T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

VERİTABANLARINDA TARİHSEL İŞLEM DENETİMİ VE BUNUN MODELLENMESİNE YÖNELİK YENİ BİR YAKLAŞIM

YÜKSEK LİSANS TEZİ

İbrahim DOKUZER

Enstitü Anabilim Dalı	**:**	**BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ**
Enstitü Bilim Dalı	**:**	**BİLİŞİM TEKNOLOJİLERİ**
Tez Danışmanı	**:**	**Yrd.Doç Dr. Hayrettin EVİRGEN**

Temmuz 2016

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

VERİTABANLARINDA TARİHSEL İŞLEM DENETİMİ VE BUNUN MODELLENMESİNE YÖNELİK YENİ BİR YAKLAŞIM

YÜKSEK LİSANS TEZİ

İbrahim DOKUZER

Enstitü Anabilim Dalı	:	**BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ**
Enstitü Bilim Dalı	:	**BİLİŞİM TEKNOLOJİLERİ**

Bu tez 29.07.2016 tarihinde aşağıdaki jüri tarafından oybirliği ile kabul edilmiştir.

Yrd.Doç.Dr. Hayrettin EVİRGEN	**Prof.Dr. Erman COŞKUN**	**Yrd.Doç.Dr. Muhammed Ali AYDIN**
Jüri Başkanı	**Üye**	**Üye**

BEYAN

Tez içindeki tüm verilerin akademik kurallar çerçevesinde tarafımdan elde edildiğini, görsel ve yazılı tüm bilgi ve sonuçların akademik ve etik kurallara uygun şekilde sunulduğunu, kullanılan verilerde herhangi bir tahrifat yapılmadığını, başkalarının eserlerinden yararlanılması durumunda bilimsel normlara uygun olarak atıfta bulunulduğunu, tezde yer alan verilerin bu üniversite veya başka bir üniversitede herhangi bir tez çalışmasında kullanılmadığını beyan ederim.

İbrahim DOKUZER

22.05.2016

TEŞEKKÜR

Yüksek lisans eğitimim boyunca değerli bilgi ve deneyimlerinden yararlandığım, her konuda bilgi ve desteğini almaktan çekinmediğim, araştırmanın planlanmasından yazılmasına kadar tüm aşamalarında yardımlarını esirgemeyen, teşvik eden, aynı titizlikte beni yönlendiren değerli danışman hocam Yrd.Doç.Dr.Hayrettin EVİRGEN'e teşekkürlerimi sunarım.

İÇİNDEKİLER

SİMGELER VE KISALTMALAR LİSTESİ

AOP	: Aspect Oriented Programming
API	: Application Programming Interface
CDC	: Change Data Capture
DNS	: Domain Name Server
DDD	: Domain Driven Design
ETL	: Extract-Transform-Load
EJB	: Enterprise Java Beans
HQL	: Hibernate Query Language
JDBC	: Java Database Connectivity
JPA	: Java Persistence Api
JDBC	: Java Database Connectivity
JSR	: Java Spesification Request
JDO	: Java Data Object
POJO	: Plain Old Java Object
PHP	: Personal Home Page
SQL	: Structured Query Language
URL	: Uniform Resource Locator
XML	: Extensible Markup Language

ŞEKİLLER LİSTESİ

TABLOLAR LİSTESİ

ÖZET

Anahtar kelimeler: Veri modelleme, veritabanı, veri denetimi

Veritabanlarında depolanan verilerdeki değişimin denetlenmesi ve kayıt altına alınması önem arz etmektedir. Veri değişimlerinin nesneye dayalı ilişkisel veri tabanlarındaki veri bütünlüğünü bozmadan versiyonlu olarak değişiminin gözlemlenebilmesi ayrı önem arz eden bir husustur.

Bu işlemlerin kayıt altına alınması sırasında sistem kaynaklarını fazla meşgul etmesi ve bilgilerin denetim amacıyla tutulurken gerekli olan yüksek depolama kapasitesi ihtiyacı bu yapıların başlıca problemleridir.

Bu mekanizmanın oluşturulması için genellikle belli platformlara bağımlı olan spesifik çözümler bulunmaktadır. Nesneye dayalı veritabanlarında genel olarak kullanılacak bir yapıya ihtiyaç olduğu gözlemlenmektedir.

Bu çalışmada nesne tabanlı veritabanlarındaki veri değişimlerinde denetim mekanizmalarına bu bakış açısıyla değinilmiş, günümüzde bu işi yapan alternatif sistemler hakkında bilgi verilmiş, sonrasında veritabanındaki veri denetimleri için önerdiğimiz uygulama katmanında, veri denetimini yapan ve platform bağımsız olarak kolaylıkla java mimarisi kullanan tüm sistemlerde veritabanı yapısı bağımsız çalışabilen ve verilerin belli bir zamanda ki durumunu gözlemlemeyi sağlayan, ileriye dönükte veri değişimlerinin otomatik yapılabileceği yeni yaklaşım dan bahsedilmiştir.

HISTORICAL PROCESS CONTROL IN THE DATABASE AND A NEW APPROACH FOR MODELING IT

SUMMARY

Keywords: Data modeling, database, data control

Monitoring of changes at databases and keeping records are important. Withdrawal without compromising data integrity in a relational database for data exchange is also an important consideration During the recording of these transactions to the more busy system resources, return the data associated with a particular version of history, maintain data integrity, high storage capacity requirements while keeping the necessary information in order to control are the main problems of these structures. There is often dependent on the particular platform specific solutions for the creation of this mechanism. All infrastructure is observed that the need for a structure to be used in general.

In this document, mentioned at in this perspective change control mechanisms of object-oriented databases, provided information about alternative systems makes this job today. Then which proposed for data audit at databeses and it works at application frame and its in undepended from platforms, after has mentioned from new approach which can data in the database at the application layer, data controls to the recommended control easily as java architecture and platform independent uses in all systems can operate independently of the database structure and a particular time, integrity of data are returned intact, which enables automatic data exchange can be done with the time forward .

BÖLÜM 1. GİRİŞ

Günümüzde veritabanlarında yapılan veri güncelleme, silme, ekleme işlemlerinin her birinde işlemin kimin tarafından hangi tarih ve saatte, hangi fiziksel ortamdan, yapıldığı bilgilerinin tutulması, şirket güvenlik politikaları, veri gizliliği ve veri değişimlerinin geriye doğru takibi açısından önemlidir. Büyük boyutlu veri tabanlarında birim zamanda yapılan işlem sayısı çok fazla olduğu için bu bilgilerin efektif bir şekilde sistem kaynaklarını en az şekilde tüketerek, hızlı doğru ve istendiğinde kolay ulaşılabilir şekilde bir mekanizma tarafından tutulması gerekmektedir. Veritabanı yönetim sistemlerinde veri denetiminin yeterli ve etkili bir biçimde sağlanması konusundaki yaklaşımlar ilgili referansta detaylı değinilmiştir [1]. Halihazırda kullanılan yapılar iki ye ayrılır bunlar veritabanı tarafındaki çözümler ve uygulma katmanında bulunan çözümlerdir. Veritabanı tarafında database tetikleyicileri [2], veri değişim yakalayıcıları (Change Data Capture-CDC)[3,4,5], işlem kaydı okuyucu sistemler [6,7], servis komisyoncusu [8] dur. Uygulama katmanında ise durum tabanlı programlama (Aspect Oriented Programing-AOP)[9] ve hibernate olayları [10,11,12] dır. Bu yapılar hakkında çalışmada detaylı bilgi verilmiştir.

Günümüzde kullanılan yapılar fonksiyonalite, sisteme getirdiği yük, bakım ve modilarite, yeterlilik kriterlerine göre karşılaştırılmıştır [13]. Nesne tabanlı veritabanlarında kullanılan, EnversHibernate, Audit(OpenJPA), HistoryPolicy (EclipceLink), EntityVersionHistory (ObjectDB), EntityAuditBundle (Doctrine) gibi arşivleme araçları hakkındada inceleme yapılmış ve detaylı biçimde anlatılmıştır.

Çalışmanın son kısmındada çalışmaya konu olan yeni yaklaşımda kullanılan JavaDatabaseConnectivity (JDBC)[19], Enterprise Java Beans (EJB) [20], Java

Persistence API (JPA) [21] gibi teknolojiler detaylı olarak incelenmiş, mimari tasarım ve çalışmanın diğer çözümlere göre öne çıkan özellikleri olan java çerçevesinde (framework) ünde yazılmış olan mimarilerde genel olarak kullanılabilir olması, yapının kurulmasındaki kolaylık, sistemin sürdürülebilmesinde ve değiştirilmesindeki esneklik ve performans olarak öne çıkan avantajlarından bahsedilmiştir.

BÖLÜM 2. VERİTABANLARINDA İŞLEM DENETİMİ ÇÖZÜMLERİNE GENEL BAKIŞ VE ÇÖZÜMLERİN KARŞILAŞTIRILMASI

Veri denetimi için kullanılan çözümümlerde öne çıkan yaklaşımlar ikiye ayrılmaktadır. Veri tabanı spesifik çözümler ve uygulama katmanında verilerin denetimini sağlayan yapılar.

2.1. Veritabanı Tarafındaki Çözümler

Veritabanı katmanında yapılan işlem bilgilerinin tutulması için aşağıdaki yapılar mevcuttur.

2.1.1. Tetikliyiciler (Triggers)

Tetikleyiciler veri tabanı yönetim sistemlerinde uygulamalardan bağımsız olarak çalışan otomatik olarak güncelleme, kayıt, silme işlemlerinde çalışan saklı yordamlardır. Tetikliyiciler uygulama arayüzlerinden ve kullanıcının isteğinden bağımsız olarak çalışırlar. Tetikleme, işlemin öncesinde veya sonrasında yapılabilir.

Değişen verinin eski hali ve sonrasında da yeni hali alınarak saklanır. Bir tetikleyici diğer bir tetikleyicinin çalışmasını sağlayabilir. Tetikleyiciler her tablo için ayrı ayrı tanımlanmalıdır. Triggerlerin tanımlanması veritabanı dizaynı sırasında karar verilecek işlemdir. İş gereksinimlerinin sağlanması ve veritabanı tutarlılığının sağlanması açısından önemli bir araçtır. İş süreçlerindeki gerekli gereksinimlerin karşılanması uygulama bağımsız sağlanabilir [2].

2.1.2. Change data capture (CDC)

Spesifik veritabanı için üretilmiş bir üründür, güncelleme, kaydetme, silme işlemlerinde veritabanı özellikli olarak gelen veriyi kendi tablolarına kayıt eder. Asenkron olarak çalışır ve belirlenen periyodlarda veritabanının işlem log tablolarına bakarak yapılan değişiklikleri kendi tablolarına kayıt eder. Sadece veritabanı yönetim sisteminde kullanıcıların yaratmış olduğu tabloların kayıtlarını tutar. CDC nin aktiflenmesi için veritabanınında ve istenilen tablolarda CDC nin aktiflenmesi gerekmektedir. Bu çözümün sorunlu yanı veritabanının işlem tablolarını boşaltabilmesi için CDC nin kendi tablosuna gerekli işlem kaydını yazmış olması gerekliliğidir. Eğer yük altında CDC nin yazma hızı işlemlerden yavaş ise işlem tabloları gerektiği hızda silinmediğinden dolacaktır ve sistemin kitlenmesine ya da performansının olumsuz yönde etkilenmesine neden olacaktır [4,5].

2.1.3. İşlem kaydı okuma

Change Data Capture (CDC) den başka veritabanı mimarisi bağımsız işlem kayıtlarını belirli aralıklar ile okuyup kendi tablolarına yazan üçüncü parti yazılımlarda mevcuttur. Veyahut çevrimiçi olarak işlem kayıtlarını okuyup anlamlandıran araçlar mevcuttur. Bu araçlar işlem kayıtlarını okuyup belli ölçütlere göre sorgulama yapabilirler. Bu yapıların sorgulama kapasiteleri basit SQL sorguları ile sınırlıdır [6].

2.1.4. Servis komisyoncusu (Service Broker)

Spesifik bir veritabanı için geliştirilmiş bir çözümdür. Sistemi anlamak için öncelikle eşzaman ve asenkron çalışma mantığı anlaşılmalıdır. Senkron işlemlerde uygulama tarafından istek geldiğinde istek sıralı ve anında işlenir, işlem başlangıcından sonuna kadar kesintisiz olarak çalışır. Uygulama eşzaman işleme başladığında yeni işlem başlamadan önce eski işlemin tamamen bitmesi beklenir. Asenkron çalışma mantığında ise uygulama tarafından yapılan istekler sıralı ve derhal işlenmez. Bu istekler en yakın zamanda işlenmek üzere kuyrukta tutulur, uygulama yaptığı işlemin

sonucunu beklemeden diğer bir istek te bulunabilir, servis broker yapısında yapılan değişiklikler asenkron olarak mesaj temelli uzak sisteme aktarılır ve kayıt işlemi burada gerçekleşir. Servis broker yapısıyla entegre çalışacak uygulamaları geliştirirken asenkron işlemlerin gerçekleşmesi için servis broker ın çalışma mantığının ve uygulma ile etkileşiminin anlaşılması gerekir.

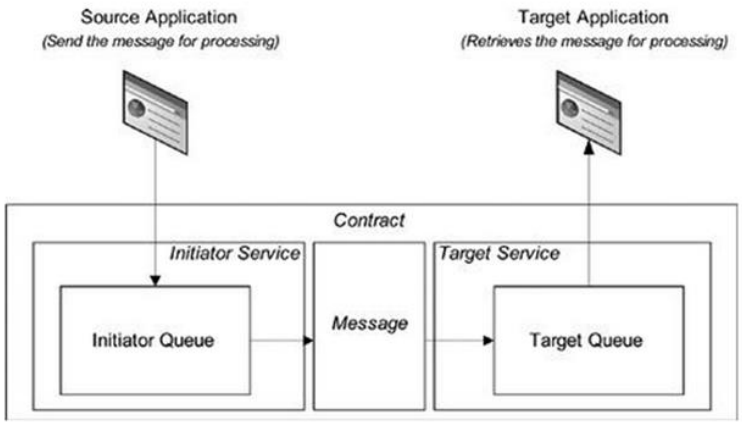
Bilgisayar sistemlerinde mesajlaşma yapısı basittir fakat veritabanı yöneticileri veritabanı yazılımcıları işlemler ve veri maniplasyonları ile uğraştıkları için mesaj kavramının ne olduğu konusunda yanılgılara sahiptir. Mesajlar işlemin gerçekleşmesi için sisteme yollanan emirlerdir. Örnek vermek gerekirse web tarayıcınsında bir sayfa açmak istendiğinde mesaj DNS sunucusuna URL ile birlikte gider. Burada mesajın ana gövdesi URL dir. Tarayıcı ise burada işlemi başlatıcı uygulamadır. DNS Sunucusu ise mesajı alan ve işleyen hedef uygulamadır.

Veritabanı dünyasından örnek olarak SQL Sunucusundan id değeri dönen object_id fonsiyonu verilebilir. Bu fonksiyon da yollanan objenin adı mesajın gövdesini oluşturur. Burada ifadenin yollandığı veritabanı yönetim sistemi arayüzü başlatıcı ve fonksiyon ise hedef olarak adlandırılır. Her mesaj ayrı ayrı aksiyon olarak ne yapılacağını bilen bir uygulamaya gönderilir uygulamada bu aksiyonlar da ne yapılacağı ile ilgili geliştirme yapılır. Mesaj kavramı ve bu mesajların işlenme mantığı anlaşıldıktan sonra Servis Broker ın uygulama ve bileşenleri ile nasıl etkileşim halinde olduğu konusuna değinilebilir. Mesajın yollanabilmesi için iki uygulamanın iletişim kurması gerekmektedir. Servis Broker larda bu uygulamalar end-point olarak adlandırılır ve bunlar fiziksel veritabanlarıdır. İletişime başlayan end point başlatıcı olarak adlandırılır. Konuşmaya dahil olan end point ise hedef (target) olarak adlandırılır. End pointler arasında iletişim için bağlantı kurulduğunda end pointler aralarında veri alıp gönderirler. End pointler farklı veritabanlarında olabileceği gibi aynı veri tabanının farklı instanceları da olabilirler.

Veri alışverişini gerçekleştiren iki end point arasındaki iletişime konuşma (conversation) denir. Service Broker tarafından gönderilen tüm mesajlar konuşmanın (conversation) ın parçasıdır.

Konuşma nın parçası olan uygulamalar arasında alışveriş halinde olan mesajları tutmak için bir yapıya ihtiyaç duymaktır bu yapı kuyruk(queue) olarak adlandırılır. Kuyruk gönderilen mesajları ve işlemleri tutan ve ilk giren işin ilk çıktığı (FIFO-First-In-First-Out) yapıya sahip olan tablodan ibarettir. Uygulama mesaj yolladığında bu tablonun en altına işlenir bu işlem kuyruğa atılma (enqueuer) olarak adlandırılır.

Uygulama bu tablodaki mesajları ilk sıradan başlayıp okuyup işleme sorumluluğuna sahiptir. Bu işleme kuyruktan çıkarma (dequeued) denir. Bu tablolar olası veri kayıplarını önlemek için uçucu bellek'te tutulmaması yerine gizli tablolarda tutulur. Bu sayede yüksek seviyede sistemin devamlılığı sağlanmış olur. Eğer işlem başarılı olarak gerçekleşmezse Servis Komisyoncusu(service broker) bu yapılan işlemi geri alır ve uzak sisteme değişikliği yollamaz. Bir çok veri tabanı aynı denetim mekanizmasını kullanabir merkezi bir denetim instance ı olusturulabilir. Şekil 2.1.'de Servis Broker yapısının mimarisi gösterilmiştir [8].



Şekil 2. Servis komisyoncusu (Servis Broker) mimarisi[8].

2.2. Uygulama Katmanına Yönelik Çözümler

2.2.1. Durum tabanlı programlama(Aspect Oriented Programing-AOP)

AOP(Aspect Oriented Programming) kesişen durumları yeni birimlerle modularize eden bir metolojidir. Her durum kesişen fonksiyonalitelere odaklanır. Ana sınıflar bu sayede fonksiyonel kesişimler den uzak hale geitrilir. Durum gözlemleyicileri (aspect

viewer) ana sınıflar ile kesişim durumlarını ayrıştırmaya yararlar bu işlem izleme (weaving) olarak adlandırılır. Bu sayede durum tabanlı programlama (Aspect Oriented Programming-AOP) uygulamaların kolay dizayn edilmesini, uygulanabilmesini ve yönetilmesine yardımcı olur.

AOP kullanılmadan nesneye dayalı programlama ile ana işlevin yer aldığı sınıflarda ana işlev haricinde yapılması gereken ve ana işlevle kesişen durumlarda implemantasyonun sağlanması konusunda dizayn aşamasında temel işlevlerin ayrıştırılması olasıdır fakat implementasyon fazında bu durumlar birbirleri ile kesişirler. Temel durumlar ile kesişen durumları aynı sınıfta yönetmek tekil sorumluluk prensibinin (Single Responsibility Principle-SRP) uygulanamamasına neden olur. Kesişen işlevselliklerde kod un değişimi gereksinimi ortaya çıktığında bu kesişim in yer aldığı tüm sınıflarda bu değişikliğin ayrı ayrı yapılması gerekir. Bu değişiklite açık/kapalı (Open/Close Prinsible) prensibinin uygulanamamasına yol açar. Prensibe göre yeni gelistirmeye açıklık var olan kodun değişimine kapalılık esastır. Geleneksel uygulamalarda ana işlevsellik ile kesişen işlevsellikler her modulde bir birinin içine girmiş durumdadır. Ayrıca bu kesişen işlevsellik durumları modullere yayılmış durumdadır. Moduller arasında tekrarlayan kesişen işlevsellik durumunda geleneksel implementasyonda kodun karışıklığı ve yayılması problemi genel bir problemdir [9].

