T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YAZILIMDA ALANA ÖZGÜ MODELLEME

ALPER ÇİFTÇİ

YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ PROGRAMI

DANIŞMAN
PROF. DR. OYA KALIPSIZ

İSTANBUL, 2011

ÖNSÖZ

Tez çalışmam süresince göstermiş olduğu yakın ilgi ve desteğinden dolayı Prof. Dr. Oya Kalıpsız'a ve yüksek lisans öğrenimimde emeği geçen Yıldız Teknik Üniversitesi Elektrik-Elektronik Fakültesi Bilgisayar Mühendisliği bölümündeki tüm öğretim üyelerine teşekkürü bir borç bilirim.

Eğitim hayatım boyunca her türlü özveriyi gösteren ve her zaman destek olan aileme ve yüksek lisans eğitimim esnasında sabırla desteğini esirgemeyen sevgili eşime teşekkür ederim.

Kasım, 2011

Alper ÇİFTÇİ

İÇİNDEKİLER

Sayfa

KISALTMA LİSTESİ

AÖM	Alana Özgü Modelleme
3GLs	Thirth Generation Languages
CASE	Computer-Aided Software Engineering
SDK	Software Development Kit
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
UML	Unified Modeling Language
OMG	Object Management Group
BNF	Backus Naun Form
GOPRR	Graphics Object Property Role Relationship
MOF	Meta-Object Facility
ORM	Object-Relational Mapping
ŞGD	Şablon Görünüm Denetçi
MVC	Model View Controller
SQL	Structured Query Language

ŞEKİL LİSTESİ

Sayfa

ÇİZELGE LİSTESİ

Sayfa

ÖZET

YAZILIMDA ALANA ÖZGÜ MODELLEME

Alper ÇİFTÇİ

Bilgisayar Mühendisliği Anabilim Dalı

Yüksek Lisans Tezi

Tez Danışmanı: Prof. Dr. Oya KALIPSIZ

Günümüzde iş ihtiyaçlarını karşılamak üzere gerekli yazılımların hızlı bir şekilde geliştirilmesi ve aynı zamanda kaliteli olması önem taşımaktadır. Bunun yanı sıra yazılımların ve yazılım geliştirme süreçlerinin değişen gereksinimlere kısa sürede cevap vermesi beklenmektedir. Bu durum yazılım geliştirmede üretkenlik artışı ihtiyacını doğurmaktadır. Yazılımların geliştirilmesinde kullanılan genel amaçlı dillerin üretkenlik artışı için yeterli olmadığı düşünüldüğünden son yıllarda özel amaçlı, alana özgü dillerin ve alana özgü modellemenin kullanımı önerilmiştir.

Alana özgü modelleme model güdümlü bir yazılım geliştirme yöntemidir ve çözümü doğrudan problem alanının kavramları üzerinde tanımlamak suretiyle soyutlama seviyesini mevcut programlama dillerinin üstüne yükseltir. Nihai ürün daha sonra bu yüksek seviye tanımlamalar kullanılarak üretilir. Hem dilin hem de kod üreticilerin sadece tek bir alanın gereksinimlerine uymak durumunda olmaları böyle bir otomasyonu mümkün kılmaktadır.

Bu tez çalışmasında alana özgü modelleme yaklaşımı detaylı olarak incelenmiş ve bu yaklaşım kullanılarak web tabanlı veri yönetim uygulamaları için alana özgü modelleme dili ve kod üretici geliştirilmiştir. Alana özgü modelleme çözümü deneysel olarak uygulanarak yaklaşımın üretkenliğe katkısı klasik geliştirme yöntemi ile karşılaştırılmıştır.

Anahtar Kelimeler: Alana özgü modelleme, alana özgü dil, kod üretme, web tabanlı veri yönetim uygulaması

ABSTRACT

DOMAIN SPECIFIC MODELING IN SOFTWARE DEVELOPMENT

Alper ÇİFTÇİ

Department of Computer Engineering

MSc. Thesis

Advisor: Prof. Dr. Oya KALIPSIZ

Nowadays, software systems need to be rapidly developed to cope with business needs, what is more, their quality must be better. In addition, changing requirements must be taken into account by software and software development processes. Thus, a productivity improvement is required in software development. Since general programming languages and traditional modeling languages like UML have not increased productivity, use of specialized, domain-specific languages and domain-specific modeling (DSM) is suggested in literature.

Domain-specific modeling is model driven software development approach. It raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. The final products are generated from these high-level specifications. This automation is possible because both the language and generators need fit the requirements of only one company and domain.

In this thesis, domain-specific modeling approach is examined in detail, and a domain specific modeling language and code generator is developed for web based data management applications. An experimental application is developed using domain specific solution and traditional development method, then productivity gain is investigated, comparing DSM solution with traditional development approach.

Keywords: Domain-specific modeling, domain-specific language, code generation, web based data management

BÖLÜM 1

GİRİŞ

Yazılım geliştirme tarihi boyunca yazılım mühendisleri yazılımda üretkenliğin artırılmasını her zaman soyutlama seviyesinin artırılmasında aramışlardır. Bulunan her yeni soyutlama daha sonra otomatik olarak bir öncekilere dönüştürülmüştür. Günümüzde ise geleneksel programlama ve modelleme dillerindeki gelişmelerin söz gelimi assembly dilinden üçüncü nesil dillere (3GLs) geçişte elde edilen üretkenlik artışı ile karşılaştırıldığında göreceli olarak üretkenliğe daha az katkı sağladığı görülmektedir. Geliştiriciler artık tek bir satır kod ile daha önce birçok satır kod ile sağlanabilen işlevselliği elde edebilmektedirler. Benzer bir üretkenlik kazanımının örneğin UML veya Java ile sağlanabildiğini söylemek ise oldukça güçtür [1], [15].

Yazılım geliştirmede soyutlama seviyesini artırmak için literatürde çeşitli yaklaşımlar önerilmiştir. Bunlardan biri de alana özgü modelleme yaklaşımıdır. Bu yaklaşımın temel hedefi alan modellerinden çalışabilen yazılımın kodunu üretmektir. Bu hedefi gerçekleştirmek için alana özgü bir modelleme dili, alana özgü kod üretici ve bazı durumlarda da ek olarak bir alan çerçevesi veya hedef ortam kullanılarak bir AÖM çözümü oluşturulur [1].

1.1 Literatür Özeti

Alana özgü modelleme son yıllarda popülerlik kazanan bir konu olmakla birlikte bu yaklaşımın iki temel unsuru olan üst modelleme ve kod üretimi yazılım mühendisliğinde daha önce de kullanılan kavramlardır.

AÖM yaklaşımına temel teşkil eden üst modelleme dillerinin kullanımı 1990'lı yıllarda görülmektedir. Tolvanen ve Lyytinen tarafından yapılan çalışmada üst modelleme

yaklaşımının CASE araçlarında kullanımı önerilmiştir [2]. Daha sonra Kelly, Lyytinen ve Rossi üst modelleme dilinin kullanıldığı tümüyle konfigüre edilebilir olan MetaEdit+ adlı CASE aracı geliştirmiştir [3]. Hillegersberg, Kumar ve Welke de nesneye dayalı analiz ve tasarım metodolojilerinin nesneye dayalı dillere uygunluğunu değerlendirmek üzere üst modellemeyi kullanmıştır [4].

Fonksiyonel kodun bir kısmını üreten standart CASE araçları ve metodları uzun yıllardır kullanılmaktadır. Kodun tamamını üreten gerçek bir kazanım ilk olarak Harel'in genel olarak durum diyagramlarının bazı formlarına dayanmaktadır [5]. Harel'in bu çalışması daha sonra Statemate and Rhapsody araçlarında kod üretiminin ilerlemesini sağlamıştır [6]. Selic, Gullekson ve Ward da kod üretiminde iyi sonuçlar elde etmişlerdir [7]. Modern kod üretiminde temel çalışmalar ise Burst, Spitzer, Wolff ve Müller–Glaser tarafından yapılmıştır [8].

AÖM yaklaşımı, temelleri eskiye dayansa da esas popülerliğini 2000'li yıllarda kazanmıştır. Tolvanen ve Kelly bu yıllarda yaklaşıma en fazla katkıda bulunanlar olmuştur. AÖM yaklaşımının kavramsal temellerini [9], [11] ve [12] de, AÖM yaklaşımının uygulanması neticesinde elde edilen kazanımları [10] ve [13] te ve alana özgü modellemede entegrasyon çalışmalarını [14] ve [16] da belirtmişlerdir.

AÖM çözümü oluşturmak için son yıllarda MetaCase MetaEdit+, Microsoft Visual Studio Visualization and Modeling SDK ve Eclipse EMF/GMF gibi araçlar üretilmiştir.

