YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

MODEL-GÜDÜMLÜ MİMARİ KULLANILARAK BİR KONUM SUNUCUSU YAZILIMININ GELİŞTİRİLMESİ

Fırat YEŞİLÜRDÜ

FBE Bilgisayar Mühendisliği Ana Bilim Dalı
Bilgisayar Mühendisliği Programında
Hazırlanan

YÜKSEK LİSANS TEZİ

Tez Danışmanı : Yrd. Doç. Dr. Banu DİRİ

İSTANBUL, 2006

İÇİNDEKİLER

Sayfa

KISALTMA LİSTESİ

CCM	CORBA Component Model
CIM	Computation Independent (Hesaplamadan Bağımsız) Model
CORBA	Common Object Request Broker Architecture
CWM	Common Warehouse Metamodel
DTF	Domain Task Force
EAI	Enterprise Application Integration
EDOC	Enterprise Distributed Object Computing
EJB	Enterprise Javabeans
IDE	Integrated Development Environment
IDL	Interface Definition Language
J2EE	Java 2 Enterprise Edition
MDA	Model-Driven Architecture
MOF	Meta Object Facility
MSISDN	Mobile Subscriber ISDN Number
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Bağımsız (Independent) Model
PSM	Platform Bağımlı (Specific) Model
SOAP	Simple Object Access Protocol
UML	Unified Modelling Language
XMI	XML Metadata Interchange

ŞEKİL LİSTESİ

ÇİZELGE LİSTESİ

ÖNSÖZ

Yazılım geliştirmede modelin önemi son yıllarda gittikçe artmaktadır. Model kullanımıyla, proje yönetimi kolaylaşmakta, proje takımındaki üyeler arasındaki iletişim güçlenmekte, dokümantasyon ve test süreçleri hızlanmaktadır. Ancak sonuçta modelin faydası, belirli bir platformda kod geliştirmeye yardımcı olmakla sınırlı kalmaktadır.

Model güdümlü mimari (MDA) ile, model, yazılım geliştirmenin temel öğesi kabul edilmektedir. Yazılım geliştirmede, platform değişkenliği yazılan kodun ömrünü kısaltırken, MDA ile geliştirilmiş platform-bağımsız modeller koddan çok daha kalıcı olacaklardır.

Bu tezin giriş bölümünde MDA ve MDA'yı oluşturan yapıtaşları hakkında genel bilgiler verilecektir.

Takip eden bölümlerde, MDA'nın ortaya çıkışı, MDA'yı hazırlayan faktörler irdelenerek anlatılacak, MDA'nın yapısı ve MDA'nın dayalı olduğu alt kavramlar incelenecektir.

Sekizinci bölümde ise, MDA'nın kullanıldığı bir uygulama geliştirme süreci ele alınacaktır. MDA ile geliştirilen, Konum Sunucusu ve bu sunucuyu kullanan örnek bir uygulamada elde edilen tecrübeler ve sonuçlar aşama aşama anlatılacaktır.

Sonuç kısmında ise, proje sürecinde elde edilen tecrübe ışığında, MDA'nın yazılım geliştirmenin bugününde ve geleceğinde neleri değiştirebileceği tartışılacaktır.

Bu tezin hazırlanmasında bana yardımlarını esirgemeyen tez danışmanım sayın Yrd.Doç.Dr.Banu Diri'ye ve bölüm başkanımız sayın Prof.Dr.Oya Kalıpsız'a teşekkür ederim.

ÖZET

Model-güdümlü mimari (MDA), modeli yazılım geliştirmenin temel öğesi kabul ederek, uygulamaların geliştirilmesi için yeni bir yaklaşım sunmaktadır. Tamamlanmış bir MDA uygulaması; eksiksiz bir platform-bağımsız UML modeli, uygulama geliştiricinin desteklemeye karar verdiği platformlara ait bir veya daha fazla platform-bağımlı model ve tamamlanmış gerçekleştirmelerden oluşmaktadır.

MDA, gerçekleştirme detayları ile iş fonksiyonlarını birbirinden ayırmaktadır. Böylece her yeni teknoloji ortaya çıktığında, uygulama veya sistemin işlevini ve davranışlarını tekrar modellemeye gerek kalmamakta, yeni ve farklı teknolojileri desteklemek kolaylaşmaktadır.

Bu çalışma, MDA ve ilgili kavramları açıklamak ve bunları örnek bir yazılım projesi üzerinde hayata geçirmek amacını taşımaktadır. Örnek proje olan Konum Sunucusu yazılımı geliştirilirken, günümüzün MDA modelleme araçlarının elverdiği ölçüde, kodun mümkün olduğu kadar çok kısmı, yaratılan platform bağımsız model üzerinden otomatik olarak oluşturulmuştur ve mümkün olduğu kadar az kısım elle kodlanmıştır.

Anahtar kelimeler: Model-güdümlü mimari, MDA, UML, MOF, yazılım modelleme, konum tabanlı sistemler.

ABSTRACT

Model-driven architecture (MDA) presents a new approach to application development by assuming ‘model’ the central element of software development. A complete MDA application consists of a complete platform-independent UML model, one or more platform-specific models and complete implementations that the application developer decided to support.

MDA separates the implementation details from the business functions. Thus, there is no need to model the functions and behavior of the application or system again whenever a new technology is introduced and it becomes easier to support new or different technologies.

This work has the aim to describe MDA and related concepts and use them on an example software project. While the example software project, Location Server is developed, as much as possible part of the code has been generated automatically from the platform-independent model as today’s MDA tools allow and minimum coding has been done manually.

Keywords: Model-driven architecture, MDA, UML, MOF, software modeling, location based systems.

1. GİRİŞ

Nesneye yönelik dağıtık sistemlerde, nesnelerin birbirleriyle haberleşmesini sağlayan platformlara *ara katman yazılımı* (middleware) denmektedir. İrili ufaklı 800 BT firmasından oluşan bir konsorsiyum olan Object Management Group'un (OMG) hazırladığı ara katman yazılımı platformu, Common Object Request Broker Architecture (CORBA), birlikte işlerlik için bir standart olarak kabul edilmektedir. CORBA, nesnelerin; donanım platformu, işletim sistemi ve programlama dili sınırlarını aşarak birbiriyle haberleşmelerini sağlayabilmektedir. Ancak, CORBA mevcut olan tek ara katman yazılımı platformu değildir. Her biri birbiriyle neredeyse aynı hizmetleri -kendi yöntemleriyle- sunan J2EE, .NET ve diğer başka platformlar da ortaya çıkmışlar ve şu anda geniş çapta kullanılmaktadırlar. Standartlaşmış, firma ve platform bağımsız bir ara katman yazılımı teknolojisi de, şirketlerin sahip oldukları heterojen donanım ve yazılım sistemlerinden istedikleri düzeyde yararlanmalarını sağlayamamıştır.

Günümüzün bilişim dünyasının entegrasyon ihtiyaçlarını karşılamak amacıyla OMG, MDA ile standardizasyon seviyesini uygulama gerçekleştirme seviyesinden, uygulama tasarlama seviyesine çıkarmaktadır.

MDA birlikte-işlerlik meselesine açık ve herhangi bir firmanın ürününden bağımsız bir yaklaşım getirmektedir. MDA yine OMG'nin tasarladığı;

- Bir çok yerde kullanılan ve bütün büyük firmalar tarafından desteklenen modelleme notasyonu olan Unified Modeling Language (UML);
- Standart modelleme ve dönüşüm yapılarını sağlayan Meta-Object Facility (MOF);
- OMG veri ambarı standardı Common Warehouse Meta-model (CWM) üzerine kurulmuştur.

Ayrıca, modelleri XML vasıtasıyla saklamayı ve dönüştürmeyi sağlayan XML Metadata Interchange (XMI) ve en çok kullanılan açık ara katman yazılımı standardı CORBA da bu mimari ile yakından ilgilidir.

MDA mimarisinde, bir uygulama için önce platform-bağımsız bir iş modeli hazırlanacak, sonra bu modelden platform-bağımsız bir tasarım modeli elde edilecektir. Daha sonra bu modelden seçilen bir platforma (CORBA, Java, .NET, XMI/XML ve ağ tabanlı platformlar) ait modeller oluşturulacak, ve bu modeller hayata geçirilecektir.

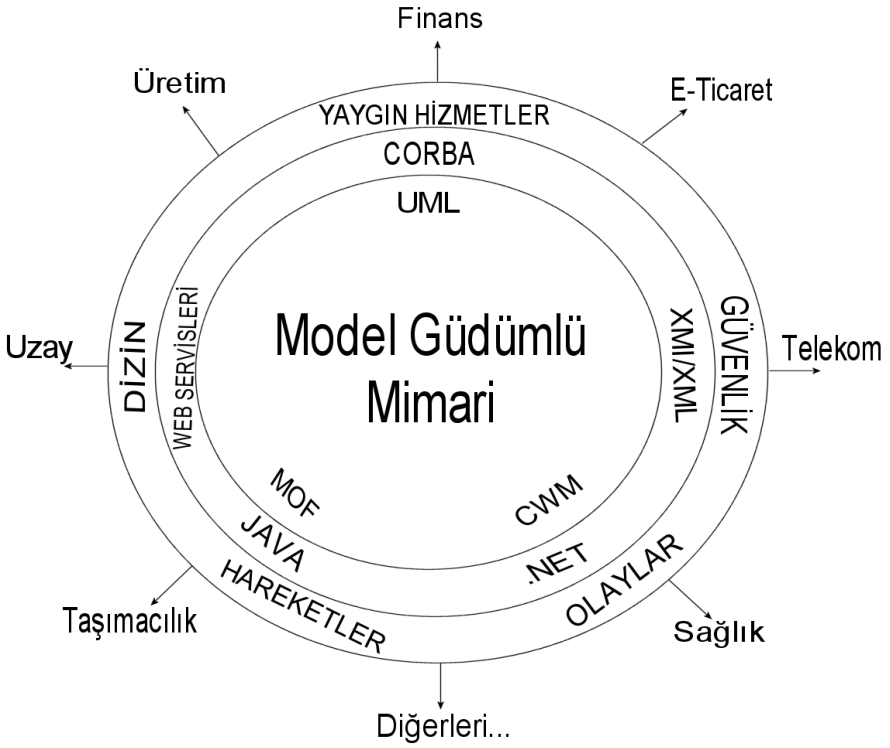
MDA, bir belirtimin (specification) arkasındaki temel iş mantığı ile, o belirtimin gerçekleştirildiği, belirli bir platformun kendine has özellikleriyle ilgili bilgileri birbirinden

ayırmaktadır. Bu, yeni teknolojileri kullanan ama test edilmiş ve kendini ispatlamış iş modelleri üzerine kurulmuş, yeni birlikte-işlerlik belirtimlerinin hızlı bir biçimde geliştirilmesine imkan vermektedir. MDA, bugünün yüksek oranda ağ üzerine kurulu, sürekli değişen sistemleri için aşağıdaki özellikleri sağlayan bir mimaridir [6]:

- **Taşınabilirlik**: Tekrar kullanımın artması, bugün ve gelecek için uygulama geliştirme ve yönetme masraf ve karmaşıklığının azaltılması.
- **Platformlar arası birlikte-işlerlik**: Birden fazla gerçekleştirme teknolojisi üzerine kurulmuş standartlar çerçevesindeki kesin metotlar ile, her gerçekleştirmenin aynı iş fonksiyonlarını yerine getirmesi.
- **Platform bağımsızlığı**: Daha ortaya çıkmamış platformlar dahil olarak farklı platformlara geçişte yaşanacak zaman kaybı, maliyet ve karmaşıklığın azaltılması.
- **Alan özellikleri**: İş sahalarına ait özel modellerle farklı platformlarda, o iş sahasına ait yeni gerçekleştirmelerin hızlı bir biçimde hayata geçirilmesi.
- **Verimlilik**: Geliştirici, tasarımcı ve sistem yöneticilerinin rahat oldukları dil ve kavramları kullanabilmesi, böylece takımlar arası iletişim ve entegrasyonun kolaylaşması.

Bu çalışmada MDA'nın ve ilgili teknolojilerin yazılım endüstrisinin bugününe ve yarınına getireceği gelişmeler araştırılacaktır. Yazılım geliştirmede modeli temel birim kabul eden bu yaklaşımın varolan gerçekleştirmeleri kullanılarak bir konum sunucusu uygulaması geliştirilecektir. Bu uygulama geliştirilirken MDA'nın hangi aşamalarda ne gibi farklılıklar getirdiği irdelenecektir.

Şekil 1.1'de MDA'nın yapısı görülmektedir. Bu bölümde MDA'yı oluşturan katmanlar ve standartlar hakkında genel bilgiler verilecektir [3].



Şekil 1.1 Model-Güdümlü Mimari (MDA)

1.1 Merkez

Mimarinin temeli, OMG'nin modelleme standartlarına dayanmaktadır: Unified Modelling Language, Meta Object Facility ve Common Warehouse Meta-Model. Merkez belli bir sayıda UML profilini içerir. Bu profillere örnek olarak, bileşen ve harekete (transaction) dayalı yapısıyla kurumsal sistemler ile ilgili olan 'Kurumsal Dağıtık Nesne Profili' (UML Profile for Enterprise Distributed Object Computing) ve kaynak kontrolü konusundaki özel gereksinimleri olan gerçek-zamanlı sistemlerini temsil edebilen 'Planlama, Performans ve Zaman Profili' (UML Profile for Schedulability, Performance and Time) verilebilir. Profillerin sayısı artmaktadır, ancak nihayetinde toplam profil sayısı büyük olmayacaktır. Her UML profili kendi kategorisindeki ara katman yazılımı platformlarının ortak özelliklerini temsil etmektedir ve platform bağımsızdır.

1.1.1 UML (Unified Modeling Language)

UML; mimarinin, nesnelerin, nesneler arasındaki etkileşimleri, uygulama yaşam döngüsünün veri modelleme tarafını, bileşen tabanlı yazılım geliştirmenin tasarım tarafını (gerçekleştirme

ve entegrasyon dahil) modellemek amacıyla kullanılmaktadır. UML, eski (legacy) sistemlere ait yapıları da temsil edebilmektedir. UML modellerinde gösterilen yapılar (Sınıflar, arayüzler, kullanım durumları, aktivite grafları vs. ile) XMI ile, yazılım geliştirme yaşam döngüsü zincirindeki diğer araçlara kolayca taşınabilmektedir. UML profilleri (CORBA, Java EJB, EDOC - Enterprise Distributed Object Computing vs. için) şu anda çeşitli standartlaşma aşamalardadırlar. Bunlar UML topluluğu (model tabanlı analiz ve tasarım), geliştirici topluluğu (Java,VB, C++ geliştiricileri) ve ara katman yazılımı topluluğunu (CORBA, EJB, SOAP geliştiricileri) birbirine bağlayan köprülerdir.

1.1.2 MOF (Meta Object Facility)

MOF, MDA'da kullanılan standart modelleme ve değişme yapılarını sağlamaktadır. Diğer standart OMG modelleri (UML ve CWM dahil), MOF yapıları cinsinden tanımlanmıştır. Bu ortak yapı, model/metaveri değişme ve birlikte işlerlik için temel oluşturmakta ve XMI'da modellerin analizinin yapıldığı mekanizmayı sağlamaktadır.

1.1.3 CWM (Common Warehouse Metamodel)

CWM, OMG veri ambarı standardıdır. Veri ambarı uygulamalarının tasarım, gerçekleştirme ve yönetimden oluşan yaşam döngüsünün tamamını kapsar. CWM, MDA'nın belli bir uygulama alanına uygulanışını gösteren bir örnek olarak da görülebilir.

1.2 Teknoloji Katmanı

Uygulama hedefi CCM, EJB veya başka bir bileşen veya hareket-tabanlı platformdan hangisi olursa olsun, MDA-tabanlı bir uygulama oluşturulurken yapılacak ilk şey, uygun UML profili kullanılarak, uygulaya ait bir platform-bağımsız model (PIM) oluşturulmasıdır. Platform uzmanları, bu uygulama modelini CCM, EJB, COM+ gibi belli bir platforma uygun bir modele dönüştürecektir. Standart eşlemlemeler bu dönüşümün bir kısmının otomasyonunu sağlayabilecektir. Şekil 1.1'de bu hedef platformlar, merkezin dışındaki halkayı kapsamaktadır.

Bu hedef platformlar içinde, gerçekleştirme dilinden bağımsız bir platform olan CORBA'nın özel bir rolü vardır. Diğer hedef platformlarda CORBA'nın sağladığı birlikte işlerliği sağlamak için ek çalışmalar yapılması gerekmektedir.

Platform-bağımlı model (PSM) uygulamanın hem iş yönünü hem de teknik yönünü temsil etmektedir. PSM de bir UML modelidir, ancak hedef platformun elemanlarını aksettiren farklı

bir UML profiliyle gösterilecektir. Platform-bağımsız orijinal modelin içerdiği anlam platform-bağımlı modele de taşınacaktır.

1.2.1 XMI (XML Metadata Interchange)

XMI çeşitli araçlar, ara katman yazılımları ve havuzlar (repository) arasındaki standart değişim (interchange) mekanizmasıdır. XMI ayrıca UML ve MOF modellerinden otomatik olarak XML DTD'leri veya XML şemaları elde edilmesini sağlamaktadır. XMI *modelleme* ve *mimari* kavramlarını XML dünyasına taşımaktadır.

1.3 Yaygın Hizmetler (Pervasive Services)

Bağlamı (şirket içi, İnternet veya gömülü sistemler) ne olursa olsun bütün uygulamalar, bazı temel hizmetleri kullanırlar. Bu hizmetler kaynağa göre değişebilir, ama tipik olarak bunlar dizin hizmetleri, Olay İşleme (Event Handling), Sürerlik (Persistence), Hareketler (Transactions) ve Güvenlik gibi hizmetlerdir.

Bu hizmetler belli bir platform için tanımlanır ve gerçekleştirilirse, ister istemez kendilerini o platformla kısıtlayacak veya en iyi o platformda çalışmalarını sağlayacak belli karakteristik özellikler alırlar. Bundan kaçınmak için OMG, bu hizmetleri *Yaygın Hizmetler* adıyla platform-bağımsız model seviyesinde UML ile tanımlamaktadır. Sadece bir yaygın hizmetin özellikleri ve mimarisi sabitlendikten sonra, MDA'nın desteklediği bütün platformlar için platform-bağımlı tanımlamalar üretilebilecektir.

Yaygın hizmetler, bir platform-bağımsız bileşen modelinin soyutlama seviyesinde sadece yüksek bir seviyede görünmektedirler (Aynen CCM ve EJB'deki bileşen geliştiricinin bakış açısında olduğu gibi). Model belli bir platforma eşleştirildiğinde, bu platformların somut hizmetlerini çağıran kod parçaları üretilecektir.

Şekil 1'de, Yaygın Hizmetler, her ortamdaki her uygulamanın kullanımına hazır olduklarını vurgulamak amacıyla dairenin en dışındaki halkada gösterilmişlerdir. Gerçek bir entegrasyon, dizin hizmetleri, olaylar, sinyaller ve güvenlik için ortak bir model geliştirilmesini gerektirmektedir.

1.4 OMG Alan (Domain) Belirtimleri

OMG faaliyetlerinin büyük bir kısmı, belirli endüstrilerdeki hizmetlerin standartlaştırılması üzerine odaklanıştır. Başlangıçta bu belirtimler OMG IDL (CORBA arayüz tanımlama dili)

ile yazılmış arayüzlerden ve bunlarla ilgili açıklayıcı metinlerden oluşmaktaydı (CORBAfacilities). Bileşenleri platform seviyesinde (CORBA) standartlaştırmak entegrasyon ve birlikte işlerlik sorunlarına katkı sağlasa da, MDA bu noktada daha fazla şey vaat etmektedir.

İyi tasarlanmış bir hizmet, her zaman olayın altında yatan anlamsal modele (hedef platformdan bağımsız olarak) dayalı olmalıdır. Buna göre, MDA'da alan belirtimlerinin kullanışlılığını ve etkisini azami hale getirmek amacıyla, alan belirtimleri, standart platform-bağımsız UML modelleri ile ve en az bir hedef platform için standart platform-bağımlı UML modelleri ile tarif edilecektir. MDA'daki ortak temel, gerçekleştirme kodunun kısmen üretilmesini sağlayacaktır, ancak gerçekleştirme kodu doğal olarak standartlaştırılmayacaktır.

DTF'ler (Domain Task Force) kendi uygulama alanlarındaki standart işlevler için standart yapılar üretmektedirler. Örneğin, tahsil olunacak hesaplar için bir Finans DTF standardı, UML ile ifade edilen bir platform-bağımsız model, IDL arayüzlerini kullanan CORBA'ya özgü bir UML modeli ve Java arayüzlerini kullanan Java'ya özgü UML modelini içerebilir.

OMG, onlarca DTF'ye sahiptir. Şekil 1.1'de bunlardan bazıları dairenin dışından yayılan oklar şeklinde gösterilmiştir.

2. MDA'NIN ORTAYA ÇIKIŞI

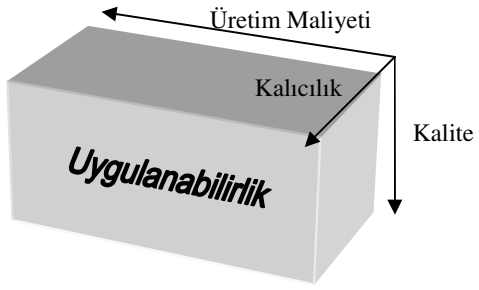
Bu bölümde öncelikle yazılım endüstrisinin karşı karşıya olduğu temel zorluklar anlatılacaktır. Daha sonra yazılım geliştirmenin tarihsel evreleri olarak tanımlayabileceğimiz makine merkezli, uygulama merkezli ve işletme merkezli yazılım geliştirme aşamaları ve nihayetine model güdümlü yazılım geliştirme incelenecektir. Ayrıca MDA'ya temel oluşturmaları açısından, işletme merkezli yazılım geliştirmeye ait kavramlar irdelenecektir.

2.1 Yazılım Endüstrisinin Karşılaştığı Temel Zorluklar

Bilindiği gibi, modern bir işletmenin çalışmasında gittikçe daha kritik öneme sahip olan yazılımları geliştiren BT yöneticileri ve girişimcilerinin yüzleşmesi gereken birçok zorluk bulunmaktadır. Bu sistemleri üretmek, tecrübeli programcıların yorucu ve oldukça detaylı çalışmalarını gerektirmektedir.

Üstelik, çoğu yazılım geliştirme yatırımı hayal kırıklığına uğratan sonuçlar vermektedir. Bazısı tamamen başarısızlıkla sonuçlanmakta veya bütçe aşımı nedeniyle yönetim tarafından sonlandırılmaktadır. Başlangıçta başarılı görünen bir kısmının ise, zaman içinde kararlı veya esnek olma özelliğinden yoksun olduğu görülmektedir. Örneğin, ABD'deki şirketlerin %85'inin BT bölümleri, şirketin stratejik ihtiyaçlarını karşılamamaktadır [Frankel, 2003].

Yazılım endüstrisinde, ekonomik uygulanabilirlik, kalitesi ve kalıcılığı, üretim maliyetiyle orantılı sistemleri hangi ölçüde üretebildiğimizle belirlenmektedir. Yüksek kaliteli, uzun ömürlü yazılım üretmek pahalıya mal olmaktadır. Bunun sonucunda, bazen kalite, kalıcılık veya üretim maliyetinden taviz vermek zorunda kalırız. Bu *uygulanabilirlik değişkenleri* arasındaki ilişki aşağıdaki şekilde görülmektedir [Frankel, 2003].



Şekil 2.1 Uygulanabilirlik değişkenleri [Frankel, 2003]

Bu değişkenlerden birini, diğerlerini etkilemeden değiştirerek uygulanabilirliği artırmak zordur. Ekonomik uygulanabilirliği artırmak için, kalite ve kalıcılıktan taviz vermeden üretim maliyetini düşürmek gerekmektedir.

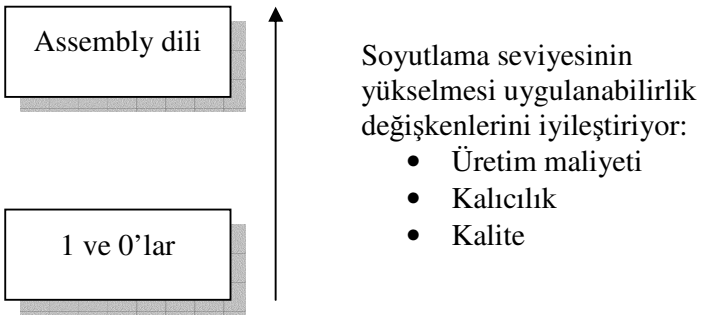
2.2 Makine-Merkezli Yazılım Geliştirme

İlk programcılar, doğal merkezi işlem birimi komutlarına karşılık gelen 1 ve 0'lardan ibaret bit dizileri şeklinde kodlama yapmışlardı. Zamanın kısıtlı hafıza ve işlemci kaynakları açısından zorunlu olan bu yapı, pahalı donanım maliyeti ile birlikte bilgisayar programlarının kullanılabildiği sahaları oldukça kısıtlamıştır.

Assembly dilinin icadıyla, programcılar bilgisayarın anlayacağı basit kısaltmalar kullanarak program yazabilir hale geldiler. Bir assembler yazılan bu kısaltmaları işlemcinin anlayacağı ikili sisteme çevirebilmekteydi.

Assembly dili kullanımı, hem program yazma sürelerini kısaltmış, hem de hata oranını çok daha aza indirgemiştir [Frankel, 2003]. Ayrıca yazılan bir program, revize edilmiş bir assembler ile bir işlemcinin farklı bir versiyonunda, kolayca uygun bit dizisi haline çevrilebilmiş, bu da programların kalıcılığının artmasını sağlamıştır.

Soyutlama seviyesinin 1 ve 0'lar düzeyinden yukarı çekilmesi, uygulanabilirlik denklemindeki üç değişkeni de iyi yönde etkilemiş (Şekil 2.2), böylece büyük firmalar ve devlet kurumlarının muhasebe, faturalama gibi işlerini bilgisayarlar yardımıyla yapabilmelerini pratik hale getirmiştir.



Şekil 2.2 Soyutlama seviyesinin yükselmesi

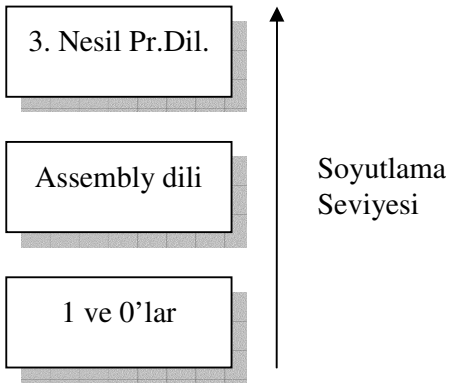
2.3 Uygulama-Merkezli Yazılım Geliştirme

Assembly dili programcıları, 1 ve 0'lardan kurtulmalarına rağmen, çok dar ve alt seviyeli komut setleri ile çalışmak zorundadırlar. Bir çalışanın maaşının tablodan okunup üzerinde bir hesap yapılması bile yüzlerce satır kod gerektirmektedir.

Üçüncü-nesil programlama dilleri büyük bir verimlilik sıçrayışı yaratmıştır. İlk temsilcileri olan FORTRAN ve COBOL soyutlama seviyesini işlemci komut setinin çok üstüne çıkarmıştır. FORTAN'daki basit bir PRINT komutu yüzlerce satır assembly kodunun yerine geçebilmektedir. *Derleyiciler* bu yüksek seviyeli komutları makine diline çevirmektedirler.

C ve Pascal gibi yapısal programlama dilleri ile, daha da güçlü programlama modelleri ortaya çıkmıştır. Artık sistem üreticileri işletim sistemi servislerini tanımlarken dahi üçüncü nesil programlama dillerini kullanmaya başlamışlardır. Ayrıca çeşitli derleyiciler vasıtasıyla yazılan programlar, farklı işlemcilerde çalışacak şekilde derlenebilmektedir. Bu taşınabilirlik kavramı ile programların kalıcılığı daha da artmıştır. Soyutlama seviyesinin daha da yukarı çekilmesi, uygulanabilirlik denklemindeki üç değişkeni de iyi yönde etkilemiş (Şekil 2.3), artık küçük firmaların da işlerini bilgisayarlar yardımıyla yapabilmelerini sağlamıştır.

Ayrıca işletim sistemlerinin de ortaya çıkmasıyla derleyiciler artık, disk ve görüntü işlemleri gibi rutin işletim sistemi operasyonları için detaylı kod oluşturmak yerine, sadece gerekli işletim sistemi servislerini çağırmaktadırlar.



Şekil 2.3 Üçüncü nesil programlama dillerinin soyutlama seviyesini artırması

2.4 Nesne-Yönelimli Diller ve Sanal Makineler

Kaçınılmaz olarak, programlardan beklenen işler karmaşıklaştıkça, varolan geliştirme metotları yetersiz kalmaya başlamıştır. Bu noktada ortaya çıkan Smalltalk, C++ gibi üçüncü nesil nesne yönelimli diller, program parçalarının farklı bağlamlarda kullanımını kolay hale getirmişlerdir.

Bazı nesne yönelimli diller (C#, Java gibi), sanal makine denilen yorumlayıcılar vasıtasıyla derleyicisi tarafından oluşturulmuş ara kodu çalıştırabilmektedir. Bu ara kod, işlemci ve işletim sisteminden bağımsızdır.

Sanal makinenin farklı işlemci ve işletim sistemlerinde gerçekleştirilmesiyle, uygulamaların derlenmiş haldeki kodu bile farklı ortamlarda çalıştırılabilir hale gelmiştir. Artan taşınabilirlik, uygulamaların kalıcılığını daha da artırmıştır.

2.5 İşletme-Merkezli Yazılım Geliştirme

Zaman ilerledikçe, bilişimden beklenen otomasyonun seviyesi artmaya devam etmiştir. Artık bir işletmede birbirinden bağımsız, ama aynı zamanda birbiriyle çakışan fonksiyonaliteye sahip, bu sebeple de kıt kaynakları boşa harcayan ürünlerin entegre edilmesi gereksinimi doğmuştur. İşletme-merkezli yazılım geliştirme aşağıdaki kavramları gündeme getirmiştir:

2.5.1 Bileşen-Tabanlı Yazılım Geliştirme

Bileşen kavramı, endüstriyel üretim mantığından esinlenerek ortaya çıkmış, ve uygulamaların birbiriyle çalışabilen çeşitli bileşenlerin birleştirilmesiyle oluşturulabilmesini sağlamıştır. Uygulamalarda bağımsız olarak kullanılabilen ve diğer bileşenlerle birlikte çalışabilen, derlenerek paketlenmiş yazılım modüllerine *bileşen* denir.

Aynı çözümün tekrar tekrar icat edilmeye çalışılmaması, verimliliği ve üretim maliyetini düşürmüş, izole edilmiş fonksiyonalite ile hata düzeltme ve geliştirme bağlamı, bileşen düzeyine indirgenerek toplam kalitenin artmasını sağlamıştır.

2.5.2 Tasarım Örnekleri

Tasarım örneği kavramı, programcıların başka insanların kapsamlı bir biçimde düşündüğü ve onayladığı ortak tasarım örneklerini kullanmaktadırlar.

Örneğin, *J2EE Value Object* tasarım örneği dağıtık bileşenlerin veri transferinin verimli bir biçimde gerçekleştirilmesini sağlamaktadır [7].

2.5.3 Dağıtık Hesaplama

En eski bilgisayarlar aynı anda en fazla bir işi yürütebilmekteydiler. Sonraları, çeşitli yöntemlerle bilgisayarların birden fazla iş yürütebilmesi sağlandı. Bununla birlikte, birden fazla kullanıcının kendine ait G/Ç cihazları ile aynı bilgisayarı kullanması da mümkün olmuştur.

