LCOM = 0, eğer\ |P| \leq |Q| \quad \textbf{(4.7)}$$

LCOM, bir sınıfın metotları ve nitelikleri arasında bağlılığı ölçmek için kullanılır. Yüksek LCOM sınıfta uyum eksikliğine işaret eder. Sınıfın karmaşıklığı artırır ve sınıfı hataya açık hale getirir. Bu nedenle bir sınıfın LCOM değerinin düşük olması tercih edilir.

Metriğin değerlendirilmesi:

- Kapsüllemeyi (encapsulation) desteklediği için metotların uyumlu olması tercih edilir
- Metotları arasında uyum eksikliği bulunan sınıflar daha küçük sınıflara bölünmelidir
- Metotlar arasındaki uyum eksikliği sınıfın tasarımındaki hataya işaret eder

Metotları arasında uyum eksikliği bulunan sınıflar yazılım geliştirme süreci boyunca çıkacak hata sayısını ve karmaşıklığı arttırır.

4.2.2.2 MOOD metrikleri

Nesneye yönelik tasarımının değerlendirilmesi için F.B.Abreu ve diğerleri tarafından tanımlanmıştır [26]. Metriklerle nesneye yönelik tasarımının temel yapı taşları olan kapsülleme, kalıtım, çok biçimlilik, bilgi saklama ve bağımlılık özelliklerinin yazılım kalitesi üzerindeki etkisi değerlendirilmiştir. Bunun yanında metriklerle ilgili resmi olarak tanımlı olma, boyutsuz olma, elde edilebilir olma, ölçeklenebilir olma ve kolay hesaplanabilir olma kriterlerini tanımlamışlardır. Ayrıca metriklerin yazılım programlama dili ve boyutundan bağımsız olması gerektiğini ifade etmişlerdir.

MOOD metrikleri, küme teorisine dayanır ve basit matematik işlemler içerir. Metrikler, ön yazılım tasarımı tamamlandıktan sonra uygulanabilir ve böylece kusurlar en erken safhada bulunabilir. MOOD metrik kümesinde tanımlanan metrikler aşağıda detaylandırılmıştır.

Metot saklama faktörü (Method hiding factor), yazılım sınıflarında tanımlı olan görünmez (erişilemez) metotların toplamının, tüm metotların toplamına oranıdır. Metrik, nesneye yönelik paradigmanın kapsülleme ilkesi ile ilişkilidir. Bu metrik hesaplanırken kalıtım ile gelen metotlar hesaba katılmamaktadır. Eğer metrik değeri yüksekse (%100) metotların hiç birisinin erişilebilir olmadığı anlamına gelir. Bu tip bir yazılım oldukça az bir işlevsellik sağlayacaktır. Eğer metrik değeri düşükse (%0) yazılımdaki hiçbir metodun korunmadığı anlamına gelir.

Nitelik saklama faktörü (Attribute hiding factor), yazılım sınıflarında tanımlı olan görünmez (erişilemez) niteliklerin toplamınının, tüm niteliklerin toplamına oranıdır. Metrik, nesneye yönelik paradigmanın kapsülleme ilkesi ile ilişkilidir. Eğer metrik

değeri yüksekse (%100) niteliklerin tümü erişilemezdir. Eğer metrik değeri düşükse (%0) niteliklerin hiç birisinin korunmadığı anlamına gelir.

Metot kalıtım faktörü (Method inheritance factor), yazılım sınıflarında kalıtım ile gelen metotların toplamının, tüm metotların toplamına oranıdır. Metrik, nesneye yönelik paradigmanın kalıtım ilkesi ile ilişkilidir. Eğer metrik değeri düşükse (%0) yazılımda yeniden kullanılabilirlik yoksunluğuna işaret eder.

Nitelik kalıtım faktörü (Attribute inheritance factor), yazılım sınıflarında kalıtım ile gelen niteliklerin toplamının, tüm niteliklerin toplamına oranıdır. Metrik, nesneye yönelik paradigmanın kalıtım ilkesi ile ilişkilidir. Eğer metrik değeri düşükse (%0) yazılımda yeniden kullanılabilirlik yoksunluğuna işaret eder.

Çok biçimlilik faktörü (Polymorphism factor), yazılım sınıflarında davranışı değiştirilen çok biçimli metotların toplamının, davranışı değiştirilebilecek tüm çok biçimli metotların toplamına oranıdır.

Metrik, nesneye yönelik paradigmanın çok biçimlilik ilkesi ile ilişkilidir. Eğer metrik değeri yüksekse (%100) yazılımda kalıtım ile alınan tüm çok biçimli metotların davranışlarının değiştirildiği anlamına gelir. Eğer metrik değeri düşükse (%0) yazılımda çok biçimlilik kullanılmadığı anlamına gelir.

Bağımlılık faktörü (Coupling factor), yazılım sınıfları arasındaki bağımlılık sayısı toplamının, oluşabilecek tüm bağımlılık sayısı toplamına oranıdır. Metrik, nesneye yönelik paradigmanın bağımlılık ilkesi ile ilişkilidir. Eğer metrik değeri düşükse (%0) sınıfların birbirlerine bağımlı değildirler.

4.2.2.3 QMOOD metrikleri

Yeniden kullanılabilirlik, işlevsellik, etkinlik, anlaşılabilirlik, genişletilebilirlik ve esneklik gibi yazılım kalite karakteristiklerinin değerlendirilebilmesi için Bansiya ve Davis tarafından tanımlanan metrik kümesidir [27].

Metrik kümesinde ele alınan yazılım tasarım nitelikleri kalıtım, kapsülleme, çok biçimlilik, soyutlama, bağımlılık, uyum, mesajlaşma, hiyerarşiler, bileşim (composition), tasarım boyutu ve karmaşıklıktır. QMOOD metrik kümesinde tanımlanan metrikler aşağıda detaylandırılmıştır.

Sınıf sayısı (Design size in classes), tasarımdaki sınıf sayısı toplamına eşittir.

Hiyerarşi sayısı (Number of hierarchies), tasarımdaki sınıf hiyeraşilerinin toplamına eşittir.

Ortalama ata sayısı (Average number of ancestors), tüm sınıfların kalıtım ağacı derinliklerinin ortalamasıdır. Metrik değeri yazılımda kalıtım kullanımı gösterir.

Veri erişim metriği (Data access metric), sınıfın erişilemez niteliklerinin tüm niteliklere oranıdır. Metrik değeri yazılımda kapsülleme ilkesi ile ilişkilidir.

Doğrudan sınıf bağımlılığı (Direct class coupling), bir sınıfın doğrudan bağımlı bulunduğu sınıf sayısıdır.

Sınıf arayüz boyutu (Class interface size), bir sınıftaki erişilebilir metotların toplamıdır. Metrik yazılımın mesajlaşma özelliği hakkında fikir verir.

Kümeleme ölçüsü (Measure of aggregation), nitelikler kullanılarak gerçekleştirilen bütün – parça ilişki sayısını ölçer.

Sınıftaki metotlar arası uyum (Cohesion among methods of class), sınıfta bulunan metotların arasındaki ilgi ilişkisini metot parametrelerin aynı olup olmaması ile hesaplar.

İşlevsel soyutlama ölçüsü (Measure of functional abstraction), bir sınıf tarafından kalıtım ile alınan metot sayısı toplamının, sınıfın üye metotları tarafından erişilebilecek metot sayısı toplamına oranıdır.

Çok biçimli metot sayısı (Number of polymorphic methods), yazılımdaki çok biçimli davranış gösterebilecek metotların sayısının toplamıdır.

Metot sayısı (Number of methods), sınıfta tanımlı olan tüm metotların sayısının toplamıdır.

4.2.3 Kod kapsama metrikleri (Code coverage metrics)

Kod kapsama, yazılım kaynak kodunun belirli bir test paketinde bulunan testler koşturulduğunda hangi ölçüde test edildiğinin ölçülmesidir. Yüksek kod kapsama değerlerine sahip olan yazılım ürünleri etraflıca test edilmişlerdir ve hata içerme olasılıkları azdır. Kod kapsama ölçümü için çeşitli metrikler bulunmaktadır. Bu metriklerden bazıları aşağıda detaylandırılmıştır [28].

Metot kapsama (Function coverage), yazılım ürününde tanımlanan her bir metodun veya alt yordamın çağrılıp çağrılmadığının ölçüsüdür.

İfade kapsama (Statement coverage), yazılım ürününde bulunan her bir ifadenin çalıştırılıp çalıştırılmadığının ölçüsüdür.

Dal kapsama (Branch coverage), yazılım ürününde bulunan her bir kontrol yapısına (örn. if ve case ifadeleri) ait dalların çalıştırılıp çalıştırılmadığının ölçüdür. Örnek olarak bir if yapısı ele alındığında yapının doğru ve yanlış olmak üzere iki adet çalıştırılması gereken dalı olduğu anlaşılacaktır.

Koşul kapsama (Condition coverage), yazılım ürününde bulunan her bir mantıksal ifadenin doğru ve yanlış olmak üzere iki adet durumunun oluşturulup oluşturulmadığının ölçüsüdür.

4.3 Yazılım Test Edilebilirliği

Yazılım testedilebilirliği, yazılım ürününün (yazılım sistemi, yazılım bileşeni, gereksinim veya tasarım dokümanları) belirli bir test bağlamında test faaliyetlerini ne kadar desteklediğinin derecesidir. Eğer yazılım ürününün test edilebilirliği yüksek ise test faaliyetleri sayesinde sistemdeki hataları bulmak daha kolay olacaktır.

Yazılım testedilebilirliğinin az olması test faaliyetleri için gerekli eforun artmasına neden olur.

4.3.1 Yazılım bileşen test edilebilirliği

Yazılım bileşenlerinin(bileşen, sınıf) testedilebilirliği aşağıdaki etkenler tarafından belirlenir.

Kontrol edilebilirlik (Controllability): Yazılım bileşeninin testleri için gerekli olan durumların kontrol edilebilme derecesidir.

Gözlenebilirlik (Observability): Test sonuçlarının gözlenebilir olma derecesidir.

İzole edilebilirlik (Isolateability): Test edilen yazılım bileşeninin diğer yazılım bileşenlerinden izole olarak test edilebilme derecesidir.

Kavramların ayrımı (Seperation of concerns): Test edilen yazılım bileşeninin iyi tanımlanmış, tek bir sorumluluğunun olmasıdır.

Anlaşılabilir olma (Understandability): Test edilen yazılım bileşeninin iyi belgelendirilmiş veya açık olmasıdır.

Otomatikleştirilebilir olma (Automability): Test edilen yazılım bileşenindeki testlerin otomatik hale getirilebilme derecesidir.

Ayrıştırılabilir olma (Heterogeneity): Ayrı teknojiler kullanılması durumda, paralelde ayrı test teknolojilerinin ve araçların gerekme derecesidir.

4.3.2 Yazılım test edilebilirliğinin ölçülmesi

Yazılım test edilebilirliğinin tanımı değerlendirildiğinde test edilebilirlik için niteliksel bir ölçüm yöntemine ihtiyaç bulunduğu çıkarılabilir. Herhangi bir yazılım ürünü için niteliksel ölçüm yazılım niteliklerinin yazılım metrikleri vasıtasıyla değerlendirilmesi yoluyla yapılır. Yazılım test edilebilirliği de benzer şekilde yazılım metrikleri vasıtasıyla değerlendirilebilir. Muro yazılım test edilebilirliği ile ilişkili metrikleri yapısal ve davranışsal ve veri akış metrikleri olarak gruplamıştır [15]. Yapısal ve davranışsal metrikler yazılım bileşenleri ve bu bileşenlerin ilişkileri ve etkileşimleri ile ilgiliyken, veri akış metrikleri ise yazılım içindeki veri yolları ve akışı ile ilgilidir.

Yazılım tasarımı esnasında yazılım ürünü yapısal olarak yazılım bileşenleri ve bu bileşenlerin ilişkilerileriyle davranışsal olarak ise yazılım bileşenleri arasındaki etkileşimlerle tanımlanır. Bu yapısal ve davranışsal modeller yazılım geliştirme faaliyetlerine temel oluşturur. Bu sebeple bu modellere ait nitelikler test edilebilirliğin değerlendirilmesi için metrikler vasıtasıyla ölçülebilir.

Baudry ve diğerleri UML sınıf ve durum diyagramlarının test edilebilirliğin ölçülmesi amacıyla analiz edilmesini önermişlerdir [29]. Test faaliyetlerini zorlaştıran etkenlerden birisinin nesneler arasındaki etkileşimlerin beklenmedik yan etkiler oluşturması olduğunu ifade etmişlerdir. Bu yan etkileri, nesnenin başka bir nesnenin durumunu birden fazla yolla değiştirmesi olarak tanımlamışlardır. Başka bir çalışmada nesneler arasındaki etkileşimlerin oluşturduğu beklenmeyen yan etkiler test edilebilirliği olumsuz etkileyen karşı örüntüler olarak tanımlanmıştır [30].

