T.C.
SÜLEYMAN DEMİREL ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GERÇEK ZAMANLI VERİ TOPLAYARAK OTOMATİK BİRİM TEST OLUŞTURMA YAZILIMI

Sevdanur GENÇ

Danışman
Prof. Dr. Ecir Uğur KÜÇÜKSİLLE

DOKTORA TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
ISPARTA - 2022

İÇİNDEKİLER **Sayfa**

ÖZET

Doktora Tezi

GERÇEK ZAMANLI VERİ TOPLAYARAK OTOMATİK BİRİM TEST OLUŞTURMA YAZILIMI

Sevdanur GENÇ

Süleyman Demirel Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Prof. Dr. Ecir Uğur KÜÇÜKSİLLE

Yazılımda kalite ve verimlilik ihtiyaçları bir arada düşünülmektedir. Bu sebeple, var olan bir yazılımın bu kriterlere uyguluğunun test edilebilmesi için bazı faktörlere ihtiyaç duyulmaktadır. Bu faktörleri yerine getirebilmek için test alanına yüklenen ciddi sorumluluklar bulunmaktadır. Testlerin hem hızlı bir şekilde yapılması hem de sonuçlarının verimli olması yazılım geliştirmede fark yaratan bir faktördür. Bu faktörü gerçekleştirebilmek için ihtiyaç duyulan bir yazılım test otomasyonudur. Yazılım test otomasyonları, yazılım test aktivitelerinin otomatikleştirilmiş halidir. Sürekli manuel yazılan testlerin otomatik hale getirilmesi ise zaman açısından tasarruf edilmesi, hata oranlarının azaltılması, daha kaliteli bir yazılım ortaya çıkması ve maliyetin düşürülmesi gibi sonuçları ortaya çıkarmaktadır.

Bu çalışmada, ürünler üzerinde gerçek zamanlı bir şekilde çalışması planlanan birim testlerin otomatik olarak üretilmesi amaçlanmıştır. Geliştirilen bu uygulama, istenilen ürün üzerinde birim test dönüşümlerini gerçekleştirmektedir. Tüm bu dönüşümlerin temelinde java ajanları (java agent) kullanılmıştır. Üzerinde çalışılacak olan örnek java sınıflarının nesne, metot ve değişkenlerine ait tüm bilgiler byte code kullanılarak gerçek zamanlı bir şekilde veri haline dönüştürülmektedir. Bu dönüşüm esnasında, bilgiler hem veri tabanına kaydedilmekte hem de şablon motor aracılığıyla birim testleri otomatik bir şekilde oluşturulmaktadır. Literatürde otomatik birim test üretimi üzerine geliştirilen ürünlere kıyasla, bu çalışmada opcode ayrıştırma yöntemi geliştirilmiştir. Bu yöntem, bir byte code'u çalışma zamanında okuyarak, ait olduğu java sınıfının özelliklerini kullanır ve otomatik bir şekilde birim test sınıfını ve test metotlarını oluşturur. Çalışma aynı zamanda, bir metot içerisindeki farklı nesne tanımlamaları ile şart ve döngü yapılarını inceleyip, alternatif test senaryolarını üretebilmektedir. Üretilen otomatik birim test senaryosu çalışma zamanında en az hatayla karşılaşabilecek esnek bir çerçeve haline getirilmiştir.

Ulusal yazılım testi alanında yapılan çalışmaların azlığı göz önüne alındığında; bu çalışma kapsamında geliştirilen, byte code kullanılarak otomatik birim test üretim ürününün çalışma alanına ciddi katkıda bulunacağı düşünülmüktedir.

Anahtar Kelimeler: Birim Test Üretimi, Java Agent, Byte Code, Yazılım Testi.
2022, 120 sayfa

ABSTRACT

Ph.D. Thesis

AUTOMATIC UNIT TEST GENERATOR SOFTWARE BY COLLECTING REAL-TIME DATA

Sevdanur GENÇ

Süleyman Demirel University
Graduate School of Natural and Applied Sciences
Department of Computer Engineering

Supervisor: Prof. Dr. Ecir Uğur KÜÇÜKSİLLE

Quality and productivity needs are considered together in software. For this reason, some factors are needed in order to test the compliance of an existing software with these criteria. In order to fulfill these factors, there are serious responsibilities imposed on the test area. The fact that the tests are carried out quickly and the results are efficient is a factor that makes a difference in software development. A software test automation is needed to realize this factor. Software test automations are automated software testing activities. Automating constantly manually written tests, on the other hand, saves time, reduces error rates, produces better quality software and reduces costs.

In this study, it is aimed to automatically produce unit tests that are planned to work on products in real time. This developed application performs unit test transformations on the desired product. Java agents are used as the basis of all these transformations. All information about the objects, methods and variables of the sample java classes to be worked on is converted into data in real time using byte code. During this transformation, information is both saved in the database and unit tests are created automatically through the template engine. Compared to the products developed on automatic unit test generation in the literature, the opcode parsing method was developed in this study. This method reads a byte code at runtime, uses the properties of the java class it belongs to, and automatically creates the unit test class and test methods. The study can also examine different object definitions and conditional and loop structures within a method and produce alternative test scenarios. The automatic unit test scenario produced has been turned into a flexible framework that can encounter minimum errors at runtime.

Considering the scarcity of studies in the field of national software testing; It is thought that the automatic unit test generation product developed within the scope of this study, using byte code, will make a serious contribution to the field of study.

Keywords: Unit Test Generation, Java Agent, Byte Code, Software Testing.
2022, 120 pages

TEŞEKKÜR

Bu araştırma için beni yönlendiren, karşılaştığım zorlukları bilgi ve tecrübesi ile aşmamda yardımcı olan değerli Danışman Hocam Prof. Dr. Ecir Uğur KÜÇÜKSİLLE 'ye teşekkürlerimi sunarım.

Tezimin ilk aşamalarında ilham veren yönlendirmeleriyle desteklerini esirgemeyen Melih SAKARYA ve Testinium Şirketine teşekkür ederim.

Tezimin her aşamasında gösterdikleri anlayış ve desteklerinden dolayı değerli annem ve abime sonsuz sevgi ve saygılarımı sunarım.

Sevdanur GENÇ
ISPARTA, 2022

ŞEKİLLER DİZİNİ

Sayfa

ÇİZELGELER DİZİNİ

Sayfa

SİMGELER VE KISALTMALAR DİZİNİ

Bytecode	Bayt kod
Java Agents	Java Ajanları
JUnit	Java Unit test framewwork'ü
JVM	Java Virtual Machine, Java Sanal Makinesi
LIFO	Last in first out – Son giren ilk çıkar
NoSql	Not Only SQL
Operand	İşlenen
POJO	Plain Old Java Object
Run-Time	Yürütme zamanı
Test case	Test senaryoları

1. GİRİŞ

Sun Microsystems tarafından geliştirilen ve 1995 yılında kullanıma sunulan Java programlama dili tasarlanırken esas amaç; taşınabilir, kolay öğrenilen, genel amaçlı, platform bağımsız bir nesneye dayalı programlama dili geliştirmek olmuştur. Java derleyicisi, kaynak kodları platform bağımsız bir ara dil olan java bayt koduna (java byte code) dönüştürür. Bu kod, daha sonra her platformda bulunan java sanal makinesi (jvm - java virtual machine) aracılığıyla işlenerek, çalıştırılır. Kodun bu çalışma zamanında JVM tarafından yürütülen uygulama mantığını esnek bir şekilde değiştirmek için java ajanları (java agents) kullanılır. Bir java ajanı, özel hazırlanmış bir jar dosyasıdır. Bu dosya, JVM'de yüklü olan mevcut bayt kodunu değiştirmek için Instrumentation API'yi kullanmaktadır. Java ajan araçlarının en önemli özelliği, çalışma zamanında sınıfları yeniden tanımlayabilmeleri veya değiştirebilmeleridir. Metot gövdelerini, sabit ve değişken özelliklerini yeniden tanımlayarak değiştirebilirler. Aynı zamanda, metotların imzalarını veya kalıtım özelliklerini de değiştirebilmektedirler.

Yazılımda kalite ve verimlilik ihtiyaçları bir arada düşünülmektedir. Bu sebeple, var olan bir yazılımın bu kriterlere uyguluğunun test edilebilmesi için akıcı bir algoritma ve güçlü bir risk yönetimi gibi faktörlere ihtiyaç duyulmaktadır. Bu faktörleri yerine getirebilmek için test alanına yüklenilen ciddi sorumluluklar bulunmaktadır. Testlerin hem hızlı bir şekilde yapılması hem de sonuçların doğruluk oranının yüksek olması yazılım geliştirmede fark yaratan bir faktördür. Bu faktörü gerçekleştirebilmek için yazılım test otomasyonuna ihtiyaç duyulmaktadır.

Yazılım üzerine yapılmış olan projelerin odak noktası yazılım test süreçledir. Başarılı bir test süreci sonunda en az hataya sahip yüksek doğrulukta yazılımlar üretilir. Türkiye'de yazılım kalitesi hakkında yapılan araştırmalar, ülkemizde test güdümlü yazılım geliştirme sürecine ve test araçlarına olan ihtiyacın arttığını göstermektedir. Test güdümlü yazılım geliştirmedeki hedef, gerekli işi yapacak kodu yazmadan önce bu koda ait olan senaryoyla öncelikle test edilebilir bir kodun yazılması gerekmektedir. Test edilebilir bu kod, çeşitli yazılım geliştirme prensiplerine uygun olacak şekilde tasarlandıktan sonra doğruluğu yüksek olan bir sonuç verirse o

yazılım testi başarılı bir şekilde geçmiştir. Eğer yapılan test sonuçları başarısız ise tekrar başa dönülür, kod incelenerek problem düzeltilmeye çalışılır. Yazılım projelerinde her bir birimin (sınıf ya da metot) hatasız bir şekilde çalıştığını kanıtlamak için birim test (unit test) adı verilen testler yazılmaktadır. Birim testler yazılım geliştirme sürecini kolaylaştırır, hızlandırır ve her bir sınıfın, metodun doğru çalıştığını garantilemeyi sağlar.

Yazılım test otomasyonları yazılım test aktivitelerinin otomatikleştirilmiş halidir. İhtiyaç sahipleri her defasında üründeki değişiklikleri kontrol etmek zorunda kalırlar. Bunu kolaylaştırmak adına, ürünün belli fonksiyon ve özelliklerini otomatize haline getirmek gerekir. Tüm bu işleri hafifletmek için ürün üzerinde otomatik testler kullanılması tavsiye edilmektedir. Sürekli manuel yazılan testlerin otomatik hale getirilmesi ise zaman açısından tasarruf edilmesi, hata oranlarının azaltılması, daha kaliteli bir yazılım ortaya çıkması ve maliyetin düşürülmesi gibi sonuçları ortaya çıkarmaktadır. Otomatik testler; birim testleri, uyum testleri, fonksiyonel testler ve performans testleri olmak üzere dört gruba ayrılmaktadır.

Java platformlarında en çok bilinen temel iki test çerçevesi bulunmaktadır. Bunlar JUnit ve TestNG'dir. İkisi de karmaşık test senaryolarında tam olarak istenilen kod parçacıklarında test oluşturmaya izin verebilecek kadar güçlü çerçevelerdir. Junit, tekrarlanabilir testleri yazmak ve çalıştırmak için kullanılan açık kaynak kodlu bir çerçevedir. JUnit; programdan beklenen sonuçları test etmek için çeşitli test senaryolarıyla birlikte test verilerini çalıştırır. TestNG, Junit'e göre daha işlevseldir ve kullanımı kolaydır. TestNG, testlerin kendine has Thread'lerde test senaryolarının (test case) paralel olarak çalıştırılabilmesini desteklemektedir. Aynı zamanda, esnek test konfigürasyonları, hata mesajlarının detaylı analizleri, gelişmiş arşivleme ve editörler için eklenti desteği gibi birçok özelliğe de sahiptir. Tüm bu birim testler yazılımcı tarafından manuel bir şekilde kodlanmaktadır. Zamanı kaliteli bir şekilde kullanmak ve iş trafiğinin hızlanması için testlerin otomatikleştirilmesi önerilmektedir.

Gerçekleştirilen bu çalışmada, ürünler üzerinde gerçek zamanlı bir şekilde çalışması planlanlan birim testlerin otomatik olarak üretilmesi amaçlanmaktadır. Üretilen bu yazılım, bir masaüstü uygulama olarak geliştirilmiş ve belirtilen java sınıfları

üzerinde birim test dönüşümlerini gerçekleştirebilecektir. Tüm bu dönüşümlerin temelinde java ajanları (java agent) ve bytecode (bayt kod) kullanılmıştır. Üzerinde çalışılacak olan örnek java sınıflarının nesne, metot ve değişkenlerine ait tüm bilgiler gerçek zamanlı bir şekilde veri haline dönüştürülmüştür. Bu dönüşüm esnasında, bilgiler hem veri tabanına kaydedilmekte hem de şablon motor aracılığıyla birim testleri otomatik bir şekilde oluşturulmaktadır. Ancak geliştirilen bu çalışmada, hem bayt kod tarafı geliştirilmiş opcode ayrıştırma yöntemiyle analiz edilmekte hem de istenilen birimler üzerinde testler üretilmektedir. Aynı zamanda sınıflar içerisinde kullanılan nesnelerin her bir farklı kullanımına ait alternatif senaryolar da üretilmiştir. Bu senaryolar testin uygulandığı anda çalışarak otomatik birim test üretimini gerçekleştirebilmektedir. Her bir nesnenin kullanım farklılığına uygun olacak şekilde birim testler üretebilen bu otomatik birim test üretim aracı çalışma zamanında en az hatayla karşılaşabilecek esnek bir çerçeve haline getirilmiştir.

Literatürde otomatik birim testi üretimi için çeşitli yaklaşımlar sunulmuştur. Yapılan çalışmanın literatüre katkıları şu şekildedir;

1. Bayt kod dönüşümleri sırasında çalışma için java string fonksiyonlarından yardım alınarak bir opcode ayrıştırma yöntemi geliştirilmiştir. Bu yöntemle, her bir opcode karşısındaki nesne, değişken ve giriş-çıkış parametreleri gibi değerler ayırt edilmiş ve bu veriler JSON formatında listelenmiştir. Geliştirilen opcode ayrıştırma yöntemi ileride farklı ihtiyaçlar doğrultusunda geliştirilmeye açıktır. Literatürdeki çalışmalarda ise bu işlemlere benzer şekilde bayt kod API'sinin kısıtlı hazır fonksiyonları tasarlanmaktadır.
2. Bir java sınıfında bulunan her bir nesne, değişken ve giriş-çıkış parametrelerine ait değerlerin oluşturulacak birim testlerde tekrar kullanılabilmesi için ya da bu değerlere benzer rastgele değerler atanabilmesi için JSON formatında NoSql veritabanına ait koleksiyonlar kullanılarak kaydedilmiştir. Bu veriler aynı zamanda sistemde bir arşiv dosyasında da saklanılmaktadır. Literatürdeki çalışmalarda ise, XML ve oracle gibi farklı veri saklama ortamları kullanılmıştır.
3. Test senaryoları oluşturulurken bazı kurallara dikkat edilmelidir. Bunlardan ilki, metot içerisinde kullanılan koşul ve döngü yapıları gibi alternatif durumlardır. Diğeri ise, metot içerisinde kullanılan farklı nesneler ile bunlara

ait değer dönüşümleri için mock-stub ayrımı yapılması gereken aşamadır. Bunlar için geliştirilen masaüstü uygulaması aracılığıyla kullanıcıdan test metodolojisi seçmesi istenir. Bu seçim ile birlikte, istenen senaryoya göre geliştirilen çerçeve yapısında birim testler otomatik olarak oluşturulur. Aynı zamanda, Mock-Stub ayrımı için geliştirilen opcode ayrıştırma yöntemi otomatik birim testi üretim yazılımına yol göstermektedir.

4. JUnit standartlarında oluşturulacak her bir birim testi için gerekli assertion yapısı FTL şablon motoru kullanılarak hazırlanmıştır. Uygulama, çalışma zamanında otomatik bir şekilde birim testleri hazırlarken, annotation'lar kullanıcı arayüzü tarafında seçilmiş olan test metodolojisine göre oluşturulmaktadır. Literatürde otomatik birim testi üretim ürünlerinin hiçbirinde FTL şablon motorunun kullanımına rastlanmamıştır. Bunun yerine farkı yöntemler geliştirilmiştir.

2. KAYNAK ÖZETLERİ

Literatürde yer alan, son 20 yılda yayınlanmış, yazılım test uygulamalarında birim test üretimi hakkında gerçekleştirilen farklı çalışmalardan başlıcaları incelenmiştir.

Csallner vd. (2004) çalışmalarında, JCrasher isminde bir otomatik test üretme aracı geliştirmişlerdir. Bu araç, Eclipse IDE ile entegre bir şekilde çalışabilmektedir. Test üretmesi için araca verilmiş olan örnek java sınıfının bilgileri incelendikten sonra rastgele verilerle test edilmektedir. Test işlemini Junit ile gerçekleştirmişlerdir. Araç aynı zamanda test aşamasında oluşan hataları tespit edebilmektedir.

Xie vd. (2006) çalışmalarında, önsel spefikasyonlar gerektirmeyen bir kara kutu yaklaşımı olan birim test üretimi ve seçimi için operasyonel ihlal yaklaşımı sunmuşlardır. Bu yaklaşımla, mevcut birim test kodlarının çalıştırılmasından dinamik olarak operasyonel sonuçlar üretilmektedir. Bu operasyonel sonuçlar, test oluşturma araçlarına, bunları ihlal edecek testleri oluşturmak için rehberlik etmektedir.

Conroy vd. (2007) çalışmalarında, web servislerini test etmek için grafik kullanıcı arayüzü tabanlı otomatik test oluşturma uygulaması geliştirmişlerdir. Bu uygulama görsel programlama ile hazırlanmış olup, kullanıcı kullanılabilirliğini arttırmıştır. Bu yaklaşımla, test personelini tarafından dosyaları işaretleme, tıklama, sürükle ve bırak gibi gerçekleştirilen işlemlerde kolaylık sağlamıştır.

Pacheco vd. (2007) çalışmalarında, Randoop isminde JUnit kullanarak otomatik bir birim test üretim aracı geliştirmişlerdir. Bu araç, nesne yönelimli programlar için geri besleme odaklı birim test oluşturabilmektedir. Aynı zamanda oluşan hataları yakalayıp arşivleyebilmektedir. Rastgele test verilerini üretebilmek ve test sonuçlarını birleştirebilmek için geliştirilmiş bir araçtır.

Simons vd. (2007) çalışmalarında, JWalk isminde java için bir birim test aracı geliştirmişlerdir. Çalışma iki temel aşamadan oluşmaktadır. Öncelikle örnek bir sınıfın gelişmiş özellikleri ortaya çıkarılmakta ardından birim testler sistematik bir şekilde uygulanmaktadır. Bunun sonucu olarak, java test sınıflarının durumları

hakkında bilgi verebilmektedir. Aynı zamanda, JUnit gibi uzman birim test uygulamalarıyla da karşılaştırılmıştır.

Sen (2007) çalışmasında, Cute isminde java ortamında bir uygulama geliştirmiştir. Uygulama C programlama dili hedef alınarak, c dilinde yazılan kodların testi için geliştirilmiştir. Bu uygulama, otomatik ve rastgele test mantıklarını birleştirerek çalışmaktadır. Ayırt edici girdi ve kısıtlayıcı çözümlerin üstesinden gelmeye yardımcı olan sembolik kod çalıştırma işlemlerini kullanmaktadır. Bu aracın, sistem çağrılarını analiz edememe ve doğrusal olmayan tamsayı denklemlerini çözememe gibi, yetersiz olduğu durumlar bulunmaktadır.

Charreteur vd. (2010) çalışmalarında, java bayt kodu programları için kısıtlamaya dayalı bir akıl yürütme yaklaşımı kullanarak otomatik test girdisi elde etmişlerdir. Yöntem, her bayt kodu için bayt kodu programının geriye doğru araştırılmasına izin veren ve bellek şekli üzerindeki karmaşık kısıtlamaları çözmeye izin veren bir kısıtlama modeli olarak geliştirilmiştir. JAUT ismini verdikleri bu çalışma, Cute, JTEST ve PEX gibi çalışmalara emsaldir.

Albert vd. (2010) çalışmalarında, PET isminde kısmi değerlendirme yapabilen bir test aracı geliştirmişlerdir. Bu araç, isteğe bağlı parametreler kümesi olarak girdiler almakta ve iki kısmi değerlendirme işlemi gerçekleştirmektedir. Ardaşık iki kısmi değerlendirme (PE) aracılığıyla beyaz kutu testi üretimi gerçekleştirildi. İlk kısım PE, java bayt kodu programını eşdeğer bir CLP'ye (kısıtlı mantık programlaması) dönüştürür. İkinci kısım PE, CLP'yle bir test kodu oluşturur. Bu araç hem daha büyük programlarda hem de web sitelerinde serbestçe kullanılabilmektedir.

Fraser vd. (2010) çalışmalarında, EvoSuite isminde bir test üretim aracı geliştirmişlerdir. Java dilinde yazılımış olan bu test üretme aracı kapsamlı özelliklere sahiptir. Bu araç ile yapılan tüm testler istenen kriterlerle karşılaştırılır. Karşılaştırma sonucu analiz ve optimizasyon işlemleri gerçekleştirilir.

de Andrade vd. (2012) çalışmalarında, genel sınıflar tarafından tanımlanan soyut veri türlerine ait (Abstract Data Type - ADT) uygulamalarının test edilmesi için bir

yaklaşım sunmuşlardır. Önerilen teknik, her bir test hedefi için model örnekleri bulmaya yarayan Alloy Analyzer uygulaması kullanılmıştır.
Păsăreanu vd. (2013) çalışmalarında, Symbolic Path Finder (SPF) isimli açık kaynak kodlu otomatik test üretme aracını geliştirmişlerdir. SPF genellikle kullanıcı tarafından belirlenmiş sembolik girdileri sistematik test senaryolarına dönüştürebilmektedir. Ayrıca, eş zamanlı olarak hataları kontrol edebilmektedir.

Sakti vd. (2014) çalışmalarında, JTExpert isminde java programlamada kullanılabilecek bir otomatik test üretim aracı geliştirmişlerdir. Nesne yönelimli programlama için test verisi oluşturma, kaynak kodun bir kısmına doğrudan erişim sağlayamayan soyutlama ve kapsülleme gibi özelliklerden dolayı zordur. Bu sebepten dolayı, tüm bu yüksek özellikli kod parçaları için test senaryoları oluşturma probleminin üstesinden gelebilmektedir. Çalıştırılabilir bir jar dosyası olan JTExpert aracı, bir java dosyasını veya java proje dizinini girdi olarak alır ve test edilen her java sınıfı için otomatik olarak JUnit biçiminde bir test verisi paketi oluşturur.

Tanno vd. (2015) çalışmalarında, CATG isminde karma bir birim test aracı geliştirmişlerdir. Sembolik ve somut durumda bulunan girdileri, dinamik bir şekilde yürütmeyi gerçekleştiren konkolik test isminde bir kavramı kullanmışlardır. Bu kavramla birlikte, somut durumdaki tüm değişkenleri eşzamanlı olmayan bir yöntemle somut değerlerle eşleştirmişlerdir. CATG isminde bu çalışma TesMa yazılımı ile entegre olduğunda, veri tabanı tanımlarını, süreç akış şemalarını, ekran görüntülerini ve tasarım dökümanlarını otomatik olarak test senaryolarına çeviren bir test aracı haline gelmiştir. TesMa yazılımı ise bu çıktılarla java programlarını oluşturabilmektedir. Özetle, CATG uygulamasıyla java programları için konkolik testi gerçekleştirir ve veri tabanı oluşturacak şekilde gerekli test senaryolarını oluşturur.

Lei vd. (2015) çalışmalarında, Kılavuzlu Rastgele Test yani GRT isminde java için otomatik bir test üretme aracı geliştirmişlerdir. GRT statik ve dinamik program analizleri ile birlikte çalışma zamanında test üretimi yapabilmektedir. Bununla birlikte geribildirim odaklı rastgele test kullanabilmektedir. GRT karmaşık algoritmalar karşısında daha yüksek kod kapsamı elde etmek için sembolik kod yürütme tekniklerine başvurarak çeşitli şekillerde kendisini geliştirebilmektedir.

Prasetya (2015) çalışmasında, T3i isminde otomatik birim test oluşturan bir araç geliştirmiştir. Rastgele girdi olarak verilen java sınıfları için yöntem çağrılarını analiz edebilmektedir. Bu araç aynı zamanda beklenmeyen hataları da tespit edebilmektedir. Komut satırında çalışabilen bu araç oldukça hızlıdır ve birkaç saniye içerisinde binlerce test dizisi oluşturabilmektedir.

Yoshida vd. (2017) çalışmalarında, C veya C++'da uygulanan endüstriyel güçlü gömülü yazılımların test edilmesini otomatikleştirmek için bir metodoloji geliştirmişlerdir. Metodolojinin çekirdeği, araştırmacıların testi otomatikleştirmek için özelleştirdiği, sembolik yürütme adı verilen bir program analiz tekniğidir. Metodoloji, mükemmel test kapsamı sağlarken test oluşturma süresini ve maliyetini büyük ölçüde azaltan birim düzeyinde testler üretmiştir.

Tufano vd. (2020) çalışmalarında, gerçek dünyadan geliştirici tarafından yazılmış test senaryolarından öğrenerek birim test senaryoları üretmeyi amaçlayan bir yaklaşım olan AthenaTest'i önermişlerdir. Yaklaşım, dentlenemeyen bir java programının ön eğitimden oluşan iki aşamalı bir eğitim prosedürünü benimseyerek, diziden diziye öğrenme görevi olarak formüle edilir. Doğal dil işlemenin yanı sıra bir transformatör modeli geliştirilmiştir.

3. MATERYAL VE YÖNTEM

3.1. Yazılım Testi

Gelişen dünya çerçevesinde, insanlığın ihtiyaç duyduğu birçok nesne için bilgisayar komutlarından varolmuş yazılım uygulamaları bulunmaktadır. Sadece yazılım olarak tabir edilen bu uygulamalar, aslında çevremizdeki her türlü soruna çözüm olarak üretilmişlerdir. En dar kapsamlı, basit birim otomasyonlarından en geniş kapsamlı yapay zeka ile geliştirilmiş karmaşık sistemlere kadar tek amaçlanan şey aslında problem çözmektir. İnsanlığın hayatını kolaylaştıran ve hatta onları tembelliğe sürükleyen, aslında birçok görevi kendi üstlenen bu yazılım sistemlerinin üretimi aşama aşama gerçekleştirilmektedir. Yazılımı aşama aşama geliştirme mantığı esasen algoritmanın ne anlama geldiğini çağrıştırmaktadır. Algoritmaların verimli bir şekilde çalışabilmesi için bir sonraki adıma geçiş evresi çok önemlidir. Bir sonraki adıma geçiş evresi için de o anki adımın doğru bir şekilde çalıştırılmış olması gerekir. İşte bu noktada doğruluk kelimesinin anlamı, test edilebilirlilik ifadesinin anlamıyla örtüşmeye başlamaktadır.

Yazılım testi, geliştirilmesi planlanan bir yazılımın mantık hatalarını bulduktan sonra yazılımın kaliteli bir şekilde çalışmasına olanak sağlayan bir yöntemdir. Bu yöntem, yazılım kalite faktörlerini temel alarak geliştirilmektedir. Bu kalite faktörleri; yazılımın bütünlüğü, uygunluğu, yeterliliği, verimliliği ve kullanılabilirliği olarak nitelendirilmektedir. Bu faktörlerin yanı sıra, test geliştirmenin temeli ve ilkeleri bulunmaktadır. Bu ilkeler doğrultusunda test süreçleri başlamaktadır ve sonrasında alınan sonuçlarla analiz edilebilmektedir. Bu bölüm başlığı altında yazılım testi ile ilgili temel alt konu başlıkları verilmiştir.

3.1.1. Yazılım geliştirme sürecinde test

Yazılım testleri, bir yazılım ürününün geliştirilmesi için önemli bir unsurdur. Yazılım ürününün, bu ürünle muhattap olacak kişilerin tüm beklentilerine cevap verebilecek bir kalitede tasarlanmış olması gerekir. Ancak, her zaman için yüzde yüz bir verim beklenilmemektedir. Bu tasarımın kusursuz olabilmesi için ürünün her

aşamasında kaliteyi arttıracak ve tüm olasılıklara cevap verebilecek yöntemler uygulanır. Bu yöntemlere yazılım testi denilmektedir ve her bir test doğru sonuçlarla ilerleyerek yazılımın performanslı bir şekilde çalışabilmesi için bir bütüne ulaşmaktadır.

Bir yazılımın kullanılabilirliğini arttırmak için yazılım test kodlarında uyulması gereken bazı kurallar bulunmaktadır. Bu kurallar, yapılacak testler hakkında konuya hakim olan çalışanlar tarafından uygulanmaktadır. Uygun bir test metodu seçilmeli, en fazla hata sayısını elde etme amacıyla planlanmalı ve tüm olasılıklara yanıt vermelidir. Test yapılırken yazılım ürünü üzerinden herhangi bir değişiklik yapılmamalıdır ve testin her aşamasında planlı ve güncel bir şekilde ilerlenilmelidir. Test sonucunda, her bir testin durumunu ve bu durumlara ait beklenen sonuçlarını içeren bir rapor hazırlanılmadır (Mustafa, 2007).

Söz konusu her bir test beklenen ve gerçekleşen durumlar arasındaki farklılık ya da benzerliklere bakılarak analiz edilmektedir. Benzerlik oranı yüksek veya tam ise test edilen kodun doğru çalışıyor olma ihtimali çok yüksektir. Tersi durumda, farklılık oranı yüksek veya tam ise test edilen kodun hatalı çalışıyor olma ihtimali çok yüksektir.

3.1.2. Yazılım testinin temelleri ve ilkeleri

Öncelikle her bir test, kod parçasında bir hatanın var oluşunu kabul ederek çalıştırılmalıdır. Herhangi bir hatanın var olmaması demek herşeyin yolunda gitttiği anlamına gelmez. Tüm test olasılıklarının ve durumlarının test edilebilir olması imkansızdır. Bu çok fazla zaman ve emeğe sebep olmak demektir. Yazılım ürününün geliştirilmesi sırasında olabildiğince ürünün erkenden test edilmeye başlanması gerekir. Ne kadar erken hatalar tespit edilip düzeltilirse o kadar kaliteli bir ürün ortaya çıkacaktır. Sonuç itibariyle, düzeltilen her bir birim ile yola devam edildiğinde entegre edilmiş tüm birimlerin birlikte çalışma uyumları önemlidir. Herhangi bir yazılım ürünü veya kod parçası farklı geliştirilebileceği gibi bunlara ait test senaryoları da farklı geliştirilmektedir. Test senaryolarının ve durumlarının içeriklerinin birbirleriyle bağlantılı olması gerekir. Sürekli aynı senaryo ile aynı ürün

üzerinde çalışılırsa verimli sonuç alınamaz. Bunun yerine daha güncel test senaryoları kullanılırsa yeni hataların bulunması da kolaylaşacaktır. Yazılımda hata yok ifadesi tamamen bir yanılgıdır. Bu yanılgı, o ürünün kaliteli bir yazılım olduğu anlamına gelmez.

3.1.3. Yazılım test süreçleri

Bir yazılıma ait test senaryolarının gerçekleştirilmesi için gerekli olan bazı aşamalar bulunmaktadır. Bunlar testin ilgili senaryolarının hazırlanması için testin planlanması, ilgili test senaryolarının kontrolü, test senaryolarından sonra çıkan sonuçların rapor ve analizleri, yazılım testinin tasarımı, testin uygulanması ve değerlendirilmesi ile ilgili aşamalardır. Tüm bu aşamalar birbirini takip eden süreçlerde veya eş zamanlı olarak gerçekleştirilmektedir.

3.1.3.1. Testin planlanması ve kontrolünün sağlanması

Bir testi oluşturmadan önce o testin hangi amaçla yazılması gerektiği planlanmalıdır. Amacı karşılayan uygun test senaryolarının ve durumlarının süreç içerisinde planlı bir şekilde koşturulması (çalıştırılması) gerekmektedir. Bu nedenle, tüm bu süreç içerisinde testin kontrolünün sağlanması projenin tamamlanması için gerekli bir aktivitedir.

3.1.3.2. Testin analizi ve tasarımı

Belli bir amaca yönelik planlanan test senaryoları ve durumlarının somut bir şekilde koda dönüştürülmesi ve belli bir sırayla bu kodların koşturulması durumudur. Tüm bu süreci gerçekleştirirken dikkat edilmesi gereken önemli adımlar bulunmaktadır. Bunlar; test kaynaklarının gözden geçirilmesi, kaynakların test edilebilirliğinin değerlendirilmesi, senaryoların belirlenmesi ve önceliklendirilmesi, test verilerinin belirlenmesi, test araçlarının belirlenmesi, test ortamının tasarlanması ve tüm bu sürecin izlenebilirliliğinin sağlanmasıdır (ISTQB, 2011).

3.1.3.3. Testin uygulanması ve testin koşturulması

Hazırlanmış olan her bir test senaryosu, seçilmiş olan test araçları vasıtasıyla arka planda bulunan yazılım üzerinde koşturulur. Bu koşturulma esnasında, test verilerinin doğru kullanılması, senaryoların doğruluğu ve beklenen sonuçlarla elde edilen sonuçların karşılaştırılması gibi durumlara dikkat edilir. Kısacası, test koşturma anında tüm prosedürlerin doğru bir şekilde gerçekleşmesi için özen gösterilir. Bu aşamada, yapılan testlerle ilgili bilgilerin kayıt altına alınması gereklidir. Alınan her bir kayıt aslında bir rapordur. Bu raporlarda testlerin sonuçları, çalışıp çalışmadığı, hangi beklenen ve gerçekleşen verilerde testin reaksiyon verdiği takip edilmektedir.

3.1.3.4. Testin değerlendirilmesi, raporlandırılması ve test sonlandırma işlemleri

Arka planda çalışan yazılım üzerinde koşturulan testler tamamlandıktan sonra, yapılan işlemlerin yeterli olup olmadığı hakkında bir kontrol yapılır. Bu aşamada, daha fazla test eklenilmesi veya fazla testlerin silinmesi ve test kriterlerinde yapılacak herhangi bir değişikliğe karar verilebilir. Test sonucunda tüm kriterlerin doğru olduğundan emin olunduktan sonra test özet raporları alınabilir. Bu raporlar, her bir test gerçekleştikten sonra sisteme otomatik kayıt alınan bilgilerden yola çıkılarak oluşturulmaktadır.

3.2. Yazılım Test Düzeyleri

Kaliteli bir yazılım testi için prosedürlerin kendi karakteristik özelliklerinin olması gerekir. Her bir testin amacı farklı olduğu gibi, uygulanacak prosedürlerin de farkı olması beklenmektedir. Yazılım ürününün geliştirilmesi sırasında, test analiz ve tasarımının da eş zamanlı olarak yapılması gerekir. Bu noktada, yazılabilecek her bir testin amacına yönelik test yöntemleri bulunmaktadır. Bu yöntemler aracılığıyla, testlerin karakteristik davranışları belirlenebilir.

3.2.1. Birim testi

Birim test; bileşen, alt küme, modül ya da program testi olarak düşünülebilir. İsminden de anlaşılacağı üzerine her türlü test edilebilir algoritmalarda, bu algoritmalara sahip olan metot ve sınıflarda test etmek adına çalışan modüller yazılmaktadır.

Birim testlerinde kullanılan test senaryoları öncelikle geliştirilir ve bu senaryoların sonucuna göre ürün yazılımı kodlanır. Kullanım amacı test sonuçlarında doğruları ve hataları aramaktır. Test kodunu geliştirenler tarafından bulunan tüm hatalar eğer çalışma anında düzeltebilme imkanları varsa düzeltebilirler eğer yoksa bununla ilgili birimlere test sonuçlarının raporlarını yönlendirerek yardımcı olabilirler. Doğruluğu kabul edilen her bir birime ait teste ürün yazılımının kodu yazılarak devam edilir ve böylelikle her bir modül tamamlandıktan sonra her modül entegre olarak ürün tamamlanmış olur.

3.2.2. Entegrasyon testi

Birbirinden bağımsız olarak test edilmiş yazılım modüllerinin birbirine bağlanması ve birbirleri arasında veri aktarımının sağlanmasını kontrol eden bir testtir. Çalışma mantığı birim teste benzer. Birim test her bir birimi test ederken, entegrasyon testi birimlerin beraber çalışmasını test eder. Modüller arasında bir birliktelik söz konusudur. Bu testin amacı, entegrasyona uğrayan modüllerde olası hataları yakalamak ve sorunu gidererek entegrasyonu sorunsuz bir şekilde sağlamaktır.

Dört çeşit entegrasyon testi vadır (Solheim & Rowland, 1993), (Gökalp, 2015).

I. Big Bang Entegrasyon Testi: Birleştirilecek tüm modüllerin testleri bir arada kabul edilerek test gerçekleştirilir.
II. Top – Down Entegrasyon Testi: Birleştirilen modüller bir bütün olarak yukarıdan aşağıya doğru hiyerarşik bir şekilde ele alınır. Sebebi, her bir modül kendi içerisinde test edilirken kesinlikle hatasız olmalıdır. İki

hatasız modül birleştirildiğinde sonuç doğru olacaktır ve olası bir hatayla karşılaşılmayacaktır.

III. Bottom – Up Entegrasyon Testi: Tüm modüller birimler halinde test edilmektedir. bu sebepten dolayı birim testler kullanılmaktadır. Aşağıdan yukarıya doğru hiyerarşik bir şekilde ilerlenirken, her bir adımda yapılan birim testlerin başarılı sonuçlanması gerekmektedir. Her bir birim testi bittikten sonra modüllerin birleştirilmesi gerçekleştirilir.