İlk etapta bütün bu kullanıcılara ait işler tek bir bilgisayarda yapılırken, dağıtık hesaplama ile, kullanıcıların birçok işi kendilerine ait az maliyetli bilgisayarlarda (PC) yapabilmesi mümkün olmuştur. Bu istemci-sunucu yapısı ile, ana bilgisayarların yükü oldukça hafiflemiştir.

Sonraları ise, istemci-sunucu yapısı, gittikçe her düğümün duruma göre istemci veya sunucu olarak davranabildiği daha eşten-eşe (peer-to-peer) bir yapıya bürünmüştür.

2.5.4 Ara Katman Yazılımı (Middleware)

Başlangıçta, dağıtık hesaplama çoğunlukla işlemciler veya alt seviyeli ağ protokolleri arasındaki firmaya has, özel mesajlaşma sistemlerine dayanıyordu. Sonraları, bu özel

sistemler çeşitli platform ve işletim sistemlerini bağlayan genel-amaçlı sistemlere dönüştü ve ara katman yazılımı adını aldı.

Ara katman yazılımları, uygulama programcılarının, iş mantığına daha çok; mesajlaşma, hareket kontrolü, güvenlik vs. gibi her uygulamada olan yeteneklere daha az konsantre olmalarını sağlayarak, *platform soyutlama seviyesi*ni artırmaktadır.

CORBA, J2EE, .NET günümüzün önemli ara katman yazılımlarına örnek gösterilebilir. Bunlar aynı zamanda bileşen-tabanlı yazılım geliştirmeyi de desteklemektedir. Ayrıca CORBA, programla dilinden tamamen bağımsız olarak bir ara katman yazılımı olarak *programlama soyutlama seviyesi*ni de yukarı çekmektedir.

2.5.5 Bildirimsel Belirtim (Declarative Specification)

Bildirimsel Belirtim bir sistemin nitelikler belirlenerek programlanabilmesini sağlamaktadır. Bir sistemin sıralı, yordamsal (procedural) komutlar vasıtasıyla programlanması manasındaki *komutsal belirtim* kavramının karşıtıdır.

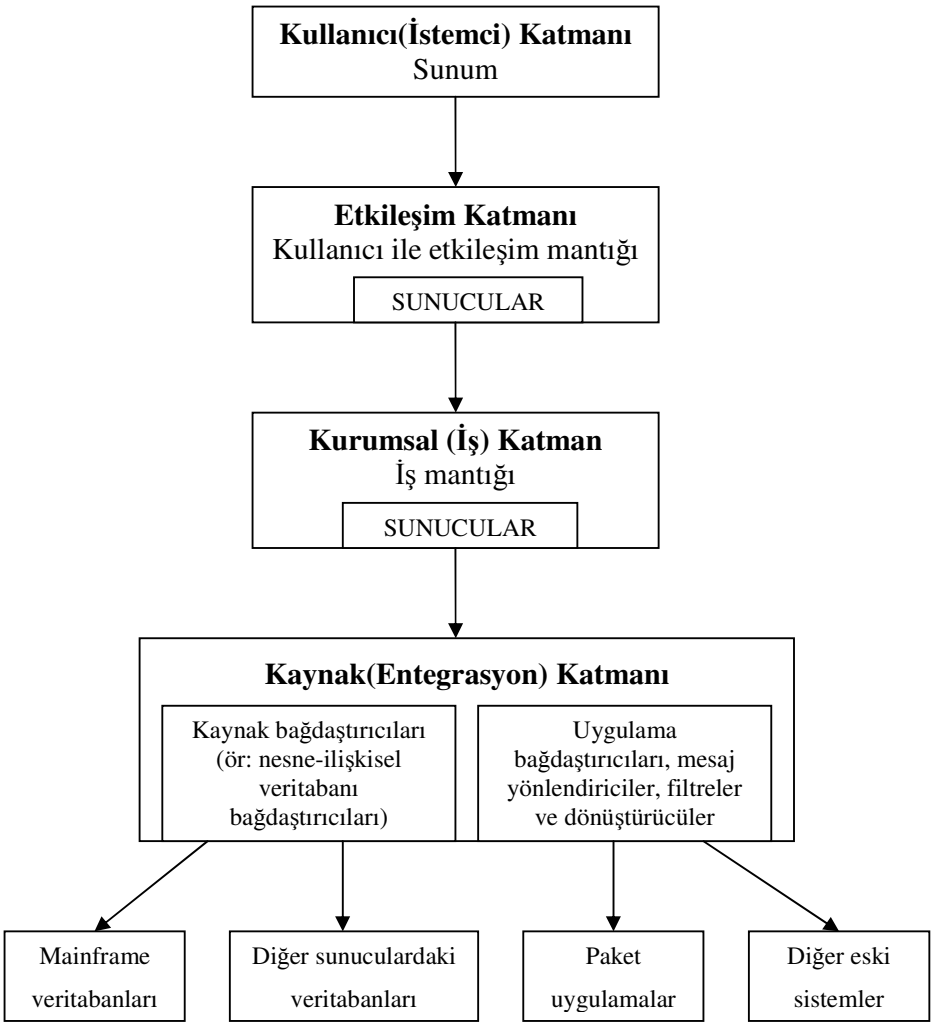
Grafik kullanıcı arayüzü tasarım araçları, veritabanı sistemleri bildirimsel belirtimin en çok kullanılan örnekleridir.

EJB, .NET, CORBA bileşen modeli ile bir bileşenin bir çok niteliği kod yazmak yerine, bildirimsel olarak belirlenebilmektedir. Bildirimler iki şekilde işlenir:

- Otomatik kod üretimi (Code generation)
- Yürütüm süresinde yorumlama (Runtime interpretation)

2.5.6 Kurumsal Mimari ve Katmanların Ayrılması

Yazılım mimarisi, işleri birbirinden ayırarak karmaşık sistemleri organize etmeye çalışmaktadır. Şekil 2.4'te günümüzün modern uygulamalarında kullanılan dört-katmanlı kurumsal mimari görülmektedir [HS, 2000]:



Şekil 2.4 Dört-katmanlı kurumsal mimari

Bakış açısı (viewpoint) kavramı da, işlerin birbirinden ayrılmasını destekleyen başka bir kavramdır. Bakış açısı, bir sistemin o bakış açısı ile ilgisi olmayan yönlerini filtreleyen bir izdüşümüdür.

2.5.7 Kurumsal Uygulama Entegrasyonu (Enterprise Application Integration)

Her ne kadar, kurumsal mimari iyi entegre olmuş sistemler tasarlamayı mümkün kılsa da, eski sistemler ve paket uygulamalar gözardı edilemez. Kurumsal Uygulama Entegrasyonu, kaynak katmanına *uygulama bağdaştırıcıları* kavramını eklemektedir. EAI sistemleri, bağdaştırıcılar arasında olay-tabanlı bir iletişimi destekleyerek, birbirinden bağımsız eski sistemleri kurumsal sisteme entegre edebilmektedir. EAI

sistemleri, çeşitli işleri tamamladıklarında asenkron mesajlar gönderen ve diğer modüllerden gelen mesajları işleyen farklı modüller ile çalışmaktadır.

Mesaj tabanlı ara katman yazılımlarına örnek olarak, IBM Websphere MQ Integrator, Microsoft MSMQ ve Java Messaging System (JMS) verilebilir.

2.5.8 Kontrat ile Tasarım (Design By Contract)

Kontrat ile Tasarım, yazılım modüllerinin kontratını açık ve formal bir biçimde yapmaya odaklanan güvenilir yazılım geliştirme yaklaşımıdır. Kontrat ile Tasarım, önkoşul, son koşul ve değişmezler biçiminde formal kısıtlar yazılmasını içermektedir. Bu konuda daha detaylı bilgi için 4.4 bölümüne bakınız.

2.5.9 İşletme-Merkezli Yazılım Geliştirmedeki Zorluklar

Üretim Maliyetinin Düşürülmesi:

Karmaşık taleplerin karşılanmasına karşı kurumsal programlamanın yoğun emek gereksinimi, üretim maliyetlerini baskı altına almaktadır. Örneğin bir müşteriden-işletmeye yapılan harekete (customer-to-business transaction) yeni bir veri elemanı eklenmesi, birçok sistem katmanında değişiklik gerektirecektir (GUI, XML dosyaları, veritabanı, bağdaştırıcılar, filtreler vs.). Yazılım geliştiriciler, bir insanın yönetmesi ve senkronize halde tutması oldukça zor olan bu karmaşa ile uğraşmakla zaman kaybetmektedirler.

Kalitenin Sağlanması:

Çoğu zaman, programcılar özenli bir mimari ve tasarıma sahip programlar yazmamaktadırlar. Makine, elektrik gibi diğer mühendislik dallarında her zaman, gerçekleştirme başlamadan önce özenli ve tam bir mimari ve teknik ürün tasarımı gerekmektedir. Oysa, yazılım endüstrisinde kodlamayı önemli olan tek şey olarak görme, bunun dışındaki her şeyi konu dışı görme eğilimi ağır basmaktadır. Gittikçe karmaşıklaşan yüksek kaliteli yazılımları kısa zamanda geliştirmek durumunda kalan programcılar, örneğin Kontrat ile Tasarım gibi özen gerektiren teknikleri sahip olamadıkları bir lüks olarak görmektedirler.

Kalıcılığın Sağlanması:

Günümüzde platform değişiklikleri kaçınılmazdır. Bir yazılım geliştirme yöneticisi,

endüstrinin rekabet ortamı içinde, varolan bir projede platform değişikliği kararı aldığında, geliştiricilerinin bu teknolojilere uyum sağlama zamanını, varolan diğer sistemlerle bu platformun uyuşturulmasını düşünmek zorundadır. Böyle bir durumda geçmişteki bir çok emek boşa gidebilecek ve geçiş süresi de yine ciddi emek gerektirebilecektir. Çizelge 2.1'de çok kullanılan bazı teknolojilerin kısa sürede nasıl değiştiğini görülmektedir:

Çizelge 2.1 Platform Değişkenliği

FİRMA	**ÖNCEKİ**	**ŞİMDİKİ**
Sun	Java dili	J2SE, J2EE
W3C	XML DTD	XML Schema, WSDL, vs.
OMG	CORBA	CORBA Bileşen Modeli
Microsoft	MTS	COM+/.NET
Sun ve OMG	RMI veya IIOP	IIOP üzerinden RMI

2.6 Model Güdümlü Yazılım Geliştirme

Çoğu geliştirici, yazılım modellerine tasarım materyali, program kodlarına ise geliştirme materyali olarak bakmaktadır. Birçok firma model tasarımcısı ve programcı rollerini birbirinden ayırmaktadır.

Bunun sonucunda, modeller kuralsız(informal) olmakta ve bu yüzden makinenin işleyeceği bir yapıya sahip olamamaktadırlar. Programcılar bunları rehber ve tanım olarak kullanmakta, ancak üretime doğrudan katkı sağlayan şeyler olarak kullanmamaktadırlar.

MDA, makine tarafından işlenebilir formal modeller kullanmaktadır. Böyle modeller üretim sürecinin doğrudan parçalarıdırlar. Böyle bir ortamda, model tasarımcısı ve programcının rolleri birbirinden çok ayrı değildir. Modelleme bir programlama aktivitesidir.

MDA, bileşen tabanlı yazılım geliştirmeyi, tasarım örneklerini, ara katman yazılımlarını, bildirimsel belirtimi, soyutlamayı, çok katmanlı sistemleri, EAI ve Kontrat ile Tasarımı yeniden icat etmemekte, bunların üzerinde gelişerek, daha iyi çalışmalarına yardım etmektedir.

MDA ile soyutlama seviyesini 3. nesil programlama dillerinin üzerine çıkarmak her üç uygulanabilirlik değişkenini de iyileştirmektedir.

- Verimlilik artmaktadır, çünkü modeldeki bir komposizyon yapısı, 3. nesil programlama dilinin satırlarca koduna denk düşmektedir.
- Kalite artmaktadır, çünkü kod üreticiler iyi test edilmiş kodları tekrar oluşturmaktadırlar.
- Kalıcılık artmaktadır, çünkü anlamsal olan kısımlar, 3. nesil programlama dilleriyle alakalı kısımlardan ayrılmıştır.

2.6.1 Platform Bağımsızlığı

OMG Platform-Bağımsız Model (PIM) ve Platform-Bağımlı Model (PSM) kavramlarının tanımını yapmıştır. Platform bağımsızlığı göreceli bir kavramdır. Belli platform veya platformlar nazarında bir anlam taşır.

Örneğin, CORBA ilk ortaya çıktığında, 3. nesil programlama dilleri ve işletim sistemlerinden bağımsız olduğu için platform bağımsız sayılıyordu. Oysa, şimdi bir kaç belli başlı ara katman yazılımı var olduğu için, CORBA bir platform kabul edilebilir.

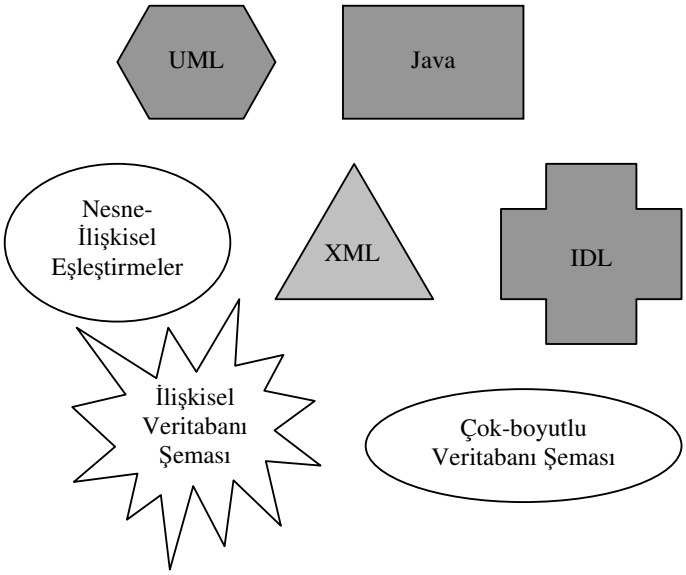
Bu yüzden, hangi platformlardan bağımsız olduğunu belirtmeden bir model veya modelleme diline platform-bağımsız demenin bir anlamı yoktur. Platform olarak ara katman yazılımını kabul edersek, MDA bağlamında, ara katman yazılımından bağımsız bir bileşen modeline platform bağımsız diyebiliriz.

2.6.2 Metaveri (Metadata) Entegrasyonu

EJB gibi ara katman yazılımı teknolojileri, belli noktalarda detaylı 3. nesil program kodlama yerine, bildirime dayalı programlama kullanmaktadır. Ancak, EJB'nin bildirimsel XML-tabanlı bileşen betimleyicileri Java ile pek iyi entegre olamamışlardır. Bunun ötesinde bir EJB bileşeninin CORBA nesnesi olarak kullanılabilmesi için bileşenin bir CORBA IDL'si olması gerekmektedir. Bu durumda bileşenin farklı taraflarını tanımlamak için üç dil kullanmak gerekmektedir (Java, XML, CORBA IDL). Bunun ötesinde bileşenin bağlı olduğu veriyi tanımlayan veritabanı şemaları, veri modelleme dillerinde ifade edilmektedir. İlişkisel veritabanlarını tanımlamak için veya çok boyutlu şemaları tanımlamak için farklı veri

modelleme dilleri bulunmaktadır. Nesne-ilişkisel eşlemleme araçları, kaynak katmanında kurumsal katman ve ilişkisel şema arasındaki eşlemlemeleri tanımlayan başka bildirimlere de ihtiyaç duymaktadır. Ayrıca, bileşeni anlatan UML modelleri de olabilecektir.

Bütün bu tanımlamalar, bileşen hakkındaki metaverileri oluşturmaktadır. Bunların hepsi birbirinden tamamen farklı dillerde ifade edilip, duruma göre birleştirilmektedirler (bkz. Şekil 2.5).



Şekil 2.5 Plansız (ad hoc) metaveri entegrasyonu

Problem, bir bileşenin farklı taraflarını anlatmak için farklı diller kullanılması değildir. Tek bir dilin bir sistemin bütün yönlerini anlatabilmesini beklemek gerçekçi değildir. Problem, farklı dillerde yapılanları entegre edecek bütün bir mimarinin bulunmamasıdır.

MDA'nın bu resme getirdiği en önemli geliştirmelerden birisi, çok çeşitli dillerde ifade edilseler dahi, metaverilerin bütünleşmiş bir biçimde yönetilmesini sağlayan bir mimaridir (bkz. Şekil 2.6).

UML	Java
XML	IDL
Nesne-İlişkisel Eşleştirmeler	İlişkisel Veritabanı Şeması
Çok-boyutlu Veritabanı Şeması	B2B İşbirliği Tanımları

Şekil 2.6 Düzenli metaveri entegrasyonu

5. bölümde MDA'nın önemli yapıtaşlarından biri olan Meta Object Facility (MOF)'nin model-tabanlı teknolojiyi kullanarak birbirinden farklı metaverilerin nasıl düzenli bir şekilde yönetilmesini sağladığı incelenecektir. Temel fikir her dilin formal bir modelinin tanımlanmasıdır. Bu modeller MOF-tabanlı metaveri yönetim olanaklarını yürütmektedir. MOF bir uygulama geliştiricinin farklı çeşitlerdeki metaveriyi birleştirerek bunun belirli bir amaca göre anlamlı olmasını garanti altına alamamaktadır. Ancak, metaveriye erişimin ve onu değiştirmenin tutarlı bir biçimde gerçekleşmesini sağlamaktadır.

2.6.3 MDA ve Bileşen-Tabanlı Yazılım Geliştirme

MDA, bileşen geliştirme ve bileşenleri bir araya toplayarak uygulama geliştirme sürecinin daha fazla bölümünü otomatikleştirmeye çalışarak, bileşen tabanlı yazılım geliştirmenin endüstriyel üretimdeki prensiplerinin yazılım geliştirme sahasına uygulamasını güçlendirmektedir.

Bileşen-tabanlı yazılım geliştirmede Herzum-Sims yaklaşımı, bir iş bileşeninin içeriğinin tanımına tasarım nesnelerini de dahil ederek MDA'nın ön habercisi olmuştur [Herzum, 2000]. Herzum ve Sims bir bileşenin tekrar kullanılabilirliğinin, teknik olarak birleştirilebilir olmanın ötesine bağlı olduğunu anlamışlardır. Bir bileşenin tasarımı, onu kullanmaya niyeti olan kişiler tarafından anlaşılmalıdır. Bir bileşenin kurulup çalıştırılabilmesi için, yürütme süresi öğelerini paketine dahil etmek ne kadar kritikse, tasarım öğelerini de pakete dahil etmek onları kullanmak isteyen

kişilerin erişebilmesi açısından önemlidir.

MDA ile, bir iş bileşeninin daha önce tasarım öğesi olarak görülen bazı modelleri yüksek derecede formalleşerek, geliştirme öğesi karakteristiğine bürünmektedirler. Bu öğeler, bir iş bileşeninin etkileşim ve kurumsal katmanlarındaki uygulama arayüzlerinin üretilmesini otomatikleştiren kod oluşturucuların çalışmasını sağlamakta ve uygulama arayüzlerinin en azından bir kısmının gerçekleştirilmesini de otomatikleştirmektedirler.

Bunun yanında, MDA bağlamında, bileşen gerçekleştirmelerinin bunları kullanacak uygulamalar geliştirilmeden önce, tamamen üretilmesi gerekmemektedir. Olası senaryolardan biri şöyledir: Programcı bileşenlerin modellerini yeniden kullanmakta, çeşitli bileşen konfigürasyon parametrelerine değerler vermektedir. Bileşenin içeriği, uyarlama gereksinimlerini karşılayacak minimum düzeydedir. Bu bileşenin bütün potansiyel fonksiyonalitesini içeren bir oluşturmanın tersidir. Bir MDA kod-üreticisi sadece konfigürasyon parametrelerinin gerektirdiği kadar fonksiyonaliteyi oluşturmak durumundadır.

2.6.4 Tasarım Örneklerinin Otomatik Kullanımı

MDA kod-üreticileri tasarım örneklerini otomatik olarak uygulama kabiliyetine sahiptir. Bu, tasarım örneklerinin yazılım geliştirmedeki endüstriyel üretimdekine benzer rolünü kuvvetlendirmektedir.

Bir J2EE Örneği

J2EE Değer Nesnesi (Value Object) tasarım örneğini ele alalım. Değer Nesnesi tasarım örneğinin ana fikri, birden fazla bileşen niteliğinin tek bir uzak çağrı ile değerlerinin alınması veya değiştirilmesini sağlamaktır. Her nitelik için birer değer alma/değiştirme (get/set) operasyonları içeren bir nesne yerine, bu tasarım örneği *façade* (uzaktan erişim için kullanılacak sınıf) ve değer objelerini içeren birkaç nesne kullanır.

Bu tasarım örneğini programcıların her varlık bileşeni için defalarca uygulaması yorucu ve hataya müsait olabilir. MDA yaklaşımı, varlık bileşeninin sadece niteliklerini tanımlayan platform-bağımsız bir modelini (platform=ara katman yazılımı) yaratmayı gerektirmektedir. Kod-üretici tasarım örneğinin gerektirdiği

birkaç Java nesnesini yaratma işini yerine getirir. Varlık bileşenin nitelik tanımlarında, bir kod-üretici aşağıdakileri üretebilecektir:

- Bir *façade* arayüzü ve bunu gerçekleştiren sınıfın tamamı
- Tam bir Değer Objesi sınıfı
- Bileşen (bean) arayüzü
- Bileşen arayüzünü gerçekleştiren iskelet sınıf

Bileşen gerçekleşme sınıfının bir kısmı dışında, bu kodun tamamı mekaniktir. Bu da zaten bir kod-üreticinin üretimi otomatikleştirebilmesinin sebebidir. Bunun yanında, Değer Nesnesi tasarım örneği Java'ya özel değildir. Başka bir ara katman yazılımına, örneğin .NET'e göre ayarlanmış bir kod-üretici, aynı platform-bağımsız modelle başlayarak .NET için anlam taşıyacak şekilde tasarım örneğini uygulayabilir.

Bu biçimdeki otomatik tasarım örneği uygulanması, geliştirme ortamının soyutlama seviyesini, ara katman yazılımının üzerine çıkarmanın avantajlarından biridir.

2.6.5 Model Güdümlü Kurumsal Mimari

UML, bir kurumsal mimarinin bütün katmanlarını modellemek için yeterli değildir. Model güdümlü kurumsal bir mimari, kurumsal sistemin çeşitli soyutlama seviyelerinde formal bir biçimde ifade edilebilmesini sağlayacak, farklı ama birbiriyle uyumlu bir biçimde çalışan modelleme dillerini gerektirmektedir. UML profilleri ve Meta Object Facility bu dilleri tanımlamaya yarayan mekanizmalardır.

Modelleme dillerinin birbirinden ayrılması, meselelerin mimari seviyede birbirinden ayrılmasını sağlamaktadır. Bunun yanında, bir sistemin çeşitli katmanlarının fonksiyonel karakteristiklerini anlatan diller yeterli değildir. Sistemin hareketsel davranış (transactional behaviour), güvenlik ve sürerlik (persistence) gibi fonksiyonel olmayan yönlerini de tanımlayan dillere ihtiyaç vardır.

Platform bağımsız olarak gösterilen diller platformlara kadar inen eşlemlemeleri (mapping) gerektirmektedir. Örneğin, ara katman yazılımından bağımsız diller, J2EE, .NET ve CORBA gibi ara katman yazılımı teknolojilerine ait eşlemlemeleri gerektirir.

Standartlaştırma kuruluşları bu dillerin ve eşlemlemelerin bazılarını tanımlamaktadırlar. Kurumlar, kendi dilleri ve eşlemlemelerini tanımlayarak bu işi daha da geliştirmek durumunda kalacaklardır. Model güdümlü kurumsal bir mimari,

bu dil ve eşlemlemeleri saptamalı, tanımlamalı ve onları mimarinin bütününde konumlamalıdır.

Standart ve firmaya ait diller, eşleşmeler ve eşleşmeleri gerçekleştiren kod üreticiler model güdümlü kurumsal bir mimariyi destekleyen altyapının önemli bir parçasını oluşturmaktadır.

Ara katman yazılımları, MDA'da çok önemli bir yer taşımaktadır. Çünkü ara katman yazılımları platform soyutlama seviyesini yükselterek kod üreticilerin işini kolaylaştırmaktadır.

2.6.6 Kontrat ile Tasarımın MDA'daki Konumu

Kontrat ile Tasarım, yazılım geliştirmede fazla yer edinememiştir. Yazılım geliştiriciler üzerindeki baskı, bu tarzda özenli bir tasarımın çok fazla zaman harcayarak pratik olmaktan çok uzak olduğunu düşündürmektedir.

Disiplinli tasarım tekniklerinin savunucuları tipik olarak, bu gibi tekniklerin başlangıçta zamana mal olduğunu, ama daha az hatalı ve kolay bakımlı yazılımlar üretilmesiyle uzun vadede zaman kazandırdığını söyleyerek buna karşı çıkmaktadırlar. Ancak bu düşünce, kendilerini bekleyen kısa vadeli baskıları göz ardı edemeyecek durumdaki yazılım geliştirme müdürlerini ikna etmeye yeterli olmamaktadır.

MDA, bu tartışmanın koşullarını değiştirmektedir. Kod üreticiler, sabitleri, ön koşul ve son koşulları kullanarak iyi kalitede varsayım kontrolü, istisna yönetimi ve test malzemeleri üretebilmektedirler. Bunlar bir yazılım projesini başarıyla tamamlamak için gerekli şeylerdir ve böylece MDA üzerinden Kontrat ile Tasarım, projenin tamamlanmasını gerçekten hızlandırabilir. Böylece sabitler, ön koşul ve son koşullar ait oldukları model ile birlikte sadece tasarım bileşeni değil, geliştirme bileşeni haline gelmektedirler.

MDA'ya şüpheci yaklaşan yazılım geliştiriciler, MDA'nın formal modeller kullandığını gördüklerinde MDA hakkında daha iyimser bir bakış açısı edinmektedirler. Modellemeden bu kişileri soğutan genelde formalliğin eksikliğidir. Matematiksel olarak tam olan UML Nesne Kısıt Dili (OCL) ile Kontrat ile Tasarımı içeren formal modelleme, bir yazılım geliştirici için çok daha çekici olacaktır.

3. MDA İLE YAZILIM GELİŞTİRME

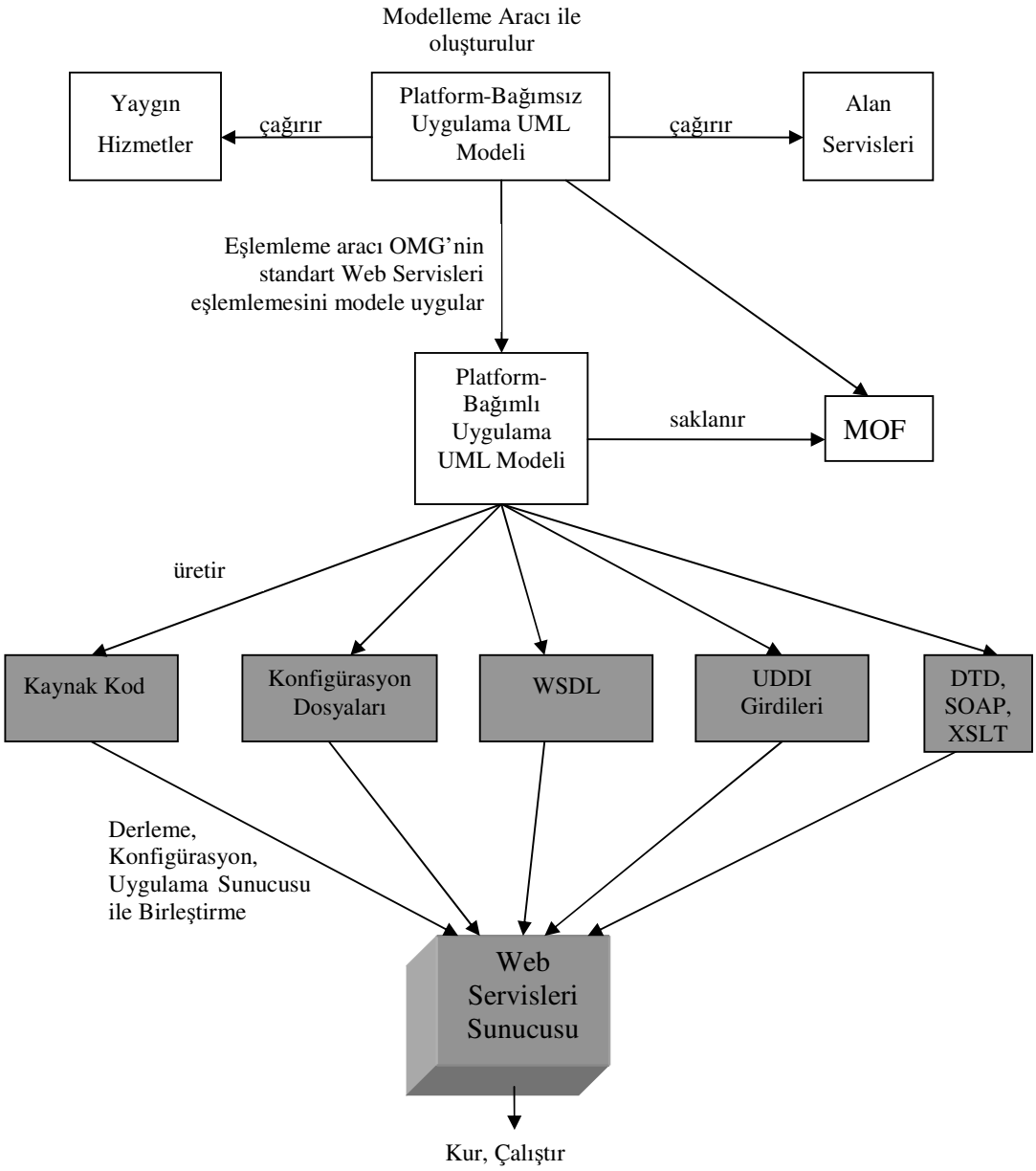
Bu bölümde öncelikle MDA kullanılarak bir uygulamanın seçilen bir platform için nasıl üretileceği anlatılacaktır. İkinci aşamada ise, MDA'nın aynı mekanizmayı birden fazla hedef platform için nasıl tekrar kullandığı gösterilecektir. MDA ile yazılım geliştirmenin asıl avantajları, aynı temel modeli kullanarak her türlü ara katman yazılımı platformu için uygulamalar üretirken ortaya çıkmaktadır [5].

3.1 Birinci Adım: Platform-Bağımsız Model (PIM)

Bütün MDA geliştirme projeleri, bir platform-bağımsız modelin (PIM) yaratılmasıyla başlar. UML ile ifade edilen bu model, Şekil 3.1'de en üstte yer almaktadır. Bir MDA modeli birden fazla seviyede platform-bağımsız model içerir. Bunların her birinin platform-bağımsız olmasına karşın, temel model hariç diğerleri teknolojik davranışlara ait platform-bağımsız bilgiler içerir. Temel PIM, sadece iş mantığı ve davranışlarını tarif eder ve buna Hesaplamadan Bağımsız Model (Computation Independent Model - CIM) de denmektedir.

İş ve modelleme uzmanlarının birlikte çalışmasıyla üretilen bu model, iş kurallarını ve fonksiyonlarını teknolojiyle mümkün olduğu kadar karıştırmadan sunmaktadır. Modelleme ortamının bu kadar açık olması, iş uzmanlarının olayı teknolojik bir model veya uygulama ile anlayacaklarından çok daha iyi anlayabilmelerini sağlamaktadır. Bu teknolojik bağımsızlık, CIM'in uzun yıllar sonra bile değerini korumasını sağlamaktadır. Sadece iş koşulları değiştiğinde bu model değişecektir.