1.2 Tezin Amacı

Bu tez çalışmasında alana özgü modelleme yaklaşımının detaylı olarak incelenmesi, özellikleri ve getirilerinin araştırılması hedeflenmiştir. Ayrıca yaklaşımı kullanarak web tabanlı veri yönetim uygulamaları için alana özgü bir modelleme dili ve alana özgü kod üreticinin geliştirilmesi ve geliştirilen bu alana özgü modelleme çözümünün deneysel olarak uygulanarak yaklaşımın üretkenliğe ve kaliteye katkısının araştırılması çalışmanın amaçlarındandır.

Alana özgü modelleme yaklaşımını Türkçe olarak, gerçek bir uygulama üzerinde anlatan çalışmaların oldukça az olması, alana özgü modellemenin öğrenilmesini ve kullanımını

yavaşlatmaktadır. Bu tezde anlatılanların, alana özgü modellemenin etkin bir şekilde kullanılmasına ve anlaşılmasına yardımcı olması çalışmanın bir diğer amacıdır.

1.3 Orjinal Katkı

Kullanımı yaygın olan web tabanlı veri yönetim uygulamaları genelde katmanlı mimari kullanılarak ve tam otomatik olmayan yöntemler kullanılarak geliştirilmektedir. Bu tür uygulamaların geliştirilmesi için kullanılan mevcut yöntemlerin verimliliği ise tartışma konusudur.

Bu tür uygulamaların geliştirilme sürecini basitleştirmek ve hızlandırmak üzere tez çalışmasında yüksek seviye modellerden program koduna dönüşüm yapan bir AÖM çözümü alana özgü bir modelleme dili ve alana özgü kod üreticinin geliştirilmesi suretiyle oluşturulmuştur. Bu çözüm ile şablon-görünüm-denetçi mimarisindeki hedef ortam için otomatik ve tüm kod üretimi sağlanmıştır.

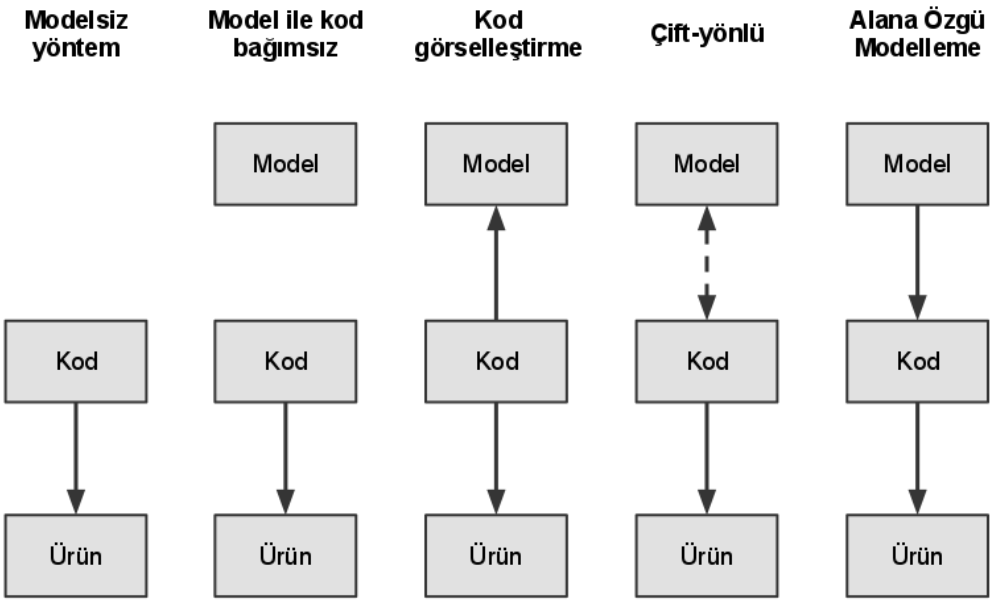
BÖLÜM 2

ALANA ÖZGÜ MODELLEME

Alana özgü modelleme çözümü doğrudan problem alanının kavramları üzerinde tanımlamak suretiyle soyutlama seviyesini mevcut programlama dillerinin sağladığı soyutlama seviyesinin üstüne yükseltir. Nihai ürün daha sonra bu yüksek seviye tanımlamalar kullanılarak üretilir. Hem dilin hem de kod üreticilerin sadece tek bir alan veya şirketin gereksinimlerine uymak durumunda olmaları bu şekilde bir otomasyonu mümkün kılmaktadır. Alan teknik bir alan olabileceği gibi, fonksiyonel bir iş alanı da (business domain) olabilmektedir. Pratikte her AÖM çözümü daha iyi otomasyon imkanı sağlaması ve tanımlanması daha kolay olması nedeniyle mümkün mertebe daha dar bir alana odaklanır. AÖM çözümleri genelde belirli bir ürün, ürün hattı, ortam veya platform ile ilişkili olarak kullanılır [38].

2.1 Kod-Model İlişki Türleri

Programcılar genellikle modelleme ile kodlamayı birbirinden ayırırlar. Modeller sistem tasarlamak ve sistemi daha iyi anlayabilmek, ihtiyaç duyulan işlevselliği belirleyebilmek ve dokümantasyon oluşturabilmek için kullanılırlar. Daha sonra tasarımları gerçeklemek için de kod yazılır. Hata ayıklama, test etme ve bakım işlemleri de aynı şekilde kod seviyesinde yapılır. Model ve kod sıklıkla birbiri ile bağlantısız olarak varsayılsa da bu iki süreci birbiriyle bağlantılı hale getiren farklı yöntemler de mevcuttur. Şekil 1'de bu farklı yaklaşımlar görülmektedir [1].



Şekil 2. 1 Kod-model bağlantı yöntemleri

En temel yöntem herhangi bir model üretmeden tüm işlevselliğin doğrudan kodda belirtildiği yöntemdir. Eğer geliştirilen özellik küçükse ve işlevsellik direkt olarak kod içinde ifade edilebiliyor ise bu yaklaşım işe yaramaktadır. Çünkü programlama ortamları bir programlama dili ile yapılmış kodu çalıştırılabilir programa veya başka bir ürüne çevirebilmektedirler. Kod daha sonra test edilebilir ve hatalar ayıklanabilir, değişiklik ihtiyacı doğması durumunda da, çalıştırılabilir program değil kod değiştirilir [1].

Çoğu yazılım geliştirici bununla beraber model de üretir. Kodlamada kullanılan kavramlar çoğu durumda gereksinimlerden ve gerçek problem alanından oldukça uzaktır. Modeller soyutlama seviyesini artırmak ve gerçekleme detaylarını gizlemek için kullanılırlar. Bununla birlikte geleneksel geliştirme sürecinde model tamamıyla koddan ayrı tutulur ve dolayısıyla modelden koda otomatik bir dönüşüm söz konusu değildir. Bunun yerine programcılar uygulamayı kodlarken ve çalıştırılabilir yazılımı üretirken modelleri okur ve yorumlarlar. Gerçekleme sürecinde modeller güncellenmez ve kodlama bittiğinde de sıklıkla göz ardı edilir. Bunun nedeni modellerden elde edilecek faydanın modeli sürekli güncel tutmanın maliyetini karşılamamasıdır. Aynı bilgiyi iki ayrı

yerde tutmanın bakım maliyeti sürecin otomatik olmaması ve hataya açık olması nedeniyle yüksektir [31], [32].

Modeller aynı zamanda tasarlanıp üretildikten sonra yazılımı anlamaya çalışmak için tersine mühendislikte de kullanılırlar. Kod görselleştirme olarak bilinen bu yöntem model tabanlı dokümantasyon üretmekte ve modelde kullanılmak üzere çeşitli kütüphaneler veya bileşenlerin koda dahil edilmesinde kullanışlı olmaktadır. Bununla beraber bu tip modeller kodda yapıldığı gibi yazılımı gerçeklemek, test etmek ve hata ayıklamakta kullanılmazlar.

Çift yönlü yapı aynı bilgiyi kod ve model gibi iki ayrı yerde tutma işini otomatikleştirmeyi amaçlar. Çift-yönlü yapı sadece model ve kod formatlarının birbirine çok benzer olması halinde ve geçişler arasında bilgi kaybı yaşanmadığı durumda faydalı olabilmektedir. Yazılım geliştirmede çok kısıtlı kullanımı bulunmaktadır. Örneğin, veritabanından bir şema modeli oluşturulabilir ve modelden bir veritabanı şeması üretilebilir. Çift yönlü yapı modelleme dilleri ise programlama dillerinin detaylarını tamamıyla karşılayamadığından problemlidir. Genellikle sadece sınıf çatıları modellerde gösterilir, davranış, etkileşim ve dinamikler gibi geri kalan kısımlar çift yönlü süreç içinde yer almaz ve kodda kalırlar. Bu kısmi bağlantı Şekil 1'de kesik çizgi ile ifade edilmiştir. Daha detaylı incelenecek olursa, yapısal kod ile model arasındaki ilişkinin de problemsiz olmadığı görülür. Örnek olarak, sınıf diyagramlarında kullanılan sahiplik, meydana gelme gibi farklı ilişki türlerinin program koduyla ne şekilde ilişkili olduğunu gösteren iyi tanımlanmış eşlemeler bulunmamaktadır. Kodda bu ilişki türleri açık bir şekilde belirtilmez. Bunu çözmek için kodun sadece açıklayıcı belirli bölümleri modelde gösterilir ki bu da aynı bilgiyi iki yerde göstermek şeklinde olmaktadır. Dolayısıyla çift yönlü yapı da soyutlama seviyesini artırmamakta ve bilgi saklama işlevini tam olarak gerçekleştirememektedir.