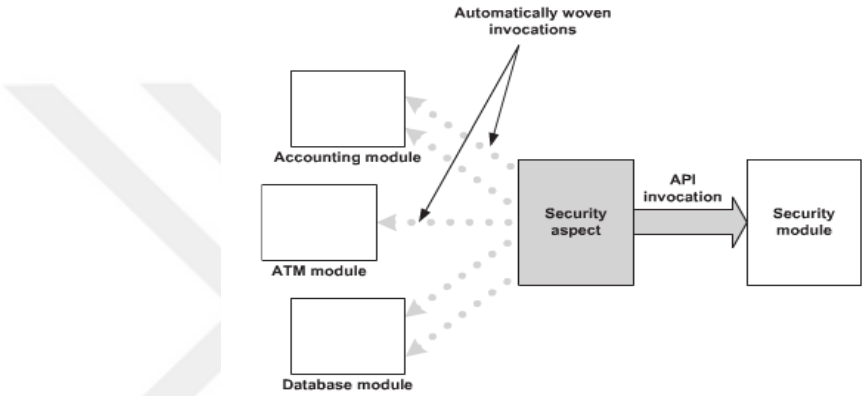
2.2.1.1. AOP ile modülarite

Nesneye dayalı programlamada ana işlevsellik arayüzler (interfaces) ile bağlantılıdır. Fakat kesişen işlevselliklerde bunun sağlamanın kolay yolu yoktur. Bu nedenle bu sunucu tarafı parça ve istemci tarafı parça olarak iki bölüme ayrılmıştır. Nesneye dayalı programlama sunucu tarafını arayüz (interfaces) ve sınıflar ile başarılı bir şekilde modülerize eder. Fakat kesişen işlevsellik durumlarında istemci tarafında ise code tüm istemcilere yayılır.

Nesneye dayalı programlama da genel kesişim konseptini ele alırasak arayüz üzerinden fonksiyonaliteyi sağlayan güvenlik modülünü düşünürsek istemci fonksiyonaliteyi implementasyonun yapıldığı arayüz (interface) e ulaşarak sağlar.

İnterface üzerindeki herhangi bir değişiklik istemciyi etkilemez. Fakat yinede interface'i çağırmak için her istemci de gömülü kod bulunmaktadır. Güvenlik modülünü kullanan her istemci ayrı ayrı bu çağırımı içlerinde yapmak zorundadır buda kod karmaşıklığını arttırır.

Durum tabanlı programlama yı aynı durum için ele alırsak istemcide bir çağırıma gerek kalmadan moduller güvenlik modülünü kendileri çağırırlar. Şekil 2.2.'de bu senaryoya özgü durum tabanlı programlama nın çalışma mantığı yer almaktadır [9].



Şekil 2.2. Güvenlik modulünün durum tabanlı programlama ile implementasyonu[9]

2.2.1.2. Durum gözlemleyici kuralları (Viewing Rules)

Gözlemleyici kuralları implemente edilen foksiyonalitenin son olarak nasıl tasarlanacağını belirleyen yapılardır. Sistem deki hangi genel operasyonların loglanması belirlemek için bir iki satır kod yazılması yeterlidir.Denetleme durumları için gözlemleme özellikleri aşağıdaki gibidir.

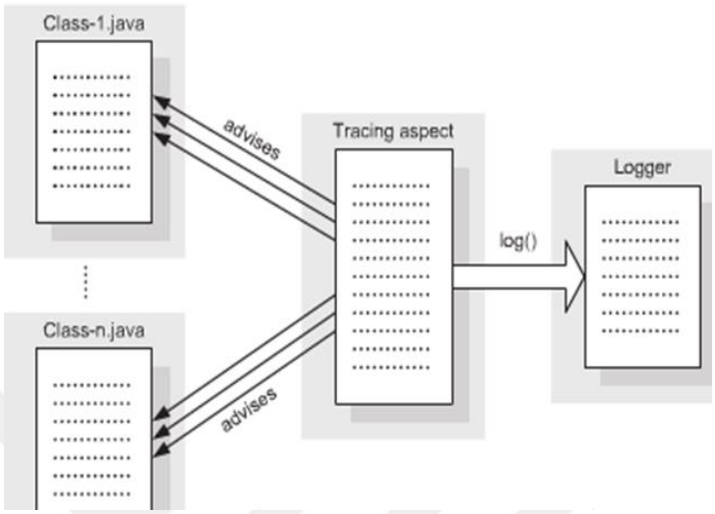
Kural1: Denetim (Logger) Nesnesinin yaratılması.

Kural2: Her genel operasyonun başlangıcında log yaz.

Kural3: Her genel operasyonun bitişinde log yaz.

Bu işlem her operasyon içinde denetim logunun yazılmasının tanımlamasından çok daha basittir. Böylece denetim mekanizması sınıflarda uzaklaştırılmış ve merkezileştirilmiş olmaktadır. Kurallar spesifik olarak bir operasyon veya iş mantığı için tanımlanabilir. Kuralların tanımlanmasında genel olarak java programlama dili aspectJ adıyla kullanılır yada XML tabanlı diller ile de tanımlama yapılabilir. Şekil

2.3.'de denetleme mekanizmasının aspectJ tabanlı olarak ilgili sınıflarda nasıl uygulandığı gösterilmiştir [9].



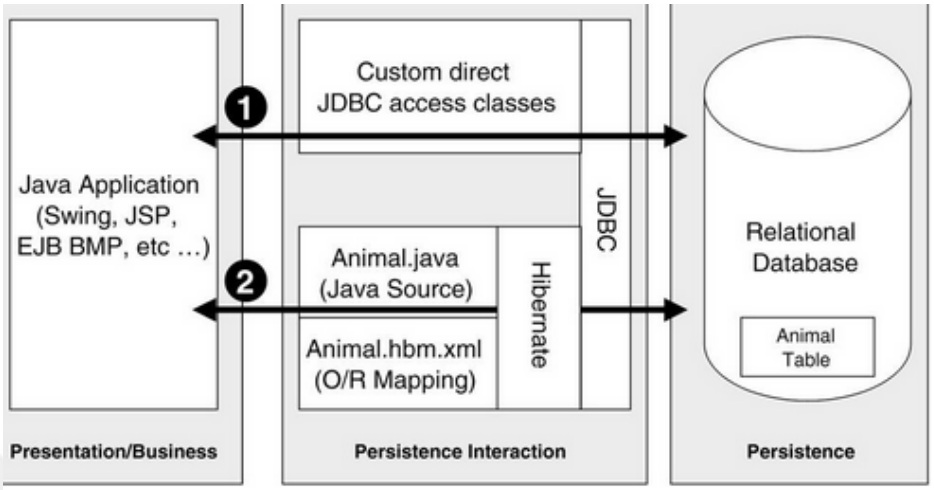
Şekil 2.3. Şematik aspectJ tabanlı denetim mekanizması[9].

2.2.2. Hibernate olayları (Events)

Hibernate açık kaynak kodlu nesne-ilişkili bir çerçevedir (Framework). Hibernate in amacı basit java sınıflarını XML (Extendible Markup Language) tanımlayıcıları ile birleştirerek ilişkisel veri tabanlarına nesne tabanlı yaklaşımı sağlamaktır. Bu sayede ilişkisel vertabanındaki tablolar birer nesneye çevrilir. Java uygulamalarının çoğunluğu ilişkisel veri tabanlarına java veritabanı bağlantısı (Java Database Connectivity -JDBC) ile bağlanır. Hibernate kullanarak uygulama geliştiriciler JDBC integrasyon kodu yazmadan uygulamamnın iş mantığı ile ilgili geliştirmelere yoğunlaşabilir. Ayrıca hibernate JDBC bağlantısınını kullanıldığı projelerde bu yapıdan bağımsız olarak farklı sınıflar ile ilişkisel veritabanı hybrid olarak sağlıyabilir Şekil 2.4.'de bu yapı gösterilmiştir [10].

Hibernate olay (event) ve olay dinleyicilerine (event listener) a sahiptir. İlgili java methodunda işlem gerçekleştiğinde olay dinleyicileri olayın öncesinde yada sonrasında yapılan işlemi yakalar ve denetim tablosuna otomatik kayıt atar.

Hibernate olayları da uygulama katmanında her hangi bir platforma mesaj olarak bu değişiklikleri yollayabilir [12].



Şekil 2.4. Hibernate yapısının JDBC ile hybrid olarak kullanımı[10].

2.2.2.1. Temel hibernate arayüzleri (Frameworks)

Hibernate in yapısında 5 ana arayüz(interface) mevcuttur. Bunlar Oturum(session), oturum fabrikası(sessionfactory), işlem(transaction), kuyruk(queue), yapılandırma (configuration) arayüzleridir. Bu arayüzler her gelistirme aktivitesinde kullanılır. Birbirlerinden ayrıştırılamazlar. Oturum arayüzü kalıcı nesnelerin (persistent object) yaratma, güncelleme, okuma, silme işlemlerinden sorumludur. Hibernate oturumu JSP uygulamalarının Http oturumundan farklıdır. Oturum Fabrikası (Session Factory) hibernate'i başlatmaktan sorumludur. Kendisi veri saklayıcısının elçisi gibi davranır ayrıca oturum objelerinin oluşturulmasından sorumludur. Birden çok veritabanında işlem yapmak istendiğinde oturum fabrikasında gerekli ayarlama yapılabilir.

Yapılandırma arayüzü hibernate in yapılandırılması, başlatılması ve oturum fabrikasının (Session factory) oluşturulmasından sorumludur. Hibernate in başlatılması işleminde yapılandırma sınıfı instance'ı ilk önce adreslenen dökümanın pozisyonunu yapılandırma dosyasından okur. İşlem arayüzü (transaction interface) iş

mantığı ile ilgili geliştiricinin yazdığı kod kısmından sorumludur. Opsiyoneldir ve geliştirici isterse işlem yönetiminde kendi iş mantığını burada oluşturabilir.

Kuyruk arayüzü (queue interface) veritabanı sorgularının implemente edilmesi ile sorumludur. Kuruk arayüzü HQL (Hibernate Query Language) veya SQL (Structured Query Language) ifadelerini kullanabilir. Hibernate iki seviye ön belleği destekler. İlk seviye önbellek oturum seviyesi önbellektir ve servislerin kapsamı dahilindedir. Herhangi bir müdehaleye gerek kalmadan hibernate in kendisi tarafından yönetilir. İkinci seviye önbellek ise oturum fabrikası önbelleğidir. Bu önbellek işlem aralığı veya küme (cluster) kapsamına (scope) aittir. Bu ön bellek yapılandırılabilir ve değiştirilebilir yapıdadır, dinamik olarak yüklenip silinebilir.

Hibernate de ikini seviye önbelleğin yapılandırmasına göre sorgu sonuçları için sorgu önbelleğinede sahiptir. Varlık sınıflarının özelliklerinin ifade edildiği ve veritabanı ve veritabanında ilşkili tabloların alanlarının tutulduğu adresleme dosyaları, hibernate arayüzünün yapılandırma dosyalarının en temel ve en önemlisidir [11].

2.3. Çözümlerin Karşılaştırılmasında Kullanılan Kriterler

İlgili denetleme çözümlerinde aşağıdaki ölçütler önem kazanmaktadır.

1. Fonksiyonelite:
Tüm verilerin doğru bir biçimde kayıt altına alınabilmesi.

2. Sisteme getirdiği yük:
Denetim ve kayıt sırasında sistemi durdurup durdurmaması.

3. Yeterlilik:
Sistemin hata alması durumunda hata toleransının olması ve bu esnadaki işlemlerin kayıba uğramıyor olması.

4. Bakımının kolay olması ve moduler yapıda olması :

Veritabanında model değişikligi yada uygulama katmanında kod değişikligi olmadan çoklu platformlarda çalışabilir olması ve istendiginde diğer sistemlere de yayılabilirliğinin fazla maliyetli olmaması.

2.4. Kriterlere Göre Çözümlerin Analiz Edilmesi

2.4.1. Fonksiyonalite

Tetikliyiciler bir işlem gerçekleşirken hata alırsa o işlem gerçekleşmez ve bu nedenle denetim altında olmayan hiçbir işlem gerçekleşmez.
Değişen veri nin elegeçirilmesi (CDC_Change Data Capture) yönteminde ise tüm gerekli denetim datasını veritabanının işlem loglarından alır , fakat işlemin kimin tarafından ve hangi tarih ve saatte gerçekleştirildiği bilgisini tutmaz , tablodaki sutunların aynısını kendi denetim tablosunda tutar. Eğer işlemin kimin tarafından gerçekleştiğini bulmak istiyorsak data model de değişikliğe gidip tüm tablolara sutun eklenmesi gereksinimi mevcuttur.

Üçüncü parti yazılımlar ,islem loglarını okuyarak bize gerekli datayı sağlarlar, bu çözümde işlemin kimin tarafından ve hangi saatte yaptıgı bilgisini elde edebiliriz

Servis Komisyoncusu (Service Broker) lar ise yapılan değişikleri triggerlardan toplayarak merkezi bir sisteme mesaj olarak yollarlar. Aspect Orianted Programming yada Hibernate de olduğu gibi tüm işlem denetim kayıtlarının kayıt altına alındığını garanti altına alır.

Görünüş odaklı programlama da (Aspect oriented Programing_AOP) denetim uygulama katmanında yapılır, kayıdı değiştirecek methoda girmeden önce verinin ilk değeri ve değişimden sonraki değeri okunur ve kayıt altına alınır. Değişikligi yapan birim ve tarih bilgileri uygulama katmanında sorunsuz olarak kayıt altına alınabilir. Uygulama katmanı haricindeki veritabanı değişiklikleri denetim altına alınmaz. Denetim datalarının başka sistemde tutulabilmesi için mesaj yapısı kurulması yada var olan mesajlaşma servisleri ile entegrasyon yapılması gerekmektedir.

Hibernate olayları uygulama katmanında çalışır, işlem gerçekleşirken ilgili metod da işlem yapıldığında eski ve yeni kayıt için denetim sistemine kayıt atılır. Aspect oriented programing da olduğu gibi direk veritabanı katmanından yapılan işlemler de denetleme mekanizması devre dışı olur. Denetim kayıtlarının uzak sisteme yollanması için mesajlasma sistemleri ile entegrasyonu gereklidir.

2.4.2. Veritabanı sistemlerine getirdiği yük

Tetikleyiciler denetim kayıtlarının kaydı sırasında ilgili tabloyu kilitler. Bu nedenle sistemin performansını etkiler. Değişen veri nin elegeçirilmesi (CDC_Change Data Capture) yönteminde denetleme kayıtlarını işler ken tabloları kilitlemez. Asenkron olarak işlem kayıtlarını belli periyodlar la okuyarak denetim kayıtlarını işler performansının sistemin yüküne gore ayarlar eğer sistem yükü fazla ise işlem kayıtlarını çekme sıklıgını düşürür. Bu yapının dez avantajı ise kayıt altına alınan işlemler de sadece işlem gören sutunu değil datanın tamamını kaydeder, bu nedenle de aşırı işlem gören yapılarda kayıtları saklamak için cok fazla disk alanına ihtiyaç duyar. Değişen verinin elegeçirilmesi (CDC_Change Data Capture) yöntemindeki gibi tabloyu işlem sırasında kilitlemez, kendi denetim tablolarını ayarlanabilir CDC deki gibi disk alanı harcama gibi bir problem ortaya çıkmaz. Servis komisyoncuları (Service Brokers) tetikleyici ile beraber kullanıldığında tabloyu mesaj uç sisteme gönderilinceye kadar kilitler. Aynı durum uygulama katmanındaki denetleme yapılarındada geçerlidir. Uygulama katmanında kaydetmeden önce mesajın uç sisteme gönderildigine dair cevap alana kadar işlem yapılmaz. Görünüş odaklı programlamada (Aspect oriented Programing_AOP) sisteme getirdiği yük bakımından diğer yaklaşımlara çok az da olsa olumsuz yönde bir fazlalığı mevcuttur. Kayıt, silme ve güncelleme ile ilgili metodlarda java katmanında aspectj de [5] build time da ekstra işlemler sırasında beklemeye yol açar. Güncelleme işleminde verinin değişimden önceki halini veritabanından okuduğu için tabloyu kitler. Güncelleme işlemi bittikten sonra denetim verisini uç sisteme yollamak için zaman harcar. Görünüş odaklı programlama da(Aspect oriented Programing_AOP) in aksine hibernate eski ve yeni verinin olduğu diziye aynı anda ulaşır.

2.4.3. Çözümlerin yeterliliği

Tetikleyiciler kayıt esnasında sorun olduğunda denetim verisi kayıt etmez böylece sistemin tutarlılığında bir soruna neden olmaz. Değişen verinin elegeçirilmesi (CDC_Change Data Capture) yöntemi Kayıt esnasında herhangi bir sorun da islem kayıtlarını asenkron olarak okuduğu için sorunlu işlemlerde yanlış denetim kayıdı atmaz. İşlem kayıtlarının okunması yönteminde İşlem kayıtlarını okuduğu için sistemsel bir sorunda yanlış denetim kayıdı oluşmaz. Servis komisyoncuları (Service Brokers) yapılan işlemlerin loglarını uç sisteme iletmekle sorumludur, eğer iletimde bir hata oluşursa bu kayıt kuyruğa atılır ve işlenmesi için beklenir bu arada yapılan işlem geri alınırsa kuyruktan da silinir. Görünüş odaklı programlama da(Aspect oriented Programing_AOP) yapısında denetim kayıtları aynı veritabanında tutuluyorsa işlem sırasında hata alınırsa işlem kaydı gercekleşmeden denetim kayıdı atılmaz, fakat işlem uç sisteme kaydediliyosa mesaj sistemine bilgi gider mesaj sisteminin yetenegine gore kuyruktaki iş silinebilir yada silinmez. Görünüş odaklı programlama da(Aspect oriented Programing_AOP) ile çalışma mantığı ile aynıdır.

2.4.4. Bakımının kolaylıgı ve moduler yapıda olması

Tetikleyicilerin görevini yapabilmesi için veri değişikli olan tablolarda tek tek tanım yapılmasına ihtiyaç vardır. Triggerlar daki sorunlarda toplu müdehale yapılamaz ayrı ayrı her triger için işlem yapılmalıdır. Değişen veri nin elegeçirilmesi (CDC_Change Data Capture) yönteminde yeni bir kolon eklendiğinde düzeltme yapılması gerekmektedir. İşlem Kaydı Okuma yöntemi işlem kayıtlarını okuduğu için bir değişiklikte herhangi bir şekilde etkilenmez. Servis komisyoncuları (Service Brokers) uygulamada değişikliklerden etkilenmez veritabanı işlem kayıtlarından aldığı bilgiyi uç denetleme veritabanlarına aktarır.

Görünüş odaklı programlama da(Aspect oriented Programing_AOP) değişikliklerden etkilenmesi sadece uygulama katmanında olur. Bu değişiklik uygulama katmanında gerekli basit değişikliklerle kolay bir biçimde yapılır. Hibernate Olayları (Events) değişiklikleri dinleyiciler aracılğıyla (event listener) okur. Bir değişiklik sonrasında uygulama katmanın da gerekli notasyonlar eklenmelidir [10].

BÖLÜM 3. GÜNÜMÜZDE KULLANILAN ARŞİV ARAÇLARI

Günümüzde nesneye dayalı veri tabanı mimarilerinde kullanılan arşiv çözümleri JPA mimarisi üzerine kurulmuşlardır. Aşağıda bu araçları detayları yer almaktadır. Tablo 3.1'de yeni yaklaşım ile günümüzde kullanılan arşiv araçlarının yeteneklerinin karşılaştırılması görülebilir.

3.1. Hibernate-Envers

Hibernate Java programlama dili için bir nesne ilişkisel eşleştirme (ORM Object Relational Mapping) kütüphanesidir. Nesne yönelimli etki alanı (Object-oriented domain) modeli ile ilişkisel veritabanı arasında bir eşleştirme altyapısı sunar. Envers projesi de Hibernate kullanılan uygulamalarda tablo arşivleme işlevlerini üstlenir, Hibernate in temel uygulamalarından biridir.

Envers projesi JPA Standartlarını kullanarak persistent sınıflarının kolay izlenmesini /sürümlenmesini amaçlamıştır. Envers'i kullanmak için tek yapılması gereken izlenmek istenen sınıfın veya sınıf alanının üzerine @Audited şeklinde java annotation (açıklama) koymaktır. Her izlenen sınıf için yapılan değişikliklerin tarihçesini tutan bir tablo oluşturulacaktır. Bu tarihçe tablosundan daha sonra çok fazla maliyet gerektirmeden veri çekilebilir ve sorgulama yapılabilir.