Sonraki seviyedeki PIM'lerde platform-bağımlı detaylar olmamasına rağmen, bazı teknolojik detaylar bulunmaktadır. Örneğin, her bileşen tabanlı platform, geliştiricilerin aktivasyon örneklerini belirlemesine izin vermektedir (Varolan platformlardan birkaçı (ör: EJB) bunun için *session* ve *entity* terimlerini kullanmaktadır. MDA standart profilleri, terminoloji çatışmalarında yorumlamanın doğru olmasını da sağlamaktadır). Ek kavramlar, sürerlik, hareketsellik, güvenlik seviyesi ve konfigürasyon bilgisi aynı şekilde kullanılabilir. Bu kavramların ikinci-seviye platform bağımsız modele eklenmesiyle, modelin platform-bağımlı modele daha doğru olarak eşleşmesi sağlanır.



Şekil 3.1 MDA kullanılarak bir Web Servisi Sunucusunun Üretilmesi

Bu davranışları platform-bağımsız modele yerleştirebilen standart modelleme altyapısının bir kısmı mevcuttur: UML'nin bir parçası olan *Object Constraint Language*, tasarımcıların modellerinde ön-koşul ve son-koşulları belirtmelerine imkan vermektedir. Diğer kısıtlar bir parametrenin NULL olup olamayacağını veya bazı nitelik değerleri kombinasyonları için kısıtlamaları içermektedir. İş uygulamalarında yaygın bir işlem olan parametre değerlerini ayarlama ve alma kolayca

otomatikleştirilebilmektedir. Bu kolaylık ilk jenerasyon MDA araçlarının kod üretme modüllerinde gerçekleştirilebilmektedir.

MDA uygulama modelleme araçları, Yaygın Hizmetler ve Alan Hizmetleri'nin de uygulamaya entegre edilmesini sağlamaktadır. UML ile tanımlanmış her hizmet araca dahil edilebilir ve kolayca uygulamada kullanılabilir. Eğer hizmet farklı bir ara katman yazılımı platformunda çalışıyorsa, MDA geliştirme aracı platformlar arası çağırmaları otomatik olarak gerçekleştirecektir. Bu, platformları arası entegrasyonu sağlamak için gerekli olan elle kodlamayı azaltmakta, hatta bazı durumlarda tamamen ortadan kaldırmaktadır.

İstemciler de MDA'nın bir parçasıdır. Herhangi bir platform veya mimariye bağlı olmayan MDA her türlü istemciyi (web tarayıcı, Java Virtual Machine, telsiz cihazlar vs.) modelleyebilir ve dolayısıyla kodlamasını otomatikleştirir.

Birinci adımda üretilen platform-bağımsız model, hem istemci hem de sunucu için işlev ve davranışları, Yaygın Hizmetler, Alan Hizmetleri ve uygulamanın çağırdığı diğer hizmetlere olan bağlantıları belirtmektedir. UML sınıf ve nesne diyagramları yapıyı; sıra ve aktivite diyagramları davranışları; sınıf ve nesne adları anlamsal notasyonlarla bilikte iş faktörlerini ve modelin diğer tarafları da platform-bağımsız bileşen yapısı ve davranışlarını içine almaktadır.

3.2 İkinci Adım: Platform-Bağımlı Model (PSM)

Birinci adım tamamlandığında, platform-bağımsız model Meta Object Facility (MOF) ile saklanır ve Şekil 3.1'in ikinci sırasında görülen platform-bağımlı modeli oluşturmak üzere eşlemleme (mapping) işlemine tabi tutulur. UML'deki özelleştirme ve uzantılar, UML'ye hem PIM'leri hem de PSM'leri ifade etme olanağını vermektedir. UML Profili adı verilen standart bir uzantı kümesi (*stereotip* ve *etiketli değerler*'den oluşan) belirli bir kullanım için şekillendirilmiş UML yapıları (örneğin belirli bir platform için modelleme) tanımlamaktadır.

PSM'yi üretmek için, uygulamanın modülleri için hedef platform veya platformlar seçilmesi gerekmektedir. Eşleştirme aşamasında, genel bir biçimle uygulama tasarımına eklenilen çalışma-zamanı karakteristikleri ve konfigürasyon bilgileri, hedef olarak seçilen ara katman yazılımı platformunun gerektirdiği özel biçimlere

dönüştürülür. Standart eşleştirmeleri kullanan araçlar ile bu dönüştürme mümkün olduğu kadar yapılır ve muğlak olan kısımlar programcıların elle düzenlemesi için işaretlenir. MDA'nın ilk gerçekleştirmeleri burada ciddi miktarda elle ayarlama gerektirmektedir, bu miktar profiller ve eşleştirmeler olgunlaştıkça zamanla azalacaktır.

MDA tanımı, bir PIM'den PSM'ye geçiş için dört yol belirtmektedir [3]. Artan karmaşıklık ve otomasyon seviyesi bakımından sırası ile bunlar aşağıdaki gibidir:

1) Dönüşüm tamamen elle, her uygulama için ayrı ayrı yapılır.
2) Dönüşüm kabul edilmiş şablonlara (pattern) bakılarak yapılır.
3) MDA aracı içinde gerçekleştirilmiş, şablonlara göre tanımlanan bir algoritma ile iskelet PSM üretilir ve bu model elle tamamlanır.
4) Algoritmayı uygulayan araç PSM'nin tamamını üretir.

Kısıtlı bir ortam için üretilmiş bir araç tam veya neredeyse tam PSM seviye 4 modelleri üretebilir. Otomasyona mani olan faktörler olarak, eski tip uygulamaların varlığı, tamamlanmamış PIM'ler, olgunlaşmamış dönüşüm algoritmaları sayılabilir. MDA algoritmaları ve bunları gerçekleştiren araçlar olgunlaştıkça, uygulama tasarımcıları bunları kullanıkça model dönüşümünün hızla 3. seviyeye yükselmesi ve daha sonra 4. seviyeye yaklaşması beklenmektedir.

3.3 Üçüncü Adım: Uygulamanın Üretilmesi

Şekil 3.1'in üçüncü sırasında görüldüğü gibi, MDA aracı platformun gerektirdiği tüm dosyaları üretir. MDA aracı seçilen uygulama sunucusuna ve programlama diline uygun kaynak kodlarını üretir. Ayrıca uygulama sunucusunun uygulamayı istediğimiz gibi çalıştırması için konfigürasyon ve kurulum dosyalarını da üretir. Her MDA eşleştirmesi, hedef platformun gereksinimlerine göre farklı dosya tipleri üretmektedir.

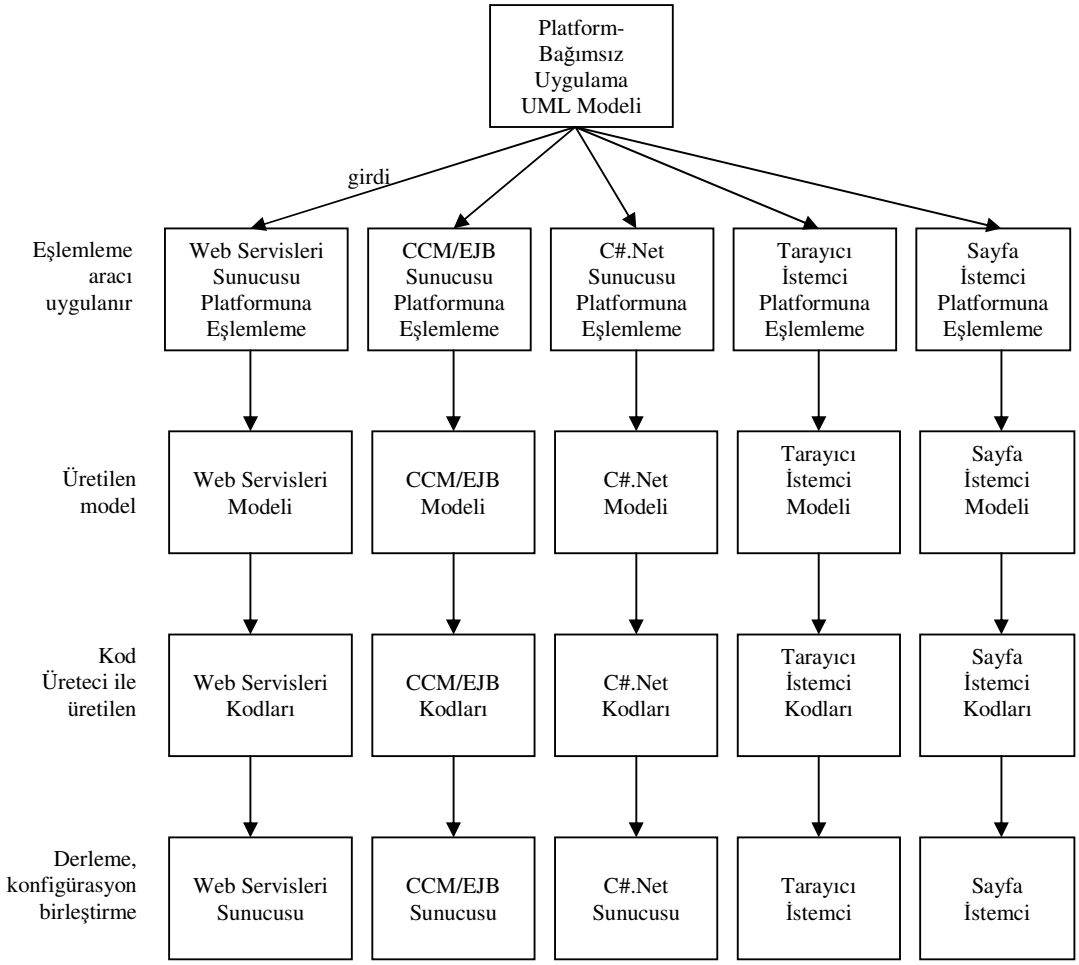
Kod üretiminden hemen sonra, programcılar üretilen koda gerekli elle müdahaleleri yapacaklardır. Takip eden derleme aşamasında, ara katman yazılımına ait bir araç, çeşitli kod dosyalarını uygun bir şekilde derleyecektir.

Uygulama sunucusu ortamlarında çalışacak MDA sunucuları ayarlanmalı ve birleştirilmelidir. MDA geliştiricileri gerekli tüm konfigürasyon bilgilerini uygulama

UML modelinde belirttiğinden bu aşama da otomatik olarak gerçekleştirilir. Böylece, MDA aracı derlenmiş dosyalarla konfigürasyon dosyalarını bir araya toplar ve sunucu böylece kurulmuş olur.

3.4 Birden Fazla Hedef Platform

Tek bir platform için modelden kod üretme kabiliyeti yeterince önemli olsa da, Şekil 3.2'de görüldüğü gibi hedef platformlar arttıkça MDA'nın faydaları katlanmaktadır.



Şekil 3.2 Diğer platformların kullanılması

Bir şirketin, aynı uygulamanın birden fazla sunucu için gerçekleştirmesini yapması olağandışı olsa da, yazılım geliştirme firmaları çeşitli işletim sistemleri ve donanımları kullanan müşterileri desteklemek amacıyla buna ihtiyaç duyar. Daha da

önemlisi hedef bağımsızlığının birlikte işlerliğe sağladığı katkıdır. Üretilen uygulamadan çağrılan Yaygın Hizmetler, Alan Hizmetleri ve MDA ortamındaki diğer uygulamalar istenirse, uygulamanın ara katman yazılımı platformu için tekrar üretilebilir.

3.5 İki Yönlü Geliştirme (Round Trip Engineering)

Çoğu geliştirme süreci iteratif olarak ilerlediği için, MDA iki yönlü geliştirme desteği önemli bir araçtır. İki yönlü geliştirme ile Geliştiriciler çalışan bir uygulamanın kodunda yaptıkları değişikliklerin MDA araçları yardımıyla UML modeli üzerine yansımasını görebilmektedirler. Aynı şekilde uygulamanın UML modeli üzerine yapılacak değişiklikler, kod üretim ve derleme aşamalarından geçerek kurulu uygulamaya tesir edebilmektedir.

3.6 Eşlemlemeler (Mappings)

MDA'da yaklaşımının en önemli özelliklerinden biri eşlemlemedir. Bir eşlemleme, bir modeli değiştirerek başka bir model elde etmeyi sağlayan kurallar ve teknikler kümesidir. Eşlemlemeler aşağıdaki dönüşümleri gerçekleştirmek amacıyla kullanılır [3]:

1. PIM -> PIM: Bu dönüşüm, geliştirme yaşam döngüsü içinde platform-bağımlı bilgiye ihtiyaç duymadan modeller geliştirildiğinde, filtrelendiğinde veya özelleştirildiğinde kullanılır. Bu dönüşüme en belirgin örnek, analizden tasarım dönüşümüdür. PIM > PIM eşlemlemeler genellikle model detaylandırma ile ilgilidir.

2. PIM -> PSM: Bu dönüşüm, PIM bir platform için tasarlanacak derecede detaylandırıldığında kullanılır. Dönüşüm platform karakteristiklerine bağlıdır. Bu karakteristiklerin tanımlanması UML ile yapılmalıdır. Mantıksal bileşen modelinden varolan bir bileşen modeline (EJB, CCM vs.) geçiş bir tür PIM > PSM eşlemlemedir.

3. PSM -> PSM: Bu dönüşüm, bileşen gerçekleştirme ve kurulumu için gereklidir. Örneğin, bileşen paketleme hizmetler seçilerek ve konfigürasyonları hazırlanarak yapılır. Paketlendiğinde, bileşen teslimi başlatma verisi, hedef platform vs. bilgileri belirtilerek yapılır. PSM > PSM eşlemleme, genellikle platform bağımlı model gerçekleştirmeyle ilgilidir.

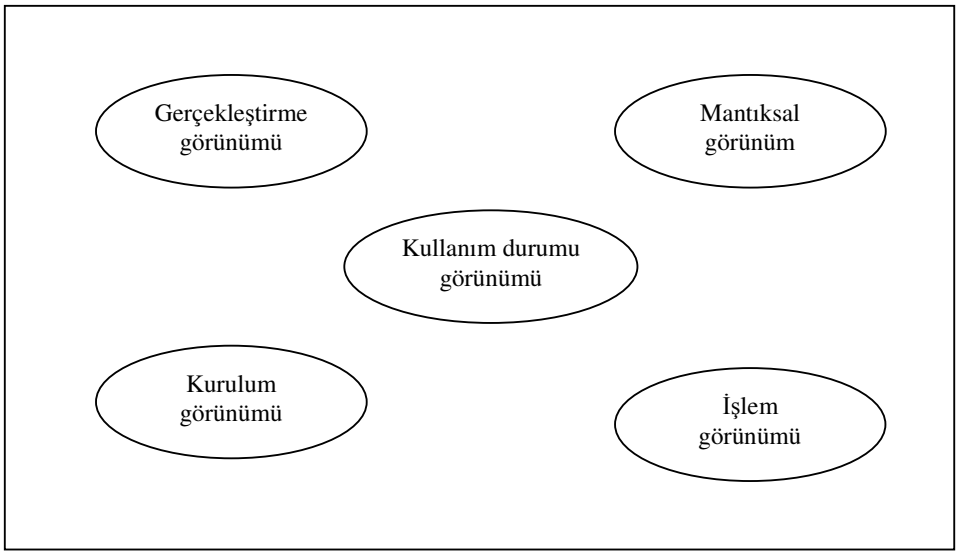
4. PSM -> PIM: Bu dönüşüm, bir teknolojiyle gerçekleştirilmiş varolan gerçekleştirmelerin platform-bağımsız modellere soyutlanması için gereklidir. Bu prosedür çoğu zaman veri gömme işlemine benzemektedir ve tamamen otomasyonu zordur.

4. UNIFIED MODELING LANGUAGE (UML)

UML, iş ve yazılımlara ait süreç ve yapıları ifade etmekte kullanılan bir modelleme dilidir. UML, statik ve dinamik davranışları olan her yapının genel mimarisini tasarlamakta kullanılabilir. UML, bir yazılım projesinde, ihtiyaçları, sistem tasarımını, kurulum talimatlarını ve program yapısını ifade etmekte kullanılabilir. Bu bölümde UML, çeşitli özellikleri açısından (görünümler, diyagramlar, model elemanları) genel olarak tanıtılacaktır. Son olarak UML'in bir parçası olan Nesne Kısıt Dili hakkında bilgi verilecektir.

4.1 Görünümler (Views)

Görünümler bir sistemi farklı yönlerden gösterir. Görünüm grafiksel bir eleman veya diyagram değil, diyagramlardan oluşan bir soyutlamadır. Her biri sistemin belirli bir yönünü gösteren çeşitli görünümler tanımlanarak, resmin tamamı oluşturulur. Ayrıca, görünümler modelleme dilini seçilen yazılım geliştirme sürecine bağlarlar [Eriksson, 2004].



Şekil 4.1 UML görünümleri

4.1.1 Kullanım Durumu Görünümü (Use-Case View)

Kullanım durumu görünümü, sistemin harici aktörler tarafından algılandığı şekilde sağlaması gereken fonksiyonalitesini tanımlamaktadır. Bir aktör sistem ile etkileşim

içindedir, aktör bir kullanıcı veya harici sistem olabilir. Kullanım durumu görünümü müşteriler, tasarımcılar, yazılım geliştiriciler ve testçiler tarafından kullanılır, kullanım durumu diyagramları ve bazen bunlara destek olan aktivite diyagramları ile gösterilir.

Kullanım durumu görünümü merkezidir, çünkü içeriği diğer görünümlerin geliştirilmesini sağlar. Sistemin nihai amacı bu görünümde anlatılan fonksiyonaliteyi sağlamaktır. Bu görünüm, ayrıca müşterilerle birlikte test edilerek sistemin beklenen şekilde çalışıp çalışmadığını anlamak için kullanılır.

4.1.2 Mantıksal Görünüm (Logical View)

Mantıksal görünüm sistemin fonksiyonalitesinin nasıl sağlandığını anlatır. Çoğunlukla tasarımcılara veya yazılım geliştiricilere yöneliktir. Kullanım-durumu görünümünden farklı olarak, mantıksal görünüm sistemin içine bakar. Hem statik yapıları (sınıflar, nesneler ve ilişkiler), hem de belirli bir fonksiyonu yerine getirmek için birbirine mesajlar gönderen nesneler arasındaki dinamik işbirliklerini anlatır. Sınıfların arayüzleri, iç yapıları, sürerlik ve eşzamanlılık gibi özellikleri de tanımlanır.

Statik yapı, sınıf ve nesne diyagramları ile anlatılır. Dinamik modelleme ise sonlu otomat (state machine), etkileşim (interaction) ve aktivite diyagramları ile ifade edilir.

4.1.3 Gerçekleştirme Görünümü (Implementation View)

Gerçekleştirme görünümü, temel modüller ve aralarındaki bağımlılıkları anlatır. Çoğunlukla yazılım geliştiriciler içindir ve temel yazılım bileşenlerinden oluşur. Bu bileşenler yapıları ve bağımlılıkları ile gösterilen farklı tiplerde kod modüllerini içerir. Bileşenler hakkındaki, kaynak tahsisi (sorumluluk), geliştirme işi için süreç raporu gibi yönetimsel ek bilgiler bu görünüme eklenebilir. Gerçekleştirme görünümü belli bir çalıştırma ortamına ait uzantıları gerektirecektir.

4.1.4 İşlem Görünümü (Process View)

İşlem görünümü sistemin işlemlere ve işlemcilere bölünmesi ile ilgilenir. Sistemin fonksiyonel özelliği olmayan bu yönü, verimli kaynak kullanımı, paralel yürütme ve ortamdaki asenkron olayların işlenmesini sağlar. Sistemi eşzamanlı çalışan izleklere (thread) bölmenin yanında, bu görünüm bu izleklerin iletişim ve senkronizasyonu ile

de ilgilenir.

Eşzamanlılığı gösteren bu görünüm, sistemin geliştirici ve entegrasyoncularına kritik bilgi sağlar. Bu görünüm dinamik diyagramlardan (sonlu otomat, etkileşim, aktivite diyagramları) ve gerçekleştirme diyagramlarından (etkileşim ve kurulum diyagramları) oluşur.

Zamanlama diyagramı ise işlem görünümü için özel bir araç sağlar. Bu diyagram, bir nesnenin durumunu zamana göre gösterir.

4.1.5 Kurulum Görünümü (Deployment View)

Kurulum görünümü, sistemin fiziksel kurulumunu, bilgisayarları, cihazları ve bunların birbirine nasıl bağlı olduğunu gösterir. İşlemcilerdeki çeşitli yürütme ortamları da belirtilebilir. Kurulum görünümü, yazılım geliştirici, entegrasyoncuları ve testçileri tarafından kullanılır ve kurulum diyagramı ile gösterilir. Bu görünüm ayrıca fiziksel mimaride bileşenlerin nasıl kurulduğunu gösteren bir eşlemlemeyi de içerir (Örneğin, her bir bilgisayarda hangi program veya nesnelerin çalıştığı gibi).

4.2 Diyagramlar

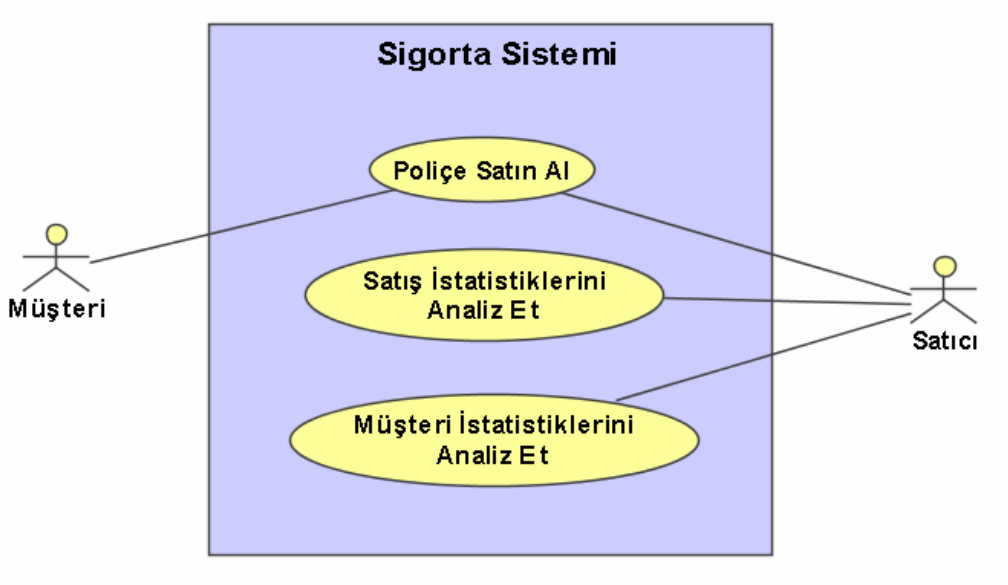
Diyagramlar sistemin belli bir kısmını veya yönünü göstermek amacıyla düzenlenmiş grafiksel elemanlardan oluşur. Bir sistem modeli tipik olarak modelin amacına bağlı olarak çeşitli türlerde farklı diyagramlar içerir. Bir diyagram belli bir görünümün parçasıdır. Bazı diyagram tipleri diyagramın içeriğine bağlı olarak birkaç görünümün de parçası olabilirler.

Bu kısımda UML 2'deki her diyagramın arkasındaki temel kavramlar anlatılacak ve her biri için basit birer örnek verilecektir.

4.2.1 Kullanım Durumu Diyagramı (Use Case Diagram)

Kullanım durumu diyagramı, harici aktörleri ve onların sistemin sağladığı kullanım durumlarına olan bağlantılarını gösterir (Şekil 4.2). Bir kullanım durumu sistemin sağladığı bir fonksiyonalitenin (sistemin belirli bir şekilde kullanımı) tanımlamasıdır. Bir kullanım durumunun tanımı düz yazı olarak veya kullanım durumuna bağlı bir doküman olarak yapılır. Fonksiyonalite veya akış aktivite diyagramı ile anlatılabilir.

Kullanım durumu tanımı sistem davranışlarını kullanıcının algıladığı şekilde anlatmalı ve sistemin içinde fonksiyonalitenin nasıl sağlandığını anlatmamalıdır. Kullanım durumları sistemin fonksiyonel gereksinimlerini tanımlar.



Şekil 4.2 Sigorta işi için kullanım durumu diyagramı

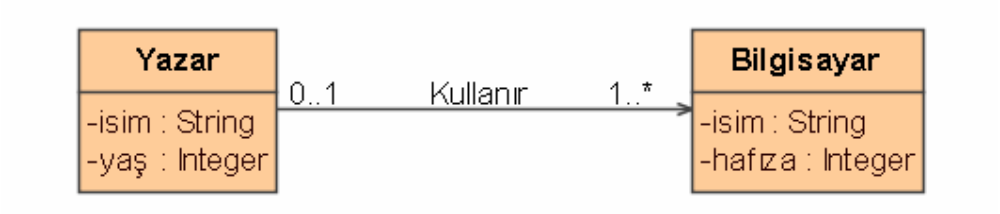
4.2.2 Sınıf Diyagramı (Class Diagram)

Sınıf diyagramı sistemdeki sınıfların statik yapılarını gösterir (Şekil 4.3). Sınıflar sistemde bulunan "şey"leri temsil eder. Sınıflar birbirleri ile birkaç şekilde ilişki içinde olabilirler:

- Birliktelik (Association) (birbirine bağlı olmak)
- Bağımlılık (Dependency) (bir sınıf diğerine bağımlı veya onu kullanıyor)
- Özelleştirme (Specialization) (bir sınıf diğerinin özelleştirmesi)
- Paketleme (Packaging) (sınıfların bir birim halinde gruplanması)

Bütün bu ilişkiler ve sınıfların özellik ve operasyonlar şeklindeki iç yapıları sınıf diyagramlarında gösterilir. Diyagram statik sayılmaktadır, çünkü anlattığı yapı sistemin yaşam döngüsünün her noktasında geçerlidir.

Bir sistem tipik olarak birden fazla sınıf diyagramı içerir. Bütün sınıflar aynı sınıf diyagramına konulmaz ve bir sınıf birden fazla sınıf diyagramında varolabilir.

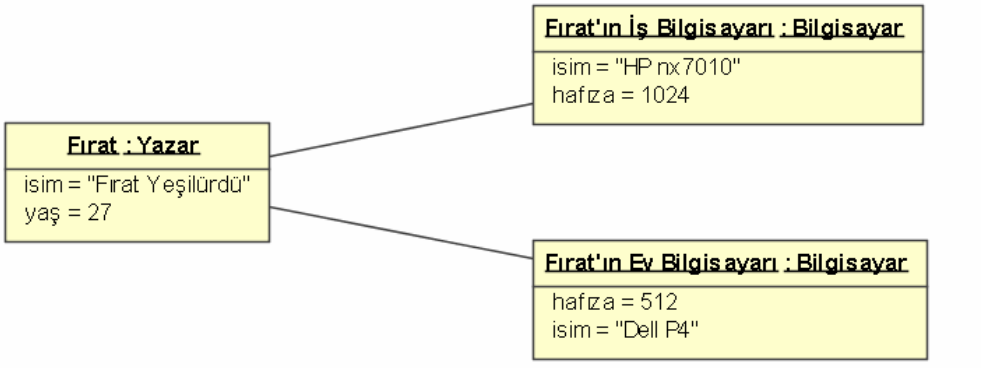


Şekil 4.3 Sınıf diyagramı örneği

4.2.3 Nesne Diyagramı (Object Diagram)

Nesne diyagramı sınıf diyagramının bir türevidir ve hemen hemen aynı notasyonu kullanır. Aralarındaki fark, nesne diyagramının sınıfların kendilerini değil, somutlaşan örneklerini (instance) göstermesidir. Nesne diyagramı, sınıf diyagramının bir örneğidir ve sistemin çalışmasında mümkün olan bir anı gösterir. Notasyonunun sınıf diyagramından iki tane farkı vardır: Nesneler, isimleri altı çizili olarak yazılırlar ve bir ilişkideki bütün örnekler (instance) gösterilir (Şekil 4.4).

Nesne diyagramları sınıf diyagramları kadar önemli değillerdir, ama karmaşık bir sınıf diyagramını örnekle açıklamak amacıyla kullanılabilirler. Nesneler, ayrıca bir nesne kümesindeki dinamik işbirliklerini gösteren etkileşim diyagramlarında da kullanılmaktadırlar.

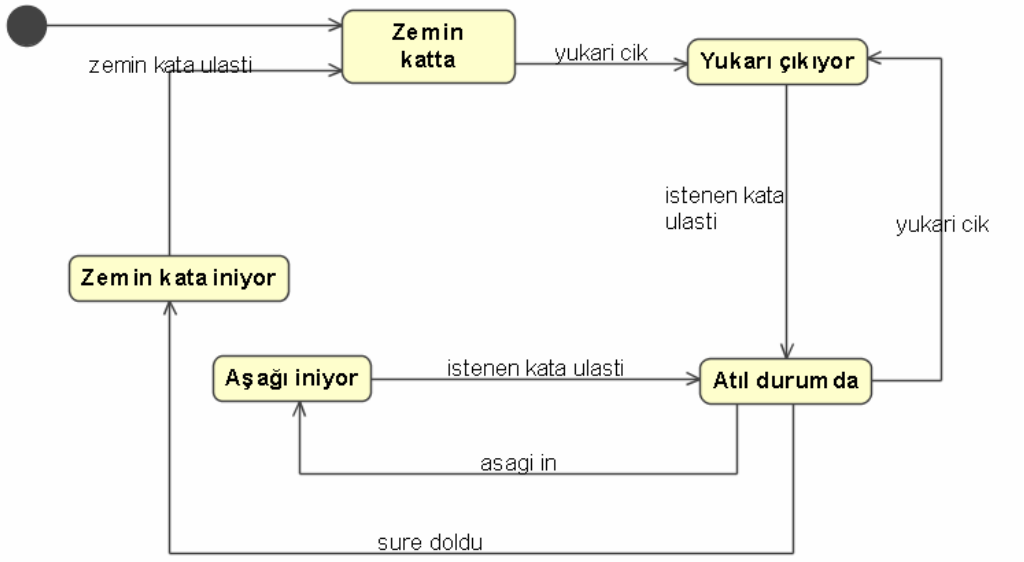


Şekil 4.4 Nesne diyagramı örneği

4.2.4 Sonlu Otomat (State Machine)

Sonlu otomat, tipik olarak bir sınıf tanımının tamamlayıcısıdır. Bir sınıfa ait

nesnelerin yaşam döngüleri boyunca içinde olabilecekleri tüm durumları ve hangi olayların durum değişikliğine sebep olduğunu gösterir (Şekil 4.5). Bir olay diğer bir nesneden gelen bir mesajla veya bir koşulun gerçekleşmesiyle tetiklenebilir. Durum değişikliklerine *geçiş* (transition) denir. Bir geçişte, ayrıca durum değişikliği ile bağlantılı olarak yapılacak bir hareket varsa bu da belirtilebilir.



Şekil 4.5 Bir asansör için davranışsal sonlu otomat

UML'de iki tip sonlu otomat vardır:

- **Davranışsal sonlu otomat** (behavioral state machine) bir sınıfın yaşam döngüsünün detaylarını tanımlar.
- **Protokol sonlu otomat** (protocol state machine) ise sadece durum değişiklikleri ve operasyonların yürütülme sırasını yönlendiren kurallara odaklanır. Protokol sonlu otomat arayüzler ve bağlantı noktaları (port) ile gerçekleştirme için kurallar sağlar. Bu kurallar, ilgili sınıfla çalışacak diğer sistemlerin uyması gereken kurallardır. Protokol sonlu otomat, farklı nesneler arasındaki iletişim için anlaşılır arayüzler ve kurallar sağlayarak UML'nin bileşen tabanlı yazılım geliştirmeyi desteklemesine yardım eder.