IV. Hibrit Entegrasyon Testi: Birleştirilmesi gerken modüllerin belli bir kısmı yukarıdan aşağıya diğer kısmı ise aşağıdan yukarıya entegrasyon testlerini kullanarak gerçekleştirilen bir test tipidir. Buradaki amaç, modülleri belli gruplara ayırmaya çalışmaktır.

3.2.3. Sistem testi

Sistem testi, yazılım ürününün çalışacağı platformdaki davranışlarını kontrol eden bir yöntemdir. Yapılan tüm testler bir nevi simülasyon ortamında gerçekleşmektedir. Testler sonrasında, ürün yazılımı gerçek sistemler üzerinde çalıştırılmaya başlanır. Simülasyon ortamındaki test sonuçlarıyla gerçek sistemler arasındaki test sonuçları karşılaştırılır ve benzer başarı oranlarına göre hareket edilir. Bu esnada dikkat edilmesi gereken diğer bir husus ise, çevresel faktörlerden dolayı oluşabilecek hatalar da tespit edilmeli ve bu doğrultuda bu hataların azaltılması amaçlanmalıdır.

3.2.4. Kabul testi

Kabul testi tamamen kullanıcı bir başka deyişle müşteri odaklı çalışabilen bir mekanizmadır. Hazırlanılan sistem kullanıcı tarafından test edilmek için uygun koşullara getirilir ve uygulanması gereken adımlar gerçekleştirilir. Bu test aşamasında, sistemin eksiklikleri veya hataları tespit edilebilir ve doğru çalışıp çalışmadığına bakılır. Sonuçlar analiz edildikten sonra yazılımın gerçekleştirilmesi tamamlanmaktadır.

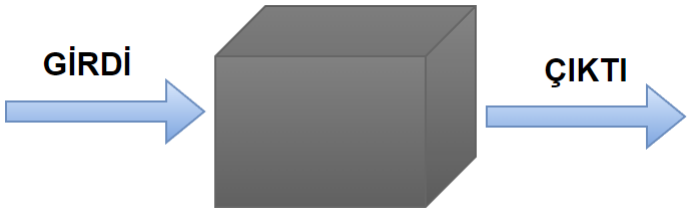
3.3. Yazılım Test Teknikleri

Oluşturulacak test senaryolarının ve test verilerinin kullanılabilmesi için gerekli test tasarım tekniğinin seçilmesi gerekmektedir. Bir testin ilk oluşturulduğu aşamadan, sonuçlandırıldığı aşamasına kadar birçok iş parçacağı gerçekleşmektedir. Bunlar; test verilerilerinin seçilmesi, testin ön koşulları, hedeflenen sonuç ve gerçekleşmesi beklenen sonuç olarak örneklenebilir.

Tüm bu süreç boyunca bazı test teknikleri uygulanmaktadır. Bunlar kara kutu testi, beyaz kutu testi ve gri kutu testi olmak üzere üçe ayrılır (Khan & Khan, 2012).

3.3.1. Kara kutu testi

Uygulamanın iç işleyişi hakkında herhangi bir bilgiye sahip olmadan test etme tekniğidir. Yalnızca sistemin temel yönlerini inceler ve sistemin dahili mantıksal yapısıyla hiç ilgisi yoktur veya çok az ilgisi vardır (Khan & Khan, 2012). Ürün yazılımı hakkında herhangi bir bilgiye ve kodlarına sahip olmadan, hedeflenen sonuç ve elde edilen sonuç arasındaki ilişkinin karşılaştırıldığı bir yazılım test etme tekniğidir. Kara kutu test tekniği ile gerçekleştirilebilecek durumlara örnek olarak; arayüz hataları, performans hataları, veri tabanlarına bağlanma hataları verilebilir.



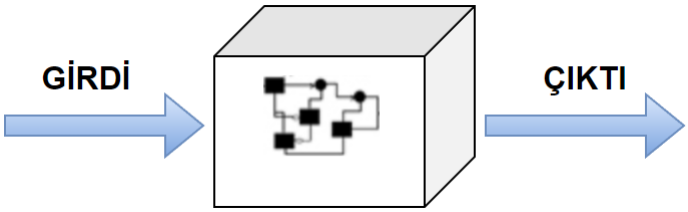
Şekil 3.1. Kara kutu test tekniği

Şekil 3.1'de görüldüğü gibi kara kutu test tekniği iç yapıyı (kodu) test edememektedir. Bu test tekniğinin kullanım alanları; sistem testi, kabul testi gibi daha yüksek seviyeli testler için idealdir (Csstricks, 2021).

3.3.2. Beyaz kutu testi

Kodun iç mantığının ve yapısının detaylı olarak incelenmesidir. Beyaz kutu testinde, bir testçinin kaynak kod hakkında tam bilgiye sahip olması gereklidir (Khan & Khan,

2012). Ürün yazılımı hakkında tüm bilgiye sahip olarak, hedeflenen sonuç ve elde edilen sonuç arasındaki ilişkinin karşılaştırıldığı bir yazılım test etme tekniğidir. Ön bilgiye sahip olunarak geliştirilen test senaryolarının çıktısındaki hataların tespiti daha hızlı bir şekilde görülebilmektedir (Jovanović, 2006).

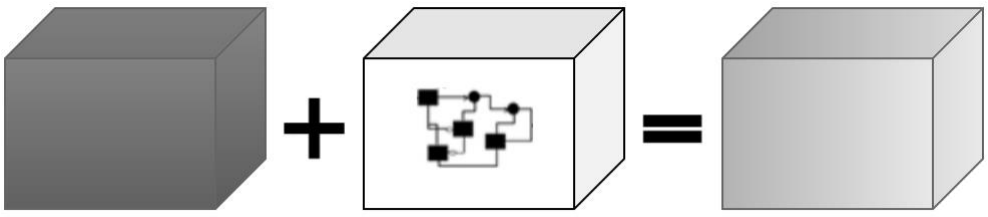


Şekil 3.2. Beyaz kutu test tekniği

Şekil 3.2'de görüldüğü gibi beyaz kutu test tekniği iç yapıyı (kodu) test edebilmektedir. Bu test tekniğinin kullanım alanları; test, birim testi, entegrasyon testi gibi daha düşük seviyeli testler için idealdir (Csstricks, 2021).

3.3.3. Gri kutu testi

Beyaz kutu + Kara kutu = Gri kutu, bir uygulamanın dahili çalışması hakkında sınırlı bilgi ile uygulamayı test etme tekniğidir ve ayrıca bu test tekniği sistemin temel yönleri hakkında bilgi sahibidir (Khan & Khan, 2012). Kısacası bu test tekniği, siyah ve beyaz kutu çalışma mantıklarının birleştirilmesinden oluşmaktadır. Entegrasyon test süreçlerinde sıklıkla kullanılmaktadır. Test senaryoları hazırlanırken, var olan yazılımın kendi teknik özelliklerine bakılmaktadır.



Şekil 3.3. Gri kutu test tekniği

Şekil 3.3'de görüldüğü gibi, gri kutu test tekniği iç yap (kod) kısmen test edilmektedir. Bu test tekniğinin öncelik kullanım alanları; entegrasyon ve sızma testleridir (Csstricks, 2021).

3.3.4. Otomatik ve elle test

Yazılım testi çalıştırılma yöntemleri bakımından ikiye ayrılır. Bunlar Otomatik ve Elle (Manuel) çalıştırılan testlerdir.

Otomatik Test, yazılan tüm test senaryolarının yardımcı bir başka yazılım otomasyonu yardımıyla çalıştırılmasıdır. Elle (Manuel) Test ise, herhangi bir başka yazılım otomasyonuna ihtiyaç duymadan bir yazılım test personeli tarafından kodlarla yazılan test senaryolarının çalıştırılmasıdır.

Otomatik test, sahip olduğu otomasyon özelliklerinin gücünden faydalanarak birden fazla test senaryosunu aynı anda çalıştırabilmektedir. Böylelikle testleri hızlı bir şekilde koşturarak zaman bakımından avantaj sağlamaktadır. Ancak sahip olunan bir otomasyonun olası problemlerinin oluşması durumunda yaşanan sıkıntılar dezavantaj olarak kabul edilmektedir.

Manuel test ile görevlendirilen test personelinin test senaryolarının hazırlamasındaki tecrübesi, testlerin hızlı bir şekilde koşturulmasıyla avantaj sağlamaktadır. Ancak, test senaryolarının tekrar edilmesi ve sık sık değişiklikler yapılması zaman ve maliyet açısından dezavantaj olarak kabul edilmektedir.

3.4. Yazılım Test Otomasyonları

Yazılım test otomasyonlarının kullanım mantığı, elle yazılan testlerin otomatikleştirilmesi ve bunun bir araç haline getirilmesidir. Otomatikleştirmeden kasıt; test senaryolarının sürekli tekrar edilmesidir. Bu durumda, test senaryolarını hazırlayan personelin kodları koşturabileceği otomatik çalışan yazılımlara ihtiyacı vardır ve bu yazılımlara da yazılım test otomasyonu denilmektedir.

Elle yapılması uzun zaman alan ve kısa zaman aralıklarında testlerin koşturulmasını sağlayan yazılım test otomasyonları, süreçlerin daha verimli ilerleyebilmesini sağlamaktadır. Elbette yazılım test otomasyonları kusursuz çalışmamaktadır. Hatasız bir şekilde çalışması veya tüm hataları bulup iyileştirmesi imkansızdır.

Test otomasyonları sahip oldukları arayüzlerin yanı sıra; programlanabilir, kullanımı kolay, hızlı, güvenilir, az maliyetli ve aynı anda çok fazla testi yapabilmesiyle çok fazla tercih edilmektedir.

Bu çalışmada, birim testlerinin üretileceği otomatik yazılım test otomasyonuna ait bir çatı yapısı tasarlanmıştır. Bu yapının tasarım aşamasında java bayt kod ve javassist gibi yapılarda kullanılmıştır.

3.5. Java Byte Code

Tıpkı C ve C++ derleyicilerini assemblerın temsil ettiği gibi java programlarını da bayt kod (byte code) temsil etmektedir. C ve C++ programcıları, derledikleri işlemcinin assembler komut kümesini bilirler. Bu bilgi, hata ayıklama yaparken ve performansı arttırmak için kullanılır. Derleyici tarafından, yazılan kaynak kod için oluşturulan derleyici talimatlarını bilmek, bellek ve performans hedeflerine ulaşmak için nasıl farklı kod yazılabileceğinin bilinmesine yardımcı olur. Buna ek olarak, bir sorunu izlerken kaynak kodunu ayırmak ve yürütülmekte olan birleştirici koduna geçmek için bir hata ayıklayıcı kullanmak genellikle yararlıdır.

Bir java derleyicisi tarafından üretilecek bayt kodunun anlaşılması, java programcısına, C ve C++ programcısının derleyici bilgisine yardımcı olduğu gibi yardımcı olur. Bayt kod, aslında yazılan programın ta kendisidir.

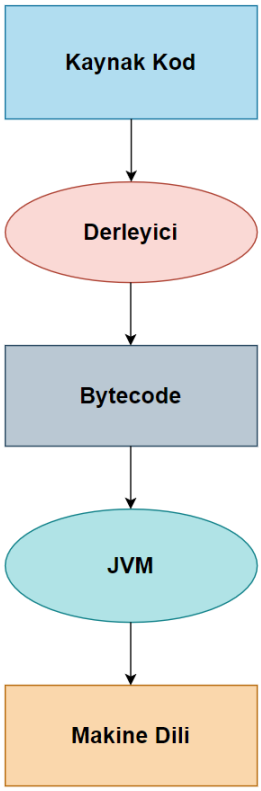
Java'nın taşınabilirlik ve güvenlik gibi problemlerine çözüm olabilmesi için bayt kod gereklidir. Java derleyicisinin çıktısı yürütülebilir bir kod olmadığı için bayt kod kullanmak zorundadır. Bayt kodlar, JVM tarafından yorumlanmaktadır. Bu sayede, bayt kodlar yürütme zamanında (run-time) iyi bir şekilde optimize edilmektedir. Bayt kod aracılığıyla, bir java programı birçok değişik ortamda çalıştırılabilmektedir.

JVM, her platformda çalışabilmektedir. Bununla birlikte, platforma ait sistemde bir yürütme zamanı var olduğu sürece, herhangi bir java programı çalıştırılabilir. JVM'lerin özellikleri platformdan platforma değişkenlik göstermesine rağmen hepsi java bayt kodunu yorumlayabilmektedir. İşte bu noktada, bayt kodunun her durumda yorumlanıyor oluşu taşınabilirlilik özelliğini teşkil etmektedir. Java programının yorumlanıyor oluşu aynı zamanda güvenli olduğunu da göstermektedir. Yürütülen java programları java sanal makine kontrolü altında olduğu için, JVM o java programını kapsar ve sistemi dış etkilerden engelleyebilir (Karal, 2004).

Bayt kod, java sanal makinesinin makine dilidir. Bir JVM tarafından bir sınıf (class) dosyası yüklendiğinde, sınıftaki her metot için bir bayt kodu akışı oluşur. Bayt kod akışları JVM'nin metot alanına saklanır. Herhangi bir sınıfın bayt kodları, programın koşturulması sırasında bu metotda yürütülmeye başlanmıştır. Bu metot, yürütme zamanında derlenir veya belirli bir JVM'nin tasarımcısı tarafından seçilen herhangi bir teknikle çalıştırılabilir.

Bir metodun bayt kod akışı, JVM için bir talimatlar dizisidir. Her komut bir baytlık opcode ve ardından sıfır veya daha fazla işlenenden (operand) oluşmaktadır. Opcode, gerçekleştirilecek eylemi gösterir. JVM'nin eyleme geçebilmesi için daha fazla bilgi gerekiyorsa, bu bilgiler opcode'u hemen izleyen bir veya daha fazla işlenene kodlanır. Her bir opcode türünün anımsatıcısı vardır.

Çalışmanın sonunda Ek-A tablosunda yer aldığı gibi, bayt kod talimat seti karmaşık olacak şekilde tasarlanmıştır. Tablo oluşturma ile ilgili iki kod hariç tüm talimatlar bayt sınırları ile hizalanır. Toplam opcode sayısı yeterince azdır, böylece bayt kodlar sadece bir bayt yer kaplamaktadır. Bu, JVM tarafından çalıştırılmadan önce sınıf dosyalarının boyutunu en aza indirmeye yardımcı olmaktadır. Ayrıca, JVM uygulamasının boyutunun da küçük olmasına yardımcı olmaktadır. JVM'deki tüm hesaplamalar yığın (stack) üzerinde gerçekleşmektedir. Bayt kod ile ilgili talimatlar bu sebeple öncelikle yığında çalışmaktadır (Venners, 1996).



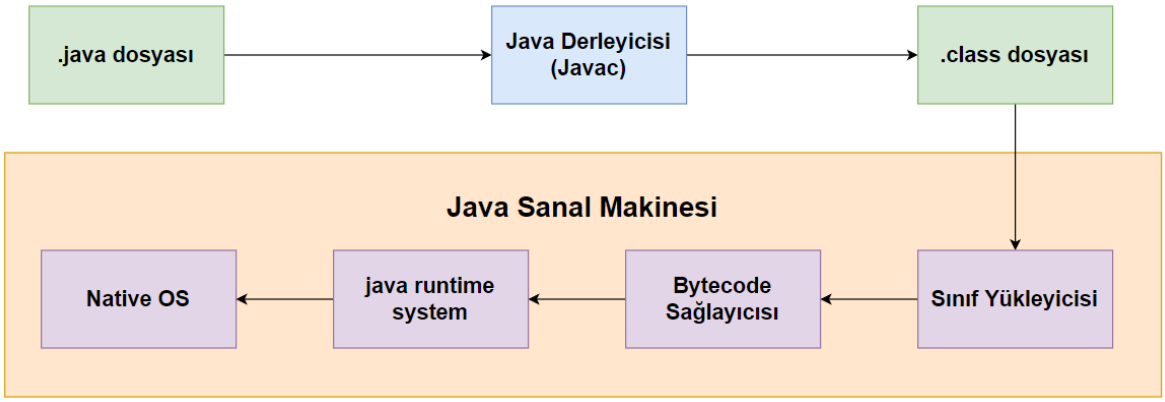
Şekil 3.4. Bytecode çalışma mantığı

Şekil 3.4'te görüldüğü üzere, Java'da bir program yürütülürken devam eden birçok işlem vardır. Bu, java'nın platformdan bağımsız olmasının nedenlerinden biri olan Bytecode'lar sayesinde gerçekleşen işlemlerdir. Java bayt kodu *.class* dosyası biçimindeki makine kodudur. Java'daki bayt kod, JVM için ayarlanan komuttur ve bir derleyiciye benzer şekilde çalışmaktadır.

3.5.1. Byte code çalışma prensibi

Bir java programı yürütüldüğünde, derleyici kod parçasını derler ve o programdaki her bir kod için bir *.class* dosyası biçimde bir bayt kod oluşturur. Bu bayt kod, başka herhangi bir platformda da çalıştırılabilir. Ancak bayt kod, bir tercüman gerektiren ve o olmadan çalıştırılamayan bir koddur. Özetle, JVM burada önemli bir rol oynamaktadır.

Şekil 3.5'te görüldüğü gibi, derlemeden sonra oluşturulan bayt kod JVM tarafından çalıştırılır. Yürütme için gereken kaynaklar, işlemcinin kaynaklarını tahsis ettikten sonra düzgün bir yürütme işlemi için JVM tarafından sağlanır.

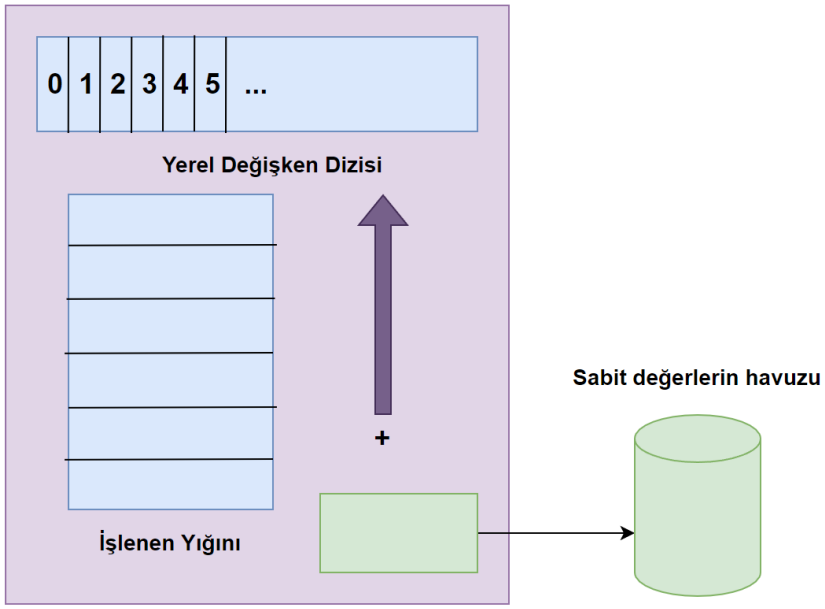


Şekil 3.5. JVM çalışma mantığı (Keifer, 2021).

Bayt kod yakından incelendiğinde, belirli opcode'ların olduğu görülmektedir. Bireysel kodlara genellikle Opcode denilmektedir. Birden fazla opcode talimatı genellikle bayt kod olarak adlandırılmaktadır. Bazı opcode'ların önünde *a* veya *i* harfi bulunmaktadır. Örneğin, *aload_0* ve *iload_2* ifadeleri gibi. Bu ön ekler, opcode'un birlikte çalıştığı türü temsil eder. a ön eki, opcode'un bir nesne başvurusunu değiştirdiği anlamına gelir. i ön eki ise, opcode'un bir tamsayıyı işlediği anlamına gelir. Diğer opcode'lar; bayt için *b*, char için *c* ve double için *d* şeklinde kullanılırlar. Bu ön ekler, ne tür verilerin işlendiği hakkında bilgi vermektedir.
Bayt kodunun ayrıntılarını anlamak için, JVM tarafından bayt kodunun yürütülmesinin nasıl çalıştığını anlamak gerekir. JVM, yığın tabanlı bir makinedir. Her iş parçacığı verilerini kendi çerçevesine depolayıp, bir çerçeve belleği haline getiren bir JVM yığınına sahiptir. Her yordam çağırıldığında bir çerçeve yığını oluşturulur ve işlenen yığını, bir dizi yerel değişken ve geçerli sınıfın çalışma zamanı gibi verileri içermektedir.

Şekil 3.6'da kavramsal olarak bir yığının çerçeve (frame) yapısı görülmektedir. Yerel değişkenler tablosu olarak da bilinen yerel değişkenler dizisi, yordamın parametrelerini içerir ve yerel değişkenlerin değerlerini tutmak için de kullanılır. Parametreler önce 0 indisinden başlayarak dizinde saklanır. Yapı eğer bir kurucu (constructor) veya dinamik (instance) metot içinse, referans 0 konumunda depolanır.

Daha sonra konum 1 ilk resmi parametreyi, konum 2 ise ikinci resmi parametreyi alır. Statik bir metot için ilk resmi metot parametresi 0 konumunda, ikincisi ise 1 konumunda depolanır.



Şekil 3.6. Yığın çerçeve (stach-frame) yapısı (Anouti, 2018).

Yerel değişkenler dizisinin boyutu derleme zamanında belirlenir ve yerel değişkenlerin sayısı ve boyutu bir biçimsel yordam parametresine bağlıdır. İşlenen yığın, değerleri itmek ve çekmek için LIFO (Last in First out - Son Giren İlk Çıkar) yöntem yığını kullanılır. Boyutu, derleme zamanında belirlenir. Belirli opcode talimatları değerleri işlenen yığına iletilir, diğerleri işlenenleri yığından alır, manipüle eder ve sonucu iletir. İşlenen yığın, yordamlardan dönüş değerleri almak için de kullanılır.

3.5.2. Byte code ve machine code arasındaki farklar

Makine kodu ile bayt kod arasındaki temel fark, makine kodunun makine dili veya ikili dosyada doğrudan CPU tarafından çalıştırılabilen bir dizi talimat olmasıdır. Bayt kodu, yürütülmesi için bir tercümana dayanan bir kaynak koddur ve derlenerek oluşturulan çalıştırılımaz bir koddur (Anouti, 2018).

3.5.3. Byte code türleri ve tanımları

Bazı bayt kodların tür ve tanımları şöyledir;

3.5.3.1. İlkel veri türleri

JVM yedi ilkel veri türünü desteklemektedir. Java programcıları bu veri türlerini ve değişkenlerini bildirim amaçlı kullanabilmektedir. Java bayt kodları bu veri türleri üzerinde çalışmaktadır. Çizelge 3.1'de ilkel veri türleri ve tanımlamaları verilmiştir.

Çizelge 3.1. İlkel veri türleri (Venners, 1996).

Veri Türü	**Açıklaması**
byte	1-bayt işaretli, ikinin tümleyeni, tamsayı
short	2-Bayt işaretli, ikinin tümleyeni, tamsayı
int	4-bayt işaretli, ikinin tümleyeni, tamsayı
long	8-bayt işaretli, ikinin tümleyeni, tamsayı
float	4-bayt IEEE 754, tek duyarlıklı kayan nokta biçimi
double	8-bayt IEEE 754, çift duyarlıklı kayan nokta biçimi
char	2-bayt işaretsiz, tek kodlu karakter

İlkel veri türleri, bayt kodu akışlarında işlenen (operands) olarak görünür. 1 byte'tan fazla yer kaplayan tüm ilkel tipler, bayt kod akışında big-endian[1] düzeninde saklanır. Bu da daha yüksek sıralı baytların alt sıra baytlardan önce geldiği anlamına gelir. Örneğin, 256 (hex 0100) sabit (const) değerini yığına atmak için önce *sipush* op kodu ve ardından kısa bir işleneni kullanılır. *sipush 256;* kodu JVM big-endian'dır ve "01 00" olarak görülmektedir. Little-endian ise "00 01" şeklinde görülmektedir (Venners, 1996).

Java opcode genellikle işlenen kodları gösterir. Bu işlenen kodların türleri JVM'ye tanımlanmamaktadır. Opcode'lar sırasıyla *iload, lload, fload* ve *dload* (int, long, float ve double) türlerindeki yerel değişkenleri yığında saklarlar (Venners, 1996).

[1] İnsanların soldan sağa veya sağdan sola alfabelere sahip olmaları gibi işlemciler de byte'ları saklarken önemli byte'ın solda veya sağda olmasına göre sınıflandırılır. Buna *endianness* da denir. Önemli byte'in solda olduğu sıralamaya *big-endian* denir. Önemli byte'in en sağda olduğu sıralama ise *little-endian* olarak adlandırılır. JVM'de big-endian kullanır.

3.5.3.2. Sabitleri yığında tutma

Birçok opcode, sabitleri yığına gönderir. Opcode'lar, gönderilecek sabit değerleri üç farklı şekilde belirtir. Sabit değer ya opcode'un kendisinde dolaylı bir şekilde bulunur ve bayt kodu akışındaki opcode'u işlenen olarak takip eder, ya da sabit havuzundan çıkarılır. Bazı opcode'lar tek başlarına yığına gönderilecek bir tip ve sabit değeri işaret eder. Örneğin, *iconst_1* opcode'u, JVM'ye bir tamsayı değerini itmesini işaret eder. Bu tip bayt kodları, çeşitli tiplerde yaygın olarak itilen bazı sayılar için tanımlanır. Bu talimatlar bayt kodu akışında yalnızca 1 bayt yer kaplar. Bayt kod yürütme verimliliğini arttırır ve bayt kod akışlarının boyutunu azaltır. Tamsayıları ve ondalıkları yığına gönderen opcode'lar Çizcelge 3.2'de gösterilmiştir (Venners, 1996).

Çizelge 3.2. Tamsayıları ve ondalıklı sayıları yığına gönderen opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
iconst_m1	(Yok)	int -1'i yığına iter
iconst_0	(Yok)	int 0'ı yığına iter
iconst_1	(Yok)	int 1'i yığına iter
iconst_2	(Yok)	int 2'yi yığına iter
iconst_3	(Yok)	int 3'ü yığına iter
iconst_4	(Yok)	int 4'ü yığına iter
iconst_5	(Yok)	int 5'i yığına iter
fconst_0	(Yok)	float 0'ı yığına iter
fconst_1	(Yok)	float 1'i yığına iter
fconst_2	(Yok)	float 2'yi yığına iter

Çizelge 3.2'de gösterilen opcode'lar 32 bit genişliğindedir. Bu nedenle, yığına bir int veya float her itildiğinde bir slotluk yer kaplar. Çizelge 3.3'de gösterilen opcode'lar long ve double türleri içindir. Bu türler yığında iki slot yer kaplamaktadır (Venners, 1996).

Çizelge 3.3. Long ve Double'ları yığına gönderen opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
lconst_0	(Yok)	long 0'ı yığına iter
lconst_1	(Yok)	long 1'ı yığına iter
dconst_0	(Yok)	double 0'ı yığına iter
dconst_1	(Yok)	double 1'ı yığına iter

Diğer bir opcode, yığına örtülü bir değer iter. Çizelge 3.4'te gösterilen *aconst_null* opcode'u, yığına boş bir nesne başvurusu gönderir. Bir nesne başvurusunun biçimi JVM uygulamasına bağlıdır. Bir nesne başvurusu bir şekilde çöp nesneleri toplayan yığın üzerindeki bir Java nesnesine atıfta bulunacaktır. Boş bir nesne başvurusu, bir nesne başvuru değişkeninin o anda geçerli herhangi bir nesneye başvurmadığını gösterir. *aconst_null* opcode'u, bir nesne referans değişkenine null atamak için kullanılır (Venners, 1996).

Çizelge 3.4. Yığına boş bir nesne başvurusu yapan opcode (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
aconst_null	(Yok)	Yığına bir boş nesne referansı gönderir

Çizelge 3.5'te bulunan iki opcode'u hemen takip eden bir işlenenle itme sabitini göstermektedir. Bu kodlar, bayt ve short tipleri için geçerli aralıktaki tamsayı sabitlerini itmek için kullanılmaktadır. Opcode'u takip eden bayt veya short, yığına itilmeden önce tamsayı değerine genişletilir çünkü java yığınındaki her slot 32 bit genişliğindedir. Yığının üzerine gönderilen bayt ve short ile ilgili işlemler aslında int (tamsayı) değerleri ile eşdeğer olarak gönderilir (Venners, 1996).

Çizelge 3.5. Byte ve Short'ları yığına gönderen opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
bipush	byte1	bayt1'i (bir bayt türü) bir int'ye genişletir ve yığına it
sipush	byte1, byte2	bayt1, bayt2'yi (bir short türü) bir int'ye genişletir ve yığına iter

Çizelge 3.6'da görülen üç opcode, sabit havuzunda kullanılır. Bir sınıfla ilişkili, son değişken değerleri gibi tüm sabitler, sınıfın sabit havuzunda saklanır. Sabit havuzdan sabitleri iten opcode'lar, sabit bir havuz indeksi belirterek hangi sabitin itileceğini belirten işlenenlere sahiptir. Java sanal makinesi dizinde verilen sabiti arar, sabitin türünü belirler ve yığının üzerine iter (Venners, 1996).

Sabit havuz dizini, bayt kod akışındaki opcode'u hemen izleyen işaretsiz bir değerdir. *lcd1* ve *lcd2* opcode'ları, 32 bitlik bir öğeyi yığına int veya float gibi iter.

lcd1 ve *lcd2* arasındaki fark, *lcd1*'in sadece 1 ile 255 arasındaki sabit havuz konumlarına başvurabilmesidir, çünkü indeksi sadece 1 bayttır. *lcd2*, 2 baytlık bir dizine sahiptir, bu nedenle herhangi bir sabit havuz konumuna işaret edebilir. *lcd2w* ayrıca 2 baytlık bir indekse sahiptir ve 64 bitlik bir long veya double içeren herhangi bir sabit havuz yerini belirtmek için kullanılır. Sabit havuzdan sabitleri iten opcode'lar Çizelge 3.6'da gösterilmektedir (Venners, 1996).

Çizelge 3.6. Sabit havuzdan sabitleri iten opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
ldc1	indexbyte1	indexbyte1 tarafından belirtilen 32-bit constant_pool girişini yığına iter
ldc2	indexbyte1, indexbyte2	indexbyte1 ve indexbyte2 tarafından belirtilen 32-bit constant_pool
ldc2w	indexbyte1, indexbyte2	indexbyte1 ve indexbyte2 tarafından belirtilen 64-bit constant_pool girişini yığına iter

3.5.3.3. Yerel değişkenleri yığına aktarma

Yerel değişkenler yığın çerçevesinin özel bir bölümünde saklanır. Yığın çerçevesi, yığının o anda yürütülmekte olan yordam tarafında kullanılan kısımdır. Her yığın çerçevesi üç bölümden oluşur; yerel değişkenler, yürütme ortamı ve işlenen yığın (Venners, 1996).

Yerel bir değişkenin yığına itimesi aslında bir değerin yığın çerçevesinin yerel değişkenler bölümünden işlenen bölümüne taşınmasını içerir. Yürütülmekte olan yordamın işlenen bölümü her zaman yığının en üstündedir. Bu nedenle, geçerli yığın çerçevesinin işlenen bölümüne bir değer itmek, yığının üstüne bir değer itmekle aynıdır. Java yığını, 32 bit slotlardan oluşan son giren ilk çıkar mantığında çalışan bir yapıdır. Yığındaki her slot 32 bit içerdiğinden, tüm yerel değişkenler en az 32 bit işgal eder. 64 bit büyüklüğünde olan long ve double tipi yerel değişkenler, yığın üzerinde iki slot kullanır. Byte ve short türündeki yerel değişkenler, int türündeki yerel değişkenler olarak saklanır. Ancak, daha küçük tür için geçerli bir değerle depolanır. Örneğin, bir byte türünü temsil eden int yerel değişkeni her zaman bir byte için geçerli bir değer içerir (Venners, 1996).

Bir yordamın her yerel değişkeni benzersiz bir dizine sahiptir. Bir yordamın yığın çerçevesinin yerel değişken bölümü, her biri dizinler tarafından adreslenebilen 32 bitlik bir dizi olarak düşünülebilir. İki slot işgal eden long ve double tipli yerel değişkenlere, iki slot indeksinin altında değinilmektedir. Örneğin, ikinci ve üçüncü slotları kaplayan bir çifte *ikili indeks* denir (Venners, 1996).

int ve float yerel değişkenleri işlenen yığınına iten birkaç opcode bulunmaktadır. Yaygın olarak kullanılan bir yerel değişken konumuna dolaylı olarak atıfta bulunan bazı opcode'lar tanımlanmıştır. Örneğin, *iload_0* int yerel değişkeni sıfır konumuna yüklenir. Diğer yerel değişkenler yığına, yerel değişken indeksinin opcode'tan sonraki ilk bayttan alan bir opcode ile itilir. *iload* komutu bu tür bir opcode örneğidir. *iload*'u izleyen ilk bayt, yerel değişkeni ifade eden unsigned 8-bit'lik bir dizin olarak yorumlanır (Venners, 1996).

Unsigned 8-bit yerel değişken indeksleri, yordamlardaki yerel değişkenler sayısını 256 ile sınırlar. *wide* olarak adlandırılan ayrı bir komut, 8 bitlik bir dizini 8 bit daha genişletebilir. Bu yerel değişken sınırını 64 kilobyte'a çıkarır. *wide* opcode'u 8 bitlik bir işlenen izler. *wide* opcode ve işleneni (operand), 8 bitlik unsigned yerel değişken indeksi alan *iload* gibi bir komuttan önce gelebilir. JVM, 16 bit unsigned yerel değişken indeksi elde etmek için *wide* komutunun 8 bit işlenenini *iload* komutunun 8 bit işleneni ile birleştirir (Venners, 1996). int ve float yerel değişkenleri yığına iten opcode'lar Çizelge 3.7'de gösterilmektedir.

Çizelge 3.7. Int ve float yerel değişkenlerini yığına iten opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
iload	vindex	Yerel değişken vindex konumundan int'i yığına iter
iload_0	(Yok)	Yerel değişken 0 konumundan int'i yığına iter
iload_1	(Yok)	Yerel değişken 1 konumundan int'i yığına iter
iload_2	(Yok)	Yerel değişken 2 konumundan int'i yığına iter
iload_3	(Yok)	Yerel değişken 3 konumundan int'i yığına iter
fload	vindex	Yerel değişken vindex konumundan float'ı yığına iter
fload_0	(Yok)	Yerel değişken 0 konumundan float'ı yığına iter
fload_1	(Yok)	Yerel değişken 1 konumundan float'ı yığına iter
fload_2	(Yok)	Yerel değişken 2 konumundan float'ı yığına iter
fload_3	(Yok)	Yerel değişken 3 konumundan float'ı yığına iter

Bir sonraki Çizelge 3.8'de yığına long ve double yerel değişkenleri iten talimatlar gösterilmiştir. Bu talimatlar, 64 biti yığın çerçevesinin yerel değişken bölümünden işlenen bölümüne taşır (Venners, 1996).

Çizelge 3.8. Yığına long ve double yerel değişkenlerini iten opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
lload	vindex	Yerel değişken vindex ve vindex+1 konumlarından long'u yığına iter
lload_0	(Yok)	Yerel değişken 0 ve 1 konumlarından long'u yığına iter
lload_1	(Yok)	Yerel değişken 1 ve 2 konumlarından long'u yığına iter
lload_2	(Yok)	Yerel değişken 2 ve 3 konumlarından long'u yığına iter
lload_3	(Yok)	Yerel değişken 3 ve 4 konumlarından long'u yığına iter
dload	vindex	Yerel değişken vindex ve vindex+1 konumlarından double'ı yığına iter
dload_0	(Yok)	Yerel değişken 0 ve 1 konumlarından double'ı yığına iter
dload_1	(Yok)	Yerel değişken 1 ve 2 konumlarından double'ı yığına iter
dload_2	(Yok)	Yerel değişken 2 ve 3 konumlarından double'ı yığına iter
dload_3	(Yok)	Yerel değişken 3 ve 4 konumlarından double'ı yığına iter

Yerel değişkenleri iten son opcode grubu Çizelge 3.9'da verilmiştir. 32 bit nesne referanslarını yığın çerçevesinin yerel değişkenler bölümünden işlenen bölümüne taşır.

Çizelge 3.9. Yerel değişkenleri iten opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
aload	vindex	Nesne referansını yerel değişken konumundan vindex'e iter
aload_0	(Yok)	Nesne referansını yerel değişken konumundan 0'a iter
aload_1	(Yok)	Nesne referansını yerel değişken konumundan 1'e iter
aload_2	(Yok)	Nesne referansını yerel değişken konumundan 2'ye iter
aload_3	(Yok)	Nesne referansını yerel değişken konumundan 3'e iter

3.5.3.4. Yerel değişkenleri çekme

Yığın üzerine bir yerel değişken iten her bir opcode için, yığının üst kısmını yerel değişkene geri gönderen karşılık gelen bir opcode vardır. Bu opcode'ların isimleri, push opcode'ların adlarındaki "*load*" yerine "*store*" kullanılarak oluşturulabilir. İşlenen yığının tepesinden yerel bir değişkene çekilen int ve float değerlerinin opcode'ları Çizelge 3.10'da listelenmiştir. Bu opcode'ların her biri, yığının tepesinden bir 32 bitlik değeri yerel bir değişkene taşır. (Venners, 1996)

Çizelge 3.10. Bir yığından yerel bir değişkene atanan opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
istore	vindex	İnt'ı yerel değişken vindex konumuna çeker.
istore_0	(Yok)	İnt'ı yerel değişken 0 konumuna çeker.
istore_1	(Yok)	İnt'ı yerel değişken 1 konumuna çeker.
istore_2	(Yok)	İnt'ı yerel değişken 2 konumuna çeker.
istore_3	(Yok)	İnt'ı yerel değişken 2 konumuna çeker.
fstore	vindex	Float'ı yerel değişken vindex konumuna çeker.
fstore_0	(Yok)	Float'ı yerel değişken 0 konumuna çeker.
fstore_1	(Yok)	Float'ı yerel değişken 1 konumuna çeker.
fstore_2	(Yok)	Float'ı yerel değişken 2 konumuna çeker.
fstore_3	(Yok)	Float'ı yerel değişken 3 konumuna çeker.