Bu örüntüler test edilebilirlik iyileştirme faaliyetlerinin yapılabileceği kısımlar olarak tavsiye edilmiştir. Sınıf diyagramlarında görülen nesneler arasındaki etkileşimlerin karmaşıklığının ölçülmesi test edilebilirliğin tahmin edilmesi açısından yararlı

olabilir. Bu etkileşimler sınıf bağımlılık grafiği (class dependeny graph) oluşturma tekniği vasıtasıyla belirlenebilir [29].

Yazılım kaynak kodunun analiz edilmesi de test edilebilirliğin ölçülmesi için bir fikir verebilir. Jungmary yazılım bileşenlerinin bağımlılık karakteristiklerini kullanarak, metrikler yardımıyla yazılım bileşeninde bulunan yerel bağımlılıkların sistemin genel test edilebilirliği üzerindeki etkisini araştırmıştır. Çalışmada test edilebilirlik ile ilgili olarak ortalama bileşen bağımlılık (average component dependency) metriği tanımlanmıştır [31]. Bu metrik belirli bir yazılım bileşeninin, diğer yazılım bileşenlerine doğrudan ve dolaylı olarak tanımlanan bağımlılıklarının ortalamasını ölçmektedir. Bunun gibi metrikler belirli bir yazılım bileşeni değiştirildiğinde bundan etkilenecek olan diğer yazılım bileşenlerinin sayısı hakkında bir fikir verebilir.

Nesneye yönelik sistemlerde ise kaynak kod analizi sistemi oluşturan sınıfların test edilebilirliğinin analiz edilmesi amacıyla kullanılabilir. Bu analiz için çeşitli nesneye yönelik metrikler kullanılabilir. Yapılan bir araştırmada kalıtım ağacı derinliği, alt sınıf sayısı, sınıf başına düşen kod satır sayısı gibi nesneye yönelik metrikler kullanılarak sistemi oluşturan sınıflara ait niteliklerin, sistemi doğrulamak için kullanılan test durumları ile ilişkisi incelenmiştir. Araştırmada kullanılan test durumu sayısı metriği sistemdeki tüm testlerin sayısının ölçülmesi amacıyla kullanılmıştır. Bu metrik sistemin test edilmesi için gerekli olan test gayretini sayısal olarak ifade etmektedir. Araştırma sonucunda sistemi oluşturan sınıflara ait niteliklerin sistem test durumları üzerindeki etkisi bulunduğu sonucuna varılmıştır [7].

Nesneye yönelik sistemler üzerinde Analitik Hiyerarşi Süreci (Analytic Hierarchy Process) yönteminin kullandığı başka bir araştırmada yazılım test edilebilirliğinin değerlendirilmesi için CK, MOOD ve QMOOD metrik kümelerinden seçilen metrikler kullanılmıştır. Araştırma sonucunda QMOOD metrik kümesinde bulunan sınıf hiyerarşi sayısı ile çok biçimli metot sayısı metriklerinin test edilebilirlik için önemli olduğunu belirtilmiştir [8].

4.3.3 Yazılım test edilebilirliği ile yazılım metrikleri ilişkisi

Yazılım sistemlerinde kalitenin ölçümü yazılım metrikleri kullanılarak yapılmaktadır. Yazılım test edilebilirliği de yazılım sürdürülebilirliği altında tanımlanan kalite karakteristiklerden birisidir.

Yazılım test edilebilirliği basitçe yazılım ürününün test faaliyetlerinin kolayca yapılabilmesi olarak tanımlanmaktadır. Yazılım ürününde test edilmek istenen bileşene göre uygun metrikler seçilerek test edilebilirlik değerlendirilebilir. Test edilmek istenen bileşen eğer bir metot ise metodun çevrimsel karmaşıklığı metodun test durumlarının az/çok olması hakkında bir fikir verecektir. Benzer şekilde test edilemek istenen bileşen nesneye yönelik bir yazılım ürünündeki sınıf ise nesneye yönelik metriklerden sınıf ağırlıklı metot sayısı, sınıfın tetiklediği metot sayısı, sınıf metotları arasındaki uyum eksikliği gibi sınıf karmaşıklığı ile doğrudan ilgili olan metrikler değerlendirilebilir. Çünkü sınıfın karmaşıklığının yüksek olması sınıf test edilebilirliği üzerinde olumsuz etki yapmaktadır.

Nesneye yönelik yazılım ürünlerinde nesneler arasındaki etkileşim nesnelerin birbirlerine olan statik bağımlılıkları ile sağlanmaktadır. Bu bağımlıklar sınıfın test edilebilirliği üzerinde doğrudan etkilidir. Birim testler açısından değerlendirildiğinde bir nesneyi test edebilmek için nesnenin bağımlı olduğu nesneleri de sağlamak gerecektir. Bu nedenle nesnelerin kolay test edilebilmeleri için olabildiğinde az sayıda bağımlılığa sahip olmaları beklenir. Nesnelerin bağımlılıklarını değerlendirmek için nesneye ait bağımlılık sayısı metriği kullanılabilir. Nesneye yönelik yazılım ürünlerinde sıklıkla kullanılan kalıtım işlemi nesneler arasındaki bir bağımlılık olarak kabul edilmektedir. Kalıtımın test edilebilirlik açısından değerlendirilmesi için kalıtım ağacı derinliği ve alt çocuk sayısı metriği kullanılabilir.

4.3.4 Yazılım test edilebilirliğinin iyileştirilmesi

Yazılım bileşenlerinin test edilebilirliği bir takım yazılım tasarım ve geliştirme yöntemleri kullanılarak iyileştirilebilir. Bu yöntemlerden bazıları aşağıda detaylandırılmıştır.

4.3.4.1 Test yönelimli yazılım geliştirme (Test-driven development)

Test yönelimli yazılım geliştirme birbirini takip eden iterasyonlardan oluşan bir yazılım geliştirme yöntemidir. Her bir iterasyonda yazılım geliştirici yeni müşteri isteri veya mevcut bir yazılım yeteneğinin iyileştirilmesi için başlangıçta doğrulanmayan bir birim test yazar. Ardından yazdığı birim testin doğrulanması için gerekli olan en az kodlamayı yapar ve en son olarak yazdığı kodu proje kodlama standartlarına göre düzenler.

Test yönelimli yazılım geliştirme yöntemini Kent Beck icat etmiştir [34]. 2003 yılında yaptığı bir konuşmada yöntemin basit ve etkili yazılım tasarımlarının ortaya çıkmasına yardımcı olduğunu ve yazılım geliştiricin yaptığı işe olan güvenin artmasına neden olduğunu belirtmiştir. Yöntem yeni yeni yazılım geliştirme camiasında popülerlik kazanmasına karşın yöntemin temelleri 1999 yılında ortaya çıkan Uç Programlama (Extreme Programming) disiplininin test öncelikli yazılım geliştirme alt disiplinine aittir.

Yöntem kullanılırken sıksıkla karşılaşılan stub ve mock nesne kavramları bulunmaktadır. Yöntemin terminolojisinde:

Stub nesne, iki nesne arasındaki bağımlılığın koparılması yardımcı olan nesne olarak tanımlanır. Stub nesneler birim testlerin hızlı koşmasına yardımcı olurlar. Stub nesneler birim testin doğrulanması için kullanılmazlar [32].

Mock nesne, stub nesne ile benzer olarak iki nesnenin arasındaki bağımlılığın koparılması için kullanılır. Stub nesneler ile benzer olarak birim testlerin hızlı koşmasına yardımcı olurlar. Mock nesneler stub nesnelerden farklı olarak birim testlerin doğrulanması için kullanılırlar [32].

4.3.4.2 Yazılım test edilebilirliği için yazılım tasarımı

Nesneye yönelik yazılım sistemleri sınıflardan ve arayüzlerden (interface) oluşmaktadır. Bir sınıf bir veya birden fazla arayüzün metotlarını miras olarak alabilir, başka bir sınıfa bağımlılık içerebilir. Kalıtım, çok biçimlilik, kapsülleme ve bilgi saklama nesneye yönelik tasarım ve programlamayı daha etkili yapan özelliklerdir. Fakat karmaşık kalıtım hiyerarşileri, çok fazla bağımlılık ve gereksiz soyutlama yazılım test edilebilirliğini düşürebilir. Yazılım tasarımı yapılırken yüksek test edilebilirlik ve sürdürülebilirlik için en doğru yöntemlerden yararlanmak gerekir. SOLID ilkeleri Robert C. Martin tarafından adlandırılan nesneye dayalı yazılım tasarımı ve programlama için kullanılan 5 adet ilkeden ibarettir [33]. İlkeler birlikte doğru olarak kullanıldığında kolay test edilebilen, bakımı kolay yapılabilen ve kolay genişletilebilen yazılım sistemleri oluşturmak için yardımcı olurlar. Her bir ilke ve bu ilkenin yazılım testleri üzerindeki etkisi aşağıda detaylandırılmıştır.

Tek sorumluluk ilkesi (Single responsibility principle), bir sınıfın sadece bir sorumluluk üstlenmesini tavsiye etmektedir. Bu ilkeye uygun geliştirilen sınıflar tek bir sorumluluk üstlenecek şekilde tasarlanmalıdırlar.

Tek sorumluluk üstlenen sınıf sadece aynı görevle ilgili işlemleri yapmalıdır. Bu ilke bir sınıfa ait bir işlev sadece bir tek bir iş yapmakla sorumludur şeklinde detaylandırılabilir. Bu ilkenin önerdiği tavsiyeye uyarak geliştirilen sınıflar sistemin test edilebilirliğini test edilmesi gerekli işlev sayısını azaltarak artırırlar. Bununla birlikte sınıfların içerdiği işlev sayıları sorumlu oldukları görevin ihtiyaç duyduğu kadar olacağından yazılım tasarımında diğer sınıflardan izole olarak test edilebilen küçük sınıflar oluşur. Bu ilke aynı zamanda kendi içinde uyumlu ve diğer sınıflara daha az bağımlı sınıf tasarımlarına yardımcı olur.

Açık/kapalı ilkesi (Open/closed principle), bir sınıfın genişletilmeye açık fakat mevcut sorumluluklarının değiştirilmesine kapalı olmasını tavsiye etmektedir. Bu ilkeye uyarak tasarlanan sınıflar sorumlulukları değiştirilmeyecek bir anlayışla atanır. Sınıfınn sorumluluğunun kapsamını değişmesi için bir neden bulunmamalıdır. Eğer gerekli ise sınıf kalıtım kullanılarak yeni bir sınıf olarak kullanılabilir. Yeni oluşturulan sınıfa ait nesne gerekli yerlerde mevcut sınıfa ait nesnenin davranışı değiştirilmeden kullanılabilir. Bu ilke sınıflara ait nesnelerin sahte nesnelere uyumunu artırır. Sınıf nesnelerine ait mevcut birim testlerin yeniden yazılmasına gerek kalmaz çünkü mevcut testler türetilen yeni sınıf nesnesi için aynı şekilde çalışmalıdır.

Liskov yerine geçme ilkesi (Liskov substitution principle), türetilmiş sınıfa ait nesnenin türediği sınıf nesnesinin yerine geçtiğinde aynı davranışı göstermesini tavsiye eder. Eğer aynı davranışı göstermiyorsa bu ilkeye aykırıdır. Bu ilkeye uygun sınıf tasarımı yapıldığında sınıfa ait nesnenin bağımlı bulunduğu bir diğer sınıf nesnesi sahte nesne, stub nesne veya mock nesneler ile kolayca değiştirilebilir. Bu yerine koyma ilkesi sınıfa ait nesnenin birim testlerinin diğer sınıflara ait nesnelere olan bağımlılığını ortadan kaldırır. Böylece birim testler daha hızlı koşabilir ve sadece test etmekle sorumlu olduğu sınıfın davranışa bağlı olur.

Arayüz ayırma ilkesi (Interface segregation principle) bir sınıf nesnesinin türetildiği sınıf nesnesinin kullanmadığı işlevlerine bağımlı olmamasını tavsiye eder. Bu ilke doğru uygulandığında bir tane büyük sınıf arayüzü yerine birden fazla küçük ve birbiri içinde uyumlu sınıf arayüzleri üretilmesine yardımcı olur. Küçük sınıf arayüzleri yazılım güncellemelerine daha dayanıklıdır ve sistemin test edilebilirliğini büyük sınıf arayüzünün getirdiği karmaşıklığı azaltarak arttırır. Bu ilke aynı zamanda sahte, stub ve mock nesnelerin testler için kullanılmasını kolaylaştırır.

Bağlımlık evirme ilkesi (Dependency inversion principle) bir sınıf nesnesinin somut bağımlıklar yerine soyut bağımlılıklara sahip olmasını tavsiye eder. Bu ilke ayrıca soyut nesnelerin somut nesnelere bağlı olmasının yerine somut nesnelerin soyut nesnelere bağlı olmasını tavsiye eder. Doğru uygulandığında sınıf nesnelerinin diğer sınıf nesnelerine gevşek bağlı (loosely coupled) olmasını sağlar. Gevşek bağlı olma bir gerçekleştirmenin (implementation) kullanımın şeklinin değiştirilmeden bir başka gerçekleştirme ile değiştirilmesini sağlar. Yazılım test edilebilirliği açısından bu ilkenin etkisi büyüktür. Birim testler için hayati olan bu ilke, "Stub" ve "Mock" nesnelerin yazılım sınıflarının yerine içine kolayca yerleştirilmesini sağlar.