Tarihçe kütüphanesi revizyonlardan oluşur. Temelde her bir işlem bir revizyon ilerletir. Revizyonlar uygulamada genel bir revizyon numarasına sahiptir, ilgili revizyondaki çeşitli nesneler sorgulanabilir, ilgili revizyondaki veritabanı görünümü alınabilir. Herhangi bir revizyon numarasından hangi tarihte yapıldığı bilgisi bulunabilir, ya da tersi herhangi bir tarihte hangi revizyon yapıldığı anlaşılabilir.

Envers sadece hibernate ile çalışır ve hibernate annotation'ları veya entity manager (nesne yöneticisi) gerektirir. İzlemenin düzgün çalışabilmesi için, enitiy lerin değişmez tekil tanımlayıcıları olması gerekmektedir (primary key). Hibernate'in çalıştığı heryerde envers kullanılabilir, örneğin Jboss uygulama sunucusu içinde, Jboss Seam altyapısında veya Spring altyapısında kullanılabilir.

Özelliklerinden önemli olanlar;

1.JPA standartlarına uyan tüm eşleştirmelerde izleme yapılabilir:

JPA'dan türemiş olan hibernate eşleştirmelerinde izleme yapılabilir, örneğin özel tipler ve basit tiplerin (String, Integer vb.) sıralama/dizinlerinde (collection/map listelemeleri).

2.Revizyon tablosu olan tüm revizyonlar loglanabilir:

Tarihçe verileri sorgulanabilir. Bazı tekil tanımlayıcısı içermeyen listeler (java List tipi) desteklenmeyecektir, örneğin bir liste içinde birbirinin aynısı olan nesneler olabilir ve envers hangisinin izlenmesi gerektiğine karar veremeyecektir. Bu şekilde olan listelerde envers hata mesajı verecektir. Bu durumu aşmanın iki yolunu önermektedir envers; @IndexColumn annotation'ı kullanarak listelerin indexlenmesi veya @CollectionId annotation'ı kullanarak objeler için bir tekil id üretilmesi.

3. Components collections:

Birleşen dizinleri henüz desteklenmemektedir fakat yapılması düşünülmektedir.

4.@OneToMany+@JoinColumn kullanılması durumu;
Bu iki annotation'ın kullanıldığı collection (dizin) larda, Hibernate join tablo oluşturmaz. Fakat Envers bu durumda bunu yapmak zorundadır, böylece değişikliğe uğrayan nesnenin revizyonları okunurken yanlış sonuçlar alınmaz. Bu ek join tablosunu adreslemek adına @AuditJoinTable adında özel bir annotation kullanılır ki JPA standartlarında @JoinTable'a çok benzerdir.

@OneToMany+@JoinColumn gibi özel bir duruma benzer tekil ilişki eşleştirmesi gibi @ManyToOne+@JoinColumn(insertable=false, updatable=false) çoğul ilişki eşleştirmesi de vardır. Bu ilişkiler ikiyönlü olaylardır, ama sahip taraf dizindir. Buna

benzer ilişkilerde Envers'te izlemeyi düzgün yapabilmek adına @AuditMappedBy annotation'ı kullanılabilir. Bu kullanıcılara (mappedBy kullarak) tersi özelliği belirtmesine olanak sağlar. Indexlenmiş dizinlerde, index kolonunun referans nesnede eşleştirilmiş olması gerekir ve positionMappedBy özelliğinin belirtilmesi gerekir. Bu annotation sadece Envers kullanıldığında geçerli olur[14].

3.2. Audit (OpenJPA)

OpenJPA'da tam anlamıyla bir veri tarihçe izleme ürünü bulunmamakla birlikte, birkaç basit adımda tüm kalıcı varlıklar için izleme yöntemi etkinleştirilebilir.

1. Konfigürasyon:

Herhangi bir kalıcı varlık, annotation (Java ek açıklaması) kullanılarak izleme için etkinleştirilebilir, org.apache.openjpa.audit.Auditable. auditable annotation'ı ile oluşturma, güncelleme ve silme işlemleri için izlemeyi etkinleştirir. META-INF/persistence.xml yapılandırma dosyası içinde yapılacak düzenleme ile JPA Audit çağırılabilir.

<property name="openjpa.Auditor" value="default"/>

Bu şekilde oluşturulan izleme işlemi aslında çok fazla şey yapmaz. Tüm izlenebilir varlıkların son ve ilk durumlarını standart dış konsola veya belirtilebilir bir dosyaya yazar.

2. Özel İzleme Geliştirme:

Gerçek kullanımlar için herhangi bir uygulama, veri değişimlerinin yazılmasından daha fazla şey tercih eder. Bu yüzden uygulama, herhangi bir durum için de olsa, org.apache.openjpa.audit.Auditor arayüzünü uygulamak durumunda kalacaktır. OpenJPA, veritabanına bir kayıt yapmadan önce bu metodu çalıştırır. Bu metod sayesinde uygulama 3 tane izlenebilir obje elde eder, bu objeler org.apache.openjpa.audit.Auditable tipinde 3 ayrı bloktan oluşur ki bunlar newObjects (yeni kaydedilecek objeler), updates (güncellenen objeler) ve deletes (silinen objeler) obje bloklarıdır. Auditable sınıfı, bir kayıt nesnesinin en son ve orijinal durumunu tutar[15].

3.3. History Policy (EclipseLink)

History Policy Eclipse Link içinde veri izlemeye yönelik oluşturulmuş bir yol olarak gözükmektedir, bu sayede EclipseLink kullanılan uygulamalarda veritabanında yapılan tüm değişikliklerin iz kaydının tutulması desteklenmektedir.

Histrory Policy, zaman içinde herhangi bir noktada bir nesnenin durumunu kaydedecek bir ayna tablo oluşturmak üzere yapılandırılabilir. Böylece veri tarihçeleri oluşturulabilir, zaman içinde geçmişteki noktalarda sorgulama yapılabilir ve eski veriler tekrar geri kaydedilebilir.

Asıl tabloya herhangi bir veri kaydetme işlemi gerçekleştiğinde, tarihçe tablosuna bir satır eklenir. Tarihçe tablosuna ayrıca başlangıç ve bitiş tarih kolonları tanımlanabilir [16].

Örnek tarihçe işlemi;

```
public class HistoryCustomizer implements DescriptorCustomizer {
    public void customize(ClassDescriptor descriptor) {
        HistoryPolicy policy = new HistoryPolicy();
        policy.addHistoryTableName("EMPLOYEE_HIST");
        policy.addStartFieldName("START_DATE");
        policy.addEndFieldName("END_DATE");
        descriptor.setHistoryPolicy(policy);}
```

3.4. EntityAuditBundle (Doctrine)

Doctrine PHP yazılım dilinde MVC (Model-View-Controller) mimarisi ile yazılmış bir web uygulama çatısı olan symfony üzerinde kullanılır. Bu proje, tıpkı JPA sağlayıcıları gibi, veri saklama ve PHP program kütüphaneleri arasında eşleştirme hizmetlerini sağlamaya odaklanmıştır.

EntityAuditBundle, Doctrine uygulamasının bir uzantısı olarak yeralır ve Hibernate Envers'ten esinlenerek entity'lerin ve ilişkilerinin sürümlerini tutar.

Bu uzantı her izlenecek tablo için _audit uzantısı ile biten bir eşlenik tablo tutar, eşlenik tabloda tüm kolonların yanısıra rev ve revtype adında iki ek alan olur. Bunların yanında genel bir revizyon tablosu vardır, bu tabloda id, zaman, kullanıcı ve değişiklik açıklaması alanları vardır. Bu yaklaşımda, belirli bir zamanda uygulamanın ilişkili tüm değişiklikleri versiyonlanabilir.

Bu uygulamada yapılacaklar listesinde ise; kalıtımsal tablolarda henüz tam çalışmaması ve many-to-many (çoka çok) ilişkilerin versiyonlanamaması vardır [17].

Tablo 3.1. Veri değişiklik yöntemlerinin karşılaştırma tablosu.

Ürünler	Veritabanı Triggerlar	Özel çözümler	Hibernate Envers	Yeni yaklaşım
Revizyon Entitylerin otomatik oluşturulması	YOK	VAR	VAR	VAR
Geriye Dönük Sorgulama	VAR	VAR	VAR	VAR
İleriye Dönük Kayıt	YOK	YOK	YOK	VAR
Collections of Components	YOK	YOK	YOK	VAR
@OneToMany + @JoinColumn	YOK	YOK	YOK	VAR
Desteklediği Platform	Veritabanı bağımlı	Sadece özel çözümün kullanıldığı platform	Hibernate	Tüm JPA standartları; Hibernate, TopLink, EclipseLink, OpenJPA

BÖLÜM 4. YENİ YAKLAŞIMDA KULLANILAN TEKNOLOJİLER VE MİMARI TASARIM

4.1. Yeni Yaklaşımda Kullanılan Teknolojiler

4.1.1. Java database connectivity (JDBC)

JDBC Java programlama diliyle kullanılan, sanal olarak her türlü tablosal veriye erişim sağlanmasına olanak tanıyan yapıdır. JDBC yapısı java programlama literatüründe JDBC API olarak anılır. JDBC API uygulama geliştiricilerin java programlama dilini kullanarak endüstüriyel güçte veritabanı uygulamaları yazmalarına olanak sağlar. SQL yordamlarının ilişkisel veri tabanlarına gönderilmesini kolaylaştırır, bunun da ötesinde veritabanı haricinde tablosal veri nin tutulduğu dosya sistemleri ile etkileşimide sağlar. Veritabanı yönetim sistemi bağımsız olarak veri erişimi yazılan JDBC kodu ile sağlanır.

JDBC API sürücüleri aşağıdaki işlemleri gerçekleştirir.

1. Veri kaynağı ile bağlantıyı sağlar

2. Veri kaynağına güncelleme ve okuma sorguları nı yollar

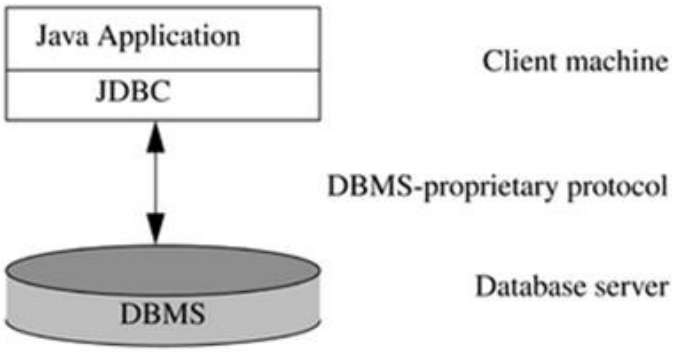
3. Sonuçları işler

JDBC API de veritabanı erişimini sağlanması için iki mimari kullanılır. Bunlar iki katmanlı ve üç katmanlı modeldir [19].

4.1.1.1. İki katmanlı veri erişim modeli

İki katmanlı erişim modelinde java applet veya java uygulaması direk veri kaynağı ile konuşur. Belirli bir veri kaynağına iletişim kurulabilmesi için JDBC sürücüsüne ihtiyaç vardır. Kullanıcının komutları veritabanına yada başka bir veri kaynağına

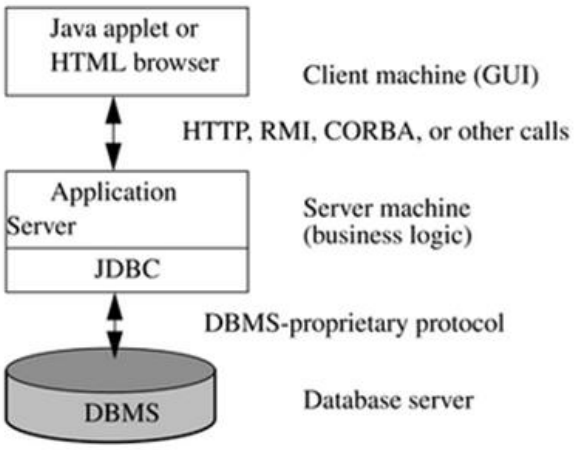
ulaştırılır ve bu yordamların sonuçları yine kullanıcıya iletilir. Veri kaynağı kullanıcının ağ bağlantısı ile bağlı olduğu başka bir lokasyonda olabilir. Bu model istemci/sunucu mimarisi şeklinde çalışır kullanıcı bilgisayarı istemci veri, kaynağının bulunduğu makina ise sunucu rolündedir. Şekil 4.1.'de iki katmanlı veri erişim mimarisinin detayı bulunmaktadır [19].



Şekil 4.1. İki katmanlı veri erişim mimarisi [19].

4.1.1.2. Üç katmanlı veri erişim modeli

Üç katmanlı veri erişim modelinde komutlar önelikle üçüncü katmana gönderilir sonrasında üçüncü katman bu komutları veri kaynağına iletir. Veri kaynağı komutları işledikten sonra tekrar işlenmiş veriyi üçüncü katmana iletir sonrasında üçüncü katman sonuçları kullanıcıya döner. Bu model veri erişimi izinlerini yönetebilen bir ara katman sağlaması, uygulamaların yüklenmesindeki kolaylık ve performans konusunda ki avantajları nedeniyle daha tercih edilen bir modeldir. Şekil 4.2.'de üç katmanlı veri erişim mimarisinin detayları mevcuttur.



Şekil 4.1. Üç katmanlı veri erişim mimarisi [19].

Eski zamanlarda üç katmanlı veri erişim modeli c, c++ için daha iyi bir performans sunmakta idi. Java bytecode larını makine diline çeviren derleyicilerin performanslarının iyileştirilmesinden sonra Java platformu üç katmanlı kod geliştirme, standart haline gelmiştir. Yazılım geliştiricilere bu mimari sağlamlık, multithreading, güvenlik özellikleri, bağlantı havuzu (connection pooling), ayrık işlem yeteneği olanaklarını sunmuştur.

JDBC API iki ve üç katmanlı mimarilerde ana rolü oynar. Sunucu kodu yazan Java programlama dilini kullanan işletmelerde JDBC API çok yaygın olarak üç katmanlı modelin orta katmanında yer alır [19].

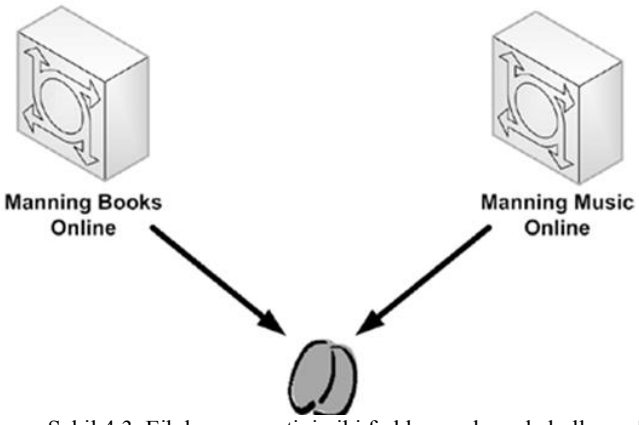
4.1.2. Enterprise java beans (EJB)

EJB java programlama dilini kullanarak taşınablilir, yeniden kullanılabilir, ve ölçeklenebilir iş uygulamaları yapılmasını sağlayan bir platformdur. İl ortaya çıktığından beri işlem yönetimi, güvenlik, otomatik kayıt gibi uygulama yaparken gerekli olan kavramları yeniden bulma gereksinimi olmadan Sirket Java uygulamaları geliştirmeyi sağlayan component model yada çerçeve olarak lanse edilmiştir. EJB yazılım geliştiricilere yapısal kodlamaya girmeden sadece iş mantığı üzerine kod yazmaya odaklanmaya olanak sağlar.

Yazılım geliştiricilerin bakış açısı ile EJB özel runtime ortamlarında çalışan ve içinde component ve servisleri barındıran EJB konteyner ları çalıştıran java kod parçacıklarıdır. Özelleştirilmiş çerçeve(framework) tarafından sağlanan kayıt servisleri kayıt sağlayıcı olarak adlandırılır (Peristence Provider) [20].

4.1.2.1. EJB komponentleri

Komponent mantığının arkasında içinde uygulama davranışını barındırmasıdır. Komponent in kullanıcısı komponentin içsel çalışma mantığını bilmez sadece uygulamaya vereceği parametre ile beklediği parametre ile ilgilenir. Üç Ejb component türü vardır bunlar session bean, message-driven bean ve varlıklar (entities) dır. Session bean ve message-dirven beanler iş mantığını ejb uygulamasında implemente etmek için kullanılırlar. Varliklar (Entities) ler ise kayıt işlemleri için kullanılır. Bileşenler (Components) yeniden kullanılabilir yapılardır ve değişik modullerin veya uygulamaların ortak işlemleri için aynı ejb bileşenlerini kullanılır. Şekil 4.3.'de bir ejb komponentinin iki farklı uygulamada ortak olarak kullanışı gösterilmiştir [20].

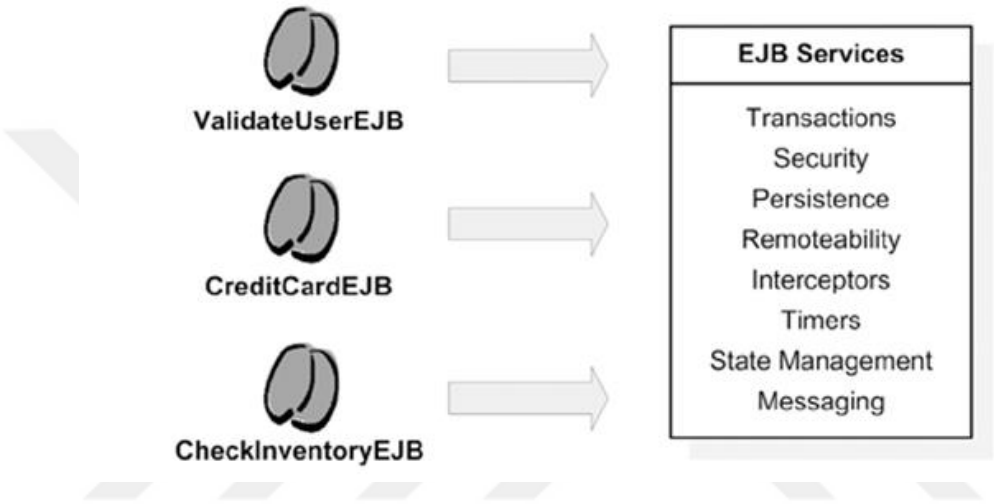


Şekil 4.3. Ejb komponentinin iki farklı uygulamada kullanışı [20].

4.1.2.2. EJB çerçeve mimarisi

Ejb bileşenlerri konteynerlar içinde yaşarlar. Şirket uygulamaları geliştirmeye değer katan bileşenler ve konteynerlar çerçeve (framework) olarak adlandırılırlar. Çoğu sunucu tarafı uygulamalar iş mantığı, uygulama durum kontrolü, ilişkisel veri tabanlarından kayıt okunması ve yazılması, işlemlerin yönetilmesi, güvenlik

kavramlarının uygulanması, senkron olmayan işlemlerin yönetilmesi, diğer sistemlerle entegrasyon gibi ortak gereksinimlere sahiptir. Ejb çerçevesi (framework) si ejb konteynerları vasıtasıyla bu işlevselliklerin her uygulama için yeniden yazılmasını önleyerek uygulamalarda bu ortak yapının kullanılması için servis sağlarlar. Şekil 4.4.'de EJB Framework ünün ejb komponentlerine sağladıkları servisler gösterilmiştir [20].



Şekil 4.4. Örnek EJB Frameworkünün ejb komponentleine sağladığı servisler [20].

Ejbler yüklenirken konteynerların komponentlere sağlıyacağı servisler metadata açıklamaları yoluyla önceden belirlenir. Java5 ile beraber metadata açıklamaları ortaya çıkmıştır. Bu açıklamalar sınıf yada metodların önünde tanımlanır. Bu bildirim stili programlama kategorisine girmektedir. Yazılım geliştirici hangi işlemin yapılacığına karar verir, sistemde tanımlanan açıklamaya göre gerekli işin yapılması için kod u ekler.