Çizelge 3.11. Yerel değişken içindeki long ve double değerlerinin opcode'ları (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
lstore	vindex	Long'u yerel değişken vindex ve vindex+1 konumuna çeker.
lstore_0	(Yok)	Long'u yerel değişken 0 ve 1konumuna çeker.
lstore_1	(Yok)	Long'u yerel değişken 1 ve 2 konumuna çeker.
lstore_2	(Yok)	Long'u yerel değişken 2 ve 3 konumuna çeker.
lstore_3	(Yok)	Long'u yerel değişken 3 ve 4 konumuna çeker.
dstore	vindex	Double'ı yerel değişken vindex ve vindex+1 konumuna çeker.
dstore_0	(Yok)	Double'ı yerel değişken 0 ve 1 konumuna çeker.
dstore_1	(Yok)	Double'ı yerel değişken 1 ve 2 konumuna çeker.
dstore_2	(Yok)	Double'ı yerel değişken 2 ve 3 konumuna çeker.
dstore_3	(Yok)	Double'ı yerel değişken 3 ve 4 konumuna çeker.

Çizelge 3.11, bir yerel değişken içindeki long ve double değerlerinin talimatlarını gösterir. Bu yönergeler, 64 bit değerini işlenen yığınının tepesinden yerel bir değişkene taşır.

Yerel değişkenlere açılan son opcode grubu Çizelge 3.12'de gösterilmektedir. Bu opcode ile, işlenen yığınının tepesinden yerel bir değişkene 32 bitlik bir nesne başvurusu atanır.

Çizelge 3.12. Yerel değişkenlere çekilen son opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	**Operand'lar**	**Açıklama**
astore	vindex	Nesne referansını yerel değişken konum vindex'e çeker
astore_0	(Yok)	Nesne referansını yerel değişken konum 1'e çeker
astore_1	(Yok)	Nesne referansını yerel değişken konum 2'ye çeker
astore_2	(Yok)	Nesne referansını yerel değişken konum 3'e çeker
astore_3	(Yok)	Nesne referansını yerel değişken konum 4'e çeker

3.5.3.5. Dönüşüm türleri

JVM, bir ilkel türden diğer türlere dönüşebilen birçok opcode'a sahiptir. Bayt kod akışındaki dönüştürme işlem kodlarını hiçbir işlenen izlemez. Dönüştürülecek değer yığının üstünden alınır. JVM, yığının en üstünde değeri açar, dönüştürür ve sonucu yığına geri iter. int, long, float ve double arasında dönüştürme yapan opcode'lar Çizelge 3.13'de gösterilmektedir. Bu dört türün olası her bir kombinasyonu için bir opcode vardır (Venners, 1996).

Tam sayıdan daha küçük tamsayıların dönüşümleri Çizelge 3.14'te gösterilmiştir. Tamsayıdan daha küçük bir long, float ve double'dan doğrudan tür dönüştüren hiçbir opcode yoktur. Bu nedenle, örneğin bir float-byte dönüşümü için iki adım gerekmektedir. İlk olarak *f2i* ile float'u int'e sonrasında *int2byte* ile int'dan byte'a dönüştürülür (Venners, 1996).

Çizelge 3.13. Int, long, float ve double arasındaki değer dönüşümlerini yapan opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	Operand'lar	Açıklama
i2l	(Yok)	İnt'ı long'a çevirir
i2f	(Yok)	İnt'ı float'a çevirir
i2d	(Yok)	İnt'ı double'a çevirir
l2i	(Yok)	long'u int'a çevirir
l2f	(Yok)	long'u float'a çevirir
l2d	(Yok)	long'u double'a çevirir
f2i	(Yok)	float'ı int'a çevirir
f2l	(Yok)	float'ı long'a çevirir
f2d	(Yok)	float'ı double'a çevirir
d2i	(Yok)	double'ı int'e çevirir
d2l	(Yok)	double'ı long'a çevirir
d2f	(Yok)	double'ı float'a çevirir

Çizelge 3.14. Tam sayıdan daha küçük tamsayıların dönüşümleri için kullanılan opcode'lar (Venners, 1996).

Opcode (İşlem Kodu)	Operand'lar	Açıklama
int2byte	(Yok)	İnt'ı byte'a çevirir
int2char	(Yok)	İnt'ı char'a çevirir
int2short	(Yok)	İnt'ı short'a çevirir

Byte, short ve char gibi daha küçük ilkel veri türlerini int'e dönüştüren mevcut opcode'lar olsa da ters yönde dönüşen hiçbir opcode yoktur. Bunun sebebi; byte, short ve char yığına aktarılmadan önce int'e etkili bir şekilde dönüştürülmesidir. byte, short ve char üzerinde aritmetik işlemler önce int'a değerlerine dönüştürülür, int üzerinde aritmetik işlemler yapılır ve sonuç olarak bir int elde edilir. Bunun anlamı, eğer 2 byte eklenirse bir int elde edelir ve bir byte sonucu istenirse, int sonucunu açıkça bir byte'a dönüştürmesi gerekir (Venners, 1996).

3.5.4. Boyut ve hız sorunları

Performans, java kullanan birçok masaüstü ve sunucu sistemi için kritik bir konudur. Java bu sistemlerden daha küçük yerleşik cihazlara geçtikçe, boyut sorunları da önem kazanır. Bir dizi java talimatı için hangi bayt kodunun üretildiğini bilmek, daha küçük ve daha verimli kod yazmaya yardımcı olabilir. Örneğin, Çizelge 3.15'te gösterilen java ile senkronizasyon incelenecek olursa, şekildeki iki metot, üst öğeyi

dizi olarak uygulanan bir tamsayı yığınından döndürür. Her iki metot da senkronizasyonu kullanır ve işlevsel olarak eşdeğerdir (Java bytecode, 2020).

Çizelge 3.15. Opcode senkronizasyon örneği (Java bytecode, 2020).

public synchronized int top1() { **return intArr[0];** **}**	**public int top2() {** **synchronized (this) {** **return intArr[0];** **}** **}**

Bu metotlar senkronizasyonu farklı şekilde kullanılmasına ragmen, işlevsel olarak aynıdır. Ancak açık olmayan şey, farklı performans ve boyut özelliklerine sahip olmalarıdır. Bu durumda, *top1* metodu *top2* metodundan yaklaşık yüzde 13 daha hızlı ve çok daha küçüktür. Bu yöntemlerin nasıl farklı olduğunu görmek için oluşturulan bayt kodu incelenebilir. *Top2* metodu, senkronizasyon ve istisna işleminin yapılış şeklinden dolayı *top1* metodundan daha büyük ve daha yavaştır. *top1* metodunun ekstra kod üretmeyen senkronize yöntem değiştiriciyi kullandığı dikkate alınmalıdır. Aksine, *top2* metodunun gövdesinde senkronize bir deyim olarak kullanılır.

3.5.5. Derleyici seçenekleri

Javac derleyicisi, kullanım için birkaç seçenek sunar. Birincisi –O seçeneğidir. JDK belgeleri, –O'nun kodu yürütme hızı için optimize edeceğini iddia eder. Sun java 2 sdk'lı javac derleyicisinde kullanılan -O, oluşturulan kod üzerinde hiçbir etkisi bulunmamaktadır. Sun javac derleyicisinin önceki sürümlerinde bazı temel bayt kod optimizasyonları yapılmıştır, Ancak bunlar daha sonra kaldırılmıştır. Bununla birlikte, sdk dökümanları güncellenmemiştir. –O'nun bir seçenek olarak kalmasının tek sebebi, eski *make* dosyalarıyla uyumluluktur. Bu nedenle, şu anda kullanmak için bir neden bulumamaktadır. Ayrıca, javac derleyicisi tarafından oluşturulan bayt kodunun yazılan koddan daha iyi olmadığı anlamına gelir. Örneğin, *invariant* içeren bir döngü yazılır, *invariant* javac derleyicisi tarafından döngüden kaldırılmaz. Programcılar, kötü yazılmış kodu temizlemek için diğer dillerdeki derleyicileri

kullanırlar. Ne yazıkki, javac bunu yapamaz. Daha da önemlisi, javac derleyicisi döngü açma, cebirsel basitleştirme, güç azaltma ve diğer basit optimizasyonları yapamaz. Bu avantajları ve diğer basit optimizasyonları elde etmek için programcının bunları java kaynak kodunda gerçekleştirmesi ve bunları gerçekleştirmek için javac derleyeicisine güvenmemesi gerekir. Java derleyicisinin daha hızlı ve daha küçük bayt kodu üretmesini sağlamak için kullanabilecek birçok teknik bulunmaktadır. Javac derleyicisi ayrıca -g ve -g: none seçeneklerini sunar. -g seçeneği derleyiciye tüm hata ayıklama bilgilerini oluşturmasını söyler. -g: none seçeneği derleyiciye hata ayıklama bilgisi oluşturmamasını söyler. -g: none ile derlemek mümkün olan en küçük .class dosyasını oluşturmayı sağlar. Bu nedenle, dağıtımdan önce mümkün olan en küçük .class dosyalarını oluşturmak için bu seçenek kullanılmalıdır (Java bytecode, 2020).

3.5.6. Java hata ayıklayıcıları

Bir Java hata ayıklayıcısı, C veya C ++ hata ayıklayıcısına benzer bir hata ayıklayıcısı özelliğine sahiptir. Java kaynak kodunun oluşturulması, bayt kodunu ortaya çıkarır, tıpkı C veya C ++ kodu gibi ayrıştırırak kodu gösterdiği gibi. Bu özelliğe ek olarak, bir başka kullanışlı özellik ise, baytkod vasıtasıyla tek bir adımda, bir opcode tek seferde çalıştırılabilme yeneğine sahiptir. Bu işlevsellik düzeyi, programcının kodu hatadan arındırmasının yanı sıra java derleyicisi tarafından ilk elden bayt kodun oluşturulduğunu görmesini sağlar. Bir programcı, oluşturulan kod hakkında ne kadar çok bilgi sahibi olursa, sorunlardan kaçınma şansı o kadar artar. Bu tür hata ayıklayıcı özelliği, programcıları kaynak kodları için yürütülen bayt koduna bakmaya ve anlamaya teşvik eder (Java bytecode, 2020).

3.6. Opcode ve Yığın Çerçevesi Mantığı

Bir java sınıf dosyasındaki her bir metodun, her biri opcode (1 byte), operand1 (optional), operand2 (optional) gibi farklı biçimlere sahip bir dizi talimattan oluşan bir kod segmenti bulunmaktadır. Bu, bir baytlık opcode ve çalıştırılacak verileri içeren sıfır veya daha fazla işlenenden oluşan bir talimattır. Yürütülmekte olan yığın çerçevesi içinde, bir yönerge, değerleri işlenen yığınına itebilir veya çekebilir ve

potansiyel olarak dizi yerel değişkenlerine değerler yükleyebilir veya depolayabilir (Anouti, 2018).

Çizelge 3.16'da main metodu için method etiketi kullanıldığı görülmektedir. Bu, String'lerden oluşan bir dizine ait olduğunu belirten bir tanımlayıcıdır ve void (değer döndürmeyen) dönüş türüne sahiptir. Bu metot, ACC_PUBLIC ve ACC_STATIC gibi bir dizi etiketler kullanmaktadır. Kodun en önemli kısımları, işlenen yığının maksimum derinliği (bu durumda 2) ve bu methodun içerisinde kullanılan yerel değişkenlerin sayısı gibi özniteliklerdir. Args argümanı ile tüm yerel değişkenler indeks 0'da referanslanmıştır. Diğer 3 yerel değişken, kaynak kodundaki a, b ve c değişkenlerine karşılık gelmektedir (Anouti, 2018).

Çizelge 3.16. Bir java kodunun byte code'a dönüştürülmesi örneği (Anouti, 2018).

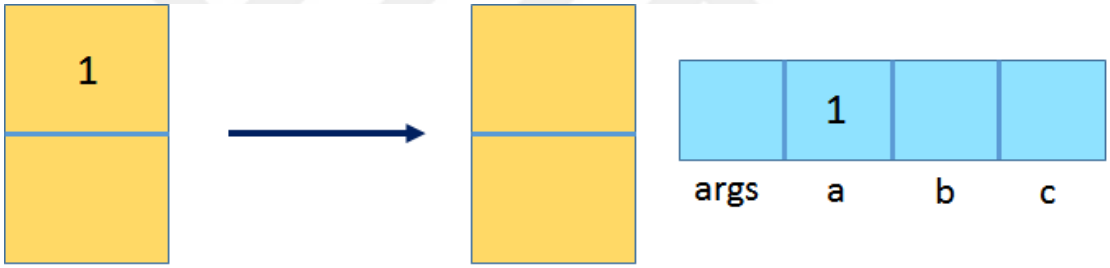
Java Code	**Byte Code**
public static void main(String[] args) { **int a = 1;** **int b = 2;** **int c = a + b;** **}**	**public static void main(java.lang.String[]);** **descriptor: ([Ljava/lang/String;)V** **flags: (0x0009) ACC_PUBLIC, ACC_STATIC** **Code:** **stack=2, locals=4, args_size=1** **0: iconst_1** **1: istore_1** **2: iconst_2** **3: istore_2** **4: iload_1** **5: iload_2** **6: iadd** **7: istore_3** **8: return**

Opcode'ları inceleyecek olursak;

iconst_1 (integer constant 1): Şekil 3.7'de gösterilen tamsayı sabitini işlenen yığına ekler (push).

Şekil 3.7. iconst_1 kullanımı *(Anouti, 2018)*.

istore_1: Şekil 3.8'de gösterilen en üstteki işlenen tamsayı (int değeri) çeker ve bu 1 indeksini dizinde a yerel değişkenine karşılık gelen yerde saklar.

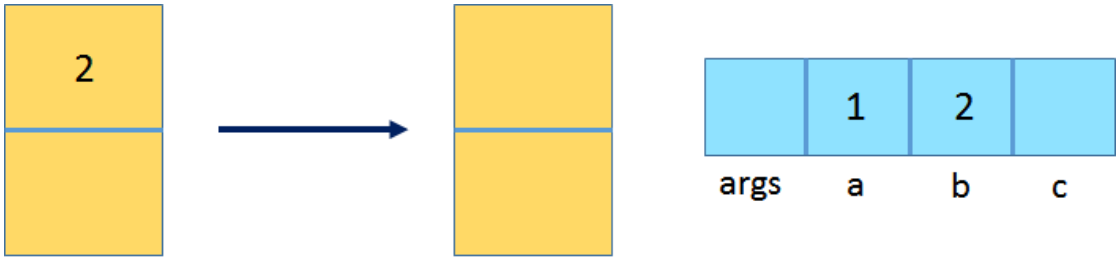


Şekil 3.8. istore_1 kullanımı (Anouti, 2018).

iconst_2 (integer constant 1): Şekil 3.9'da gösterilen tamsayı sabitini işlenen yığına ekler (push).

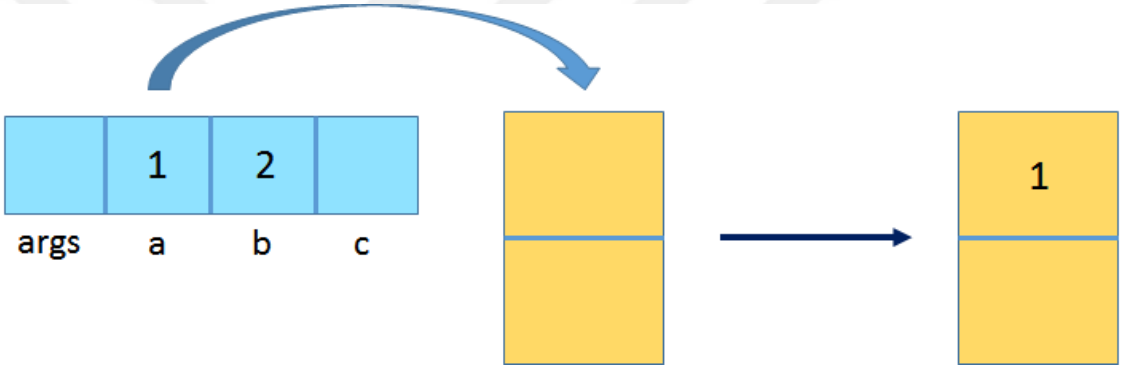
Şekil 3.9. iconst_2 kullanımı (Anouti, 2018).

istore_2: **Şekil** 3.10'da gösterilen üstteki işlenen tamsayı (int değeri) çeker ve bu 2 indeksini dizinde b yerel değişkenine karşılık gelen yerde saklar.



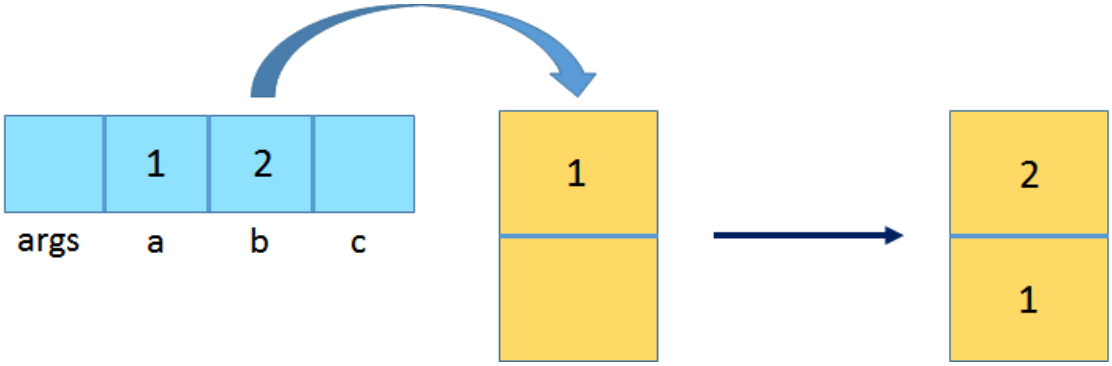
Şekil 3.10. istore_2 kullanımı (Anouti, 2018).

iload_1: **Şekil** 3.11'de gösterilen tamsayı değişkeni olan 1 indeksini yerel değişkenden alır ve işlenen yığına ekler.



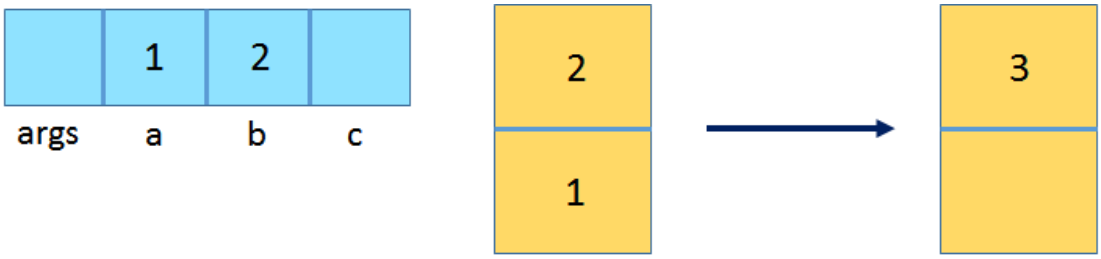
Şekil 3.11. iload_1 kullanımı (Anouti, 2018).

iload_2: **Şekil** 3.12'de gösterilen tamsayı değişkeni olan 1 indeksini yerel değişkenden alır ve işlenen yığına ekler.



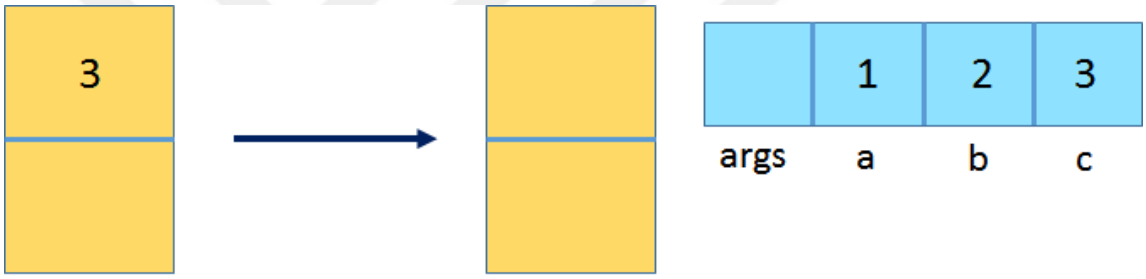
Şekil 3.12. iload_2 kullanımı (Anouti, 2018).

iadd: Şekil 3.13'de gösterilen yığındaki iki tamsayı değerini alır, birbirlerine ekler ve sonucu bir sonraki yığına yazar.



Şekil 3.13. iadd kullanımı (Anouti, 2018).

istore_3: Şekil 3.14'de gösterilen üstteki işlenen tamsayıyı (int değeri) çeker ve bu 3 indeksini dizinde c yerel değişkenine karşılık gelen yerde saklar.



Şekil 3.14. istore_3 kullanımı (Anouti, 2018).

return: void metodundan sonucu döndürür.

Tüm bu talimatların her biri sadece JVM tarafından yürütülecek işlemi tam olarak belirten bir opkod'dan (opcode) oluşur.

3.7. Java Agent

Java agent, Java Enstrümantasyon API'sini kullanarak, JVM'de çalışan uygulamalara müdahale ederek bayt kodları değiştirebilen özel bir java kütüphanesidir. Genel olarak, bir jar dosyası olarak hazırlanmıştır. Java agent'lar son derece basit ancak, çok güçlü ve aynı zamanda tehlikelidir (Balakrishan, 2020).

Java agent, JVM'de çalışan bir java programına izinsiz girilmesini sağlayan hizmetler vererek, en düşük seviyede çalışmayı hedefler. Java'nın bu güçlü ama esrarengiz kısmı, yanlış kullanılırsa JVM'yi çökertme özelliğine sahiptir (Debnath, 2020). Java agent'ları temsil eden sınıflar, Java API kütüphanesinde bulunan diğer sınıflardan başka bir şey değildir. Ancak onları özel yapan şey, java kodunun JVM'de çalışan başka bir uygulamayı engellemesine izin veren belirli bir kuralı takip etmeleridir. Burada tek amaç, basitçe bayt kodunu araştıran veya değiştiren agent'lar yapmaktır. Bu, java programının normalde yaptığının ötesine geçen güçlü bir özelliktir. Bir bakıma, bir programa girilebilir ve bayt kodunu değiştirebilir veya tahribat yaratılabilir. Bu Java'ya eklenen yeni bir teknoloji veya özellik değildir, JDK 1.5'ten beri kütüphanenin bir parçası olmuştur (Debnath, 2020).

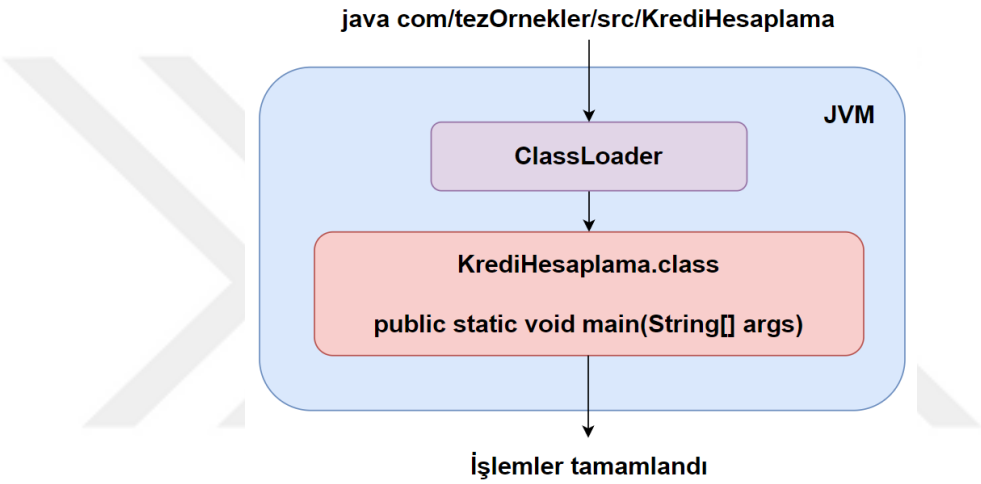
Java agent'lar, java sanal makinesinde çalışan programları kullanmasına izin veren hizmetler sunar. Bu hizmetler, java platformunda bulunan *java.lang instrument* isimli bir paket kullanmaktadır. Bu paket içerisinde; birkaç istisna sınıfı (exception class), bir veri sınıfı, sınıf tanımlayıcısı ve iki arayüz sınıfı (interface) gibi bilgileri içeren basit ve bağımsız yapılar bulunmaktadır. Eğer bu bir java agent yazılmak istenirse, bu iki arayüz sınıfından sadece *classFileTransformer* isimli arayüz sınıfının kullanılması önerilmektedir (Balakrishan, 2020).

Bir agent sınıfını tanımlamanın iki yolu vardır. Bunlardan ilki *static agent*'tır. Araç oluşturulup bir jar dosyası olarak paketlenmekte ve java uygulaması başlattıldığında java agent adlı özel bir sanal makinesine argüman olarak gönderilmektedir. Ardından, diskteki agent jar'ının lokasyonu verilir ve işleme başlatılır. JVM tarafından ilk yorumlama yapıldığında *-javaagent* isimli bir parametre kullanılarak agent statik olarak yüklenir. Statik agent'lar için kullanılan bu sınıfa *premain class* ismi verilmiştir. Bu yapı için premain yordamı iki parametre almaktadır. Bunlar, String veri türünde *agentArgs* argümanı ve Instrumentation türünde enstrümantasyon argümanıdır. Kullanıcı agentArgs'a istediği değeri atayabilir. Enstrümantasyon değeri ise, java.lang.instrument paketindendir ve agent'ların gerçek mantığını içeren yeni bir classFileTransformer nesnesi atanabilir (Balakrishan, 2020).

Agent sınıfı tanımlamanın ikinci yolu *dynamic agent*'tır. Dinamik agent'lar, uygulamayı başlatma şeklini denetlemek yerine, mevcut JVM'ye bağlanabilen ve

ona belirli bir agent'ın yüklemesini sağlayan küçük bir kod parçası yazmaktır. Bu kod *agentFilePath* argümanıdır ve statik agent yaklaşımıyla tamamen aynıdır. Agent jar'larının dosya adı olmalıdır, böylece hiçbir girdi akışı bayt olmaz. Bununla birlikte dikkat edilmesi gereken iki konu vardır. İlki, com.sun ortak alan kütüphanesinin altında bulunan özel bir API'nin var oluşu ve genellikle ornak erişim uygulaması olarak çalışmasıdır. İkincisi, java 9 ile yazılmış bir kodu jvm çalışırken ekleyemezsiniz (Balakrishan, 2020).

Bir java agent'ı anlamak için tipik bir java süreci şekil 3.15'de verilmiştir.

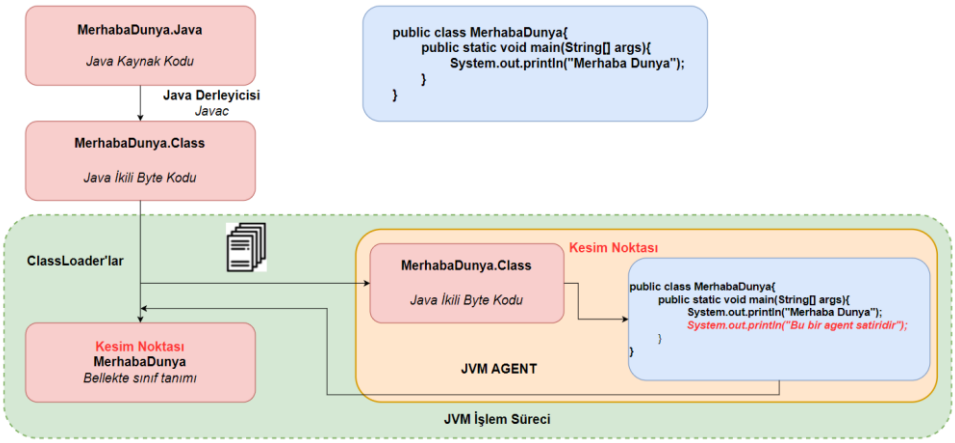


Şekil 3.15. Tipik bir java sürecinin adımları (Puls, 2014).

Java komutu, bir giriş parametresi olarak main yordamını içeren sınıfla çalıştırılır. Bu bir java çalışma zamanı ortamını (run-time environment) başlatır. Giriş sınıfının yüklenebilmesi için bir *Classloader* (sınıf yükleyici) kullanılır ve sınıf içerisinde main yordamını çağırır.

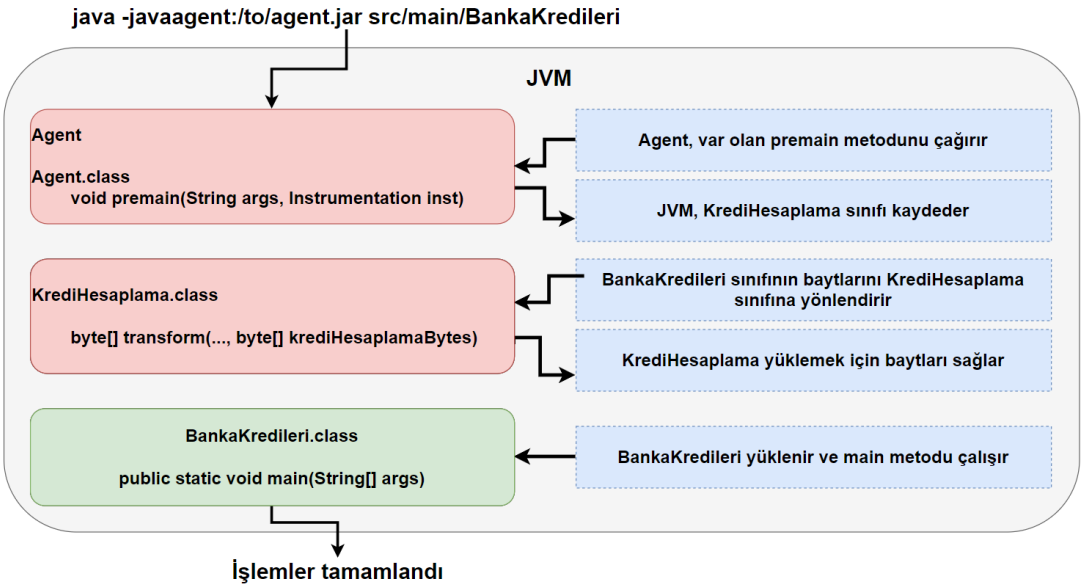
Şekil 3.16'da görüldüğü gibi, Sınıf yükleyicileri (Classloader) sınıfları binary ortamdan belleğe yüklemek için kullanılmaktadır. Derlenmiş bir .class uygulaması çalıştırıldığında, agent'lar çalışma zamanında sınıf yükleyicilerin davranışını engellemenin bir yolunu bulurlar. Java bayt kodunun nasıl yeniden yapılandırılacağı bilindikten sonra, agent'ların kod içerisinde doğru yerlere yerleştirilmesinin bilinmesi yeterli olacaktır. Java programları için bayt kodunun yapısı, orijinal java

programı kaynak koduna gerçekten yakındır. Bu nedenle, java programının kendisi kullanılmasa da onun çok yakın bir temsili kullanılmaktadır (Balakrishan, 2020).



Şekil 3.16. ClassLoader'ların çalışma mantığı (Debnath, 2020).

Java Agent kullanılan bir java sınıfının çalışma süreci Şekil 3.17'de gösterilmiştir.



Şekil 3.17. Java agent kullanan bir java sürecinin adımları (Puls, 2014).

İki girişli parametreye sahip olan java kodlarıyla çalışır. Birincisi, agent jar paketini işaret eden JVM argümanı -javaagent'tır. İkincisi ise, main yordamını içeren sınıftır. JavaAgent bayrağı, JVM'ye önce agent yüklemesini söyler. Agent'ın esas sınıfı, agent jar paketinin manifest'in de belirtilmelidir. Sınıf yüklendikten sonra, sınıftaki premain yordamı çağırılır. Bu premain yordamı, agent'lar için bir kurulum kancası

görevi olmaktadır (Puls, 2014). Agent'ın bir ClassTransformer kaydetmesine izin vermektedir. ClassTransformer, sınıfların dönüşümü için agent'ları uygulamak adına gerekli bir arayüz sınıfıdır. JVM'ye bir sınıf dönüştürücüsü (ClassTransformer) kaydedildiğinde, bu dönüştürü JVM'ye yüklenen sınıftan önce her sınıfın baytlarını alacaktır. Bu, sınıf dönüştürücüsüne bir sınıfın baytlarını gerektiği gibi değiştirme fırsatı verir. Sınıf dönüştürücüsü baytları değiştirdikten sonra, değiştirilen baytları JVM'ye geri döndürür. Bu baytlar daha sonra JVM tarafından doğrulanır ve yüklenir (Puls, 2014).

Sınıf dönüştürücüsünün kullanmış olduğu bazı parametreler bulunmaktadır. Bunlardan ilki, *ClassName*'dir. Bu parametrenin amacı, seçilmek istenen sınıfı belirtip diğerleri arasında ayrım yapılmasına yardımcı olmaktadır. Yani, koşullu bir ifade kullanarak istenen sınıfı diğer sınıflara karşı kontrol edilebilmektedir. Ardından *ClassLoader* temel uygulamalar için düz bir sınıf alanına sahip olmayan ortamlarda kullanılabilir. *ClassFileBuffer*, enstrümantasyondan önceki sınıfın geçerli tanımıdır. Bu tanımı engelmek için, kütüphane kodları kullanan bayt dizisinin okunması gerekir ve ardından kodu tekrar bayt koda dönüştürmek gerekir (Balakrishan, 2020).

Bayt kod oluşturmak için birkaç kütüphane bulunmaktadır. Tercih edilmeden önce yüksek seviyeli API mi yoksa düşük seviyeli API mi kullanılacağı hakkında karar verilmeli, boyut ve lisans açısından araştırılma yapılması gerekmektedir. Bunlardan en çok tercih edileni *Javassist*'tir. Sebebi, hem yüksek seviyeli hem de düşük seviyeli API'lerde dengeli bir şekilde kullanılmasıdır. Aynı zamanda, herkesin kullanımına açık bir lisanslamaya sahiptir. Böylelikle, ClassFileTransformer uygulamasının gövdesi için Javassist'in kullanılması idealdir.

Özetle, bir java agent uygulayabilmek için (Debnath, 2020);

- İki java sınıfı oluşturulması gerekir. Bunlardan biri premain metotlu bir sınıf (JavaAgent), diğeri ise ClassFileTransformer'ı kullanan bir başka sınıftır (CustomTransformer).
- Premain metodunun içerisinde, ClassFileTransformer sınıfına ait bir nesne oluşturularak kalıtımı gerçekleştirilmelidir.

- Ardından, CustomTransformer içindeki override edilen transform metodunun dönüşümünü gerektiren mantıksal kod yapısı geliştirilmelidir.
- Transform metodu içerisindeki bayt kod dönüştürülürken, kullanım amacına göre bayt kodu oluşturma kütüphaneleri kullanılması gerekir.
- Manifest'te ana sınıfın belirtilmesi ve jar dosyasının oluşturulması gerekmektedir.
- Durdurulmak istenen uygulamanın agent'ının yüklenmesi için javaagent etiketinin kullanılması gerekir.

3.8. Java Instrumentation API

Java agent, java enstrümantasyon API'nin bir parçasıdır. Enstrümantasyon API'leri, sınıf içerisindeki yordamların bayt kodlarını değiştirmek için bir mekanizma sağlar. Bu, hem statik hem de dinamik olarak yapılabilir. Bu durum, bir programın kaynak koduna dokunmadan kod eklenerek değiştirebileceği anlamına gelir. Sonuç, uygulamanın genel davranışı üzerinde önemli bir etkiye sahip olabilir. Java agent ve enstrümantasyon API'leri *java.lang.instrumentation* adlı pakette bulunur (Debnath, 2020). Bu API'ler aynı zamanda, izleme, analiz ve olay günlükleri gibi amaçları gerçekleştirebilir ve verileri yeniden bu amaçlar doğrultusunda derleyebilir. Uygulamalar hakkında hayati bilgiler edinmeyi sağlar, sorunları çözmeye veya verileri elde etmeye yardımcı olur (Blanco, 2010).

Mevcut bir sınıfın sahip oldugu bayt kodu değiştirebilmek için Java Instrumentation API'sinin bir parçası olan ClassFileTransformer interface sınıfının uygulanması gerekir.

Bir java agent oluşturulması için gereken adımlara Java Agent başlığı altında değinilmiştir. Özetle, bir yordam geliştirilir, Manifest.mf dosyası yapılandırılır ve ardından java agent'lı kütüphane sanal makineye bildirilir. Geliştirilen yordamda JVM'deki uygulamaların yürütülmesiyle ilgili bilgiler yer almalıdır. Bu durumda, enstrümantasyon API'si iki farklı yol sunmaktadır (Blanco, 2010).

premain: JVM'deki herhangi bir başka uygulamanın yürütülmesinden önce Java Agent'ın ve bununla ilişkili yöntemin yürütülmesine izin vermektedir. Bunun için JVM başlangıcında Java Agent'ının belirtilmesi gerekmektedir.
agentmain: JVM bir uygulama çalıştırmaya başladığında, java agent'ın çalıştırılmasına izin verir. Bu birlikte çalışma mantığı programlı bir şekilde ilerlemektedir.