5. VAKA ÇALIŞMASI: DEMİRYOLU ANKLAŞMAN YÖNETİM YAZILIMI

Tezin amacına uygun olarak Tübitak ve İstanbul Teknik Üniversitesi (İTÜ) işbirliği ile Türkiye Cumhuriyeti Devlet Demiryolları (TCDD) için geliştirilen Ulusal Demiryolu Sinyalizasyonu Projesi (UDSP) kapsamında geliştirilen Demiryolu Anklaşman Yönetim Yazılımı analiz edilmiştir [36].

Yazılım, ISO 25010 kalite standardında belirlenmiş olan Ürün Kalitesi (Product Quality) kalite modeli ana kalite karakteristiklerinden sürdürülebilirlik altında tanımlı olan yazılım test edilebilirliği alt kalite karakteristiğine açısından analiz edilmiş ve sonuçlar değerlendirilmiştir. Analiz sonuçlarına göre "Demiryolu Anklaşman Yönetim Yazılımı v2" sürümü yazılım test edilebilirliğini artıracak yazılım tasarım ve gerçekleme ilkeleri dikkate alınarak geliştirilmiştir.

Geliştirilen yeni yazılım sürümü, yazılım testedilebilirliği açısından analiz edilmiştir. Son analiz sonuçları, ilk analiz sonuçları ile karşılaştırılmış ve iyileştirme sonuçları detaylandırılmıştır.

UDSP kapsamında Kontrol Merkezi (KM) ve Demiryolu Anklaşman Sistemi olmak üzere iki adet alt sistem geliştirilmiştir. İncelenen demiryolu anklaşman yönetim yazılımı, KM yazılım bileşenlerinden birisidir ve çalıştığı sistemin bütünü değerlendirildiğinde emniyet kritik yazılım kategorisine girmektedir.

5.1 Demiryolu Anklaşman Sistemi

Demiryolu anklaşman sistemi bir istasyon içerisindeki ve komşu istasyonlar arasındaki trafiği kontrol eden sistemdir. Bu sistem tren rotalarının tahsisini, makas tanzimlerini ve diğer bütün raylı sistem araçlarının yönetimini demiryolu kurallarına, düzenlemelerine ve işletmedeki demiryolu istasyonunun teknolojik süreç gereksinimlerine uyarak gerçekleştirir.

5.2 Kontrol Merkezi

Kontrol Merkezi (KM), demiryolu anklaşman sistemlerinin içinde bulunan sorumluluk alanındaki demiryolu bölgesinde gerçekleşen tüm demiryolu trafiğinin izlenmesi, kontrolü ve yönetilmesi ile görevli donanım ve yazılımları içeren bir alt sistemdir. Kontrol Merkezi yazılım bileşenleri aşağıda detaylandırılmıştır.

- KM
    - Demiryolu Anklaşman Yönetim
    - Saha Kumanda ve Görüntüleme
    - Protokol Bilgi Kayıt
    - Trengraf
    - Tren Takip
    - Veritabanı Kayıt ve Sorgulama

yazılım bileşenlerini içerir.

5.3 Demiryolu Anklaşman Yazılımı

Demiryolu anklaşman yazılımı, demiryolu anklaşman sisteminin kontrolünü sağlayan yazılımdır. Yazılım KM'den gelen güzergah tanzimi, tekrarlı güzergah tanzimi, güzergah iptali, makas konum değiştirme, makas bloke, makas bloke iptal, sinyal bloke, sinyal bloke iptal, sinyal kapama vb. talepleri değerlendir ve uygun bulduğu talepleri yerine getirir.

5.4 Demiryolu Anklaşman Yönetim Yazılımı v1

Demiryolu anklaşman yönetim yazılımı, demiryolu anklaşman yazılımı ile birlikte demiryolu sahalarının yönetiminden sorumlu olan bileşendir. Burada kastedilen yönetim taleplerin iletimi, talep cevaplarının ve demiryolu ekipmanları durum bilgilerinin okunmasıdır. Bu bileşenin ana görevi bir veya birden fazla anklaşman yazılımı ile iletişimi sağlayarak paralel olarak güvenli işlem yapabilmesidir.

Kontrol edilen anklaşman yazılımlarından elde edilen ekipman durum bilgilerini okuyarak bu bilgileri saha kumanda ve görüntüleme yazılımına göndermek ve saha kumanda ve görüntüleme yazılımı tarafından gönderilen talepleri işleyerek ilgili demiryolu anklaşman yazılımına iletmek ve talebin kabul veya ret durumlarını takip

ederek yine saha kumanda ve görüntüleme yazılımına bildirmek bu bileşenin görevidir. Bileşen, demiryolu anklaşman yazılımı ile Modbus endüstriyel iletişim protokolü üzerinden haberleşmektedir. Demiryolu anklaşman yönetim yazılımı v1 C++ programlama dili ve nesneye yönelik yazılım geliştirme yaklaşımı kullanılarak 2 kişi tarafından 3 ay sürede geliştirilmiştir. Proje genelinde şelale (waterfall) yazılım geliştirme metodolojisi kullanılmıştır.

Bileşenin başlıca görevleri aşağıda detayladırılmıştır:

- Demiryolu anklaşman yazılımı ile birlikte demiryolu trafiğini yönetmek
- İlgili demiryolu sahasına ait ekipman bilgilerini saha kumanda ve görüntüleme yazılımına göndermek
- Saha kumanda ve görüntüleme yazılımından gelen talepleri değerlendirmek ve anklaşman yazılımına iletmek
- Saha kumanda ve görüntüleme yazılımı ve demiryolu anklaşman yazılımı arasındaki iletişimi sağlamak

Anklaşman yönetim yazılımı, saha kumanda ve görüntüleme yazılımından güzergah, makas, sinyal, ray devresi, hemzemin geçit ve demiryolu sahası ile ilgili çeşitli talepler almaktadır ve bu talepleri anklaşman yazılımına göndermektedir. Anklaşman yönetim yazılımının anklaşman yazılımına gönderdiği talepler aşağıda detaylandırılmıştır.

- Güzergah ile ilgili talepler
  - Güzergah tahsis talebi
  - Güzergah tahsis iptal talebi
  - Güzergah tahsis zorunlu iptal talebi
  - Manevra güzergahı tahsis talebi
- Makas ile ilgili talepler
  - Makas pozisyon değişiklik talebi
  - Makas hareket bloke talebi
  - Makas güzergah bloke talebi
  - Makas veri arızası normalize etme talebi
- Sinyal ile ilgili talepler
  - Sinyal bloke talebi
  - Sinyal varış bloke talebi
  - Sinyal veri arızası normalize etme talebi

- Ray devresi ile ilgili talepler
    - Ray devresi bloke talebi
    - Ray devresi veri arızası normalize etme talebi
    - Ray devresi beklenmedik meşguliyet arızası normalize etme talebi
- Hemzemin geçit ile ilgili talepler
    - Hemzemin geçit açık arızası normalize etme talebi
    - Hemzemin geçit kapalı arızası normalize etme talebi
    - Hemzemin geçit veri arızası normalize etme talebi
    - Hemzemin geçit bariyer kırık arızası normalize etme talebi
- Demiryolu sahası ile ilgili talepler
    - Sinyaller gece/gündüz voltajı ayarlama talebi
    - Makas ısıtıcıları açma/kapama talebi

5.4.1 Yazılım kalitesinin ölçülmesi ve sonuçların analizi

UDSP projesinde geliştirilen demiryolu anklaşman yönetim yazılımı, nesneye yönelik tasarım ve programlama kullanılarak geliştirilmiştir. Bu nedenle mevcut bileşenin yazılım test edilebilirliğinin ölçülebilmesi için nesneye yönelik yazılım metriklerini ölçebilen bir araç kullanılmıştır.

Scientific Toolworks, Inc. tarafından geliştirilen Understand yazılımı, yazılım projelerinde üretilen büyük ölçekli ve kritik yazılım kodlarının bakım yapılabilmesi, ölçülebilmesi ve analiz edilebilmesi amacıyla kullanılan statik analiz aracıdır [35]. Aracın özellikleri aşağıda detaylandırılmıştır.

- Çeşitli yazılım metriklerinin hesaplanması ve raporlanması
- Yazılım bağımlılık analizinin yapılması
- Birden çok yazılım geliştirme diline destek verilmesi
- Yazılım kodlarının çeşitli kodlama standartlarına göre karşılaştırılması
- Yazılım akış diyagramlarının yazılım kodlarından üretilmesi
- Çeşitli kod arama seçenekleri sunması
- Yazılım kodlarının düzenlenmesi için kullanıcı dostu bir arayüz sunması

Understand aracı çeşitli yazılım metriklerini ölçebilmektedir. Bu metrikler arasında:

- Proje metrikleri
- Sınıf metrikleri (Nesneye yönelik yazılım için)
- Nesneye yönelik metrikler (Nesneye yönelik yazılım için)

- Program birim metrikleri
- Dosya metrikleri
- Dosya ortalama değer metrikleri

bulunmaktadır.

Demiryolu anklaşman yönetim yazılımı ait çeşitli yazılım metrikleri Understand aracı ile ölçülmüştür. Yazılım metriklerinin ile beraber yazılım içinde bulunan sınıf nesnelerinin bağımlılık bilgileri aynı araçla çıkarılmıştır. Yazılım metriklerinin ölçüm sonuçları aşağıda detaylandırılmıştır.

Proje metriklerine göre demiryolu anklaşman yönetim yazılımında:

- 23 adet sınıf bulunmaktadır
- Toplam satır sayısı 8817'dir
- Toplam boş satır sayısı 1612'dir
- Toplam kod satır sayısı 6031'dir

Sınıf metriklerine göre demiryolu anklaşman yönetim yazılımında:

- Çevrimsel karmaşıklığı 10'dan yüksek sınıf sayısı 13 adettir

Çevrimsel karmaşıklık için metriğin tanımladığı ilk çalışmada 10 sınır değer olarak tavsiye edilmiştir [23]. Bu çalışma kapsamında da 10 sınır değer olarak kabul edilmiştir.

Nesneye yönelik yazılım metriklerine göre demiryolu anklaşman yönetim yazılımı v1 sürümüne ait ölçüm sonuçları Çizelge 5.1'de gösterilmiştir.

LCOM değeri sınıfa ait ortalama uyumluluk değeri 100 üzerinde çıkarılarak yüzde olarak hesaplanmıştır.

Ölçüm sonuçları değerlendirildiğinde analiz edilen demiryolu anklaşman yönetim yazılımını oluşturan sınıfların %50'den fazlasının çevrimsel karmaşık değerinin 10 ve üzerinde olduğu görülmüştür.

Yüksek çevrimsel karmaşıklık değeri sınıfın karmaşık ve sınıfta test edilmesi gereken bağımsız dal sayısının fazla olduğuna işaret eder. Test edilmesi gereken bağımsız dal sayısının fazla olması test yükünü artırır ve hata çıkma olasılığını yükseltir.

Çizelge 5.1 : Demiryolu anklaşman yönetim yazılımı v1 metrik ölçüm sonuçları.

Sınıf Adı	LCOM	CBO	RFC	WMC
InterlockingManager	92	32	39	37
ModbusStream	14	2	7	7
Mthread	87	4	13	11
Log	78	4	14	12
QextModbus	79	12	19	19
ReqBenderReset	81	7	22	9
ReqBlockNormalize	73	7	22	9
ReqBlockProtection	72	7	22	9
ReqLevelCrossingNormalize	73	7	22	9
ReqRemoveTanzim	76	7	24	11
ReqSignalClose	75	8	22	9
ReqSignalEntranceBan	72	7	22	9
ReqSignalExitBan	72	7	22	9
ReqSignalNormalize	76	7	22	9
ReqSignalsDayNight	82	7	23	10
ReqSwitchDataInConsistenceNormalize	76	7	22	9
ReqSwitchHeater	82	8	23	10
ReqSwitchMoveBan	72	7	22	9
ReqSwitchPosition	72	7	23	10
ReqSwitchRouteBan	72	7	22	9
ReqTanzim	71	7	26	13
Timer	66	0	8	8
XmlReader	66	3	4	4

Nesneye yönelik metrikler değerlendirildiğinde bileşeni oluşturan sınıfların, sınıf içi metot uyumsuzluğu değerlendirinin oldukça yüksek olduğu görülmektedir. Yüksek LCOM değeri sınıfın birden fazla sorumluluğa sahip olduğu anlamına gelmektedir. Fazla sayıda sorumluluk içeren sınıflarda test yükü fazladır ve hata çıkma olasılığı yüksektir. Bunun yanında sınıfa ait nesnenin doğrudan bağımlı bulunduğu nesne sayılarılarıda bir çok sınıf için 7 değerinin üstündedir. CBO değeri yüksek olan sınıflar için birim test yazmak daha zordur. Bunun nedeni birim testlerde sınıfa ait nesne oluşturulmak istendiğinde sınıf nesnesinin bağımlı bulunduğu nesnelerinde oluşturulmasının gerekli olmasıdır. Bağımlı bulunulan nesneler oluşturulurken çıkabilecek alt bağımlılıklar birim testleri birden fazla nesneye bağımlı yapar ve bu bağımlılıklar eğer ağ ve veritabanı gibi kaynakları kullanmakta ise birim testlerin hızlı koşturulmasını engeller.