Metadata açıklayıcıları uygulamanın geliştirmesini ve testini kolaylaştırır. Geliştiricilere EJB komponentlerini metot yada sınıfların başına bu deklarasyonları ekleyerek rahatlıkla kullanabilirler. Şekil 4.5.'de metadata açıklamaları ile basit bir pojo sınıfın ejb halini aldığı gösterilmektedir [20].

Şekil 4.2. Açiklama (Annotation) eklenerek pojo sınıfından ejb oluşturulması [20].

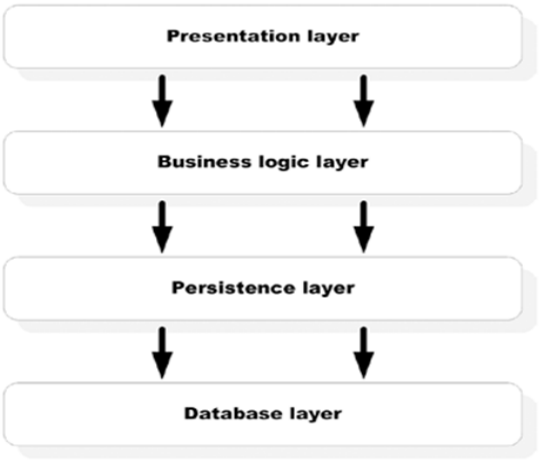
4.1.2.3. Tabakalı mimari ve EJB

Neredeyse tüm şirket uygulamaları bir çok bileşene e sahiptir. Bu uygulamalar belli bir müşteri problemini çözmek için oluşturulmuşlardır fakat içlerinde yaygın olarak kullanılan karakteristikleri barındırırlar. Örnek vermek gerekirse her şirket uygulamasının bir ön yüzü mevcuttur ve bu uygulamalar iş süreçlerini implemente ederler, problem domain ini modellerler ve veriyi veritabanına kayıt ederler. Bu ortak kullanımlar nedeniyle şirket uygulaması geliştirirlirken yaygın olarak kullanılan mimari ve dizayn prensipleri mevcuttur bunlara kısaca pattern olarak adlandırılır. Sunucu tarafaındaki geliştirmeleri için kullanılan yaygın patern tabakalı mimari (Layered Architecture) dir. Tabakalı mimaride komponentler katman (Layer) larda guruplanır.

Her katman önceden tam olarak kendisi için belirlenmiş işi yapan fabrika hatları gibidir. Her hat kendisi için belirlenmiş olan işi yaptıktan sonra diğer işlerin yapılması için kendinden sonraki hatta işi yollar. EJB uygulama geliştirilirken iki çeşit tabakalı mimarinin uygulanmasına izin verir. Bunlar geleneksel 4 katmanlı mimari ve Etki alanı odaklı tasarımdır (Domain.-Driven Design –DDD) [20].

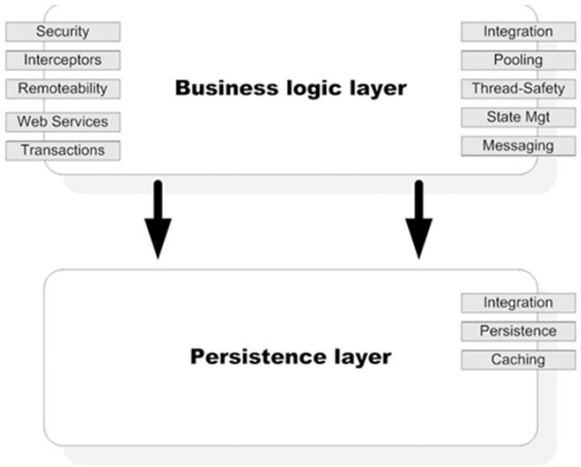
4.1.2.4. Dört katmanlı mimari

Şekil 4.6.'da geleneksel 4 katmanlı mimariyi göstermektedir. Bu mimari yaygın olarak kullanılan mimaridir. Mimarideki sunum (presentation) katmanı kullanıcı arayüzünün gösteriminden sorumludur.



Şekil 4.3. Dört katmanlı mimari[20].

Bu katmandan kullanıcının yolladıpı istekler iş mantığı katmanına gönderilir. Bu katmanda iş mantığı ile ilgili fonksiyonaliteler işletilir. İş mantığı katmanı uygulamanın kalbidir ve iş ile ilgili fonksiyonaliteler bu katmanda yer alır. İş mantığı katmanından veritabanına kayıt edilecek çıktılar kayıt katmanına (persistence layer) yollanır. Kayıt katmanı gelen bilgiyi nensneye dayalı mimariden veritabanı yönetim sisteminin anlıyacağı şekile dönüştürür ve veritabanı katmanına yollar. Şekil 4.7.'de işmantığı katmanındaki servisler ile katmanındaki etkileşimin detayı yer almaktadır [20].



Şekil 4.4. Dört katmanlı mimaride iş mantığındaki EJB servisleri ile kayıt katmanındaki servislerin etkileşim detayı [20].

4.1.2.5. Etki alanı odaklı tasarım (Domain-driven Design- DDD)

EJB'de devamlılık konusu da veritabanı ile ilişkileri içermektedir. EJB v2 'ye kadar devamlılık entity bean (Varlık Nesneleri)'ler vasıtayla sağlanmaktaydı. Fakat gelişen programlama dünyasıyla ortaya çıkan JPA ve JDO gibi standartlar nedeniyle, EJB3 ile birlikte devamlılık EJB ana katmanlarından ayrılmış ve yerini bu yeni standartlara bırakılmıştır. DDD de etki alanı objelerinin veritabanı objelerinin sadece bir kopyası değil iş mantığını da içermesi gerekir. EJB 3 de etki alanı objeleri varlık (entities) olarak adlandırılır.

Bu tasarım mimarisi yanlızca EJB3 ile gelen bir özelliktir. Diğer eski ejb versiyonlarında nesne tabanlı özelliklerinin bir çoğunu deteklememesinden dolayı bu tasarım metodu uygulanamaz. EJB3 de Java Varlık API si (JAVA Persistence API-JPA) ile gerekli kalıtım ve çokyüzlülük gibi nesne tabanlı özellikler desteklenmiş hale gelir [20].

4.1.3. Java persistence API (JPA)

Java Persistence API java saklama teknolojisi için kullanılan hafif POJO (Plain Old Java Object) temelli bir çerçeve(framework) dur. Bu API'nin ana bileşeni nesne-ilişkili adreslemedir. Büyük ölçekli uygulamalarda veri kayıt mekanizmasında mimari açıdan çözüm sunar [21].

Veri içeren nesneleri ilişkisel bir veritabanına kaydeden bir java standart arayüzüdür. JPA standartı, java annotation'lar (java nesne ek açıklamaları) ile sorgular kullanarak nesneleri çekmenin ve işlemler kullanarak veritabanı etkileşimlerinin arayüzlerini tanımlar. JPA arayüzünü kullanan bir uygulama, herhangi bir veritabanı kodu kullanmadan farklı veritabanları ile çalışabilir ve böylece JPA farklı veritabanı sunucuları arasındaki uygulama taşıma işini kolaylaştırır. İlk versiyon JPA 1.0 tanımlaması JSR 220'nin bir parçası olarak 11 Mayıs 2006 'da yayınlanmıştır. JPA 'nın son tamamlanmış tanımlaması JPA 2.0, JSR 317'nin parçası olarak 10 Aralık 2009'da yayınlanmıştır. JPA, veritabanı tarafında tüm tabloların program tarafındaki

bir sınıfa karşılık denk gelmesi ve tüm işlemlerin bu sınıf üzerinde yapılmasını sağlar [21].

4.1.3.1. Klasik JDBC bağlantısına göre avantajları

1. Sınıflar ile çalışmak:
Nesne yaklaşım modelini baz alan Java programlama dilinde veritabanı ile alakali bir işlem yapıldığında, string verilerle degil de OOP (Object Oriented Programing – Nesnesel Yaklaşım Yazılım) 'ın en temel yapısı olan class (Sınıf) ve object (Nesne)'lerle uğraşılır.

2. Sadelik:
Geliştiriciler, veritabanı yönetim ve kullanımından ayrılarak asıl yapması gereken iş mantığına odaklanırlar. Veritabanı tarafindaki tabloların daha standart ve uyumlu bir sınıf haline çevrilmesi sağlanır.

3. Güvenlik:
Bir çok dış güvensizliğe karşı önlemler barındırılır.

4. Hata önizleme:
SQL ile yazılabilecek basit seviyedeki sorgularda oluşabilecek yanlış yazımlardan kaynaklanan hataların önüne geçilir.

5. Modelleme kolaylığı:
JPA kullanılarak MVC (Model View Controller - Veriler Arayüz Kontrolcü) gibi bir mimari desenin model katmanı daha profesyonel bir yaklaşım ile yapılabilir [21].

4.1.3.2. Java persistence API (JPA 2.0) kullanan altyapılar

JPA 1.0 yayınlandıktan sonra, bu standardı kullanmaya çalışan popüler sağlayıcılar tam bir uzlaşma içine giremediler ve bu nedenle JPA 2.0 sürümü ortak özellikleri bulabilmek için yayınlandı.

Bu sürümün içerdiği temel özellikleri şunlardır:

1. Genişletilmiş nesne ilişkisel eşleme işlevselliği
2. İlişkisel veritabanlarında kullanılan çoka-bir (many to one) ilişkinin java'da kullanılan collection sınıflarınca desteklenmesi
3. Embedded nesnelerin çoklu seviyesi
4. Sıralı listeler
5. Erişim türleri kombinasyonları
6. Sorgularda kriter kullanımı
7. Sorgularda kullanılan ipuçlarının (hints) standardizasyonu
8. DDL üretimini desteklemek için ek meta data standardizasyonu
9. Doğrulama (validation) için destek

JPA 2.0 standartını kullanan belli başlı sağlayıcılar şunlardır;

1. Hibernate
2. EclipseLink (önceden Oracle TopLink)
3. OpenJPA
4. DataNucleus (önceden JPOX)
5. ObjectDB
6. BatooJPA

1. Hibernate:

Hibernate Java programlama dili için bir nesne ilişkisel eşleştirme (ORM Object Relational Mapping) kütüphanesidir. Nesne yönelimli etki alanı (Object-oriented domain) modeli ile ilişkisel veritabanı arasında bir eşleştirme altyapısı sunar.

Hibernate Jboss altında geliştirilen ücretsiz bir yazılımdır. En önemli hedefi Java sınıflarını veritabanı tablolarına ve Java veri tiplerini SQL veri tiplerine eşleştirmek, ayrıca veri sorgulama da yapmaktır. Sorgulamada SQL'den esinlenen Hibernate Sorgulama Dili (HQL) kullanır, HQL hibernate veri nesneleri üzerinde sorgulama yapabilmektedir. Hibernate tek başına Java uygulamalarında ve servlets, EJB sessions benzeri yapıların kullanıldığı Java EE uygulamalarında kullanılabilmektedir.

Hibernate altyapısı içinde bir çok parçadan oluşmaktadır bunlar Hibernate ORM (temel fonksiyonlar), hibernate annotations (nesne ilişkisel model ve ilişkisel veritabanı modeli arasındaki geçişi sağlayan temel bilgiler), hibernate entitymanager (Hibernate Annotations ile birlikte ORM'nin üst katmanını oluşturur), Hibernate Envers (veritabanı kayıt sınıfları için izleme ve versiyonlama), Hibernate OGM, Hibernate Shards (çoklu ilişkisel veritabanlarında yatay ayrıştırma), Hibernate Search (tam yazı araması yapan Apache Lucene ile JPA modelin uyarlamanması), Hibernate Tools (Eclipse için araçlar), Hibernate Validator (doğrulama), Hibernate Metamodel Generator dir [22].

2. EclipseLink:

EclipseLink, Eclipse Foundation tarafından üretilen bir açık kaynak projedir. Bu proje Java geliştiricilerinin veritabanları, web servisleri, Nesne XML eşleştirme (OXM - Object XML Mapping), kurumsal bilgi sistemleri gibi çeşitli veri servisleri ile etkileşimlerini sağlayan, gelişen bir altyapı yazılımıdır.

EclipseLink, daha önceleri oracle tarafından geliştirilen TopLink ürünü baz alarak ortaya çıkmıştır. TopLink ürünü üzerinde bazı Oracle'a has parçalar yaygınlaştırılarak geliştirilmeye devam edilmiştir [23].

3. OpenJPA:

OpenJPA açık kaynak kodlu bir Apache Software Foundation projesidir. Tek başına bir POJO (Plain Old Java Object) persistence katmanı ile veya Java EE standartlarını içeren bir altyapı ile birlikte kullanılabilir, örneğin Tomcat, Spring [24].

4. DataNucleus:

DataNucleus projesi bir Java dünyasında uygulama verilerinin yönetimi için ürünler sağlar. DataNucleus Apache 2 lisansı ile açık kaynak ürünlerdir. Standardize olmuş JDO, JPA gibi altyapılar altındaki çok geniş bir yelpazedeki veritabanı sistemlerinde (İlişkisel veritabanları, Harita tabanlı veritabanları, Web tabanlı veritabanları,

Dokümanlar xml ve xls gibi, Grafik tabanlı veritabanları vb. gibi) geçerlidir ve tüm bu sistemlerde sorgulama dillerini de desteklemektedir [25].

5. ObjectDB:

ObjectDB, Java sınıfları ve veritabanı tablolarını eşleştiren bir altyapı değil, Java için oluşturulmuş bir nesne veritabanıdır. Diğer veritabanlarının aksine ObjectDB, Java API'lerinden JPA ve JDO (Java Data Objects) standartlarını kendi yapısı içinde barındıran bir veritabanı servisidir. Bu nedenden dolayı Java ve veritabanı arasında eşleştirmeleri sağlayan yazılımlara gereksinim duymaz.

ObjectDB, son derece kolay kullanıma sahip bir Java Nesne Veritabanıdır. Eşleştirme gerektirmeden JPA ve JDO standartlarını sağlar. Diğer tüm veritabanı özelliklerinin (veritabanı sorgulama, görüntüleme, backup, replikasyon, transaction vb.) hepsini sağlar ve JDBC, sürücüler, tablolar, kayıtlar, ORM (Object Relational Mapping) yazılımlarına gerek yoktur. Sadece Java sınıfları ve nesneleri ile daha kullanışlı veritabanı kodlaması gerçekleştirilebilir [26].

JDO, java nesne devamlılığı için oluşturulmuş bir spesifikasyondur, veri tabanlarında kalıcı verilere erişmek için standart bir yol sunar. Kalıcı verileri temsil etmek için POJO (Plain Old Java Objects - Düz Java Nesneleri, veri taşıyıcı java sınıfları) kullanır. Bu yaklaşım ile veri işlemleri ile veritabanı işlemlerini ayrıştırır. JDO ilişkisel veritabanlarının yanısıra düz veri kaynakları ile de çalışılabilmesini sağlar.

JPA 3.0 standartını kullanan belli başlı sağlayıcılar şunlardır;
1. DataNucleus (JPOX)
2. ObjectDB

4.2. Nesneye Dayalı Mimari Yapıda İşlem Denetimine Temel Bakış

Yaklaşımda kullanılan nesneye dayalı mimaride nesne (0, A, M) üçleme ile ifade edilir. 0 benzersiz nesne belirleyicisi, A örnek değişken kümesi, M ise method kümesidir. Her örnek değişken çifttir (Ai, V(Ai)), Ai örnek değişken ismi, V(Ai) (Ai

nin değeri) ya ilkel bir nesne yada obje belirleyicisidir. Aynı şekilde her method çifttir (Mi, P (Mi)), Mi method adı ve P (M i) program adı olarak belirtilebilir.

İsim ve değer çiftleri yerine örnek değişkenleri ve yöntemleri olarak adlandıracağız. Nesneler ileti gönderip alabilirler. Denklem(4.1) de ileti tanımı yer alır.

$$\mu = (m, I) \quad (4.1)$$

m ileti adıdır. Denklem(4.2) parametre listesini ifade eder.

$$l = (P I , ..., pk), \ p l , ..., pk \quad (4.2)$$

Eğer l boş ise Parametre sayısını belirten k 0 degerini alabilir.Nesne iletiye Denklem(4.1) deki iç yöntemi çağırarak cevap verir.

Basitleştirmek için her yönteme karşılık gelen her iletinin yöntemle aynı isimde olduğunu varsayıyoruz. Örnek olarak sınıfın bir nesnesi olan QUEUE, DEQUEUE mesaja cevap olarak adlandırılacaktır. Onun için μ e verilecek doğru cevap yöntemin aynı adlı programını bulup onu çalıştırmaktır.

Nesne ye mesaj yollamak ve yöntemin çalısarak nesneye bir dönüş değeri olarak cevap almak işlemine nesne üzerindeki operasyon olarak adlandırılmaktadır.

Nesnenin dış katmandan görünümü (0 , F) dir. Denklem(4.3) de F nesnenin arayüzü tanımlanmıştır.

$$F = (M1 ,..., Mn) \quad (4.3)$$

F nesnenin Verdiği tüm cevap isimlerini barındırır.(Nesnenin muhteva ettiği tüm yordam isimlerini). Arayüzler nesnenin kapsadığı tüm operasyonları belirler. Arayüzler yordamların icindeki örnek değişkenlerinin isim ve değerlerinden oluşur. Arayüzler yordamların içindeki mantıksal islemlerin dış dünyadan saklanmasını sağlarlar.

4.2.1. Nesneye dayalı mimarilerde işlem denetim modeli

Nesneye dayalı modelin denetimi içim 3 ana nesne tipine ihtiyaç vardır.

4.2.1.1. Denetlenebilen veri objeleri

Bölüm 4.2'de değinilen nesnenin değişkenlerinden Ai denetim açısından bakıldığında tekil değil değer setleri bakışıyla ele alınmalıdır. Set deki her değer değerin yaratılmasından sorumlu olan kullanıcı bilgisi ve zaman değeri ile işaretlenir. Bu bakışa göre yeni tanım aşağıdaki gibi olur;

V(Ai)(Denetlenebilen veri objesi) = ((vi^1, Ui^1, ti^1), ..., (Vit, Uit, tit)) (4.4)

Bu üçlü setler zaman damgalarına göre sıralanır (t1i). Ai güncel değerini temsil etmesi yerine V(Ai) olarak tüm tarihsel geçmişi temsil etmesi amaçlanmıştır.

Ai yi instance ın güncel değeri olarak tanımlarsak Ai için üçlemede (V(Ai)) en yüksek zaman damgasına sahip olur. Cur(Ai) yi Ai nin şu anki değerini gostermek için kullanırsak;

Güncel(Ai) = vim s.t. tim = max1l≤j≤r(t1i) (4.5)

V(Ai) sıralı bir set olması nedeniyle Vİr' yi V(Ai) nin son üçlemesi olması nedeniyle aşağıdaki şekilde gösterebiliriz.

Güncel(Ai)=Vİr (4.6)

Nesne üzerinde yapılan operasyon instance değişkenin de değişikliğe yol açtığında değişkenin eski değerinin değişmesi yerine V(Ai) ye (vir+1,uir+1,lir+1) olarak yeni bir üçleme eklenmesi ile sonuçlanır [27].

1.Kullanıcı Nesneleri

Veri nesneleri ile kullanıcı nesneleri ayrı ayrı ele alınır. Kullanıcı nesnelerinde kullanıcı hakkındaki bilgiler tutulur. Kullanıcı bazlı yapılan işlemlerin tarihsel olarak

denetime tabi tutulmasında bu kullanıcı nesneleri ile veri nesneleri arasındaki ilişkisel kayıtları kullanıcı bazlı denetiminde rol oynar [27].