Bir diğer önemli nokta da Java agent'ların geliştirmesi sırasında kaynak dizininin META-INF klasörüne bir Manifest.MF dosyası eklemesi gerektiğidir. Bu dosya, paket dağıtımıyla ilgili meta veri bilgilerini içerir. Aynı zamanda bu dosya, jar paketinin bir parçası olarak dahil edilir (Debnath, 2020). MANIFEST.MF dosyasının bir maven eklentisi aracılığıyla projeye jar dosyası olarak dahil edilebilmesine rağmen dosyanın manuel olarak yapılandırılması daha uygundur (Blanco, 2010).

Manifest.MF dosyalarında bulunan öznitelikler, bunun neden gerekli olduğuna dair bir ipucu vermektedir (Debnath, 2020). Nitelikler şöyledir:

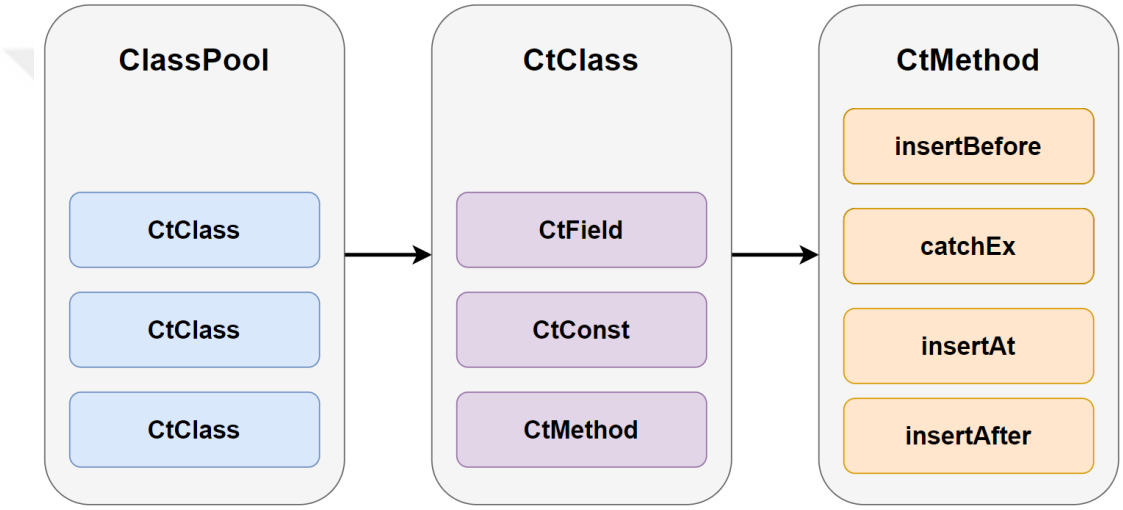
Premain-Class: premain yordamını uygulayan sınıfı belirtmek için kullanılır.
Agent-Class: JVM başladıktan sonra java agent'ların başlatma mekanizmasını tanımlar. Bu öznitelik tanımsızsa aracılar başlamaz.
Can-Redefine-Classes: Java Agent'ın müdahale ettiği sınıfları yeniden tanımlayıp tanımlayamayacağını belirtmek için kullanılır. Değeri true veya false olabilir.
Can-Retransform-Classes: Java Agent'ın müdehale ettiği sınıfı dönüştürüp dönüştürmeyeceğini belirtmek için kullanılır. Değeri true veya false olabilir.
Can-Set-Native-Method-Prefix: Agent tarafından yerel bir yordam öneki ayarlama yeteneğini tanımlar. Değeri true veya false olabilir.
Boot-Class-Path: Önyükleme sınıfı yükleyicinin arama yolu listesini tanımlar.

3.9. Javassist

Javassist, java'da bayt kodları düzenlemek için kullanılan bir kütüphanedir. Java programlarının çalışma zamanında yeni bir sınıf tanımlanmasını ve JVM'ye yüklendiğinde bir sınıf dosyasını değiştirmeyi sağlar. Diğer benzer bayt kodu

düzenleyicilerinden farklı olarak, Javassist iki düzey API sağlar. Bunlar, kaynak düzeyi ve bayt kodu düzeyidir.

Kullanıcılar kaynak düzeyinde API kullanıyorlarsa, java bayt kodunun özelliklerini bilmeden bir sınıf dosyasını düzenleyebilirler. Tüm API sadece java dilinin kelime hazinesi ile tasarlanmıştır. Girilen bayt kodunu kaynak metin biçimde bile belirtilebilir ve javassist tarafından anında derlenir. Diğer yandan, bayt kodu düzeyindeki API, kullanıcıların bir sınıf dosyasını diğer düzenleyiciler gibi doğrudan düzenlemesine olanak tanır.



Şekil 3.18. Javassist yapısı (Puls, 2014).

Şekil 3.18'de görüldüğü gibi, Javassist bir sınıfı temsil etmek için *CtClass* nesnesi kullanır. Bu CtClass nesneleri bir *ClassPool*'dan alınabilir ve sınıfları değiştirmek için kullanılır. ClassPool, bir HashMap[2] listesi olarak uygulanan CtClass nesnelerinin bir kabıdır. HashMap listeleri anahtar-değer (key-value) ilişkisiyle çalışmaktadır. Anahtar bilgisi sınıfın adı ve değer bilgisi ise sınıfı temsil eden CtClass nesnesidir. Varsayılan ClassPool temel alınan JVM ile aynı sınıf yolunu kullanmaktadır. Bu nedenle, bazı durumlarda bir ClassPool'a sınıf yolları veya sınıf baytları eklenmesi gerekebilir. Değişkenler, metotlar ve kurucu metotların içerdiği bir Java sınıfına benzer şekilde, bir *CtClass* nesnesi *CtFields* isminde değişkenleri, *CtConstructor*

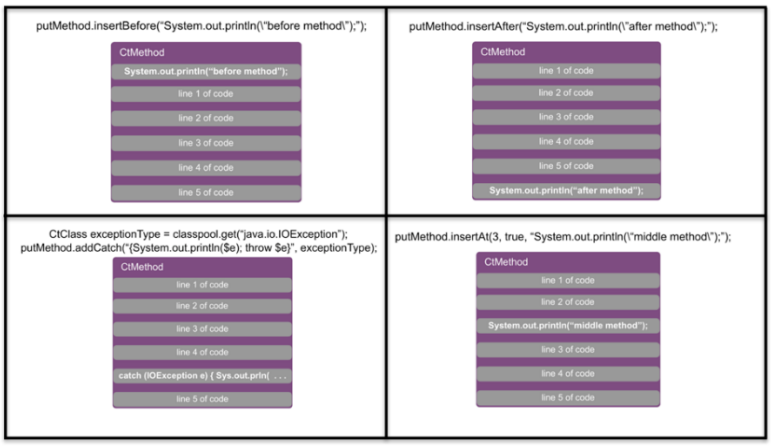
[2] Java HashMap, yazılım tarafında anahtar ve değer çiftleri olarak verileri depolamak için bir dizi oluşturmaktadır. Java Collection yapılarında harita tabanlı Map arayüz sınıfını kullanılmasıyla bilinir. Eşleşen anahtar-değer verilerine göre arama yapılabilir ve senkronize bir şekilde çalışmaktadır.

isminde kurucu yordamları ve *CtMethods* isminde metotları içermektedir. Bu nesnelerin tümü değiştirilebilir (Puls, 2014).

ClassPool'dan main yöntemiyle çalışmak istenilen classFileBuffer seçilebilir. Sınıf tanımlaması yapıldıktan sonra tüm yordamlar dolaşılabilir ve istenilen sınıf elde edilebilir. Bayt kod çalıştırmak yerine basit birkaç java kodu ile javassist kodu derler ve yeni bir bayt kodu oluşturabilir. Bir metoda java kodu eklemenin birkaç yolu bulunmaktadır.

- insertAfter(): gerekli kod gövdesinin sonuna istenilen bayt kodu eklenebilir.
- insertAt(): gerekli kod gövdesinin istenilen yerine bayt kodu eklebilir.
- insertBefore(): gerekli kod gövdesinin başına istenilen bayt kodu eklenebilir.

Şekil 3.19'da metot manipülasyonları için örnek verilmiştir ve Javassist'in esas avantajlarından birini göstermektedir. Aslında bayt kodu yazmaya gerek olmayabilir. Bunun yerine java kodu yazılabilir. Tek farkı, java kodunun tırnak içinde kullanılması gerekmektedir. Olumlu yanı, yazılan kod miktarı oldukça az ve Javassist'i kullanmak için aslında bayt kodu yazmaya gerek kalmamasıdır. En büyük dezavantajı, java kodunu tırnak içine almanın zorluğudur. Bayt kodu işleme çerçevelerinin hızları test edilerek proje için uygun olanı seçilebilir (Puls, 2014).



Şekil 3.19. Bir metodu değiştirmenin birkaç yolu (Puls, 2014).

Bayt kodunu değiştirmek için Javassist ile birlikte bazı Instrumentation API yöntemleri kullanılarak java agent'ı oluşturulabilir. Bu API içerisinde kullanılan bazı metotlar bulunmaktadır. Bunlar;

addTransformer: Enstrümantasyon motoruna bir dönüştürücü ekler.
getAllLoadedClasses: JVM tarafından o anki çalıştırılan tüm sınıfların bir dizisini döndürür.
retransformClasses: Bayt kodu ekleyerek önceden yüklenmiş sınıfların enstrümantasyonunu kolaylaştırır.
removeTransformer: Gerçekleştirilen dönüştürücünün kaydını siler.
redefineClasses: verilen sınıf dosyaları kullanılarak sınıf kümesi yeniden tanımlanır. Bu, sınıfın tamamen değiştirileceği, retransformClasses ile olduğu gibi değiştirilemeyeceği anlamına gelir.

3.10. JaCoCo - Java Code Coverage

Kod kapsamı (Code Coverage), Otomatik testler sırasında kodun kaç satırı yürütüldüğünü ölçmek için kullanılan bir yazılım ölçüm tekniğidir. Java projeleri için bir kod kapsamı raporlarını oluşturucusu olan JaCoCo, Java için ücretsiz bir kod kapsamı kütüphanesidir. Temel olarak 3 önemli ölçüm sağlar;

Lines Coverage (Satır Kapsamı): Testler tarafından çağırılan Java Bayt kod talimatlarının sayısına göre uygulanan kod miktarını yansıtır.
Branches Coverage (Şube Kapsamı): Tipik olarak if-else ve switch deyimleriyle ilgili olarka kodda kullanılan dallanma ifadelerinin yüzdesini gösterir.
Cyclomatic Complexity (Döngüsel Karmaşıklık): Doğrusal kombinasyon yoluyla bir koddaki tüm olası yolları kapsamak için gereken yol sayısını vererek kodun karmaşıklığını yansıtır.

Lines Coverage için en düşük puan %50 ile sınırlandırılır. JaCocCo, bir java agent olarak çalışır. Testleri çalıştırırken bayt kodunun enstrümantasyonundan sorumldur.

3.11. Mock ve Stub

Bir yazılımın testi yapılırken senaryoları test ekibini zorlamaktadır. Bunun nedeni, yazılım bir bütün olarak düşünüldüğünde bağımlılık kavramının ortaya çıkmasadır. Örneğin, yazılım bir veritabanına bağımlı olarak çalışıyor olabilir. Test senaryosunu oluştururken bu bağımlılığa dikkat edlimesi ve bu bağımlılık kullanılmadan test edilmesi gerekmektedir. Bu noktada, sahte nesnelere ihtiyaç duyulmaktadır. Birim testler yazılırken gerçek nesnelere bağımlı olmadan çalışabilmek adına, bunların yerine geçebilen sahte nesneler oluşturulur. Bu olaya Mocklamak (Mock-Mocking) denir.

Mock nesnesi, istenilen bir nesneyi birebir taklit etme özelliğindedir. Örneğin, veritabanı ile ilgili bir işlemde çalışan nesne yerine mock nesnesi kullanılabilir ve o işlemden beklenen sonuç mock nesnesi ile koda yönlendirilebilir. Mocklama tercih edilmediğinde, işlemler çok uzun sürebilir. Sayısı bilinmeyen testlerin olduğu düşünüldüğünde, yazılım her test edildiğinde çok uzun süreler boyunca beklenilebilir. Mock nesneleri işlemleri gerçekten gerçekleştirmediği için testlerin hızlı bir şekilde çalışmasını sağlamaktadır. Kısacası mock nesneleri, istenilen gerçek bir nesneyi birebir taklit eder ve test aşamalarında testçilere zaman kazandırır.

Metotlar içerisinde iki farklı senaryoya daha dikkat edilmiş ve projeye dahil edilmiştir. Bunlar, mock ve stub nesnelerdir. Bu nesneler birim testlerinin test ettikleri sistemi izole etmede kullanılan yöntemlerdir. Bu yöntemler, geniş anlamıyla test dublörleri olarak tanımlanabilir. Test dublörleri, test edilen sistemin bağımlı olduğu diğer birimlerin yerini tutar. Bu izolasyona birim testlerinde ihtiyaç duyulmasının temelde iki sebebi vardır. Birincisi, birim testleri, genelde test ettikleri sistemin kendisi ile ilgili varsayımları doğrulamak için yazılır. İkincisi, test dublörleri, davranış ve kullanım şekillerine göre çeşitlenir. Bunlardan en çok kullanılanları mock ve stub'tır. Bu çeşitliliğe sebep olan genel faktörler, bu dublörlerin beklenen işi yapıp yapmadığı ve yaparken nasıl bir davranış gösterdiği ile ilgilidir. Avantajı, genellikle kendi içlerinde doğrulama mekanizmaları bulundurmaları ve mock işlemi uygulanan nesne üzerindeki beklentilerin karşılanıp karşılanmamasına göre çağırıldıkları testin başarı durumunu etkilemeleridir. Dezavantajı ise, mock kullanılacak sistemde neredeyse her şeyin birer arayüz

(interface) üzerine inşa edilmesi gerekebilmesidir. Test ortamını kimi zaman karmaşıklaştırması da yine sistemi olumsuz etkileyebilmektedir. Çalışmada, mock ve stub kullanılabilecek örnek metotlar ve onlara ait olan sınıflar kullanılmıştır.

3.12. NoSQL

Bilgi, günümüzde elde ediliş biçimine bakılmaksızın kıymetli olarak görülmektedir. Bilgi, veriye dönüştürüldüğünde çok daha fazla önemli bir kaynak haline gelmektedir. Bu önemli kaynak ile stratejik birçok bilgi sistemi kurulabilmektedir. Tüm bu bilgi sistemlerinin özünde veri tabanı yönetim sistemleri kullanılmaktadır. Bilgiyi doğru bir şekilde kullanabilmek için onu nasıl şekilledirilmesi gerektiği bilinmelidir. Bir takım karar mekanizmaları kullanılarak bilgi veriye dönüştürülmektedir. Veri tabanı yönetim sistemleri ile birlikte bu hiyerarşi sürekliliğini korur ve aynı zamanda güvenilir bir ortamda depolanabilir. Bu süreçte sisteme yeni veriler eklenebileceği gibi var olan veriler belli aralıklarla yeniden düzenlenebilir. Bu düzenleme süreci tamamen verinin kaynağına bağlı olarak şekil değiştirebilir. Verilerin şekil değiştirilebilir olması zaman içerisinde karmaşıklıklara sebep olabilme ihtimali sunmaktadır. Bu sebepten veri tabanını yönetme şekli oldukça önem kazanmaktadır.

Veri tabanı yönetim sistemleri; tüm verileri toplu bir şekilde saklayabilmeli, saklanan bu verilerin erişim izinleri kontrol edilebilmeli, tekrar eden verilerden kaçınmalı, tabloların yapısal sorunları olmamalı, tablolar kurallara uygun bir şekilde tasarlanmalı, veri yedekleme mekanizmaları güçlü olmalı ve her zaman veri kontrol edilebilir bir şeffaflıkta olmaldır.

Veri tabanı yönetim sistemleri ikiye ayrılmaktadır. Bunlar, ilişkisel veri tabanı ve ilişkisiz veri tabanıdır.
İlişkisel veri tabanlarında, veriler arasında büyük ölçüde bir bütünlük söz konusudur. Herhangi bir güncellemede her bir veriye bu güncellemenin yansıması söz konusudur. Veriler arasında bu ilişkiyi takip edebilmek için kimlik numaralarına ihtiyaç duyulmaktadır. Bu numaralar aracılığıyla, verilerin bütünlüğünü koruyan

tablolar birbirleriyle iletişim halinde kalabilmektedirler. Sıklıkla tercih edilen bu sistemlere örnek olarak MsSql, MySql ve Oracle verilebilir.

İlişkisel olmayan veri tabanlarında, veriler arasında bir bütünlük söz konusu değildir ve bu sebeple veriler farklı şekillerde yinelenebilirler. Bu durum veri tutarsızlığına yol açmaktadır. Aynı verinin tüm sistem üzerinde farklı yerlerde değiştirilmesinin yönetilmesi zor olmaktadır. SQL arayüzü kullanılmadığı için ilişkisel olmayan anlamında No Reletion ismi verilmiştir. Böylelikle NoSQL ifadesi gündeme gelmiştir. Bu sistemlere örnek olarak MongoDB, HBase ve CouchDB verilebilir. Vaish, NoSql veri tabanı yönetim sistemleri hakkında dört temel özelliğe dikkat çekmiştir (Vaish, 2013).

- Şemasız veri gösterimi: İlişkisel veri tabanı yönetim sistemlerinde önce bir şema oluşturulur. Tablo ve sütunlar öncelikle tasarlanır, hemen ardından veriler sisteme satır olarak kaydedilir. Bu hiyerarşi mantığı ilişkisel olmayan veri tabanları sistemlerinde yani NoSql yapısında yer almamaktadır. Bir yapı tanımlamak için detaya ihtiyaç duyulmamaktadır. Örneğin, JSON yapısında yeni alanlar ekleme ve hatta verileri iç içe yerleştirme gibi özellikler sonradan yapıya dahil edilebiliyor.
- Geliştirme zamanı: Karmaşık sql sorgularından olabildiğince uzak bir yapıya sahiptir.
- Hızlılık: Özellikle mobil ve diğer bağlantısı kesintili cihazlar üzerinden, sahip olunan az miktarda veriye bile milisaniyede erişilebilmektedir.
- Ölçeklenebilirlik: NoSql yatay olarak ölçeklendirildiği için sistemlerin sürekli artan bir RAM ihtiyacı bulunmaktadır. Bu da ciddi bir maliyeti beraberinde getirmektedir.

3.12.1. MongoDB veritabanı sistemleri

NoSql alanında birçok veri tabanı sistemi bulunmaktadır. Bunlardan biri de bu çalışmada tercih edilen MongoDB'dir.

MongoDB veri tabanı sistemleri C++ programlama dili kullanılarak geliştirilmiştir. C temelli bir yapıya sahip olan bu sistemler hız ve ölçeklenebilirlik özellikleriyle verileri analiz, yorumlama ve yönetme gibi işlemleri rahatlıkla gerçekleştirebilmektedir (Chodorow, 2013).

Sistem içerisindeki kayıtlar JSON (JavaScript Object Notation – JavaScript Nesne Gösterimi) yapısında doküman haline getirilmektedir. Verileri depolayan JSON yapısı, farklı platformlarda kullanılmak için bir standart olarak kullanılır. Standart bir yapı olması birçok NoSQL arayüzünde rahatlıkla kullanılmasını da tetiklemektedir. Bu standart yapı verileri birer nesne olarak görmekte ve bunları sıralı listeler halinde tutmaktadır. Her bir nesneyi (veriyi) betimleyen bir anahtar kelimeden (etiketten) oluşmaktadır. Çoğu programlama dillerinde karşılaşılan listeleme yapılarına benzer şekilde anahtar-değer (key-value) ilişkisi bulunmaktadır. Bu sebepten dolayı, ilişkisel veri tabanındaki gibi tablo-sütun-satır hiyerarşisi bulunmayıp, onun yerine çok daha esnek bir yapıyla karşılaşılmaktadır. Öyle ki, bir JSON koleksiyon listesinde farklı anahtar kelimeler, farklı alanlar ve hatta farklı dökümanlar bulunabilmektedir. Sürekli birbirini takip eden bir listeleme yapısı yoktur.

MongoDb'nin hız ve ölçeklenebilirlik performanslarının yanı sıra sorgu yapısı, sürücü desteği ve paylaşım gibi konularda da başarılı olduğu gözlemlenmektedir.

- Sorgu Yapısı: Standar SQL cümleleri haricinde, MongoDB tarafında NoSQL kullanılarak anahtar kelimelerle sorgulama işlemi yapılabilmektir.
- Sürücü Desteği: C temelli bir yapıya sahip olduğu için; C, C++, C#, Java, Php, Python gibi programlama dillerinde desteği sağlanmaktadır.
- Paylaşım: JSON Yapısıyla birçok json temelli arayüzde kullanılabilecek bir koleksiyona sahiptir. Aynı zamanda büyük hacimdeki verilerin sunuculara paylaştırılmasını sağlamaktadır.

3.13. Maven

Apache tarafından geliştirilmiş Maven, bir JDT (Java Development Tool) yani java geliştirici aracıdır. Maven java projelerini geliştirirken proje içerisinde bir standart oluşturmayı sağlar. Bu standartlar doğrultusunda, proje geliştirme sürecini

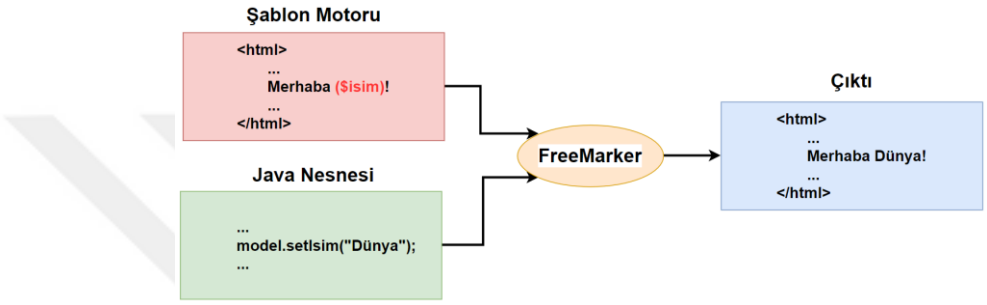
basitleştirir ve dökümantasyonu etkili bir şekilde oluşturmayı sağlar. Proje içerisindeki kütüphane ve kullanıcı arayüzüne olan bağımlılığı ortadan kaldırmaya yarayan, derleme ve raporlama gibi işlemlerde geliştirici için kolaylıklar sağlayan bir araçtır. Aslında bir araçtan ziyade kendisi bir depolama birimidir. Geliştirilen yazılımla ilgili tüm kütüphane, eklenti ve gerekli tüm bilgileri sunucularda çalıştırabilme şansı bulunmaktadır.

Projede kullanılacak tüm kütüphaneler ve eklentiler POM (Project Object Model) isimli bir dosyadan yönetilebilmektedir. Bu da Maven'in çalışma mantığı oluşturmaktadır. Maven, gerekli tüm kütüphaneleri kendi sunucularında barındırmaktadır. POM içerisinde belirtilen kütüphane dosyaları öncelikle projenin derlendiği yerel makinanın klasörlerinde var mı yok mu Maven tarafından kontrol edilir. Arama sonucunda yerel makinede herhangi bir kütüphaneye erişemezse Maven kendi sunucularında bu kütüphaneleri aramaya başlar. Eğer kendi sunucularında da bu kütüphanelere erişemezse, geliştirici tarafından tanımlanan bir sunucu adresinde dosyayı arar ve yerel makineye dosyayı indirir. Sonrasında bunu proje içerisinde kullanabilmeyi sağlar. Ayrıca, var olan bir kütüphane başka kütüphanelere bağımlı bir şekilde çalışıyorsa o kütüphaneleri de sisteme indirir ve projeye dahil eder. Bu işlemleri sadece bir kez yapmaz proje her derlendiğinde kütüphaneleri güncelleyerek çalışmaya devam eder.
POM dosyası kütüphane ve eklentileri içerebildiği gibi aynı zamanda proje hakkında dökümantasyon bilgilerini de içerebilmektedir. Aynı zamanda bu dosya sayesinde projenin derleme ya da yayınlama işlemleri birkaç satır kodla yapılandırılabilir. POM dosyası içerisinde projenin grup, yapı ve sürüm numaraları gibi ifadeler de bulunmaktadır. Kısacası geliştirilen proje hakkında bilgi edinebilmek için POM dosyası yeterli sayılmaktadır. Böylelikle Maven'in bir nevi proje yöneticisi olduğu söylenebilir. Çalışmada maven kullanılarak, tüm proje boyunca gerekli olan kütüphaneleri çalıştırılması ve her defasında güncellenmesi sağlanmıştır.

3.14. FreeMarker Java Template Engine-FTL (Java Şablon Motoru)

Apache tarafından üretilmiş bir şablon motorudur. Şablonlara ve değişen verilere bağlı olarak HTML web sayfaları, elektronik postalar, yapılandırma dosyaları ve

kaynak kodlar gibi metin çıktısı üretebilen bir Java kütüphanesidir. Kendi başına son kullanıcılar için bir uygulama değil, programcıların ürünlerine yerleştirebilecekleri bir araçtır. Şablonlar, tam gelişmiş bir programlama dili olmayan ancak basit ve özel olan FTL yani FreeMaker şablon dili ile yazılmaktadır. Genellikle, verileri hazırlamak için java gibi genel amaçlı kullanılan bir programlama dili kullanılır. Ardından, FTL şablonları kullanarak hazırlanan verileri görüntüler. Şablonlarda verilerin "nasıl" sunulacağına ve şablon dışında da "hangi" verilerin sunulacağına odaklanılmaktadır (FreeMarker, 2021).



Şekil 3.20. FreeMarker Java Template Engine yapısı (FreeMarker, 2021)

Bu yaklaşım, genellikle MVC (Model View Controller) yapısına benzer ve özellikle dinamik web sayfaları için tercih edilmektedir. Web sayfası tasarımcılarını, genellikle java programcılarından ayrımaya yardımcı olan bir yapıdır. Tasarımcılar şablonlarda karmaşık mantıkla karşılaşmazlar, yazılımcılar kodu değiştirmek veya yeniden derlemek zorunda kalmadan bir sayfanın görünümünü değiştirebilirler (FreeMarker, 2021).

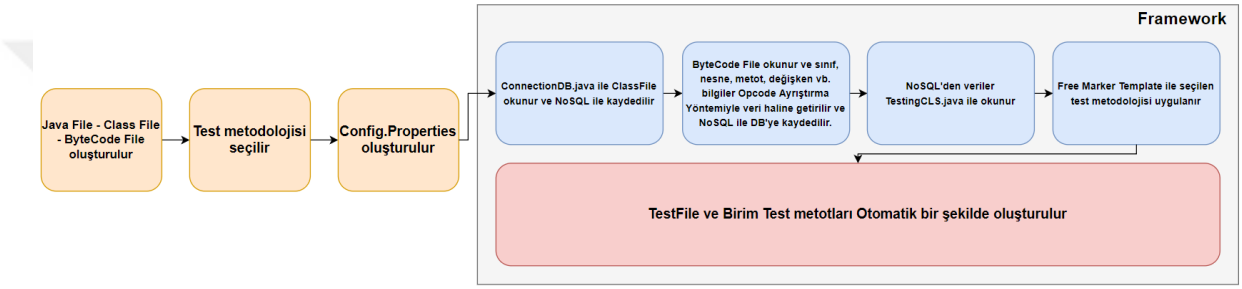
FreeMaker başlangıçta MVC web uygulama çerçevelerinde HTML sayfaları oluşturmak için oluşturulmuş olsa da sunucu uygulamalarına veya web ile ilgili herhangi bir şeye bağlı değildir. Web dışı uygulama ortamlarında da kullanılmaktadır (FreeMarker, 2021).

Güçlü bir şablon dili olan FTL'de; koşullu bloklar, döngüler, dize ve aritmetik gibi birçok işlem tasarlanabilir. Aynı zamanda, veri tabanı gibi verilerin depolandığı bir yerden çektiği verileri şablon içerisinde çıktı olarak verebilir. Kısacası FTL, çok yönlü bir veri modelidir.

4. ARAŞTIRMA BULGULARI

Bu bölümde, tez çalışmasında kullanılan yöntemler ve gerçekleştirilen işlemlerin detayları anlatılmıştır. Çalışmada, gerçek zamanlı olarak verilerin toplanmasıyla otomatik birim testlerin oluşturulmasını sağlayan bir framework geliştirilmiştir. Söz konusu araç, Netbeans ortamında Java programlama dili kullanılarak konsol ve masaüstü uygulama olacak şekilde geliştirilmiştir.

Geliştirilen çalışma ile ilgili akış şeması Şekil 4.1'de verilmiştir.



Şekil 4.1. Gerçek Zamanlı Veri Toplarak Otomatik Birim Test Oluşturma Yazılımına ait akış şeması

4.1. Gerçek Zamanlı Veri Toplayarak Otomatik Birim Testi Oluşturma

Geliştirilen framework; 4 bileşenden oluşmaktadır. Bunlar, Java programlama diline ait kodların bulunduğu .java ve .class dosyalarının okutulduğu modül, eş zamanlı olarak kodlardan okutulan değişken, nesne, yöntem ve sınıf isimlerinden oluşan tüm verilerin NoSQL ile veri tabanına kaydedildiği modül, java kodlarının bayt koda dönüştürüldüğü modül ve son olarakta bayt kod ve gelen veriler yardımıyla otomatik oluşturulan birim test sınıfına ait dosyanın oluşturulmasıyla ilgili modüldür.

4.1.2. Birim Testlerde Gereksinimler ile İlgili Senaryoların Tanımlanması ve Analizi

Yazılımların beklenen şekilde çalıştırılması için denetlenmesi gerekir. Bu denetleme aşamasında, yazılım içerisinde değişken veya yöntemler gibi en küçük test edilebilir birimlerin tek tek ve bağımsız olarak doğru bir şekilde çalışıp çalışmadığı analiz

edilmelidir. İşte bu noktada, her bir birim için oluşturulan testlerden gelen davranışların ayırt edilmesi Birim Test'ler (Unit Test) tarafından yapılmaktadır. Birim testler, yazılımın ilk oluşturulduğu evrelerde yapılır ve tüm test edilecek birimlerin birleştirilmesinden önce tamamlanmış olmalıdır. Birim testleri oluşturulurken bazı kurallara dikkat edilmesi gerekir. Bunun için öncelikle birim testlerini oluşturmak, çalıştırmak ve bu testlerin davranış sonuçlarını analiz edip raporlayabilmek için bir birim test çerçevesine (Unit Test Framework) ihtiyaç duyulmaktadır.

Herşeyden önce test odaklı bir yazılım geliştirme yapılıyorsa, ilgili algoritmaya ait kodu yazmadan önce bununla ilgili basit seviyede bir test senaryosu tasarlanır ve birim testlerin kodlanması gerekir. En küçük birimlerin test edildiği yapılarda, test metotlarının isimlendirilmesine dikkat edilmelidir. Verilen bu isimlerin okunaklı, anlaşılabilir ve sürdürülebilir bir yapıya sahip olması gerekir. Test edilen her bir kısım bağımsız olmalıdır. Bağımsız bir şekilde en küçük birimin test edilmesi demek, testlerin daha hızlı çalışabilmesi ve sonuçların daha hızlı bir şekilde raporlanmasıdır.

Tüm bu gayrete rağmen birim testler tüm hataları ortaya çıkarmazlar. Çünkü, her birim ayrı ayrı test edilmektedir ve yazılımın bütünüyle bir ilişkisi yoktur. En son entegrasyon işlemleri gerçekleştirildiğinde yazılımın düzgün çalışıp çalışmaması da yine birim testleri ilgilendiren bir konu değildir. Bu işlev tamamen, ilgili birimlerin kontrolü ve doğruluğu için kullanılmaktadır.

Birim test çerçevelerinin her birine ait standart bir yapısı bulunmaktadır. Bu çalışma, java programlama dili üzerine kurgulandığı için java tabanlı kodların test edilebilmesi için JUnit çerçevesi kullanılmıştır ve bu kapsamda geliştirilme yapılmıştır.

Bu bölümün ilerleyen başlıklarında bahsedildiği üzere, çalışma içerisinde birçok senaryoya odaklanılmıştır. Özellikle koşul veya döngü yapılarını barındıran metotların test edilmesi için olasılık test metodları oluşturulmuştur. Aynı zamanda mock ve stub yapıları da çokça kullanılmıştır. Tüm bu yapılarla ilgili test aşamalarını gerçekleştirmek için otomatik birim test oluşturacak bir çerçeve geliştirilmiştir. Bu çerçevede, otomatik oluşturulan test metotları ve sınıflarının analizlerini yapabilmek

için yazılım içerisinde kullanılan veriler örnek olarak alındığı gibi, aynı zamanda rastgele veriler de üretilerek birer değişken olarak kullanılmıştır.

4.1.2.1. Method içerisinde Alternatif Durumlar

Birim test yazarken bir sınıfa ait metodun yalın olduğu durumlarda tek bir birim test senaryosu oluşturulmalıdır. Ancak metot içerisinde birden fazla durumun olduğu senaryolar çok sık karşılaşılan bir durumdur. Şekil 4.2'de basit oluşturulmuş bir sınıf ve ona ait bir metot görülmektedir.

```
package edu.sdu.ornek.proje;

public class KrediHesaplama {
    public double faizHesapla(double anaPara, double oran){
        return anaPara/100*oran;
    }
}
```

Şekil 4.2. Basit seviyede oluşturulmuş bir sınıf ve metot yapısı

KrediHesaplama sınıfında bulunan *faizHesapla* isimli metot anapara ve oran isimli iki parametre almaktadır. Dönüş değeri olarak belli bir hesaplama sonucunu vermektedir. Buna ait oluşturulabilecek örnek birim test kodu ise Şekil 4.3'de verilmiştir.

```
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.*;
import org.junit.Test;
import edu.sdu.ornek.proje.KrediHesaplama;

public class KrediHesaplamaTest {
    KrediHesaplama krediHesaplama = new KrediHesaplama();
    @Test
    public void faizHesaplamaTest(){
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 1.20);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.3. KrediHesaplama sınıfına ait test kodları

Şekilde görüldüğü gibi, önce KrediHesaplama sınıfından bir nesne oluşturulmuştur. Sonrasında ise, JUnit kütüphanesinde kullanılan assertEquals fonksiyonuna test edilmesi gereken metot için beklenen ve gerçek değerler paremetre olarak verilmiştir. Bu test kodu normal bir şekilde çalıştırılmış ve gerekli sonuca ulaşılmıştır. Ancak bir metot içerisinde birden fazla durumun yönetildiği yöntemler bulunmaktadır. Şekil 4.4 buna örnek olarak verilmiştir. Metot içerisinde faiz oranının 1'den az olması durumunda oranın en düşük 1.0 değerini almasını sağlayan bir durum bulunmaktadır. Yani bir koşul söz konusudur (if bloğu).

```
package edu.sdu.ornek.proje;

public class KrediHesaplama {
    public double faizHesapla(double anaPara, double oran){
        if(oran < 1.0 ){
            oran = 1.0;
        }
        return anaPara/100*oran;
    }
}
```

1

Şekil 4.4. İçerisinde koşul bulunan bir metot yapısı

Koşul yapıları söz konusu olduğunda, her bir koşul olasılığını karşılayacak bir birim testine ihtiyaç bulunmaktadır ve bunlar için alt senaryolar geliştirilmesi gerekir. Bu durumda her iki senaryoyu karşılayacak tek bir birim testi olmayacağından Şekil 4.5'de görüldüğü gibi iki ayrı birim testin oluşturulması gerekmektedir.

```java
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.*;
import org.junit.Test;
import edu.sdu.ornek.proje.KrediHesaplama;

public class KrediHesaplamaTest {
    KrediHesaplama krediHesaplama = new KrediHesaplama();
    @Test
    public void faizHesaplamaTest(){
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 1.20);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
    @Test
    public void faizHesaplamaBirAltiTest(){
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 0.80);
        assertEquals(1.00, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.5. İçerisinde koşul bulunan bir metodun birim testlerine ait metot yapıları

Bu iki durumu tek bir test yönteminde yönetmek bir anti-pattern[3] olarak bilinir. Sebebi, birinci durumun hata vereceği durumlarda ikinci durum çalışırsa testin sonucundan tam verim alınamaz. Bu durumda birbirinden bağımsız iki testin olması beklenmektedir. Bu duruma çözüm olarak, metotlar içerisinde farklı senaryolar ayrı birim testler ile karşılanmalıdır.

4.1.2.2. Method İçerisinde Farklı Nesneler

Birim testi yazarken metotlar içerisinde farklı nesneler yer alabilmektedir. Şekil 4.6'da gösterilen örnek metotta her bir hesaplama için hesaplamaları arşivlemeye yarayan *HesaplamaKayit* isimli bir kayıt sınıfı oluşturulmuştur. Buna göre, bu *faizHesapla* metodunun *HesaplamaKayit* sınıfına ait bir bağımlılığı bulunmaktadır. Yani *hesaplamaKayıt* sınıfının hazır bir şekilde var olması beklenir. Çünkü, bu metot içerisinde *HesaplamaKayit* için oluşturulmuş nesneye ait *hesaplamayiKaydet* isimli bir alt metot bulunmakta ve iki parametre almaktadır. Özetle, *faizHesapla* metodunun *hesaplamayiKaydet* isimli metot ile bir bağımlılığı bulunmaktadır.