Sınıfların tetiklediği metot sayılarına bakıldığında bir çok sınıf için değerin 20 üzerinde olduğu tespit edilmiştir. RFC değerinin yüksek olması sınıfı daha karmaşık hale getirir. Sınıfın artan karmaşıklığı sınıfın test etme ve hata ayıklama

faaliyetlerinin daha uzun sürmesine neden olur ve sınıfın detaylarının derinlemesine anlaşılmasını gerektirir.

Sınıfların, sınıf başına metot ağırlığı değerlerine bakıldığında bir çok sınıf için değerin 9 ve üzerinde olduğu tespit edilmiştir. Bir çok sınıfın WMC değeri yüksektir. Yüksek WMC değeri sınıfın bakım yapılabilme faaliyetlerinin ne kadar çok kaynak (zaman, efor vb.) gerektireceğini gösterir. Sürdürülebilirliği düşük olan sınıflar yeniden kullanılabilir değildirler, test edilmeleri zordur ve uygulamaya özeldirler.

Yazılım test edilebilirliğini ölçmek amacıyla kullanılan yazılım metriklerinin ölçüm sonuçları değerlendirildiğinde mevcut yazılımın testedilebilirliğinin düşük olduğu tespit edilmiştir.

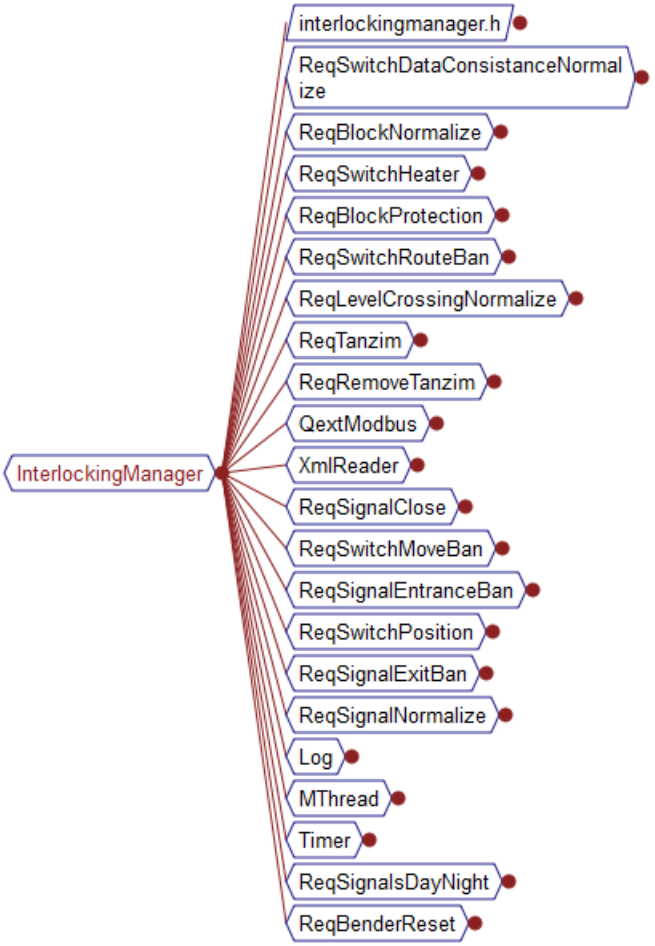
5.4.2 Yazılım tasarım analizi

Yazılım testedilebilirliği düşük olarak görülen yazılım tasarımı kod gözden geçirme ve tersine mühendislik yaklaşımıyla analiz edilmiştir. Tersine mühendislik yaklaşımı için Understand aracı kullanılmıştır.

Yazılım test edilebilirliğini etkileyen metriklerin önemli bir bölümünde olumsuz sonuçlar elde edilen InterlockingManager sınıfına ait sınıf statik bağımlılık grafiği Şekil 5.1'de gösterilmiştir.

Grafikten anlaşılacağı üzere InterlockingManager, saha kumanda ve görüntüleme yazılımından gelen taleplerin her biri için bir sınıf nesnesine bağımlıdır. Anklaşman yazılımı ile veri alışverişi yapması için gerekli olan adres değerlerini XML formatındaki bir dosyadan okumaktadır. Böylece dosya sistemine bağımlıdır. Anklaşman yazılımı ile haberleşmeyi Modbus endüstriyel iletişim protokolü ile yapabilmesi için QextModbus sınıfının nesnesine bağımlıdır. MThread, Log ve Timer sınıf nesnelerine de bağımlı durumdadır.

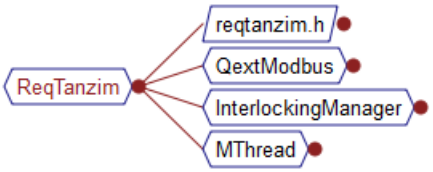
Bu sınıflardan MThread her bir talep sınıfının türediği sınıf olup; talep sınıflarına bir iplik (thread) olma özelliği kazandırmaktadır. Log sınıfı, InterlockingManager sınıfının metotları çağrıldığında oluşan ve dosya sisteminde tutulmasının yararlı olacağını düşünülen mesajları dosyaya yazmak için kullanılır. Kod gözden geçirme faaliyetleri esnasında Timer sınıfının kullanılmayan bir bağımlılık olarak yazılım kodu içinde kaldığı tespit edilmiştir.



Şekil 5.1 : InterlockingManager sınıf statik bağımlılık grafiği.

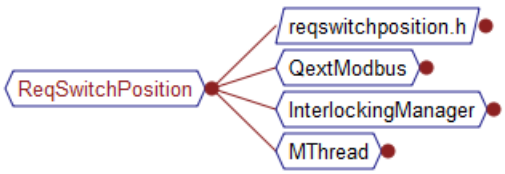
Yazılım testedilebilirliği bakımından iyi metrik ölçüm sonuçlarına sahip olmayan talep sınıflarına ait bağımlılık grafikleri çıkarılırken tüm talep sınıflarının 6 grupta incelenmiştir. Bu gruplar makaslar ile ilgili talepler grubu, güzergahlar ile ilgili talepler grubu, sinyaller ile ilgili talepler grubu, hemzemin geçitler ile ilgili talepler grubu, ray devreleri ile ilgili talepler grubu ve diğer talepler grubu olarak tespit edilmiştir. Her bir talep grubuna ait bağımlılık grafikleri grupta bulunan taleplerden birisi seçilerek gösterilmiştir.

Güzergah tahsis talebini temsil eden sınıfın statik bağımlılık grafiği Şekil 5.2'de gösterilmiştir.



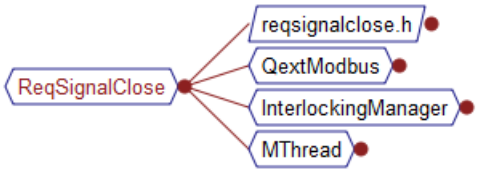
Şekil 5.2 : Güzergah tahsis talebi sınıf statik bağımlılık grafiği.

Makas pozisyon değiştirme talebini temsil eden sınıfın statik bağımlılık grafiği Şekil 5.3'te gösterilmiştir.



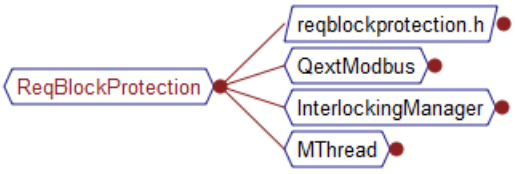
Şekil 5.3 : Makas pozisyon değiştirme talebi sınıf statik bağımlılık grafiği.

Sinyal kapama (kırmızı bildirim dışında bildirim(ler) veren sinyalin kırmızı bildirime çekilmesi) talebini temsil eden sınıfın statik bağımlılık grafiği Şekil 5.4'te gösterilmiştir.



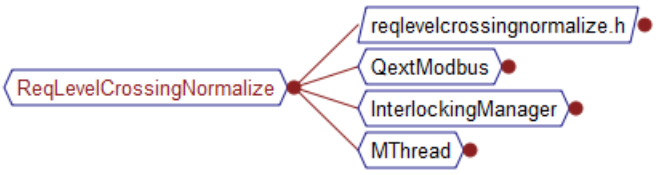
Şekil 5.4 : Sinyal kapama talebi sınıf statik bağımlılık grafiği.

Ray devresi koruma (korunan ray devresini kullanan herhangi bir güzergahın kurulumuna sistem izin vermeyecektir) talebini temsil eden sınıfın statik bağımlılık grafiği Şekil 5.5'te gösterilmiştir.



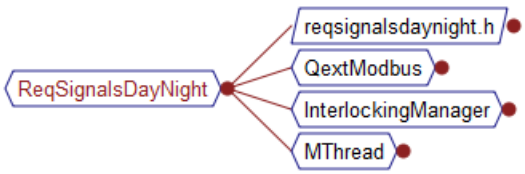
Şekil 5.5 : Ray devresi koruma talebi sınıf statik bağımlılık grafiği.

Hemzemin geçit arıza normalize (fiziki arıza kalkması durumunda, arıza alarm durumunun giderilmesi) talebini temsil eden sınıfın statik bağımlılık grafiği Şekil 5.6'da gösterilmiştir.



Şekil 5.6 : Hemzemin geçit arıza normalize talebi sınıf statik bağımlılık grafiği.

Diğer talepler grubundan seçilen sinyaller gece/gündüz voltajı ayarlama talebini temsil eden sınıfın statik bağımlılık grafiği Şekil 5.7'de gösterilmiştir.



Şekil 5.7 : Diğer talepler grubu sınıf statik bağımlılık grafiği.

Grafiklerden anlaşılacağı üzere her bir talep sınıfı gömülü anklaşman yazılımı ile haberleşebilmek için QextModbus sınıf nesnesine bağımlıdır ve ayrıca InterlockingManager sınıf nesnesine de bir bağımlılık içermektedir.

InterlockingManager sınıfının her bir talep sınıfına kendi içinde bağımlılık içermesi sebebiyle talep sınıf nesneleri ile InterlockingManager sınıf nesnesi arasında dairesel bir bağımlılık bulunmaktadır. Dairesel bağımlılıklar yazılım testedilebilirliği düşüren kötü yazılım tasarımına işaret etmektedir.

5.5 Demiryolu Anklaşman Yönetim Yazılımı v2

Demiryolu Anklaşman Yönetim Yazılımı v2 sürümü gerçeklenirken hedefimiz test edilebilirliği iyileştirilmiş bir yazılım bileşeni geliştirmektir. Bir yazılım bileşeninin test edilebilirliğinin sınanması için yazılım metrikleri yol gösterici olabilir. Fakat yazılım metrik ölçümleri iyi sonuçlar veren bir yazılım bileşeni test edilmeye çalışıldığında diğer yazılım varlıklarına bağımlılıkları olması sebebiyle zorlanılabilir. Burada bahsedilen test tipi birim testlerdir. Herhangi bir yazılım bileşeni için yazılan birim testlerin sağlaması gereken bazı gereksinimler vardır. Bunlar:

- İzole koşturulabilme (mümkün mertebe diğer yazılım bileşenlerden bağımsız olma)
- Eksiksiz olma (test ettiği birimi eksiksiz test etme)
- Tekrar edilebilir olma
- Otomatik koşturulabilme
- Hızlı koşturulabilme

olarak sayılabilir.

Bu gereksinimlerin sağlanabilmesi için yeni sürüm geliştirilirken birim test geliştirme kütüphanesi olarak ilk sürümün geliştirildiği yazılım kütüphanesinin alt kütüphanelerinden biri olan "Qt Test" kullanılmıştır. Bu birim test kütüphanesi birim testlerin otomatik ve tekrarlı koşturulmasını desteklemektedir.

Demiryolu Anklaşman Yönetim Yazılımı v2 sürümü, v1 sürümü ile benzer olarak C++ programlama dili ve nesneye yönelik yazılım geliştirme yaklaşımı kullanılarak 1 kişi tarafından 2 ay sürede geliştirilmiştir. Yazılımın v2 sürümünü geliştiren kişi v1 sürümünü geliştiren ekipte yer almamıştır.

5.5.1 Yazılım yaşam döngüsü seçimi

Emniyet kritik yazılım ürünleri, emniyet kritik olmayan yazılım ürünlerine benzerdirler. En önemli farkları emniyet kritik olan yazılım ürünlerinin sorumlu oldukları işleri yaparken neden olacakları bir hatanın felaketle sonuçlanabilecek olmasıdır. Bu nedenle emniyet kritik yazılım ürünleri geliştirilirken en küçük bir detayın bile gözden kaçmaması gerekir. Detaylar bu denli önemli olduğu için emniyet kritik yazılım ürünleri ağır dokümantasyon faaliyetlerinin olduğu yazılım geliştirme süreçleri ile geliştirilirler.