Model güdümlü geliştirmede ise modeller birincil kaynaktır, kaynak kod yerine kaynak modeller kullanılır. Model bir defa oluşturulduğunda hedef kodlar otomatik olarak üretilebilir, daha sonra çalıştırmak için derlenebilir veya yorumlanabilir. Bir modelleyicinin perspektifinden bakıldığında üretilmiş kod tamdır ve üretildikten sonra herhangi bir nedenle değiştirilmesine gerek yoktur. Burada sadece modelin değil, kod

üretici ve alan uygulama çatısının da bu otomatikleştirilmiş süreci desteklemesi gerekmektedir. Aksi takdirde soyutlama seviyesi yükseltilmiş olmayacaktır [33].

Model güdümlü geliştirmede soyutlama seviyesini artırmak için hem modelleme dili hem de kod üretici sadece belirli türden uygulamalar geliştirmek üzere alana özgü olmak durumundadır. Daha dar bir ilgiler bölgesine odaklanmak dili problem alanına yaklaştırmakta ve tüm kod üretimini mümkün kılabilmektedir. Bütün kodun üretilmesi genel amaçlı programlama dillerinde ise imkansız olmasa da oldukça zordur.

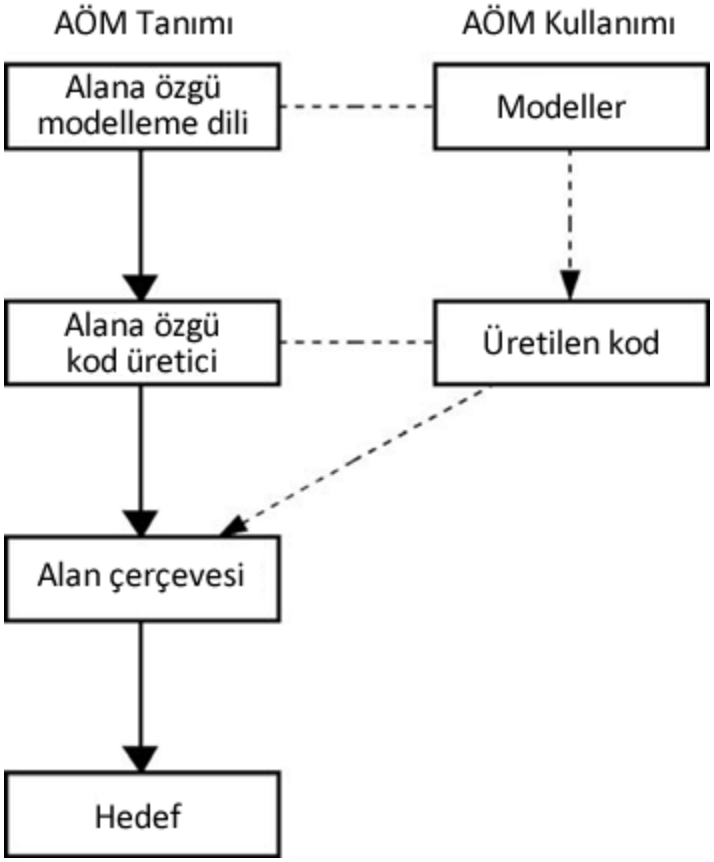
2.2 Alana Özgü Modelleme Bileşenleri

AÖM'nin temel bileşenleri diller, modeller, kod üreticiler ve alan çerçevesidir.

- Alana özgü modelleme dili belirli bir alandaki karmaşıklıkla başa çıkabilmek için bir soyutlama mekanizması sağlar. Programlama dilinin kavramları yerine uygulama alanındaki öğeleri ifade eden kavramlar ve kurallar kullanılarak bu mekanizma sağlanır. Genelde alanın temel kavramları modelleme dili nesnelerine karşılık gelir. Diğer alan kavramları da nesne özellikleri, bağlantılar, alt modeller ve diğer dillerdeki modeller ile bağlara karşılık olarak kullanılır. Böylece dil, uygulama geliştiricilere doğrudan alan kavramları ile çalışma imkanı sunar. Dil ilgili gösterimler ve araç desteği ile birlikte bir üst model olarak tanımlanır.
- Bir kod üretici modellerden bilginin nasıl elde edileceğini ve bu elde edilen bilginin koda nasıl dönüştürüleceğini belirler. En basit durumda, her modelleme sembolü, bu sembole girilen değerleri içeren belirli, sabit bir kod üretir. Kod üretici aynı zamanda semboldeki değerlere, sembolün diğer semboller ile ilişkilerine ya da modeldeki başka bir bilgiye bağlı olarak farklı kod da üretebilir. Daha sonra bu kod alan çerçevesi ile birlikte derlenerek çalıştırılabilir uygulama elde edilir. Bir AÖM çözümü oluşturulurken, üretim işleminden sonra üretilen kodun değiştirilmesi veya geliştirilmesinde elle yapılacak bir işlemin gerekmiyor olması hedeflenir. Böylece üretilmiş kod söz gelimi bir C programı derlemesinde ortaya çıkan ".o" uzantılı dosyalar gibi nihai ürüne gidişte bir ara ürün niteliğini taşır.

- Bir alan çerçevesi üretilmiş kod ile kodun çalışacağı platform arasındaki arayüzü sağlar. Bazı durumlarda böyle bir çerçeveye ihtiyaç duyulmaz ve üretilen kod doğrudan platform bileşenlerini çağırabilir. Bununla birlikte bazı yardımcı kodlar veya bileşenler tanımlamak üretilmiş kodu daha basit hale getirebilmek için sıklıkla başvurulan bir yöntemdir. Alan çerçevesi bileşenlerden ilgili alan kodunda yaygın olarak kullanılan bireysel programlama dili ifadelerine kadar farklı boyutlarda oluşturulabilir. Bunlar aynı zamanda daha önceki geliştirme faaliyetlerinin ve ürünlerinin neticesinde hali hazırda var olan bileşenler de olabilir.

AÖM tanımı alana özgü modelleme dili, alana özgü kod üretici, alan çerçevesi ve hedef ortamı içerir ve AÖM çözümünü oluşturan kişileri ilgilendirir. AÖM kullanımı ise modeller ve üretilmiş kodu içerir. Bunlar AÖM kullanarak uygulama geliştiren kişilerin gördüğü kısımlardır (Şekil 2.2).



Şekil 2. 2 AÖM tanımı ve kullanımı

2.2.1 Dil

Geliştiriciler için en fazla görünen kısım olan dil, uygulama geliştirmede belirli bir soyutlama sağlar. AÖM'de programcıların kaynak kod olarak görecekleri tanımlamalar yapılır. Dil doğru bir şekilde düzenlenirse, belirli bir problem alanının koşullarını ve kavramlarını uygular. Bu aynı zamanda alana özgü bir dilin başka problem alanlarında kullanımının uygun olmayacağı anlamına da gelir.

Genelde alanın temel kavramları modelleme dili nesnelerine karşılık gelir. Diğer alan kavramları da nesne özellikleri, bağlantılar, alt modeller ve diğer dillerdeki modeller ile bağlara karşılık olarak kullanılır. Böylece dil, uygulama geliştiricilere doğrudan alan kavramları ile çalışma imkanı sunar. Dilin modelleme kavramları, hesaplama modeli ve gösterim sembolleri gibi özellikleri sayesinde daha dar bir alana odaklanma sağlanmış olur.

2.2.1.1 Dilin temelleri

Genel amaçlı dillere uygulanan tanımlamalar benzer şekilde alana özgü diller için de uygulanır. Modelleme dilleri tipik olarak sentaks ve semantikten meydana gelir. Sentaks tarafında, soyut ve somut sentaks ayrımına da gidilebilir. İlk olarak dilin yapısı ve gramer kuralları belirlenir. Daha sonra dilin kullandığı gösterim sembolleri ve biçimleri ortaya çıkarılır. Tasarım soyutluğunu artırmak ve daha düzgün bir kod üretebilmek için, genellikle sentaksın ve semantiğin genişletilmesine ihtiyaç duyulur.

Sentaks: Sentaks dilin kavramsal yapısını ifade eder. Modelleme dilinde dilin yapıları, özellikleri ve birbirlerine olan bağlantıları sentaks ile ifade edilir. AÖM'de modelleme yapıları ideal olarak doğrudan problem alanından gelir. Bir modelleme dilinin soyut sentaksı normal olarak bir üst model içinde ifade edilir.