[3] Anti-pattern, genellikle etkisiz olan ve son derece verimsiz olma riski taşıyan, yinelenen bir soruna verilen ortak bir isimdir.

```java
package edu.sdu.ornek.proje;

public class KrediHesaplama {
    public HesaplamaKayit hesaplamaKayit;
    public double faizHesapla(double anaPara, double oran){
        if(oran < 1.0 ){
            oran = 1.0;
        }
        hesaplamaKayit.hesaplamayiKaydet(anaPara, oran);
        return anaPara/100*oran;
    }
}
```

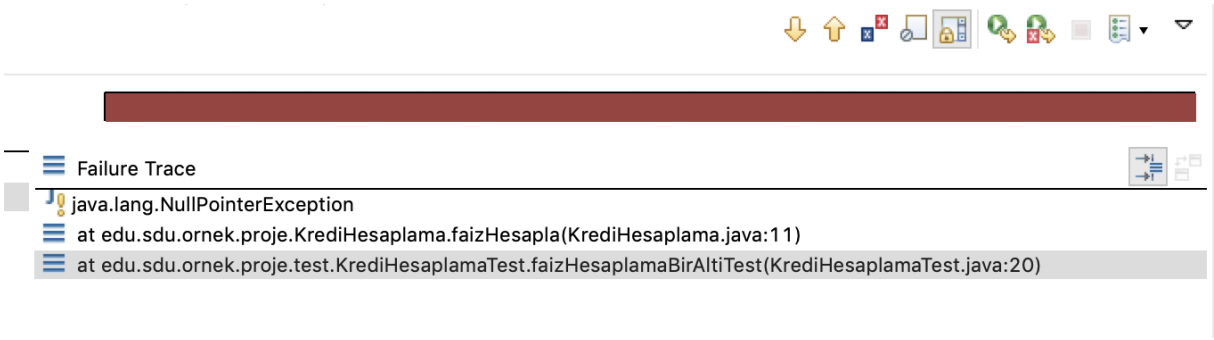
Şekil 4.6. İçerisinde farklı nesne bulunan bir metot yapısı

Bu durumda yazılabilecek en basit test metodu Şekil 4.7'de verilmiştir.

```java
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.*;
import org.junit.Test;
import edu.sdu.ornek.proje.KrediHesaplama;

public class KrediHesaplamaTest {
    KrediHesaplama krediHesaplama = new KrediHesaplama();
    @Test
    public void faizHesaplamaTest(){
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 1.20);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.7. İçerisinde farklı nesne bulunan bir metodun olası birim test metot yapısı

Şekil 4.7'deki birim test çalıştırıldığında Şekil 4.8'de görüldüğü gibi bir NullPointerException hatası alınır.



Şekil 4.8. Farklı nesneler için geliştirilmiş bir birim testin vermiş olduğu hata

NullPointerException hatası, faizHesapla metodu içerisindeki *hesaplamaKayıt* nesnesinin içeriğinin bulunmamasından dolayı meydana gelmiştir. Yani instance (nesne) oluşturulmamıştır. Bu hata iki durumla yönetilebilir. İlki Stub yöntemi, diğeri ise Mock yöntemidir.

4.1.2.2.1. Stub yöntemi

Stub yöntemi, test sırasında yapılan çağrılara hazır yanıtlar sağlamaktadır. Genellikle test için yazılmış durumların dışında hiçbir şeye yanıt vermez. Şekil 4.9'da görüldüğü gibi, Metot içerisindeki nesne çağrılır. Bunun için de *krediHesaplama.hesaplamaKayit = new HesaplamaKayit();* satırının eklenmesi gerekir.

```
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.*;
import org.junit.Test;
import edu.sdu.ornek.proje.HesaplamaKayit;
import edu.sdu.ornek.proje.KrediHesaplama;

public class KrediHesaplamaTest {
    KrediHesaplama krediHesaplama = new KrediHesaplama();
    @Test
    public void faizHesaplamaTest(){
        krediHesaplama.hesaplamaKayit = new HesaplamaKayit();
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 1.20);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.9. Stub yönteminin kullanılması

Bu yönetilebilmesi kolay olmayan bir seçenektir. Çünkü, *HesaplamaKayit* sınıfına bağlı alt bileşenler ve kolay oluşturulamayacak bir nesne yapısı yer almaktadır. Tüm bunların yanında, Spring ve EJB gibi içerik bağımlı bileşenler de olabilir. Dolayısıyla, tüm bu dezavantajlar nedeniyle bu nesnelerin birer karşılığını oluşturmayı sağlayan stub yöntemi mümkün olduğunca tercih edilmemektedir.

4.1.2.2.2. Mock yöntemi

Mock yöntemi sahteleme işlemi yapan bir yöntemdir. Bu yöntem, bir nesnenin gerçek varlığı olmadan normal bir şekilde çalışıyormuş gibi hizmet vermesini sağlayan bir yapıya sahiptir. Yani gerçek yöntemlerin ve nesnelerin yerini boş yöntem ve nesneler alır. Şekil 4.10'da görüldüğü gibi, Mocklama yapabilmek için JUnit ile birlikte Mockito kütüphanesi kullanılmaktadır.

```
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.*;
import org.junit.Test;
import org.mockito.Mockito;
import edu.sdu.ornek.proje.HesaplamaKayit;
import edu.sdu.ornek.proje.KrediHesaplama;

public class KrediHesaplamaTest {
    KrediHesaplama krediHesaplama = new KrediHesaplama();
    @Test
    public void faizHesaplamaTest(){
        krediHesaplama.hesaplamaKayit = Mockito.mock(HesaplamaKayit.class);
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 1.20);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.10. Mock yönteminin kullanımı

Bu örnekte eklenen,

krediHesaplama.hesaplamaKayit = Mockito.mock(HesaplamaKayit.class);

satırı o nesneyi mocklamak için kullanılmıştır. Nesneye ait bir class dosyasının varlığı mock metoduna paremetre olarak tanımlanmıştır. Buna göre, artık o nesne içerisinde yer alan kaydet metodunun içeriği çalıştırılmayacaktır.

Dezavantajı, *HesaplamaKayit* sınıfı ile bir entegrasyon yapılmamaktadır. Ancak birim testler doğası gereği alt birimler ile olan entegrasyonlarla ilgilenmezler. Dolayısıyla, bu tarz testlerin entegrasyon testleri içerisinde yapılması beklenir.

Mocklamayı yönetmek için annotation[4] tabanlı kullanımlar da mevcuttur. Mockito kütüphanesi, direkt olarak nesneye annotation'lar ekleyerek mocklanmasını sağlayan bir yönteme sahiptir. Şekil 4.11'de görüldüğü gibi, test sınıfında annotation'lar tanımlanırsa, kod daha da okunabilir ve özet bir hal almış olacaktır.

```
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import edu.sdu.ornek.proje.HesaplamaKayit;
import edu.sdu.ornek.proje.KrediHesaplama;

@RunWith(MockitoJUnitRunner.class)
public class KrediHesaplamaTest {
    @InjectMocks
    KrediHesaplama krediHesaplama;
    @Mock
    HesaplamaKayit HesaplamaKayit;

    @Test
    public void faizHesaplamaTest(){
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0, 1.20);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.11. Mock yönteminin annotation'lu kullanımı

Böylelikle, annotationlar içerisinde hem *KrediHesalama* sınıfı hem de *HesaplamaKayit* sınıfı tanımlanmış ve bunlardan birer nesne üretilmiştir. Sonrasında test metodları içerisinde bu nesneler kullanılmıştır.

4.1.2.3. Mocklanmış Nesnelerde Değer Dönüşümleri

Mocklanan bir nesne içerisinde metotlar veya nesneler varsayılan değerlerini taşıdığından zaman zaman gerçek birim test ortamı yaratılamaz. Şekil 4.12'de görüldüğü gibi, aylık oran bilgisi *faizOran* isimli farklı bir sınıftan çağrılmaktadır.

[4] Java bilgisayar programlama dilinde açıklama belirtmek için kullanılmaktadır. Java kaynak koduna eklenebilen bir sözdizimsel meta veri biçimidir. Sınıflar, yöntemler, değişkenler, parametreler ve Java paketleri açıklamalı olabilir.

```java
package edu.sdu.ornek.proje;

public class KrediHesaplama {
    FaizOran faizOran;
    public double faizHesapla(double anaPara){
        return anaPara / 100 * faizOran.aylikOran();
    }
}
```

Şekil 4.12. Başka bir sınıftan değer dönderen bir metot yapısı

Faiz oran sınıfına ait *aylikOran* isimli metot yapısı Şekil 4.13'de verilmiştir.

```java
package edu.sdu.ornek.proje;

public class FaizOran {
    public double aylikOran(){
        return 1.20;
    }
}
```

Şekil 4.13. Değer dönderen bir metot yapısı

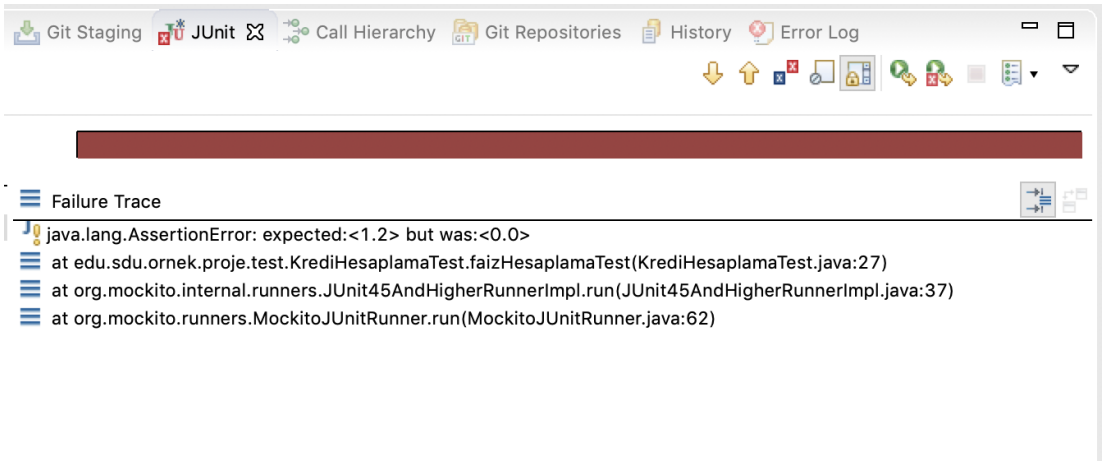
Bu durumda, *aylikOran* metodundan dönen 1.20 değeri *KrediHesaplama* sınıfındaki *faizHesapla* metodu içerisindeki hesaplamaya dahil olmuştur.

Şekil 4.14'de *faizHesapla* metoduna ait mock yöntemi kullanılarak hazırlanan bir test sınıf ve metodu yer almaktadır.

```java
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import edu.sdu.ornek.proje.FaizOran;
import edu.sdu.ornek.proje.KrediHesaplama;

@RunWith(MockitoJUnitRunner.class)
public class KrediHesaplamaTest {
    @InjectMocks
    KrediHesaplama krediHesaplama;
    @Mock
    FaizOran faizOran;

    @Test
    public void faizHesaplamaTest(){
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.14. Başka bir sınıftan değer döndüren nesnenin mocklanmış hali

Bu birim testi Şekil 4.15'de olduğu gibi AssertionError hatasını vermektedir. Bu hata beklenen ve gerçek değerlerin kıyaslanması sonucu farklı bir sonucun oluştuğunu göstermektedir. Sonuç olarak, 1.2 beklenen sonucun 0.0 geldiğini göstermektedir.



Şekil 4.15. Başka bir sınıftan değer döndüren nesnenin mocklanmış haline gelen hatalı test sonucu

Bunun sebebi, Şekil 4.16'da verilen bir metot aslında mocklandığında yerini Şekil 4.17'de ki gibi metota bırakmaktadır.

```
public double aylikOran(){
        return 1.20;
    }
```

Şekil 4.16. Değer döndüren bir metot yapısı

```
public double aylikOran(){
        return 0.00;
    }
```

Şekil 4.17. Değer döndüren bir metot yapısı

Mocklama yöntemi bir nesneyi sahtelediğinde içerisindeki yöntem gövdelerini ortadan kaldırmaktadır. Bunu yönetebilmek adına, mocklanan nesnelere belirli bazı durumlar atanır. Şekil 4.18'de, *FaizHesaplama* yöntemi içerisindeki *faizOran*'ın bu birim test için 1.20 dönmesi koşulu sağlanmaktadır. Bunu gerçekleştirebilmek için when-thenReturn yapısı kullanılmıştır.

```
package edu.sdu.ornek.proje.test;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import static org.mockito.Mockito.*;
import org.mockito.runners.MockitoJUnitRunner;
import edu.sdu.ornek.proje.FaizOran;
import edu.sdu.ornek.proje.KrediHesaplama;

@RunWith(MockitoJUnitRunner.class)
public class KrediHesaplamaTest {
    @InjectMocks
    KrediHesaplama krediHesaplama;
    @Mock
    FaizOran faizOran;

    @Test
    public void faizHesaplamaTest(){
        when(faizOran.aylikOran()).thenReturn(1.20);
        double hesaplananFaiz = krediHesaplama.faizHesapla(100.0);
        assertEquals(1.20, hesaplananFaiz, 0.001);
    }
}
```

Şekil 4.18. Mocklama yaparken When-Then Standardının kullanılması

Böylelikle, nesne mocklansa da istenilen durumları yaratıp testler yazılabilir. Bu yapı içerisinde *when* ifadesi kullanılmıştır. Her birim testin içeriği, belli adımlara göre ilerlemektedir. Bunun için Given-When-Then standartları takip edilmektedir. Given aşamasında elde olması istenen ve kullanılacak olan değişkenler oluşturulur. Test edilecek nesne yapılandırılmış olur. When aşamasında, Given aşamasında yapılandırılan nesne ile değerler ve değişkenler buluşturulup test edilecek kod işleme alınır. Then aşamasında ise, beklenen sonuç test koduna iletilir.

4.2. Otomatik Birim Test Oluşturma Yazılımının Tasarımı

Bu bölümün ilerleyen başlıklarında geliştirilen yazılımın önyüz ve arkayüzüyle ilgili çalışan algoritmanın akışından bahsedilmiştir.

4.2.1. Java-Class-ByteCode Dönüşümleri

Geliştirilen çerçevede öncelikle birim testleri oluşturulması için bir java sınıfı sisteme yüklenilmektedir. Bu java dosyası bir butona tıklanarak .class uzantılı dosyaya dönüştürülür ve yazılım hemen her yerinde .class dosyası üzerinden çalışma yapılır. Class dosyasını derleyebilmek için Şekil 4.19'da görüldüğü gibi *JavaCompiler* kütüphanesi kullanılmıştır.

```
public static void generateClass(String pathName, String fName) throws IOException {

    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    int result = compiler.run(null, null, null, pathName + fName + ".java");
    System.out.println("Compile result code = " + result);
    if (result == 1) {

        DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector();
        StandardJavaFileManager manager = compiler.getStandardFileManager(diagnostics, Locale.ENGLISH, Charset.forName("UTF-8"));
        File[] folder = new File(pathName).listFiles();
        for (File f : folder) {
            if (f.getName().endsWith(".class")) {
                f.delete();
            }
        }

        Iterable<? extends JavaFileObject> compilationUnits = manager.getJavaFileObjects(part1, part2);

        JavaCompiler.CompilationTask task = compiler.getTask(null, manager, diagnostics, null, null, compilationUnits);
        Boolean suc = task.call();

        manager.close();
        System.out.println("Compile result code : " + suc);
    }
}
```

Şekil 4.19. Class dosya dönüşümü için JavaCompiler kullanımı

Burada aynı zamanda birden fazla java sınıfına gerek duyan bir sınıf okutulduysa, belirtilen lokasyon içerisindeki diğer sınıflarında class dosyaları okunmaktadır. Geliştirilen sistem, okuduğu tüm dosyaları *Iterable* koleksiyon listesi içerisinde saklamaktadır ve sonrasında bu liste dönüşüm işlemleri için kullanmaktadır.

Java dosyasından Class dosyasına dönüşüm gerçekleştikten sonra bayt kod dönüşüm işlemleri gerçekleştirilmektedir. Bu işlem için javassist kütüphanesi içerisindeki *ClassPool*, *CtClass*, *CtMethod* ve *InstructionPrinter* alt sınıflar kullanılır. Şekil 4.20'de görüldüğü gibi, öncelikle class dosyası okutulmaktadır.

```
public static void Main(String FileName) throws Exception {
    ByteCodeEditor _byteCodeEditor = new ByteCodeEditor();
    CtClass _ctClass = _classPool.makeClass(new FileInputStream(filePath));

    _byteCodeEditor.printMethodCode(_ctClass);
}

public static void printMethodCode(CtClass _ctClass) throws Exception {
    InstructionPrinter instructionPrinter = new InstructionPrinter(ps);
    for (CtMethod method : _ctClass.getDeclaredMethods()) {
        System.out.println("MethodName : " + method.getName());
        instructionPrinter.print(method);
}
```

Şekil 4.20. Javassist kütüphanesinin ByteCode dönüşümü için kullanımı

Okutulan dosyanın içerisindeki sınıf ve metot bilgileri classPool isimli sınıf havuzu içerisinde tutulmaktadır. Sonrasında, InstructionPrinter ile sınıf havuzundan bilgiler çekilir ve sonuçlar bir çıktı halinde görüntülenir.

Şekil 4.21'de Javassist kütüphanesinin üretmiş olduğu bayt kodlara bir örnek verilmiştir.

```
MethodName : faizHesaplama
0: dload_0
1: ldc2_w #2 = int 100.0
4: ddiv
5: dload_2
6: dmul
7: dreturn
MethodName : faizHesaplamaBirAlti
0: dload_2
1: dconst_1
2: dcmpg
3: ifge 8
6: dconst_1
7: dstore_2
8: dload_0
9: ldc2_w #2 = int 100.0
12: ddiv
13: dload_2
14: dmul
15: dreturn
```

Şekil 4.21. ByteCode örneği

Bu örnekte, sınıf içerisinde bulunan her bir metot *MethodName* satırları arasında ayrıştırılmıştır. Her metot 0. satır numarası ile başlamaktadır. Bu metot dreturn anahtar kelimesi ile bir değer döndürdüğü ve aynı zamanda lcd2_w anahtar kelimesi ile parametreler aldığını göstermektedir.

Bu bayt kodlar Javassist kütüphanesi tarafından okunur ve ilgili sınıfın dosya ismi ve sahip olduğu yolu *ClassPool*, *CtClass* ve *CtMethod* parametrelerinde tutar. Bayt koda dönüştürülmüş tüm kod satırları Netbeans arayüzüne ait konsol ekranında görülebildiği gibi metin belgesi halinde de çıktı alınmaktadır. Yazılım için geliştirilen bir başka sınıf, *FindObjectsInClasses* sınıfıdır. Bu sınıf yardımıyla Byte code dosyası ile ilgili üretilen çıktıdan her bir bayt kod satır satır okutulur ve okunan bu kodlar MongoDB veritabanında NoSQL formatında saklanı. Bunun için çerçeve içerisinde *ConnectionDB* isimli bir sınıf oluşturulmuştur. Bu sınıf içerisinde *com.mongodb* kütüphanesine ait metotlar kullanılarak koleksiyon oluşturma ve koleksiyon içindeki verilere ulaşma gibi özellikler ortak bir sınıfta toplanmıştır. Bu sınıfta ayrıca, tüm JSon formatındaki verilerin bir metin belgesinde arşivlemesi sağlanmıştır.

Buradaki önemli nokta, bilgilerin çekilip veri tabanına kaydedebilmesi için java string fonksiyonlarından yardım alınmıştır. Bunu çalışma için geliştirilen opcode ayrıştırma yöntemi yapmaktadır. Şekil 4.22'de görüldüğü gibi, sınıf içerisinde kullanılan değişkenler, metot isimleri, varsa almış oldukları parametreler ve göndermiş oldukları sonuçlar, oluşturulmuş tüm nesneler, kullanılmış koşul ve döngü blokları hakkında bilgiler sırasıyla string fonksiyonları yardımıyla alınır ve MongoDB'ye kaydedilir.

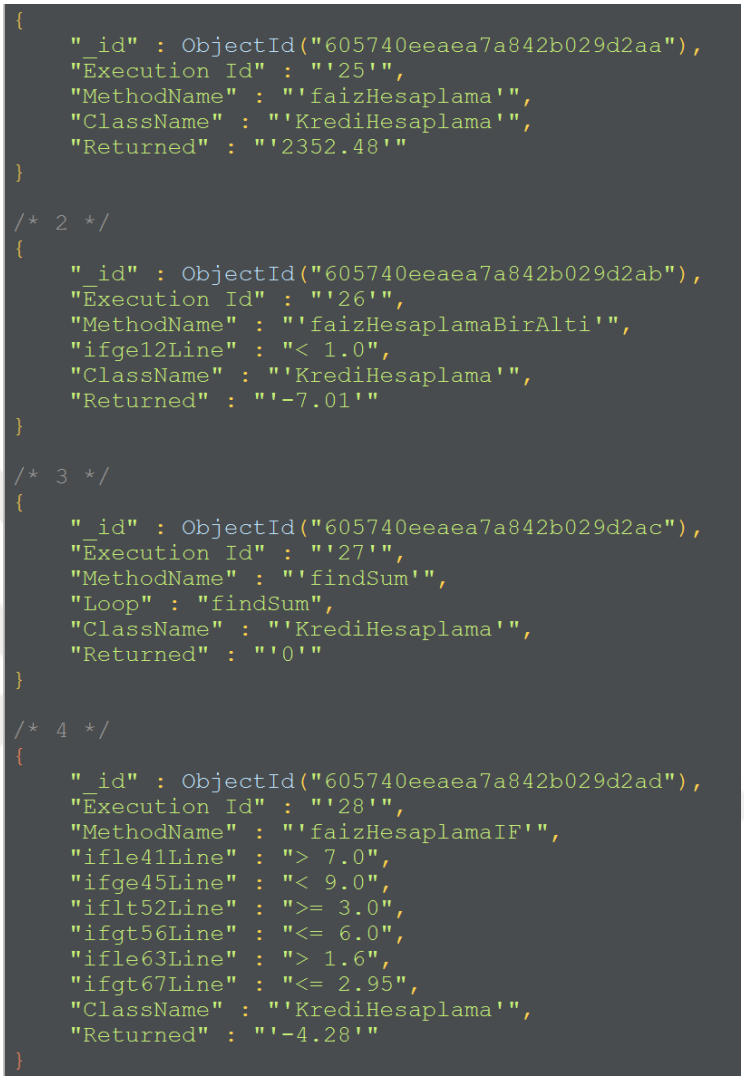
```
if (line_.contains(": invokevirtual #")) {
    objectName1 = line_.substring(line_.indexOf("Method") + 7, line_.indexOf("("));
    document.append("$set", new BasicDBObject().append("ObjectClassName_Method", convertToByteCodeMethod(objectName1)));
    searchQuery = new BasicDBObject().append("Execution Id", "'" + executionId + "'");
    table.update(searchQuery, document, true, false);
    document.append("$set", new BasicDBObject().append("Mocking", convertToByteCodeMocking(objectName1)));
    searchQuery = new BasicDBObject().append("Execution Id", "'" + executionId + "'");
    table.update(searchQuery, document, true, false);
}

if (line_.contains(": ifge")) {   //value>=0
    ifMap.add(new bcList("ifge", Integer.parseInt(line_.substring(line_.indexOf("ifge") + 5, line_.length()))));
    ifSetVariable = ifGetVariable("<", linenumber);
    writeToDBtoIf("ifge" + linenumber, ifSetVariable);
}

if (line_.contains(": iinc")) {   //increment
    writeToDBtoLoop(MethodName);
    int getIINCLineNumber = 0, firstLoopNumbet = 0, secondLoopNumber = 0;
    briinc = new BufferedReader(new FileReader(fpath));
    for (int i = 1; i <= countLine; i++) {
        line_iinc = briinc.readLine();
        if (line_iinc.contains("iinc")) {
            getIINCLineNumber = i;
        }
        if (line_iinc.contains("if_icmp")) {
            firstLoopNumbet = i;
        }
        if (line_iinc.contains("goto")) {
            secondLoopNumber = i;
        }
    }
    if ((getIINCLineNumber > firstLoopNumbet) && (getIINCLineNumber < secondLoopNumber)) {
        setingMethodName = MethodName;
    }
    if (getIINCLineNumber < firstLoopNumbet) {
        setingMethodName = MethodName;
    }
}
```

Şekil 4.22. ByteCode satırlarının String fonksiyonlarıyla ayrıştırılması ve verilerin MongoDB veritabanına kaydedilmesi

Ek-A'da uygulama içerisinde kullanılan *invokevirtual, getfield, invokestatic, getstatic, ifge, ifle, iflt, ifgt, ifeq, ifne, iinc, if_icmp* ve *goto* gibi opcode'lar hakkında bilgi verilmiştir. Aynı zamanda, tüm bu bilgiler işlemin gerçekleştiği gün ve saate göre metin belgesi (.txt) formatında sistem dosyaları altında arşivlenmektedir.

Şekil 4.23'de MongoDB'de bulunan byteCoding koleksiyonu görülmektedir. NoSQL yapısında veriler anahtar-değer (key-value) yapısı altında listelenmiştir. Key alanlarının isimlendirilmesi, kimisinde yapılacak işlemlerle ilgili olurken kimisinde de bayt kod terimlerini anımsatmaktadır. Örneğin, MethodReturnType anahtar ismi o

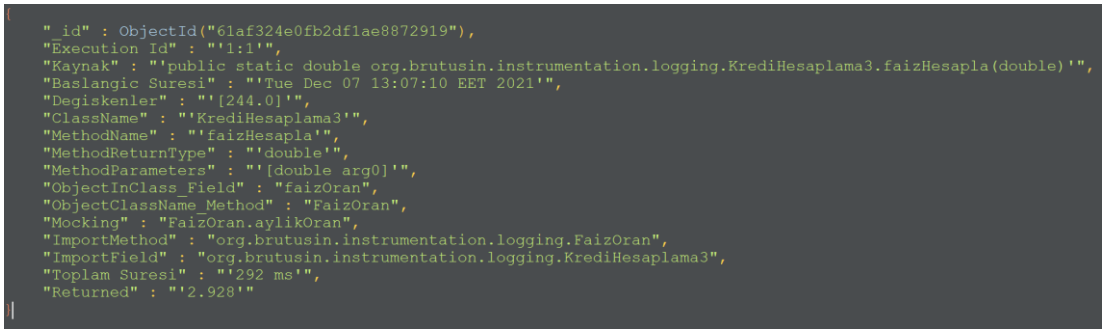
yöntemin sonuç olarak gönderdiği değeri saklarken, ifge45Line anahtar ismi ise if bloğunun karşılık geldiği işleme ait değeri saklamaktadır.



```
{
    "_id" : ObjectId("605740eeaea7a842b029d2aa"),
    "Execution Id" : "'25'",
    "MethodName" : "'faizHesaplama'",
    "ClassName" : "'KrediHesaplama'",
    "Returned" : "'2352.48'"
}

/* 2 */
{
    "_id" : ObjectId("605740eeaea7a842b029d2ab"),
    "Execution Id" : "'26'",
    "MethodName" : "'faizHesaplamaBirAlti'",
    "ifge12Line" : "< 1.0",
    "ClassName" : "'KrediHesaplama'",
    "Returned" : "'-7.01'"
}

/* 3 */
{
    "_id" : ObjectId("605740eeaea7a842b029d2ac"),
    "Execution Id" : "'27'",
    "MethodName" : "'findSum'",
    "Loop" : "findSum",
    "ClassName" : "'KrediHesaplama'",
    "Returned" : "'0'"
}

/* 4 */
{
    "_id" : ObjectId("605740eeaea7a842b029d2ad"),
    "Execution Id" : "'28'",
    "MethodName" : "'faizHesaplamaIF'",
    "ifle41Line" : "> 7.0",
    "ifge45Line" : "< 9.0",
    "iflt52Line" : ">= 3.0",
    "ifgt56Line" : "<= 6.0",
    "ifle63Line" : "> 1.6",
    "ifgt67Line" : "<= 2.95",
    "ClassName" : "'KrediHesaplama'",
    "Returned" : "'-4.28'"
}
```


Şekil 4.23. MongoDB'de bulunan byteCoding isimli koleksiyonunun yapısı

Şekil 4.24'de MongoDB'de bulunan *kayitlar* koleksiyonu görülmektedir. Burada sınıf ve metotlarla ilgili tüm bilgiler kaydedilmiştir. Öyle ki, mocklama yapılacak nesnelerin sınıf ve nesne isimleri bile bu listede mocking anahtar kelimesi altında yer almaktadır.

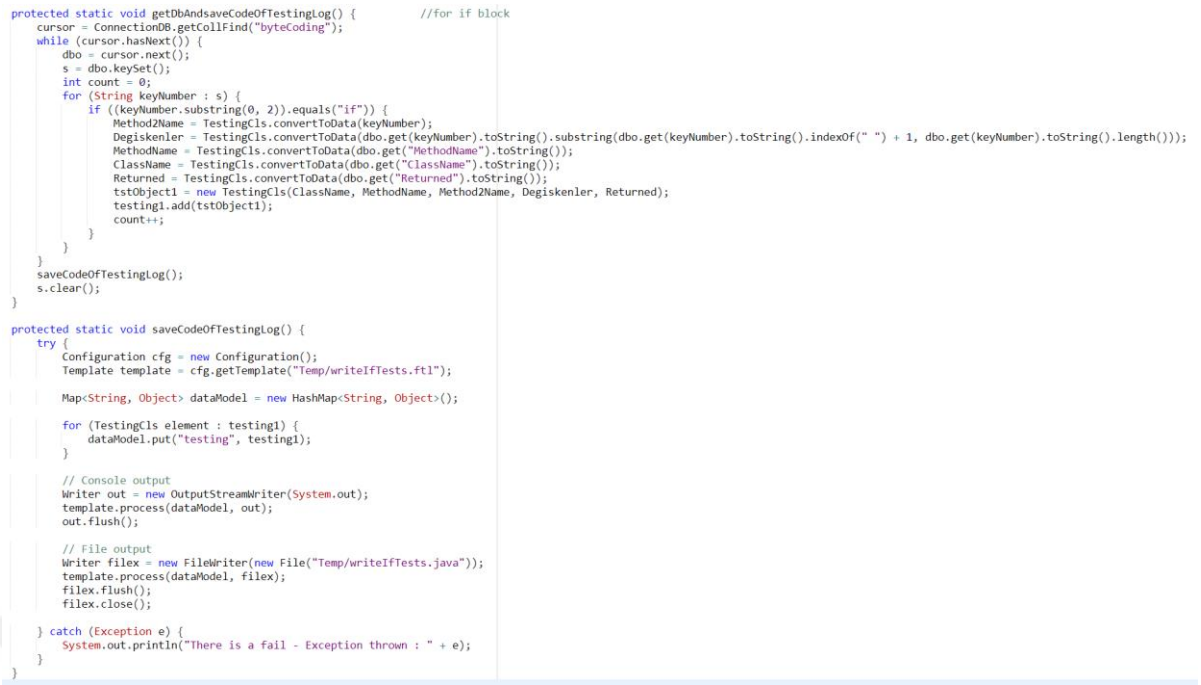


Şekil 4.24. MongoDB'de bulunan kayitlar isimli koleksiyonunun yapısı

4.2.2. Veri Taşıma – Şablon Çıkarımı

MongoDB üzerinde arşivlenen tüm verilerin veri okuma veya yazma gibi işlemler için otomatik birim test yazılımı ile iletişimde kalabilmesi gerekmektedir. Bu sebeple, veritabanının yazılım ile arasındaki iletişimi sağlayabilmesi için bir *POJO* sınıfı yazılmış ve senaryolara uygun olabilecek tüm kurucu (constructora) ve getter-setter metotlar, değişkenleri ile birlikte tanımlanmıştır. POJO sınıfı, çerçeve kapsamında bir bakıma veri taşıyıcısı rolü üstlenmektedir. Veriler veri tabanında iki tabloda saklanmaktadır. Bu tablolar join mantığıyla id'ler ile birleştirilmiş ve birbirleriyle iletişimi sağlanmıştır. Bu join işlemleriyle veritabanından okunacak veriler için iki farkı sınıf yazılmıştır.

Bunlardan ilki Şekil 4.25'de verilen örnek kod ile yazılım için geliştirilmiş olan *GetItFromMongoDB* sınıfıdır. Gerçek zamanda okutulan kodlar içerisindeki koşul ve döngüler için veri tabanında *byteCoding* koleksiyonu oluşturulmuştur. Bu koleksiyonda okunan veriler pojo sınıfı yardımıyla birim test koduna dönüştürmek üzere FTL formatına yönlendirilir.



```
protected static void getDbAndsaveCodeOfTestingLog() {          //for if block
    cursor = ConnectionDB.getCollFind("byteCoding");
    while (cursor.hasNext()) {
        dbo = cursor.next();
        s = dbo.keySet();
        int count = 0;
        for (String keyNumber : s) {
            if ((keyNumber.substring(0, 2)).equals("if")) {
                Method2Name = TestingCls.convertToData(keyNumber);
                Degiskenler = TestingCls.convertToData(dbo.get(keyNumber).toString().substring(dbo.get(keyNumber).toString().indexOf(" ") + 1, dbo.get(keyNumber).toString().length()));
                MethodName = TestingCls.convertToData(dbo.get("MethodName").toString());
                ClassName = TestingCls.convertToData(dbo.get("ClassName").toString());
                Returned = TestingCls.convertToData(dbo.get("Returned").toString());
                tstObject1 = new TestingCls(ClassName, MethodName, Method2Name, Degiskenler, Returned);
                testing1.add(tstObject1);
                count++;
            }
        }
    }
    saveCodeOfTestingLog();
    s.clear();
}

protected static void saveCodeOfTestingLog() {
    try {
        Configuration cfg = new Configuration();
        Template template = cfg.getTemplate("Temp/writeIfTests.ftl");

        Map<String, Object> dataModel = new HashMap<String, Object>();

        for (TestingCls element : testing1) {
            dataModel.put("testing", testing1);
        }

        // Console output
        Writer out = new OutputStreamWriter(System.out);
        template.process(dataModel, out);
        out.flush();

        // File output
        Writer filex = new FileWriter(new File("Temp/writeIfTests.java"));
        template.process(dataModel, filex);
        filex.flush();
        filex.close();

    } catch (Exception e) {
        System.out.println("There is a fail - Exception thrown : " + e);
    }
}
```

Şekil 4.25. GetItFromMongoDB sınıfında bulunan koşul yapıları için oluşturulmuş olan verilerin okunması ve FTL dosyasına yazılması

Şekilde bulunan örnek kodda, *byteCoding* koleksiyonu içerisinde if ile başlayan anahtar kelimelerin karşılığında bulunan değişkenler sınıf ismi, metot ismi ve geri dönüş değerleri ile birlikte pojo sınıfına gönderilmiştir. Ardından *writeIfTests.ftl* isimli şablon dosyasına bu değerler birim test metodu olacak şekilde yazdırılmıştır.

Şekil 4.26'da MongoDB veritabanından veri çeken *ReadDataFromDB* sınıfından bir bölüm görülmektedir. Bu sınıfta yine join mantığıyla iki koleksiyonun bağlantısı sağlanmıştır. İlgili tüm değişkenler hem veri tabanından okunmakta hem de join işlemleri için güncellenmesi gereken alanlar varsa onlar kontrol edilmektedir. Tüm değişkenler çerçevenin ana sınıfına yönlendirilmektedir. Aynı zamanda, FTL dosyası için seçilmiş olan şablon değişkenleri kopyalanarak birim test metotları oluşturulmaktadır.

```
while (cursor1.hasNext()) {
    dbObject = cursor1.next();
    if (dbObject.get("MethodName").equals(data_)) {
        objectClassName = dbObject.get("ObjectClassName_Method");
        objectName = dbObject.get("ObjectInClass_Field");
        Mocking = dbObject.get("Mocking");
        ImportMethod = dbObject.get("ImportMethod");
        ImportField = dbObject.get("ImportField");
    }
}
while (cursor2.hasNext()) {
    dbObject = cursor2.next();
    if (dbObject.get("MethodName").equals(data_)) {
        //Burada kayitlar tablosunu guncelleyecegiz.
        document = new BasicDBObject();

        document.append("$set", new BasicDBObject().append("ObjectClassName_Method", objectClassName));
        searchQuery = new BasicDBObject().append("MethodName", data_);
        collection2.update(searchQuery, document, true, false);

        document.append("$set", new BasicDBObject().append("ObjectInClass_Field", objectName));
        searchQuery = new BasicDBObject().append("MethodName", data_);
        collection2.update(searchQuery, document, true, false);

        document.append("$set", new BasicDBObject().append("Mocking", Mocking));
        searchQuery = new BasicDBObject().append("MethodName", data_);
        collection2.update(searchQuery, document, true, false);

        document.append("$set", new BasicDBObject().append("ImportMethod", ImportMethod));
        searchQuery = new BasicDBObject().append("MethodName", data_);
        collection2.update(searchQuery, document, true, false);

        document.append("$set", new BasicDBObject().append("ImportField", ImportField));
        searchQuery = new BasicDBObject().append("MethodName", data_);

        LoggingInterceptor.ObjectInClass_Field = objectClassName;
        LoggingInterceptor.ObjectClassName_Method = objectName;
        LoggingInterceptor.Mocking = Mocking;
        LoggingInterceptor.ImportField = ImportField;
        LoggingInterceptor.ImportMethod = ImportMethod;

        collection2.update(searchQuery, document, true, false);
    }
}
```

Şekil 4.26. ReadDataFromDB sınıfında bulunan koleksiyon joinleri sonucunda framework'e yönlendirilen tüm verilerin okunması ve FTL dosyasına yazılması

Şekil 4.27'de bir FTL şablonuna ait örnek verilmiştir. Veri tabanından çekilen tüm veriler, taşıyıcı sınıf olan POJO ile getirilmektedir. POJO sınıfındaki kurucu metotların yönlendirmiş olduğu parametreler şekilde olduğu gibi bir FTL şablonunda yerlerini almışlardır.

FTL şablonları html kodlarına benzer bir mantık sergilemektedir. Parantezler arasında verilmiş olan değişken isimleri gönderilen veriler ile bir nevi yeni bir yapı oluşturmaktadır.