2. İşlem Nesneleri:

Nesneye dayalı mimaride işlem nesne üzerinde kullanıcı tarafından yapılan ve baska bir nesne nin tetiklenmesi ile gerçekleşen olaylardır [27]. İşlem nesneleri yordamlardan oluşur ve fiziksel veri de değişiklige yol açan yordamları barındırır. Kullanıcı bir method a mesaj yolladıgında yordam diğer bir nesnenin yordamına mesaj yollayarak gerekli işlemi yapmış olur. Literatürde işlem nesnelerinin yaptığı işlemleri belli bir veri bütünlüğüne göre kayıt altına alan modeler tasarlanmıstır bunlar işlem ağaçları adıyla anılırlar [27]. İşlem kullanıcı tarafından başlatılır ve ilişkisel nesneler ve onların yordamları arasında mesajlasılarak gerekli işlem yapılır. İşlem çocuk işlemlere bölünür ve her çocuk işlem içinde belli zaman damgaları tutulur. Kullanıcının başlattığı her işlem bir işlem ağacı olarak veritabanında saklanır ve her kullanıcı için bir işlem ormanını bu sayede oluşturulur. İşlem ormanı denetim bilgisi için ikincil bir kaynak oluşturur ve işlem ağacı ile kullanıcının işlem ormanı arasında bağlantı kurularak hangi kullanıcının hangi işlemleri yaptığı bilgisi elde edilir.

Model olarak detaylandırmak gerekirse O nesnesinin instance değişkeni A, μ(m,l) operasyonu sırasında güncellenirse yeni üçleme (v,u,t), V(A) değer setine eklenmmelidir. v değeri m yordamının çalıştırılması sırasında hesaplanır. Üçlünün diğer iki elemanı hesaplanamaz. Onlar işlem ağacındaki μ sayesinde bulunur. Tüm işlem ağacındaki t zaman damgasın daki güncelleme için kök ağaçtaki kullanıcı bilgisi (u) bize güncellemeden sorumlu kullanıcının bilgisini verir.

Günümüzde nesneye dayalı uygulamalarda bu işlemsel bilgilerin kayıt altına alınması için nesneler tanımlanırken işlem denetimlerini kayıt altına alınmmasını sağlayan yapılardan yararlanılır. Bu yapılar nesne tanımlanırken tanımlanan açıklamalar (annotations) sayesinde uygulamanın çalışması sırasında devreye girerek nesnelerdeki etkileşimi kayıt altına alabilirler, makaleye konu olan yaklaşımda da bu

yapılar kullanılarak işlem denetimi için gerekli olan dinleme mekanizmaları kullanılmıstır.

Bu yaklaşım sayesinde belli uygulama lara bağlı kalınmaksızın her hangi bir JPA standardını destekleyen java projesinde kullanarak uygulama katmanında veri değişimlerinin varlık bazlı olarak kayıt altına alınması ve bu işlemlerin aynı zamanda ileriye dönük planlanan değişikliklerinde otomatize edilerek ilişkisel veritabanlarındaki ileriye dönük değişiklerin yapılması ve kayıt altına alınması saglanmıştır.

4.3. Nesne Tabanlı Veritabanlarında Uygulama Alt Yapısı Kullanrak Tarihsel Veri Denetimi Yapan Yeni Yaklaşım

Nesneye dayalı olarak modellenen veritabanlarında yapılan işlemlerin kayıt altına alınması yaklaşımıızın ana konusudur. 3.Kısımda değinilen veri değişikliği işlem kayıtlarının saklanması ve denetlenmesi ile ilgili çözümler daha cok veri tabanı katmanında çalışan çözümlerdir. Uygulama katmanındaki çözümlerin ise farklı yapılara uyarlanmasındaki sıkıntılar bu yaklaşımın ortaya çıkmasında ana sebebi teşkil etmistir. Uygulamaya has üretilen çözüm yöntemleri doğal olarak başka altyapılara ve benzer dahi olsa diğer uygulamalara uyum sağlayamamakta ve genelleştirilememektedir. Özel çözümler uygulama için en uygun yöntem olmaktadır, fakat her özel çözüm yöntemi ayrı ayrı efor ve maliyet gerektirmekte ve tekrar kullanılabilmesi de pek mümkün olmamaktadır.

Tüm yapılan işlemlerde, işleme konu olan veriler birden fazla tablodaki verilerin karışımından oluşabilmekte, bu işlemlerde geçen veriler işlem bazında saklanmaktadır. İşlem tarihçeleri veritabanında tek bir tabloda saklanmaktadır. Veritabanındaki bu tek tablonun boyutu çok fazla büyük boyutlara ulaşmakta, tablodan geçmişe dönük sorgulamalar uzun süreler almaktadır. Ayrıca istenilen geçmiş bir tarihteki durumu anlamak için, saklanan işlem verilerinin (birden fazla tablodaki verilerden oluştuğu için) mantıksal süzgeçlerden tekrar geçirilmesi gerekmektedir. Mantıksal suzgeçler ile ilgili bilgi tutma politikaları alakalı çalışmalar yapılmıstır [3]. Kayıt altına alınan değişikliklerde redaction (belli tarih aralığındaki

verinin gorunmez olması) ve expuction (belli tarih aralığındaki verinin silinmesi) işlemleri gereksinimleri nedeniyle performans sıkıntıları yaşanmaktadır.

Yaklaşım ile ilgili konseptin anlaşılması ve nesne tabanlı yaklaşım konusunda bilgi sahibi olunmalıdır. İlgili referanslardan bilgi sahibi olunabilir [22,23]. Çalışmanın olumlu ve olumsuz tarafları için Tablo 4.1.' de SWOT analizine bakılabilir.

Tablo 4.1. Çalışmanın SWOT analizi.

	OLUMLU	OLUMSUZ
İÇ ETKENLER	• Geçmişe dönük veri saklaması • İleriye dönük veri saklaması • JPA standartlarının yeterli olması • Kolay entegre edilebilmesi	• JPA uygulayıcılarının standart dışı yöntemlerinde sıkıntı olabilir • Sadece Java platformlarında olması
DIŞ ETKENLER	• Açık kaynak kodlu olması • Kurumlara oluşacak bilgi birikimi sunulabilmesi	• Kurumların kullanmaya çekinmesi • Kurumların varolan veri saklama yöntemlerinin olması • İleriye dönük veri saklanmasına güvenilememesi

4.3.1. Uygulama bazlı arşivleme yapan platform bağımsız çalışacak yaklaşımın çalışma mantığı

4.3.1.1. Uygulamanın çalışmaya başlamasıyla oluşturulacak olan tabloların oluşturulması

Tablo oluşturmada aşağıdaki sıra takip edilir:

a. Önce değişiklik temel bilgilerini tutacak tabloların oluşturulması.

b.Zaman içinde değişimlerinin izlenmesinin istendiği uygulama tablolarının, öngörülen sistemde karşılık gelecek tabloların oluşturulması.

4.3.1.2. Uygulamanın normal çalışması sırasında, değişimleri izlenecek tablolarda bir veri değişikliği olursa, bu değişikliği kaydetme işlemlerinin yapılması

İşlemler aşağıdaki sıra ile gerçekleştirilir:

a. Geçmişte kalacak değişikliklerin kaydedilmesi.

b. İleriye yönelik bir değişiklik bilgisinin kaydedilmesi.

c. Sistemde kaydedilen veri değişiklerinin, sorgulamalarının düzgün olarak, çalışması sağlanması.

4.3.2. Mimari tasarım

Arşivlemede ihtiyaç duyulan bilgiler genellikle eski tarihteki verilerin kendisi olmakla beraber, verilerin değişiminin kim tarafından, hangi amaçla yapıldığının bilgisi de önemlidir. Bunların yanısıra uygulamaların önceliklerine göre veri değişiminin kaynağının ne olduğu, hangi uygulama sunucu tarafından yapıldığı, değişimin yapıldığı lokasyon bilgisi gibi bir takım özel bilgilere de ihtiyaç duyulabilmektedir.

Bu nedenlerden dolayı arşiv bilgilerine kullanıcı bilgisi ve işlem bütünlüğü bilgisi de eklenmiştir. Bu bilgiler uygulama tarafından sağlanırsa arşivde tutulacak, sağlanmadığı zamanlarda herhangi bir bilgi tutulmadan arşiv işlemlerine devam edilecektir. Bu iki bilginin yanısıra uygulamanın ihtiyacı olan herhangi bir ek bilgiyi de arşivde tutulmasını sağlayacak yapı oluşturulmuştur. Uygulamanın sağlayacağı herhangi bir bilgi anahtar değeri ve veri değeri olarak arşiv bilgilerinde tutlabilecektir.

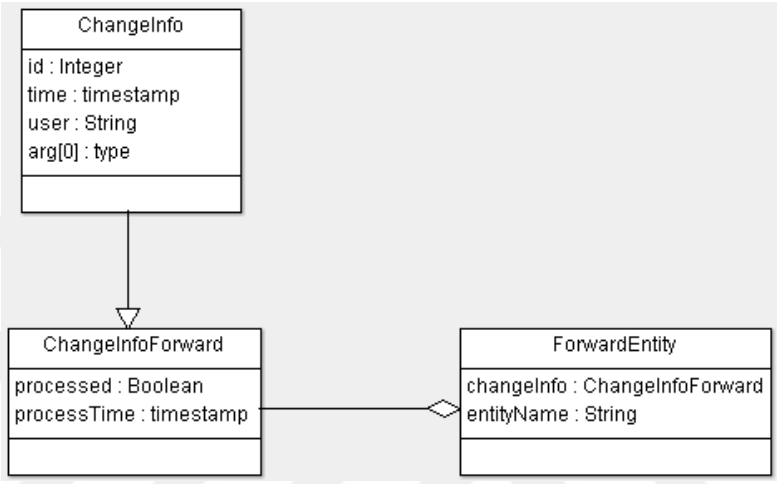
4.3.2.1. Denetim bilgilerinin yapılandırılması

Arşiv bilgilerinin ne şekilde saklanacağı yöntemi konfiguratif olarak bir sınıf ta tutulmaktadır. Ortam değişikliklerine ve ihtiyaca göre denetim kayıtlarının tutulacagı yer ve konfigurasyonu bir xml dosyasından okunup gerektiğinde değişiklik yapılmaktadır.

4.3.2.2. Denetim bilgilerinin saklanma yöntemi

Entity'ler üzerinde yapılan değişiklikler için bir ana sınıf oluşturulmuştur [ChangeInfo], bu ana sınıf üzerinde değişilik birincil anahtarı, değişiklik zamanı, değişikliği yapan kullanıcı ve uygulamanın koyacağı kendine özel bilgi yer almaktadır. İleri tarihli değişiklikler için [ChangeInfo] sınıfından türeyen ileri tarih ana sınıfı [ChangeInfoForward] oluşturulmus, bu sınıf üzerine ek olarak ileri tarihli

kaydın gerçek tabloya işlendiği bilgisi ve işlendiyse zaman bilgisi eklenmiştir. İleri tarihli değişiklikleri zamanı geldiğinde gerçek tabloya işlenmesini kolaylaştırmak için hangi entitylerin bu ileri tarihli kayıttan etkilendiği ayrı bir sınıfta tutulacak [ForwardEntity] ileri tarihli kaydın işlenme zamanı geldiğinde bu sınıf yardımı ile asıl entity ve bu entity'nin değişiklik entity'si bulunabilecektir. Denetim bilgilerinin tutulduğu anasınıflar Şekil 4.8.'de görülebilir.



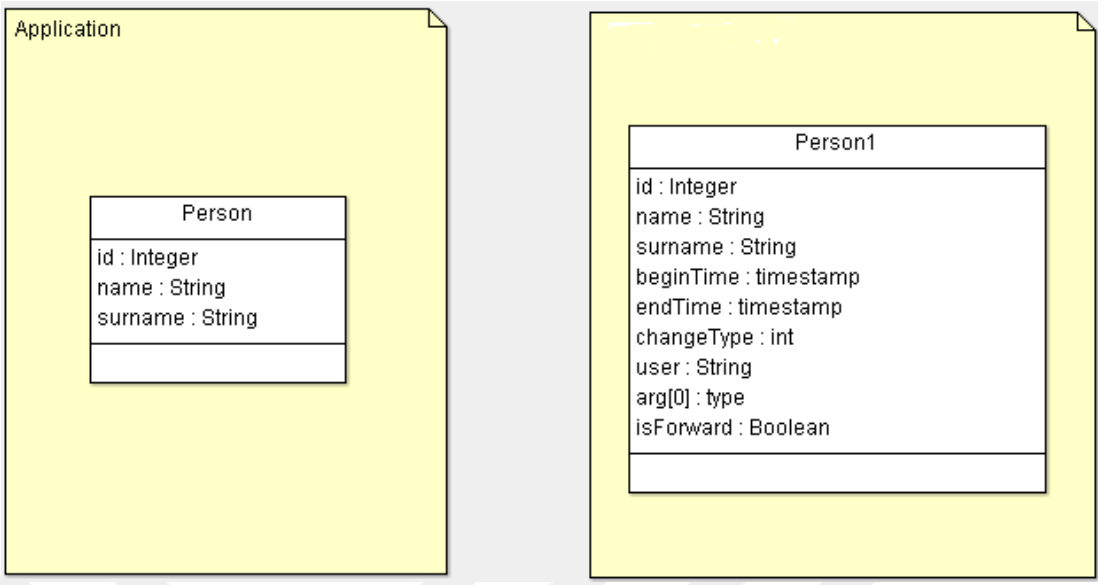
Şekil 4.8. Denetim bilgilerinin tutulduğu ana sınıflar.

4.3.3. Arşiv bilgilerinin veritabanında tutulma yöntemleri

Arşiv bilgilerinin veritabanında nasıl tutalacağı için birden fazla yöntem üzerinde çalışılmıştır. Bu yöntemlerden en uygun olanı belirlenip bu yöntem üzerinden geliştirme yapılmıştır.

4.3.3.1. Arşiv bilgilerinin asıl tablolarda tutulması

Bu yöntemde arşiv bilgileri veritabanındaki asıl tablolar üzerinde tutulur. Asıl tablolara arşivle ilgili ek alanlar eklenir. Bu alanlar; değişiklik başlangıç ve bitiş tarihleri, değişiklik tipi (veri kayıt, güncelleme ve silme), kullanıcı bilgisi, uygulamanın ekleyebileceği ek alanlar, ileri tarihli kayıt bilgisi olacaktır. Bu yöntem ile veritabanı işlemleri ve sorgulamalar çok hızlı olur, fakat asıl tabloların boyutu olması gerekenden çok daha büyük olur ve zamanla büyüyen tablo performansı olumsuz etkiler.Şekil 4.9.'da kayıt işleminde asıl tablonun değişimi gözlemlenebilir.

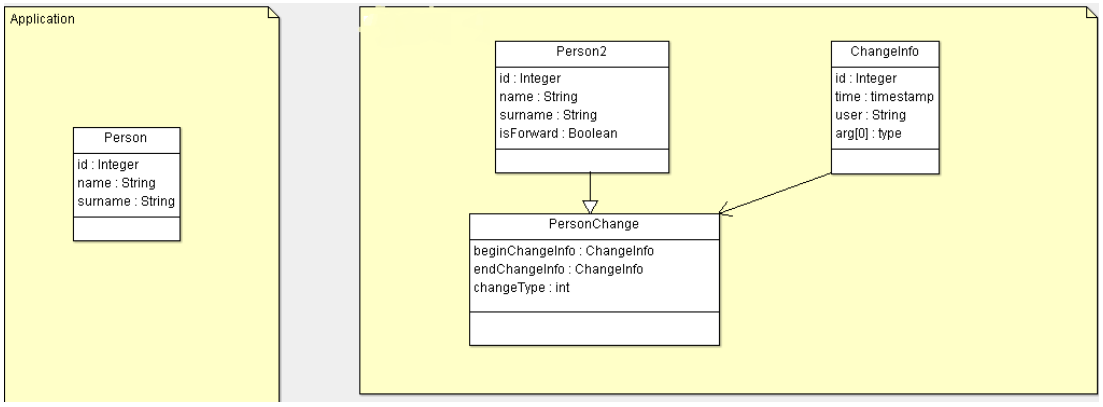


Şekil 4.9. Uygulama kayıt işlemleri kullanım senaryosu arşiv bilgilerinin asıl tabloda tutulması.

4.3.3.2. Arşiv bilgilerinin farklı tablolarda tutulması, ileri tarih bilgisinin asıl tabloda tutulması

Bu yöntemde değişiklik bilgileri veritabanında asıl tabloların bir kopyası alınarak tutulur. Yapılan her değişikliğin zamanı, kullanıcı bilgisi ve uygulamanın ekleyebileceği kendine özel bilgiler bir değişiklik ana sınıfında [ChangeInfo] tutulur. Asıl tabloların alanlarına, değişiklik başlangıç ve bitiş anahtarıları (değişiklik ana sınıfının birincil anahtarı) ve değişiklik tipi (veri kayıt, güncelleme ve silme) alanları eklenerek arşiv tablosu oluşturulur. Bu yöntem ile veritabanı işlemleri 1. yönteme göre daha yavaştır, fakat arşiv bilgileri asıl iş bilgilerinden ayrıştırılmış olur.

İleri tarihli kayıt olması durumunda bu kayıt asıl tabloya kaydedilir, ileri tarih zamanı geldiğinde herhangi bir kayıt, taşıma işlemi yapmadan, ileri tarih bilgisi silinerek asıl kayıt çok kolay biçimde oluşturulmuş olur, işlem kolaylığı sağlar. Fakat yine de iş bilgilerine iş ile ilgisi olmayan bir bilgi eklenmiş olur. Şekil 4.10.'da bu seçenek sonrasındaki kullanım senaryosu görülebilir.



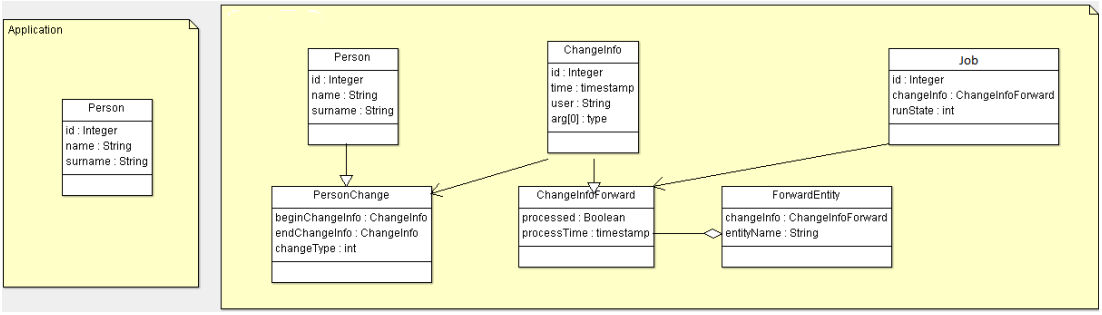
Şekil 4.10. Uygulama kayıt işlemleri kullanım senaryosu arşiv bilgilerinin farklı tabloda tutulması.

4.3.3.3. Arşiv bilgilerinin ve ileri tarih bilgisinin farklı tablolarda tutulması, ileri tarihli kayıtların gerçek yerine işlenmesi için Job (iş) oluşturulması

Bu yöntemde arşiv bilgileri yine veritabanında asıl tabloların bir kopyası alınarak tutulur. Yapılan her değişikliğin bilgisi de [ChangeInfo] sınıfında tutulur. İleri tarihli değişikliğin temel bilgileri de [ChangeInfo] sınıfından türetilen ileri tarihli değişiklik ana sınıfında [ChangeInfoForward] tutulur.

[ChangeInfoForward] sınıfında ek olarak ileri tarihli değişikliğin gerçek yerine işlenip işlenmediği bilgisi ve işlenme zamanı bilgisi tutulur. Bu sınıfa bir kayıt oluşturulurken aynı anda kaydın gerçek yerine işlenmesi için ileri tarihli zamanda çalışacak bir Java Job (iş) [Job] oluşturulur. Bu yöntemde job zamanı geldiğinde çalışır ve ileri tarihli kaydedilen değişiklikleri asıl uygulamaya kaydeder. Job'ın çalışmasını kolaylaştırmak için, ileri tarihli değişikliklerin hangi entity'ler üzerinde olduğu bilgileri [ForwardEntity] sınıfı üzerinde tutulur. [ForwardEntiy] sınıfı üzerindeki bilgiler [ChangeInfoForward] sınıfının birincil anahtarı ve entity isimleridir.