Modelleme dilinin sentaksı yalnızca belirli kelimelerden ibaret değildir. Modeller oluşturulurken izlenmesi gereken bazı gramer kuralları da sentaks tarafından karşılanır. AÖM'de bu kurallar problem alanından çıkarılır ve modelleme kavramlarına ilişkiler şeklinde dilde tanımlanırlar. Modellerden hatalı kod üretmeyi engellemek için bu kurallar kullanılır. Bu kuralları modelleme aşamasında konumlandırmak aynı zamanda

kod üreticinin gerçeklenmesini de kolaylaştırır. Zira bu durumda kod üretici modelin geçerliliğini kontrol etmek durumunda kalmaz. AÖM'de kurallar mümkün olduğunca hataları tespit etmek ve önlemek üzere düzeltilme maliyetinin en düşük olduğu erken safhalarda kontrol edilir. Bunun alternatifi hataları kod üretme aşamasında ve üretilmiş kodda bulmaktır. Kuralları kod üretici değil de dil içinde konumlandırmak birden fazla kod üreticinin olması durumunda daha iyi bir hassasiyet sağlasa da, modeli bir kere kontrol ettikten sonra her kod üretici için ayrı ayrı kontrol etmek daha doğru bir yaklaşımdır.

Dilin kuralları genelde modellerin nasıl yapılacağı ile ilgili kısıtlar koyar. Geçerli değerleri, kavramlar arasındaki ilişkileri ve kavramların nasıl kullanılabileceğini tanımlar. Kurallar katı model doğruluk kurallarından ve tutarlılık kontrollerinden belirli bir modelleme yolunu zorlamaktan ziyade geliştirmeye yardımcı kurallara kadar çeşitli seviyelerde olabilir. Kurallar bir kere tanımlandığında modelleme dili bütün geliştiricilerin aynı alan kurallarını takip ederek geliştirme yaptıklarını garanti eder. Kurallar yine tasarım büyüklüğünü önemli ölçüde düşürür ve tasarımcıların sadece uygun uygulamalar yaptığından emin olunmasını sağlar. Geliştirilmesi gereken uygulamaların sahası genişletilmek istendiğinde, modelleme dili daha sonra genişletilebilir.

Semantik: Her modelleme kavramının bir anlamı (semantik) vardır. Bir modele eleman eklendiğinde veya modelin elemanları birbirine bağlandığında bir anlam oluşturulmuş olur. AÖM'de modelleme dilinin semantiği doğrudan problem alanından gelmektedir. Modelleme dilindeki kavramlar uygulama alanında zaten tanımı olan anlamlara sahiptir. Genel amaçlı programlama dillerinde ise durum bu şekilde değildir. Bu dillerde dilin semantiği belli bir problem alanına karşılık gelmez ve dilin semantiğinin ve kavramlarının problem alanı ile eşlenmesi işi geliştiricilere bırakılır. Modelleme dili kullanımı üzerine yapılan [17]'deki çalışmada görülmüştür ki her geliştirici bu eşleme işlemini farklı şekilde yapmaktadır. Genel amaçlı dilleri kullanan modelleyicilerin aynı problem için farklı türlerde modeller ortaya koyması kaçınılmazdır. AÖM ise modelleyicinin direk problem alanından gelen kavram ve semantikleri temel almayı amaçlaması yönüyle farklıdır.

Dilde problem semantiğinin kullanımı sadece kavramlar ile kısıtlı değildir. Aynı zamanda bazı kurallar ile beraber modelleme yapıları arasındaki bağlantılar da problem semantiğinden gelir.

Problem alanının semantiği AÖM semantiği için tek kaynak değildir. Kod üretim amaçlı diğer tüm modelleme dillerinde olduğu gibi gerçekleme tarafının semantiği de hesaba katılmak durumundadır. Burada modelleme yapılarının sözkonusu problem alanı ile nasıl eşleşeceği sorusuna cevap aranır. Bu eşleşme problem alanına değil, başka bir programlama diline yapılır. Bu işlem genelde operasyonel semantik olarak adlandırılır. Operasyonel semantiğin de AÖM semantiğinin tek kaynağı olmaması oldukça önem taşımaktadır. Tersi durumda modelleme dili ile programlama dili arasında birebir eşleme yapılmış olur. Bu durumda soyutlama aynı seviyede kalır ve kod üretiminden elde edilen fayda çok düşük olur. Burada klasik bir örnek olarak bir sınıf diyagramındaki sınıfın kod içindeki bir sınıfa eşlenmesi verilebilir. AÖM'de soyutlama seviyesi ve üretkenlik artışı hedeflendiğinden dolayı problem alanının semantiğinin çözüm alanının semantiğinden daha fazla hesaba katılması gerekmektedir.

Gösterim: Sadece soyut sentaks ve semantik, bir dil tanımı için yeterli değildir: modeller bazı görsel şekiller üzerinden erişilebilir olmalıdır. Dolayısıyla soyut sentaksa ek olarak bir de somut sentaksa ihtiyaç vardır. Her modelleme dili belirli bir notasyonlu bazı temsili biçimler ile ifade edilir. Çoğu modelleme dilinin gösteriminde grafiksel ve metinsel biçimler birlikte kullanılır. Modelleme dilleri matris, tablo, formlar veya sadece metinsel gibi diğer gösterimlerde de olabilir.

Bir AÖM dili için notasyon seçimi problem alanın gerçek gösterimi yakın bir şekilde takip edilerek yapılır. İdeal durumda, modelleme dilinin her kavramı bir sembol gibi tam olarak bir gösterimsel temsile sahiptir. Bu prensip gösterimsel yapıların birden fazla kavramı temsil için kullanılmasını en aza indirir ve bütün kavramların dil içinde temsilini garanti eder. Dolayısıyla, temsillerin tamlığı [18], [19] ya da temsilsel doğruluk [20], yani, her kavram için sadece bir gösterimsel yapının mevcut olması, modelleme kavramları ile gösterimler arasındaki yorumlamalar ile başa çıkmak için iyi bilinen bir kriterdir.

2.2.1.2 Hesaplama Modeli

Bir modelleme dili genelde durum makinesi, veri akışı veya veri yapısı gibi bir hesaplama modeline dayanır. Bu modelin seçimi ya da birkaç modelin kombinasyonu modelleme hedefine bağlıdır. Bazı sistemler dinamik olduğu için sonlu durum makinesi gibi modeller uygulanır, bazıları da statik yapılarına odaklanma ihtiyacı olduğundan özellik diyagramları veya bileşen diyagramları ile daha iyi belirtilir. Bu nedenlerden dolayı çok çeşitli modelleme dilleri mevcuttur. Modellenecek yazılım sistemini nasıl gördüklerine bağlı olarak diller arasında büyük farklılıklar görülür. Sistem örneğin eş-zamanlı, dağıtık, asenkron iletişim kullanan, paralel karakteristikte, çift-taraflı bağlantılar kullanan ya da belirsizce davranan bir sistem mi? Bu türden karakteristikler modelleme dili tarafından kullanılan hesaplama modelini tanımlar. Bunlardan bazısı soyut sentaks ile belirtilir (ağaç, yönlü graf, paralel akış gibi), bazısı da somut bir sentaks içinde görselleştirilir (iki yönlü bir bağlantıyı temsil eden çizginin her iki ucunda bulunan ok işaretleri). Bu farklı gösterimler sayesinde modelleme dillerinin çok farklı türleri bulunmaktadır. Örneğin, sonlu durum makinelerinin farklı versiyonları, Petri ağı diyagramlarının birçok sınıfı ve veri akış diyagramlarının çeşitli tipleri vardır.

Modelleme dilleri statik yapıları modelleyenler ve dinamik davranışı belirtenler olarak iki kısma ayrılabilir. Gerçekte, bu ayırma işlemi çok açık ve net bir şekilde yapılamamakta ve iki tarafı da belirten diller bulunmaktadır. Modelleme dillerinde kullanılan farklı hesaplama modelleri üzerine teferruatlı bir inceleme [21] de yapılmıştır.

Statik Yapıları Modelleme: Bazı modelleme dilleri sadece veya daha ziyade bir uygulamanın statik yapılarına hitap eder. En yaygın olan alana özgü modelleme dilleri veri yapılarını belirtmek, verileri normalize etmek ve veritabanı şemaları üretmek üzere veritabanı tasarımında kullanılanlardır. Diğer bir alan da kullanıcı arayüzü için kod üretimi sağlayan bir alana özgü modelleme dili ile yapılan tanımlamalardır.

Kodlama temelli modelleme dillerine dikkatlice bakılacak olursa, sınıf diyagramlarının da bu kategoriye girdiği görülebilir. Sınıflar arasında bir etkileşim olsa da genelde bunlar modelde ifade edilmezler. Böyle olmakla birlikte bazı sınıf diyagramları sınıflar arası metot çağrılarını da içerir ve davranışsal tarafı da bir ölçüde modellemiş olur.