```
import org.junit.*;
import static org.junit.Assert.*;

<#list testing as tst>
import ${tst.importMethod};
import ${tst.importField};
</#list>

public class ${className}Test{

    ${className} _${className} = new ${className}();

    <#list testing as tst>
    @Test(groups="${groupName}")
    public void ${tst.methodName}Test() {
        _${className}.${tst.objectClassName} = new ${tst.objectName}();
        ${tst.methodReturnType} hesaplananFaiz = _${className}.${tst.methodName}(${tst.degiskenler});
        assertEquals(${tst.returned}, hesaplananFaiz, ${tst.degiskenler});
    }
    </#list>
}
```

Şekil 4.27. Bir test sınıfı için hazırlanmış FTL Java Şablon Motorunun örneği

.ftl uzantılı taslak şablonu içerisinde kullanılan bazı ifadeler şöyledir;

- $ etiketi ile gösterilen tanımlamalarda, { } parantezler içerisinde kullanılan değerler tek seferlik kullanım içindir.
- <#List > etileti ile gösterilen tanımlamalar ise, liste halindeki verileri yazdırabilmek için kullanılır. List parametresinden sonra kullanılan *testing* değişken ismi, POJO sınıfından for döngüsüyle çekilen verilerin içeriğini tutmaktadır. List etiketi bir bakıma ftl şablonlarında for döngüsü olarak kullanılmaktadır. *as* parametresinden sonra kullanılan *tst* değişkeni *testing* listesindeki tüm verileri dolaşmaktadır. *tst*'den sonra nokta işareti ile listede istenilen parametre yazdırılır.

Veri tabanında depolanan bu verilerle birlikte FTL kullanılarak birim testleri otomatik bir şekilde istenen dosya formatında hazırlanarak bir çıktı haline getirilmektedir. Bu çıktıların içeriği gerçek zamanlı toplanan verilerden yola çıkarak oluşturulan yeni bir sınıfa ait kodlardır. Şekil 4.27'de görülen FTL şablonları birim test kodları baz alarak hazırlanmıştır.

Çerçeveye okutulan java sınıfında kullanılan paremetrelerden üretilen veriler mongodb'ye kaydedilir ve veri tabanındaki veriler ftl şablonlarıyla çekilir. Çerçeve içerisinde *GenerateRandomJavaDataType* isimli rastgele veri üreten bir sınıf yazılmıştır ve uygulamanın ihtiyaç duyulan yerlerinde bu rastgele veriler kullanılmaktadır. Bu sınıf içerisinde yine java string fonksiyonları kullanılmıştır ve bu fonksiyonlar yardımıyla ilgili test sınıfına hangi veri türünde rastgele veri

üretileceği belirlenmektedir. Üretilen tüm veriler java koleksiyon listelerinde tutulmaktadır ve ilgili test sınıfına gönderim için yönlendirilmektedir.

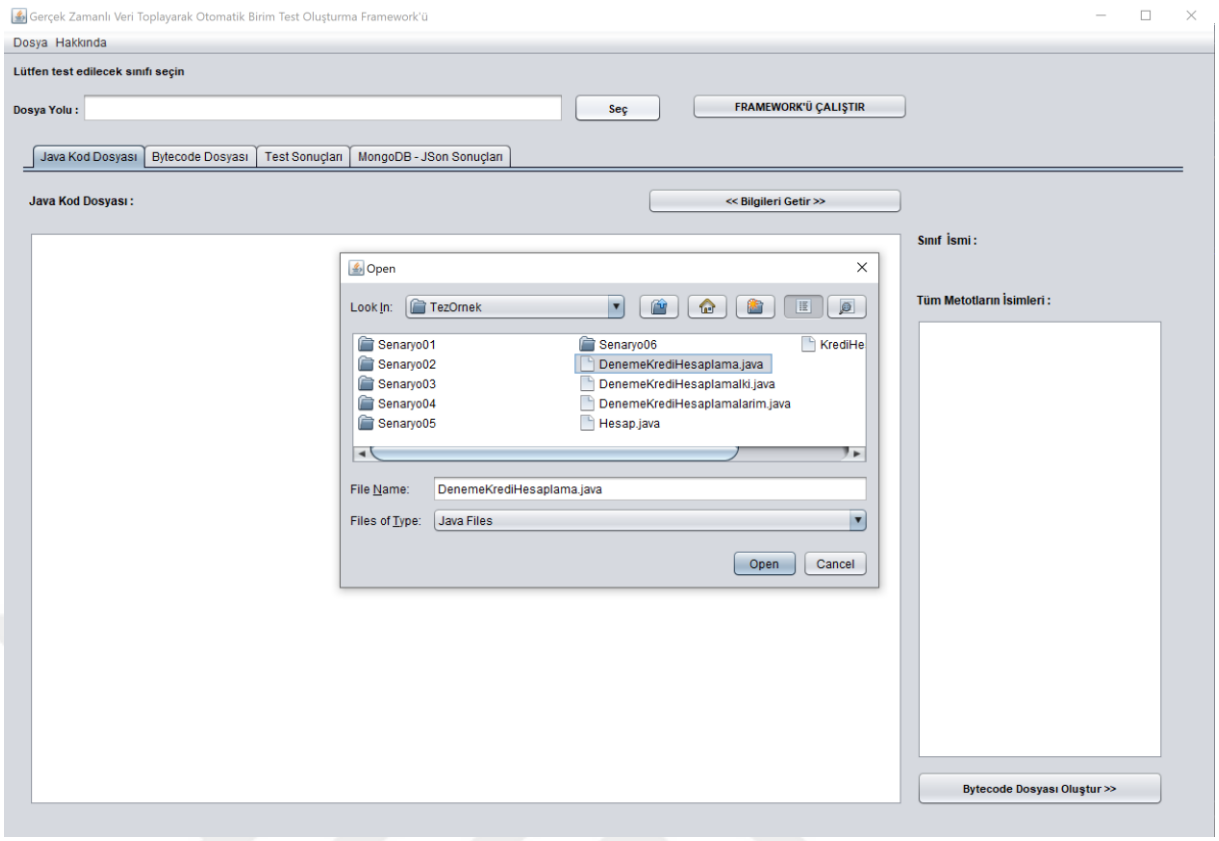
4.3. Otomatik Birim Test Oluşturma Yazılımının Gerçekleştirilmesi

Uygulamanın arkayüz kodları tamamlandıktan sonra önyüz kodları hazırlanmış ve gereksinimler bölümünde belirtilen senaryolar üzerinde gerçek zamanlı olarak çalışacak şekilde birim test kodu üretimi gerçekleştirilmiştir.

Uygulama sadece konsol ortamı olarak değil bir masaüstü platform olarak da geliştirilmiştir. Bunun için *UnitTestGeneratorGUI* isimli sınıf tasarlanmıştır. Şekil 4.28'de verilen ekran görüntüsü bu sınıf ile tasarlanmıştır. Sınıf, çerçevede ilgili yerlerin çalışabilmesi için gerekli hiyerarşiyi kullanarak tüm komut düğmelerinin, liste ve metin kutularının görevlerini içermektedir.

Söz konusu çerçeve üzerinde birçok metot ve sınıf türleri üzerinde çalışma yapılmış, her bir alternatif durum için oluşturulan senaryolar denenmiş ve hepsinden planladığı üzere uygun olabilecek otomatik birim test kodları oluşturulmuştur.

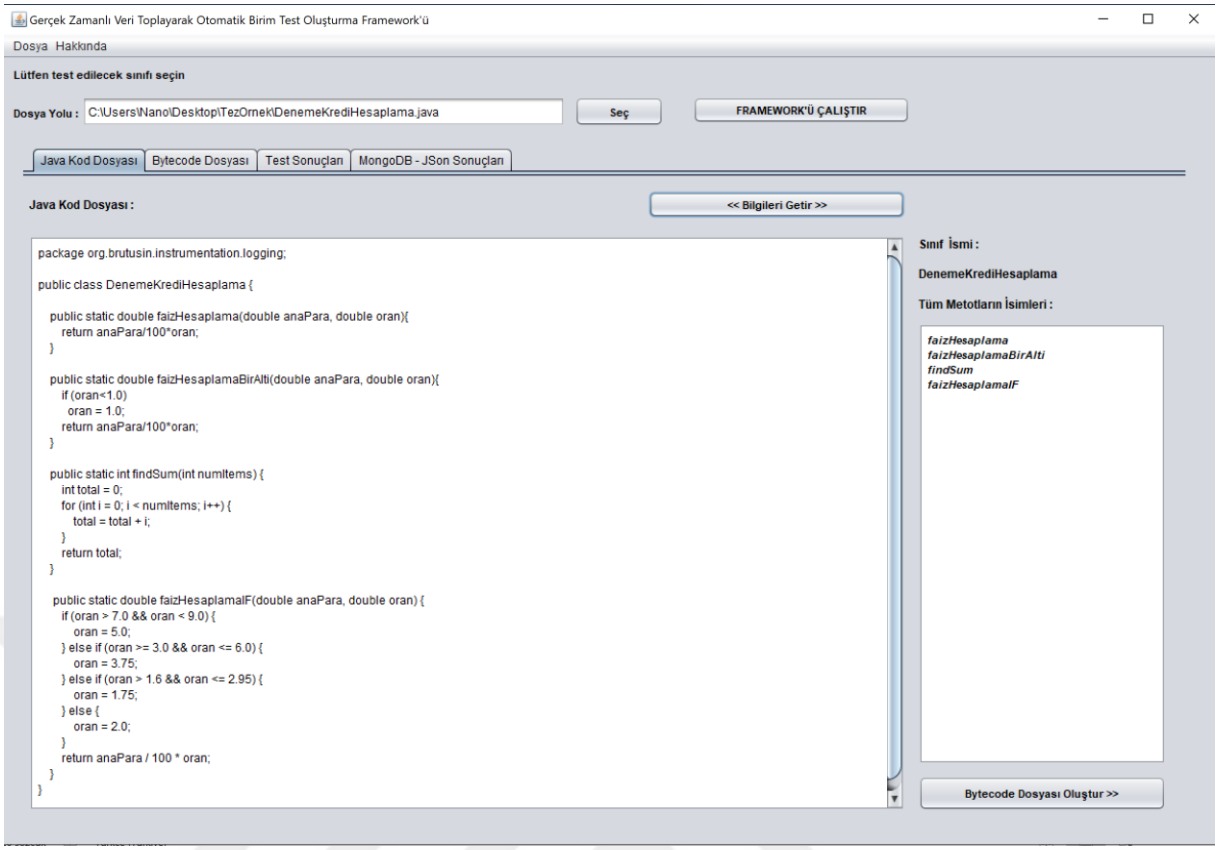
Çalışma kapsamında çeşitli deneme testleri ve performans analizleri gerçekleştirilmiştir. Çerçeve sadece değişkenler, nesneler ve metotlar üzerinde çalışmamaktadır. Buna ek olarak, koşul yapıları ve döngüler gibi ek özellikler de dahil edilmiştir. Söz konusu bu ek özelliklerle birlikte çerçeve üzerinde hazırlanan senaryolarla çalışmalar yapılmış ve hepsinden planlandığı gibi uygun olabilecek otomatik birim test kodları ve bunları içeren bir test sınıfı dosyası elde edilmiştir.



Şekil 4.28. Otomatik birim test oluşturma Framework'ün dosya seçme ekranı

Şekil 4.28'deki ekran görüntüsünde görüldüğü üzere, arayüz iki ana kısımdan oluşmaktadır. İlk olarak, seç butonuyla test dosyası hazırlanmak istenen java kodu seçilmektedir. Ardından ikinci kısımda ise, ekranda bulunan tab nesnesi kullanılarak çerçeveye gönderilecek verilerin hazırlığı yapılmaktadır.

Şekil 4.29'da görüldüğü üzere *java kod dosyası* penceresine seçilen java sınıfının içeriği döküm halinde gelmektedir.

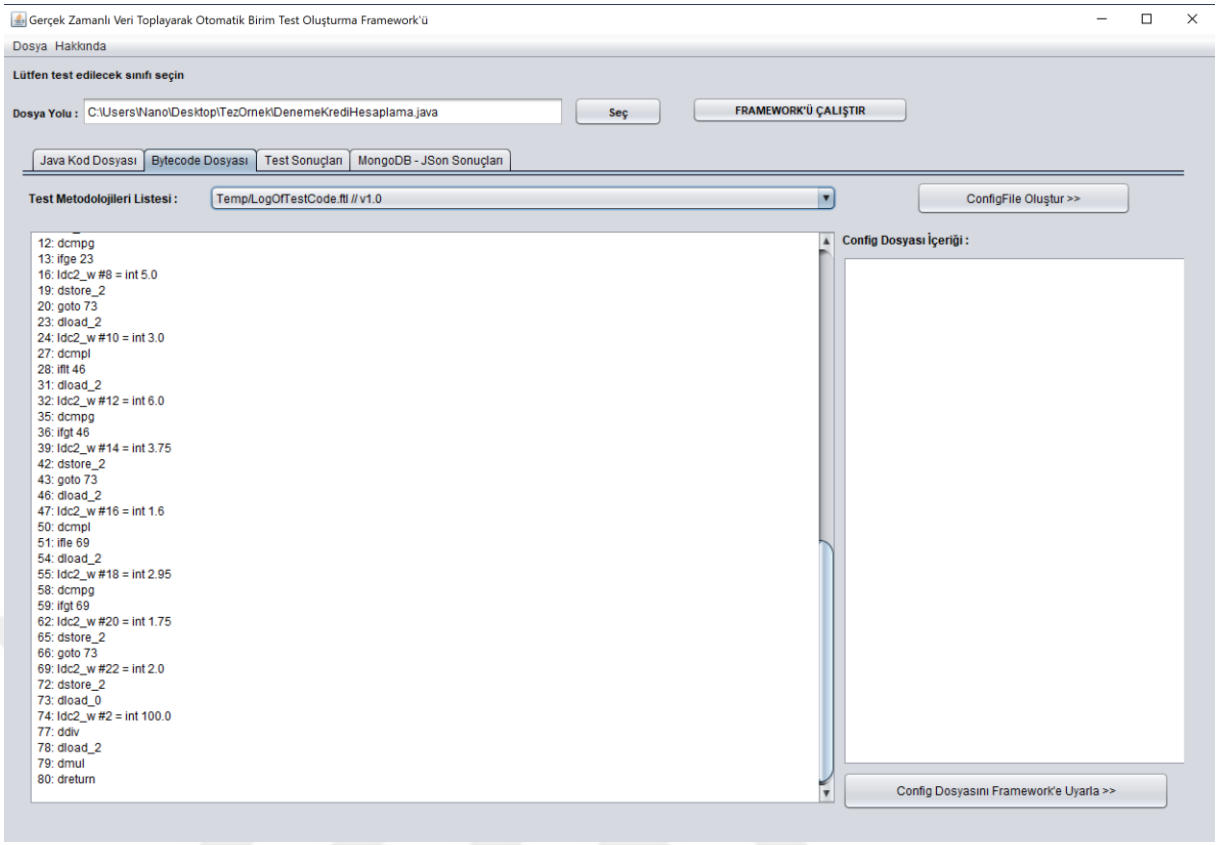


Şekil 4.29. Framework'e okutulacak olan java sınıf dosyasının yüklenmesi

Sınıf kodları içerisindeki her bir kod bayt koda çevirilecektir ancak, yorum satırlarındaki herhangi bir kod yapısı veya yorum satırları bayt kod tarafından dönüştürülmemektedir. Bu özellik tüm derleyicilerde bulunmaktadır. Şekilde verilen örnek üzerinde, bu sınıfta hem koşul hem de döngü yapıları bulunmaktadır.

Bilgileri Getir butonuna tıklanıldığında, sınıfa ait isim ve ait olduğu metodların isimleri ekranın sağ tarafında listelenecektir. Bunun için *GetInfoAboutJavaAndByteCodeFile* isimli bir sınıf tasarlanmıştır ve arka planda çalıştırılmıştır. Burada öncelikle dosya yolu ile birlikte okutulan java dosyasının *JavaCompiler* kütüphanesi kullanılarak *class* dosyasına dönüşümü sağlanmaktadır. Bu dönüşümle birlikte, bayt kod dönüşümü de yapılmaktadır. Bu dönüşümler tamamlandıktan sonra, ilgili sınıfa ait sınıf ismi ve ona ait metodların isimleri bir dizi değişkeni ile toplanır ve ekranın sağ tarafındaki listeye bu bilgiler aktarılır.

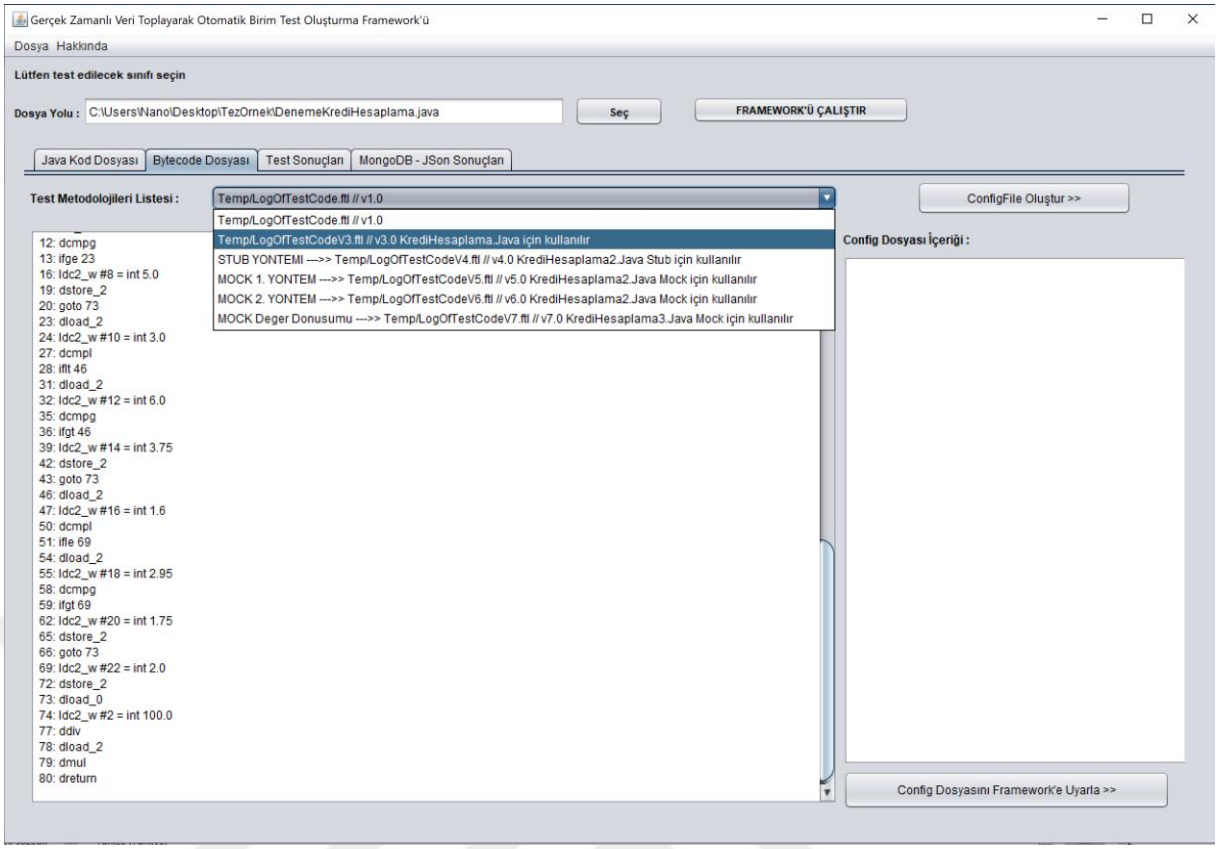
Bytecode Dosyası oluştur butonuna tıklanıldığında Şekil 4.30'da görülen *Bytecode Dosyası* sekmesi açılacaktır.



Şekil 4.30. Okutulan Java sınıfına ait bayt kod dönüşümü

Javassist kütüphanesi yardımıyla ilgili örnek sınıfının class dosyası oluşturulmuş ve bayt koda çevrilmiştir. Bayt kodlar javassist kütüphanesinin belirlemiş olduğu standartta olacak şekilde metot isimleri ile ayrılmış şekilde oluşturulmaktadır. Bu bayt kodundan gerekli değerler şablon halinde oluşturulmuş birim test sınıf ve metotlarının içerisine yönlendirilerek çalışma anında otomatik bir şekilde birim testler oluşturulacaktır. Bunun için hangi test senaryosu kullanılacaksa ona ait test metedolojisi ekranın yukarısında bulunan açılır listeden seçilmelidir.

Şekil 4.31'de senaryolarının bulunduğu açılır liste görülmektedir. Normal test metedolojilerinin yanısıra mock ve stub yöntemleriyle de geliştirilmiş test senaryolarının isimleri bulunmaktadır. Çerçeve ile açılan sınıfın yapısına göre bu test senaryoları seçilebilmekte ve sonuçları alınabilmektedir.



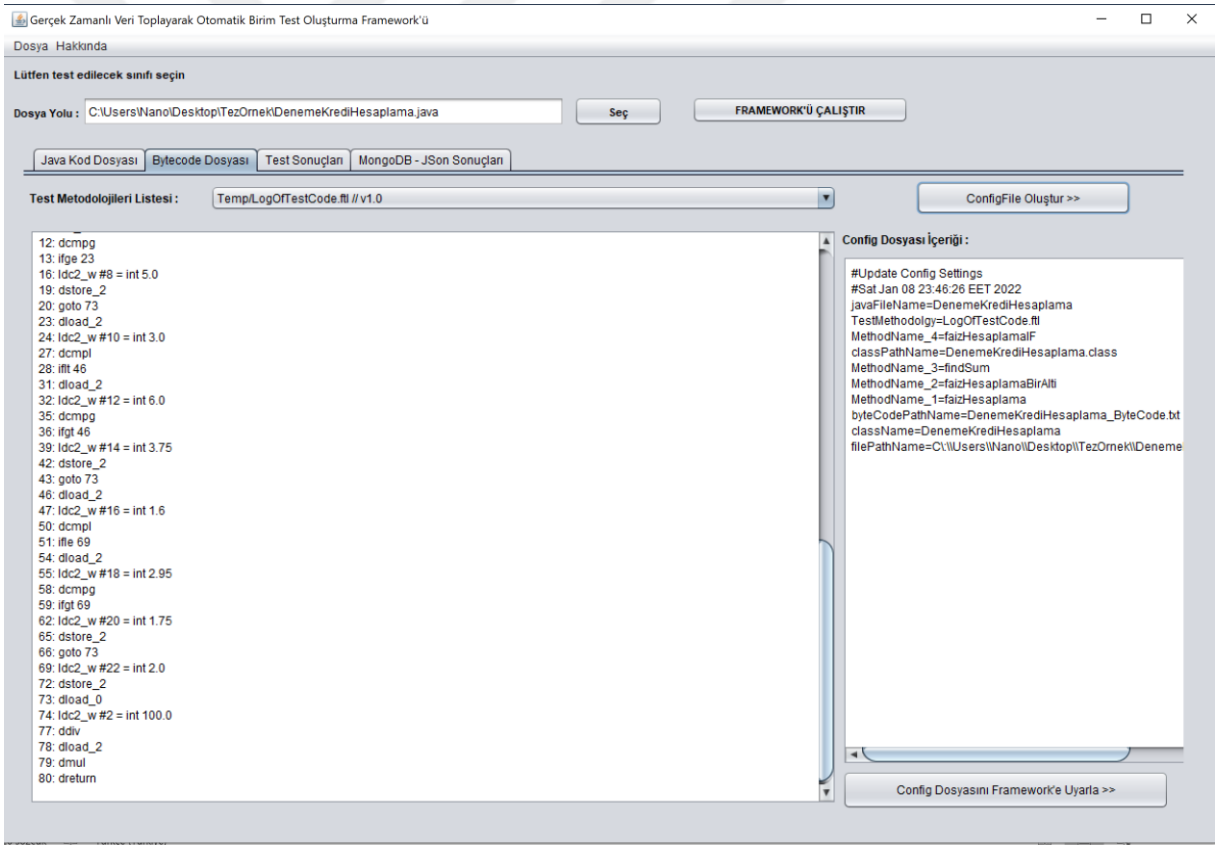
Şekil 4.31. Test senaryosunda kullanılacak olan test metodolojilerinin listesi

Test metodoloji seçimiyle, hangi test senaryosuyla otomatik birim test üretileceği belirlenmektedir. Açılır listede bu senaryoların tanımları ve örnek dosya isimleri açıklamalarıyla verilmiştir. Bu seçim yapıldıktan sonra *ConfigFile Oluştur* butonuna tıklanılarak Şekil 4.33'deki gibi bir yapılandırma dosyası oluşturulmaktadır. Oluşturulan yapılandırma dosyasında; java ve class dosyalarının isimleri ve bulundukları dosya yolu, seçilen test yöntemi, java sınıfının ismi ve ait olduğu method isimleri, bayt koda ait dosya ismi ile birlikte bu yapılandırma dosyasının oluşturulma tarih ve saati gibi bilgiler yer almaktadır. Şekil 4.32'de görülen *config.properties* dosyasının oluşturulmasını sağlayan kodlar görülmektedir.

Framework'ü Çalıştır butonuna tıklanıldığında, gerekli tüm bilgiler bu yapılandırma dosyasından sırayla çekilecektir.

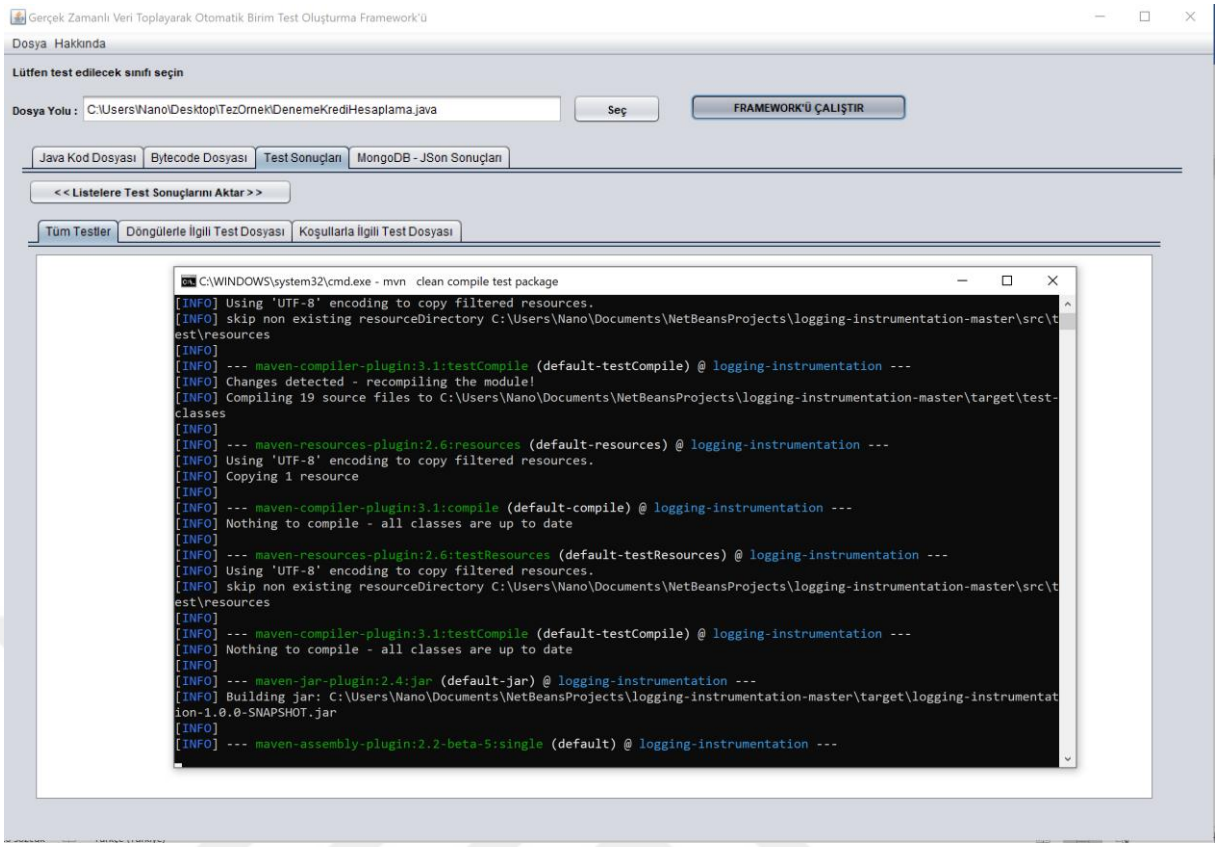
```
props = new Properties();
props.setProperty("TestMethodolgy", TestMethodolgy);
props.setProperty("javaFileName", javaFileName);
props.setProperty("filePathName", filePathName.toString());
props.setProperty("classPathName", javaFileName + ".class");
props.setProperty("byteCodePathName", javaFileName + "_ByteCode.txt");

props.setProperty("className", javaFileName);

String txt = jTextArea1.getText();
countMethodName = 0;
String[] arrayOfLines = txt.split("\n");
ArrayList<String> linesAsAL = new ArrayList<String>();
for (String line : arrayOfLines) {
    //linesAsAL.add(i + " . Method Name : " + line);
    linesAsAL.add(line);
}
for (String line : arrayOfLines) {
    System.out.println(linesAsAL.get(countMethodName));