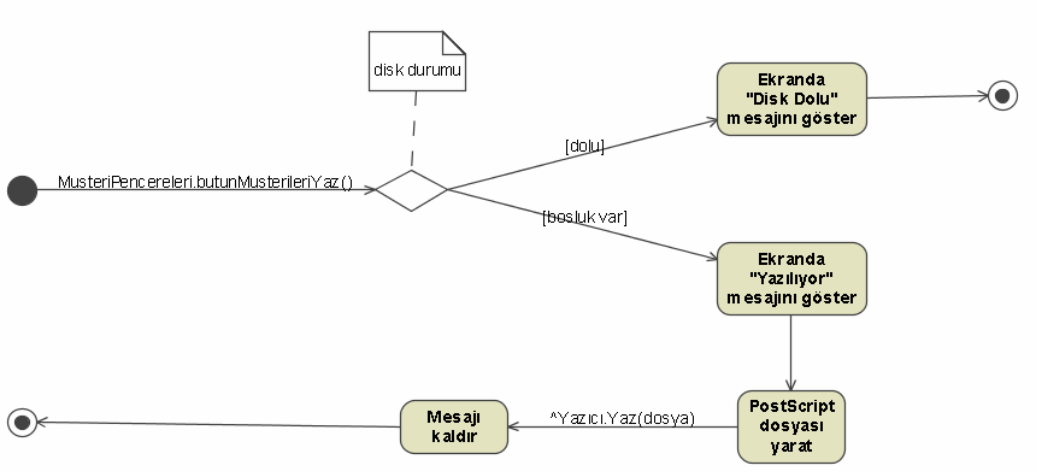
Gerçekleştirme esnasında, davranışsal sonlu otomatlar eşzamanlı davranışları modelledikleri, iç içe durumları gösterdikleri için oldukça karmaşık bir hal alabilirler. Bununla birlikte temelde sonlu otomatlar bir nesnenin durum değişiklerini göstermek

gibi nispeten basit bir düşünceyi tasvir etmektedirler.

Sonlu otomatlar bütün sınıflar için çizilmezler, sadece tanımlanmış durumları olan ve sınıfın davranışlarının durum değişlikleri ile farklılık gösterdiği sınıflar için çizilirler. Ayrıca sonlu otomatlar sistemi bir bütün olarak da tasvir edebilir.

4.2.5 Aktivite Diyagramı (Activity Diagram)

Aktivite diyagramı işlerin sıralı akışını gösterir (Şekil 4.6). Bir aktivite diyagramı tipik olarak genel bir iş akışı içinde gerçekleştirilen aktiviteleri anlatır. Bunun dışında, kullanım durumları veya detaylı kontrol akışları gibi diğer aktivite akışlarını anlatmakta da kullanılabilir.



Şekil 4.6 Bir yazıcı sunucusu için aktivite diyagramı

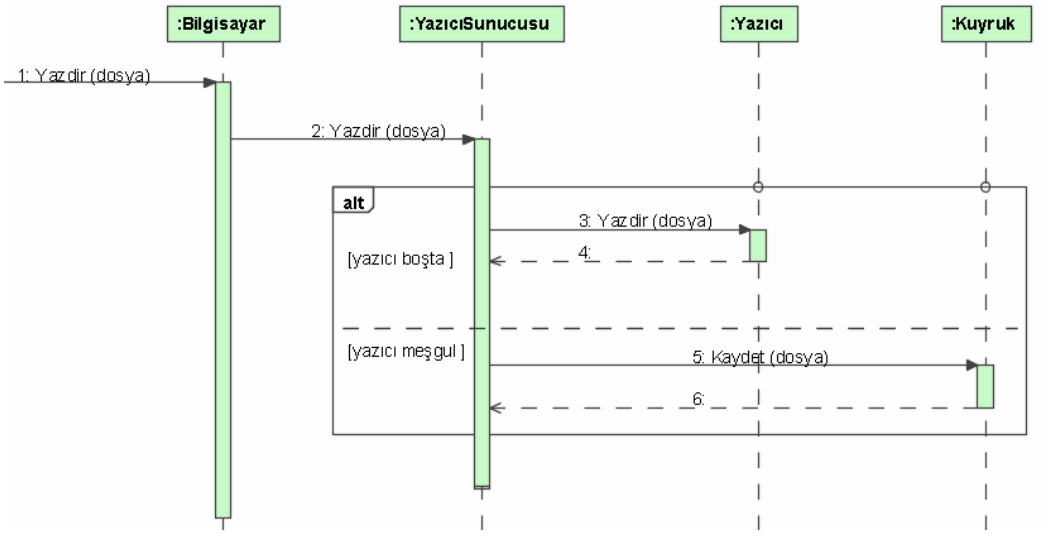
Aktivite diyagramı, bir aktiviteyi oluşturan en temel birim olan eylemlerden (action) meydana gelir. Aktivite diyagramları sistemdeki kontrol akışını göstermek amacıyla andaçları (token) kullanır. Aktivite diyagramları bu akış kontrol mekanizmasını kullanarak harici olarak tetiklenen olaylara verilen karşılıkları veya önceden belirlenmiş noktalarda (örneğin belli bir zamanda) ne yapılacağını gösterir. Diyagram, gerçekleştirilen eylemlerin bir parçası olarak mesajları ve gönderilen/alınan nesneleri de belirtebilir. Kararlar, koşullar ve paralel olarak eylemlerin yürütülmesi de diyagramda gösterilebilir.

4.2.6 Etkileşim Diyagramları (Interaction Diagrams)

UML, bir yazılımın çalışması sırasında nesneler arasındaki etkileşimi göstermek amacıyla dört çeşit diyagrama sahiptir:

4.2.6.1 Sıra Diyagramı (Sequence Diagram)

Bir sıra diyagramı, nesneler arasındaki dinamik işbirliklerini gösterir (Şekil 4.7). Bu diyagramın en önemli yönü nesneler arasında gönderilen mesajların sırasını göstermesidir. Ayrıca sistemin çalışması sırasında belli bir noktada meydana gelen nesneler arasındaki etkileşimi de gösterir. Diyagram, dikey yaşam çizgileri (lifeline) ile gösterilen nesnelerden oluşur. Diyagramda aşağı doğru gidildikçe zaman ilerlemekte ve sıra veya fonksiyon içinde zaman ilerledikçe karşılıklı gönderilen/alınan mesajlar gösterilmektedir. Mesajlar dikey yaşam çizgilerin arasında oklarla gösterilir. Zaman şartları diyagramda kısıt olarak gösterilebilir. Yorumlar (comment) yazı olarak diyagramın kenarlarına eklenebilir. Bir sıra diyagramı bir etkileşim parçasını tasvir eder. Bu parçalar üst köşelerinde yazılabilen operatörler sayesinde o bölüm için geçerli olan özel bir durumu ifade edebilir. Örneğin Şekil 4.7'de alternatif (alt) operatörü, etkileşimin baskıyı yapmak veya baskı kuyruğunda beklemek gibi seçenekleri olduğunu gösterir.

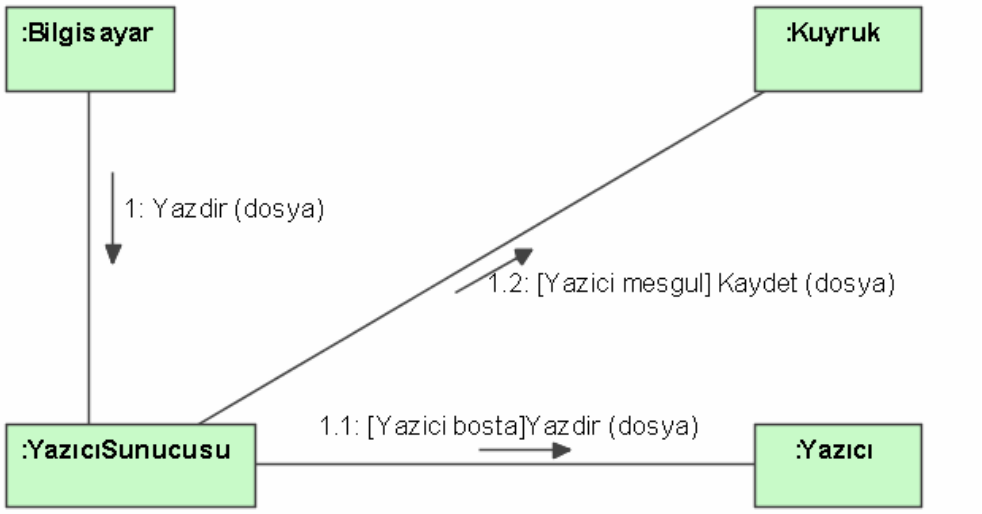


Şekil 4.7 Bir yazıcı sunucusu için sıra diyagramı

4.2.6.2 İletişim Diyagramı (Communication Diagram)

Bir iletişim diyagramı, temel sıra diyagramındaki etkileşim parçası gibi dinamik bir işbirliğini gösterir. Mesaj alışverişinin yanı sıra, iletişim diyagramı nesneleri ve aralarındaki ilişkileri (bağlam) de gösterir. Sıra diyagramı veya iletişim diyagramından hangisinin kullanılacağına genellikle amaca göre karar verilir. Zaman veya sıra vurgulanmak isteniyorsa ve birden fazla etkileşim parçası gösterilecekse sıra diyagramı seçilir. Eğer bağlam vurgulanmak isteniyorsa etkileşim diyagramı seçilir. Nesneler arasındaki etkileşim her iki diyagramda da gösterilir.

İletişim diyagramı nesneleri aralarındaki ilişkilerle birlikte gösterir.Mesaj akışını göstermek amacıyla nesneler arasına mesaj okları çizilir. Mesajların üzerine mesajların gönderilme sırasını da belirten etiketler konur. Koşullar, iterasyonlar, geri dönüş değerleri de gösterilebilir. Bununla birlikte, iletişim diyagramı bir mesajın alınma sırasının bilinmediği durumları desteklememektedir. Şekil 4.8'de örnek bir iletişim diyagramı gösterilmektedir.



Şekil 4.8 Bir yazıcı sunucusu için iletişim diyagramı

4.2.6.3 Özet Etkileşim Diyagramı (Interaction Overview Diagram)

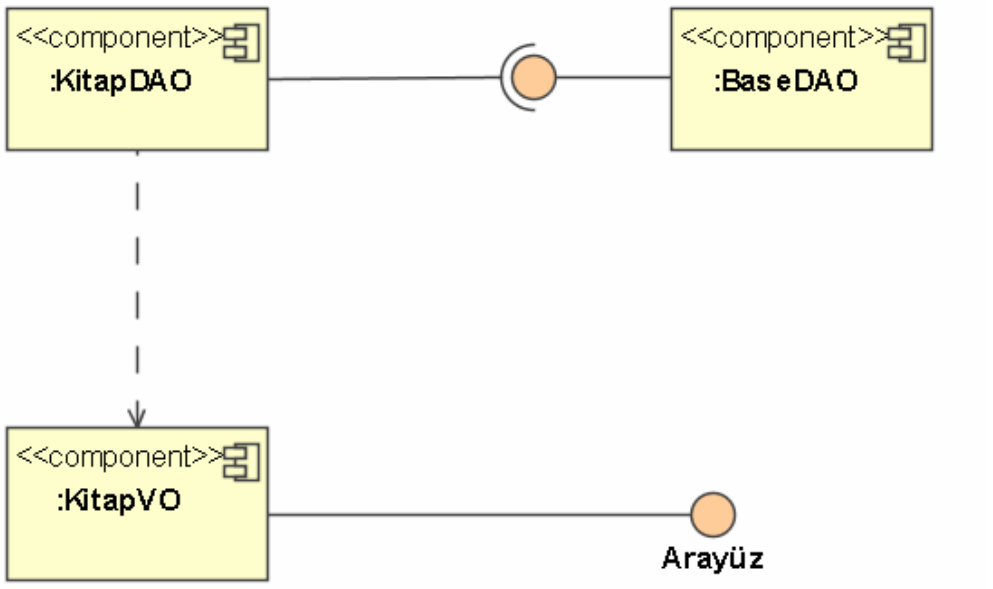
Özet etkileşim diyagramı model geliştiriciye etkileşimlerin ana akışını üst seviyeli olarak gözden geçirebilme olanağı sağlar. Bu diyagram, yapılan tasarımın kullanım

durumunda tanımlanan bütün akış elemanlarını içerip içermediğini anlamakta oldukça faydalıdır.

Bir özet etkileşim diyagramı temel olarak, ana düğümleri etkileşim parçaları veya sıra diyagramı parçaları ile değiştirilmiş bir aktivite diyagramıdır. Ayrıca diyagram, etkileşim boyunca akış kontrolünü göstermek için başka bir yol sunmuş da olmaktadır.

4.2.7 Bileşen Diyagramı (Component Diagram)

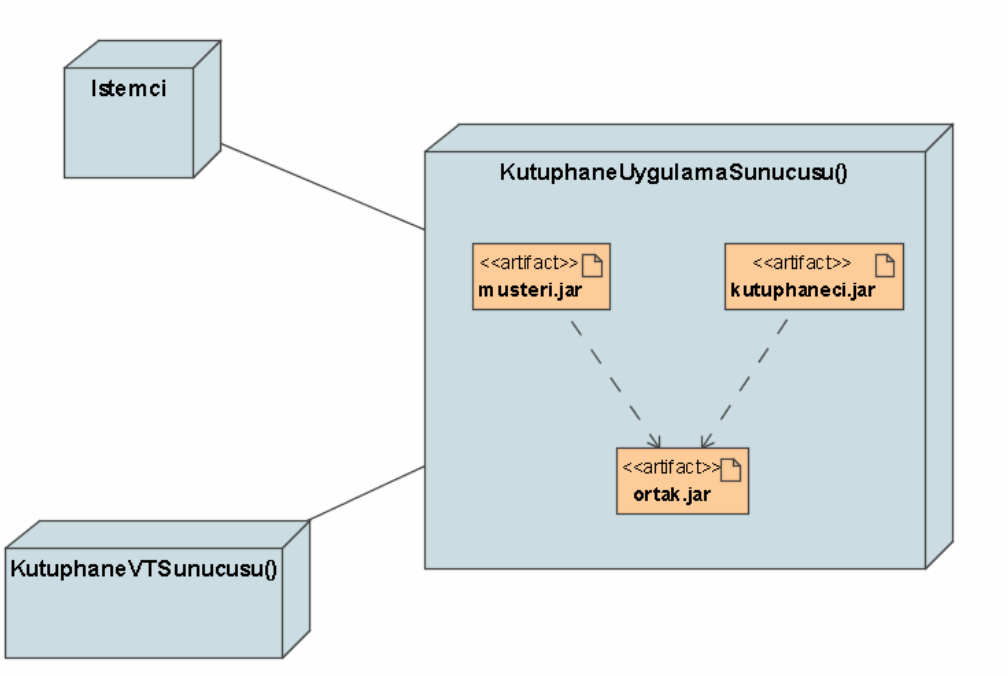
Bir bileşen diyagramı kod bileşenleri olarak kodun fiziksel yapısını gösterir. Bileşen; kaynak kodu bileşeni, ikili (binary) bileşen veya çalıştırabilir bir bileşen olabilir. Bir bileşen gerçekleştirdiği mantıksal sınıf veya sınıflar hakkında bilgiler içerir. Böylece mantıksal görünümden bileşen görünümüne bir eşlemleme yaratılmış olur. Bileşenler arasındaki bağımlılıklar gösterilir, böylece bir bileşende yapılan değişiklikten başka hangi bileşenlerin etkileneceği kolay bir şekilde analiz edilebilir. Bileşenler sundukları arayüzlerle de gösterilebilir (örneğin OLE/COM) veya paketler halinde gruplanabilirler. Bileşen diyagramı programlama işleri için kullanılmaktadır (Şekil 4.9).



Şekil 4.9 Kod bileşenleri arasındaki bağımlılıkları gösteren bir bileşen diyagramı

4.2.8 Kurulum Diyagramı (Deployment Diagram)

Kurulum diyagramı, sistemdeki yazılım ve donanımın fiziksel mimarisini göstermektedir. Bilgisayarlar ve cihazlar (düğümler), bunların birbirleri arasındaki bağlantılar, ve bağlantıların türleri gösterilebilmektedir. Düğümlerin içindeki çalıştırılabilir bileşenler ve nesneler, hangi yazılım biriminin hangi düğümde çalıştığını göstermektedir. Ayrıca bileşenler arasındaki bağımlılıklarda gösterilebilir (Şekil 4.10).



Şekil 4.10 Bir sistemin fiziksel mimarisini gösteren kurulum diyagramı

Kurulum görünümüne ait olan kurulum diyagramı, sistemin gerçek fiziksel mimarisini anlatmaktadır. Bu görünüm kullanım-durumu görünümündeki fonksiyonel tanımdan oldukça uzaktır. Bununla birlikte, iyi tanımlanmış bir modelde, fiziksel mimaride yer alan bir düğümden bileşenlerine, bileşenlerden gerçekleştirdiği sınıflara, sınıflardan sınıfa ait nesnelerin yer aldığı etkileşimlere ve buradan da ilgili kullanım durumuna ulaşmak mümkündür. Sistemin farklı görünümleri, sistemin bir bütün olarak tutarlı bir tanımlamasını vermeye yaramaktadır.

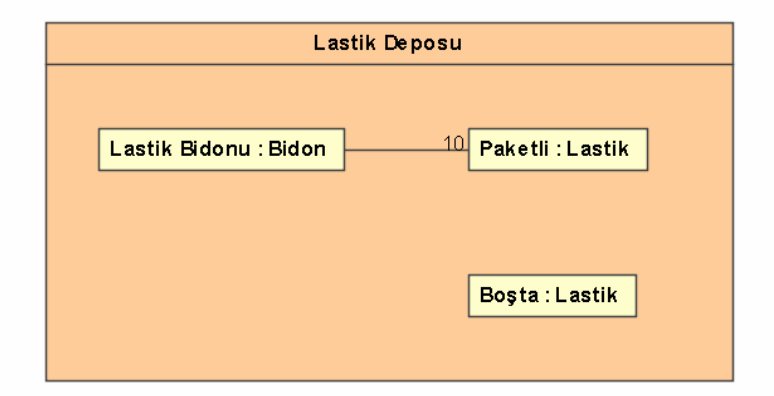
4.2.9 Bileşik Yapı Diyagramı (Composite Structure Diagram)

İyi tanımlanmış bir model elemanların rollerini ve sorumluluklarını açık bir şekilde

açıklamaktadır. Ancak yürütme zamanına ait mimariler tasarım zamanda bilinememektedir.

Örneğin, bazı sistemler tipik nesne ve sınıf diyagramlarıyla açıkça anlaşılamayan yürütme zamanı mimarileri ortaya koyarlar. Elemanlara ait belirli işbirlikleri, diğer statik diyagramlardakinden farklı ilişki ve kurallar içerebilirler. Bu sorunları çözebilmek için UML 2'de, ilgili elemanları ve ilişkileri belirli bir sınıflandırıcının (kullanım durumu, nesne, işbirliği, sınıf, aktivite) bağlamında gösteren bileşik yapı diyagramı eklenmiştir.

Örneğin Şekil 4.11'deki bileşik yapı diyagramı lastik depolama ile ilgili elemanları göstermektedir. Diyagrama göre, 10 lastik bir depolama bidonuna konmaktadır. Sistem ayrıca herhangi bir bidona girmeyen boştaki lastiklere de izin vermektedir. Bu lastikler başka bir bağlamda farklı ilişkilere sahip olabilirler. Örneğin lastik, bir envanter elamanı olmak yerine bir aracın parçası olsaydı, aynı bağlantılara sahip olmazdı.

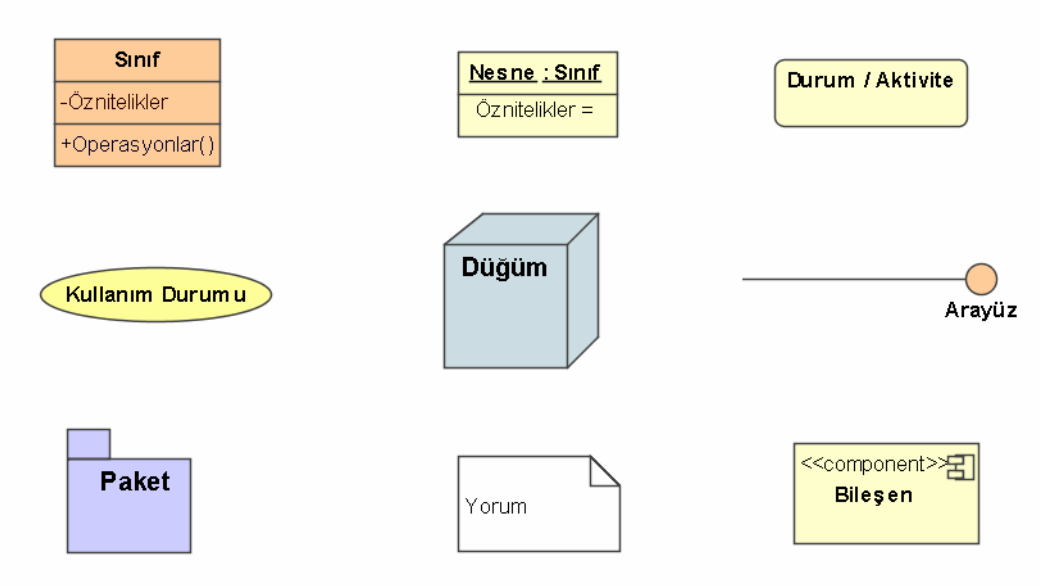


Şekil 4.11 Bir lastik depolama sistemin için örnek bileşik yapı diyagramı

4.3 Model Elemanları

Diyagramlarda kullanılan kavramlara model elemanları denir. Bir model elemanı, elemanın formal bir tanımı veya temsil ettiği şeyin tam anlamı (semantics) ile tanımlanır. Bir model elemanına karşılık gelen, onu diyagramlarda temsil eden bir görsel simge veya grafiksel sembol elemanı olabilir. Bir eleman farklı birkaç tip diyagramda varolabilir, ancak her tip diyagram için hangi elemanların gösterilebileceğine dair kurallar vardır. Örnek model elemanları Şekil 4.12'de

gösterilmiştir.



Şekil 4.12 Model elemanlarından örnekler

Şekil 4.13'te, kendileri de model elemanı olan ve model elemanlarını birbirine bağlamakta kullanılan *ilişki* örnekleri yer almaktadır:

Şekil 4.13 İlişki tipi örnekleri

- **Birliktelik (Association):** Elemanları birbirine bağlar, örnekler arasında bağ kurar.
- **Genelleştirme (Generalization):** Kalıtım (inheritance) da denir, bir elemanın başka bir elemanın özel hali olması anlamına gelir.
- **Bağımlılık (Dependency):** Bir elemanın bir biçimde diğer bir elemana bağımlı olduğunu gösterir.

- **İçerme (Aggregation):** Bir elemanın diğer elemanları içermesi şeklindeki bir *birliktelik* türü.

4.4 Object Constraint Language (OCL) (Nesne Kısıt Dili)

Nesne Kısıt Dili (OCL) yazılım modelleri geliştirmekte kullanılabilecek bir modelleme dilidir. UML'ye standart bir ek olarak tanımlanmıştır. OCL ile yazılan her ifade UML diyagramlarında tanımlanan tiplere (sınıf, arayüz vs.) bağımlıdır.

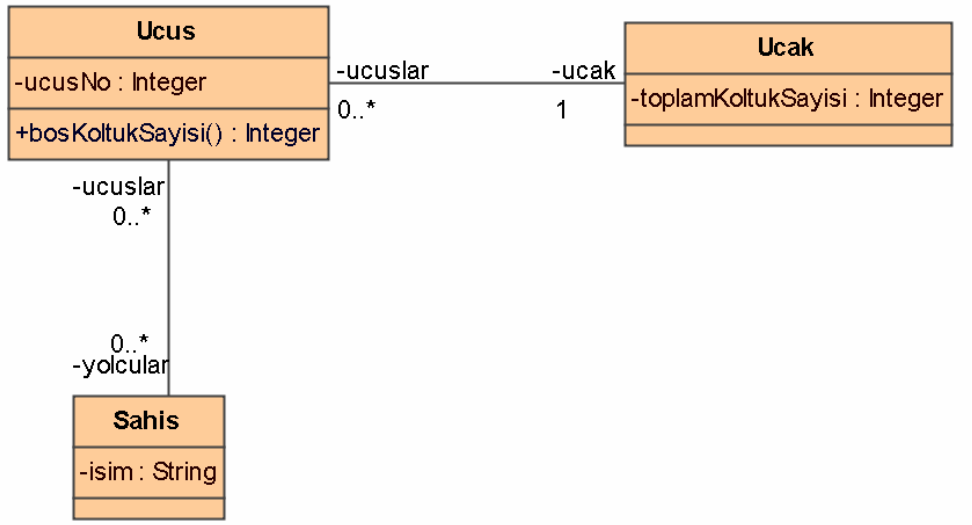
OCL ile yazılan ifadeler nesne-yönelimli modellere çok önemli bilgiler eklemektedirler. Bu bilgiler çoğunlukla bir diyagram ile ifade edilemeyecek cinstendir. UML 1.1'de, bu bilgilerin sadece kısıtlardan ibaret olduğu düşünülmüştü. Kısıt, nesne-yönelimli bir modele ait bir veya daha fazla değer üzerine getirilen bir sınırlama olarak tanımlanmıştı. UML 2'de ise, kısıtlar haricinde ek bilgilerin de modelde bulunmasına karar verilmiştir. İfadeler yazarak, sorgu tanımlamak, değerleri referans etmek, koşul ve iş kuralları yazmak mümkündür. OCL bu ifadelerin muğlak olmayan ve açık bir şekilde yazılabileceği standart dildir [Warmer, 2003].

4.4.1 OCL ve UML'in Birlikte Kullanımı

Yazılım modelleme, geleneksel olarak hep diyagram çizmek olarak düşünülmüştür. Çoğu model bir takım şekiller, oklar ve bunlarla ilgili metinlerden oluşmaktadır. Böyle modeller ile ifade edilen bilgilerin eksik, informal, kusurlu ve hatta tutarsız olma eğilimleri bulunmaktadır.

Modellerdeki birçok kusur kullanılan diyagramların sınırlamalarından kaynaklanmaktadır. Bir diyagram eksiksiz bir şartnamenin parçası olan ifadeleri gösteremez. Örneğin, Şekil 4.14'de gösterilen UML modelindeki, bir grup insanın uçağın yolcusu olduğunu gösteren *Uçuş* ve *Şahıs* sınıflarının arasındaki birliktelik ilişkisi, *Şahıs* sınıfı tarafında (0..*) katlılığına (multiplicity) sahiptir. Bu, yolcu sayısının sonsuz olduğu anlamına gelmektedir. Oysa, yolcu sayısı uçuş için kullanılan uçağın koltuk sayısı ile sınırlıdır. Bu sınırlamayı diyagram üzerinde göstermek mümkün değildir. Bu örnekte, katlılığı doğru bir şekilde belirtmenin yolu diyagrama aşağıdaki OCL kısıtını eklemektir:

```
context Ucus
inv: yolcular -> size() < ucak.koltukSayisi
```



Şekil 4.14 Diyagramla ifade edilmiş bir model

OCL gibi matematiksel olarak tam bir dil ile yazılan ifadelerin diyagramlarla birlikte bir sistemi tanımlamakta kullanılmasının bir çok faydası bulunmaktadır. Örneğin, bu ifadeler farklı kişiler –mesela bir analist ve programcı– tarafından farklı biçimde yorumlanamazlar. Muğlak değildirler ve modeli daha kusursuz ve detaylı hale getirirler. Otomatik araçlarla, bu ifadelerin doğru olup olmadıkları ve modeldeki diğer elemanlarla tutarlı olup olmadıkları kontrol edilebilir. Böylece otomatik kod oluşturma daha güçlü bir hale gelmektedir.

Bununla birlikte, sadece metinsel ifadeler kullanan bir dil ile yazılan bir model çoğu zaman kolay anlaşılabilir olamamaktadır. Örneğin, kaynak kod yazılımın nihai modeli olarak görülse de, bir çok insan sisteme diyagramlardan oluşan modeller ile bakmayı tercih etmektedir. Diyagramların iyi yanı, kastedilen anlamın kolayca kavranmasını sağlamalarıdır.

UML ve OCL'nin birleşimi yazılım geliştiriciye en iyi çözümü sunmaktadır. Model tanımlanırken, OCL ile yazılmış ifadeler içeren çeşitli diyagramlar kullanılabilir. Tam bir model için hem diyagramlar, hem de OCL ifadeleri gereklidir. OCL ifadeleri olmadan, model eksik kalacaktır. UML diyagramları olmadan, OCL'de sınıf ve ilişkileri belirtmenin bir yolu olmadığı için, OCL ifadeleri var olmayan model elemanlarına ait olacaktır.

4.4.2 OCL İfadelerinin Yapısı

Her OCL ifadesinin bir bağlamı (context) vardır, ve bu sınıflandırıcının (classifier) adıdır. *self* kelimesi ise sınıfın örneğini ifade eder. En çok kullanılan OCL yapıları değişmezler, önkoşullar ve sonkoşullardır. Bunlar çoğunlukla birlikte konrat ile programlamayı gerçekleştirmek amacıyla birlikte kullanılırlar.

Değişmez (Invariant): Değişmez bir tipe uygulanır. Bir tipe ait örneğin yaşam süresi boyunca muhafaza ettiği bir özelliği belirtir. Bu özellik, çoğunlukla örnek için geçerli olması gereken bir çeşit koşuldur. Bazı diller (örneğin Eiffel), değişmezleri doğrudan kullanabilmektedir, böylece değişmezler doğrudan nihai kodda gerçekleştirilebilirler.

Örneğin bir arabanın hızının hiç bir zaman negatif olamaması bir değişmezdir. OCL'de bu değişmez aşağıdaki gibi gösterilebilir.

```
context Araba
inv: hiz >= 0
```

Önkoşul (Precondition): Önkoşul bir operasyona uygulanır. Bir operasyon çağrılmadan önce doğru olması gereken bir koşuldur. Örneğin, sonSkoruHesapla() operasyonuna eklenen bir önkoşul, aşağıdaki şekilde operasyonun ancak şahıs oyunu tamamlamış ise çağrılabileceğini belirtebilir:

```
context Oyuncu::sonSkoruHesapla():: Integer
pre: self.tamamlandi = true
```

Sonkoşul (Postcondition): Sonkoşul bir operasyona uygulanır. Bir operasyon çağrıldıktan sonra doğru olması gereken bir koşuldur. Örneğin, oyuncu için oyundaki olayları işleyen ve oyuncunun seçebileceği, kalan işlenmemiş seçeneklerin sayısı değerini dönen bir operasyona ait sonkoşul aşağıdaki şekilde belirtilebilir (*result* kelimesi ifadenin geri dönüş değerini belirtmektedir):

```
context OyunOlayi::oyuncuSecenekleriniIsle():: Integer
post: result = 0
```

5. META OBJECT FACILITY (MOF)

Bu bölümde MDA'nın en önemli yapıtaşlarından biri olan MOF anlatılacaktır. Ayrıca XMI, CWM ve bunların MOF ile olan ilişkilerine değinilecektir.

MOF, birden çok model çeşidi olduğuna göre birden fazla modelleme diline de ihtiyaç olabildiği dayanak noktası alınarak 1997'de ortaya çıkmıştır.