Diğer örnek diller küme diyagramları, özellik diyagramları, sistem diyagramları, ağ diyagramları, bileşen diyagramları, süreç matrisleri, yaygınlaştırma diyagramları, Venn şemaları, kalıtım modelleri ve varlık-ilişki diyagramları üretenlerdir.

AÖM açısından, statik diller genelde davranışsal olanlara nazaran tanımlanması daha az zahmetlidir. Kod üretimi için kullanımları da sistemin statik özelliklerini üretmeleri fakat bunların nasıl işlenmesi gerektiği ile ilgilenmemeleri nedeniyle daha kolaydır.

Dinamik Yapı ve Davranışları Modelleme: Modelleme dillerinin ikinci ana sınıfı bir sistemin ya da sistemin alt biriminin davranışını ve dinamik ilgilerini belirtenlerdir. Sonlu durum makineleri, etkileşim diyagramları ve Petri ağları bu kategorideki en yaygın modelleme dilleridir. Bu modelleme dilleri sistemin bir olay veya durum tabanlı ele alındığı yerlerde oldukça yaygın kullanıma sahiptir.

Süreç modellemede, iş akışı modellemede, veri akışında ve işaret işlemede kullanılan süreç ve akış diyagramları da bu kategoride ele alınabilir. Model tabanlı geliştirme için kullanılan araçlara bakıldığında bunların davranış ve fonksiyonaliteyi tanımladıkları görülmektedir. Bunların dilleri genelde alana özgüdür ve kendi sınırları içinde kullanışlı olmaktadırlar. Bu tür araçların genişletilmesini ve değiştirilmesini ise aracı üreten firma yapmaktadır.

AÖM'de fonksiyonalite, iş kuralları ve diğer uygulama mantığı ile alakalı kodu üretmek için davranışsal ilgilerin tanımlanmasına ihtiyaç vardır. Alana özgü dillerde, bu tür davranış bilinen bir hesaplama modelinin bir kısmını genişletmek suretiyle elde edilir. Bu genişletmeler mantıksal operasyonlar, koşullar, karar noktaları ve diğer fonksiyonel ilgiler gibi dile özel iş süreci kavramlarını ve kuralları ekleme işini yapabilir.

Model tabanlı geliştirme literatüründe çoğunlukla yazılım sistemlerinin yapısal ilgilerini üretmek için modelleme kullanılır. AÖM'de ise modelleme dilleri ile ortaya çıkarılan modeller kullanılarak kod üreticiler sadece konfigürasyon ya da statik kodu değil, fonksiyonel kodu da üretirler.

Dilleri Genişletme: Bazı diller hem statik yapıları hem de dinamik davranışı modellediğinden iki kategoride de değerlendirilebilir. Genelde bir tarafı daha güçlü olmasına karşın diğer tarafa da hitap eder. Örneğin, iş süreci modellemede, davranışı modelleyen bir iş akışı dili aynı zamanda süreçler arasında kullanılan veri yapılarını

belirten modelleme yapılarına sahip olabilir. Yine bir veri modeli de verinin sistem tarafından nasıl kullanılacağını belirten modelleme yapıları içerebilir. Tek bir dil bu şekilde ancak belirli bir limite kadar genişletilebilir. Bu durum birden fazla dilin kullanımını gerektirir. Birçok dilin kullanılması gerekliliği sadece farklı hesaplama modelleri ihtiyacından değil, aynı zamanda alanın ve alan kurallarını büyüklüğünden, modeli yapan ve kullanan farklı kişilerin olmasından ya da tasarımların taşeronlar gibi çeşitli paydaşlar ile paylaşılmasından da kaynaklanır.

Dilin seçimi işi sadece belirli bir modelleme dilini ya da hesaplama modelini seçmeyi değil aynı zamanda modelleme yapılarının bir alt kümesinin seçimini ve bunların ilgili alan için genişletilmesini de içerir. Alt kümeye ilişkin olarak, bir durum diyagramının kullanıldığı her alanda sistem kesintiye uğratılmadan önceki durumu hatırlamak için bir geçmiş durum uygulamak gerekli değildir. Benzer şekilde UML 2 tabanlı bir sekans diyagramında birleştirilmiş parçalar kullanmak ve nesneler arasında dahili bağlantılar belirtmek çoğu zaman gerekli değildir. Bu yapıları sağlayan bir dil çoğu modelleyici için fazladan bir yük getirmiş olur. Dil genişletmelerine ilişkin olarak, sentaktik ve semantik yapılar eklemek yazılım sistemlerini yeterli miktarda tanımlamak için genellikle gereklidir. Bu yapılar isimlendirmeleri, özellikleri, kısıtları, diğer dil yapılarına bağlantıları ile birlikte alana özgüdür ve bütün semantik sadece bir alan düşünülerek biçimlendirilir. Taslak ve dokümantasyon için kullanılan modellerde farklı yaklaşımları uygulamakta sorun bulunmazken, kod üretimi hesaplama modelini seçmek için daha detaylı gereksinimler ortaya çıkarır. Diğer bir deyişle modeller kod üretimi içine girdi sağlamak üzere yeterince veriyi içermelidir.

2.2.1.3 Çoklu İlgileri Modelleme için Dilleri Entegre Etme

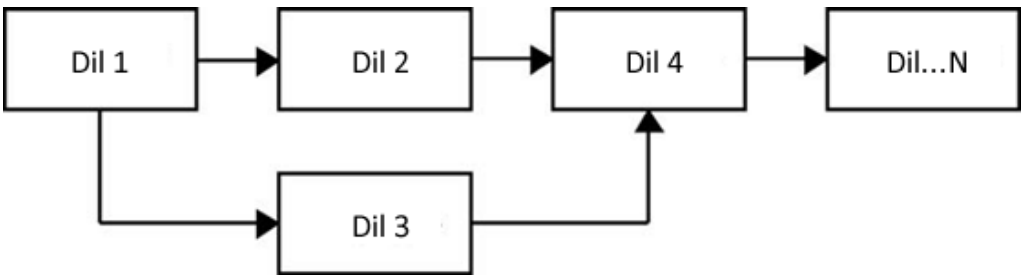
Büyük yazılım sistemleri farklı görünüşler, ilgiler ve detay seviyeleri tanımlamayı gerektirir. Bunun sonucunda çoklu modeller ve diller kullanılır. Kullanıcı navigasyonu, kalıcılık, gerçek zamanlı ilgiler ve veri yapıları gibi çoklu ilgileri kapsayan bir dil oluşturulabilse de, bu tür bir dilin kullanımı ve bakımı zordur. Sentaktik ve semantik dil yapıların sayısı bakımından böyle bir dilin büyüklüğü fazladır. Dahası, tek bir büyük dil ile oluşturulan modeller değişkenliği ve değişikliği iyi bir biçimde desteklemez. Örneğin, bir otomobil bilgilendirme sistemi geliştirilirken, birisi statik ekran tanımlamaları ve

diğeri davranışsal fonksiyonalite için olmak üzere en az iki dile sahip olmak en gerekir. Böylelikle fonksiyonalite tanımlarını değiştirmeye gerek duymaksızın ekranların değişmesi mümkün kılınır. Dilleri ayırmak tasarım görevinden karmaşıklığı gizlemeyi ve verilen görev için modelleme işine yardımcı olmayı mümkün kılar.

Farklı geliştirici rollerinin olması ve modellerin farklı zamanlarda veya geliştirme yaşam döngüsünün farklı safhalarında yapılması da çoklu dil kullanımına yönlendirir. Örneğin aynı alan içinde bir dil prototip oluşturmak için diğer bir dil de üretim kodunu üretmek için kullanılabilir. Bazen de bazı detayların sadece şirket içinde tutulabilmesi için üçüncü partilere özgü farklı bir dile de ihtiyaç duyulur. Bu tür bir dil genelde şirket içinde kullanılan dilin bir alt kümesi şeklindedir.

Modelleme dillerini entegre etmek için, iki farklı metot uygulanabilir. Birincisi, dönüşümün kod üretirken veya modelleme zamanında modelleri entegre etmesidir. Diğer metot ise dillerin dil seviyesinde entegre edilmesidir. Bu da entegre bir üst modele sahip olmak anlamına gelir.

Dönüşümleri kullanan entegrasyon dil tanımlamalarını ayrı tutar. Bir tanımlama modeli bir kere bir dil ile yapıldığında, bu tanım farklı bir dil temelli başka bir modele dönüştürülebilir. Birçok dil arasında da bu dönüşümler uygulanabilmesi her ne kadar mümkün olsa da genelde bu dönüşümler sadece iki dil arasında uygulanır (Şekil 2.3).