Bu yöntem ile uygulamanın iş mantığı ile değişiklik kayıt işlemleri tamamen ayrı sınıflar üzerinde ayrıştırılmış olur. İleri tarihli değişikliklerin işlenmesi için ek işler oluşturulmuş olur, bu işlerin herhangi bir şekilde aksaması riski de oluşmuş olur. Şekil 4.11.'de bu seçenek sonrasındaki kullanım senaryosu görülebilir.



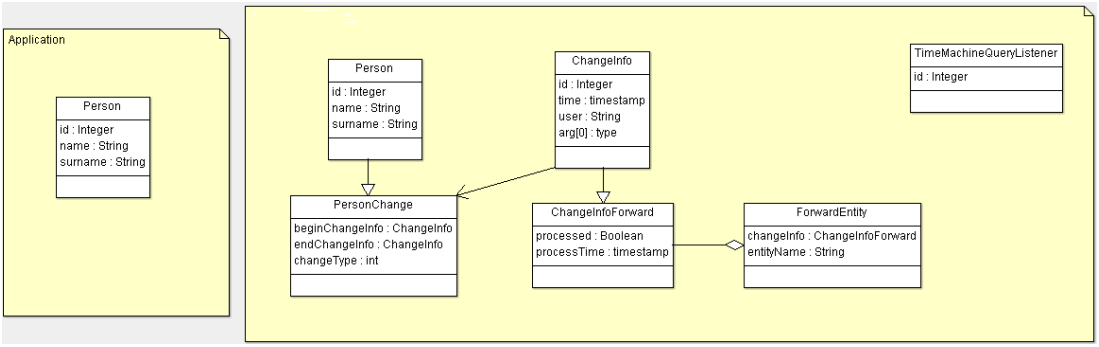
Şekil 4.11. Uygulama Kayıt İşlemleri Kullanım Senaryosu Arşiv bilgilerinin ve ileri tarih bilgisinin farklı tablolarda tutulması.

4.3.3.4. Arşiv bilgilerinin ve ileri tarih bilgisinin farklı tablolarda tutulması, ileri tarihli kayıtların gerçek yerine işlenmesi için asıl bilgilerin sorgulanmasının dinlenmesi

Bu yöntemde arşiv bilgileri yine veritabanında asıl tabloların bir kopyası alınarak tutulur. Yapılan her değişikliğin bilgisi de [ChangeInfo] sınıfında, ileri tarihli değişikliğin bilgisi de [ChangeInfoForward] sınıfında tutulur.

İleri tarihli değişikliklerin asıl yerine işlenmesi için asıl bilgilerin sorgulanması kontrol edilir. Bunun için sorguları dinleyecek bir dinleyici sınıf [QueryListener] oluşturulur. Asıl verilere ulaşacak her türlü (select, insert, update ve delete tipindeki sorgular) sorgunun çalışması öncesi [QueryListener] sınıfı dinleme yapar. Bu sınıfa gelen her sorguda eğer asıl entity için bir ileri tarihli değişiklik kaydı var ise ve ileri tarih zamanı geçtiyse, önceden kaydedilen ileri tarihli değişiklikler asıl yerine işlenir (işlem tipine göre kaydetme, güncelleme veya silme olabilir). Değişikliğin işlenmesi sonrası asıl sorgu normal şekilde çalışmasına bırakılır.

Burada bir önceki yöntemdeki oluşturulan işlerin çalışamaması riski ortadan kaldırılmış olur. Eğer uygulamaya JPA standartları dışında bir erişim olursa, ileri tarihli değişiklere sorgu gelmeden önce yapılacak her erişimde yanlış bilgi alma riski oluşur. İleri tarihli değişikliğin zamanı geçmiş ve henüz uygulama JPA standartları ile değişikliğin yapılacağı entity'e sorgu atmamış ise, JPA dışından erişimde, ileri tarihli değişiklik işlenmeden önceki bilgileri ulaşılmış olur, bu da sistemin yanlış çalışmasına neden olur. Şekil 4.12.'de bu seçenek sonrasındaki kullanım senaryosu görülebilir.



Şekil 4.12. Uygulama Kayıt İşlemleri Kullanım Senaryosu arşiv bilgilerinin ve ileri tarih bilgisinin farklı tablolarda tutulması, ileri tarihli kayıtların gerçek yerine işlenmesi.

4.3.4. Arşiv bilgilerinin veritabanında tutulma yöntemleri seçimi

Yöntemleri avantajları ve dezavantajları çıkarılmıştır. Uygulamaların genel kullanımlarına bakıldığında, öncelikli olarak uygulama verileri üzerinde hiçbir şekilde güncelleme, veri yapıları üzerinde bir değişiklik veya ekleme yapılamamasının daha sağlıklı olacağı görülmüştür. Yaklaşımın asıl amaçlarından biri de kolayca uygulamalara adapte edilmesi ve istenildiğinde de herhangi bir değişiklik yapmadan kullanım dışına bırakılabilmesidir. Bu nedenle uygulama verileri üzeirinde değişiklik öngören 1. ve 2. yöntemler uygulanabilir olmaktan çıkmış olmaktadır. 3. ve 4. yöntemler arasındaki en temel fark ileri tarihli değişikliklerin nasıl işleneceğidir. Burada 4. yöntemde kullanılacak bir sorgu dinleyicisi, işlemlerin daha güvenilir olmasını sağlamakla birlikte, JPA standartları dışındaki erişimlerde büyük sıkıntılara neden olacaktır. Kullanılan yaygın uygulamalar incelendiğinde verilere JPA standartları dışında da erişimlerin çok sık olabildiği görülmüştür. Örneğin çoğu uygulama temel yapısının yanısıra, veri ambarı, CRM (Customer Relation Management), veri entegrasyonları ve benzeri yan uygulama fonksiyonlarını kullanmaktadır. Bu yan fonksiyonlar da JPA standartları dışında, veritabanı SQL'leri, ETL'ler gibi yöntemleri kullanmaktadırlar. Bu nedenle 4. yöntem ile çalışmada yaşanacak ileri tarihli değişikliklerin yansımaması olumsuzluğu, bu yöntemi de kullanılabilir olmaktan çıkarmaktadır. Bu gibi durumlardan dolayı en uygun yöntem olarak 3. Yöntem belirlenmiş olup, tüm değişiklikler ayrı tablolarda tutulmuş, ileri tarihli değişiklikler de ayrı tablolarda tutulacak ve ileri tarihteki değişikliklerin işlenmesi oluşturulan joblar vasıtasıyla yapılmıştır. Yöntemlerin avantaj ve dezavantajları ile ilgili detaylı bilgi Tablo 4.2.'de incelenebilir.

Tablo 4.2 Arşiv bilgilerinin veritabanında tutulma yöntem karşılaştırma tablosu

Yöntem	Avantajları	Dezavantajları
1; Arşiv bilgilerinin asıl tablolarda tutulması	✓ Arşiv verilerini kayıt ve sorgu işlemleri hızlı olur, ✓ Arşiv tabloları için ayrı joinler kurulmayacağından, sorgulama kolaylığı ve hızı sağlar, ✓ Farklı tablo isimleri aranmayacağından daha kolay işlem yapılabilir.	✓ Asıl tablolar zamanla gereğinden fazla büyüyeceğinden kayıt ve sorgulamalarda yavaşlık oluşur, ✓ Varolan diğer sistem sorgulamaları ek alanlar düşünülmediğinden olumsuz etkilenebilir, ✓ İş mantığı içine arşiv bilgileri fazladan eklenmiş olur, ✓ Arşivlemenin kaldırılması durumunda tablolara müdahele gerektirir.
2; Arşiv bilgilerinin farklı tablolarda tutulması, ileri tarih bilgisini nasıl tabloda tutulması	✓ İleri tarihli işlemlerin zamanı geldiğinde işlem kolaylığı olur, ✓ İleri tarihli işlemler için ek bilgilere, tablolara ihtiyaç kalmaz,	✓ Arşiv veri kayıt ve sorgu işlemleri çok hızlı olmayabilir, ✓ Asıl tablolara ileri tarihli de olsa arşiv bilgisi eklenmiş olduğundan, iş bilgisi bütünlüğü bozulmuş olur, ✓ Arşivlemenin kaldırılması durumunda tablolara müdahele gerektirir.
3; Arşiv bilgilerinin ve ileri tarih bilgisinin farklı tablolarda tutulması, ileri tarihli kayıtların gerçek yerine işlenmesi için Job (iş) oluşturulması	✓ İş mantığı ve değişiklik bilgilerimantığı birbirinden ayrıştırılmış olur, ✓ Değişiklik bilgileri ayrı olarak daha kolay görünür. ✓ Arşivleme kolaylıkla kaldırılabilir, çünkü asıl tablolara hiç dokunulmaz, ✓ İleri tarihli işlemler ayrı tabloda daha rahat görünür, ✓ Oluşturulan joblar ileri tarih zamanı geldiğinde kendiliğinden gerçek verilere işleme işini yapar	✓ Arşiv veri kayıt ve sorgu işlemleri çok hızlı olmayabilir, ✓ Java'da çalışacak jobların çalışmama riski olabilir.
4; Arşiv bilgilerinin ve ileri tarih bilgisinin farklı tablolarda tutulması, ileri tarihli kayıtların gerçek yerine işlenmesi için asıl bilgilerin sorgulanmasının dinlenmesi	✓ İş mantığı ve değişiklik bilgilerimantığı birbirinden ayrıştırılmış olur, ✓ Değişiklik bilgileri ayrı olarak daha kolay görünür. ✓ Arşivleme kolaylıkla kaldırılabilir, çünkü asıl tablolara hiç dokunulmaz, ✓ İleri tarihli işlemler ayrı tabloda daha rahat görünür, ✓ İleri tarihli değişikliklerin, asıl bilgilerin dinlenmesi ile gerçek verilere işlenmesi eksiksiz yapılır.	✓ Arşiv veri kayıt ve sorgu işlemleri çok hızlı olmayabilir, ✓ JPA standartları dışında uygulamaya erişimlerde, ileri tarihli değişikliklerin yapıldığı verilere JPA ile sorgu gelmediyse, yanlış bilgiler ulaşılır.

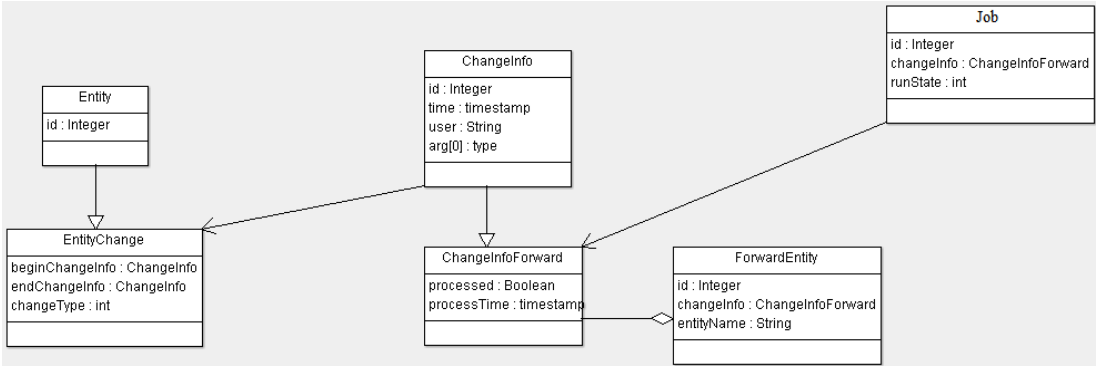
4.3.5. Değişiklik bilgilerinin tutulacağı sınıflar

Uygulamanın herbir entity'sine karşılık bir değişiklik sınıfı yer almıştır. Bu sınıfta entity üzerindeki tüm yeni kayıtlar, güncellemeler ve silmeler birer instance ta yeralmıştır. Uygulama üzerindeki herhangi bir varlık sınıfı @Entity sınıfı ile isimlendirilirse, değişiklik sisteminde bu sınıfa karşılık bir [EnityChange] sınıfı yer alır. [EntityChange] sınıfı [Enity] sınıfından türer, ek olarak üzerinde değişikliğin başlangıç / bitiş bilgileri ve değişiklik tipi bilgisi (ekleme, güncelleme, silme) yeralır. Uygulama sınıf kayıtları üzerinde yapılan her değişiklik bir değişiklik ana sınıfı üzerinde tutulur. Bu sınıf [ChangeInfo] üzerinde bir birincil anahtar [id], zaman bilgisi [time], kullanıcı bilgisi [user] ve uygulamanın ekleyebileceği diğer bilgiler [arg[0, 1, 2…]] dir.

İleri tarihli değişiklikleri daha düzgün takip edebilmek için [ChangeInfo] sınıfından türeyen bir ileri tarihli değişiklik anasınıfı oluşturulur. Bu sınıf [ChangeInfoForward] üzerinde ek olarak değişikliğin asıl yerine işlenip işlenmediği bilgisi [processed] ve işlendiyse işlenme zamanı bilgisi [processTime] tutar.

İleri tarihli değişiklikleri zamanı geldiğinde asıl yerine işleyecek bir java iş sınıfı oluşturulur. Bu sınıf [Job] üzerinde bir birincil anahtar [id], değişiklik anasınıf bağlantı bilgisi [changeInfo] ve işin çalışma durum bilgisi [runState] yer alır. [ChangeInfoForward] sınıfına her kayıt atılması sırasında [Job] sınıfına bir kayıt oluşturulur.

İleri tarihli değişiklikleri işleyecek olan [Job] sınıfının işleme zamanı geldiğinde, hangi verileri işlemesi gerektiğini kolayca bulabilmesi için ileri tarihli değişikliğin etkilendiği sınıfların bilgisini tutacak bir sınıf oluşturulur. Bu sınıf [ForwardEntity] üzerinde bir birincil anahtar [id], değişiklik anasınıf bağlantı bilgisi [changeInfo] ve değişikliğin yapılacağı sınıf adı bilgisi [entityName] alanlarını kapsar. Şekil 4.13.'de değişiklik bilgilerinin tutulacağı sınıf diyagramı gösterilmiştir.



Şekil 4.13. Değişiklik bilgilerinin tutulacağı sınıfların diyagramı.

4.3.6. Başlangıçta oluşacak konfigurasyon sınıfları

Uygulama sunucusu ayağa kalkarken, değişiklik işlemlerini yapacak temel sınıflar oluşturulur ve uygulamanın çalışması boyunca hayatlarına devam ederler. Bu sınıflar genellikle değişiklik işlemlerinin standartlarını oluşturan konfigurasyon sınıflarıdır.

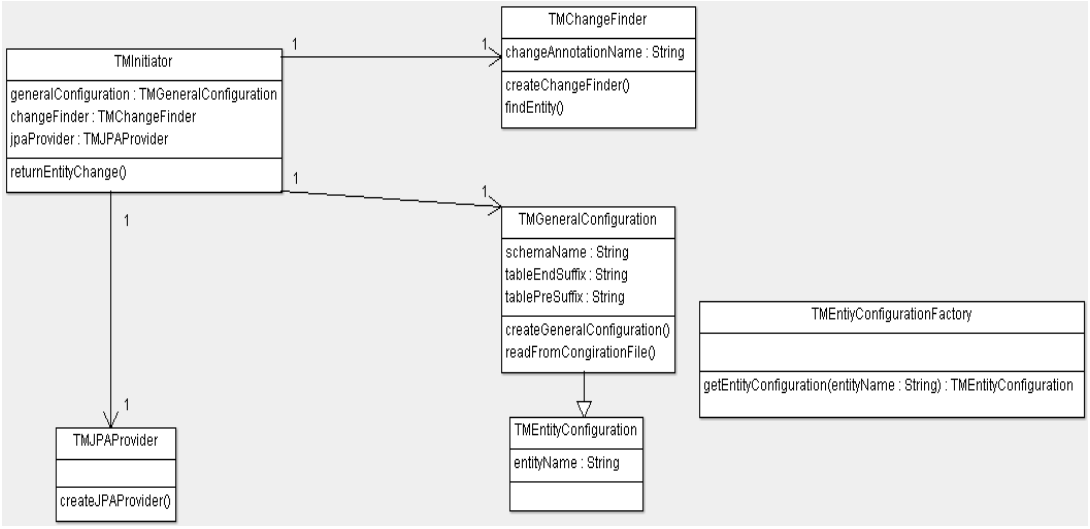
Projenin ana amaçlarından biri, varolan uygulamanın en az değişiklik ile projeyi kullanmasıdır. Bu nedenle uygulama herhangi bir konfigurasyon değişikliği istemez ise başlangıçta varsayılan proje konfigurasyonları yüklenir. Varsayılan konfigurasyonlar [TMDefaultConfiguration.properties] dosyasında tutulur. Uygulamanın istediği konfigurasyon güncellemesi olursa bunlar varsayılanları ezer.

Uygulama konfigurasyonları genel olarak da değiştirebileceği gibi entity bazında da değiştirebilir. Genel konfigurasyon değişiklikleri [persistence.xml] dosyasında yapılabilir. Dosya içinde özel property ve value'lar verilerek yaklaşımın genel konfigurasyonları değiştirilebir. Entity bazında konfigurasyon değişiklikleri ise java annotationlar kullanılarak yapılabilecek tüm sınıfları ayağa kaldıracak bir başlangıç sınıfı [TMInitiator] olur. [TMInitiator] sınıfı uygulama sunucusu ayağa kalkınca oluşturulur ve sunucu kapanana kadar yaşar.

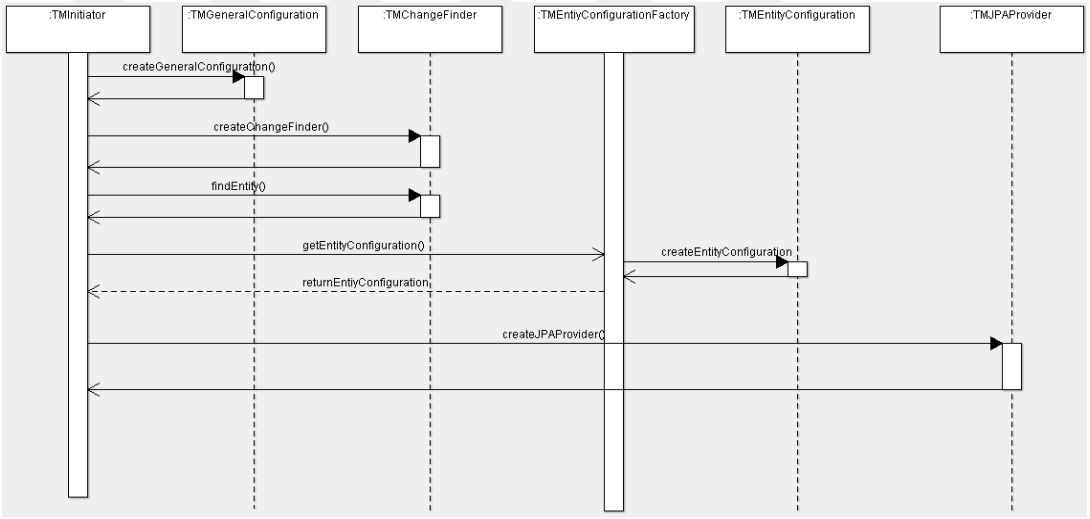
[TMInitiator] önce konfigurasyon bilgilerini tutan sınıfı oluşturur [TMGeneralConfiguration]. [TMGeneralConfiguration] sınıfı varsayılan değerleri [TMDefaultConfiguration.properties] dosyasından okur. Daha sonra eğer oluşturulmuşsa [TMChange.xml] dosyasını okur ve bu dosyadaki değerler varsayılanları ezer.

[TMInitiator] daha sonra tüm entiy sınıflarını tarayacak olan sınıfı oluşturur [TMChangeFinder]. [TMChangeFinder] sınıfı JPA standartlarına göre oluşturulmuş entityleri tarar. Eğer bir entity üzerinde @Change annotation'ı bulursa bu entity'e ait konfigurasyon sınıfını oluşturur [TMEntityConfiguration].