Farklı fonksiyonlar için farklı modelleme yapıları gerekmektedir. İlişkisel veritabanı modellerken tablo, kolon, anahtar vb. yapılar kullanılır. İş akışı modellenirken aktivite, icracı, geçiş, ayrım, kesişim gibi modelleme yapıları kullanılır. UML sınıf modellemesi yaparken sınıf, özellik, operasyon, birliktelik gibi yapılar kullanılır.

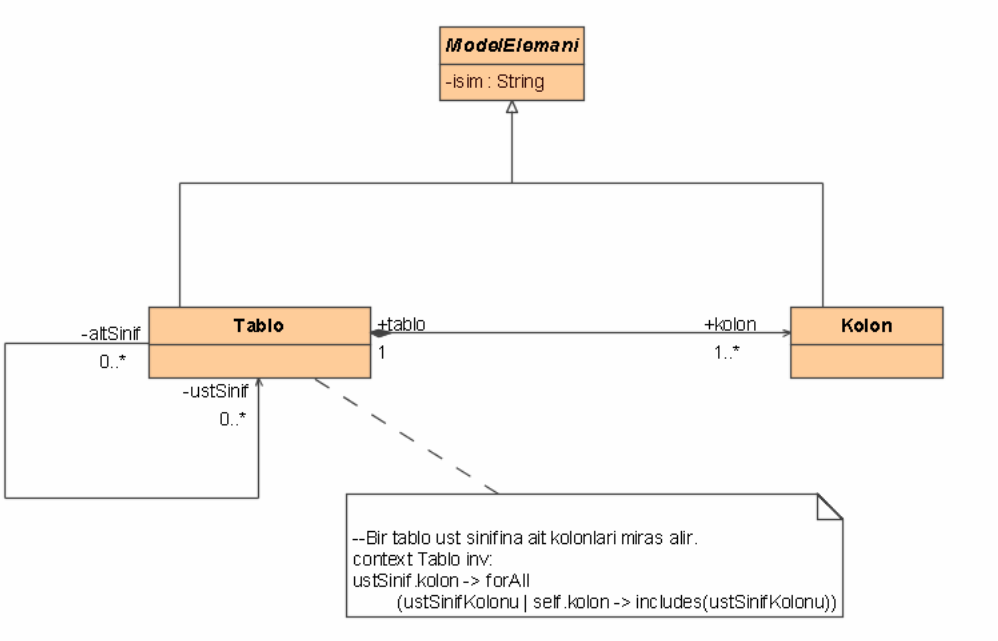
Belli bir çeşitte model yaratırken, o çeşitteki modelleri oluşturan modelleme yapıları kümesini tanımlamak gerekmektedir. Örneğin, bir veritabanı modelinde kolonların bir tabloya ait olduğunun gösterimiyle, bir nesne-yönelimli modelde operasyonların bir sınıfa ait olduğunun gösterimi aynı değildir.

MOF tasarlanırken, bu çeşitteki farklı modelleme kümelerinin tek bir kümede birleştirilmesi ve dolayısıyla bütün dillerin tek bir dile dönüştürülmesi yaklaşımı benimsenmemiştir. Bunun yerine, farklı türdeki modelleme yapılarının evrensel bir şekilde tanımlanabileceği bir yol seçilmiştir.

MOF ilişkisel veritabanlarını modellemede, UML sınıf modellemede ve başka modelleme alanlarında da kullanılmaktadır.

MOF, UML'deki nesne yönelimli sınıf modelleme yapılarını kullanır ve bunları modelleme yapılarının soyut sentaksını, yani bir metamodelin soyut sentaksını tanımlamak için kullanılacak genel bir araç olarak sunar. Böylece MOF metamodelleri UML sınıf modellerine benzemektedir. MOF ile, bir modelleme yapısı sınıf olarak ve yapının özellikleri de sınıfın özellikleri olarak düşünülebilir. Yapılar arasındaki ilişkiler birliktelik olarak modellenir. Metamodelin detayını artırmak için OCL kullanılabilir.

Aşağıdaki örnekte (Şekil 5.1) veri modelleme için basit bir metamodel görülmektedir. MOF'un altsınıf tanımlama yeteneği kullanılarak *Tablo* ve *Kolon*'un *isim* adında bir ortak özellikleri olduğu gösterilmiştir.



Şekil 5.1 Veri modelleme için MOF ile geliştirilmiş basit bir metamodel

Ayrıca bir tablonun üstsınıf ve altsınıfları olabileceği de gösterilmiştir. Buradaki önemli nokta, *Tablo* için metamodelde nesne-yönelimli tarzda altsınıflama kullanılmasına karşın, bu metamodele uygun bir veri modelinde Tablo'ya ait altsınıfların tanımlanabilmesi, metamodeldeki altSınıf, üstSınıf birliktelikleri ile mümkün olabilmektedir.

Bir OCL değişmezi ile bir tablonun, üstsınıflarının bütün kolonlarını miras aldığı da belirtilmiştir. Böylece bu metamodel, tabloların nesne-yönelimli tarzda üstsınıf kullanmasını desteklemektedir. Birlikteliğin üstSınıf ucu 0..* yapılarak metamodelin çoklu kalıtımı (multiple inheritance) desteklemesi sağlanmıştır. Eğer bu uç 0..1 yapılsaydı, metamodel sadece tekli kalıtımı destekleyecekti.

Metamodelde kullanılan altSınıf ve ustSınıf isimlerinin hiç bir ayrıcalığı yoktur. Bu metamodelin nesne-yönelimli altsınıflamayı desteklemesini sağlayan, özelliklerin ve kalıtım değişmezinin varlığıdır.

Altsınıflamayı tanımlamak başka şekilde de mümkündür. Örneğin UML metamodeli *Generalization* (genelleme) adında ayrı bir sınıf ile bunu gerçekleştirmektedir. MOF altsınıflamasını kullanan ama, kendisi altsınıflamayı veya diğer nesne-yönelimli mekanizmaları desteklemeyen metamodeller de tanımlanabilir. MOF'un önemli

mimari amaçlarından biri, birbirinden oldukça farklı modelleme olanaklarını (nesne-yönelimli olmayanları da) desteklemektir.

5.1 MDA Anlam Seviyeleri (Metalevel) ve Farklı Soyutlama Seviyelerinde Modelleme

Çizelge 5.1 MDA Anlam Seviyeleri

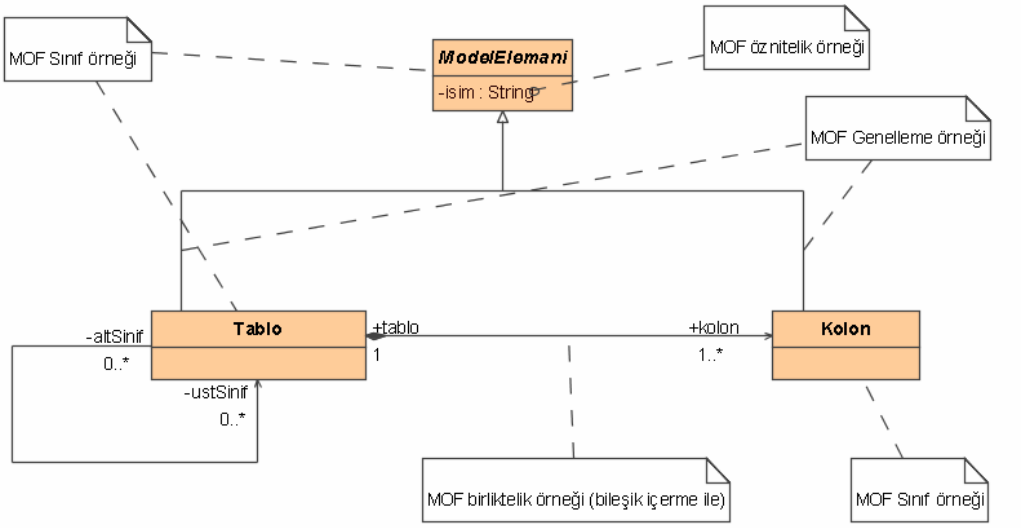
ANLAM SEVİYESİ	AÇIKLAMA	ELEMANLAR
M3	MOF; metamodel tanımlamakta kullanılan yapıları içerir	MOF sınıfı, MOF özniteliği, MOF birliktelik vb.
M2	Metamodel'ler; MOF yapılarının örneklerinden (instance) oluşmaktadır.	UML sınıfı, UML birliktelik, UML özniteliği, UML durumu, UML aktivitesi vb. CWM tablosu, CWM kolonu vb.
M1	Modeller; M2 metamodel yapılarının örneklerinden oluşmaktadır.	"Müşteri" sınıfı, "Hesap" sınıfı "Çalışan" tablosu, "Satıcı" tablosu vb.
M0	Nesneler ve veri; M1 model yapılarının örnekleridir.	Müşteri Fırat Yeşilürdü, Hesap 2989, Hesap 2344, Çalışan A3949, Satıcı 78988, vb.

M3 Seviyesi

M3 seviyesinde MOF yer almaktadır. Elemanlar MOF'un metamodel tanımlama yapılarıdır (Sınıf, özellik, birliktelik gibi). Kavramsal olarak bir tek MOF vardır. MOF'a *meta-metamodel* de denmektedir, çünkü esasında MOF bir metamodelin ne yapıda olduğunun modelidir. Burada iki tane meta sözcüğü karışıklık yaratsa da, bu kavram teknik olarak doğrudur. MOF, kendi elemanları kullanılarak yani yine MOF ile tanımlanmıştır.

M2 Seviyesi

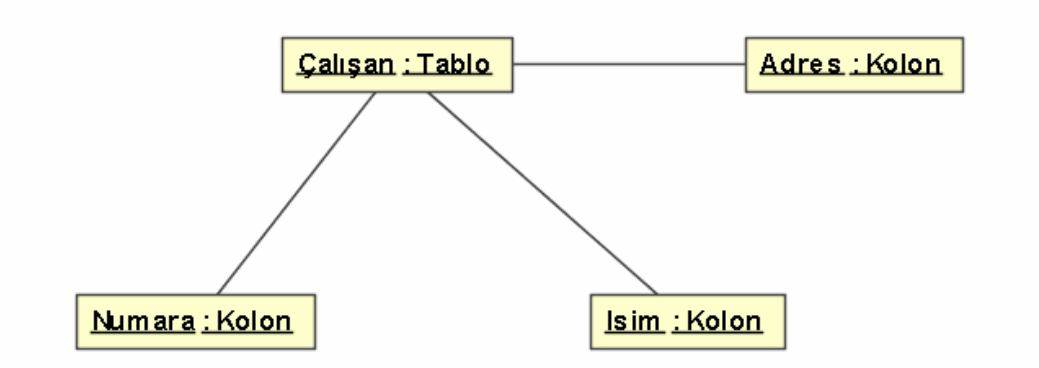
M2 seviyesi MOF yapıları kullanılarak tanımlanmış metamodelleri içermektedir. Standard metamodeller olan UML, Common Warehouse Metamodel, CORBA Component Model veya bir önceki kısımda örnek olarak gösterilen basit veri modelleme metamodeli bunlar arasındadır. Bu metamodellere ait yapılar MOF sınıfı, MOF özniteliği, MOF birliktelikleri ve diğer MOF elemanları ile tanımlanmıştır. Yani, M2 yapıları aslında M3 yapılarının bir örneğidir. Başka bir deyişle, Şekil 5.2'de gösterildiği gibi, metamodellerle tanımlanan yapılar MOF yapılarının örnekleridir.



Şekil 5.2 M3 yapılarının örneği olarak M2 metamodel yapıları

M1 Seviyesi

M1 seviyesi, M2 yapılarının örneklerinden meydana gelen modelleri içermektedir. Şekil 5.3'te M2 yapılarının örneklerinden meydana gelen bir model örneği olarak, bir önceki kısımdaki basit veri metamodeli ile geliştirilmiş bir UML nesne diyagramı görülmektedir.



Şekil 5.3 M2 veri metamodel elemanlarının örneklerinden oluşan M1 veri model elemanları

Örnek model, *Numara, İsim* ve *Adres* kolonlarını içeren *Çalışan* adında bir tablodan oluşmuş bir veri modeli tanımlamaktadır. Bu tablo metamodeldeki M2 *Tablo* elemanının örneği, kolonları da M2 *Kolon* elemanın örnekleridir. *Çalışan* tablosu ve kolonları arasındaki bağlantılar ise, metamodelde tanımlanan *Tablo* ve *Kolon* arasındaki M2 birlikteliğin örnekleridir. Buna göre:

- Böylece *Çalışan* (M1), MOF *Sınıf* (M3) örneği olan *Tablo* (M2)'nun bir örneğidir.
- Adres (M1), MOF *Sınıf* (M3) örneği olan *Kolon* (M2)'un bir örneğidir.
- Çalışan ve Adres (M1) arasındaki bağlantı, MOF *Birliktelik* (M3) örneği olan *Tablo* ve *Kolon* arasındaki birlikteliğin (M2) bir örneğidir.

M0 Seviyesi

M0 seviyesi, M1 elemanlarının örnekleri olan nesneleri ve verileri içermektedir. Veri modeli örneğine göre, "A3949" çalışan numaralı, "Fırat Yeşilürdü" adında "Yeniköy-İstanbul" adresli bir çalışanı ele alırsak, bu M1 elemanı olan *Çalışan*'ın bir örneği olan bir M0 elemanıdır.

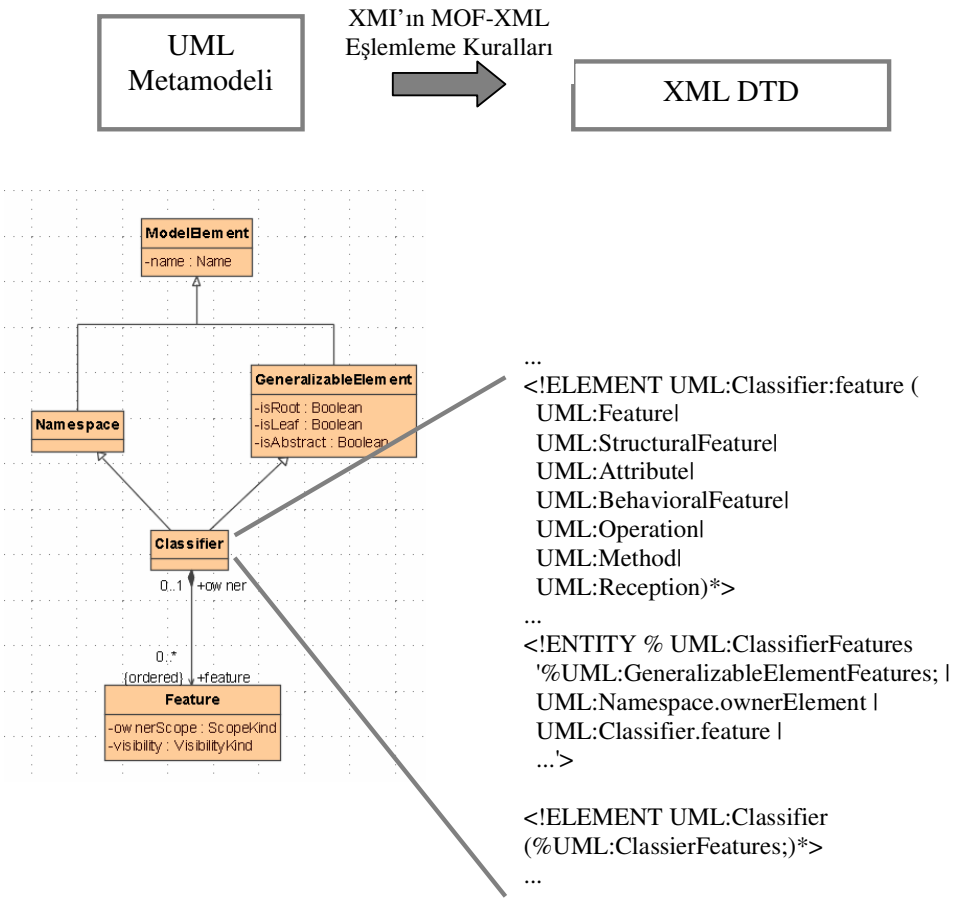
Yanı bu çalışan (M0), MOF *Sınıf* (M3) örneği olan *Tablo* (M2)'nun bir örneği olan *Çalışan* (M1)'ın bir örneğidir.

Çalışanın numarası olan A3949 da, MOF *Sınıf* (M3) örneği olan *Kolon* (M2)'un bir örneği olan *Numara* (M1)'nın bir örneğidir.

5.2 XML Metadata Interchange (XMI)

XML'in kullanımı arttıkça, metaverileri XML dokümanı olarak göstermek gibi bir ihtiyaç doğmuştur. XML standart bir yolla modelleri araçlar arasında değiş-tokuş edebilmek için mükemmel bir vasıta sunmuştur. 1998 yılı sonlarında da, OMG bir MOF-XML eşlemlemesi olan XML Metadata Interchange'i benimsemiştir.

OMG XMI'i, UML metamodeline uyguladığında, ortaya UML modellerinin değiş-tokuş edilmesini sağlayan bir DTD (Document Type Definition) çıkmıştır. Şekil 5.4'de XMI'ın UML metamodelindeki *Classifier* (Sınıflandırıcı) elemanına nasıl uygulandığı görülmektedir. Şekilde sadece *Classifier*'ı için üretilmiş DTD elemanının bir kısmı gösterilmiştir [4].



Şekil 5.4 XMI'ın MOF-XML eşlemleme kurallarının UML metamodeline uygulanması

XMI ayrıca, oluşturulan DTD'lere uygun XML dokümanlarının üretilmesi için bir dizi kural tanımlamaktadır. Bu kurallar gereklidir çünkü DTD üretim kuralları, modelleri gösteren XML'in sadece sentaksını belirlemektedir, anlamsal kuralları içermemektedir.

DTD ve doküman üretimi için oluşan XMI kuralları metamodelden otomatik olarak DTD üretilebilmesini sağlamakla beraber, bir metaveri havuzundan model almayı veya modelleri dışarıya taşımayı sağlayacak gerçekleştirme kodunun üretilmesini de sağlamaktadır.

Örneğin, MOF-Java ve MOF-XML eşlemleme kurallarını bilen bir MOF üreteci aşağıdakileri gerçekleştirebilecek kodu üretebilir:

- Java tabanlı bir metaveri havuzundan, MOF'tan türetilmiş Java uygulama programı arayüzü vasıtası ile modeller okuma.
- Bu modelleri XMI ile dış ortama taşıma.

Ayrıca bir MOF üreteci bunlara ek olarak aşağıdakileri gerçekleştirecek kodu da üretebilir:

- Bir modeli gösteren bir XMI dokümanını dışarıdan alma
- Java uygulama programı arayüzünü kullanarak modeli havuza yerleştirme

5.2.1 Java Metadata Interface (JMI)

Java Metadata Interface bir MOF-Java eşlemlemesidir, metaveriyi Java nesneleri olarak göstermek için gerekli kuralları tanımlar. Eşlemleme, bir metamodelin soyut sentaksının Java uygulama program arayüzlerine nasıl dönüştürüleceğini belirler. Uygulama program arayüzleri, metamodele uygun modelleri Java nesneleri olarak göstermeyi sağlar.

Uygulama program arayüzleri, tipik olarak bir metaveri havuzunun istemcileri tarafından kullanılacaktır. JMI sadece üretilmiş uygulama program arayüzlerinin sentaksını değil ayrıca semantiğini de belirler. Bu, farklı firmalara ait MOF üreteçlerinin uygulama program arayüzlerinin birlikte işler gerçekleştirmelerinin geliştirilebilmesini mümkün kılar.

5.3 Common Warehouse Metamodel (CWM)

Common Warehouse Metamodel veri modelleme, veri ambarı, veri dönüşümü ve veri analizi için OMG tarafından tanımlanmış standart bir dildir.

CWM, veri tabanı modelleme dillerini standartlaştırarak model merkezliliği daha ileri bir noktaya götürmektedir. Böylece farklı firmalara ait araçlar veri modellerini, veri dönüşüm kurallarını ve veri analiz belirtimlerini değiş-tokuş edebilmektedirler. CWM, firmalar modelleri ve kuralları kullanıcılarına göstermek için kendine ait farklı grafiksel arayüzler kullansalar bile bu değiş-tokuşu mümkün kılmaktadır.

CWM, Meta Object Facility (MOF) vasıtası ile tanımlanmıştır. Bu nedenle, CWM uyumlu veri modelleri ve dönüşüm kuralları, MOF tabanlı metaveri yönetim araçlarının kurumsal sistemlerle ilgili diğer metaveriler ile birlikte bütünleşik bir şekilde işleyebileceği bir metaveri oluşturmaktadır.

5.3.1 CWM ile Dönüşümlerin Modellenmesi

CWM'nin dönüşüm mimarisi sadece veri dönüşümleri ile ilgili değildir. CWM, model güdümlü veri dönüşümleri için kapsamlı destek sunmasıyla birlikte, aslında daha genel manada eşlemlemeleri formal modeller olarak tanımlamayı olanaklı kılan imkanlar sunmaktadır.

CWM aşağıdakiler için metamodeller sunmaktadır:

- İlişkisel veritabanları
- Çok boyutlu veritabanları
- Kayıt yapıları
- XML

Ayrıca, CWM ilişkisel, çok boyutlu ve kayıt yapısı metamodellerinin uzantıları olarak bir dizi ürüne özgü veritabanı metamodellerini de içermektedir:

- IBM IMS veritabanı tanımları
- Unisys DMS II veritabanı şemaları
- Hyperion Essbase
- Oracle Express

Ek olarak, veriyi farklı şekillerde işleyebilmek amacıyla kurallar tanımlamak için bir dizi metamodel içermektedir:

- Çevrimiçi Analitik İşleme (OLAP: OnLine Analytical Processing) kuralları
- Veri madenciliği kuralları
- Veri dönüşümü kuralları

Veri dönüşümü sadece veri ambarları için değil, Kurumsal Uygulama Entegrasyonu (EAI) firmadan firmaya entegrasyon (B2Bi) için de önemli bir gereksinimdir. Gereken dönüşüm, bir ilişkisel veritabanından diğerine, bir XML formatından diğerine, ilişkisel veritabanından XML'e, XML'den ilişkisel veritabanına, çok boyutludan ilişkisel veritabanına gibi dönüşümler olabilir.

Bir kurumdaki uygulamaları entegre etmekteki en büyük problemlerden birisi, benzer veriler için birçok veritabanı şemasının varolmasıdır. Örneğin, bir kurumsal uygulama entegrasyon sistemi, müşteriler hakkında aşağı yukarı benzer verileri kullanan ancak veri için tamamen farklı şemalar kullanan iki uygulamayı entegre etmek durumunda olabilir.

Firmadan firmaya entegrasyon senaryolarında, işbirliği içinde olunan bir iş ortağı ile kararlaştırılan ortak bir formatta veri alındığında, bu veriyi dahili sistemlerin kullanabileceği bir formata dönüştürmek gerekebilir. Kararlaştırılan format uluslararası bir standarda uygun olabilir, veya iki yanlı bir mutabakat ile kabul edilmiş olabilir. Aynı şekilde, iş ortağına veri gönderilirken, veri ortak formata dönüştürülmelidir.

CWM dönüşüm metamodeli, dönüşüm kurallarını belirtmek için yapılar tanımlamaktadır. Metamodele uygun olarak, birer kaynak ve hedef veri modeli bazında bir dönüşüm tanımlamak mümkündür. Kaynak ve hedef veri modeli CWM veri metamodellerinin herhangi birine uygun olmalıdır. XML metamodeli de veri metamodellerinden birisi sayılmaktadır. Çizelge 5.2'de metamodelin desteklediği dönüşümlerden çeşitli örnekler gösterilmiştir.

Çizelge 5.2 CWM Dönüşümü Kaynak ve Hedef Örnekleri

KAYNAK VERİ MODELİ	**HEDEF VERİ MODELİ**
İlişkisel Veritabanı Şeması A	İlişkisel Veritabanı Şeması B
İlişkisel Veritabanı Şeması	Çok Boyutlu Şema
Çok Boyutlu Şema	İlişkisel Veritabanı Şeması
Çok Boyutlu Şema A	Çok Boyutlu Şema B
İlişkisel Veritabanı Şeması	XML DTD
XML DTD	İlişkisel Veritabanı Şeması
XML DTD A	XML DTD B
IMS Şeması	XML DTD
Kayıt Yapısı	İlişkisel Veritabanı Şeması

Dönüşüm metamodeli bir M2 modelidir. Bir CWM dönüşüm kuralları kümesi ise, CWM dönüşüm metamodeli yapıları ile ifade edilmiş bir M1 modelidir.

6. MODELLEME DİLLERİNİN GENİŞLETİLMESİ VE YENİ MODELLEME DİLLERİNİN YARATILMASI

En çok kullanılan modelleme dili olan UML, profiller ile genişletilebilir bir modelleme dilidir. Ayrıca MOF kullanılarak, UML'in veya MOF tabanlı diğer metamodellerin genişletilmesi ve yeni modelleme dillerinin yaratılması mümkündür.

6.1 UML'in Profiller ile Genişletilmesi

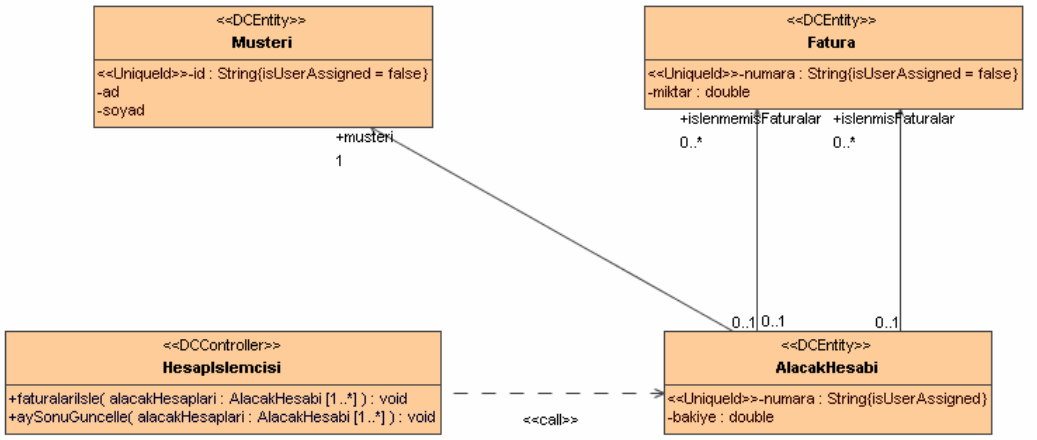
UML her amacı tamamıyla karşılayacak bir dil olarak tasarlanmamıştır, zaten her amacın standart tek bir dil ile modellenmesi olanaksızdır. UML, bunun yerine bir takım genişletme mekanizmaları sunar. Genişletme mekanizmaları, genel amaçlı bir UML aracı vasıtası ile UML'in tanımladığı tabanın ötesinde yapıların tanımlanması ve kullanılmasını mümkün kılar.

Çeşitli genişletmelerden oluşan bir küme UML lehçesini oluşturur ve buna *profil* adı verilir. Böylece UML tek bir dil değil, UML tabanlı diller için temel teşkil etmektedir. MDA, farklı sistem yönleri ve soyutlama seviyelerini desteklemek için genişletme mekanizmalarını kapsamlı bir biçimde kullanmaktadır.

UML'in sağladığı genişleme mekanizmaları stereotipler (sterotype) ve etiketli değerlerdir (tagged value). Bir UML profili, UML metamodelindeki elemanları genişleten bir stereotip ve etiketli değerler kümesinin tanımıdır.

6.1.1 Stereotipler (Stereotypes)

Şekil 6.1'de basit bir UML profili kullanılarak hazırlanmış bir sınıf diyagramı görülmektedir. Bu profilde üç tane stereotip kullanılmıştır. Bu stereotipler `DCEntity, DCControl, UniqueId` olarak adlandırılmıştır.



Şekil 6.1 Stereotip ve etiketli değerler

UML notasyonunda stereotipler << >> sembolleri arasına stereotip ismi yazılarak gösterilirler. Buna göre kullanılan stereotipler `<<DCEntity>>`, `<<DCControl>>`, `<<UniqueId>>` şeklinde gösterilir.

`DCEntity` stereotipi UML metamodelindeki *Class* elemanının bir uzantısıdır. Buna göre `DCEntity` bir dağıtık varlık bileşenini tanımlamaktadır. `DCControl` ise bir dağıtık denetleyici bileşenini tanımlamaktadır. Temel UML bu ayrımların yapılmasını desteklememektedir.

AlacakHesabı, Müşteri ve Fatura sınıflarının `DCEntity` olmaları durumu, ilgili sınıf kutularının üst kısmına `<<DCEntity>>` yazılarak ifade edilmiştir. `DCEntity` ve `DCControl` stereotipleri önemli ayrımlar yaratmaktadırlar ve her biri için çok farklı çalışacak kod üreticiler için de önemli girdilerdir.

Stereotiplerin sınıf haricindeki diğer UML metamodel elemanlarını da genişletebilmelerini göstermek amacıyla, örnekteki her `DCEntity` için özelliklerden birine `<<UniqueId>>` stereotipi eklenmiştir. `UniqueId`, sınıfın tanımladığı stereotipi destekleyen bileşenler için eşi olmayan (unique) tanımlayıcılar oluşturan bir özelliktir. Özelliğin `<<UniqueId>>` olması, özellik bildiriminin başına `UniqueId` yazılarak gösterilir.

Şekil 6.1'deki sınıf modeli bileşenlerin destekleyebileceği bir takım bilgiler sunar. Bu model, EJB ve .NET gibi bileşen ara katman yazılımlarının bir soyutlama seviyesi üstünde yer almaktadır. Bir üreteç bu modeli işleyerek, ara-katman yazılımına özgü

bir profille ifade edilmiş ara-katman yazılımına bağımlı bir modele çevirebilir ya da seçilen ara katman yazılımına uygun kod parçalarını doğrudan üretebilir. Örneğin EJB'ye özgü bir üreteç, `<<UniqueId>>` stereotipli özelliği EJB birincil anahtar sınıfına eşleyecektir. Başka ara katman yazılımlarını hedefleyen derleyiciler bu özelliği farklı şekilde eşleyecektir.

Stereotipler her tür UML metamodel elemanını genişletebilirler. Şekil 6.1'deki örnekte, Sınıf elemanını genişleten iki stereotip ve özellik elemanını genişleten bir stereotip bulunmaktadır. Örneğin birliktelik (Association) ve parametre gibi elemanlar da genişletilebilirler. Aslında stereotipler sadece sınıf modellemede kullanılan elemanları genişletmekle de sınırlı değildir. Durum modelleme, aktivite modelleme ve diğerlerinde kullanılan elemanları da genişletmek amacıyla kullanılabilirler.

Stereotip için ikon tanımlanması da mümkündür ve isteğe bağlı olarak stereotip eklenmiş elemanlar bu ikon ile de gösterilebilirler.

6.1.2 Stereotiplere Ait Etiketli Değerler (Tagged Values)

Bir stereotip tanımı ayrıca etiketli değerlerin tanımını da içerebilir. Örneğin, Şekil 6.1'deki `<<UniqueId>>` stereotipi tanımlayıcının değerinin kullanıcı tarafından mı, sistem tarafından mı atanacağını belirten bir etiket (isUserAssigned) tanımlanabilir. Eşi olmayan tanımlayıcılar ile ilgili olan temel UML dilinin desteklemediği başka özellikler de bu stereotipin etiketleri olarak tanımlanabilirler.

Şekil 6.1'de etiketli değerler küme parantezleri içinde {etiketin adı = değeri} biçiminde gösterilmiştir. `<<UniqueId>>` için tanımlanan *isUserAssigned* etiketi *true* veya *false* değerlerini alabilir.