Yeni sürüm geliştirilirken amacımız geçmişte dokümantasyon faaliyetlerinin çok fazla olduğu bir yazılım geliştirme süreci ile geliştirilmiş olan bir yazılım ürününün test edilebilirliğini iyileştirmektir. Mevcut yazılım ürününün sorumlulukları ve işlevleri detaylı bir şekilde bilinmektedir. Aynı zamanda mevcut yazılım ürünü hakkında detaylı alan bilgisinede sahip durumdayız. Bu sebeple "Demiryolu Anklaşman Yönetim Yazılımı v2" geliştirilirken yazılım yaşam döngüsü olarak test yönelimli yazılım geliştirme seçilmiştir.

5.5.2 Yazılım tasarımı

"Demiryolu Anklaşman Yönetim Yazılımı v2" sürümü yazılım tasarımı yapılırken nesneye yönelik yazılım geliştirme paradigmasının temel yapı taşları olan soyutlama, kalıtım ve çok biçimlilik gibi özelliklerden, çeşitli yazılım tasarım kalıplarından

ve nesneye yönelik yazılım geliştirme ilkelerinden yararlanılmıştır. Uygulanan tasarım yaklaşımları arasında:

- Nesneler arası bağımlılıklar için "Bağımlılık evirme ilkesi (Dependency inversion principle)"

- KM tarafından istenilen taleplerin temsilini yapan sınıflar için soyutlama (abstraction) , kalıtım (inheritance) ve çok biçimlilik (polymorphism)
- Talep sınıf nesnelerinin çağrılması için "Açık/kapalı ilkesi (Open/closed principle)"
- Talep sınıf nesnelerinin oluşturma sorumluluğu için "Fabrika tasarım kalıbı (Factory design pattern)"
- Talep sınıf nesnelerinin sorumlu oldukları işi yapmaları için "Strateji tasarım kalıbı (Strategy design pattern)"
- Anklaşman yazılımı simülasyonu için "Gözlemci tasarım kalıbı (Observer design pattern)
- Birim testlerin koşturulması için gerekli bağımlılıkları sağlamak için Liskov yerine koyma ilkesi (Liskov substitution principle)

olarak sayılabilir.

5.5.2.1 Nesneler arası bağımlılıklar

Demiryolu anklaşman yönetim yazılımında nesneye yönelik yazılım metrik ölçüm sonuçlarına göre birçok sınıf birim testlerin geliştirilmesi ve koşturulmasını zorlaştıran bağımlılıklara sahiptir. Bu bağımlılıklar bir şekilde sağlansa dahi test yönelimli yaklaşıma göre geliştirilmesi düşünülen birim testlerin yavaş koşmasına neden olacaklardır. Bunun sebebi dış kaynakları (ağ, dosya sistemi vb.) kullanan bağımlılıkların olmasıdır. Nesnelerin diğer nesnelere olan bağımlılıklarının koparılması amacıyla nesneye yönelik yazılım geliştirme ilkelerinden "Bağımlılık evirme ilkesi (Dependency inversion principle)" kullanılmıştır. Bu ilkeye göre nesneler arasındaki bağımlılıklar için özelleştirilmiş nesneler yerine koşma zamanında gerektiğinde farklı bir nesne ile değiştirilebilen soyut nesneler kullanılmalıdır. "Bağımlılık evirme ilkesi" mevcut yazılımda gömülü anklaşman yazılımı ile haberleşmek için birçok yerde kullanılan Modbus endüstriyel iletişim protokolünün kullanılması amacıyla geliştirilmiş olan QextModbus sınıf nesnesi için kullanılmıştır.

5.5.2.2 Talep sınıfları

KM tarafından istenilen her bir talep için ayrı bir sınıf nesnesine bağımlılığı bulunan InterlockingManager sınıfında ise şöyle bir yol izlenmiştir. Bütün talep sınıf

nesnelerini temsil edebilecek soyut bir sınıf tasarlanmıştır (abstraction). Tasarlanan bu soyut sınıftan her bir talep için özelleştirilmiş bir sınıf nesnesi kalıtım ilkesi kullanılarak geliştirilmiştir. (inheritance, polymorphism). Çalışma zamanında istemciden gelen talep parametrelerine göre doğru sınıfın nesnesini oluşturabilmek amacıyla bu tip nesnelerin yaratılması sorumluluğu ayrı sınıfa verilmiştir. Bu sınıfın tasarımı için "Fabrika tasarım kalıbı (Factory design pattern)" kullanılmıştır. "Fabrika tasarım kalıbı" 'nın ve soyut talep sınıf nesnesinin kullanılmasıyla InterlockingManager sınıfı işlemekle sorumlu olduğu her bir talep için ayrı bir talep sınıf nesnesine bağımlılığı kalmamıştır. Burada sağlanan InterlockingManager sınıf nesnesinin bir çok talep sınıf nesnesine var olan bağımlılığını koparmaktır. Koparılan bağımlılıklar yerine sadece talep fabrikasını temsil eden sınıf nesnesine bir bağımlılık eklenmiştir. Bu yaklaşım ile beraber InterlockingManager sınıfında nesneye yönelik yazılım tasarım ilkelerinden "Açık/kapalı ilkesi (Open/closed principle)" uygulanmış olmaktadır. Bu adımda talep sınıflarının içinde saklı iş akışı ile ilgili herhangi işlem yapılmamıştır.

5.5.2.3 Talep sınıf nesnelerine strateji tasarım kalıbı uygulanması

Talep sınıflarının kod gözden geçirme süreçleri esnasında her bir talep sınıfının bir tür iş akışını takip ettiği tespit edilmiştir. İş akışı esnasında sınıf kendi için tanımlı bulunan bazı özel (private) metotlardan yararlanmaktadır. Yapılan inceleme sonucunda bu metotların çoğunun sınıfa bir iplik (thread) olarak çalışma özelliği kazandıran MThread sınıfından miras olarak geldiği tespit edilmiştir. Bu sınıfların tasarımı tekrar değerlendirildiğinde her birinin ayrı bir iplik (thread) olarak çalıştırılmasının mevcut yazılıma bir yarar sağlamadığı anlaşılmıştır. Bu sebeple bu sınıfların yeni tasarımlarımda iplik olarak çalışma özellikleri kaldırılmıştır. Algoritmalar ayrı ayrı incelendiğinde anklaşman yazılımı ile veri alışverişi yaparak özel bir iş akışını takip ettik ettikleri tespit edilmiştir. Algoritmaların özel iş akışını takip etmelerinin yanında çalışmak için aynı parametrelere ihtiyaç duydukları görülmüştür. Algoritmalar sadece davranış olarak farklı iş akışlarını takip etmektedirler.

Yapılan değerlendirme sonucunda algoritmaların çalışma zamanında dinamik olarak değiştirilebilecekleri sonucuna varılmıştır. Bu sebeple talep sınıflarının soyutlanması için tasarlanan sınıfa bir metot eklenmiştir. Her bir özel talep sınıfı bu metodun

davranışı değiştirerek kendisine özel iş akışını uygulayabilecektir. Burada yapılan işlem "Strateji tasarım kalıbı (Strategy design pattern)" 'nın uygulamasıdır.

Yeni tasarlanan yazılımda soyutlama, kalıtım, çok biçimlilik, çeşitli tasarım kalıpları ve nesneye yazılım tasarım ilkeleri birlikte uygulanarak geliştirilecek birim testlerin hızlı ve bağımsız koşturulma gereksinimleri sağlanmaya çalışılmıştır. Yazılım tasarımında yukarıda detaylandırılan yaklaşımlara ait sınıf tasarım diyagramları Şekil 5.8'de ve Şekil 5.9'da gösterilmiştir.

5.5.2.4 Anklaşman simülatörü için gözlemci tasarım kalıbı uygulanması

Demiryolu anklaşman yönetim yazılımı demiryolu trafiğinin yönetilmesi için anklaşman yazılımı ile birlikte çalışmaktadır. Bu birlikte çalışma ortak bir veri yapısına veri yazma ve okuma işlemleri olarak yapılmaktadır. Talepler demiryolu anklaşman yazılımı tarafından ortak veri yapısına yazılmakta, anklaşman yazılımı bu talepleri ortak veri yapısından okumakta, talebi işlemekte ve sonucunu aynı ortak veri yapısına yazmaktadır. Yazılan bu sonuçlar demiryolu anklaşman yönetim yazılımı tarafından belirli aralıkla düzenli olarak okunmakta ve saha durum veri biçimine çevrilerek KM yazılımına gönderilmektedir. "Demiryolu Anklaşman Yönetim Yazılımı v2" sürümünün birlikte çalışacağı bir anklaşman yazılımı bulunmamaktadır. Bu nedenle anklaşman yazılımı ile benzer işlevleri bulunan bir simülatör tasarımı yapılmıştır. Anklaşman simülatörü ile demiryolu anklaşman yönetim yazılımı arasındaki veri alışverişini sağlamak için ortak veri yapısı sağlanmıştır. Burada duyduğumuz gereksinim demiryolu anklaşman yönetim yazılımı ortak veri yapısına bir talep yazdığında bu talebi anklaşman simülatörünün okuması ve gerekli çıktıyı üretip demiryolu anklaşman yönetim yazılımının okuması için ortak veri yapısına yazmasıdır. Bu amaçla "Gözlemci tasarım kalıbı (Observer design pattern)" anklaşman simülatörüne uygulanmıştır.

Bu tasarım kalıbına göre veri yazma ve okuma ile amacıyla kullanılan sınıf nesnesi ortak veri yapısına demiryolu anklaşman yönetim yazılımı tarafından yeni bir veri yazıldığında anklaşman simülatör nesnesini bilgilendirmektedir. Verinin değiştiğini tespit eden anklaşman simülatör nesneside gerekli iş akışını uygulamaktadır.

5.5.3 Yazılım gerçekleme ve yazılım testleri

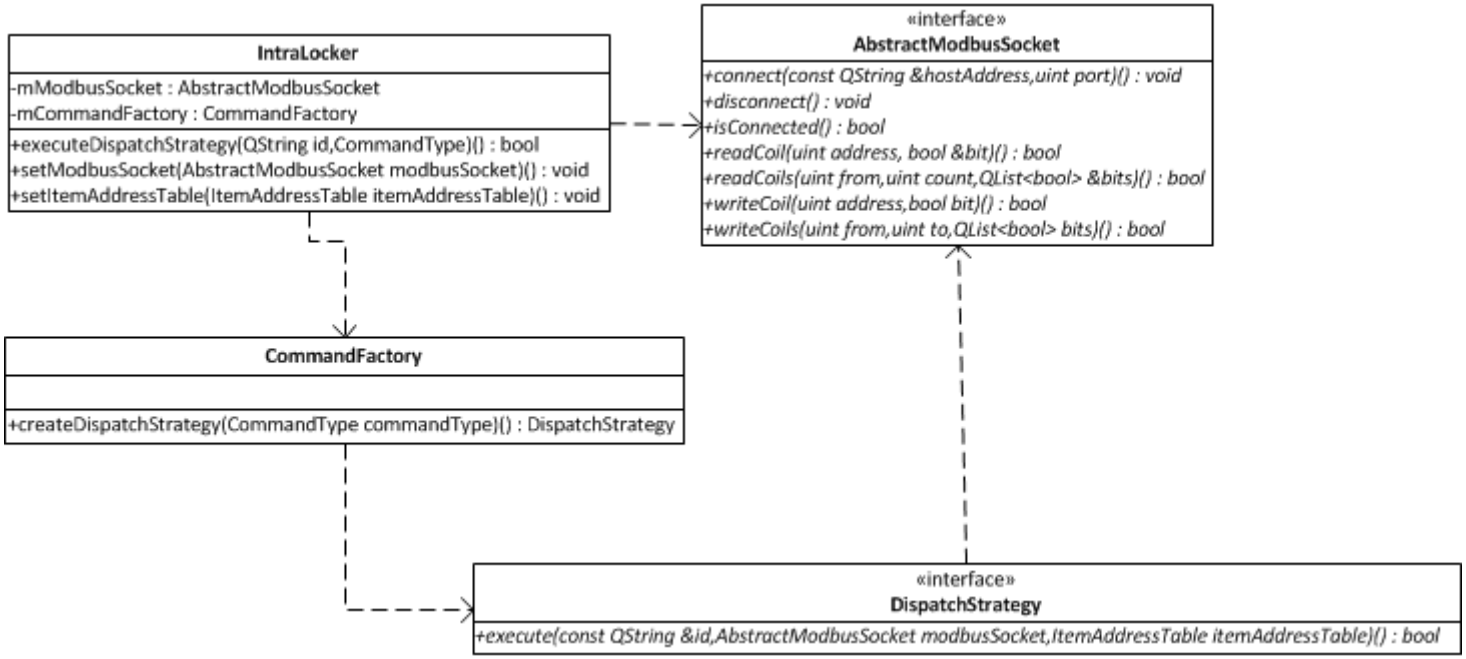
"Demiryolu Anklaşman Yönetim Yazılımı v2" sürümü gerçeklenirken ilk sürümün gerçeklendiği Qt C++ yazılım kütüphanesi kullanılmıştır [37]. Birim testler için Qt kütüphanesinin sağladığı bir alt kütüphane olan Qt Test birim test geliştirme yazılım kütüphanesi kullanılmıştır. Birim testler geliştirilirken test yönelimli yazılım geliştirme yaklaşımı kullanılmıştır. Öncelikle "Demiryolu Anklaşman Yönetim Yazılımı v2" sürümünde bulunması gereken bir işlevle ilgili bir test yazılmıştır. Henüz o işlev yazılıma eklenmediği için test başarısız olarak koşturulmuştur. Ardından testin geçmesi için gerekli en az kodlama yapılmıştır. Testin başarılı olarak koşturulduğu gözlendikten sonra mevcut işlev için gerekli kodlama yapılmış ve o işlev için ilk başta geliştirilen birim test tekrar koşturulmuştur. Böylece o işlevin doğruluğunu sınayan bir birim test yazılıma eklenmiş bulunmaktadır. Tüm birim testlerin bu yaklaşımla geliştirilmesi demiryolu anklaşman yönetim yazılımının yeni sürümünü devamlı test etme öncelikli bir yapıda geliştirilmesine neden olmuştur. Bu yaşam döngüsünün sonucunda test edilebilir bir yazılım ürünü ortaya çıkmıştır.