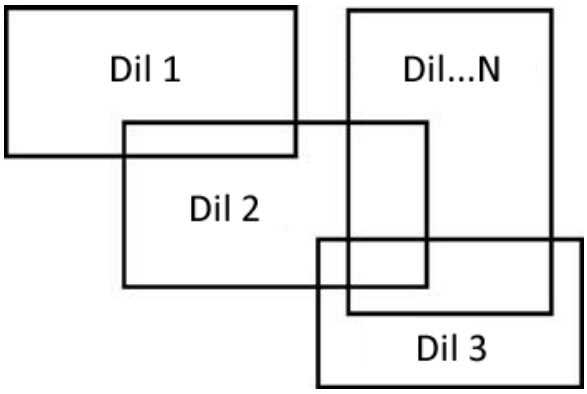


Şekil 2. 3 Dilleri dönüşümler ile entegre etme

Dönüşüm bir dilden diğerine eşlemenin nasıl olacağını bilir. Model doğruluğunun dönüşümden önce kontrol edilebilmesi için dönüşüm genelde bir dil ile yapılmış kaynak modellerin tam olmasını bekler. Aksi durumda dönüşüm tam olarak gerçekleşmeyebilir veya hatalı sonuçlara neden olabilir. Farklı dillerdeki farklı modellerdeki değişikliklerin bakımını yapabilmek zor olduğundan çoğu durumda dönüşümler sadece bir kere

yapılır. Değişikliklerin çoğaltılması zor olduğundan, dönüşüm sürecinde bir şelale modeli uygulanır. Model dönüşümleri büyük takımlarda paralel geliştirmeye çok uygun değildir ve çoklu dillerdeki tasarımların yeniden kullanımını desteklemezler.

Modelleme dilleri paylaşılmış ya da bağlı modelleme yapıları ile de entegre olabilir. Burada dil tanımı entegrasyon düşünülerek yapılır. Örneğin, bir veri kavramı bir taraftan bir iş akışı modelinde taşınan elemanları tanımlarken, diğer taraftan veritabanı şeması için veri yapılarını belirtmek için kullanılabilir. Daha sonra şema seviyesinde tanımlama değiştirilirse, bu değişim iş akışı modelinde de ek dönüşümler olmaksızın kullanılabilir (Şekil 2.4).



Şekil 2. 4 Dilleri ortak elemanlar ile entegre etme

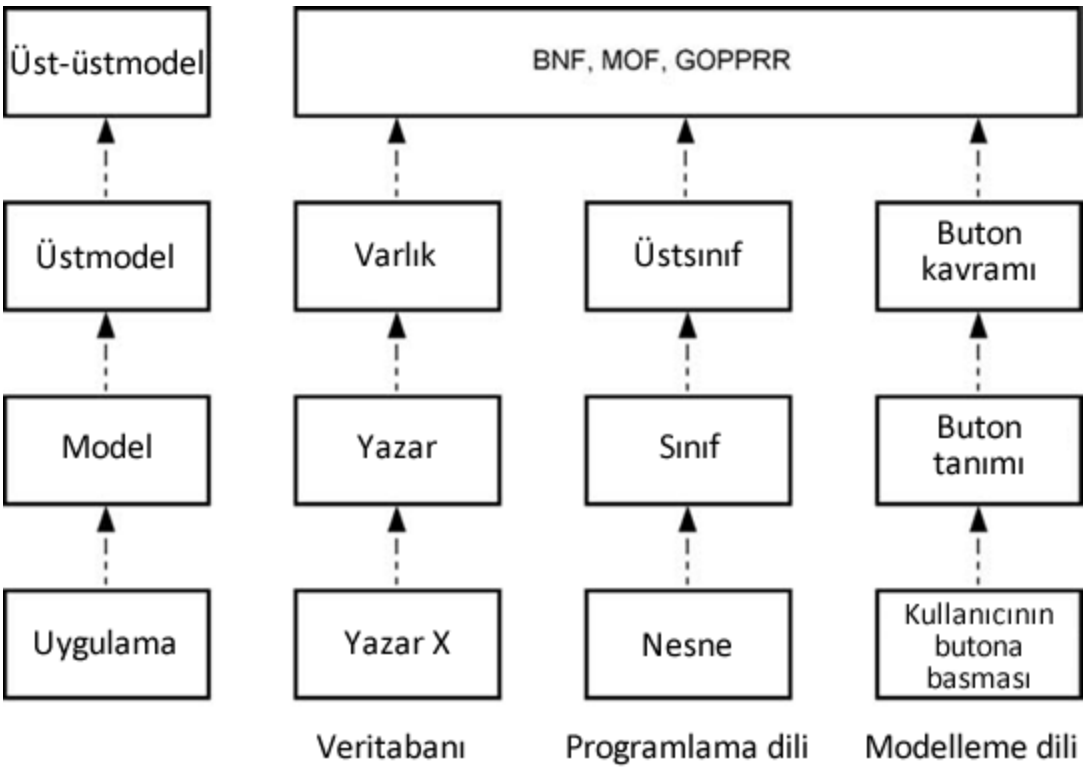
Özetle, dilleri belirtim düzeyinde entegre etmek ayrı dönüşümler kullanmaktan daha iyidir. Entegre dillerde kavramın ideal olarak sadece bir kopyası olacağı yerde, dönüşümler hem aynı tasarım elemanı hem de bir vekil eleman şeklinde birden fazla yere aynı türden bilginin kopyalanmasını gerektirir. Dil entegrasyonu tasarımların yeniden kullanılabilirliğini ve model yeniden yapımını iyi bir şekilde sağlar. Aynı zamanda eğer diller aynı üst model tabanlı olursa, alan kuralları modelleme aşamasında kontrol edilebilir. Olası hatalar model oluşturma sırasında bulunursa kolayca ve düşük bir maliyetle düzeltilir. Entegre diller aynı zamanda eş-zamanlı programlamayı da daha iyi destekler ve bir dil ile yapılan değişikliklerin diğer diller ile çok daha kolayca paylaşımını sağlar.

2.2.1.4 Dil Tanımlama: Üst model

AÖM'de modelleme dili formal olarak tanımlanmalı ve bir araç ile desteklenmelidir. Aksi durumda, model oluşturmak ve bu modellerden kod üretmek mümkün olmayacaktır. Dil belirtimi genelde üst model olarak adlandırılır. Üst kelimesinin kullanılmasının nedeni dil belirtiminin sıradan modellerden bir seviye daha yukarıda olmasındandır. En basit şekliyle, bir üst model bir modelleme dilinin kavramsal modelidir denilebilir. Üst model bir dilin kavramlarını, bu kavramların özelliklerini, dil elemanları arasındaki geçerli bağlantıları, model hiyerarşi yapılarını ve model doğruluk kurallarını belirtir. Bazı durumlarda yeniden kullanım ve farklı model entegrasyon yaklaşımları desteği de gerekli olmaktadır.

Dil tasarımı ve tanımı bir üst modelleme görevi de içerir. Bu görev alan kavramlarını nesneler, nesnelerin özellikleri, ilişkiler şeklinde tanımlı bağlantıları ve nesnelerin bu ilişkilerde oynadıkları rol gibi çeşitli dil elemanları ile eşlemek şeklindedir. Tekrar belirtmek gerekirse üst modelleme yazılım geliştirmedeki sıradan modellemeden bir soyutlama seviyesi ve mantıksal olarak daha yukarıdadır. Üst modelleme hakkında daha teferruatlı bir tanım [22] de yapılmıştır.

Örnekleme Tabanlı Dört Seviye: Üst model fikri bilgisayar biliminde uzun zamandan beri bilinmektedir. Kotteman ve Konsynski sistemlerin kullanım ve evrimlerini modellemek için en az dört seviyeli bir örnekleme gerektiğini göstermiştir [23]. Benzer bir gözlem Uluslararası Standartlar Organizayonu'nun (Information Resources Dictionary Standard) çerçeve mimarisinin temelini oluşturmaktadır [24]. Daha sonra OMG'nin dört seviye modelleme çerçevesine de temel teşkil etmiştir [24], [26]. Seviyeler ve hiyerarşileri Şekil 2.5'te görülmektedir. Örnekleme hiyerarşisinde bir kavram bir kere örneklenebiliyorsa, bu bir modeldir, ortaya çıkan şey de örneklenebiliyorsa bu durumda üst modeldir.



Şekil 2. 5 Örneklemenin dört seviyesi

Bitişik model katmanları bir veritabanı analojisi ile daha iyi anlaşılabilir: üst seviye şemadır ve alt seviye veritabanı durumudur. Bu nedenle alt seviye üst seviye olmaksızın anlaşılamaz. Veritabanı açısından dört seviyeyi incelemek gerekirse; Yazar X bir uygulamanın veritabanında bir değer olarak görülebilir. Model seviyesinde, veritabanı şemasının bir parçası olarak "Yazar"ın bir tanımı bulunmaktadır. Yukarıya doğru çıkıldığında şema kavramı bir "Varlık" olarak üst model seviyesinde tanımlanır. Son olarak üst-üst model veritabanı tasarımı için kullanılan kavramları belirtmek için kullanılır.