[TMEntityConfiguration] sınıfı [TMGeneralConfiguration] sınıfından türeyen bir sınıftır. Üzerinde ek olarak hangi entity'e ait olduğu bilgisini tutar. Entity'nin değişiklik sınıfının [EnityChange] temel bilgilerinin ve işleyişinin nasıl olacağı bilgisini tutar. [TMInitiator] daha sonra uygulamanın hangi JPA sağlayıcısını kullandığını [persistence.xml] dosyasını okuyarak bulur ve bilgilerini bir sınıfta tutar [TMJPAProvider].Şekil 4.14'de başlangıç konfigurasyon sınıflarının sınıf diyagramı ve Şekil 4.15.'de başlangıç konfigurasyon sıra diyagramı gözlemlenebilir.



Şekil 4.14. Başlangıç konfigurasyon sınıf diyagramı.



Şekil 4.15. Başlangıç konfigurasyon sıra diyagramı.

4.3.7. Başlangıçta oluşacak değişiklik sınıfları ve tablo oluşturma

Uygulama sunucusu ayağa kalkarken, konfigurasyon dosyaları oluşturulduktan sonra, değişiklik sınıflarının oluşturulmasına ve veritabanında değişiklik tablolarının oluşturulması işlemlerine başlanır.

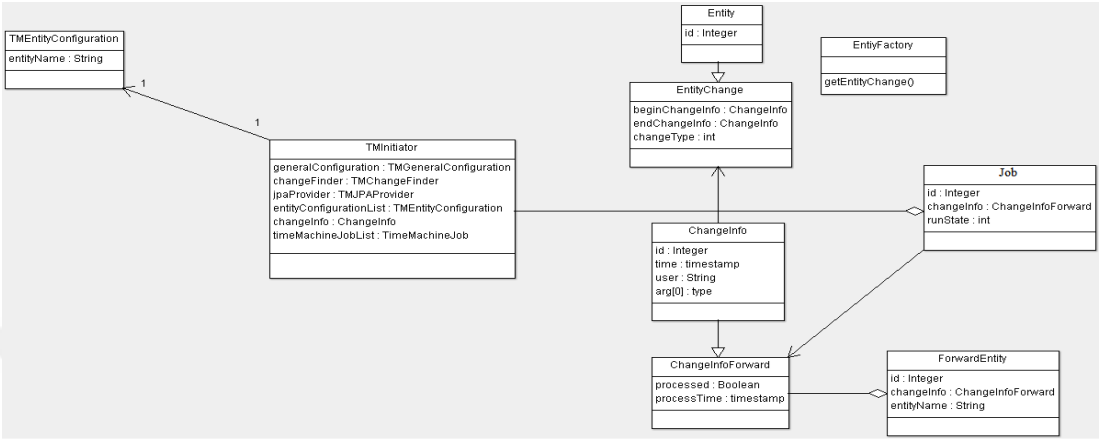
Öncelikli olarak değişiklik temel bilgilerinin tutulduğu ana sınıf ve tablolar oluşturulur Bunun için [TMInitiator] sınıfı değişiklik temel sınıfı [ChangeInfo] sınıfını oluşturur ve veritabanına kaydeder. Veritabanına kaydetme işlemi sırasında

konfigurasyon dosyalarından tablo ismi ve uzantısı, hangi şemada oluşturulacağı bilgisi, kolon ad ve tip bilgileri bilgileri alınır. [ChangeInfo] sınıfında her yapılan değişikliğin zaman, kullanıcı ve uygulama özel bilgileri tutulur. Sınıf üzerindeki birincil anahtar [id] tüm değişiklik detay sınıflarında referans olarak kullanılır.

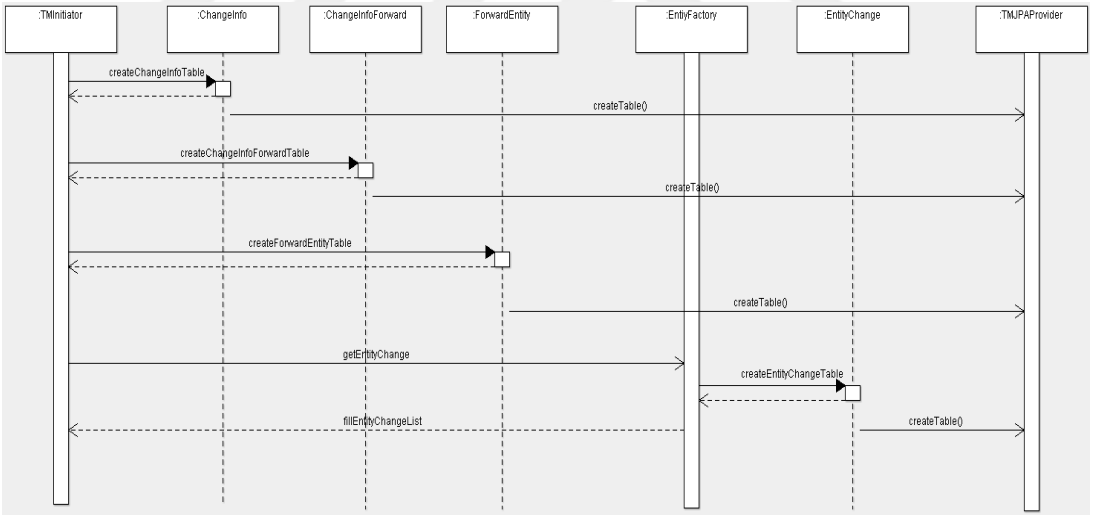
[TMInitiator] sınıfı daha sonra ileri tarihli değişiklik sınıfını [ChangeInfoForward] oluşturur ve veritabanına kaydeder. [ChangeInfoForward] sınıfı temel olarak [ChangeInfo] sınıfında türetilmiştir ve sadece ileri tarihli değişiklikler için bilgi tutar. Bu sınıfta ek olarak ileri tarihli bilginin zamanı geldiğinde asıl yerine işlenip işlenmediği bilgisi [processed] ve işlendiyse işlenme zamanı [processTime] tutulur. [TMInitiator] sınıfı ileri tarihli değişikliklerin hangi entity'leri etkilediği bilgisini tutmak için [ForwardEntity] sınıfını oluşturur ve veritabanına kaydeder. [ForwardEntity] sınıfında [Changeınfo] sınıfına bir referans bilgisi [changeInfo] ve etkilenen entity ismi [entityName] tutulur. [TMInitiator] tüm bu işlemlerde veritabanına kayıt esnasında, [TMJPAProvider] sınıfından yardım alarak tablo oluşturulması işlemlerini uygulamanın kullandığı JPA sağlayıcısını yaptırır.

[TMInitiator] sınıfı ileri tarihli değişikliklerin veritabanında asıl yerlerine işlenmesini yapacak olan [TMJob] sınıfını takip etmek için bir liste oluşturur. İleri tarihli her değişikliğin oluşturulmasına denk düşecek [TMJob] sınıfı oluşturulacak ve bu listeye eklenecek değişiklik temel sınıf ve tablolarının oluşturulmasından sonra sıra değişiklik kayıtlarının tutulacağı entity değişiklik sınıflarına gelir. [TMInitiator] sınıfı değişiklik kaydı takip edilecek her @Entity sınıfı için [EntiyChange] sınıfı oluşturur ve veritabanına kaydeder. Hangi @Entity için hangi [EntityChange] sınıfının oluşturulması işi için [EntityFactory] sınıfı görev alır. [EntiyChange] sınıfı asıl sınıf olan @Entity'den türer ve üzerinde ek olarak şu bilgileri tutar; değişiklik başlangıç kaydı referansı [beginChangeInfo] , değişikliğin ne zamana kadar devam ettiği kayıt referansı [endChangeInfo] (eğer değişiklik hala geçerli ise bu alanda en son zaman bilgisinin tutulacağı [ChangeInfo] kaydına referans edecek), değişiklik tipi (kayıt ekleme, kayıt güncelleme, kayıt silme). Tüm bu sınıf oluşturma ve veritabanı tablo oluşturma işlemleri sonrası, uygulama sunucusunun ayağa kalkması ile birlikte uygulamanın değişiklik işlemleri de hazır hale gelir. Şekil 4.16 ve Şekil

4.17.'de sırasıyla uygulamanın başangıcında oluşacak tablo sınıf diyagramı ve sıra diyagramı yer almaktadır.



Şekil 4.16. Başlangıç tablo oluşturma sınıf diyagramı.



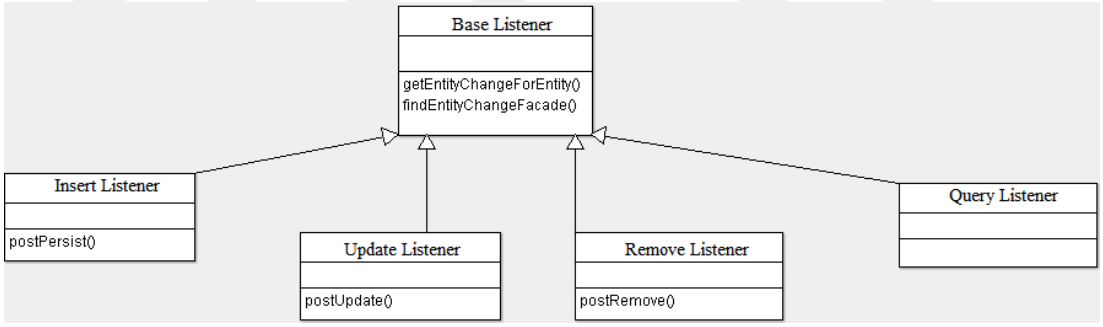
Şekil 4.17. Başlangıç tablo oluşturma sıra diyagramı.

4.3.8. Kayıt değişiklik dinleme sınıfları

Uygulama yaşam döngüsü içinde oluşacak veri değişikliklerini dinleyecek sınıflar oluşturulur ve dinleme işlemlerine başlatılır. Kayıt değişiklikleri için bir temel dinleyici sınıfı oluşturulur [BaseListener]. Bu temel sınıfta dinleyicilerin ortak

kullanacağı methotlar bulunur. Kayıt ekleme, güncelleme, silme ve kayıt sorgulama işlemleri için ayrı ayrı dinleyici sınıfları oluşturur.

Bu sınıflar [InsertListener], [UpdateListener], [DeleteListener] ve [QueryListener] sınıflarıdır ve [BaseListener] sınıfından türerler. Bu sınıfların kayıt değişikliklerinde çalışabilmesi için JPA2.0 standartında belirtilen orm.xml dosyasında . Bu dosyada <entity-listeners> bölümünde <post-persist>, <post-update> ve <post-remove> bölümlerinde ilgili dinleyici sınıfları tanımlanır. Bu tanımlama ile birlikte tüm @Entity sınıflarında kayıt işlemleri ile dinleyici sınıflar görev alırlar ve gerekli işlemlerini yaparlar. @Entity sınıflarında @Change anotation'ı da bulunuyorsa dinleyici sınıflar görev alırlar, @Change anotation'ı bulunmayan sınıflar için sınıfın başına @ExcludeDefaultListeners konularak dinleyici sınıfların bu @Entity'lerde dinleme yapması engellenir. Tüm bu sınıfların oluşturulması ve orm.xml dosyasında tanımlanması sonrası, uygulama sunucusunun ayağa kalkması ile birlikte uygulama değişiklik işlemleri de hazır hale gelir. Şekil 4.18.'de kayıt dinleyici sınıfların sınıf diyagramı yer almaktadır.



Şekil 4.18. Kayıt değişiklik dinleyici sınıf diyagramı.

4.3.9. Kayıt ekleme işlemi

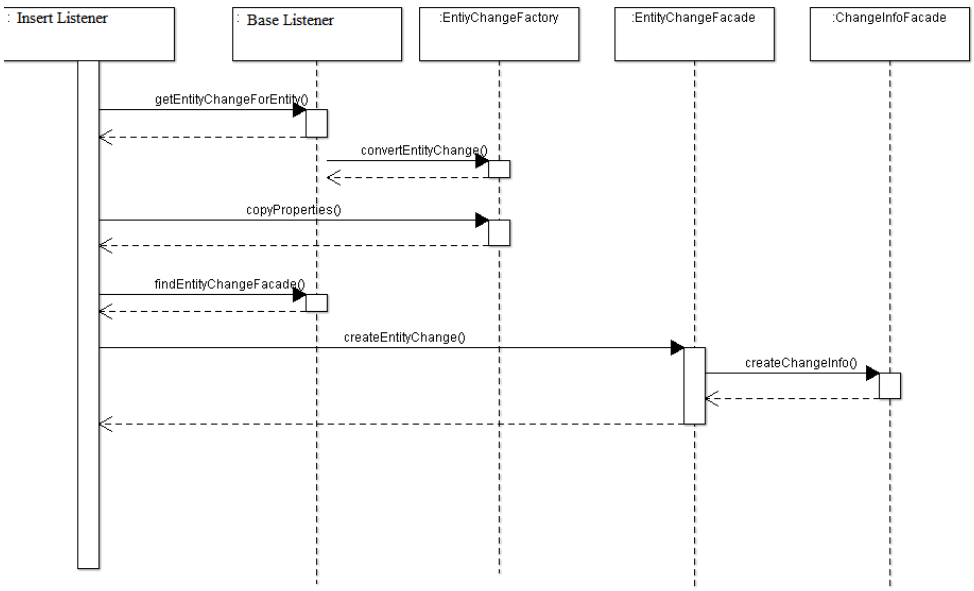
Uygulama normal yaşam döngüsüne devam ederken, herhangi bir durumda veritabanına bir kayıt ekleme işlemi gerçekleştiğinde, yeni yaklaşımdaki uygulama hemen devreye girip kaydın yaptığı eklemeyi de veritabanına kaydeder. Burada oluşturulan dinleyici sınıflar görev alır.

Yeni kayıt ekleme işlemi sonrası [InsertListener] sınıfı devreye girer. Uygulama JPA standartlarına uygun bir kayıt ekleme işlemi yaptığında, kaydın veritabanına kaydedilmesinden hemen sonra, [InsertListener] postInsert() methodu devreye girer, bu methoda standartlar gereği kayıt işlemi gerçekleştirilen @Entity gelir. Daha sonra bu @Entity'e karşılık gelecek olan [EntityChange] sınıfının ismini [EntityChangeFactory] sınıfı yardımıyla bulur.

[InsertListener] değişiklik işlemi temel bilgilerini [ChangeInfo] sınıfına kaydeder ve bir kayıt referans numarası alır. Bu referans numarası ile [EntityChange] sınıfı oluşturulacaktır;

[EntityChange] sınıfındaki bir önceki son değişiklik kaydı bulunur ve yeni oluşturulan [ChangeInfo] sınıfına ait referans bilgisi [endChangeInfo] alanında güncellenir ve veritabanına kaydedilir, bu şekilde bir önceki değişikliğin ne zamana kadar geçerli olduğu bilgisi de kaydedilmiş olur.

[EntityChange] sınıfı oluşturulur. Yapılan kayıt ekleme işlemi ile ilgili sınıf değerleri alınır ve birer kopyası [entityChange] sınıfına kopyalanır. [ChangeInfo] sınıfına ait referans bilgisi [beginChangeInfo] alanına girişi yapılır, en son [changeType] alanına 'Ekleme' bilgisi girişi yapıldıktan sonra veritabanına kayıt işlemi yapılır.

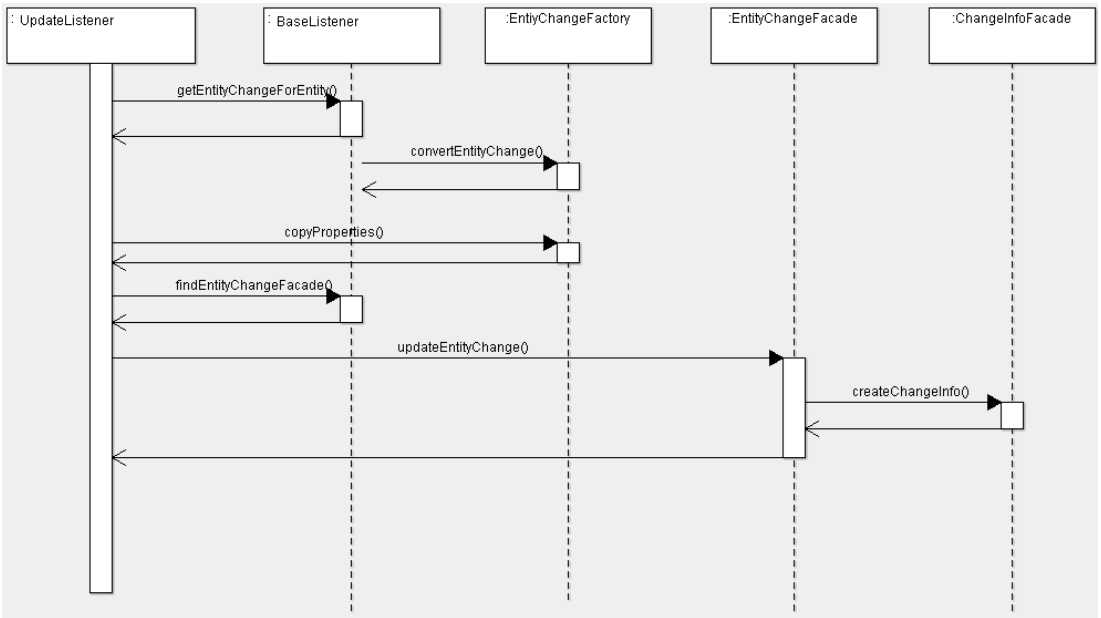


Şekil 4.19. Yeni kayıt ekleme işlemi sıra diyagramı.

4.3.10. Kayıt güncelleme işlemi

Uygulama yaşam döngüsünde veritabanında bir kayıt güncelleme gerçekleştiğinde, uygulama devreye girip kaydın yaptığı güncellemeyi de veritabanına kaydeder. Kayıt güncelleme işlemi sonrası [UpdateListener] sınıfı devreye girer. Uygulama JPA standartlarına uygun bir kayıt güncelleme işlemi yaptığında, kaydın veritabanına kaydedilmesnden hemen sonra, [UpdateListener] postUpdate() methodu devreye girer, bu methoda standartlar gereği kayıt işlemi gerçekleştirilen @Entity gelir. Daha sonra bu @Entity'e karşılık gelecek olan [EntityChange] sınıfının ismini [EntityChangeFactory] sınıfı yardımıyla bulur.

[UpdateListener] değişiklik işlemi temel bilgilerini [ChangeInfo] sınıfına kaydeder ve bir kayıt referans numarası alır. Bu referans numarası ile [EntityChange] sınıfı oluşturulur. [EntityChange] sınıfındaki bir önceki kaydın [endChangeInfo] alanı güncellenir ve veritabanına kaydedilir.Yapılan değişiklikle ilgili sınıf değerleri [EntityChange] sınıfı oluşturulup girişi yapılır, [ChangeInfo] sınıfına ait referans bilgisinin [beginChangeInfo] alanına girişi yapılır, [changeType] alanına 'Güncelleme' bilgisi girişi yapıldıktan sonra veritabanına kayıt işlemi yapılır. Şekil 4.20.'de kayıt güncelleme işleminin sıra diagramı yer almaktadır.