6.1.3 Bağımsız Etiketli Değerler

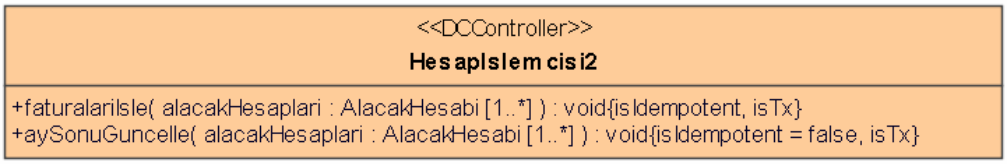
Bir stereotip tanımına ait olmayan etiketler de tanımlanabilmektedir. Bu biçimdeki etiketli değerler, model yaratılırken etiketlerin genişlettiği UML model elemanlarına ait örneklere eklenilirler.

Örneğin UML metamodelindeki *Operation* elemanını genişleten iki tane etiket tanımlayalım. *isTx* etiketi, operasyonun hareketsel olup olmadığını, yani operasyon başarıyla tamamlanamazsa operasyonun sebep olduğu durum değişiklerinin geri

döndürülmesinin gerekip gerekmediğini belirler. Bu tanım, hareketsel tutarlılığın hangi mekanizma ile sağlandığını belirtmediğimiz bu soyutlama seviyesindeki bir model için uygundur.

isIdempotent etiketi, operasyonun nötr olup olmadığını, yani operasyonun birden fazla çalıştırılmasının tehlikesiz olup olmadığını belirtir. Bu özellik, yedeğe geçme (failover) durumlarında yarım kalan operasyonların eski haline döndürülmesi için emek harcanıp harcanmayacağını belirler. Yedeğe geçme durumunda operasyon basitçe bir defa daha çağrılacaktır.

Şekil 6.2'de gösterilen *faturalariIsle* operasyonu nötrdür, çünkü bu operasyon sadece işlenmemiş faturaları okumaktadır. Eğer bütün faturalar işlendikten sonra çağrılırsa, işlenmemiş fatura bulunmadığını algılayarak hiçbir işlem gerçekleştirmemektedir. Bu kural ara katman yazılımı platformunda bağımsız olarak geçerlidir ve platform-bağımsız modelde gösterilmiştir.



Şekil 6.2 Bir stereotip ile ilişkilendirilmemiş etiketli değerler

aySonuGuncelle operasyonu nötr değildir, çünkü operasyon her çalıştırıldığında o anki bakiyeyi 30 günlük bakiyeye taşımakta, 30 günlük bakiyeyi 60 günlük bakiyeye taşımaktadır. Bir defa daha çağrılması bakiyelerin 30 gün daha ötelenmesine sebep olacaktır.

6.2 UML'in MOF ile Genişletilmesi ve Yeni Modelleme Dillerinin Yaratılması

UML metamodeli MOF ile tanımlanmıştır, bu yüzden UML'i MOF ile genişletmek de mümkündür. MOF ile UML'den tamamen farklı modelleme dillerinin yaratılması da mümkündür. MOF ile metamodel oluşturulurken UML'deki sınıf modelleme yapılarının çoğunluğu kullanılabilir. Aşağıdakiler MOF'da kullanılamaz:

- Birliktelik sınıfları
- Niteleyiciler (Qualifiers)

- n-li birliktelikler (sadece ikili birliktelikler kullanılabilir)
- Bağımlılıklar

MOF ile genişletme yapılırken, nesne yönelimli sınıf modellemenin olanakları kullanılmaktadır. Bu da MOF kullanmanın profil kullanmaya karşı en büyük avantajıdır. Örneğin, profil ile UML metamodel elemanları veya stereotipler arasındaki birliktelikleri tanımlamak mümkün değildir. Bu tip kısıtlamalar özellikle karmaşık genişletmelerde ciddi problemler yaratabilir.

7. MDA'NIN RİSKLERİ

MDA, çok şeyler vaat etmesi ile birlikte bünyesinde bir takım güçlükler de barındırmaktadır. UML'in geniş çapta kabul görmüş olması ve yazılım geliştirmede MDA tipindeki yaklaşımların trendi, MDA'yı otomatikman başarılı yapmaya yetmeyecektir. Üstelik MDA, UML 2'de dilin mimarisinin tekrar tanımlanması ve iş modellemede getirilen geliştirmeleri o kadar etkilemiştir ki, UML'in başarısının sürmesi de MDA'nın başarısına bağlı hale gelmiştir.

Bu bölümde MDA'nın gelişimi ve kullanımında ortaya çıkabilecek güçlükler ve aşılması gereken engeller anlatılacaktır:

7.1 Çok Fazla Modelleme Karmaşıklığı

Bir platform bağımsız model, otomatik olarak çalıştırılabilir veya büyük oranda üretilebilir kod oluşturmak için matematiksel kesinlikte bir kısıt dili kullanılarak ciddi miktarda soyut programlama yapmayı gerektirmektedir. Öte yandan, böyle bir çalışma programa dayalı çalıştırma ile ilgili becerilere sahip olmayan iş modelcilerini zorlayabilir. Ayrıca, yazılım geliştiricileri kaynak kod geliştirmekten uzaklaştırabilir.

Modelleme ile ilgili bir başka ek iş de, UML profillerini türetmek ve gözden geçirmek gibi metamodel aktivitelerdir. MDA gelecekte başarılı profiller sağlayacak olsa da, MDA'yı erken döneminde benimseyenler bu profilleri üretmek için gerekli maliyeti de göğüslemek zorunda kalacaklardır. Ek olarak, UML'i verimli bir şekilde gerçekleştirmek tecrübe ve eğitim gerektirmektedir.

UML'in eski versiyonlarının karmaşıklığı, çoğu durumda verimli kullanılamayacak çeşitli modelleme tipleri ve görünümleri sonucunu doğurmuştur. Bazı organizasyonlar UML küçük bir bölümünü kullanmış (sınıf diyagramları veya kurulum diyagramları gibi) ve UML'i yazılım geliştirme süreçlerine hiç bir zaman tam anlamıyla entegre etmemişlerdir. MDA ile kullanılan UML, UML'i yazılım geliştirmeye tam anlamıyla entegre etme baskısını artırmaktadır, fakat bu tarzdaki organizasyonlar için bu çok karmaşık olacak ve mali açıdan mümkün olmayabilir.

7.2 Araç Gerçekleştirmelerinin Evrensel Olmayışı

OMG eğer kullanışlı standartlar elde etmek amacıyla profilleri etkin bir biçimde

yönetemezse, modelleme camiası UML'in ortaya çıkışı öncesinde yaşanan metot karmaşasının benzerini MDA'da yaşayabilir. Ayrıca, OMG bir standart tanımladı diye, bütün firmalar ve müşterileri bu standardı benimsemek zorunda değildir.

Modelleme için birçok seçenek sunmak ve bunları belirli ortamlara uygun hale getirmek, gerçekleştirme detaylarına sahip modellerin çıkmasını sağlayacaktır, ancak bu aynı zamanda bu modellerin kaliteli olması demek değildir.

Çoğu MDA gerçekleştiricisi, işlerinde kullandıkları bir profil veya uzantıyı değiştirmek pahasına, sadece standart olduğu için bir standardı kullanmaya değer bulmayacaklardır. Araçlar arasında iletişimi sağlayacak XMI tanımı bu tehlikeyi önleyebilir. UML araçları kullanıcının tanımladığı her türlü profili bu şekilde desteklerse, modeller arasında iletişimi sağlayacak ortak bir platform sağlanarak bu evrensel olmayış en azından yönetilebilir.

7.3 Test Edilmemiş Davranışsal Modelleme

Şu ana kadarki MDA gerçekleştirmeleri çoğunlukla yapısal diyagramlara yoğunlaşmış durumdadır. UML 2 ile davranış modelleme için birçok iyileştirme bulunmaktadır. Zaten MDA için, davranış ve iş akışı içinde daha net ve iyi modelleme gerekmektedir. Ancak, bu elemanların bir model derleyici ile koda nasıl dönüştürüleceği net değildir.

UML'in davranışsal modelleme özellikleri, sınıf modelleri ile karşılaştırıldığında kod üretimi açısından nispeten denenmemiş durumdadır. Tabi bu gözlem hem tehlike hem de fırsat olarak yorumlanabilir, çünkü davranışsal modellemeyi çalıştırılabilir hale getirmek için yapılacak daha çok iş vardır. Burada önemli soru, bu geliştirmelerin mali açıdan değip değmeyeceğidir.

7.4 MDA'nın Yanlış Kullanımı

MDA'nın önündeki ana engel, teknolojinin nasıl kullanılacağı bilgisi ile teknolojiyi gerçekleştiren araçlar arasındaki mesafedir. Ayrıca eğer endüstrideki önemli firmalar MDA'yı kötü bir şekilde gerçekleştirirse, MDA ile ortaya çıkan fikirler havada kalacak ve insanlar bundan gittikçe uzaklaşacaklardır.

MDA gerçek problemleri çözmekte kullanılmalıdır. Eğer modelleme gerçek problemleri çözmüyorsa, entelektüel bir meraktan öteye gidemez. Bunun için de

herkesin, modellerin kendisi için önemli olan noktalarına odaklanabilmesini sağlayacak bir biçimde yönetilebilmesini mümkün kılan bir süreç büyük önem taşımaktadır.

8. MDA PROJESİ: KONUM SUNUCUSU

8.1 Proje Tanımı

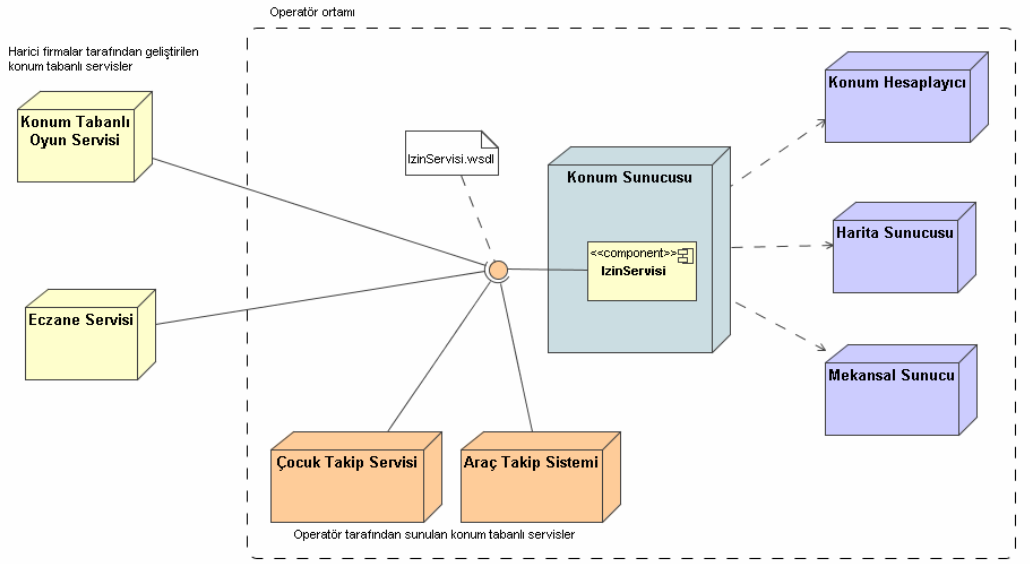
Konum Sunucusu, ağ operatörleri için katma değerli servislerin geliştirileceği uygulama geliştirme arayüzlerini, uygulama, abone ve gizlilik yönetimi hizmetlerini sunan bir platformdur. Proje kapsamında Konum Sunucusu kullanılarak geliştirilmiş bir örnek uygulama olan *Çocuk Takip Servisi* de gerçekleştirilmiştir.

Projede odaklanılan nokta, MDA'nın büyük ölçekli ve dağıtık yapıdaki bir uygulama geliştirilirken sağladığı kolaylık ve zorluklardır. Proje gerçekleştirilirken, mümkün olan en ileri aşamaya kadar platform bağımsız bir model oluşturulmaya çalışılmıştır.

8.2 Konum Sunucusu'na Üst Düzeyden Bakış

Ağ operatörleri, (örneğin GSM operatörleri) gün geçtikçe, konum bilgisini kullanan katma değerli servisleri hayata geçirmektedir. Bu servislerin ortak bir platform üzerinde geliştirilmesi, bunların hem geliştirilme sürecini kolaylaştıracak hem de operatörün bunları merkezi olarak denetleyebilmesini sağlayacaktır. Ayrıca konum gizliliği de merkezi olarak yönetilecektir. Çünkü konum gizliliği tam manasıyla sağlanmadığı sürece, aboneler ister istemez bu servisleri kullanmakta çekingen davranmaktadır.

Konum Sunucusu için geliştirilen uygulamalar operatör tarafından geliştirilip sunulacağı gibi, harici firmalar tarafından geliştirilip, doğrudan müşterilere sunulabilir. Burada harici firmanın sunduğu uygulama doğal olarak uzaktan çalışmaktadır ve Konum Sunucusu ile Web Servisleri vasıtası ile haberleşebilmektedir. Aslında hem operatöre, hem de harici firmalara ait uygulamalar Konum Sunucusu'nun dışında bir yere kurulur ve bunların Konum Sunucusu ile tek iletişimleri Web Servisleri üzerindendir. Şekil 8.1'de Konum Sunucusu kurulum diyagramı görülmektedir.



Şekil 8.1 Konum Sunucusu Kurulum Diyagramı

8.2.1 Konum Sunucusu Hizmetleri

Konum Sunucusu'na sorgulamalar, günümüzde birlikte işlerlik konusunda standartlaşmış olan Web servisleri vasıtasıyla gerçekleştirilebilmektedir. Sorgulamalar üç tiptir:

1) **Konum sorgusu:** Sorulan abonenin koordinatını X,Y olarak döner.

2) **Adres sorgusu:** Sorulan abonenin adresini döner. Bu sorgu kendi içinde bir konum sorgusu gerçekleştirmekte, ardından elde ettiği koordinatı adrese dönüştürmektedir. Bu dönüşüm işlemine *reverse geocode* denmektedir.

3) **Harita sorgusu:** Sorulan abonenin bulunduğu bölgenin haritasını bir URL olarak döner. Bu sorgu kendi içinde bir konum sorgusu gerçekleştirmekte, ardından elde ettiği koordinatı orta nokta olarak alan bir harita oluşturmaktadır.

Bir konum sunucusu, bu üç tip sorguyu da harici sistemleri kullanarak gerçekleştirir. Örneğin konum için operatöre ait bir GMLC sistemi ile iletişim kurmakta, adres için bir mekansal (spatial) sunucuya sorgular gerçekleştirerek uygun bir adres oluşturmakta, harita içinse bir harita sunucusundan ilgili bölgenin haritasını istemektedir. Aslında konum sunucusunun önemli bir özelliği, kullanıcıların bu çok çeşitli ve karmaşık yapıdaki sunucularla doğrudan iletişim kurmak yerine, Konum

Sunucusu'nun sağladığı basit bir uygulama geliştirme arayüzünü kullanarak, zengin içerikli konum tabanlı sistemlerin kolayca gerçekleştirebilmesine imkan sağlamaktır. Ayrıca her bir harici sistemin farklı firmalar tarafından geliştirilmiş çeşitlerini destekleyerek, operatörlerin ve Konum Sunucusu'nu kullanacak programcıların işini oldukça kolaylaştırmaktadır.

Bu proje kapsamında geliştirilecek Konum Sunucusu bu üç tip sorguyu da simülasyon vasıtası ile gerçekleştirecektir. Yapılan sorgular sonucunda önceden hesaplanmış koordinat, adres ve haritalar dönülecektir.

8.2.2 Uygulama Tipleri

Gizlilik ayarları, uygulamalar üzerinde bir takım kısıtlamalar koyarak uygulamaların sadece kendisine verilen yetkiler dahilinde sorgulama yapabilmesini sağlamaktadır. Özellikle operatör tarafından yönetilmeyen harici firmalar tarafından geliştirilen uygulamalarda gizlilik ayarlarının önemi daha da artmaktadır.

Konum Sunucusu'nda uygulamalar gizlilik kuralları açısından iki türe ayrılmaktadır:

AÇIK: Bu tipteki bir uygulamaya üye olan aboneler, birbirlerinin kara listelerinde olmadıkları sürece birbirlerini sorgulayabilirler. Ayrıca *soran* abonenin belirtilmesi gerekli olmadığı için işlemlerinde tek abone kullanan servisler için uygundur. Buna örnek olarak bir abonenin o anda bulunduğu konuma en yakın eczaneleri elde etmesini sağlayan bir servis düşünülebilir.

KAPALI: Bu tipteki bir uygulamaya üye olan abonelerin sorgulama yapabilmek için birbirlerini beyaz listeye eklemeleri gerekmektedir. Yani A abonesinin B abonesini sorgulayabilmesi için B abonesinin beyaz listesinde A abonesi yer almalıdır. Bu tipteki uygulamaları kullanarak sorgulama yapılırken, konum soran abonenin belirtilmesi zorunludur. Kapalı tipteki uygulamalar, yüksek güvenlik gerektiren ve abonelerin birbirleri arasında sorgulama yapabildikleri servisler için uygundur. Örneğin, bu proje kapsamında gerçekleştirilecek ve *kapalı* tipte bir uygulama olan *Çocuk Takip Servisi*'nde bir veli, sadece kendi çocuklarını sorgulayabilmeli, diğer velilere ait çocukları sorgulayamamalıdır.

8.2.3 Konum Sunucusu'nun Arayüzleri

Konum Sunucusu, Yönetici ve abone web arayüzlerini sunmaktadır:

8.2.3.1 Yönetici Arayüzü

Yönetici arayüzü, konum sunucusunu yöneten operatöre hitap etmektedir. Bu arayüz ile, uygulama tanım işlemleri ve abone tanım işlemleri gerçekleştirilebilmektedir.

8.2.3.2 Abone Arayüzü

Abone arayüzü, konum servislerini kullanacak bütün abonelere hitap etmektedir. Bu arayüz ile, bir abone Konum Sunucusu'nun sağladığı uygulamalara üye olabilmekte, beyaz ve kara listelerine istediği aboneleri ekleyebilmektedir. Abonenin yaptığı bu ayarlamalar bütün uygulamalar üzerinde etkili olmaktadır. Abone arayüzünden giriş yapabilmek için, öncelikle yönetici arayüzünde abonenin tanımlanması gerekmektedir.

8.2.4 Çocuk Takip Servisi

Çocuk Takip Sistemi, anne-babaların istedikleri zaman çocuklarının konumlarını cep telefonu üzerinden elde etmelerini sağlayan Konum Sunucusu hizmetleri kullanılarak geliştirilmiş bir servistir. Bu servis Konum Sunucusu'nun dışında yer almaktadır.

Kapalı tipte bir uygulama olan Çocuk Takip Servisi'nde velinin çocuklarını sorgulayabilmesi için çocukların Konum Sunucusu Abone arayüzünü kullanarak, veliyi beyaz listelerine eklemeleri gerekmektedir. Çoğu zaman bu işlemi veli ve çocuk birlikte gerçekleştirecektir. Bu ayarlamalar, uygulamanın farklı amaçlarla, konudan alakasız insanların yerini sorgulamasını engellemek için gerekmektedir.

8.3 Projede Kullanılan Araç ve Teknolojiler

Projede kullanılacak araçları seçmek amacı ile varolan MDA araçlarının MDA'yı ne derecede gerçekleştirilebileceği araştırılmış ve konum sunucusu için uygun teknolojiler seçilmiştir.

8.3.1 Gözden Geçirilen MDA Modelleme Araçları

AndroMDA 3.1:

Bir açık-kodlu MDA çatısı olan AndroMDA yazılım geliştirme endüstrisinde yoğun olan kullanılan teknolojilere (EJB, Hibernate, Struts, Spring) destek vermekte ve platformdan bağımsızlık açısından diğer ürünlerden önde gözükmektedir.

Bu çatı, MDA bağlamında aşağıdaki PIM->PSM otomasyon hizmetlerini sağlamaktadır:

- Sınıflar ve sınıflar arası ilişkilerin oluşturulması
- EJB/Hibernate tanımlayıcı dosyalar ve yardımcı sınıfların oluşturulması
- EJB/Hibernate sorgularının OCL ile tanımlanabilmesi
- Spring servislerinde OCL ile önkoşul/sonkoşul tanımlanabilmesi
- Kullanım Durumu, Aktivite Diyagramı kullanarak web tarafında Struts ile MVC (istemci/sunucu tarafındaki geçerlik işlemlerini de içeriyor)
- Spring (sürerlik, iş) + bpm4Struts (sunuş)
- Web servis hizmetleri ve Sınıf -> XML Şema dönüşümü

Ürünün uygulamanın farklı katmanları için, modelden otomatik oluşturabildiği kodun, toplam koda yaklaşık olarak oranı aşağıdaki gibidir [1]:

- Sürerlik (EJB/Hibernate): 100%
- İş (EJB): 30%
- Web Servisleri (Axis): 100%
- Sunuş (Struts): 80%

Ürün içerisinde bir UML aracı veya IDE barındırmamaktadır, XMI destekleyen herhangi bir UML aracı ile kullanılabilmektedir. Tavsiye edilen *MagicDraw UML* aracı kullanılarak, bu aracın desteklediği, sınıf diyagramları üzerinden alan/operasyon isimleri düzeyinde iki yönlü geliştirme (round-trip engineering) olanakları kullanılabilmektedir [2].

Borland Together Architect 2006:

Eclipse IDE'si ile entegre olarak çalışan bu ürün, OCL kısıtlarını sınıflar ve operasyonlar üzerinde etkin biçimde kullanılabilir ayrıca XMI standardını da desteklemektedir. Sınıf ve sıra diyagramları ile round-trip engineering de

desteklenmektedir. Together, sunuş katmanı için herhangi bir kolaylık sağlamamakta, ancak geliştirilecek ürünlerin çekirdek (core) katmanında kullanılabilecek MDA yaklaşımını benimseyen çeşitli imkanlar (dönüşümler, UML profilleri) sunmaktadır. Bir çok tasarım örneklerini de kullanmayı kolaylaştıran bu ürün, platform bağımsızlık açısından AnroMDA'nın gerisinde kalmasına karşın, bir bütün olarak oldukça kullanışlı görünmektedir.

IBM Rational Software Architect 6.0:

Eclipse IDE'si ile entegre olarak çalışan bu kullanışlı ürün, MDA ve UML 2.0 üzerine bina edilmesine karşın, hem XMI desteği konusunda yetersiz kalmakta, hem de platform bağımsızlık açısından AndroMDA ve Together'ın gerisinde kalmaktadır. Bu da ürünün ancak tek başına kullanıldığında verimli çalışması anlamına gelmektedir. Sunuş için JSP/Struts modellemesi de ürün ile gerçekleştirilebilmektedir.

iO-Software ArcStyler 5.1:

Ürün, içinde gömülü olarak MagicDraw UML aracını barındırmaktadır. ArcStyler, J2EE ve .NET'i aynı anda desteklemesiyle platform bağımsızlık konusunda diğer ürünlerin önünde görünmektedir. Ancak, ürün hem otomatik kod oluşturma konusunda yetersiz kalmakta, hem de sunuş katmanı için firmaya bağımlı olan Accessor çatısını kullanmaktadır.

8.3.2 Seçilen Araç ve Teknolojiler

Konum sunucusunun geliştirilmesi için, incelenen ürünler arasından, hem azami sayıda teknolojiyi desteklemesi açısından, hem de MDA prensiplerine en yakın ürün olması açısından, açık kaynak kodlu AndroMDA ürünü seçilmiştir. Projenin UML modellemesinde MagicDraw UML aracı kullanılmıştır.

Proje bir *J2EE* uygulamasıdır. Mümkün olduğu kadar platform bağımsız olarak geliştirilen proje, bütün J2EE uygulama sunucularına (JBoss, BEA Weblogic vs.) kurulabilir durumdadır. Uygulamada, servis tabanlı *Spring Framework* ile konum servisleri *EJB session bean* olarak gerçekleştirilmiştir. Bu servisler harici uygulamaların dışarıdan erişebilmesi için web servisleri olarak sunulmaktadır. Web servisleri gerçekleştirilmesi için *Apache Axis* kullanılmıştır.

Veritabanından da bağımsız olarak geliştirilen Konum Sunucusu gizlilik yönetimi ve

uygulama detayları vb. bilgilere bir nesnel-ilişkisel veritabanı eşlemleme teknolojisi olan *Hibernate* vasıtası ile erişmektedir.

Konum Sunucusu'nun iç yapısını gerçekleştirilmesini sağlayan yukarıdaki teknolojiler için AndroMDA'nın *Spring, WebServices, Hibernate* kartuşları kullanılmıştır.

Konum Sunucusu'ndaki web arayüzleri *Apache Struts* ve *Java Server Pages* ile gerçekleştirilmiştir. Burada iş proseslerini içeren kullanım durumu, aktivite şemaları hazırlanmış ve bunlar üzerinden web uygulamasını oluşturabilen AndroMDA *BPM4Struts* kartuşu kullanılmıştır.

8.4 Klasik Yazılım Geliştirme Yöntemleri Açısından Projenin Değerlendirilmesi

Uygulamayı dağıtık çalışan bir konum sunucusu kapsamında ele alırsak, kullanılacak J2EE teknolojileri uygulama alanına özel kodlar ile birlikte oldukça fazla kod ve tanımlayıcı dosya içerecektir.

Örneğin basit bir servisten ibaret olacak bir Session EJB'si için asgari olarak;

- Bir *EJB* arayüzü
- Bir *EJB Bean* gerçekleştirme sınıfı
- Bir *EJB Home* arayüzü
- *ejb-jar.xml* tanımlayıcı dosyası
- J2EE uygulama sunucusu tanımlayıcı dosyasında bu EJB'ye ait tanımlar

oluşturulması gerekmektedir. Üstelik bunların uygun şekilde dizinlere yerleştirilmesi ve arşivlenmesi de gerekir.

Web servisleri için de benzer bir durum söz konusudur. Sunulan servisler muhtemelen kod içinde varolan bazı metotlara karşılık gelmektedir. Ancak bu metotları doğrudan web servisi olarak sunmak mümkün değildir. Bir WSDL dosyası hazırlanmalı ve kullanılan teknolojiye göre bu metotları çağıracak çeşitli sınıflar geliştirilmelidir.

Modern IDE'ler, EJB, Hibernate veya WEB servisleri için bu tip kodları oluşturabilmektedirler. Örneğin *Borland JBuilder, Oracle JDeveloper* gibi ticari IDE'ler doğrudan, açık kaynak kodlu bir IDE olan *Eclipse* ise eklentiler (plugin) vasıtasıyla bunu yapabilmektedir. Bu IDE'ler veya eklentiler, kendi görsel model

geliştirme arabirimlerini de içermektedir. Ama tabii ki temel problem, bu modellerin teknolojiye ve IDE'ye son derece bağımlı olmalarıdır. Kullanılan ürün birden fazla teknoloji desteklese bile, PIM tarzı modeller geliştirmek mümkün olmamaktadır. Birden fazla teknoloji için gerçekleştirme yapılacaksa, her teknoloji için sıfırdan ayrı modeller geliştirilmesi gerekmektedir.

MDA'nın en kolay tatbik edilebildiği alan bu tip ekstra kod ve tanımlayıcı dosyalardır.

Konum sunucusunu kullanacak bir web uygulaması geliştirilmesi söz konusu olduğunda, uygulamanın *Apache Struts* gibi bir çatı (framework) kullanması gerekecektir. Bu çatı ile, web sayfaları arasındaki geçişlerin kontrolünün merkezi olarak gerçekleştirilmesi sağlanacak, form geçerlik (validation) işlemleri ve kontrolcü ve sunum arasındaki iletişim kod kirliliği yaratmayacak şekilde sağlanabilecektir.

Bu IDE'ler veya eklentiler, *Apache Struts* için görsel model geliştirme arabirimlerini de içermektedir, ancak yukarıda söz edilen problem burada da geçerlidir. Oluşturulan model hem IDE'ye, hem de doğal olarak *Apache Struts* teknolojisine bağımlıdır.

Ayrıca, geliştiricilerin problemlere çabuk çözüm getirebilmek maksadıyla iş fonksiyonları ve ekstra kodları birbiriyle iç içe geçmiş şekilde gerçekleştirmeleri uzun vadede yazılımın bakımını zorlaştıracaktır. Bunu engellemek için özellikle tecrübeli geliştiricilerin, az tecrübeli geliştiricilerin yazdıkları kodu gözden geçirmeleri gerekir. Her ne olursa olsun, çeşitli seviyelerde geliştiricilerden oluşan bir yazılım geliştirme ekibinde kod kalitesini yüksek tutmak efor gerektirecektir ve projede zaman açısından baskı olması durumunda –ki bir çok yazılım projesinde böyledir– bir noktadan sonra kod kalitesini muhafaza etmek zorlaşacaktır.

8.5 Konum Sunucusu'nun Tasarımı ve Gerçekleştirilmesi

Bu bölümde AndroMDA kullanılarak geliştirilen Konum Sunucusu ve Çocuk Takip Servisi'nin geliştirilme adımları anlatılacaktır.

AndroMDA proje yapılandırması için *Apache Maven* aracını kullanmaktadır. *Apache Maven*, yazılım projelerinde sıklıkla kullanılan *Apache Ant* aracına göre oldukça gelişmiş bir proje yapılandırma aracıdır.

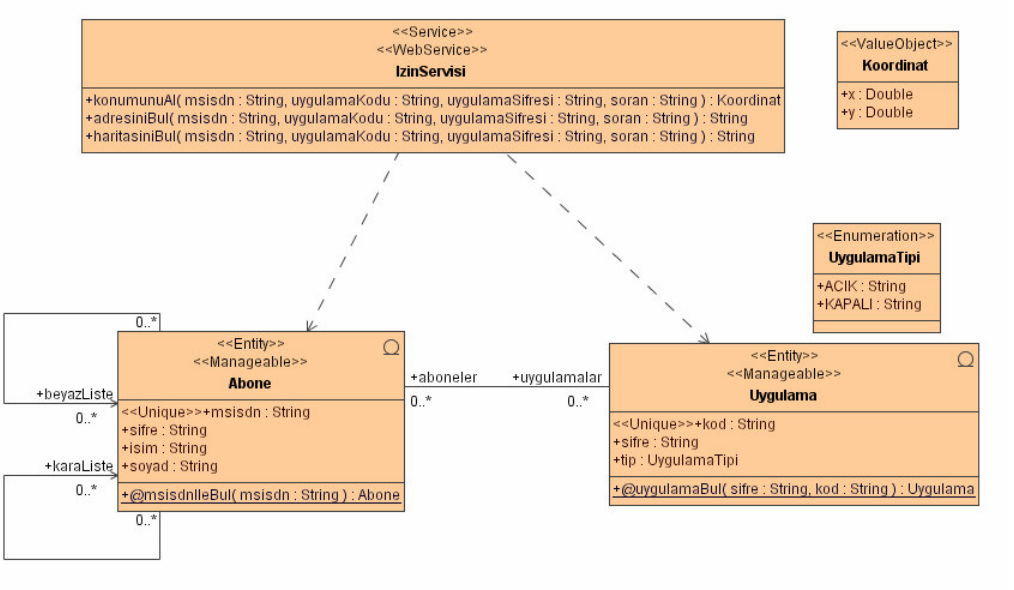
Öncelikle seçilen teknolojilere göre *Maven* ile bir AndroMDA projesi oluşturulmuştur. Konum Sunucusu için oluşturulan proje aşağıdaki modüllere ayrılmaktadır:

- **Mda:** MDA modülü projenin kalbini oluşturmaktadır. Burada uygulamanın oluşturulması için gereken UML modeli ve konfigürasyon dosyaları bulunmaktadır. Bütün diğer modüllere ait temel kodlar, bu model ve konfigürasyon dosyaları üzerinden otomatik olarak üretilecektir.
- **Common:** Bu modülde diğer modüller tarafından ortak olarak kullanılacak olan kaynaklar ve sınıflar bulunmaktadır.
- **Core:** Uygulamanın çekirdeği diyebileceğimiz kısmını içeren bu modülde Spring, EJB, Hibernate'i kullanan kaynak ve sınıflar bulunmaktadır. Ayrıca uygulamanın veri tabanı şeması da bu modül altında bulunmaktadır.
- **Web:** Bu modülde sunuş katmanına ait kaynaklar ve sınıflar bulunmaktadır.
- **Webservice:** Bu modülde web servisleri için Axis kullanana kaynaklar ve sınıflar bulunmaktadır.
- **App:** Bu modülde J2EE uygulama sunucusuna kurulacak olan ear paketini oluşturacak kaynaklar ve sınıflar bulunmaktadır.