    props.setProperty("MethodName_" + (countMethodName + 1), linesAsAL.get(countMethodName));
    countMethodName++;
}
```

Şekil 4.32. Config.Properties dosyasının oluşumunda kullanılan örnek kodlar



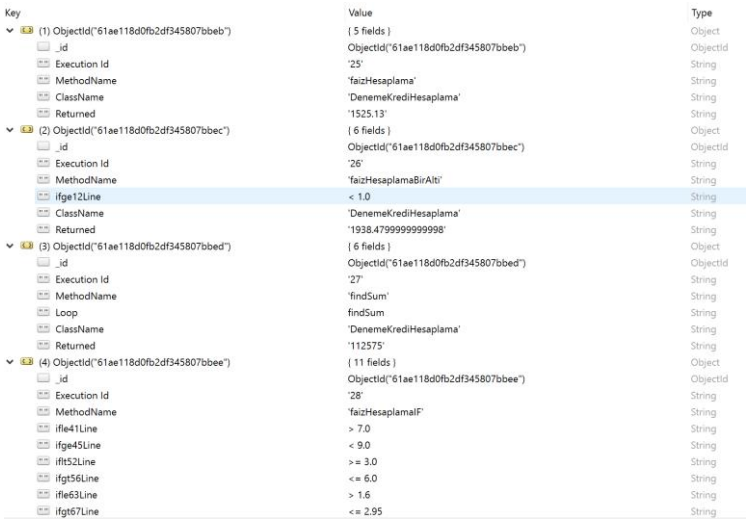
Şekil 4.33. Tercih edilen tüm işlemlerin yapılandırma dosyasına dönüşümü

Şekil 4.34'de görüldüğü üzere, komut satırı ekranı açılmakta ve çerçeve config.properties yapılandırma dosyasından okuduğu bilgiler ile çalışmaya başlamaktadır.



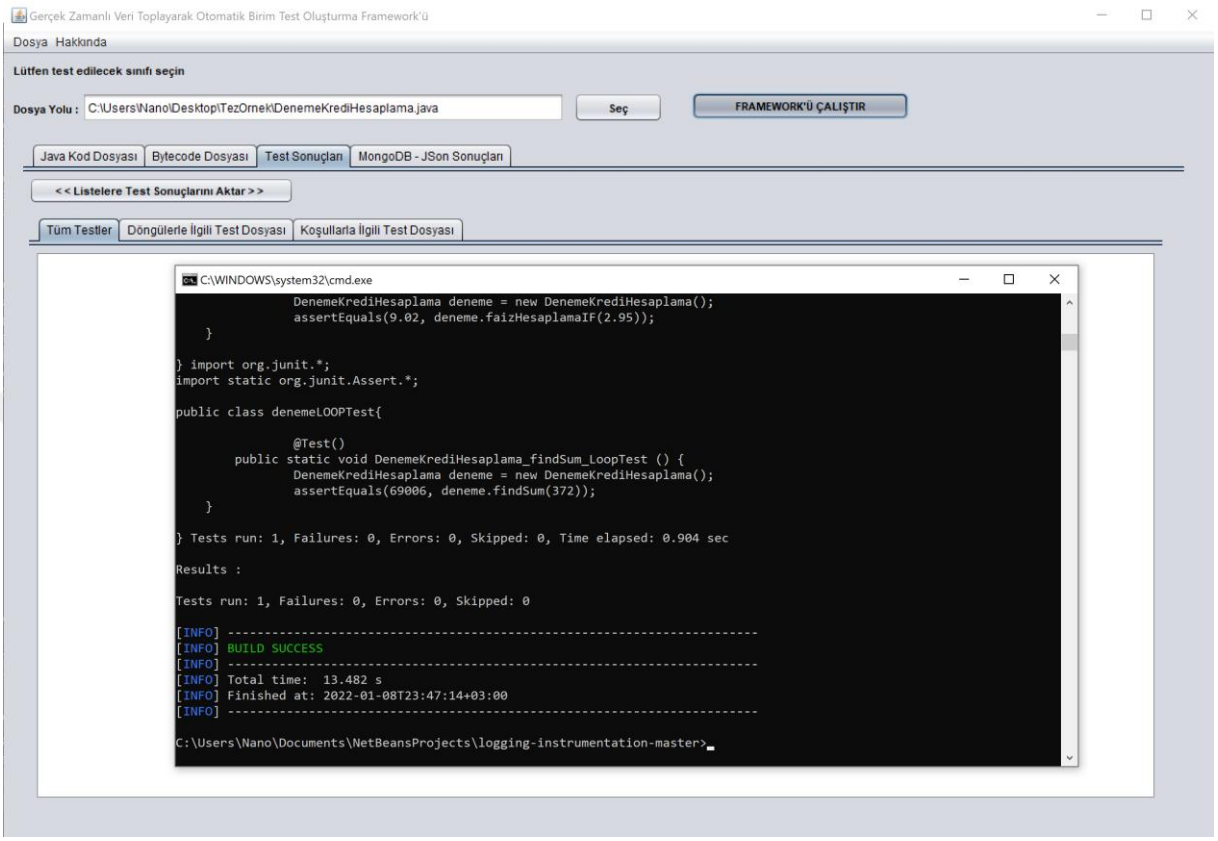
Şekil 4.34. Framework'ün çalışma anı

Maven yapısının çalıştırılmasıyla, yapılandırma dosyasından çektiği verilere göre verilmiş olan sınıfın test kodlarına ait olan 3 farklı test sınıfı örneği oluşturmaktadır. Aynı zamanda Şekil 4.35'de görüldüğü gibi, gerçek zamanda veriler veritabanına kaydedilmektedir.



Şekil 4.35. MongoDB'de kayıtlı olan koleksiyon örnekleri

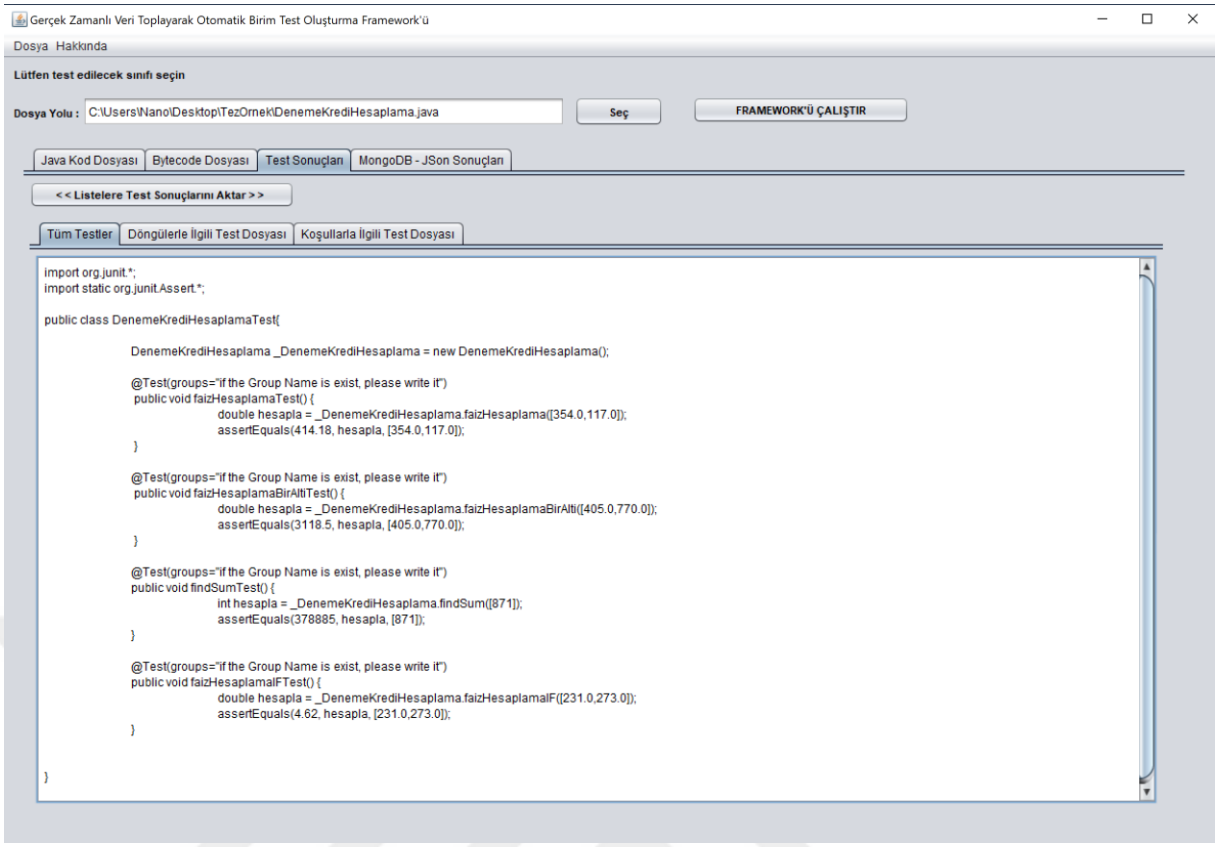
Çıkan test sınıflarına ait dökümanlar hem dosya olarak sistemde kaydedilmektedir hem de Şekil 4.36'da görüldüğü gibi komut satırı ekranında sonuçları başarılı bir şekilde oluşturduğuna dair toplu bir şekilde ekrana yansıtmaktadır.



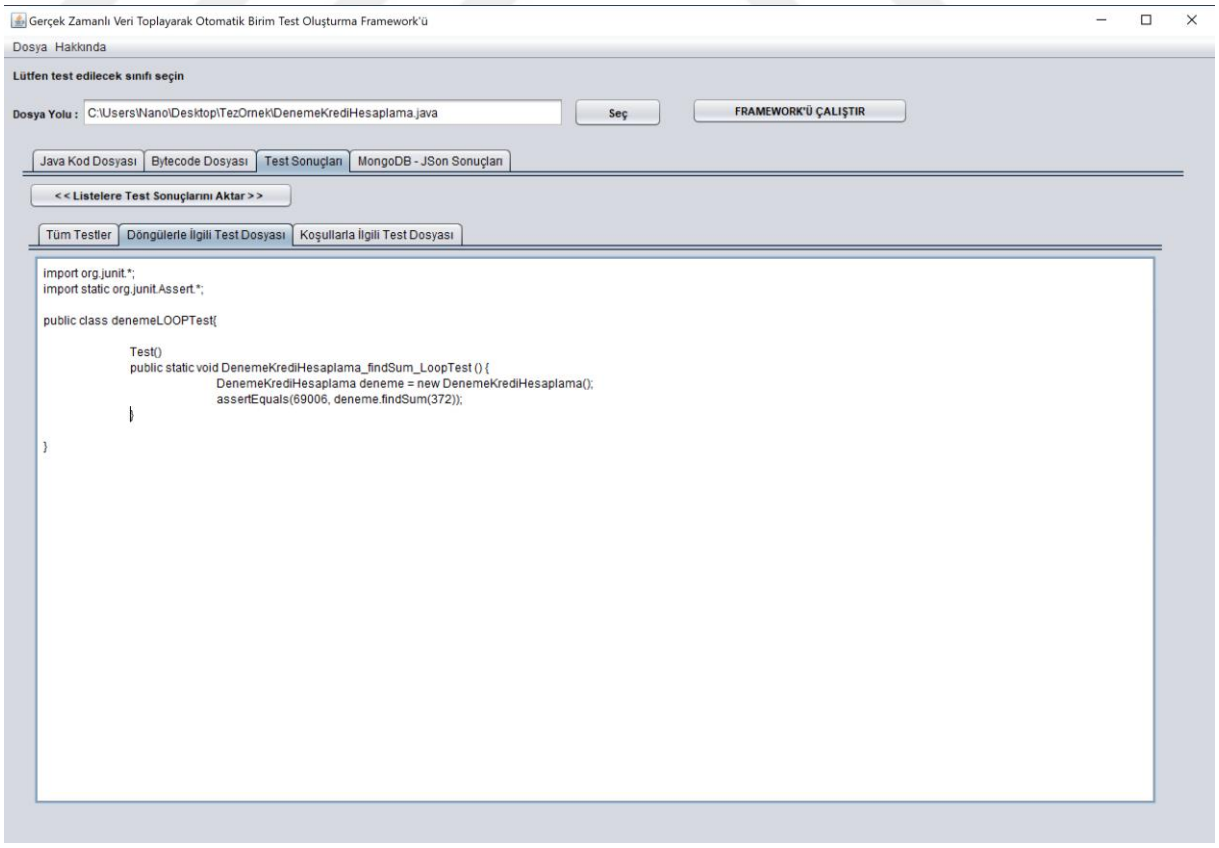
Şekil 4.36. Birim test metot ve sınıflarının başarılı bir şekilde oluşturulması

Komut satırı ekranı kapatıldıktan sonra *Listelere Test Sonuçlarını Aktar* butonu tıklanılır. Oluşturulan ve sonrasında sisteme kaydedilen bu birim test dosyalarıyla ilgili envanterler uygulamanın *Test Sonuçları* isimli üçüncü sekmesinde Şekil 4.37'de görüldüğü gibi ekran çıktısı olarak kullanıcıya sunulmaktadır. Bu ekranda üç adet alt sekme bulunmaktadır. Bu sekmelerden ilki *Tüm Testler* isimli alandır ve bu alanda olası tüm test metotlarına ait kodlar yer almaktadır.

Şekil 4.38'de Döngülerle İlgili Test Dosyası isimli ikinci sekme görülmekdir. Okutulan java sınıfı içerisinde bulunan döngü içeren tüm metotlara ait olası test metotları yer almaktadır.

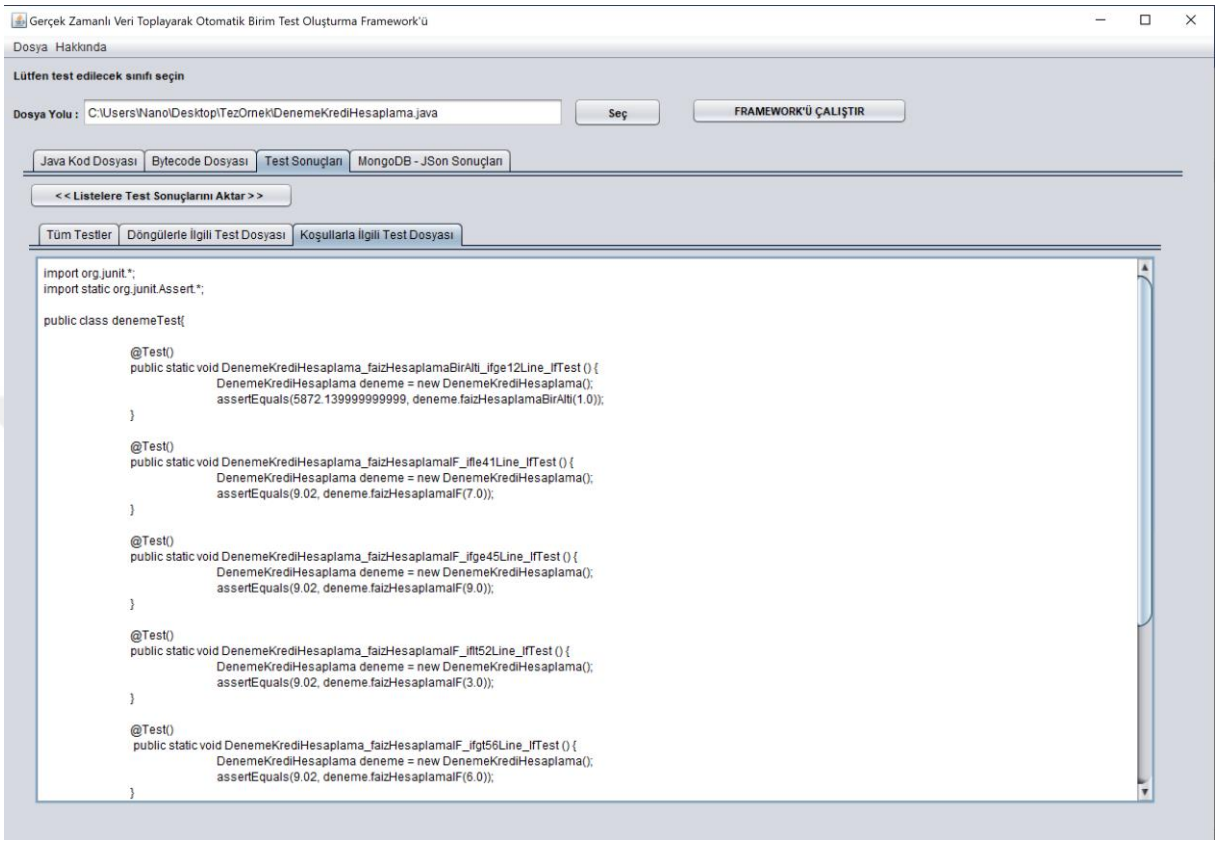


Şekil 4.37. Olası tüm birim test metotlarının görüntülenmesi



Şekil 4.38. İçerisinde döngü yapısı bulunduran bir metodun birim test metodu

Şekil 4.39'da Koşullarla İlgili Test Dosyası isimli üçüncü sekme görülmektedir. Okutulan java sınıfı içerisinde bulunan şart bloklarını içeren tüm metotlara ait olası test metotları yer almaktadır.



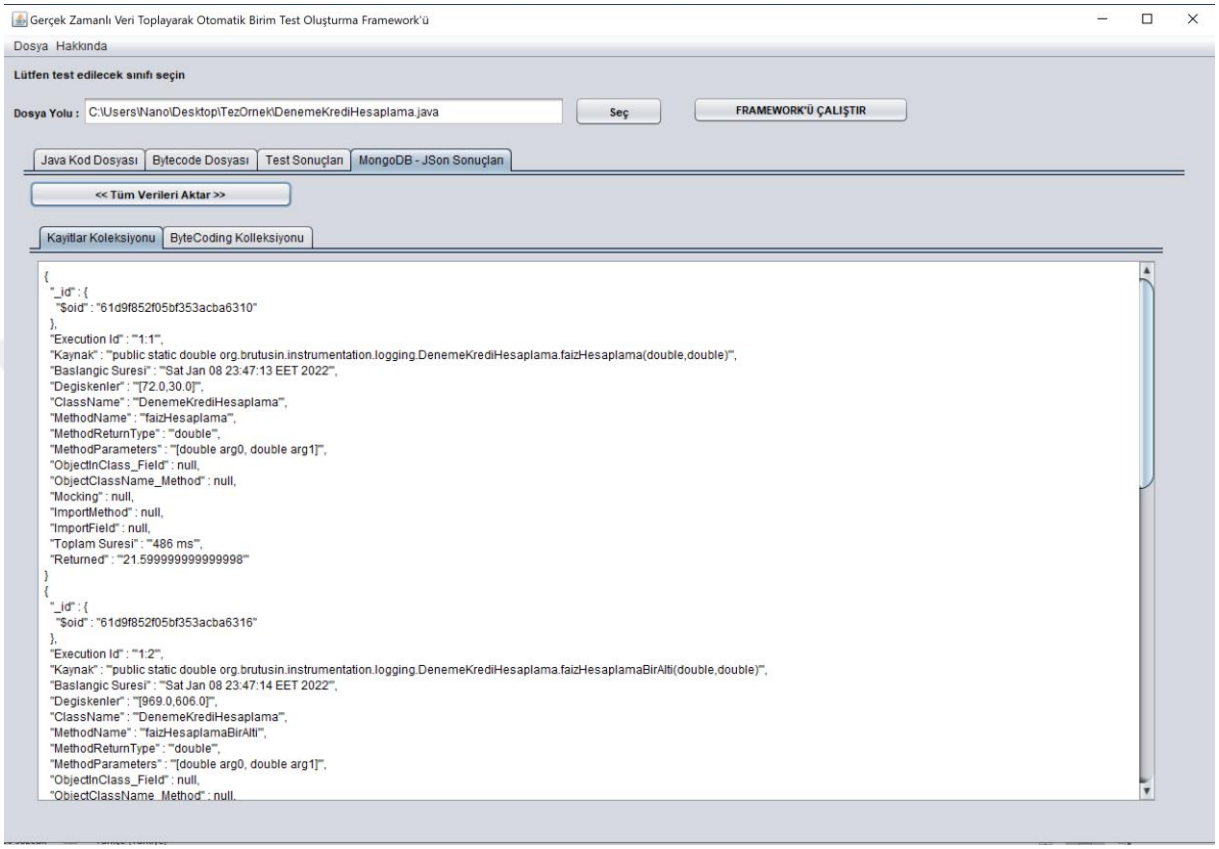
Şekil 4.39. İçerisinde koşul yapısı bulunduran bir metodun birim test metotları

Uygulamanın dördüncü sekmesi ise *MongoDB - JSon Sonuçları* isimli ekrana aittir. Framework aracılığıyla koşul yapıları ve döngülerin de dahil olduğu bir metot veya birden fazla metoda ait olan bir sınıf içerisindeki tüm gerekli değerler gerçek zamanlı olarak veri tabanına kaydedilmiştir.

Arayüzler arasında geçiş yapmamak adına bu sekme altında veri tabanında tutulan her bir NoSQL koduna ait JSON yapı ağaçları yer almaktadır. Bu yapılar aynı zamanda metin belgesi formatında da hiyerarşik bir şekilde tüm test sınıflarının da saklanıldığı dosya lokasyonu altında kaydedilmektedir.

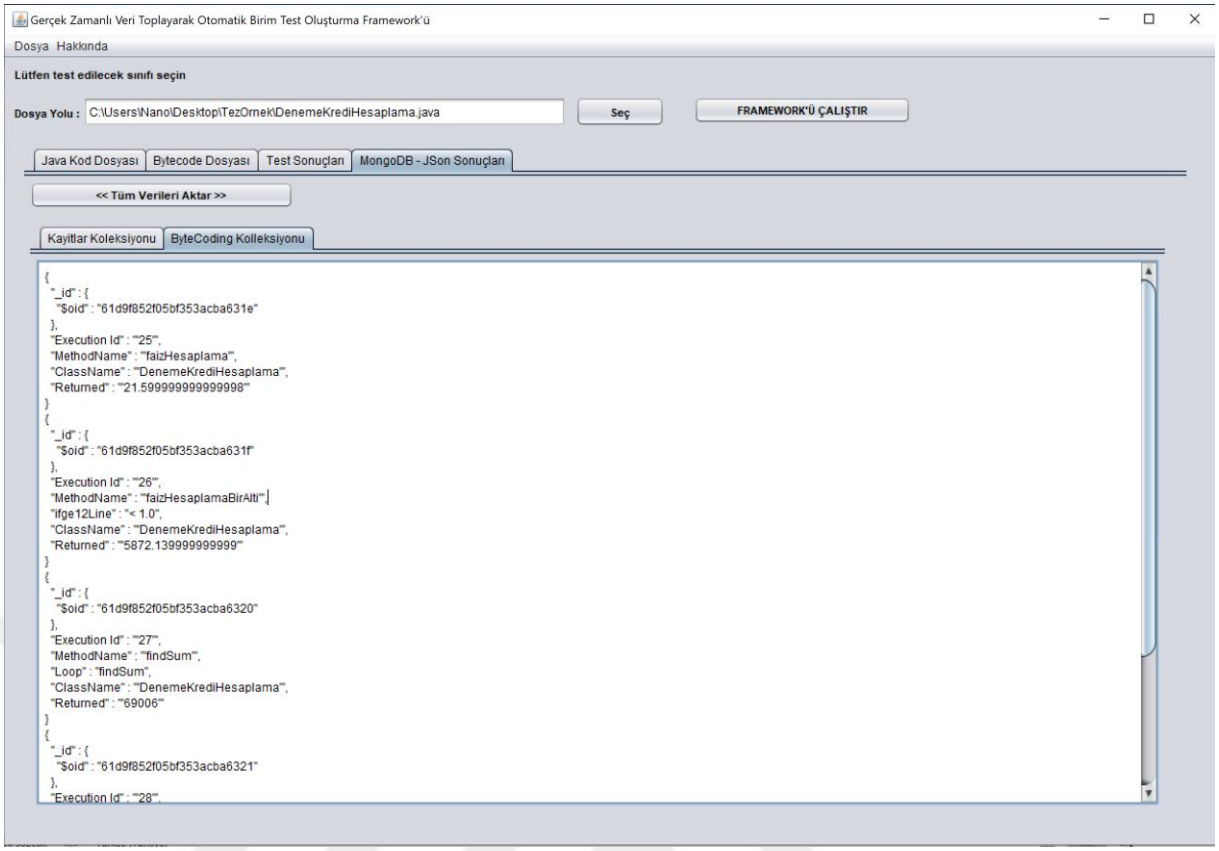
Şekil 4.40'da iki sekme görülmektedir. MongoDB yazılımı içerisinde de iki NoSQL koleksiyon yapısı bulunmaktadır. Şekilde görüldüğü üzere, bunlardan ilki çerçevenin

toplamış olduğu verilerin ne zaman loglandığı, ilgili java sınıfına ait sınıf ismi ve metot isimleri, aldığı parametreler, nesnelerle ile ilgili değerler, sahip oldukları sınıf isimlerine ait isimler, metotlardan dönen değerler, mocking var mı yok mu gibi bilgilerin yer aldığı *kayitlar* koleksiyonuna ait listedir.



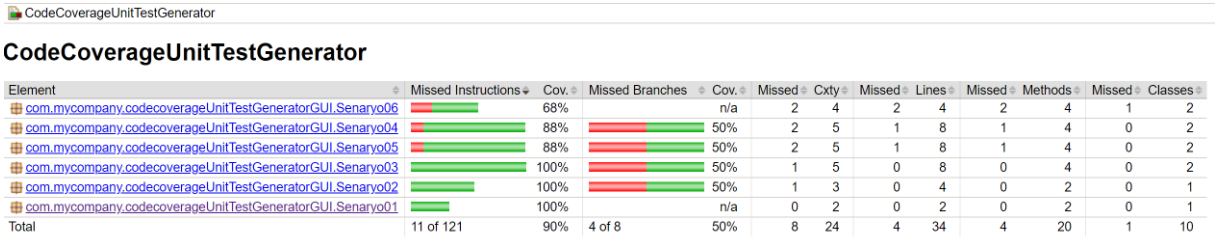
Şekil 4.40. MongoDB'de yer alan kayıtlar koleksiyonun JSon formatı

Şekil 4.41'de İkinci koleksiyon *byteCoding*'e ait veriler görülmektedir. Java sınıfı içerisindeki her bir metoda ait döngü ve koşul yapılarının analizi ayrı ayrı yapılıp bunlarla ilgili değerlerin özet bilgileri yine ayrı ayrı olacak şekilde toplanılmış ve kaydedilmiştir.



Şekil 4.41. MongoDB'de yer alan bytecoding koleksiyonun JSon formatı

Şekli 4.42'de bu yazılımdan üretilmiş test senaryolarının kod kapsamına ait sonuçlarının ekran görüntüsü verilmiştir.

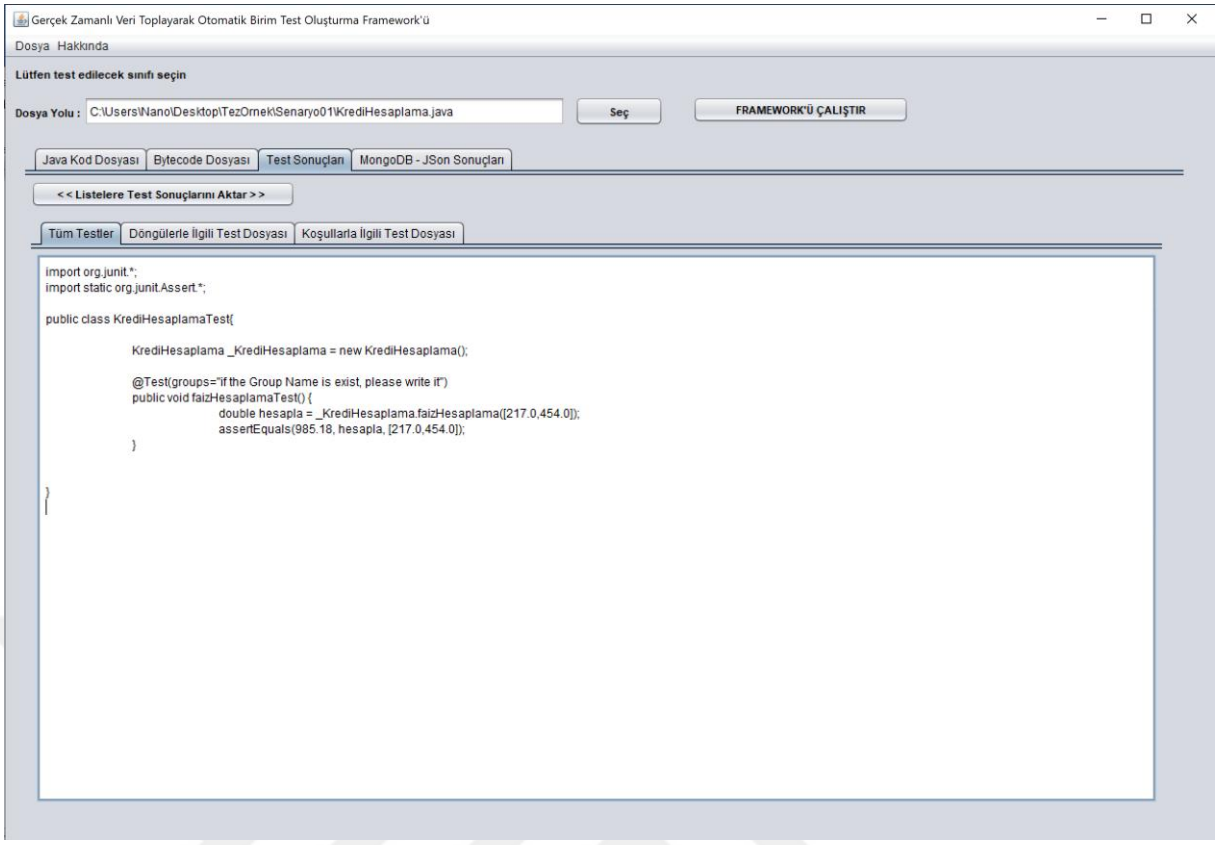


Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo06		68%		n/a	2	4	2	4	2	4	1	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo04		88%		50%	2	5	1	8	1	4	0	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo05		88%		50%	2	5	1	8	1	4	0	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo03		100%		50%	1	5	0	8	0	4	0	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo02		100%		50%	1	3	0	4	0	2	0	1
com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo01		100%		n/a	0	2	0	2	0	2	0	1
Total	11 of 121	90%	4 of 8	50%	8	24	4	34	4	20	1	10

Şekil 4.42. Senaryoların kod kapsamı sonuçları

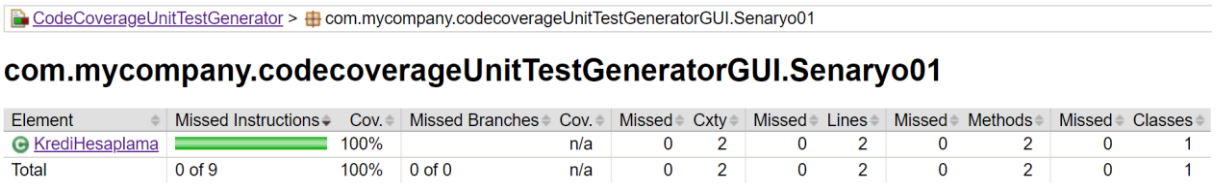
4.3.1. Method İçerisindeki Alternatif Durumlar Senaryosunun Gerçekleştirilmesi

Metot içerisindeki alternatif durumlar başlığı altında verilen senaryolar bu bölüm başlığı altında analiz edilmiştir. Bu durumda, öncelik olarak Şekil 4.2'de görüldüğü üzere, basit seviyede oluşturulmuş olan bir sınıf ve bir metot yapısının çıktısı Şekil 4.43'de yer almaktadır.



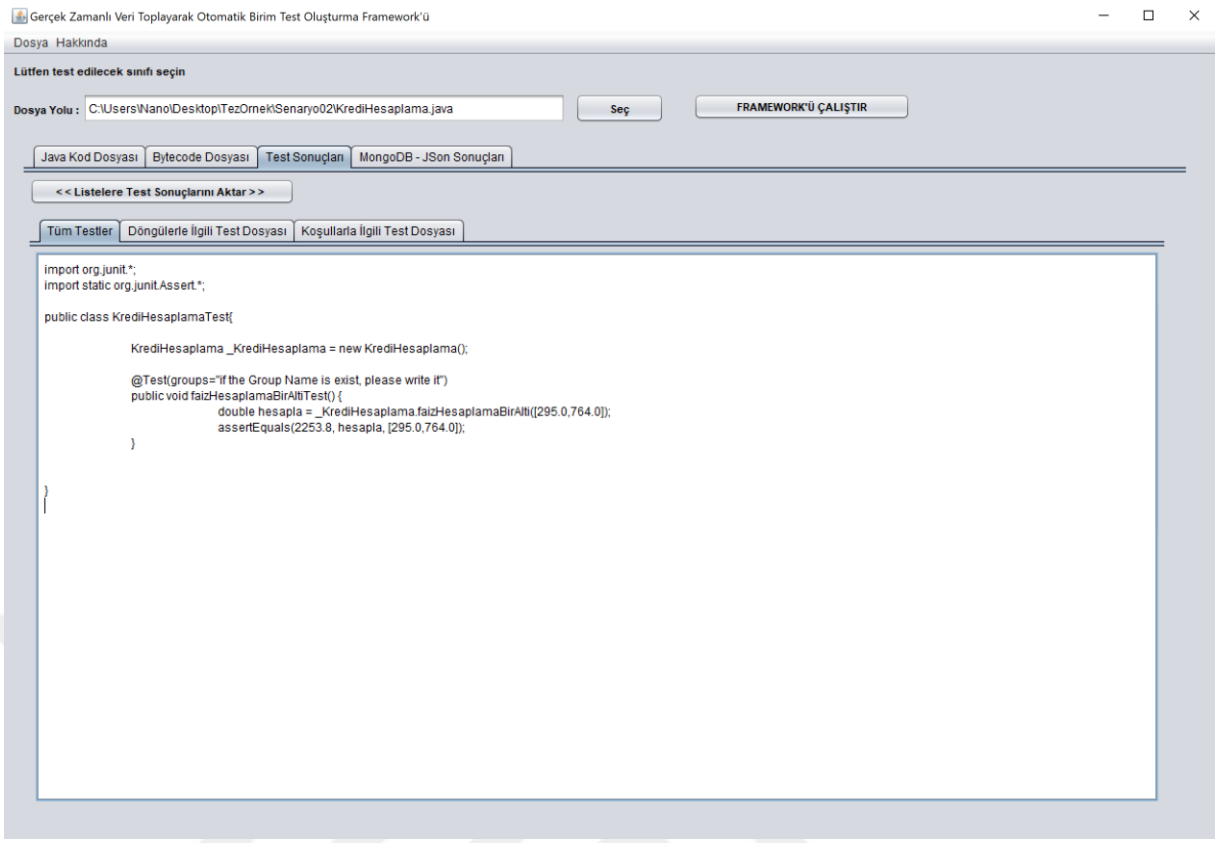
Şekil 4.43. Senaryo 1'in gerçekleştirilmesi

Şekli 4.44'de ilk senaryonun kod kapsamı sonucu verilmiştir. Bu sonuca göre, üretilen test kodunun yüzdelik değeri ile başarılı bir test üretildiği görülmektedir.



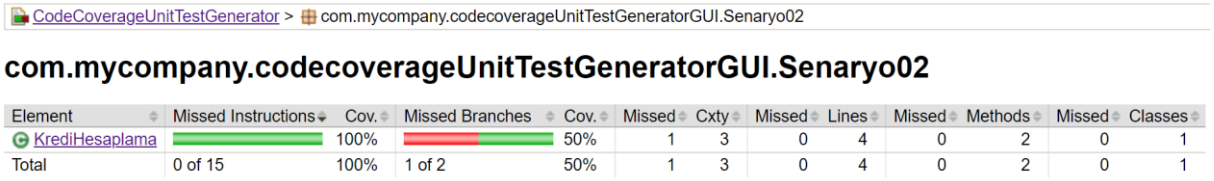
Şekil 4.44. Senaryo1'in kod kapsamı sonucu

Her zaman basit bir metot ile çalışılmamaktadır. Sınıf içerisinde alternatif başka metot yapıları da yer almaktadır. Bu sebepten dolayı, Şekil 4.4'te yer alan metot içerisindeki koşullu yapı sayesinde test yazılımcısı test senaryosunu hazırlarken alternatif yollara başvurmaktadır. Bu koşullu yapı kullanıcıya birim testi yazılacak metodun doğru çalışıp çalışmayacağı hakkında bir nevi yol göstermektedir. Bununla ilgili otomatik birim test oluşturma yazılımının vermiş olduğu çıktı Şekil 4.45'de yer almaktadır.



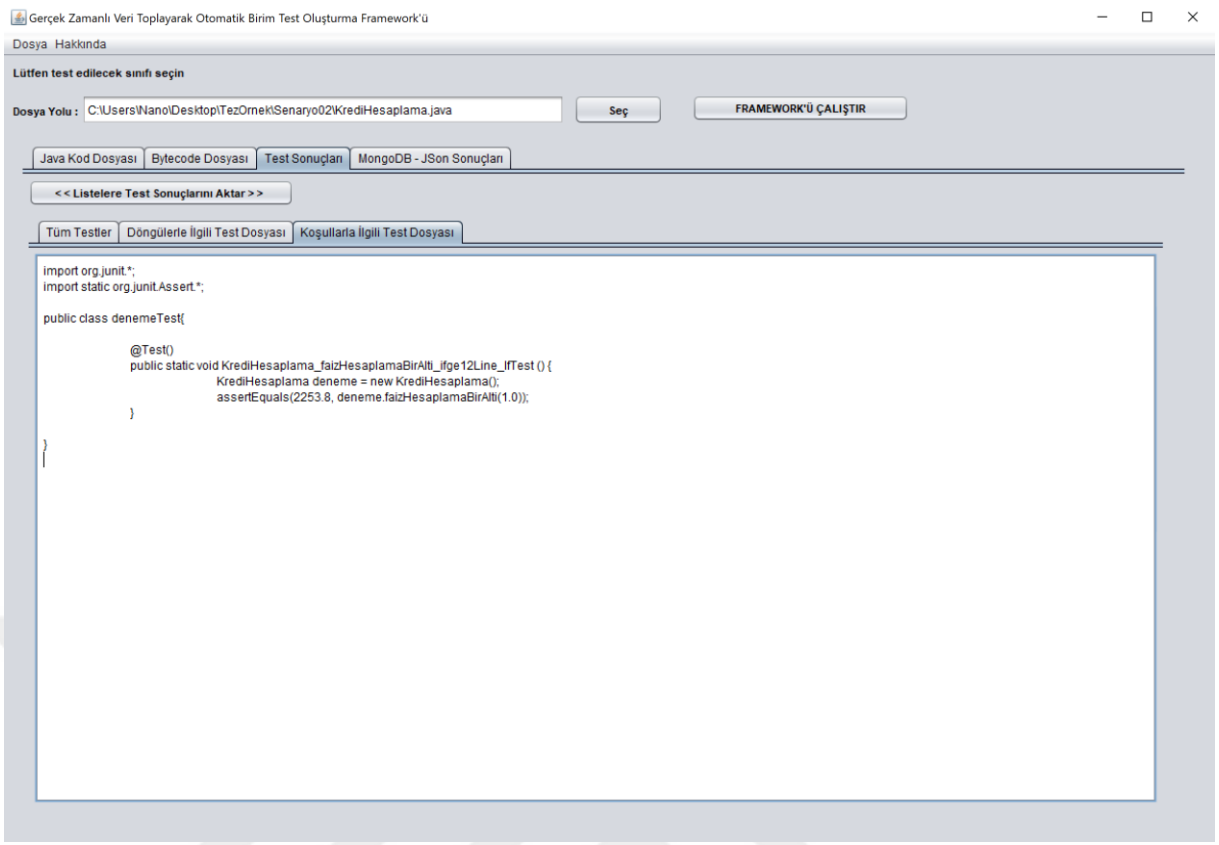
Şekil 4.45. Senaryo 2'nin gerçekleştirilmesi

Şekli 4.46'da ikinci senaryonun kod kapsamı sonucu verilmiştir. Bu sonuca göre, üretilen test kodunun yüzdelik değeri ile başarılı bir test üretildiği görülmektedir. Bu senaryoda koşul yapısı kullanıldığı için missed branches sütununda yüzdelik değer değişmiştir.



Şekil 4.46. Senaryo2'nin kod kapsamı sonucu

Şekil 4.47'de görüldüğü gibi, metot içerisindeki koşul ifadesi ile hem if bloğu ile ilgili yapı gözetilmiş hem de normal bir birim test metodu için örnek verilmiştir.



Şekil 4.47. Senaryo 2 için olası koşul yapısının birim test metodu

Bu bölümün analiz sonuçları özetlenecek olursa, bir metot içerisindeki kullanılan nesneler, döngüler ve koşul cümleleri kontrol edilerek ilgili verileriyle birlikte çalışma anında kayıt altına alınmış ve birim test kodu oluşturulmuştur. Bununla birlikte, bir sınıfta birden fazla metot olduğu zaman yine her bir metodun içerisindeki tüm değerler kontrol edilerek her biri için çözümler geliştirilmiştir.

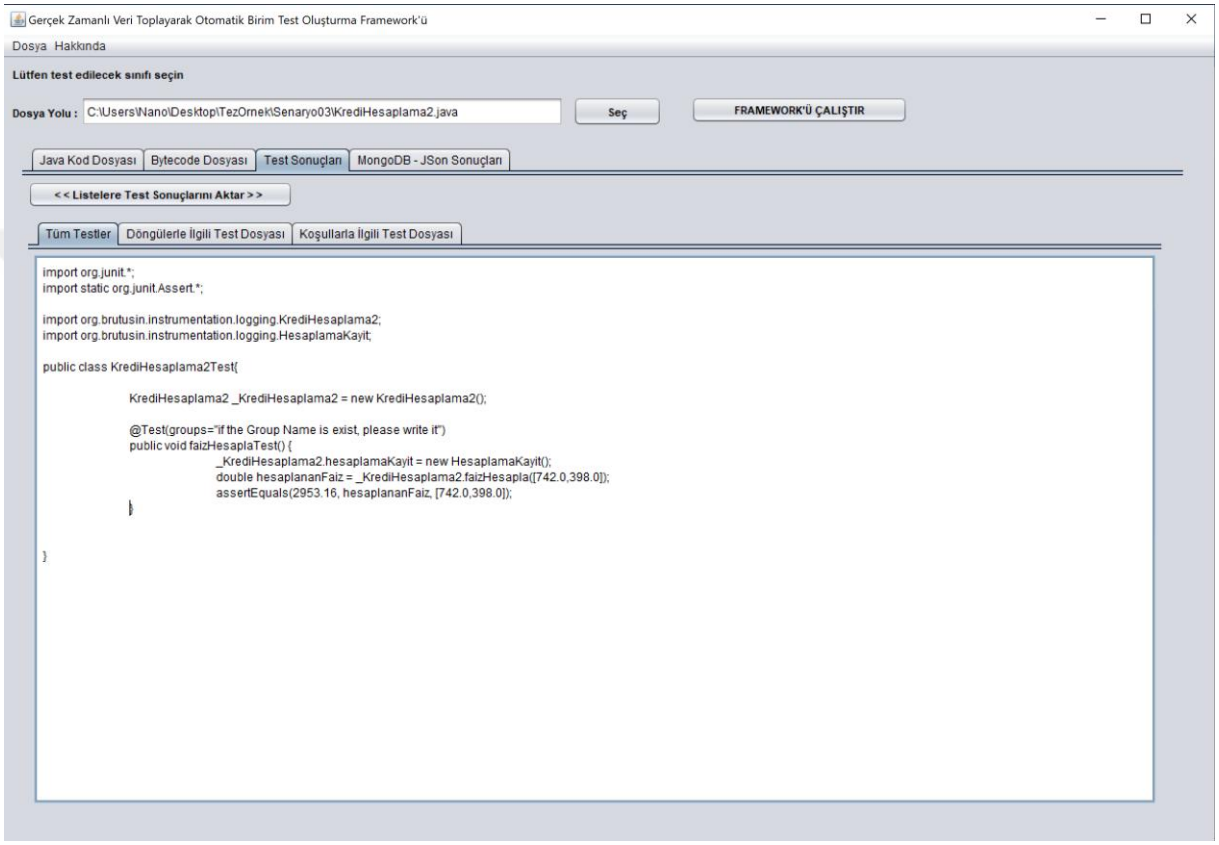
4.3.2. Method İçerisindeki Farklı Nesneler Senaryosunun Gerçekleştirilmesi

Metot içerisindeki farklı nesneler başlığı altında verilen senaryolar bu bölüm başlığı altında analiz edilmiştir. Bu durumda, öncelik olarak Şekil 4.6'da yer alan metot içerisinde farklı nesneler kullanılmıştır. Bu metot da her bir hesaplama için hesaplamaları arşivlemeye yarayan bir kayıt sınıfı bulunmaktadır. Aynı zamanda, *faizHesapla* yönteminin *HesaplamaKayit* sınıfna ait bir bağımlılığı bulunmaktadır. Yani *HesaplamaKayit* sınıfının aslında sistemde hazır bir şekilde var olması gerekir.

Metot içerisindeki bu senaryo için iki farklı yöntemle birim testi hazırlanabilir. Bu yöntemler, mock ve stub nesnelerinin kullanımı ile gerçekleşmektedir. Şekil 4.48'de görüldüğü üzere stub yöntemi kullanılmıştır. Metot içerisindeki nesne oluşturulmaktadır. Bunun için,

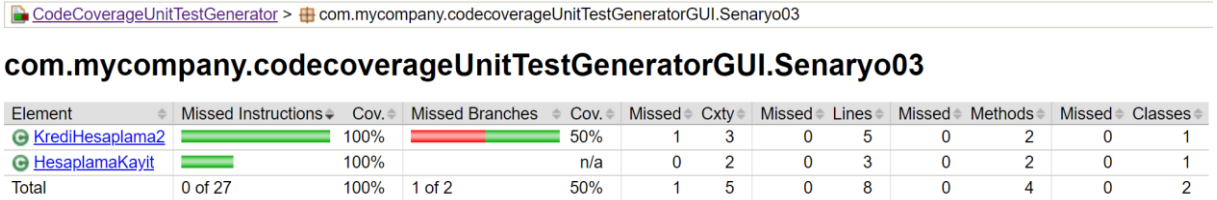
_krediHesaplama2.hesaplamaKayit = new HesaplamaKayit();

satırı eklenmiştir.



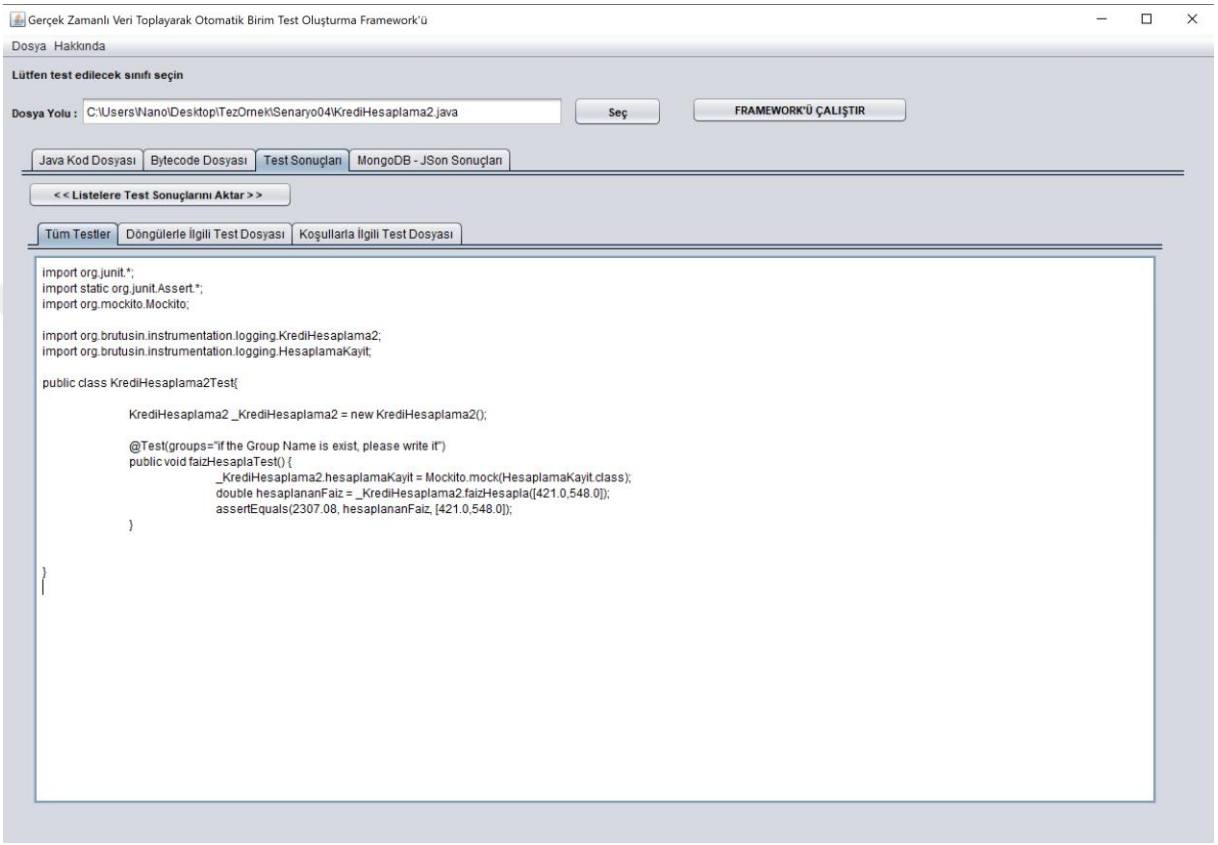
Şekil 4.48. Senaryo 3'ün gerçekleştirilmesi

Şekli 4.49'da üçüncü senaryonun kod kapsamı sonucu verilmiştir. Burada, stub yapısı kullanıldığından dolayı *KrediHesaolama2*'nin yüzdelik değeri düşük gösterilmektedir.



Şekil 4.49. Senaryo3'ün kod kapsamı sonucu

Şekil 4.50'de JUnit ile birlikte Mockito kütüphanesinin kullanımı gösterilmiştir. Bu kütüphanenin eklenmesiyle birlikte aynı zamanda _KrediHesaplama2.hesaplamaKayit = Mockito.mock(HesaplamaKayit.class) satırı da test metodu içerisine eklenmiştir. Böylelikle nesne sahte bir şekilde kullanıma hazırlanmıştır.



Şekil 4.50. Senaryo 4'ün gerçekleştirilmesi

Şekli 4.51'de dördüncü senaryonun kod kapsamı sonucu verilmiştir. Burada, mocklama yapıldaktan sonra hem *KrediHesaplama2*'nin yüzdelik değeri hem de *HesaplamaKayit*'in değeri düşük görülmektedir.

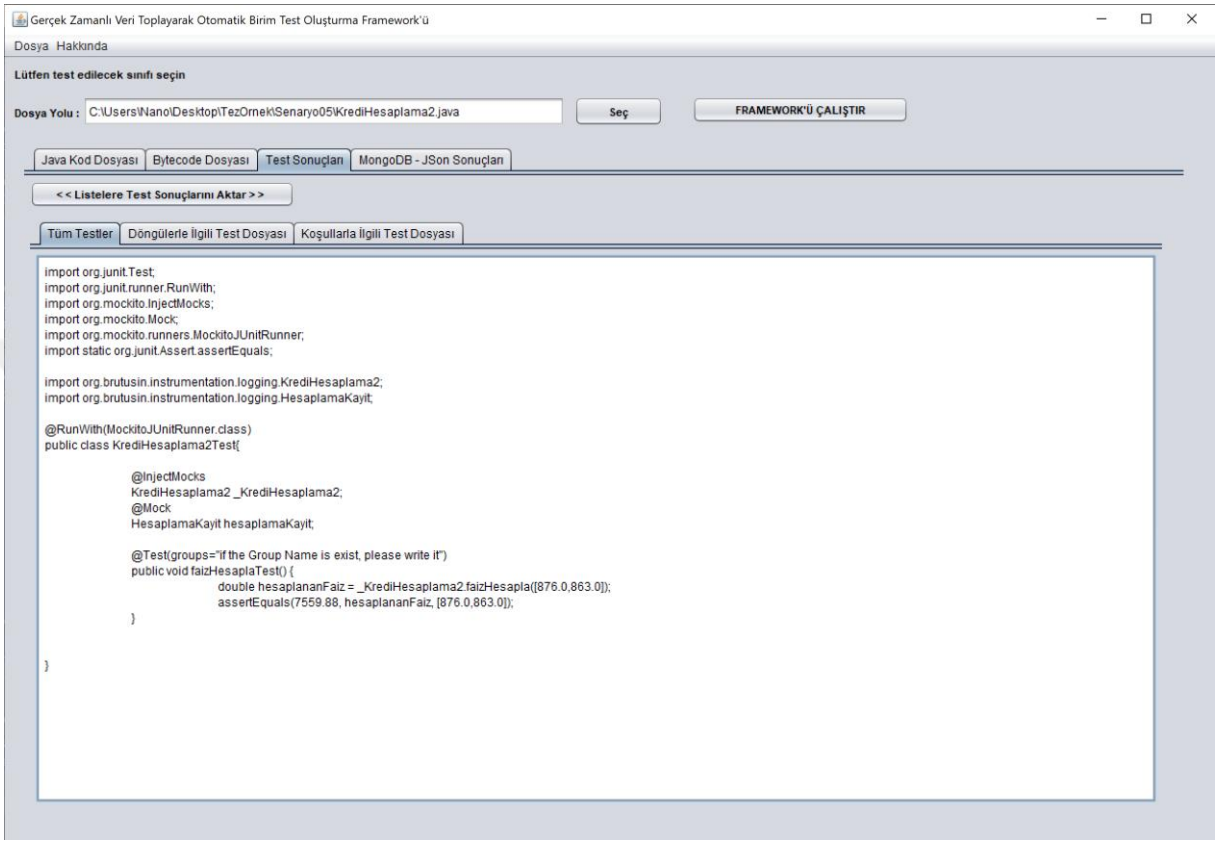
CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo04

com.mycompany.codecoverageUnitTestGeneratorGUI.Senaryo04

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HesaplamaKayit		57%		n/a	1	2	1	3	1	2	0	1
KrediHesaplama2		100%		50%	1	3	0	5	0	2	0	1
Total	3 of 27	88%	1 of 2	50%	2	5	1	8	1	4	0	2

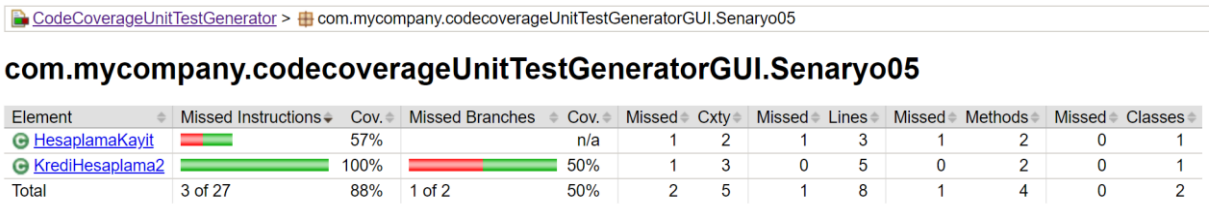
Şekil 4.51. Senaryo4'ün kod kapsamı sonucu

Mock yönteminin diğer bir kullanımı ise annotation'larla yapılmaktadır. Mockito, direkt olarak nesneyi annotation'lar ile işaretler ve sahteleme işlemini gerçekleştirir. Şekil 4.52'de kullanılması olası nesnelerin ilgili sınıflarından örneklenmesi gösterilmiştir.



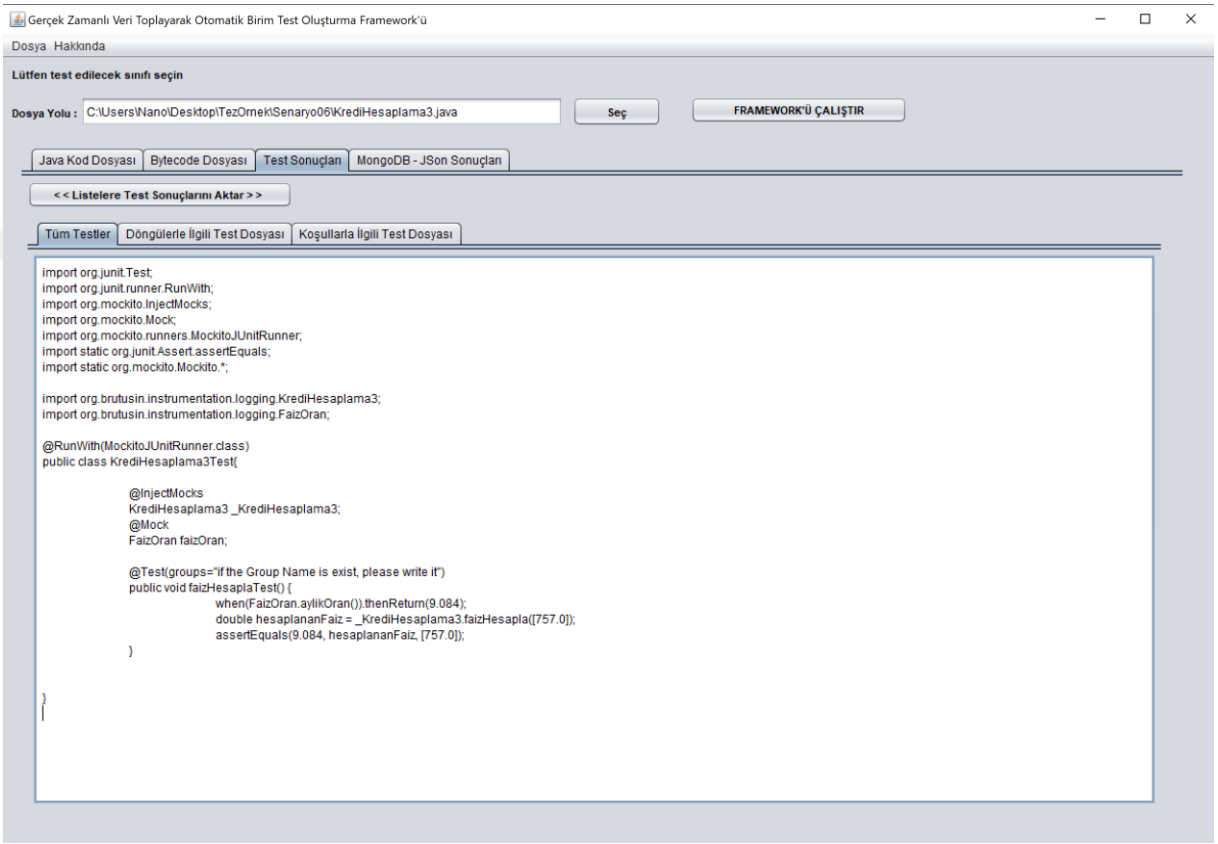
Şekil 4.52. Senaryo 5'in gerçekleştirilmesi

Şekli 4.53'de dördüncü senaryonun kod kapsamı sonucu verilmiştir. Burada, mocklama yapıldaktan sonra hem *KrediHesaplama2*'nin yüzdelik değeri hem de *HesaplamaKayit*'in değeri düşük görülmektedir.



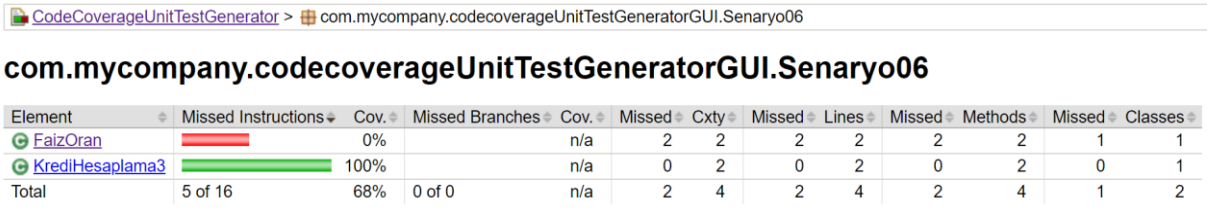
Şekil 4.53. Senaryo5'in kod kapsamı sonucu

Mocklanmış nesnelerde bir diğer dikkat edilmesi gereken özellik, nesnelerin değer dönüşümleridir. Mocklanan nesne içerisinde metotlar ve nesneler varsayılan değerlerini taşımaktadır. Bu sebepten dolayı, zaman zaman birim test ortamında esas değerler oluşturulamaz. Bunu aşabilmek adına, *Given-When-Then* standartları takip edilmektedir. Şekil 4.54'de, bu durumun nasıl bir test senaryosu ile çözüldüğü gösterilmiştir.



Şekil 4.54. Senaryo 6'nın gerçekleştirilmesi

Şekli 4.55'de dördüncü senaryonun kod kapsamı sonucu verilmiştir. Burada, mocklanmış nesnenin varlığı ile FaizOran sınıfına uğranılmadığı düşen yüzdelik orandan yorumlanabilmektedir.



Şekil 4.55. Senaryo6'nın kod kapsamı sonucu

Böylelikle otomatik birim test oluşturmada altı önemli senaryonun nasıl çözüme ulaştığı gösterilmiştir.

Oluşturulan gerçek zamanlı veri toplayarak otomatik birim test oluşturma yazılımı ve bununla birlikte gerçekleştirilen işlemlerin, yaygın olarak kullanılan ve literatürdeki diğer otomatik birim test oluşturma araçları ile kıyaslanması tartışma ve sonuç bölümünde ele alınmıştır.

Geliştirilen bu çalışmaya ait uygulama https://github.com/SevdanurGENC/Nano-Automatic-Unit-Test-Generator adresinde yayınlanmıştır.

5. TARTIŞMA VE SONUÇLAR

Bu çalışmada, Java Agent yardımı ile çalışma zamanında verileri toplayan, topladığı verileri NoSql veri tabanında saklayan ve bir JTL şablon motoru kullanarak bu verileri birim testine dönüştüren bir uygulama geliştirilmiştir. Yapılan çalışmalar ve geliştirilen araç, çeşitli yönlerden literatürdeki geçmiş çalışmalara göre daha farklı bir yapı kullanmaktadır.

Çalışma içerisinde bahsedilen tüm olası test senaryolarına cevap verebilecek bir yapı oluşturulmuştur. Ortaya çıkan test sınıfları, JUnit'te çalıştırıldığında hangi sınıf için test işlemi yapılıyorsa onunla ilgili sonuçlar kolaylıkla alınabilmektedir. Kullanıcılar için kolay bir kullanıma sahip olan bu framework aracılığıyla, alınan birim testler oldukça kısa sürede otomatik bir şekilde oluşturulmaktadır.

Literatürdeki son çalışmalar gösteriyor ki, çalışmalar için hazırlanmış her bir test aracının otomatik olarak test senaryolarını sağlıklı bir şekilde üretmesi amaçlanmıştır. Yapılan çalışmaların çoğu Java programlama dili ile hazırlanmışken, C/C++ ve C# gibi programlama dilleri de geri kalan kısmını oluşturmaktadır. Bunların birçoğu masaüstü uygulamayken, geri kalan kısmı da web uygulaması veya eklenti olarak geliştirilmiştir. Genellikle bu test senaryoları programların çalışmasını kontrol etmek için rastgele üretilir. Rastgele test ve türevlerinin geliştirilmesini için, çoğunlukla dinamik sembolik yürütmeye dayalı teknikler (dynamic symbolic execution) tercih edilmiştir.

Literatürde bulunan önemli çalışmalardan biri olan Pex, C# kodunun birim testi için geliştirilmiş bir araçtır. Dinamik sembolik yürütme tekniğini kullanarak, test senaryolarına farklı parametreli test girdileri üretmektedir. Aynı zamanda, metotların dönüş değerlerine dayalı sonuçlar da üretmektedir. Ancak karmaşık metot dizileri gerektiren sınıflara göre sınırlıdır. Java programlama dilinde yapılan literatürdeki diğer önemli çalışmalara ise Randoop ve EvoSuite örnek olarak verilebilir. Randoop, kullanım kolaylığı ile bilinir, ancak EvoSuite'in aksine karmaşık kod yapılarını rehberlik olmadan test edememektedir. Aynı zamanda, yüksek kod kapsamı içeren kompakt test senaryolarını üretmeyi de amaçlamaktadır. Kod kapsamını kullanırken, yaygın bir sistematik yaklaşım, her seferinde bir kapsama hedefi seçmek (örneğin,

bir kontrol akışı) ve bu özel hedefi uygulayan bir test senaryosu üretmektedir. Bayt kod API'siyle çalışarak bu tekniği geliştirmişlerdir. Bu tez çalışmasında, koşul ve döngü bloklarına ait her bir kontrol akışı bayt kodu üzerinde geliştirilmiş dize ayrıştırma yöntemi ile kontrol edilmiştir. Tüm şartlara uygun olan çözümlerle birlikte ayrı ayrı otomatik birim test metotları oluşturulmuştur. Aynı zamanda, Java sınıfı içerisinde tanımlanan tüm nesneler bayt kod tarafından tek tek belirlenir. Birim test üretimi sırasında bu nesneler aktarılırken seçilen test metodolojisine uygun olacak şekilde mock veya stub yöntemi kullanılıp kullanılmayacağına karar verilir. Birim teste ait kodların çıktısı hazırlanırken seçilen metodolojiye göre notation işlemleri de otomatik bir şekilde oluşturulmaktadır.

TestFul ve eToc araçlarının her ikisi de yapısal kapsamı en üst düzeye çıkarmak için JUnit test senaryoları oluşturmayı hedefleyen arama tabanlı bir yaklaşım kullanmıştır. Ancak, eToc birkaç yıldır güncellenmemektedir. Bu nedenle, test verisi oluşturabilmek için en son gelişmeleri içermemektedir. Öte yandan TestFul, birçok kritik ayrıntıda EvoSuite'ten farklıdır ve tam otomatik bir özelliğe sahip değildir. Örneğin, TestFul test edilen her sınıf için XML dosyalarının manuel olarak düzenlenmesini gerektirir. Bu çalışma kapsamında ise, hem uygulamaya dışarıdan alınan Java sınıfının içerisinde kullandığı veriler hem de bu verilere benzer yaklaşım sergileyen rastgele veriler üretilerek birim test için gerekli metotlar içerisinde parametre olarak kullanılmıştır. Tüm bu işlemler, çalışma zamanında hem sisteme bir yedekleme dosyasına hem de JSON yapısında bir NoSql koleksiyonuna kaydedilmektedir.

Charreteur vd. bayt kod kullanmış oldukları çalışmalarında, java sanal makinesinde sınırlı bellek değişkeni yöntemini kullanmışlardır. Bytecode düzeyinde giriş üretimini test eden JAUT isimli uygulamaları, Java bayt kodundan kısıtlama tabanlı test girişi üretimi gerçekleştirir. Bu nedenle, esas olarak JPF, Cute ve Pex isimli diğer çalışamalar ile ilişkilidir. Bu üç araçtan farklı olarak, JAUT geriye doğru keşif uygular, yani bir hedef bayt kodu konumundan başlar ve adım adım girişe doğru uygun bir yol keşfeder. Aslında, JPF, Cute ve Pex, yürütme ile aynı sırada bir yol boyunca talimatları sembolik olarak değerlendirmeyi içeren ileri sembolik yürütmeye dayanır. Bu çalışma kapsamında, bayt kod'ları dize ayrıştırma yöntemiyle dönüştüren bir sistem oluşturarak bu çalışmalara farklı bir bakış açısı getirilmiştir.