Aynı katmanlar bazı programlama dillerinde de benzer şekilde mevcuttur. Uygulama seviyesinde nesneler, model seviyesinde sınıflar, üst model seviyesinde Smalltalk'ta olduğu gibi üst sınıflar bulunmaktadır. Nesne tabanlı olmayan programlarda sırasıyla programın çalıştırılması, program kodu ve programlama dili şeklinde tanımlanabilir. Üst model seviyesinde programlama dillerinin tanımlanmasında yaygın kullanılan BNF (Backus-Naur Form)'dur.

Modelleme dili açısından bakıldığında, aynı katmanlar benzer şekilde tanımlanabilir. Uygulama seviyesinde sistem kullanılır. Model seviyesinde örneğin bir buton tanımlanır. Üst model seviyesinde buton kavramını içeren modelleme dilinin tanımı bulunmaktadır. Son olarak en üst seviyede bir modelleme dilini tanımlamak için kullanılan üst modelleme dili bulunmaktadır. Genelde bu tür örnek eşleme yapılmaksızın modelleri, şemaları ve kodu anlayabilmek zordur.

Üst modelleme için, GOPRR [27] ya da MOF [28] gibi farklı türden diller kullanılabilir. Eğer kod üretimini desteklemek için dil biçimlendirilip kullanılmayacaksa üst modellemenin nasıl yapıldığının çok fazla önemi bulunmamaktadır. AÖM felsefesi gereğince, bir üst modelleme dili dilleri tanımlamak için yapılmalı, dil yapımına rehberlik etmeli, gereksiz detayları gizlemeli ve tanımlanan dili kullanan araçları üretmekte destek sağlamalıdır. Son olarak, üst modelin nasıl tanımlanacağı kullanılan araca bağlıdır. Yaygın bir yaklaşım daha sonra genel amaçlı bir modelleme aracını konfigüre edecek bir formata çevrilecek olan modelleme dilini tanımlamak için bir dil kullanmaktır. Daha ileri bir yaklaşım modelleme ve üst modellemeyi aynı araçta desteklemektir ki bu dil yapımını ve evrimini destekleyen bir yaklaşımdır.

2.2.2 Modeller

AÖM'de modeller geliştiricilerin bir sistemin tanımlamalarını oluşturduğu, değiştirdiği ve sildiği temel, birincil kaynaktır. Dilde, kod üreticide ve alan çatısında değişiklikler mümkün olsa da çoğu geliştirici genellikle modellemeye odaklanır. Alana özgü modeller daha sonra doğrudan kod üretmek için kullanılır. AÖM'de modellerden ilerde düzenleme yapmak üzere başka bir modele çevrilmesi durumlarından kaçınılması gerekmektedir. Bu yaklaşım geçmişte kodlar ile başarılı olmamıştır ve modeller ile başarılı olma olasılığı da düşüktür.

Her modelin temelinde dolaylı veya doğrudan tanımlı bazı diller bulunur. Genel olarak, AÖM'de ifade edilen modeller diğer modeller ile benzer karakteristiklere sahiptir. Aralarında bazı önemli farklar da vardır. AÖM'de modelleyici alanın kavramlarını kullanarak çalışır. Bu kavramlar dil tanımlayıcı tarafından verilir. AÖM'de modeller aynı zamanda formaldir, kodlamadan daha soyut seviyededir, alan kurallarını bağlı kalır ve alanda çalışan geliştiricilere aşina olan kavramlara dayanır.

Genellikle diyagram gibi tek bir model, bir sistemin, programın veya özelliğin seçilen bir görüntüsünü temsil eder. Bütün sistemi belirtmek için sistemin farklı ilgilerini ifade eden birçok modele ve modelleme kavramına ihtiyaç duyulur. Modellemenin uygulama geliştirmede AÖM'deki şekliyle kullanımına karşı olanlar modeli sistemin basitleştirilmiş bir açıklaması, programlama dili ile yapılan kodu ise gerçeği ifade etmek için daha iyi görür. Bu çalışan sistemi belirtmek için yeterli bilgiyi taşımayan modelleme dilleri için doğru olsa da, AÖM'deki modeller için geçerli değildir.

AÖM'de modeller formaldir ve kod üretici, alan çerçevesi ve hedef platform ile desteklenir. Bunlar modelleri öne çıkarmak için gerekli işi yaparlar ve böylece modeller bir modelleyici bakış açısından bütün sistemi geliştirmek için yeterli belirtimler olur. Elle yapılan kodlama için bu yeni bir şey değildir: programcılar genelde elle kodlamayı mümkün kılan derleyicileri, bağlayıcıları ve kütüphaneleri akıllarına getirmez. Bu analojiyi izleyerek, derleyici geliştirici için C program kodunun sadece bir model olduğu görülebilir.

2.2.2.1 Modeller ile Çalışma

Bir modelleme dilinin kullanımı doğal olarak modeller ile çalışmayı gerektirir. Gerçek model tabanlı geliştirmede, çok fazla model ortaya çıkar. Elle kodlama, kodu UML ile görselleştirme veya kodlama temelli diğer modelleme dilleri ile karşılaştırıldığında AÖM modellerinde daha az tanımlama işi vardır. Bunun nedeni basitçe AÖM'deki soyutlamanın daima daha yüksek bir seviyede olmasındandır. Fakat böyle olsa da yine de genelde alışık olunandan daha fazla modele sahip olunur.

AÖM'de modellerin geliştirme sürecindeki rolleri değişir:

- **Kodlar değil, modeller versiyonlanır.** AÖM'de modeller geliştirme ilerledikçe yok sayılamayacak, yabana atılamayacak birincil öncelikli, en değerli kaynaklardır. Bu nedenle modeller versiyonlanır. C derleme esnasında oluşan nesne dosyalarının versiyonlanmadığı gibi, kod herhangi bir zamanda üretilebileceğinden, kaynak kodu da ayrıca versiyonlamaya ihtiyaç yoktur. Bununla birlikte diğer geliştirme görevleri ihtiyaç duyabileceğinden kod

kaydedilebilir. Örneğin inşa süreci kod üreticiyi çalıştıramadığı durumda bu geçerlidir.

- **Modelleme ve üreticiler daha evvel test etme işine ait olan bazı görevleri icra eder.** Tasarım aşamasında teste ile ilişkili daha fazla görev yapılır. Çünkü düzgün bir modelleme dili ilgili alanı bilir ve ideal olarak geçersiz ya da düşük performansa neden olacak tasarımların yapılmasına müsaade etmez. Tasarım esnasında hataları erkenden tespit edip önlemenin maliyeti düşük olacağından testi modelleme aşamasında desteklemek önemlidir.
- **Hata ayıklama işi modelleme seviyesinde yapılır.** Belirtilen fonksiyonalite beklendiği şekilde çalışmıyorsa üretilen kodun düşük seviyesinde hata ayıklamaktansa, çalışmanın takibi modelde yapılır.
- **Modeller iletişim amacı ile de kullanılabilir.** Modeller alan kavramlarında ifade edilir ve iletişim için daha fazla kullanılabilir. Modeller güvenilebilir olduğundan, gerçeklemeden ayrı değildir. Dilde doğrudan alan kavramlarına sahip olmak modelleri programlama kavramları kullanarak alanı modellemeye nazaran daha kolay okunabilir, anlaşılabilir, hatırlanabilir ve doğrulanabilir kılar.
- **Modeller birçok farklı çıktı için girdi durumundadırlar.** Modeller sadece kod üretmek için değil, doküman üretmek, test senaryoları ve konfigürasyon verileri gibi daha başka amaçlar için de kullanılabilir.
- **Modeller farklı gösterimsel biçimlerde ifade edilebilir.** Model kullanıcısına, görselleştirme ihtiyaçlarına ve analiz ihtiyaçlarına bağlı olarak, modeller grafiksel diyagramlar, matrisler, tablolar, formlar ve metin gibi çeşitli biçimlerde ifade edilebilir.
- **AÖM ile modelleme çeviktir.** Gerçekleme başlamadan önce yapılan yoğun analiz işleri ile karşılaştırıldığında AÖM'nin çevik olduğu görülebilir. Şöyle ki sadece gerekli görülen ilgiler modellenir. Kod üretimi ile hızlıca sonuç alınır ve modele geri bildirimde bulunulur. İhtiyaç duyulursa modelleme dili geliştirme sürecine kısmen çeviklik katmak üzere de tasarlanabilir. Örneğin, bir modelleme dili ilk olarak bir prototip üretmek için veya ürünün fonksiyonalitesini gözden geçirmek için kısmen kullanılır. Daha sonra, aynı modeller tanımlamayı

tamamlamak ve ürün kodunu üretmek için genişletilebilir. Son ürün kodu prototip için kullanılandan farklı bir programlama dilinde de olabilir.