Bütün bu modüller altında, *target* dizinleri otomatik olarak oluşturulan kaynak ve sınıfları içermekte, *src* dizinleri ise elle müdahale gerektirebilecek kaynak ve sınıfları içermektedir.

8.5.1 Konum Sunucusu Modelinin Oluşturulması

Projenin *Mda* modülünde AndroMDA tarafından boş bir model yaratılmıştır. Bu model AndroMDA'ya ait UML profillerini kullanıma sunmaktadır. Öncelikle Konum Sunucusu'nun sunacağı servisleri ve temel varlıkları içeren sınıf diyagramı



Şekil 8.2 Konum Sunucusu Sınıf Diyagramı

oluşturulmuştur (Şekil 8.2). Sınıf diyagramındaki *IzinServisi* adlı sınıf Konum Sunucusu tarafından sunulan üç servisi içermektedir. <<Service>> stereotipi, bu sınıfın *Spring* çatısına göre bir *EJB Session Bean* olarak oluşturulmasını, <<WebService>> stereotipi de bu servislerin web servis olarak dışarıya sunulacağını belirtir.

Abone sınıfı operatöre ait aboneleri temsil eden bir sınıftır. <<Entity>> stereotipi sınıfın veritabanında saklanacak (*Hibernate* ile) bir varlık olduğunu, <<Manageable>> stereotipi ise standart olarak kullanılan yarat-oku-güncelle-sil (CRUD: create-read-update-delete) servislerinin ve sunuş katmanında kullanılacak değer nesnesi (value object) sınıflarının otomatik olarak oluşturulmasını sağlar.

Abone sınıfının *msisdn* niteliği için verilmiş <<Unique>> stereotipi, bu niteliğin aboneleri birbirinden ayırt etmekte kullanılan nitelik olduğunu belirtir. *beyazListe* ve *karaListe* birliktelikleri abonenin beyaz ve kara listelerindeki aboneleri göstermektedir. Bu birliktelikler ilgili sınıflarda sınıflar arasındaki ilişkilere ve veritabanı tabloları arasındaki kısıtlara dönüşecektir.

Abone sınıfındaki bir sorgu metodu olan *msisdnIleBul* metodu,

```
context Abone::msisdnIleBul(msisdn:String):Abone
body msisdnIleBul: allInstances() -> select (abone | abone.msisdn = msisdn)
```

OCL ifadesini kısıt olarak barındıran bir metottur. Bu metot, modelin işlenmesi sonrasında bir Hibernate sorgusuna dönüştürülecektir. Bu Hibernate sorgusu da çalışma zamanında bir SQL sorgusuna dönüştürülerek veritabanına gönderilecektir.

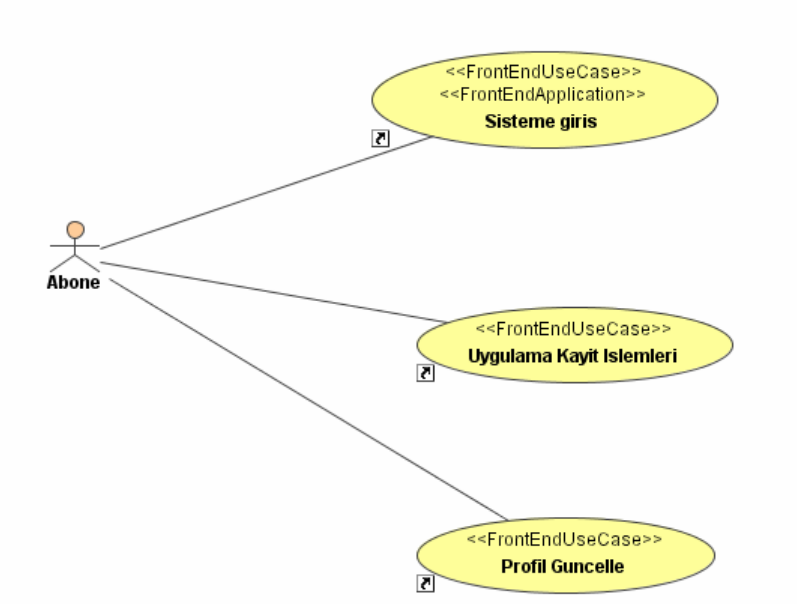
Uygulama sınıfı Konum Sunucusu'nda tanımlanacak uygulamaları temsil etmektedir. *UygulamaTipi*, <<Enumeration>> stereotipi ile uygulama sınıfının tip niteliğinin alabileceği değerleri sınırlamaktadır. Bu değerlerin veritabanına yazılmasından sayfalarda görünmesine kadar her şey otomatik olarak yapılmakta, geliştirici koda müdahale ederken, sadece ACIK ve KAPALI adlı sabit değerleri kullanmaktadır.

Abone ve *Uygulama* sınıfları arasındaki birliktelik, *Abone* sınıfından abonenin üye olduğu uygulamalara, *Uygulama* sınıfından uygulamaya üye olan abonelere ulaşmayı sağlamaktadır.

IzinServisi ile, *Abone* ve *Uygulama* sınıfları arasındaki bağımlılık ilişkisi, *IzinServisi*'nin bu varlıklara ait veri erişim nesnelerine (data access object) ulaşmasını sağlayacaktır.

8.5.2 Konum Sunucusu Abone Arayüzünün Tasarlanması

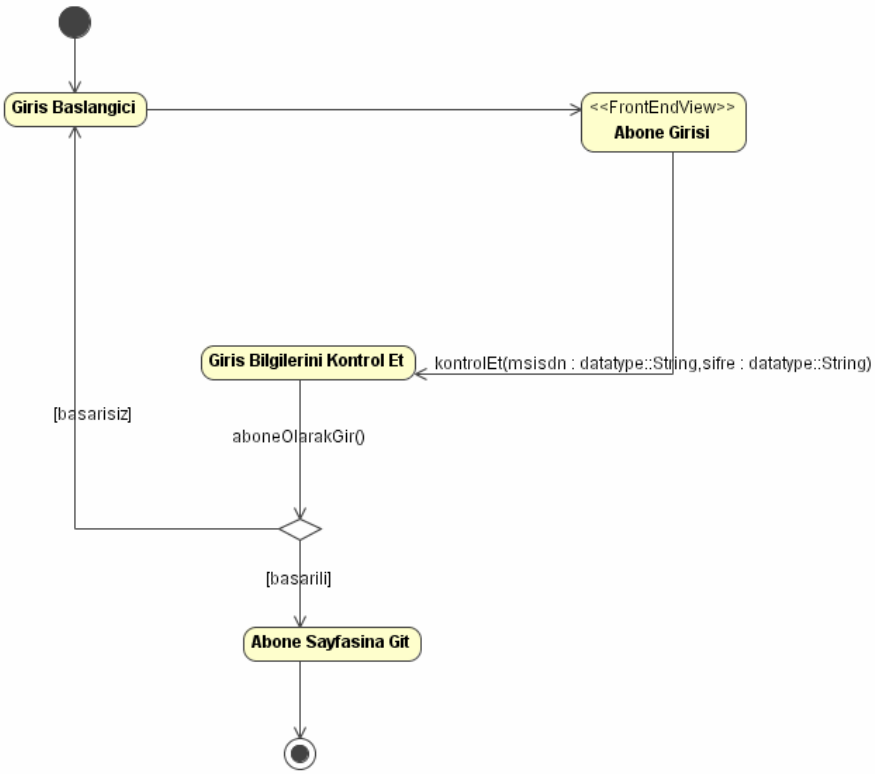
Abone arayüzünün tasarımına öncelikle bir kullanım durumu diyagramı hazırlanarak başlanmıştır (Şekil 8.3):



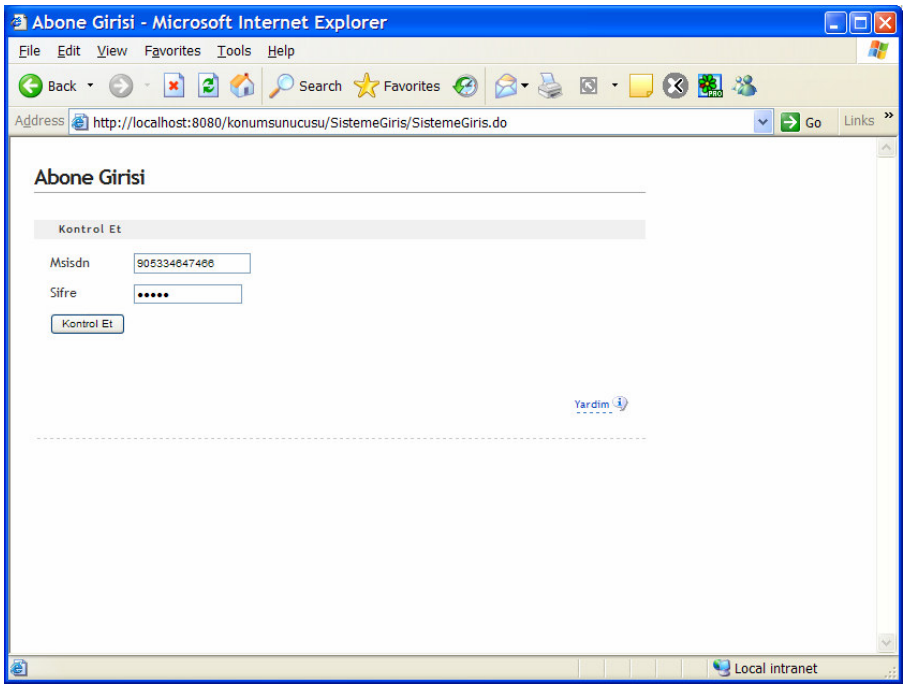
Şekil 8.3 Konum Sunucusu Abone Arayüzü Kullanım Durumları

Bu diyagramda *Abone* aktörünün sistemi kullanım amaçları gösterilmektedir. Her bir kullanım durumu <<FrontEndUseCase>> stereotipine sahiptir ve her biri ile bağlantılı aktivite diyagramları bulunmaktadır. <<FrontEndUseCase>> stereotipi *Sisteme Giriş* kullanım durumunun uygulamaya giriş noktası olduğunu belirtmektedir.

Şekil 8.4'te *Sisteme giriş* kullanım durumu için hazırlanan aktivite diyagramı görülmektedir. <<FrontEndView>> stereotipi kullanıcıdan giriş beklenen eylem durumları için kullanılır. Burada *Abone Girişi* eylemininden *Giriş Bilgilerini Kontrol Et* eylemine çizilen geçiş üzerindeki *kontrolEt* tetikleyicisi (trigger) kullanıcıdan *msisdn* ve *şifre* bilgilerinin sayfadaki metin kutuları ile alınmasını sağlamaktadır. <<FrontEndView>> stereotipli eylemlerden çıkan geçiş, sayfada bir gönder (submit) düğmesi olarak varolacaktır. Şekil 8.5'te diyagrama göre oluşan sayfa görülmektedir.

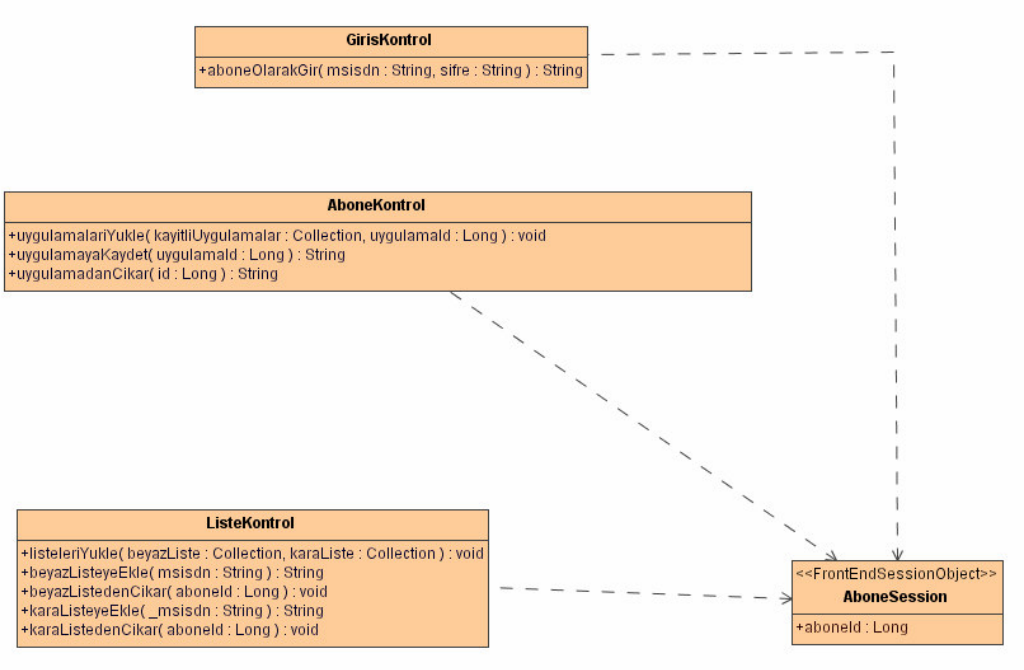


Şekil 8.4 Sisteme giriş aktivite diyagramı



Şekil 8.5 Sisteme giriş sayfası

Her aktivite diyagramına ait bir kontrol sınıfı bulunmaktadır. Bu sınıftaki metotlar, sunuş ve iş katmanı arasındaki kontrol katmanı işlevini görmektedir. Şekil 8.6'daki sınıf diyagramında Konum Sunucusu Abone kontrol sınıfları görülmektedir. Aktivite diyagramındaki *aboneOlarakGir()* geçişi *GirisKontrol* sınıfındaki metodu çağırmaktadır. Kontrol sınıflarındaki metotların içi geliştirici tarafından doldurulacaktır. Sınıfların boş hali ve ilgili sayfa değişkenlerinin metot tarafından kullanılabilir hale getirilmesi AndroMDA tarafından otomatik olarak gerçekleştirilmektedir.



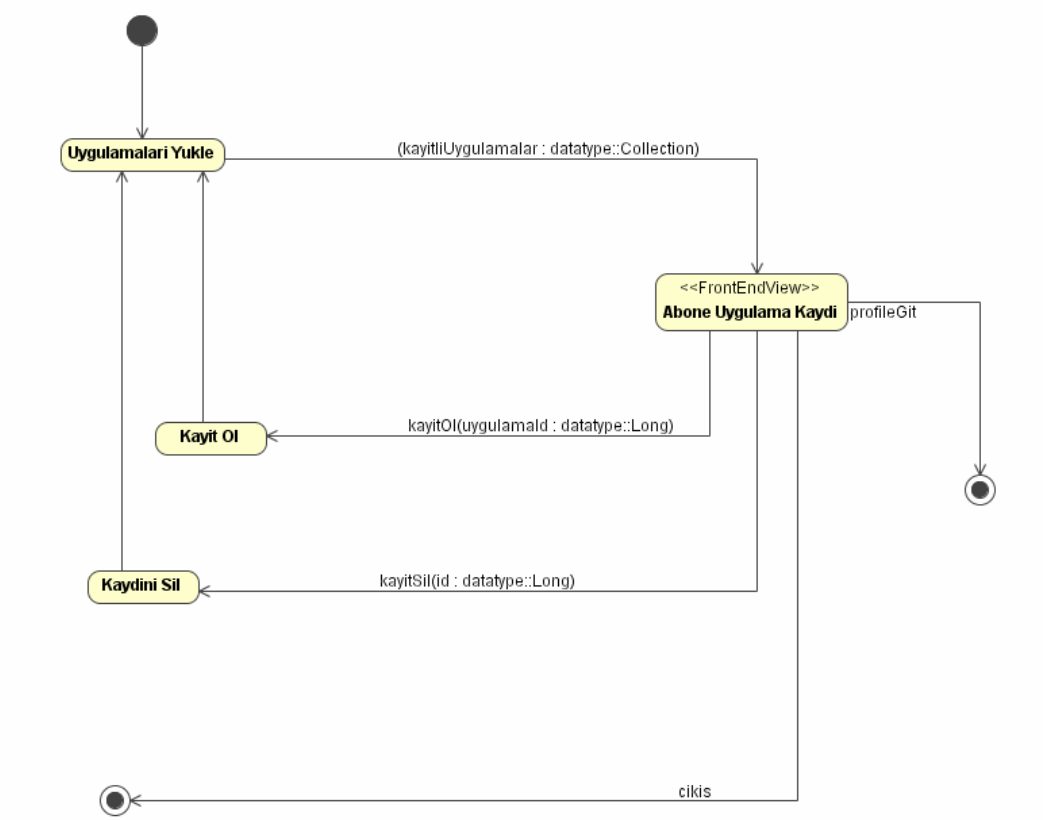
Şekil 8.6 Konum Sunucusu Abone Kontrol Sınıfları

Üç kontrol sınıfı ile <<FrontEndSessionObject>> stereotipi eklenmiş olan *AboneSession* sınıfı arasında bağımlılık ilişkisi bulunmaktadır. Böylece kontrol sınıfları, oturum boyunca *aboneId*'yi muhafaza eden *AboneSession* sınıfına erişebilecektir.

Eğer *msisdn* ve *şifre* geçerliyse abone sisteme girebilecek, değilse hata mesajıyla birlikte aynı giriş sayfasına dönecektir. Bu denetleme, aktivite diyagramında, kontrol metodundan geri dönülen değere göre bir karar noktası ve denetleyiciler (guard) ile sağlanmaktadır. *aboneOlarakGir()* metodunun geri dönüş değeri *başarılı* ise *Uygulama Kayıt İşlemleri* kullanım durumuna yönlendirilmiş olan son duruma gidilecek ve *aboneId AboneSession*'a yazılacak, *başarısız* ise *Giriş Başlangıcı*'na dönülecektir.

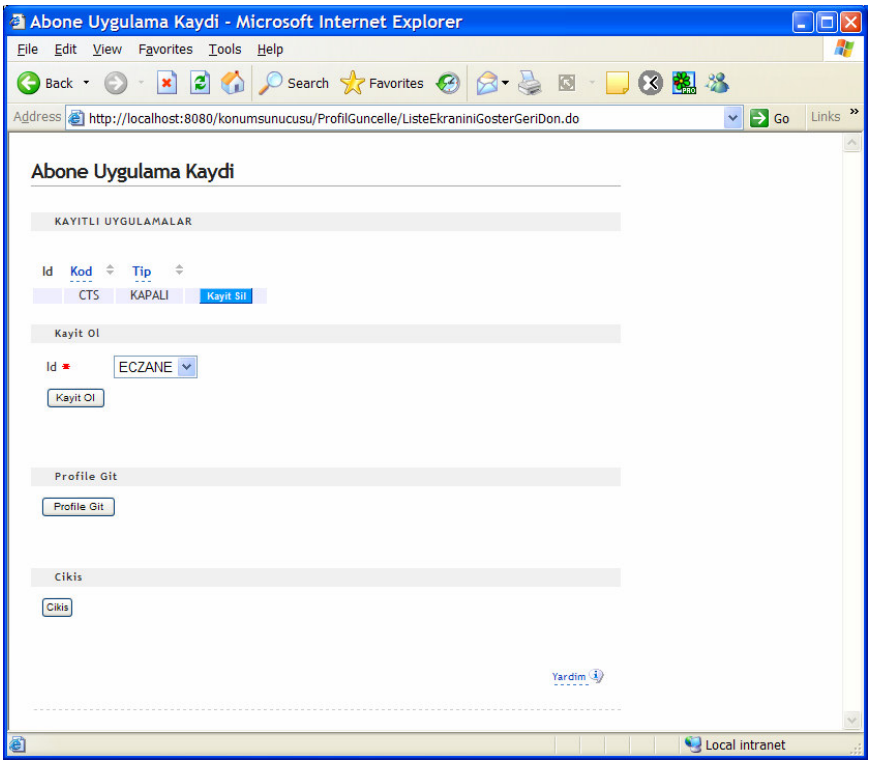
Msisdn-şifre'nin geçersiz olduğunu belirten hata mesajı, *[başarısız]* geçişine bir etiketli değer olarak yazılmıştır (andromda.presentation.action.warning.message= Msisdn-şifre geçersiz).

Şekil 8.7'de *Uygulama Kayıt İşlemleri* kullanım durumu için hazırlanan aktivite diyagramı görülmektedir:



Şekil 8.7 Uygulama Kayıt İşlemleri Aktivite Diyagramı

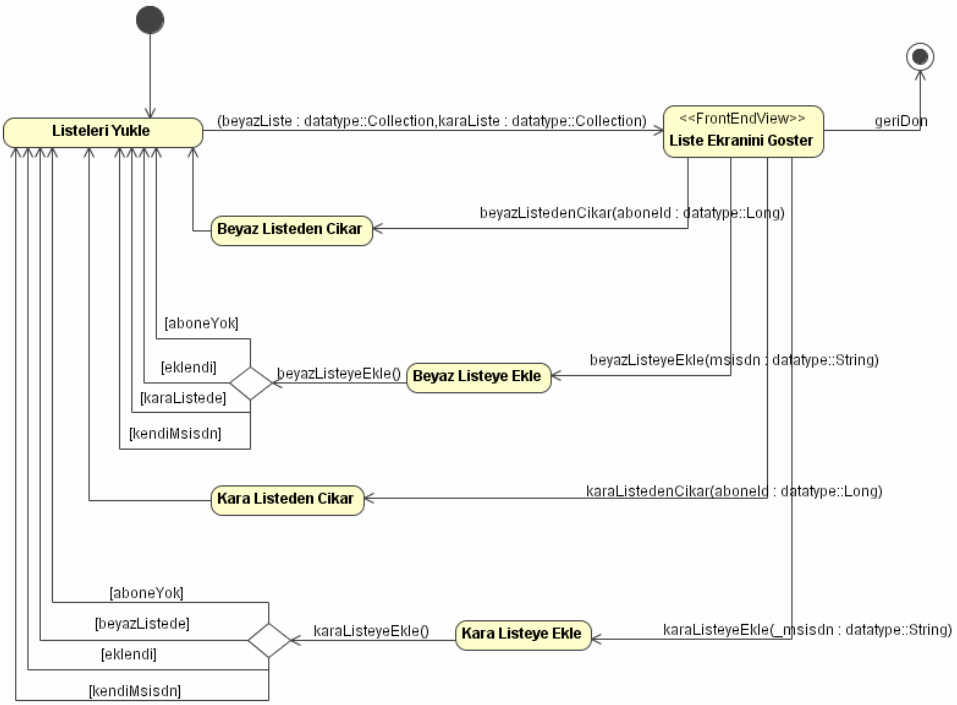
Şekil 8.8'de *Uygulama Kayıt İşlemleri* aktivite diyagramına göre oluşan sayfa görülmektedir:



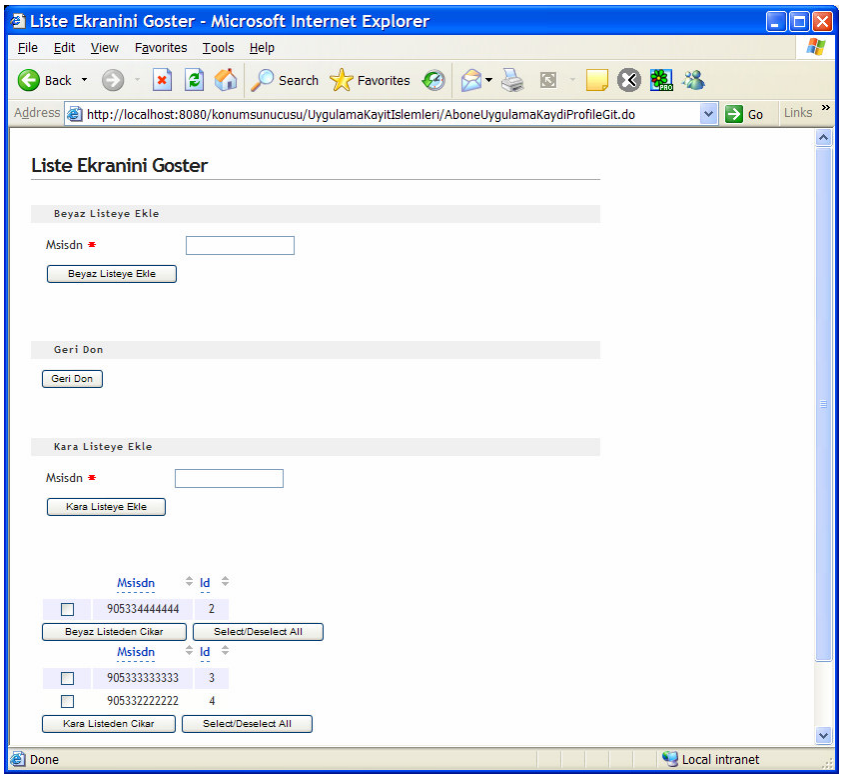
Şekil 8.8 Uygulama Kayıt İşlemleri sayfası

Uygulamaları Yükle eyleminde, aboneye ait uygulamaların yüklenmesi *AboneKontrol* sınıfındaki *uygulamalarıYükle* metodunu çağıran bir ertelenebilir olay (deferrable event) ile sağlanmıştır. *Abone Uygulama Kaydı* eylemine giren geçişteki *kayıtlıUygulamalar* ekranda bir tablo olarak görünecektir. *Kaydını Sil* bu tablodaki seçime bağlı olarak çalışan bir eylemdir. Bütün bu detaylar, geçişler üzerinde yer alan tetikleyecilere ve parametrelerine etiketli değerler eklenerek sağlanmaktadır. *Kayıt Ol* ve *Kaydını Sil* eylemleri de kontrol sınıfındaki ilgili metotları çağırmaktadır.

Şekil 8.9'da beyaz liste ve kara liste ekleme, çıkarma işlemlerini sağlayan Profil Güncelle kullanım durumuna ait aktivite diyagramı görülmektedir. Şekil 8.10'da ise oluşan sayfa görülmektedir.



Şekil 8.9 Profil Güncelle aktivite diyagramı



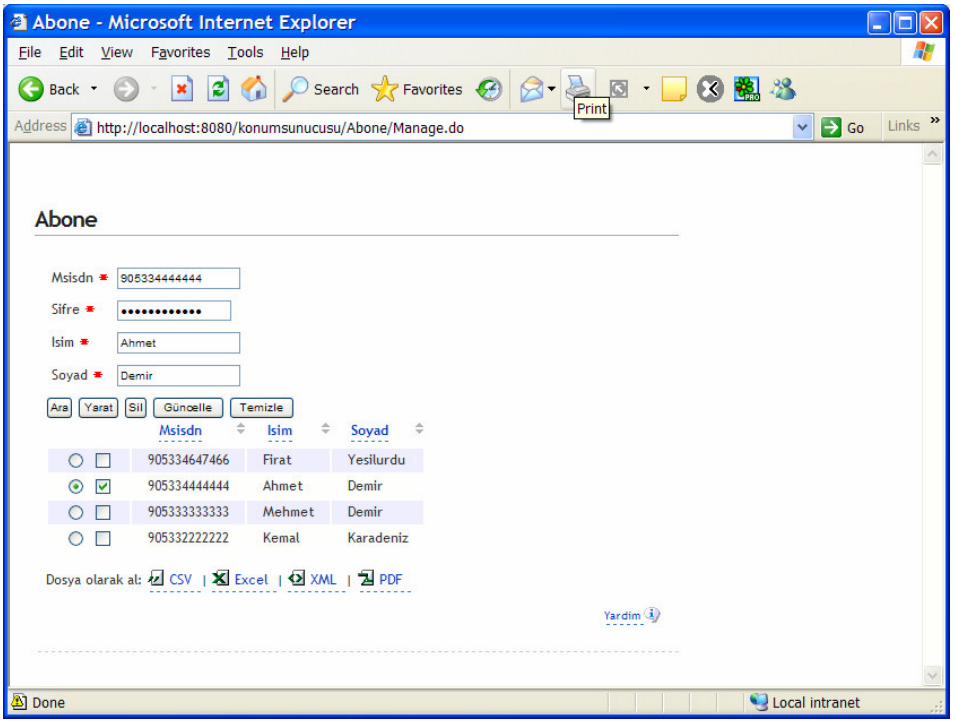
Şekil 8.10 Profil güncelleme sayfası

8.5.3 Konum Sunucusu'nun Geliştirilmesindeki Diğer Aşamalar

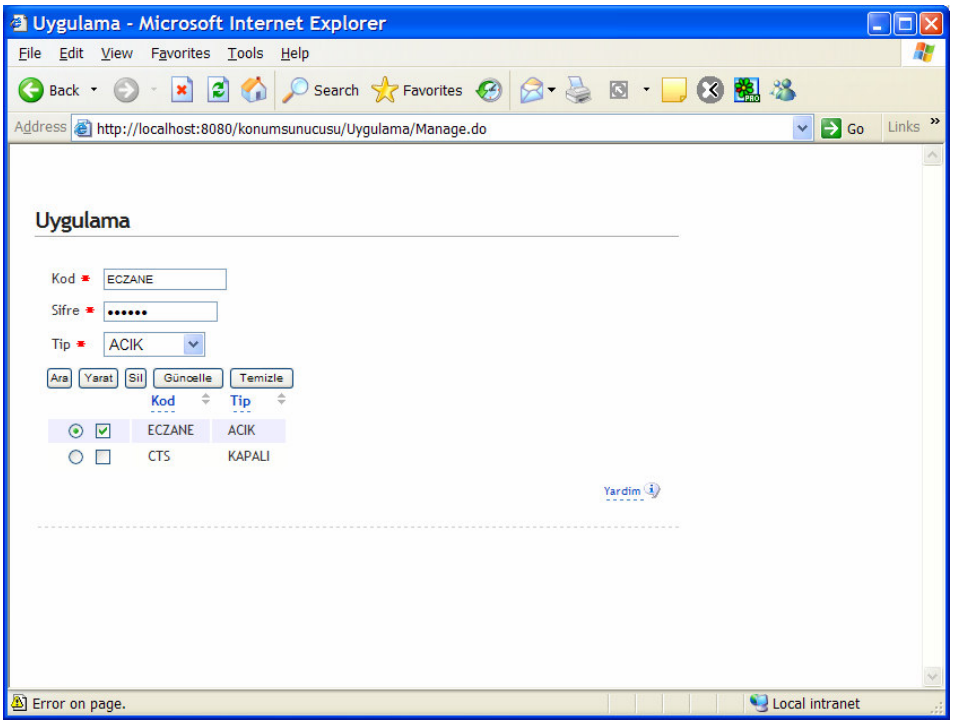
Abone ve Uygulama sınıfları üzerindeki yarat-oku-güncelle-sil-ara tipinde operasyonları sağlayan servisler <<Manageable>> stereotipi sayesinde otomatik oluşturulmuştur. Bu servisler *Web* modülündeki kontrol sınıflarına ait metotlar doldurulurken kullanılarak istenen eylemler gerçekleştirilmiştir.

Konum Sunucusu'nun yönetici arayüzündeki Abone Yönetimi ve Uygulama Yönetimi sayfaları, yine <<Manageable>> stereotipi sayesinde otomatik olarak oluşturulmuştur. Şekil 8.11 ve 8.12'de abone ve uygulama yönetimi arayüzleri görülmektedir.