Demiryolu anklaşman yönetim yazılımının ilk sürümü, anklaşman yazılımı ile işbirliği içinde çalışmaktadır. Geliştirilen yeni sürümün işbirliği yapacağı bir anklaşman yazılımı bulunmadığı için anklaşman yazılımının benzetimini yapan bir simülatör sınıfı geliştirilmiştir. Bu simülatör sınıfı demiryolu anklaşman yazılımı tarafından işlenmesi gereken taleplerin büyük bir çoğunluğu benzetecek şekilde tasarlanmıştır.

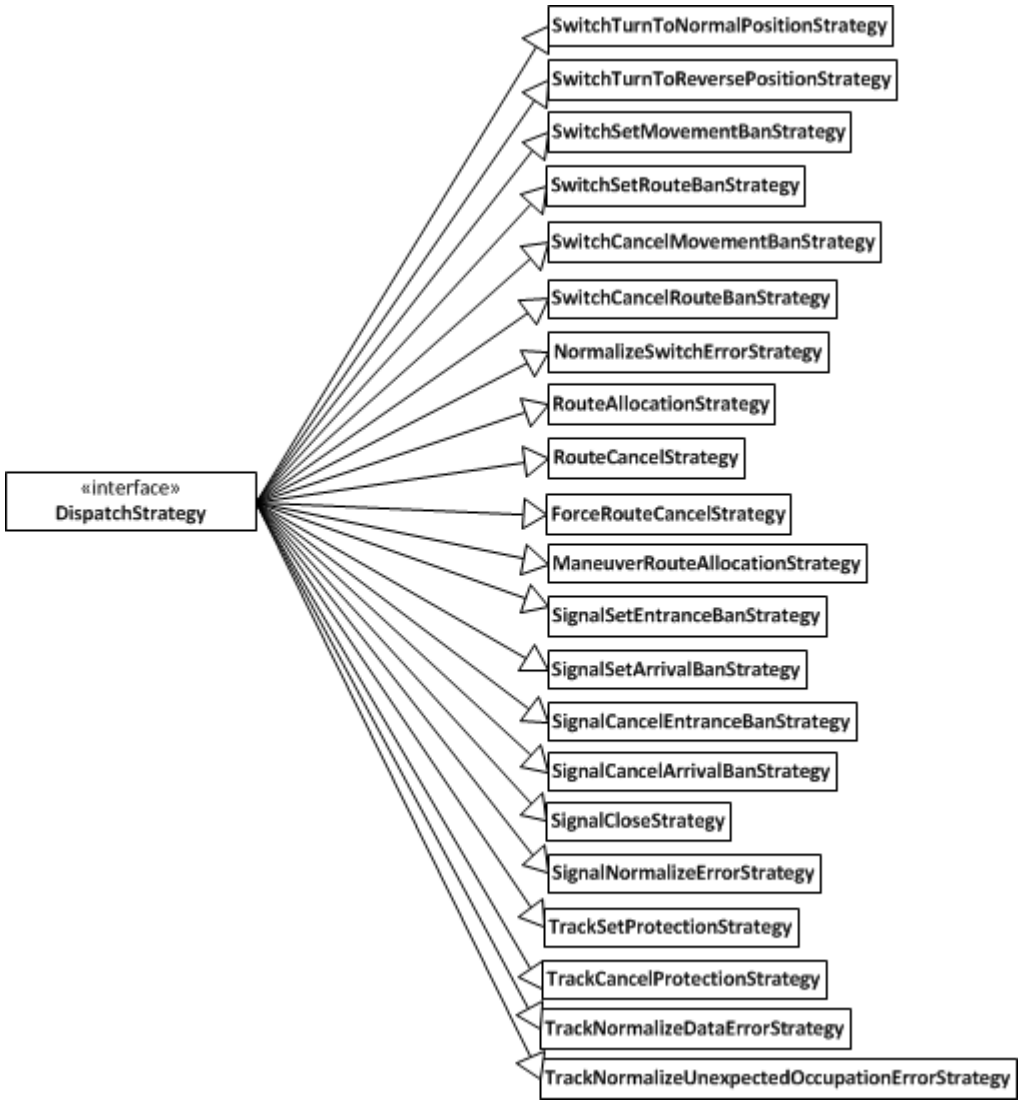
Yeni sürümde bağımlılıklar soyut nesneler olarak tasarlandığı için birim testlerin koşturulması amacıyla soyut nesnelerden türeyen "Stub" nesneler geliştirilmiştir. Bunun yanında birim testlerin koşturulması için gerekli veriyi sağlamak amacıyla bir demiryolu test sahası tasarlanmıştır. Demiryolu test sahasında bulunan demiryolu ekipmanlarının yeni geliştirilen sürümde kullanılabilmesi için gerekli veri XML veri biçimi kullanılarak üretilmiştir.

Testlerin koşturulduğu demiryolu sahasına ait çizim Şekil 5.10'da gösterilmiştir.
Demiryolu test sahasında:

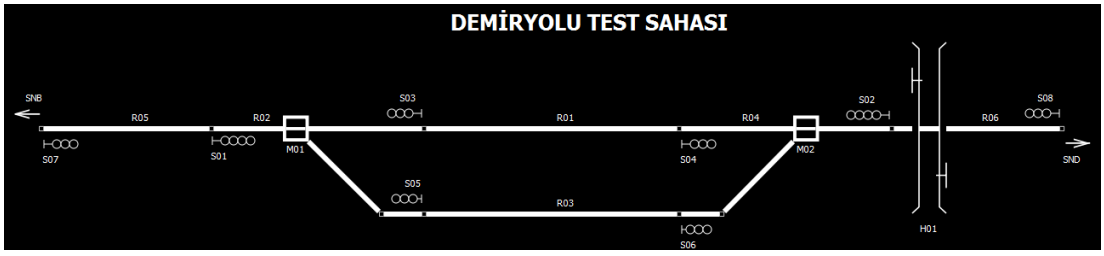
- 2 adet makas
- 6 adet ray devresi



Şekil 5.8 : Demiryolu anklaşman yönetim yazılımı v2 sınıf diyagramı.



Şekil 5.9 : DispatchStrategy arayüzünden türeyen sınıflar.



Şekil 5.10 : Demiryolu test sahası.

- 8 adet sinyal
- 1 adet hemzemin geçit
- 10 adet güzergah

bulunmaktadır.

Test sahası için çeşitli birim testler geliştirilmiştir. Koşturulan birim testlerin açıklamaları ve tipleri Çizelge 5.2'de verilmiştir.

Çizelge 5.2 : Koşturulan testlerin açıklamaları ve tipleri.

Test Adı	Test Açıklaması	Test Tipi
test_initTestCase 1	Modbus soketinin var olmaması testi	Negatif
test_initTestCase 2	Modbus soketinin bağlı olmama testi	Negatif
test_initTestCase 3	Modbus soketinin bağlanma testi	Pozitif
test_nonExistentSwitchMoveBan	Sahada var olmayan bir makas için hareketi bloke etme testi	Negatif
test_setSwitchMoveBan	Makas hareketi bloke etme testi	Pozitif
test_cancelSwitchMoveBan	Makas hareket bloke iptal etme testi	Pozitif
test_turnSwitchNormal	Makas normal pozisyona dönme testi	Pozitif
test_turnSwitchReverse	Makas ters pozisyona dönme testi	Pozitif
test_setSwitchRouteBan	Makas güzergah bloke etme testi	Pozitif
test_cancelSwitchRouteBan	Makas güzergah blokesi iptal testi	Pozitif
test_fixSwitchIndicationError	Makas indikasyonsuzluk arızası düzeltme testi	Pozitif
test_normalizeSwitchDataError	Makas veri arızası düzeltme testi	Pozitif
test_allocateRoute	Güzergah tahsis testi	Pozitif
test_cancelRoute	Güzergah tahsis iptal testi	Pozitif
test_forceRouteCancel	Güzergah tahsis zorunlu iptal testi	Pozitif
test_allocateAutomaticRoute	Otomatik güzergah tahsis testi	Pozitif
test_allocateManeuverRoute	Manevra güzergahı tahsis testi	Pozitif
test_setTrackProtection	Ray devresi koruma testi	Pozitif
test_cancelTrackProtection	Ray devresi koruma iptal etme testi	Pozitif
test_normalizeTrackDataError	Ray devresi veri arızası düzeltme testi	Pozitif
test_normalizeTrackUnexpected OccupationError	Ray devresi beklenmedik meşguliyet arızası düzeltme testi	Pozitif
test_setSignalEntranceBan	Sinyal başlangıç bloke etme testi	Pozitif
test_cancelSignalEntranceBan	Sinyal başlangıç bloke iptal etme testi	Pozitif
test_setSignalArrivalBan	Sinyal varış bloke etme testi	Pozitif
test_cancelSignalArrivalBan	Sinyal varış bloke iptal etme testi	Pozitif
test_closeSignal	Sinyal kapama testi	Pozitif
test_cleanupTestCase	Modbus soketi bağlantı koparma testi	Pozitif

Koşturulan birim testlerin yazılım kodunda kapsadığı ifade, metot ve dal metrik ölçümleri Çizelge 5.3'te verilmiştir. Kod kapsama ölçüm sonuçlarından anlaşılacağı üzere birim testler, ifade ve metot kapsama ölçümlerine göre oldukça başarılı olmuşlardır. Dal kapsama ölçümleri değerlendirildiğinde derleyici optimizasyonu olmaması nedeniyle ifade ve metot kapsama sonuçlarına göre daha düşük sonuçlar elde edilmiştir.

Geliştirilen birim testler koşturulduğunda oluşan çıktı Şekil 5.11'de gösterilmiştir. Test sonuçlarından anlaşılacağı üzere yeni sürüm için geliştirilen tüm birim testler başarılı olmaktadır.

Çizelge 5.3 : İfade, metot, dal kapsama metrik ölçüm sonuçları.