Java sınıfının java bayt koda dönüşümü sonrasında elde edilen her bir opcode satırı sırasıyla analiz edilmiş ve opcode'a ait varsa eğer nesne, değişken ya da giriş-çıkış parametreleri tespit edilmiştir. Bu değerler daha sonra birim testi üretiminde kullanılmıştır.

Assertion konusu ele alındığında, Randoop annotation oluşturmak için kullanılacak gözlemci metotlarını belirlemek adına kaynak kodun açıklamalarına izin verir. Orstra, gözlemlenen dönüş değerlerine ve nesne durumlarına dayalı olarak assertion üretir ve bu gözlemlere karşı gelecekteki çalışmaları kontrol etmek için assertion ekler. Bu tür yaklaşımlar verimli nesne üretmek için kullanılabilse de bu assertion'lardan hangisinin gerçekten yararlı olduğunu belirlemeye hizmet etmezler ve bu nedenle bu tür teknikler yalnızca regresyon testinde kontrol edilebilir. Buna karşılık μTest aracı, EvoSuite aracılığıyla etkili bir assertion alt kümesini seçmek için mutasyon testi kullanır. Bu çalışma kapsamında ise, uygulama aracılığıyla kullanıcıdan test senaryosuna ait isteğe bağlı olacak şekilde bir listeden metodoloji seçilebilmektedir. Bu metodolojiye uygun olacak şekilde annotation'lar birim test haline getirilirken FTL aracılığıyla çıktı haline dönüşmektedir.

Bu tez çalışması kapsamında geliştirilen aracın aynı zamanda ulusal yazılım test alanında önemli bir yer alacağı düşünülmektedir. Geliştirilen aracın, yazılım testçileri tarafından gerçekleştirilecek birim testlerinde aktif olarak kullanılması amaçlanmaktadır. En temel test senaryolarına mevcut haliyle cevap verebilen bu araç, bayt kodu tabanlı çalışan bir çerçeveye sahip olduğu için çok daha ileri seviye senaryolarda nasıl davranması gerektiği ile ilgili geliştirilmeye müsait bir yapıya sahiptir. Bunun en büyük sebeplerden birisi de Java'nın açık kaynak kodlu bir sisteme sahip olmasıdır. Bu framework için, gelecekte gerekli olabilecek test senaryolarında ilgili diğer bayt kodları çevirecek farklı modüller de geliştirilebilir.

KAYNAKLAR

Albert, E., Miguel, G.-Z., & Germán, P. (2010). "PET: a partial evaluation-based test case generation tool for Java bytecode.". Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation.

Anouti, M. (2018). *Introduction to Java Bytecode. 2018,*. 06 22, 2020 tarihinde https://dzone.com/articles/introduction-to-java-bytecode adresinden alındı

Balakrishan, S. (2020). *Understanding Java Agents 2020.* 12 10, 10.12.2021 tarihinde https://dzone.com/articles/java-agent-1 adresinden alındı

Blanco, D. S. (2010). *What is a Java Agent and what is it for?* 12 10, 2021 tarihinde https://medium.com/nerd-for-tech/what-is-a-java-agent-and-what-is-it-for-7a7896729c76 adresinden alındı

Charreteur, F., & Arnaud, G. (2010). "Constraint-based test input generation for java bytecode.". IEEE 21st International Symposium on Software Reliability Engineering. IEEE.

Chodorow, K. (2013). MongoDB: the definitive guide: powerful and scalable data storage. USA: O'Reilly Media Publisher, 3-5.

Conroy, K. (2007). "Automatic test generation from GUI applications for testing web services.". IEEE International Conference on Software Maintenance. IEEE.

Csallner, C., & Yannis, S. (2004). JCrasher: an automatic robustness tester for Java. Software: Practice and Experience 34.11 (2004): 1025-1050.

Csstricks. (2021). 01 06, 2021 tarihinde https://tr.csstricks.net/8222637-what-is-black-box-testing-techniques-example-and-types adresinden alındı

de Andrade, F. (2012). "Specification-driven unit test generation for java generic classes.". International Conference on Integrated Formal Methods. Springer, Berlin, Heidelberg.

Debnath, M. (2020). *What are Java Agents?* 12 10, 10.12.2021 tarihinde https://www.developer.com/design/what-is-java-agent/ adresinden alındı

Fraser, G., & Andrea, A. (2011). "Evosuite: automatic test suite generation for object-oriented software.". Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

FreeMarker. (2021). *What is Apache FreeMarker?* 12 10, 2021 tarihinde https://freemarker.apache.org adresinden alındı

Gökalp, G. (2015, 9 6). 12 31, 2021 tarihinde https://www.gokhan-gokalp.com/entegrasyon-integration-testi-nedir-ve-tipleri-nelerdir/ adresinden alındı

Java bytecode. (2020). 06 22, 2020 tarihinde https://www.ibm.com/developerworks/library/it-haggar_bytecode/ , https://blog.fearcat.in/a?ID=00250-f337a089-e2a1-48aa-8f90-81a2a44162a2 adresinden alındı

Jovanović, I. (2006). Software testing methods and techniques. The IPSI BgD Transactions on Internet Research, 30.

Karal, Ö. (2004). JAVA ortamında bulanık mantık kontrol: Kamyon yükleme-boşaltma uygulaması. Master's thesis, Pamukkale Üniversitesi Fen Bilimleri Enstitüsü.

Keifer, J. (2021). *Programming on Linux Part 2: C++, Java, .Net Programming on Linux.* 06 22, 2020 tarihinde http://www.techlila.com/write-programs-linux/ adresinden alındı

Khan, M., & Khan, F. (2012). A comparative study of white box, black box and grey box testing techniques. Int. J. Adv. Comput. Sci. Appl, 3(6).

Ma, L. (2015). "Grt: Program-analysis-guided random testing (t).". 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015.

Pacheco, C. (2007). "Feedback-directed random test generation.". 29th International Conference on Software Engineering (ICSE'07). IEEE.

Pacheco, C., & Michael, E. D. (2007). Randoop: feedback-directed random testing for Java. Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.

Păsăreanu, C. (2013). "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis.". Automated Software Engineering 20.3: 391-425.

Prasetya, I., & Wishnu, B. (2015). "T3i: A tool for generating and querying test suites for java.". Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.

Puls, A. (2014). *Diving Into Bytecode Manipulation: Creating an Audit Log With ASM and Javassist.* 12 10, 2021 tarihinde https://newrelic.com/blog/best-practices/diving-bytecode-manipulation-creating-audit-log-asm-javassist adresinden alındı

Sakti, A., Gilles, P., & Yann-Gaël, G. (2014). "Instance generator and problem representation to improve object oriented code coverage.". IEEE Transactions on Software Engineering 41.3 (2014): 294-313.

Sen, K. (2007). Cute :A concolic unit testing engine for c and java. Web, [cited at p.39, 44]. http://osl.cs.uiuc.edu/~ksen/cute/ adresinden alındı

Simons, A. J. (2007). JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. Automated Software Engineering 14.4: 369-418.

Smeets, N., & Simons, A. J. (2011). Automated unit testing with Randoop, JWalk and µJava versus manual JUnit testing. . Research report, Department of Computer Science, University of Sheffield/University of Antwerp, Sheffield, Antwerp.

Solheim, J. A., & Rowland, J. H. (1993). "An empirical study of testing and integration strategies using artificial software systems.". IEEE Transactions on Software Engineering 19.10: 941-949.

Tanno, H. (2015). "TesMa and CATG: automated test generation tools for models of enterprise applications.". IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 2. IEEE, 2015.

Tufano, M. (2020). "Unit Test Case Generation with Transformers.". arXiv preprint arXiv:2009.05617.

Vaish, G. (2013). Getting started with NoSQL. Vol. 2103. Packt Publishing.

Venners, B. (1996). *Bytecode basics : A first look at the bytecodes of the Java virtual machine.* 06 22, 2020 tarihinde https://www.javaworld.com/article/2077233/bytecode-basics.html adresinden alındı

Xie, T., & David, N. (2006). "Tool-assisted unit-test generation and selection based on operational abstractions.". Automated Software Engineering 13.3: 345-371.

Yoshida, H. (2017). "KLOVER: Automatic test generation for C and C++ programs, using symbolic execution.". IEEE Software 34.5: 30-37.

ISTQB Foundation Syllabus ,International Software Testing Qualification Board,2011.

MUSTAFA, K. ve KHAN, R.A., Software Testing: Concept and Practices, India, Lucknow, 5-21, 227-228, 2007.

What is Apache FreeMarker? freemarker.apache.org Erişim Tarihi : 10.12.2021

EKLER

EK A. Projede kullanılan bazı bayt kodlar ile ilgili tablo.

Mnemonic	Opcode (in hex)	Opcode (in binary)	Other bytes [count]: [operand labels]	Stack [before]→[after]	Description
aconst_null	01	0000 0001		→ null	push a *null* reference onto the stack
aload	19	0001 1001	1: index	→ objectref	load a reference onto the stack from a local variable *#index*
aload_0	2a	0010 1010		→ objectref	load a reference onto the stack from local variable 0
aload_1	2b	0010 1011		→ objectref	load a reference onto the stack from local variable 1
aload_2	2c	0010 1100		→ objectref	load a reference onto the stack from local variable 2
aload_3	2d	0010 1101		→ objectref	load a reference onto the stack from local variable 3
areturn	b0	1011 0000		objectref → [empty]	return a reference from a method
astore	3a	0011 1010	1: index	objectref →	store a reference into a local variable *#index*
astore_0	4b	0100 1011		objectref →	store a reference into local variable 0
astore_1	4c	0100 1100		objectref →	store a reference into local variable 1
astore_2	4d	0100 1101		objectref →	store a reference into local variable 2
astore_3	4e	0100 1110		objectref →	store a reference into local variable 3
bipush	10	0001 0000	1: byte	→ value	push a *byte* onto the stack as an integer *value*
dload	18	0001 1000	1: index	→ value	load a double *value* from a local variable *#index*
dload_0	26	0010 0110		→ value	load a double from local variable 0
dload_1	27	0010 0111		→ value	load a double from local variable 1
dload_2	28	0010 1000		→ value	load a double from local variable 2

dload_3	29	0010 1001		→ value	load a double from local variable 3
dreturn	af	1010 1111		value → [empty]	return a double from a method
dstore	39	0011 1001	1: index	value →	store a double *value* into a local variable *#index*
dstore_0	47	0100 0111		value →	store a double into local variable 0
dstore_1	48	0100 1000		value →	store a double into local variable 1
dstore_2	49	0100 1001		value →	store a double into local variable 2
dstore_3	4a	0100 1010		value →	store a double into local variable 3
fdiv	6e	0110 1110		value1, value2 → result	divide two floats
fload	17	0001 0111	1: index	→ value	load a float *value* from a local variable *#index*
fload_0	22	0010 0010		→ value	load a float *value* from local variable 0
fload_1	23	0010 0011		→ value	load a float *value* from local variable 1
fload_2	24	0010 0100		→ value	load a float *value* from local variable 2
fload_3	25	0010 0101		→ value	load a float *value* from local variable 3
getfield	b4	1011 0100	2: indexbyte1, indexbyte2	objectref → value	get a field *value* of an object *objectref*, where the field is identified by field reference in the constant pool *index* (indexbyte1 << 8 | indexbyte2)
getstatic	b2	1011 0010	2: indexbyte1, indexbyte2	→ value	get a static field *value* of a class, where the field is identified by field reference in the constant pool *index* (indexbyte1 << 8 | indexbyte2)
goto	a7	1010 0111	2: branchbyte1, branchbyte2	[no change]	goes to another instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
goto_w	c8	1100 1000	4: branchbyte1, branchbyte2, branchbyte3, branchbyte4	[no change]	goes to another instruction at *branchoffset* (signed int constructed from unsigned bytes branchbyte1 << 24 | branchbyte2 << 16 | branchbyte3 << 8 | branchbyte4)
iconst_0	03	0000 0011		→ 0	load the int value 0 onto the stack

iconst_1	04	0000 0100		→ 1	load the int value 1 onto the stack
iconst_2	05	0000 0101		→ 2	load the int value 2 onto the stack
iconst_3	06	0000 0110		→ 3	load the int value 3 onto the stack
iconst_4	07	0000 0111		→ 4	load the int value 4 onto the stack
iconst_5	08	0000 1000		→ 5	load the int value 5 onto the stack
if_acmpeq	a5	1010 0101	2: branchbyte1, branchbyte2	value1, value2 →	if references are equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_acmpne	a6	1010 0110	2: branchbyte1, branchbyte2	value1, value2 →	if references are not equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_icmpeq	9f	1001 1111	2: branchbyte1, branchbyte2	value1, value2 →	if ints are equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_icmpge	a2	1010 0010	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is greater than or equal to *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_icmpgt	a3	1010 0011	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is greater than *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_icmple	a4	1010 0100	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is less than or equal to *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_icmplt	a1	1010 0001	2: branchbyte1, branchbyte2	value1, value2 →	if *value1* is less than *value2*, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
if_icmpne	a0	1010 0000	2: branchbyte1, branchbyte2	value1, value2 →	if ints are not equal, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
ifeq	99	1001 1001	2: branchbyte1, branchbyte2	value →	if *value* is 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)

ifge	9c	1001 1100	2: branchbyte1, branchbyte2	value →	if *value* is greater than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
ifgt	9d	1001 1101	2: branchbyte1, branchbyte2	value →	if *value* is greater than 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
ifle	9e	1001 1110	2: branchbyte1, branchbyte2	value →	if *value* is less than or equal to 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
iflt	9b	1001 1011	2: branchbyte1, branchbyte2	value →	if *value* is less than 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
ifne	9a	1001 1010	2: branchbyte1, branchbyte2	value →	if *value* is not 0, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
ifnonnull	c7	1100 0111	2: branchbyte1, branchbyte2	value →	if *value* is not null, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
ifnull	c6	1100 0110	2: branchbyte1, branchbyte2	value →	if *value* is null, branch to instruction at *branchoffset* (signed short constructed from unsigned bytes branchbyte1 << 8 | branchbyte2)
iinc	84	1000 0100	2: index, const	[No change]	increment local variable *#index* by signed byte *const*
iload	15	0001 0101	1: index	→ value	load an int *value* from a local variable *#index*
iload_0	1a	0001 1010		→ value	load an int *value* from local variable 0
iload_1	1b	0001 1011		→ value	load an int *value* from local variable 1
iload_2	1c	0001 1100		→ value	load an int *value* from local variable 2
iload_3	1d	0001 1101		→ value	load an int *value* from local variable 3
instanceof	c1	1100 0001	2: indexbyte1, indexbyte2	objectref → result	determines if an object *objectref* is of a given type, identified by class reference *index* in constant pool (indexbyte1 << 8 | indexbyte2)
invokedynamic	ba	1011 1010	4: indexbyte1, indexbyte2, 0, 0	[arg1, arg2, ...] → result	invokes a dynamic method and puts the result on the stack (might be void); the method is identified by method reference *index* in

					constant pool (indexbyte1 << 8 | indexbyte2)
invokeinterface	b9	1011 1001	4: indexbyte1, indexbyte2, count, 0	objectref, [arg1, arg2, ...] → result	invokes an interface method on object *objectref* and puts the result on the stack (might be void); the interface method is identified by method reference *index* in constant pool (indexbyte1 << 8 | indexbyte2)
invokespecial	b7	1011 0111	2: indexbyte1, indexbyte2	objectref, [arg1, arg2, ...] → result	invoke instance method on object *objectref* and puts the result on the stack (might be void); the method is identified by method reference *index* in constant pool (indexbyte1 << 8 | indexbyte2)
invokestatic	b8	1011 1000	2: indexbyte1, indexbyte2	[arg1, arg2, ...] → result	invoke a static method and puts the result on the stack (might be void); the method is identified by method reference *index* in constant pool (indexbyte1 << 8 | indexbyte2)
invokevirtual	b6	1011 0110	2: indexbyte1, indexbyte2	objectref, [arg1, arg2, ...] → result	invoke virtual method on object *objectref* and puts the result on the stack (might be void); the method is identified by method reference *index* in constant pool (indexbyte1 << 8 | indexbyte2)
new	bb	1011 1011	2: indexbyte1, indexbyte2	→ objectref	create new object of type identified by class reference in constant pool *index* (indexbyte1 << 8 | indexbyte2)
pop	57	0101 0111		value →	discard the top value on the stack
return	b1	1011 0001		→ [empty]	return void from method