Konum Sunucusu'nun dışarıya sunduğu servisler de elle yazılmıştır. Bu servisler dış sistemleri taklit etmek maksadıyla önceden hazırlanmış koordinat, adres, haritalar dönecek şekilde yazılmıştır. Bu servislere ait web servisi tanımlama dosyası (WSDL) da tamamen otomatik oluşturulmuştur. Uygulamaların Konum Sunucusu'nu kullanabilmeleri için sadece bu WSDL dosyası yeterlidir.



Şekil 8.11 Abone Yönetimi Arayüzü



Şekil 8.12 Uygulama Yönetimi Arayüzü

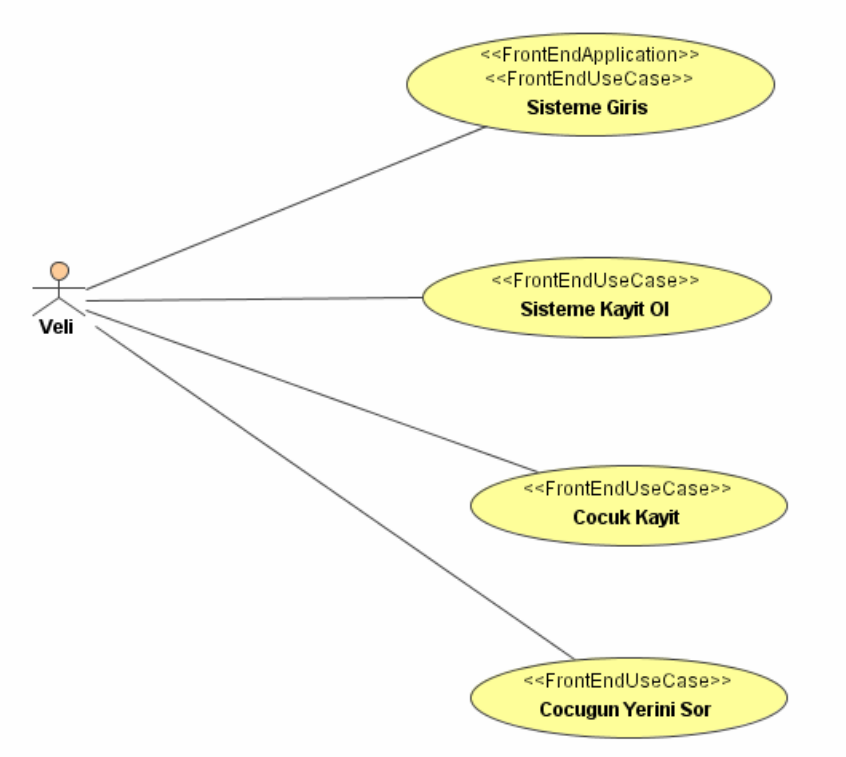
Konum Sunucusu'nun çalışması için oluşturulan EAR dosyası, JBoss Uygulama Sunucusu'na kopyalanmış ve ilgili veritabanı şemalarını oluşturacak komut dosyası çalıştırılmıştır. Veritabanı olarak, JBoss içinde entegre olarak bulunan Hypersonic kullanılmıştır.

8.6 Çocuk Takip Servisi'nin Tasarımı ve Gerçekleştirilmesi

Çocuk Takip Servisi varlık modeli, Veli ve Çocuk varlıklarından oluşmaktadır (Şekil 8.13).

Şekil 8.13 Çocuk Takip Servisi varlıkları

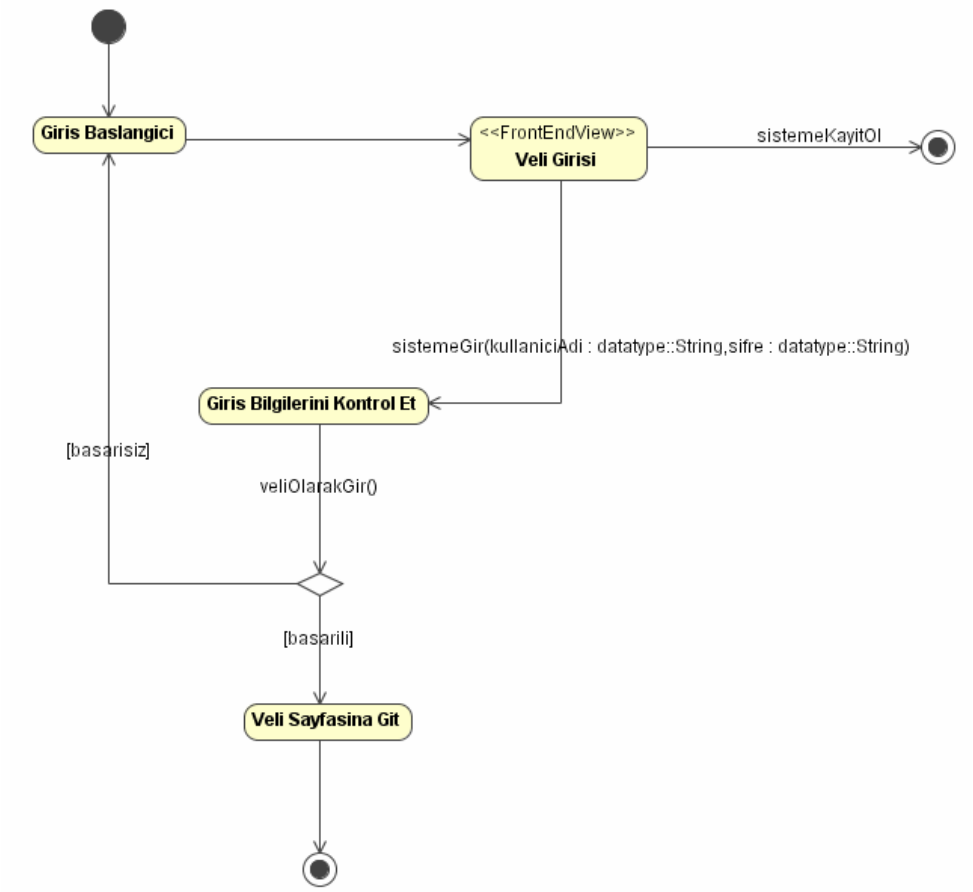
Çocuk Takip Servisi'nde dört adet kullanım durumu bulunmaktadır (Şekil 8.14):



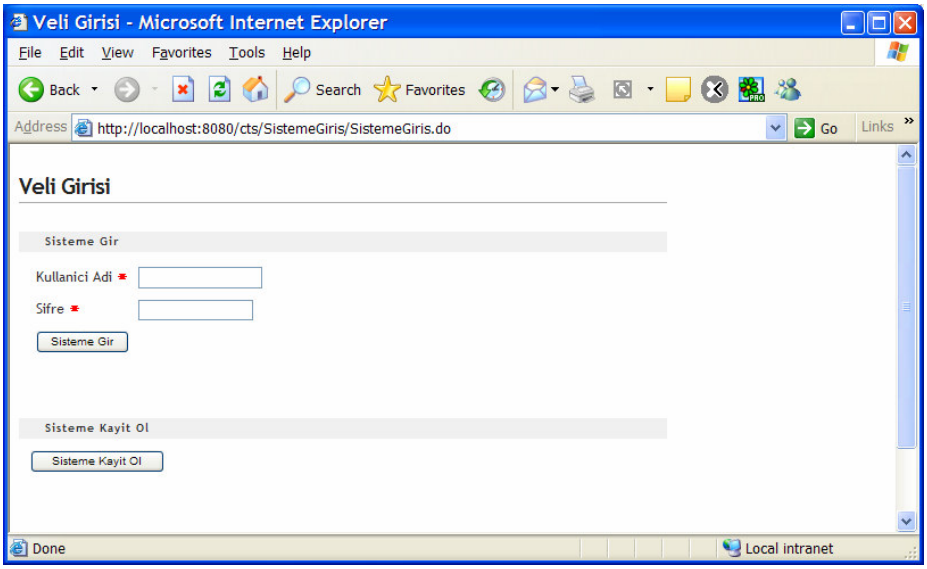
Şekil 8.14 Çocuk Takip Servisi kullanım durumları

Uygulama, <<FrontEndApplication>> stereotipi eklenmiş olan Sisteme Giriş kullanım durumu ile açılmaktadır (Şekil 8.15, Şekil 8.16). Ancak sisteme girmek için önce *Sisteme Kayıt Ol* kullanım durumuna giderek veli kayıt işlemini gerçekleştirmek gerekmektedir (Şekil 8.17, Şekil 8.18).

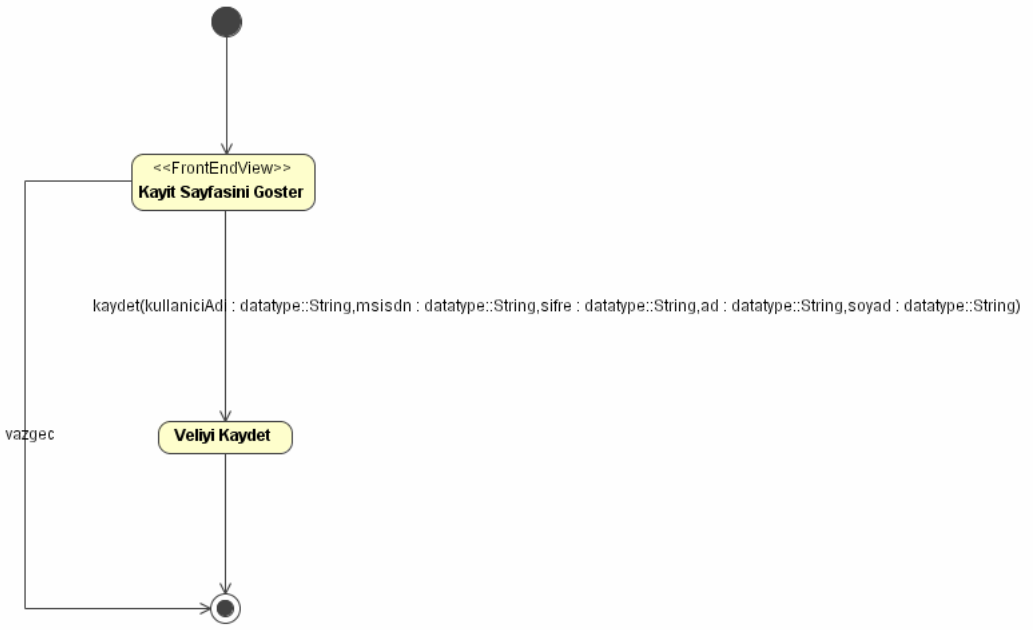
Sisteme kayıt olup giriş yaptıktan sonra *Çocuğun Yerini Sor* kullanım durumuna gelinir (Şekil 8.19, Şekil 8.20).



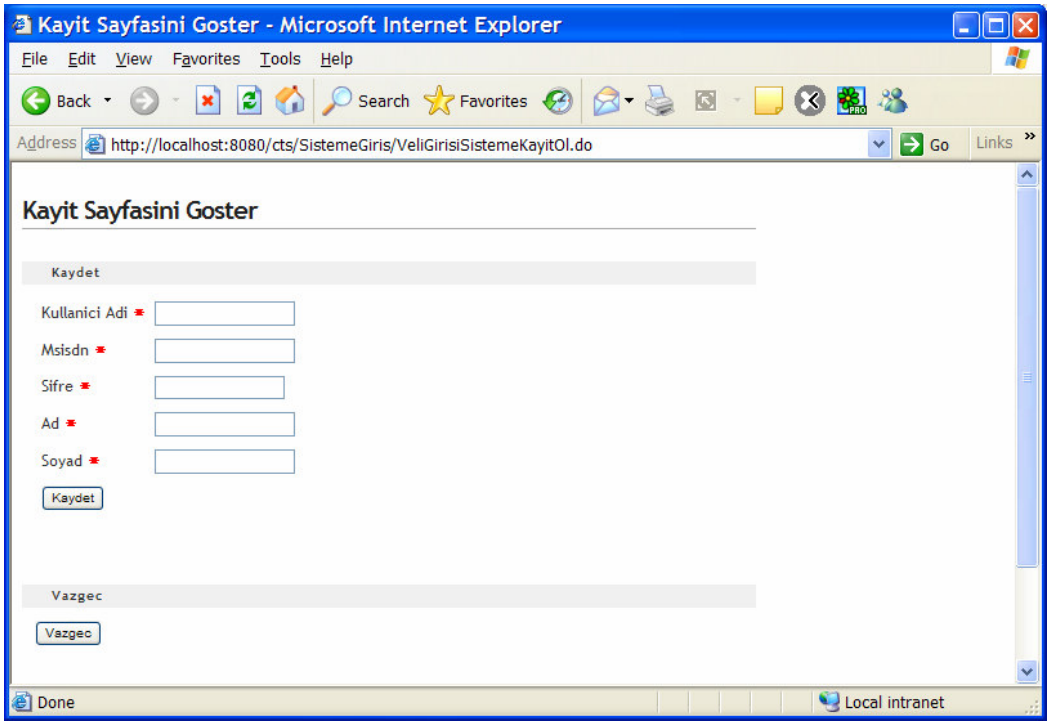
Şekil 8.15 'Sisteme Giriş' aktivite diyagramı



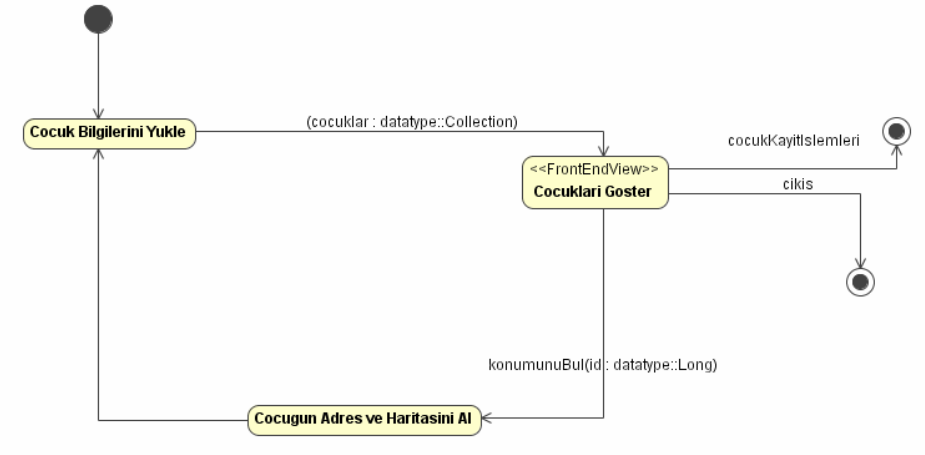
Şekil 8.16 Sisteme giriş sayfası



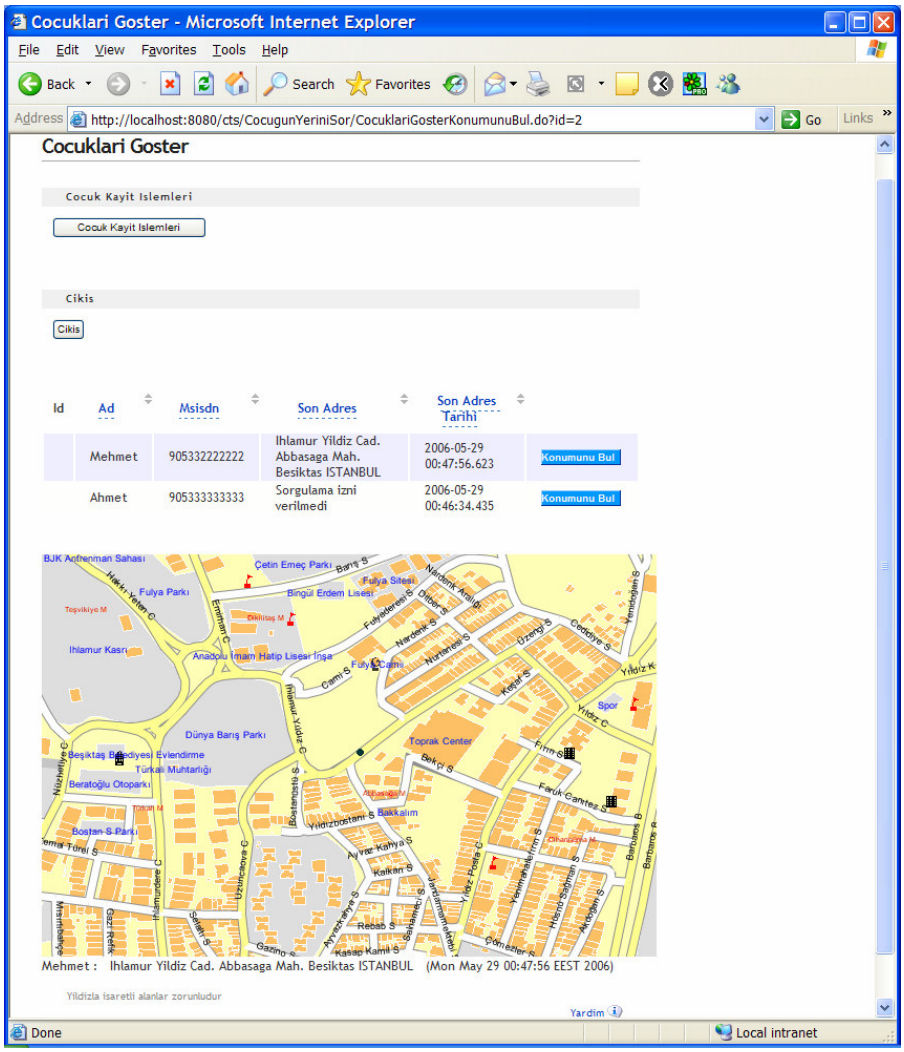
Şekil 8.17 'Sisteme Kayıt Ol' aktivite diyagramı



Şekil 8.18 Sisteme kayıt sayfası

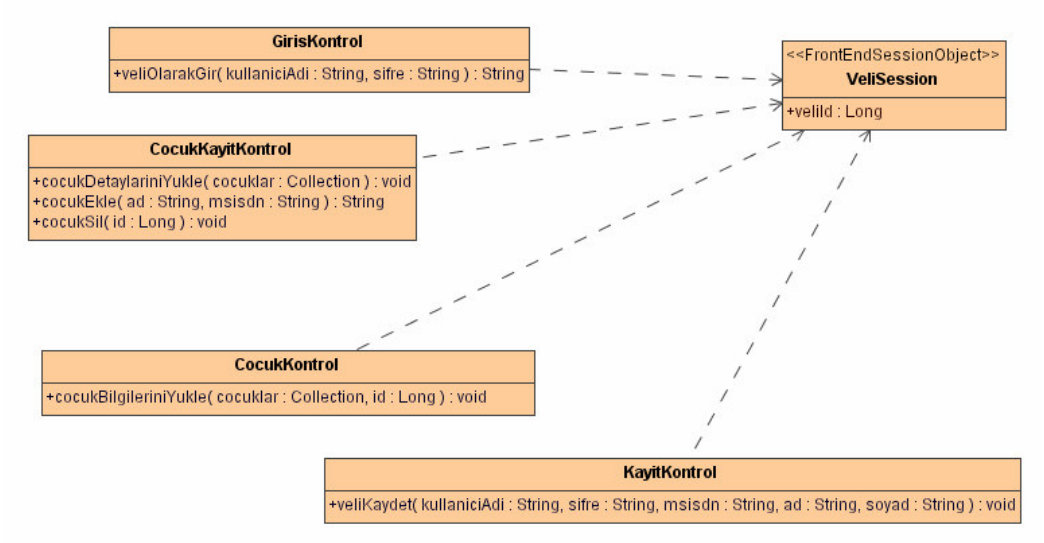


Şekil 8.19 'Çocuğun Yerini Sor' aktivite diyagramı



Şekil 8.20 'Çocuğun Yerini Sor' sayfası

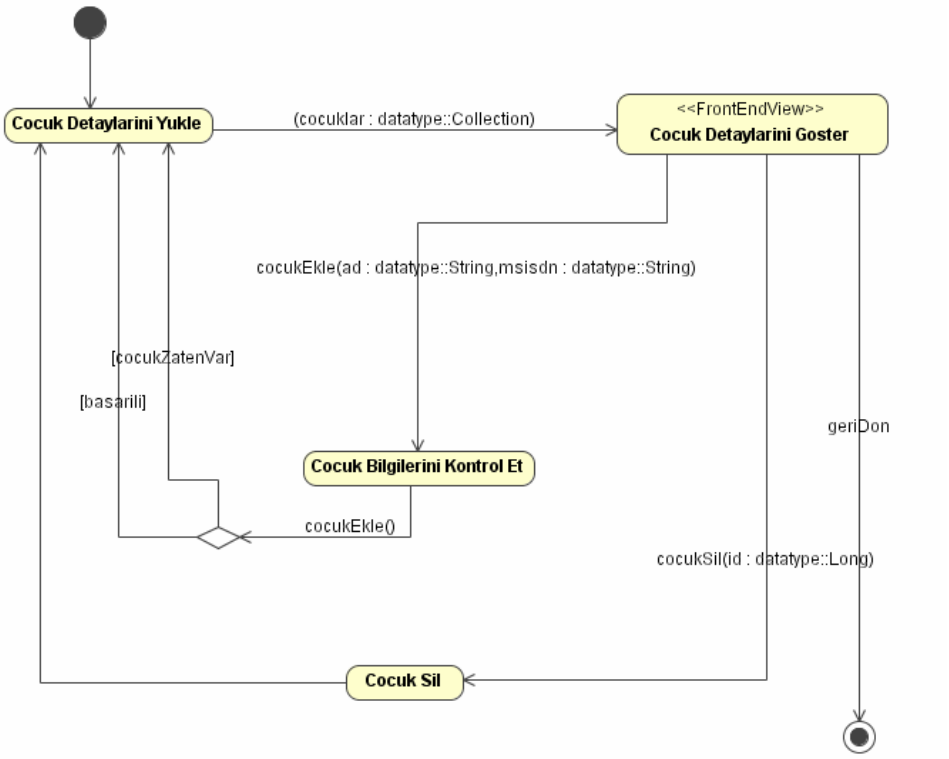
Çocuk Takip Servisi'nden Konum Sunucusu'na yapılan adres ve harita sorguları *KonumunuBul* tetikleyicisi ile gerçekleştirilmektedir. İlgili web servislerini çağıracak yardımcı sınıflar, Eclipse'in JBossIDE eklentisi ile otomatik olarak oluşturulmuş ve *ÇocukKontrol* kontrol sınıfındaki *ÇocukBilgileriniYükle* metodunda kullanılmışlardır. Şekil 8.21'de Çocuk Takip Servisi kontrol sınıfları görülmektedir. Kontrol sınıfı metotları *Veli* ve *Çocuk* sınıflarındaki <<Manageable>> stereotipinin sağladığı servisler ile yarat-oku-güncelle-sil-ara işlemlerini gerçekleştirmektedirler.



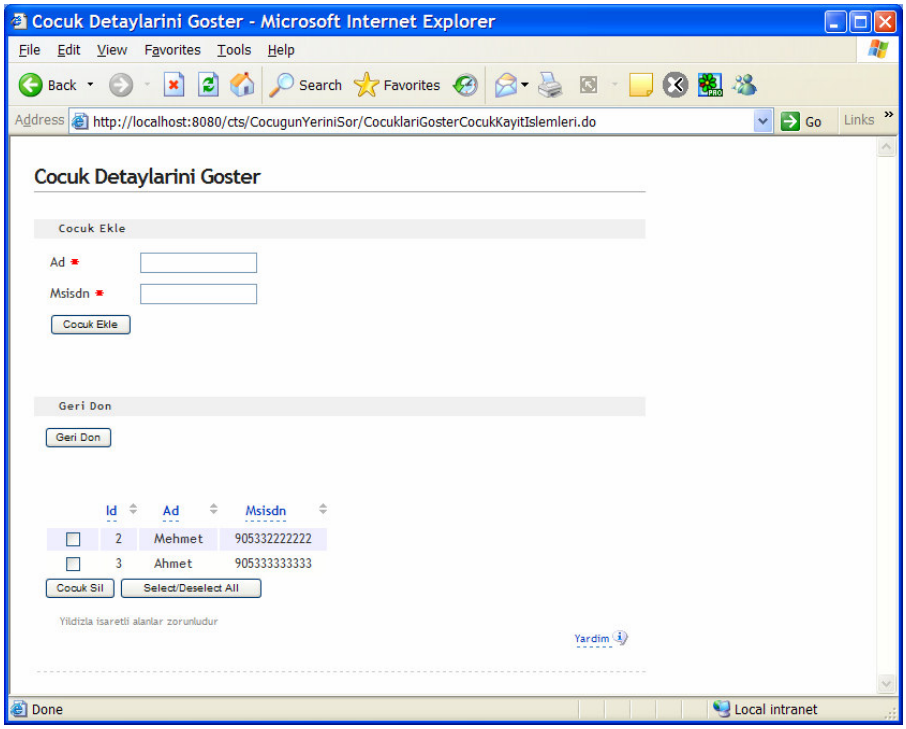
Şekil 8.21 Çocuk Takip Servisi kontrol sınıfları

Burada harita resminin gösterilmesi için jsp sayfalarına elle müdahale edilmiştir. AndroMDA'nın sonradan yapılacak yapılandırma işlemlerinde elle müdahale edilen jsp'lerin üzerine yazmaması için, bunları web modülünde target/jsp dizinine koymak gerekmektedir. Böylece sayfalar üzerindeki görsel iyileştirmeler de bir web tasarımcısı tarafından gerçekleştirilebilecektir.

Veli, *Çocuk Kayıt* kullanım durumuna giderek çocuk ekleme silme işlemlerini gerçekleştirebilir (Şekil 8.22, Şekil 8.23).



Şekil 8.22 'Çocuk Kayıt' aktivite diyagramı



Şekil 8.23 Çocuk kayıt sayfası

9. SONUÇLAR

Bu bölümde, projede elde edilen tecrübelere göre MDA'nın yazılım geliştirmenin bugününde ve geleceğinde neleri değiştirebileceğine değinilecektir.

Yazılım projelerinde değişim kaçınılmazdır. Bir yazılım projesi, geliştirilirken de değişir, kurulup kullanılmaya başlandıktan sonra gelen taleplere göre de değişir, teknolojik ilerlemelere göre de değişir. Değişim ne kadar az emekle gerçekleştirilebiliyorsa yazılım o kadar esnek demektir. Bu açıdan bakıldığında, esneklik orta ve uzun vadede bir yazılımın kalitesini belirleyen önemli faktörlerden birisi haline gelmektedir.

Kötü tasarımlar sonucunda, yeni talepleri karşılayamaz hale gelen yazılım projelerinde kodun en baştan yazılması veya bir teknolojiden diğerine geçildiğinde projeye neredeyse en başından başlanması, yazılım firmalarının sık sık karşılaştıkları durumlardır.

Konum Sunucusu projesinde MDA'nın kullanımı ile, uygulamanın geliştirilme süresi ve yazılım kalitesi açısından oldukça tatmin edici sonuçlar elde edilmiştir. Yazılım katmanlarının kesin çizgilerle birbirinden ayrılarak tasarlanması ve bunun koda da aynı şekilde yansıtılması, klasik yöntemlerle kolay elde edilebilecek bir şey değildir. Bunun için yazılım geliştirme takımının oldukça disiplinli ve ahenkli biçimde çalışması gerekmektedir. Bu disiplin ve ahenk, proje başlangıcında sağlansa bile, ileri safhalarda karşılaşılabilecek zaman baskısı altında sağlanamayabilir.

MDA, uygulamanın prototipini oluşturmakta da oldukça iyi sonuç vermektedir. Prototipler, analiz ve tasarım aşamasında fark edilemeyen bazı gerçeklerin, projenin erken safhalarında fark edilmesinde önemli rol oynar. Çok çeşitli teknolojileri içeren Konum Sunucusu projesinde geliştirme aşama aşama gerçekleşmiş, bazı durumlarda varlık modeline dönülerek değişiklikler yapılmıştır. Bu değişiklikler model üzerinden kolayca yapılabilmiş, kod tekrar üretilerek değişiklikler projeye kolayca yansıtılmıştır.

Elle yazılan kodlar, kodun toplamına göre çok düşük orandadır. Bu kodlarda da otomatik üretilmiş sınıflar kullanıldığı için bütünlük bozulmamıştır.

Web tarafının aktivite diyagramları ile tasarımı, alışılagelen yöntemlere göre farklı olduğu için istenilen sonuçların alınması zaman almıştır. Ancak ilk yapılan Konum Sunucusunun web arayüzünün tasarımı bittikten sonra Çocuk Takip Servisi'nin

tasarımı çok daha kısa sürede gerçekleştirilmiştir.

MDA araçları ticari amaçla kullanılacaksa, yazılım firmasının kendi kurumsal mimarisini oluşturması en doğru yaklaşım olacaktır. Örneğin firmanın, ürünlerinde ortak bir sunuş katmanı tarzını uygulaması, MDA yaklaşımı ile verimli bir şekilde gerçekleştirilebilir.

Yazılım mimarları, firmalarında kurumsal bir mimarinin geliştirilmesini sağlamalıdır. MDA ile yazılım evleri, elde ettikleri tecrübeleri kurumsal bir mimari halinde toplayarak, birçok projede doğrudan kullanabilecek, yeni teknolojilerin kullanılması veya bir projenin bir teknolojiden diğerine geçirilmesi kolaylaşacaktır.

KAYNAKLAR

Eriksson, H., Penker M., Lyons, B. ve Fado, D., (2004) "UML 2 Toolkit", Wiley Publishing, Inc., Indianapolis, Indiana, A.B.D.

Herzum P. ve Sims O., (2000) Business Component Factory: A Comprehensive Overview of Component Based Development for the Enterprise, John Wiley & Sons.

Frankel, David S., (2003) "Model Driven Architecture: Applying MDA to Enterprise Computing", Wiley Publishing, Inc., Indianapolis, Indiana, A.B.D.

Warmer, J. ve Kleppe, A., (2003) "The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition", Pearson Education, Inc., Boston, MA, A.B.D.

İNTERNET KAYNAKLARI

[1] "AndroMDA 3.1 Reference Document", http://galaxy.andromda.org/docs-3.1

[2] No Magic, Inc., "MagicDraw 11.0 User's Guide", http://www.magicdraw.com/files/manuals/11.0/MagicDraw%20UserManual.pdf

[3] OMG, (2003) "MDA Guide V1.0.1", http://www.omg.org/cgi-bin/doc?omg/03-06-01

[4] OMG, (2005) "UML 2.0 Superstructure Specification", http://www.omg.org/cgi-bin/doc?formal/05-07-04

[5] Siegel, J., (2001) "Developing in OMG's Model Driven Architecture", ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf

[6] Soley, R., (2000) "Model Driven Architecture", http://www.omg.org/docs/omg/00-11-05.pdf

[7] "J2EE Patterns: Value Object Pattern", http://java.sun.com/j2ee/patterns/ValueObject.html

ÖZGEÇMİŞ

FIRAT YEŞİLÜRDÜ

Doğum tarihi	11.05.1979	
Doğum yeri	İstanbul	
Ortaokul	1989-1993	Özel Ata Lisesi
Lise	1993-1996	Behçet Kemal Çağlar Lisesi
Lisans	1997-2001	İstanbul Üniversitesi Mühendislik Fak. Bilgisayar Bilimleri Mühendisliği Bölümü
Yüksek Lisans	2001-2006	Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği

Çalıştığı kurumlar

2003-2004	Infotech Bilişim ve İletişim Teknolojileri A.Ş. Yazılım Mühendisi
2004-Devam ediyor	Telenity İletişim Sistemleri San. Tic. A.Ş. Yazılım Geliştirme Ekip Lideri