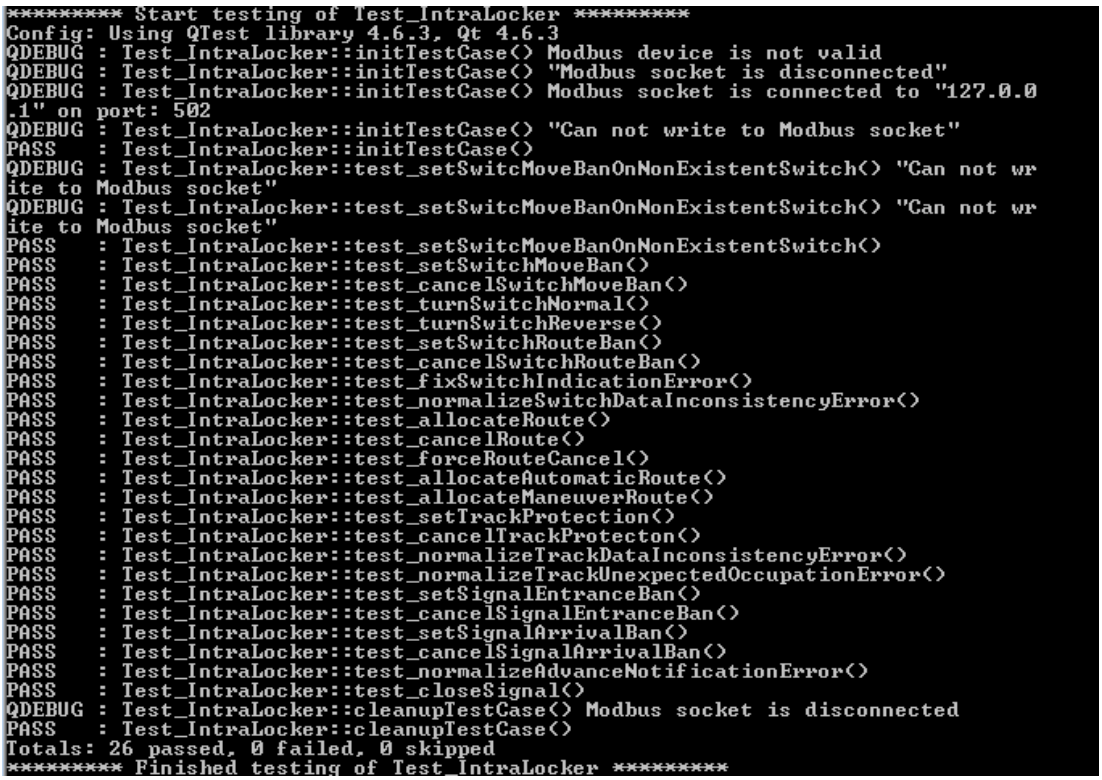
Dosya İsmi	İfade Kapsama		Metot Kapsama		Dal Kapsama	
addressparser.cpp	95.0 %	57/60	100.0 %	7/7	49.1 %	159/324
automaticrouteallocation.cpp	100.0 %	16/16	100.0 %	6/6	60.0 %	6/10
commandfactory.cpp	100.0 %	53/53	100.0 %	5/5	64.8 %	46/71
common.h	100.0 %	48/48	-	0/0	-	0/0
dispatchstrategy.cpp	100.0 %	2/2	75.0 %	3/4	50.0 %	3/6
dispatchstrategy.h	100.0 %	1/1	100.0 %	1/1	-	0/0
forceroutecancel.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
intralocker.cpp	92.3 %	24/26	100.0 %	7/7	50.0 %	12/24
maneuverrouteallocation.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
normalizeswitcherror.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
routeallocation.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
routecancel.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
signalcancelarrivalban.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
signalcancelentranceban.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
signalclose.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
signalnormalizeerror.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
signalsetarrivalban.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
signalsetentranceban.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
switchcancelmoveban.cpp	100.0 %	13/13	100.0 %	6/6	60.0 %	6/10
switchcancelrouteban.cpp	100.0 %	13/13	100.0 %	6/6	60.0 %	6/10
switchsetmoveban.cpp	100.0 %	13/13	100.0 %	6/6	60.0 %	6/10
switchsetrouteban.cpp	100.0 %	13/13	100.0 %	6/6	60.0 %	6/10
switchturntonormal position.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
switchturntoreverse position.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
trackcancelprotection.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
tracknormalizedataerror.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
tracknormalizeunexpected occupationerror.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
tracksetprotection.cpp	100.0 %	14/14	100.0 %	6/6	60.0 %	6/10
util.h	100.0 %	5/5	100.0 %	1/1	60.0 %	6/10

5.5.4 Yazılım kalitesinin ölçülmesi ve sonuçların analizi

"Demiryolu Anklaşman Yönetim Yazılımı v2" sürümüne ait çeşitli yazılım metrik değerleri Understand aracı ile ölçülmüştür.

Yeni geliştirilen sürümde:

- Proje metriklerine göre
  - 28 adet sınıf bulunmaktadır
  - Toplam satır sayısı 1528'dir
  - Toplam boş satır sayısı 306'dır
  - Toplam kod satır sayısı 995'tir
- Sınıf metriklerine göre
  - Sınıfların %93'nün çevrimsel karmaşıklık değeri 3 olarak ölçülmüştür
  - Talep sınıf nesnelerini yaratmakla sorumlu sınıfın çevrimsel karmaşıklığı 28 olarak ölçülmüştür
  - Ekipman adres değerlerini XML biçimli dosyadan okumakla sorumlu sınıf nesnesinin çevrimsel karmaşıklığı 7 olarak ölçülmüştür
- Nesneye yönelik yazılım metriklerine göre ölçüm sonuçları Çizelge 5.4'te gösterilmiştir



Şekil 5.11 : Test sonuçları çıktısı.

Ölçüm sonuçları değerlendirildiğinde yazılımın yeni sürümünde proje metrik ölçümlerine göre sınıf sayısında bir artış olmuştur. Toplam satır sayısı, toplam yorum satırı sayısı, toplam kod satır sayısı gibi metriklerde ise düşme gözlenmiştir.

Bunun başlıca nedeni yeni geliştirilen sürümün ana amacının test edilebilirliği yüksek bir yazılım ürünü geliştirmek olmasıdır. Bu nedenle yazılımın ilk sürümüne göre bazı işlevleri tam olarak desteklememektedir.

Çizelge 5.4 : Demiryolu anklaşman yönetim yazılımı v2 metrik ölçüm sonuçları.

Sınıf Adı	LCOM	CBO	RFC	WMC
IntraLocker	40	5	5	5
AbstractModbusSocket	0	2	7	7
AutomaticRouteAllocationStrategy	0	4	4	3
AddressParser	0	7	5	5
CommandFactory	0	23	3	3
DispatchStrategy	0	3	1	1
ForceRouteCancelStrategy	0	4	4	3
ManeuverRouteAllocationStrategy	0	4	4	3
NormalizeSwitchErrorStrategy	0	4	4	3
RouteAllocationStrategy	0	4	4	3
RouteCancelStrategyStrategy	0	4	4	3
SignalCancelArrivalBanStrategy	0	4	4	3
SignalCancelEntranceBanStrategy	0	4	4	3
SignalCloseStrategy	0	4	4	3
SignalNormalizeErrorStrategy	0	4	4	3
SignalSetArrivalBanStrategy	0	4	4	3
SignalSetEntranceBanStrategy	0	4	4	3
SwitchCancelMovementBanStrategy	0	4	4	3
SwitchCancelRouteBanStrategy	0	4	4	3
SwitchSetMovementBanStrategy	0	4	4	3
SwitchSetRouteBanStrategy	0	4	4	3
SwitchTurnToNormalPositionStrategy	0	4	4	3
SwitchTurnToReversePositionStrategy	0	4	4	3
TrackCancelProtectionStrategy	0	4	4	3
TrackNormalizeDataErrorStrategy	0	4	4	3
TrackNormalizeUnexpected OccupationErrorStrategy	0	4	4	3
TrackSetProtectionStrategy	0	4	4	3
Util	0	1	1	1

Sınıf metrik ölçümlerine göre yazılımındaki sınıfların bir çoğunun çevrimsel karmaşıklık değeri oldukça düşmüştür. Bu değerin düşmesi sınıf test edilebilirliğinin yükselmesi ile ters orantılıdır. Yeni geliştirilen sınıflardan bir tanesinin çevrimsel karmaşıklık değeri 28'dir. Bu yüksek değer sınıfa atanan sorumluluktan kaynaklanmaktadır. Bu sınıfın sorumluluğu çeşitli talep sınıf nesneleri yaratmaktır. Çevrimsel karmaşıklık değeri 7 olarak ölçülen bir sınıf daha bulunmaktadır.

Bu sınıfın sorumluluğu ise demiryolu test sahasına ait verileri XML biçimli bir dosyadan okumaktır. Genel olarak bakıldığında yeni geliştirilen sınıfların çevrimsel karmaşıklık değerleri düşmüştür.

Nesneye yönelik metrik ölçüm sonuçlarına göre en büyük iyileşme sınıfların kendi içindeki metot uyumsuzluğu değerlerinin düşmesi olmuştur. Bir önceki sürümde en yüksek LCOM değerini sahip sınıf olarak görülen InterlockingManager, yeni geliştirilen sürümde eski değerinin yarısından daha düşük LCOM değerine sahip olmuştur. Diğer sınıfların ise LCOM değerleri 0 olarak ölçülmüştür. Bu sonuca bakarak yeni geliştirilen sürümünde bulunan sınıfların sorumluluklarının daha tutarlı hale getirildiği ve test yükünün azaltıldığı söylenebilir. Sınıflara ait nesnenin doğrudan bağımlı olduğu nesne sayılarında eski sürüme elle tutulur bir iyileşme sağlanmıştır. Eski sürümde diğer sınıf nesnelerine bağımlılık değerleri arasında en küçük değer 4 olarak ölçülmüşken yeni sürümde bulunan sınıfların %89'unun CBO değeri 4 olarak ölçülmüştür. Bunun dışında kalan sınıflarda ölçülen değerler 5,7 ve 23'tür. 23 tane farklı nesneye bağımlılığı bulunan sınıf ise talep sınıflarının yaratmakla sorumlu olan sınıftır. Yeni sürümde genel olarak CBO değerleri düşük ölçülmüştür. CBO değerlerinin düşmesi sınıfların test edilebilirliğini artıran bir etkendir. Bir önceki sürüme göre sınıfların tetiklediği metot sayısı ve sınıf ağırlıklı metot ölçüm değerleri daha düşük olarak ölçülmüştür. Sınıfların tetiklediği metot sayılarındaki azalma sınıfların karmaşıklığının azalması anlamına gelmektedir. Sınıfın karmaşıklığını azalması sınıfın test faaliyetlerini azaltan ve test edilebilirliğini artıran bir etkendir. Sınıf ağırlıklı metot sayılarındaki azalma sınıfların daha sürdürülebilir (maintainable) hale geldiğinin göstergesidir. Sürdürülebilir sınıflar test edilebilirliği yüksek sınıflardır ve bakım yapma faaliyetleri için az kaynak (zaman ,efor) gerektirirler.

5.6 Çalışmanın Geçerliliğine Yönelik Olası Tehditler (Threats to Validity)

Herhangi bir çalışma ile benzer olarak bu çalışmanın da geçerliliğine ilişkin bir bazı olası tehditler bulunmaktadır.

Öncelikle çalışma emniyet kritik yazılım sistemleri üzerine olduğu için seçilen örnek vakanın boyutu (kod satır sayısı, süre, ekip büyüklüğü vb.) yeterince büyük değildir. Bunun başlıca nedeni emniyet kritik yazılım sistemlerinin çoğunlukla açık kaynak kod alt yapısı ile geliştirilmemesidir.

Bu sebeple daha büyük boyutlu bir vaka üzerinde çalışılamamıştır. Seçilen vaka yazarın daha önce çalıştığı ticari bir üründen alınmıştır. Vaka boyutunun yeterince büyük olmaması çalışmadan elde edilen istatistiki verilerin geçerliliğini tartışmalı hale getirebilir.

Bununla beraber diğer bir tartışmalı nokta ise sadece metrik analizi yaparak test yönelimli geliştirmenin yazılım test edilebilirliğine katkısının ölçülemeyeceğidir. Yazılım test edilebilirliği, farklı çalışmalarda farklı bakış açılarıyla ele alınmaktadır ve doğrudan yazılım metrikleri ile ölçülmemektedir. Buna rağmen bu çalışma kapsamında ele alınan yazılım metrikleri çevrimsel karmaşıklık ve uyum, bağımlılık gibi yazılım test edilebilirliğini doğrudan etkileyen etmenlerdir. Bu metriklerin ölçüm sonuçlarındaki iyileşme yazılım test edilebilirliğini olumlu olarak etkileyecektir.

Ayrıca vaka çalışmasında uygulanan test yönelimli geliştirmenin doğru olarak uygulanmış olup olmadığının doğrulamasının yetersizliği de tartışılabilir. Test yönelimli geliştirmenin uygulaması ile ilgili doğrulama kod kapsama (test coverage) metrikleri kullanılarak yapılabilir. Bu amaçla "v2" sürümü için sağlanan kod kapsama metrikleri değerlendirildiğinde test yönelimli geliştirme yaklaşımının başarılı olarak uygulandığı gözlenmektedir.

Tartışılabilecek bir diğer nokta ise test yönelimli geliştirme yaklaşımı uygulanarak yazılım test edilebilirliğinin iyileştirilip, iyileştirilemeyeceğidir. Test yönelimli geliştirme yaklaşımı yazılım kodunu üretmeden önce hedef kodu test edecek test kodunun üretilmesi mantığına dayanmaktadır. Test yönelimli geliştirme test hedefli bir yazılım geliştirme yaklaşımı olduğu için hedef yazılım kodunun test edilebilir olmasını zorlayacaktır. Burada kastedilen zorlama eğer test yönelimi geliştirme doğru olarak uygulandığı takdirde özellikle statik bağımlılıkların arayüz veya soyut sınıf nesneleri ile sağlanması ve mevcut kodun karmaşıklığının azaltılması yoluyla yapılacaktır.

Tartışmalı olan son nokta yazarın, vaka çalışmasında test yönelimli geliştirme yaklaşımını uygularken yazılım test edilebilirliğini arttırmak için fazla çaba göstermiş olabileceğidir. Bu konuda ki endişe yazarın sorumluluğunun bilincinde olan bir uzman yazılım mühendisi olduğu gerçeğini göz önünde bulundurarak bir ölçüde azaltılabilir.

6. SONUÇ, ÖNERİLER VE GELECEK ÇALIŞMA

Bu çalışmada yazılım test edilebilirliği ile yazılım metrikleri arasındaki ilişki incelenmiş ve test yönelimli yazılım tasarım ve geliştirme yöntem ve ilkeleri kullanılarak yazılım test edilebilirliğinin iyileştirilebileceği gösterilmiştir.