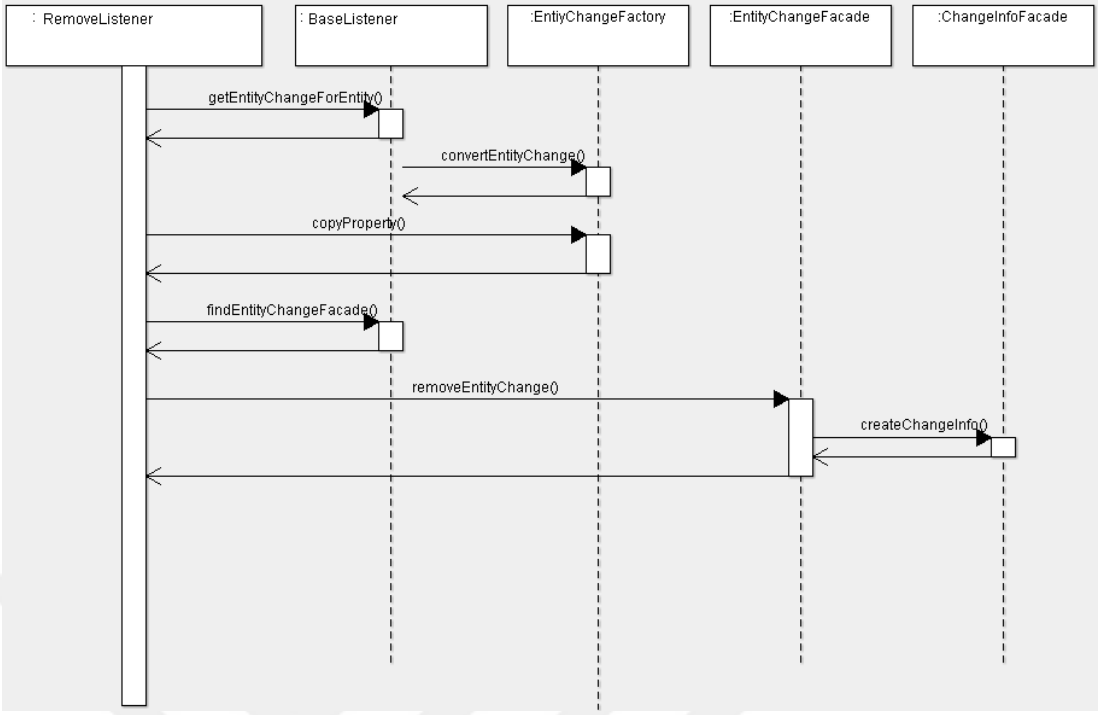


Şekil 4.20. Kayıt güncelleme işlemi sıra diyagramı.

4.3.11. Kayıt silme işlemi

Uygulama yaşam döngüsünde veritabanında bir kayıt silme gerçekleştiğinde, uygulama devreye girip silinen kayıt ile ilgili bilgileri de veritabanına kaydeder. Kayıt silme işlemi sonrası [RemoveListener] sınıfı devreye girer. Uygulama JPA standartlarına uygun bir kayıt silme işlemi yaptığında, kaydın veritabanından silinmesinden hemen sonra, [RemoveListener] postRemove() methodu devreye girer, bu methoda standartlar gereği kayıt işlemi gerçekleştirilen @Entity gelir. Daha sonra bu @Entity'e karşılık gelecek olan [EntityChange] sınıfının ismi [EntityChangeFactory] sınıfı yardımıyla bulur.

[RemoveListener] silme işlemi temel bilgilerini [ChangeInfo] sınıfına kaydeder ve bir kayıt referans numarası alır. Bu referans numarası ile [EntityChange] sınıfı oluşturulur. [EntityChange] sınıfındaki bir önceki kaydın [endChangeInfo] alanı güncellenir ve veritabanına kaydedilir. Yapılan silme işlemi için [EntityChange] sınıfı oluşturulur, tüm verileri boş olarak girişi yapılır, [ChangeInfo] sınıfına ait referans bilgisinin [beginChangeInfo] alanına girişi yapılır, [changeType] alanına 'Silme' bilgisi girişi yapıldıktan sonra veritabanına kayıt işlemi yapılır.Şekil 4.21.'de kayıt slime işleminin sıra diyagramı görülmektedir.



Şekil 4.21. Kayıt silme işlemi sıra diyagramı.

4.3.12. İleri tarihli kayıt değişiklik işlemi

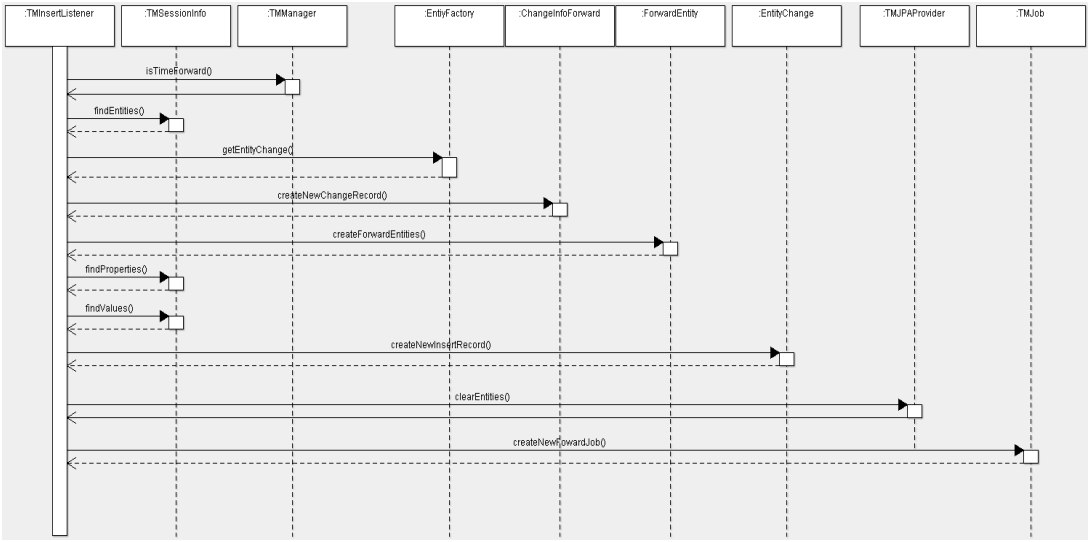
Uygulama ileri tarihli bir kayıt değişiklik işlemi yapmak istediğinde öncelikle yeni geliştirilen uygulamaya ileri tarih zaman bilgisini geçmek zorundadır. İleri tarih bilgisi verildikten sonra uygulama normal işlemlerine devam eder fakat bundan sonra yapılacak her türlü kayıt işlemi ileri tarihte kaydedilmek üzere yeni uygulamanın değişiklik tablolarına kaydedilir, normal uygulama tablolarına herhangi bir kayıt yapılmaz.

[TMManager] sınıfı uygulamanın yeni uygulamaya bildireceği parametreleri tutar. Uygulama kullanıcı bilgisini [setUser(String user)] metodu ile, ileri tarih zaman bilgisini [setTime(timestamp time)] metodu ile ve istenilen uygulamaya özel bilgileri de [addChangeParam(String name, Object value)] metodu ile bildirebilir. Eğer bu bilgilerden herhangi biri bildirilmez ise, veritabanında karşılık gelen alanlar boş bırakılır. Uygulama denetim kayıtlarını tutmayı sağlayacak yaklaşımımızdaki uygulamaya ileri tarihli bir zaman bilgisi verdikten sonra, bir yeni kayıt ekleme işlemi sonrası [InsertListener] sınıfı devreye girer. Kaydın veritabanına kaydedilmesinden hemen sonra, [InsertListener] ileri tarihli bir zaman olup

olmadığını [TMManager] sınıfından öğrenir. Eğer ileri tarihli kayıt ise değişiklik işlemi temel bilgilerini [ChangeInfoForward] sınıfına kaydeder ve hangi entitylerin etkilendiği bilgisini de [ForwardEntity] sınıfına kaydeder. [ChangeInfoForward] sınıfına kayıt sonrası bir referans numarası alır. Bu referans numarası ile [EntityChange] sınıfı oluşturulur. [EntityChange] sınıfındaki bir önceki son değişiklik kaydında şimdilik bir güncelleme yapılmaz, çünkü bir önceki kayıt gerçek zamanda hala geçerli kayıttır.

[EntityChange] sınıfı oluşturulur. Yapılan kayıt ekleme işlemi ile ilgili sınıf değerleri alınır ve birer kopyası [entityChange] sınıfına kopyalanır. [ChangeInfoForward] sınıfına ait referans bilgisi [beginChangeInfo] alanına girişi yapılır, en son [changeType] alanına 'Ekleme' bilgisi girişi yapılır. [ForwardEntity] sınıfında da aynı referans bilgisinin [changeInfo] alanına girişi yapılır, hangi entityler etkileniryorsa isimlerinin [entityName] alanına girişi yapılır. Tüm kayıtlardan sonra en son veritabanına kayıt işlemleri yapılır. Bu arada [TMJPAProvider] sınıfı yardımı ile gerçekte veritabanına kaydedilecek işlemler silinir.

Tüm bu işlemler sona erdiğinde ileri tarihli kayıtlar yeni veri denetim sistemine alınmış ve gerçekte uygulama tablolarına bir veri kaydı yapılmamış olur. Son olarak ileri tarih zamanı geldiğinde veri denetim uygulamasındaki ileri tarihli değişiklikleri gerçek uygulamaya yansıtacak iş kalmıştır. Veri denetim uygulaması en son [TMJob] sınıfına bir iş oluşturur. Bu iş sınıfında [changeInfo] alanında ileri tarihli değişiklik bilgileri referans olarak kaydedilir. İleri tarih zamanı geldiğinde [TMJob] daki bu iş çalışacak ve değişiklikleri asıl uygulama veritabanına kayıt işlemlerini gerçekleştirecektir. Şekil 4.22.'de ileri tarihli yeni kayıt ekleme işlemi nin sıra diyagramı görülmektedir.



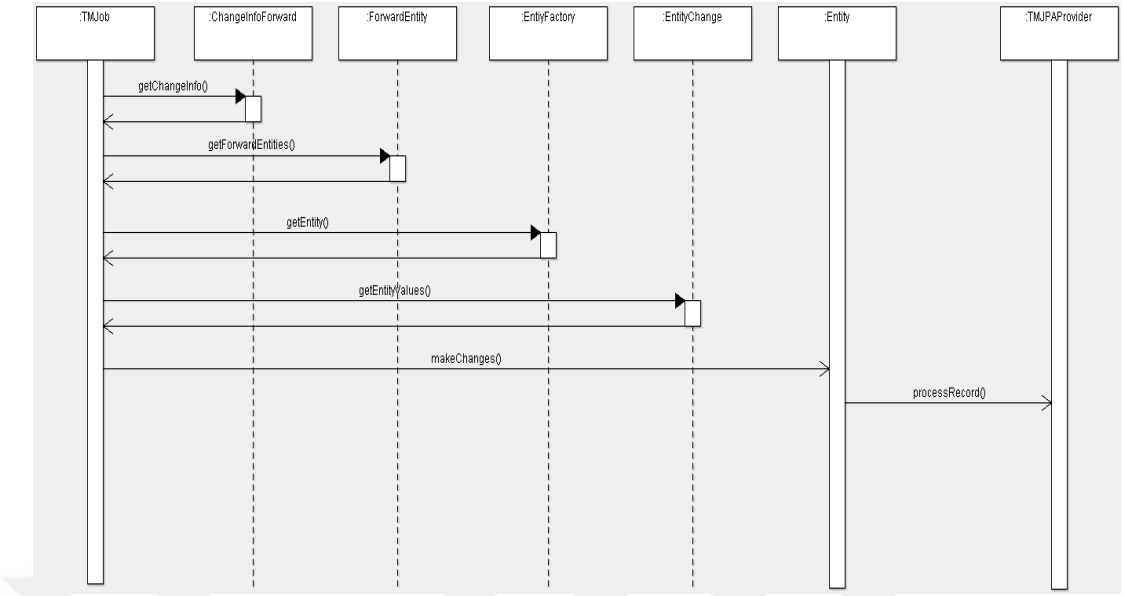
Şekil 4.22. İleri tarihli yeni kayıt ekleme işlemi sıra diyagramı.

4.3.13. İleri tarihli kayıtların uygulamaya işlenmesi

Veri denetim uygulaması ileri tarihli herbir değişiklik işlemi için zamanı geldiğinde çalışmak üzere [TMJob] sınıfından bir iş oluşturmuş olacaktır. [TMJob] sınıfı üzerinde bulunan zaman geldiğinde veri denetim uygulaması na kaydedilmiş değişikleri alıp gerçek uygulama veritabanına kaydetme işlemini yapacak. [TMJob] değişiklik zamanı geldiğinde öncelikle üzerinde buluna [changeInfo] referansı ile değişiklik temel bilgilerini bulur, bu referans ile [ForwardEntity] sınıfında hangi entity'lerin değişiklikten etkilendiklerini bulur.

Sırası ile değişiklikten etkilenen entity isimlerinden [EntityFactory] sınıfı yardımı ile gerçek @Entity sınıflarını oluşturur. Karşılık gelen [EntityChange] sınıflarından entity özellik ve değerlerini @Entity sınıflarına kopyalar. Tüm işlemler bittikten sonra uygulamanın gerçek veritabanına kaydetme işlemlerini [TMJPAProvider] sınıfı yardımı ile gerçekleştirir. Değişiklik işlemleri kayıt ekleme, güncelleme ve silme işlemlerinden biri olabilir.

[TMJob] işlemleri bitirince tüm değişiklik sınıflarını bırakır [ChangeInfoForward] sınıfı üzerindeki [processed] alanına işlenmiş ve [processTime] alanına da işlenme zamanı bilgisini kaydeder. Şekil 4.23.'de ileri tarihli kayıtların uygulamaya işlenmesi ile ilgili sıra diyagramı yer almaktadır.



Şekil 4.23. İleri tarihli kayıtların uygulamaya işlenmesi sıra diyagramı.

BÖLÜM 5. SONUÇ

Bu çalışma uygulama katmanında, veri denetimini yapan ve platform bağımsız olarak kolaylıkla java mimarisi kullanan tüm sistemlerde veritabanı yapısı bağımsız çalışabilen ve verilerin belli bir zamanda ki durumunu gözlemlemeyi sağlayan, ileriye dönükte veri değişimlerinin otomatik yapılabileceği yeni yaklaşım dan oluşmaktadır.

Günümüzde kullanılan yapılar incelendiğinde veritabanına bağlı çözümler (Veritabanı tetikliyicileri, Change Data Capture(CDC), Servis komisyoncusu(Service Broker)) yada uygulama katmanında sadece belirli uygulamalara bağlı çözümler olduğu gözlemlenmiştir. Çözümlerin kendi aralarında fonksiyonalite, veritabanısistemlerine getirdiği yük, çözümlerin yeterliligi, bakımının kolay olması ve moduler yapıda olması kriterlerine göre analiz çalışması da nicel olarak yapılmıştır. Bu çalışmada ise JPA standartlarını kullanan tüm uygulamalarda kullanılabilir bir yapı inşa edilmiştir ki şu ana kadar böyle bir çalışma bulunmamaktadır.

Zaman içinde değişime uğrayan ve ileride uğrayacak olan verileri saklamak, geriye ve ileriye dönük sorgulama imkanını platform bağımsız sunması , çalışmanın önemli artısıdır. Şimdiye kadar benzer durumlarda hep geçmişe dönük verileri saklama üzerine çalışılmıştır. İleri dönük olarak yapılması planlanan veri değişikliklerini de saklama ve zamanı geldiğinde gerçekleştirme işlemleri ile çalışma kendini benzerlerinden bir adım öne çıkarmaktadır.

Çalışmanın asıl hedeflerinden biri de ara yapı oluşturarak var olan uygulamalara kolaylıkla entegre edilebilmesidir. Kullanıldığı uygulamalarda en az kodlama ve değişim ile çalışabilir bir yapı oluşturmuştur.

Şu an java platformunda çalışan büyük uygulamaların veritabanlarında yaşanan geçmiş veri kayıpları önlenerek kurumlara ekonomik değer sağlayacaktır. Ayrıca ileriye dönük veri değişimi özelliği ile kurumların ileride kendilerini daha iyi görebilmesini sağlayacak ve önceden hareket edebilmelerini sağlayacaktır.

KAYNAKLAR

[1] Olumuyiwa O, Carl Dudley, Critical Assesment of Auditing Contributions to Efficient Security in Database, CS & IT-CSCP 2015.

[2] Hernando Bedoya, Daniel Lema, Cintia Marques,Vijay Marwaha, Stored Procedures and Triggers on DB2 Universal Database for iSeries, 0-7384-2398-X,IBM Redbooks, 2001.

[3] Robert Pearl, Healthy SQL : A Comprehensive Guide to Healthy SQL Server Performance, 978-1-4302-6773-7, Apress, 2015.

[4] About Change Data Capture (SQL Server), http://technet.microsoft.com/en-us/library/cc645937.aspx, Erişim Tarihi: 16.04.2016.

[5] About Change Data Capture (SQL Server), http://technet.microsoft.com/en-us/library/cc645235.aspx, Erişim Tarihi: 14.04.2016.

[6] Gabriele Giuseppini, Microsoft log parser toolkit, Rockland, MA, Syngress Publishing, c2004, pp.1-2.

[7] SQL Log Rescue - Undo for SQL Server, http://www.red-gate.com/products/dba/sql-log-rescue, Erişim Tarihi: 18.04.2016.

[8] Mark Horninger, The Real MCTS SQL Server 2008 Exam 70-433 Prep Kit, 978-1-59749-421-2, 2009.

[9] Ramnivas Laddad, AspectJ in Action, Manning Publications, 1-933988-05-3, 2009.

[10] Will Iverson, Hibernate: A J2EE™ Developer's Guide,Addison-Wesley Professional, 0-321-26819-9, 2004.

[11] Peng Wu, Application Research on a Persistent Technique Based on Hibernate, International Conference On Computer Design And Appliations (ICCDA 2010), 2010.

[12] HibernateTechnology, Adres: http://www.hibernate.org, Erişim Tarihi: 16.03. 2016.

[13] Windhouwer, Efficient database auditing and entity reversion, http://www.topicus.nl/ Efficient database auditing and entity reversion.pdf, ErişimTarihi: 16.03.2016.

[14] HibernateEnvers, http://www.jboss.org/envers, Erişim Tarihi: 12.04.2016.

[15] Audit(OpenJPA), http://openjpa.apache.org/builds/2.2.1/apache-openjpa/docs ref_guide_audit.html, Erişim Tarihi: 15.04.2016.

[16] History Policy (EclipseLink), http://wiki.eclipse.org/EclipseLink/ Examples/ JPA/History, Erişim Tarihi: 02.04.2016.

[17] EntityAuditBundle (EclipseLink), http://www.doctrine-project.org/, Erişim Tarihi: 10.04.2016.

[18] EntityVersionHistory(ObjectDB), http://www.objectdb.com/database/issue/ 25, Erişim Tarihi: 10.04.2016.

[19] R.M Menon, Expert Oracle JDBC Programming, 978-1-59059-407-0, Apress, 2005.

[20] Debu Panda, Reza Rahman, Derek Lane, EJB 3 in Action ,1-933988-34-7, Manning Publications, 2007.

[21] Mike Keith, Merrick Schincariol, Pro JPA 2:Mastering the Java™ Persistence API, 1-4302-1956-4, pp. 12-15, Apress, 2009.

[22] Won Kim , Frederick H. Lochovsky, Object-Oriented Concepts, Databases, AddisiOn-Wesley, 1989.

[23] Bruce Shriver, Peter Wegner, Research directions in object-oriented programming, MIT Press, Cambridge, 1987.

[24] OpenJPA Technology, http://openjpa.apache.org/, Erişim Tarihi: 16.04. 2015.

[25] DataNucleus , http://www.datanucleus.org/, Erişim Tarihi: 16.03 2016.

[26] ObjectDB, http://www.objectdb.com/, Erişim Tarihi: 10.04.2016.

[27] Boris Kogan, Sushi Jajodia, An Audit Model for Object-Oriented Databases, Computer Security Applications Conference, 1991.

ÖZGEÇMİŞ

Ibrahim Dokuzer, 14.11.1980 Sakarya'da doğdu. İlk, orta ve lise eğitimini Sakarya da tamamladı. 1998 yılında Mithatpaşa Lisesi'nden mezun oldu. 1998 yılında başladığı Yakın Doğu Üniversitesi Bilgisayar Mühendisliği Bölümü'nü 2002 yılında bitirdi. 2004 yılında Sakarya Üniversitesi Bilgisayar Mühendisliği Bölümü'nde yüksek lisans eğitimine başladı. Aynı yıl Sakarya Üniversitesi'nde araştırma görevlisi olarak çalışmaya başladı akabinde yüksek lisans eğitimine Sakarya Üniversitesi Bilgisayar Mühendisliği bölümünde devam etti. 2006 yılında Araştırma Görevliliği görevini bırakarak özel sektörde yazılım mühendisi olarak çalışmaya başladı. Halen Turkcell Şirketinde Kıdemli Yazılım mühendisi olarak çalışmaktadır. 2014 yılında girdiği Özyeğin Üniversitesindeki Bilgisayar Mühendisliği doktora programına devam etmektedir.