Uyguladığımız yaklaşım nesneye yönelik yaklaşımla geliştirilen yazılım sistemlerinde test yönelimli yazılım tasarım ve geliştirme yöntem ve ilkelerinin kullanılması ve yazılım metrikleri ile yazılım test edilebilirliğinin değerlendirilmesinden oluşmaktadır. Yazılım test edilebilirliğinin ölçülmesi için yazılım metriklerine başvurulmuştur. Metrikler kullanılarak yazılım test edilebilirliği değerlendirilen örnek bir yazılım ürünü üzerinde çalışılmıştır.

Yazılım ürününün "v1" sürümünün test edilebilirliği nesneye yönelik olmayan ve nesneye yönelik olan yazılım metrikleri kullanılarak ölçülmüştür. Alınan sonuçlara göre "v1" sürümünün test edilebilirliği düşük olarak değerlendirilmiştir. Aynı ürünün "v2" sürümü nesneye yönelik yazılım geliştirme yaklaşımı ile test yönelimli yazılım tasarım ve geliştirme yöntem ve ilkeleri kullanılarak geliştirilmiştir. "v2" sürümünün test edilebilirliği "v1" sürümü ile aynı metrikler kullanılarak ölçülmüştür. Bunun yanında "v2" sürümünde uygulanan test yönelimli geliştirme yaklaşımının etkinliği kod kapma metrikleri kullanılarak ölçülmüştür.

"v2" sürümünde "v1" sürümüne göre yazılımı oluşturan sınıfların %93'ünün çevrimsel karmaşıklık değeri 3'e düşürülmüştür. Çevrimsel karmaşıklık değeri ilk sürüme göre yüksek olarak ölçülen tek sınıf talep nesnelerini yaratmakla sorumlu olan sınıftır. Bu sınıfın sorumluluğu sebebiyle çevrimsel karmaşıklık değeri yüksek olarak çıkmaktadır. Çevrimsel karmaşıklık değerinin düşük olarak ölçülmesi sınıf yapılarının basitleştirildiğini göstermektedir. Sınıfların %96'sının sınıf metotları arası uyumsuzluk değeri 0 olarak ölçülmüştür. "v1" sürümünde en yüksek sınıf metotları arasındaki uyumsuzluk değerine sahip olan sınıfın "v2" sürümündeki değeri 40'a düşürülmüştür. Bu metrik değerine bakarak sınıf sorumlulukların tutarlı, metotların ise uyumlu hale getirildiğini söyleyebiliriz. Sınıfların %89'unun nesneler arasındaki bağımlılık değeri 4 olarak ölçülmüştür.

Bu metrik değerine göre yazılım ürünü için geliştirilen birim testlerin izole olarak koşturulabilmesinin kolaylaştığını söyleyebiliriz. Sınıfların %100'ünün sınıfın tetiklediği metot sayısı ve sınıf başına ağırlıklı metot sayısı değerleri 7 veya altında ölçülmüştür. Bu iki metrik değerine göre sınıfların yapılarının basitleştirildiğini, sınıfların detaylarının anlaşılmasının kolaylaştığını ve sınıfların test yüklerinin azaltıldığını söyleyebiliriz.

Çalışılan sistem, emniyet kritik yazılım ürünü olarak seçilmiştir. Bunun başlıca nedeni emniyet kritik yazılım sistemlerin sorumlu oldukları işi yaparken hata yapmaları durumunda felaket (ölüm, kalıcı sakatlık, çevrede veya kullanılan donanımda cidde tahribat) senaryolarına neden olabilmesidir. Bu nedenle emniyet kritik yazılım sistemlerinden hataların ayıklanması diğer yazılım sistemlerine göre daha önemlidir. Felaketle sonuçlanabilecek hata durumlarının tümünün ayıklanması için yazılım sisteminin planlamasından gerçeklenmesine kadar geçen tüm evrelerde yazılım test edilebilirliği üzerinde durulmalıdır. Çalışılan sistemin emniyet kritik olarak seçilmesinin bir diğer nedeni ise bu tip sistemlerin emniyet kritik işler yapan donanımlara sahip olmasıdır. Donanım ve yazılımın birlikte uyumlu çalışmasının gerektiği emniyet kritik sistemlerde yazılım test edilebilirliği daha ön planda değerlendirilmelidir. Çünkü bu tip sistemlerde yazılım daha karmaşık olmaktadır. Karmaşık yazılımda hata çıkma olasılığı daha fazladır.

Herhangi bir yazılım ne kadar iyi test edilirse edilsin tamamen hatasız olarak kabul edilemez. Yazılımın tüm giriş ve buna karşılık gelen çıkışları ile test etmek proje zaman ve bütçe kısıtları nedeniyle olası değildir. Bu açıdan bakıldığında yazılım test edilebilirliğine önem verilmesi gereklidir. Yazılım test edilebilirliği yazılım kalitesi ile doğrudan ilişkilidir. Kalitenin yüksek olduğu yazılım ürünlerinde test edilebilirlikte yüksektir. Yazılım kalitesinin artırılması için yazılım tasarımına önem verilmelidir. Bu amaçla doğru yöntemlerin ve ilkelerin seçilmesi gerekir. Burada kullanılabilecek yaklaşım herkes tarafından kabul görmüş ve belirli bir temele dayanan yöntemlerin ve yaklaşımların seçilmesidir. Ama bilinmelidir ki hiçbir yöntem ve yaklaşım mucize yaratmaz.

Bu çalışmadan elde edilen sonuçlar emniyet kritik olsun olmasın yazılım test edilebilirliğinin önemsendiği nesneye yönelik olarak geliştirilen herhangi bir yazılım ürününe uygulanabilir. Bu çalışmadan edinilen tecrübe ileride yazılım testedilebilirliğinin ön planda olduğu herhangi bir projede kullanılacaktır.

KAYNAKLAR

[1] **EN, B.** (2011). 50128: 2011. Railway applications—Communication, signaling and processing systems—Software for railway control and protection systems, BSI Standards Publication.

[2] **Jonsson, H., Larsson S. ve Punnekkat S.** (2012). Agile Practices in Regulated Railway Software Development, In Proc. of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Dallas, Texas, pp. 355-360. IEEE.

[3] **Ge X., Paige R. F. ve McDermid J. A.** (2010). An Iterative Approach for Development of Safety-Critical Software and Safety Arguments, Agile Conference (AGILE), 2010, pp. 35-43. IEEE.

[4] **Fitzgerald B., Stol K., O'Sullivan R., ve O'Brien D.** (2013). Scaling agile methods to regulated environments: An industry case study, In 35th International Conference on Software Engineering (ICSE-SEIP) pp. 863-872, IEEE.

[5] **Wolff S.** (2012). Scrum Goes Formal: Agile Methods for Safety-Critical Systems, Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in., pp. 23-29. IEEE.

[6] **Shrivastava D. P. ve Jain R. C.** (2011). Unit Test Case Design Metrics In Test Driven Development, Communications, Computing and Control Applications (CCCA), pp. 1-6. IEEE.

[7] **Bruntink, M. ve Deursen, A.** (2004). Predicting class testability using object-oriented metrics. Source Analysis and Manipulation, 2004 IEEE International Workshop, pp.136-145. IEEE.

[8] **Khanna, P.** (2014). Testability of Object-Oriented Systems: An AHP-Based Approach for Prioritization of Metrics. 2014 International Conference on Contemporary Computing and Informatics (IC31). pp. 273-281. IEEE.

[9] **Alwardt, L. A.**, **Mikeska, N., Pandorf, J. R. ve Tarpley, R. P.** (2009). A Lean Approach to Designing for Software Testability. In AUTOTESTCON. pp. 178-183. IEEE.

[10] **Sahay G.** (2011). Design for testability in embedded software projects. Digital Avionics Systems Conference (DASC). pp. 7D4-1.IEEE.

[11] **Joshi, M. ve Sardana, N.** (2014). Design and code time testability analysis of object oriented systems. Computing for Sustainable Global Development (INDIACom). pp. 590-592. IEEE

[12] **Tahir, A., MacDonnell, G. S., Buchan, J.** (2014). Understanding Class-level Testability Through Dynamic Analysis. arXiv preprint arXiv:1410. 1155.

[13] **Kanstrén, T.** (2008). A study on design for testability in component-based embedded software. In Software Engineering Research, Management and Applications, 2008. SERA'08. Sixth International Conference on (pp. 31-38). IEEE.

[14] **Alanen, J., ve Ungar, L. Y.** (2011). Comparing software design for testability to hardware DFT and BIST. In AUTOTESTCON, 2011 IEEE (pp. 272-278). IEEE.
[15] **Mulo, E.** (2007). Design for Testability in Software Systems. Master's Thesis, *35*(10), 101-107.
[16] **NASA Software Safety Guidebook** (2004). NASA-GB-8719.13, National Aeronautics and Space Administration, Vaşington.
[17] **EN, B.** (2010). 61508-3: 2010 Functional safety of electrical/electronic/programmable electronic safety-related systems.
[18] **Software Formal Inspections Guidebook** (1993). NASA-GB-A302, National Aeronautics and Space Administration, Vaşington.
[19] **Bray, M., Brune, K., Fisher, D. A., Foreman, J., ve Gerken, M.** (1997). C4 Software Technology Reference Guide – A Prototype (No. CMU/SEI-97-HB-001). Carnegie-Mellon Univ Pittsburg Pa Software Engineering Inst.
[20] **Erdemir, U.**, **Tekin, U. ve Buzluca F.** (2008). Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi. Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu.
[21] **ISO/IEC-25010** (2011). Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Systems and software quality models, *International Organization for Standardization/International Electrotechnical Commission*, New York.
[22] **Fenton, N. E. ve Pfleeger, S. L.** (1996). Software Metrics – A Rigorous and Practical Approach. pp. 5. International Thompson Computer Press.
[23] **McCabe, T. J.** (1976). A complexity measure. Software Engineering, IEEE Transactions on, (4), 308-320.
[24] **Chidamber, S.R. ve Kemerer, C. F.** (1994). A metric suite for object oriented design. Software Engineering, IEEE Transactions on, 20(6), 476 – 493.
[25] **Briand, L. C., Daly, J., Porter, V. ve Wüst, J.** (1998). A Comprehensive Empirical Validation of Design Measures for Object Oriented Systems. 5th International Symposium on Software Metrics. pp. 246.
[26] **e Abreu, F. B.** (1995). Design metrics for object-oriented software systems. In Workshop Quantitative Methods for Object-Oriented Systems Development, *ECOOP* (Vol. 95).
[27] **Bansiya, J., ve Davis, C. G.** (2002). A hierarchical model for object-oriented design quality assessment. Software Engineering, IEEE Transactions on,*28*(1), 4-17.
[28] **Code coverage.** (t.y.). Wikipedia. Alndığı tarih: 26.06.2015, adres: http://en.wikipedia.org/wiki/Code_coverage.
[29] **Baudry, B., Traon, Y. L., Sunyé, G.** (2002). Testability analysis of a UML class diagram. In Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on (pp. 54-63). IEEE.
[30] **Baudry, B., Traon, Y. L., Sunyé, G., Jézéquel, J. M.** (2003). Measuring and improving design patterns testability. In Software Metrics Symposium, 2003. Proceedings. Ninth International (pp. 50-59). IEEE.
[31] **Jungmayr, S. (2002).** Identifying test-critical dependencies. In Software Maintenance, 2002. Proceedings. International Conference on (pp. 404-413). IEEE.
[32] **Osherove, R.** (2009). *The art of unit testing: with examples in .NET*. Manning Publications, Greenwich.

[33] **Martin, C. R. ve Martin, M.** (2006). *Agile Principles, Patterns and Practices in C# 1st ed.*, Pearson Education.
[34] **Test-driven development**. (t.y.). Wikipedia. Alındığı tarih: 17.04.2015, adres: http://en.wikipedia.org/wiki/Test-driven_development.
[35] **Scientific Toolworks Inc.** <https://scitools.com>, alındığı tarih: 17.04.2015.
[36] **Söylemez, M. T.** (2010). Raylı Sistemlerde Fonksiyonel Güvenlik Uygulamaları: Ulusal Demiryolu Sinyalizasyon Projesi. *2.* Uluslararası Endüstriyel Güvenlik Sistemleri Konferansı.
[37] **Qt.** <http://www.qt.io>, alındığı tarih: 17.04.2015.

ÖZGEÇMİŞ

Ad Soyad **: Onur ÖZÇELİK**

Doğum Yeri ve Tarihi **: Kırklareli/1985**

E-Posta **: onur.ozcelik@tubitak.gov.tr**

ÖĞRENİM DURUMU:

- **Lisans** **:** 2007, Gebze Teknik Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği

TEZDEN TÜRETİLEN YAYINLAR, SUNUMLAR VE PATENTLER:

- **Altilar, T. D. ve Özçelik, O.** (2015). Test-driven approach for Safety-critical Sofware Development, 2015 International Conference on Computer and Information Technology (ICCIT 2015), Accepted