2.2.2.2 Model Kullanıcıları

Model yapmaya sıklıkla kod üretmek amacıyla odaklanıldığından, yaygın dil kullanıcıları uygulama geliştiricilerdir. AÖM'de modellerin olası kullanıcı tabanı daha da yaygınlaştırılabilir. Daha yüksek bir soyutlama seviyesi ve alana daha yakın bir eşleme müşterilerin ve diğer son kullanıcıların geliştirme sürecinde daha fazla bulunmasını sağlar. Bunlar tanımlamaları kolayca okuyabilir, kabul edebilir ve gerektiğinde bazı durumlarda değiştirebilir. Bu çok önemli bir husustur, zira müşterilerin sürece katılım seviyesi projenin başarısını doğrudan etkiler. AÖM yazılım geliştirici olmayan kişilerin de tanımlar yapmasına izin verir. Alan uzmanları, genelde yazılım geliştirme temeli olmayan kişilerdir fakat kod üretimi için uygulamaları tanımlayabilir.

Modellerin rolleri değiştiğinden, gereksinimlerin ve gerçeklemenin sınırları değişebilir. Örneğin, alan uzmanları kavram prototip ya da kavram gösterimi için modelleri tanımlayabilir, daha sonra da uygulama geliştiriciler bu modeller üzerinden devam edebilir. İş daha sonra diğer dilleri kullanma ya da mevcut modelleri ek gerçekleme detayları ile genişletme üzerine kurulu olabilir. Bir AÖM çözümü aynı zamanda geleneksel uygulama yazılım geliştiricilerinden farklı bir kullanıcı grubu için de oluşturulabilir. AÖM kullanımının bir sınıfı test mühendislerini hedefler. Bunlar test senaryoları, test program betikleri ya da testleri çalıştıran programları üreten modeller oluşturur. Diğer bir grup ürün yaygınlaştırılması, kurulumu ve sunum hizmetini yapan sistem uzmanlarıdır. Bunlar yazılım birimlerinin donanıma yaygınlaştırılmasını tanımlamak ya da kesintisiz servisler için yüksek erişilebilme ayarlarını tanımlamak gibi konfigürasyonun özel karakteristiği ile direk ilişkili kavramları uygulayan modeller ile çalışabilir. Özel bir uygulama alanında uygulama mimarisini tanımlayan mimarlar için özel AÖM çözümleri de yapılabilir.

2.2.3 Kod Üreticiler

AÖM'de kod üreticiler çalıştırılabilir bir program oluşturmak üzere yorumlama veya derleme için modelleri koda dönüştürür. Otomasyon sağlayarak, AÖM yaklaşımının

üretkenlik ve kalite kazanımlarına katkıda bulunurlar. Üretilmiş kod tipik olarak modelleyicinin perspektifinden bakıldığında tamdır. Yani üretilmiş kod çalıştırılabilir durumdadır ve üretim ortamında çalışabilecek kalitededir. Üretimden sonra kod elle müdahaleye ihtiyaç duymaz. Kod üretici ve modelleme dili bir şirket içinde kullanılan dar bir uygulama alanının taleplerini karşılamak üzere yapıldığından bu mümkündür. Burada şunu vurgulamak gerekir ki bu kullanılan tüm kodun üretilmiş olması anlamına gelmez. Zira ek olarak bir alan çerçevesi ve hedef platform da kullanılabilir. Bunlar farklı modellerden üretilebilir veya günümüzde yaygın olarak yapıldığı gibi elle geliştirilebilir. Kod üreticinin kendisi, alan çerçevesi ve hedef platform gibi, kara kutu bileşenler veya derleyiciler ile aynı biçimde geliştiricilere görünmez olabilir.

Kod üreticiler farklı şekillerde sınıflandırılabilir. En fazla kullanılan ise bunları bildirimsel, operasyonel veya bu ikisinin karışımı şeklinde sınıflandırmaktır. Bu sınıflandırma kod üreticileri belirtmek için kullanılan yaklaşıma dayanır. Bildirimsel yaklaşımda kaynağın (üst model) elemanları ile hedef programlama dili arasındaki eşleme tanımlanır. Operasyonel yaklaşımlar, graf dönüşüm kuralları gibi verilen bir kaynak modelden hedef kodu üretmek için gerekli adımları tanımlar.

AÖM uygulamalarında üretilen kodu görmek ve değiştirmek her ne kadar mümkün olsa da geliştiriciler genellikle kod üreticinin çıktılarını inceleme ihtiyacı duymazlar. Üretilen kodu değiştirmek demek bir C programı derlemesinden sonra makine kodunu değiştirmek gibidir ki bu gereksizdir. AÖM'de modifikasyonlar modeller üzerinde yapılır, basitçe bir ara ürün seviyesinde olan üretilmiş kodda yapılmaz. Derleyicilerin bu imkanı sağladıkları gibi kod üreticiler de aynı şekilde sağlayabilir.

2.2.3.1 Kod Üretme Prensibi

Temel olarak bir kod üretici modellere erişen ve onlardan bilgiyi elde eden ve bu bilgiyi özel bir sentakstaki çıktıya dönüştüren bir otomattır. Bu süreç üst modele, kavramları, semantiği ve kuralları ile modelleme diline ve alan çerçevesi ile hedef platformun ihtiyaç duyduğu girdi sentaksına bağlıdır ve bunlar tarafından yönlendirilir.

Modeldeki Veriye Erişme: Kod üreticiler dilin üst modeli tabanlı olan modellere erişir. Bunlar modelde belirli bir kök elemandan başlayarak dolaşmaya başlayabilirler. Bazı

nesne tiplerini aramaya ya da modelin sahip olduğu çeşitli ilişki türlerine ve bağlantı türlerine bağlı biçimde de dolaşıyor olabilirler. Eğer kod üretici modeldeki örnek değerleri kullanırsa daha fazla dolaşma seçenekleri de mevcut olabilir. Model elemanlarını bir sonrakine erişeceği belirli bir değere bağlı şekilde seçmek bunun bir örneğidir. Bir kod üretici bağlantıları ve alt modelleri dolaşmayı, "depth-" ya da "breadth-first" gibi çeşitli dolaşma yöntemleri ile ilerletebilir veya dolaşma ve erişim için belirli bir sırayı gözetebilir. Genellikle erişmek ve dolaşmak için model verisinin çoğu tasarım bilgisi ile aynıdır. Ek bir model verisi de kullanılabilir. Bu veriler şunlar olabilir:

- Model elemanlarının konumu, büyüklüğü veya diğer elemanlara göre konumu
- Model yönetim verileri, oluşturma zamanı, sürüm ya da oluşturan kişi gibi
- Kod üreticiye yardımcı olacak fakat problem alanında bir çözüm bulmak için gerekli olmayan model bilgileri. Hedef ortam, derleyici ve çıktı dizinini seçmek bunlardandır.

Eğer faklı dillere dayanan birden fazla model mevcutsa, dolaşma tek bir modele erişmekte olduğu gibi aynı prensiplere dayanabilir. Eğer diller entegre üst modelleri kullanıyorsa, kod üretici ayrı modelleri entegreymiş gibi farzeder. Eğer her model diğerinden ayrı kabul edilirse, kod üretici modelleri üretim safhasında entegre eder. Örneğin, model içinde modeller arasındaki bağlantıları gösteren katar eşleşmeleri ve notları kullanarak bunu yapar.

Model Verilerini Çıkartma: Modellerde dolaşırken, kod üretici tasarım verilerini çıkarır ve kullanılan bir alan çerçevesi ile birleştirir. Kod üretici modellerden sadece üst modelde sağlanan bilgiyi elde edebilir. En basit durumda, her modelleme elemanı her defasında modelleyici tarafından girilen değerleri içeren aynı kodu üretir. Kod üreticiler aynı zamanda veriyi model elemanları arasındaki diğer bağlantılar gibi model elemanlarının kombinasyonlarını analiz ederek çıkarır.

Modelleri Koda Dönüştürme: Modellerde dolaşırken erişilen veri kod üretme amacıyla birleştirilir. Burada kod üretici çatı kodu ile entegrasyon veya hedef ortam ve kütüphanelerine çağrı yapmakla birlikte çıktı için ek bir bilgi ekler. Tipik olarak bir kod üretici üretilen kod için bir sentaks üretir. Kod üretici aynı zamanda model verisini

dönüştürür, bazen üst model verisini de çıktı koda dönüştürür. Üretilen kodun yapısı gerçeklemenin ihtiyaçlarına bağlıdır.

Bir kod üretici modellere girilen bilgiyi gerçekleme dilinde uygulanabilir bir formata dönüştürmek için aynı zamanda bir çevirme mekanizması da kullanır. Örnek bir durum üretilen kodda değişken isimleri olarak kullanılacak olan modellerde girilen değerlerden boşlukları silmek olabilir. Aynı zamanda, eğer üretilen programlama dili değişken, sınıf, işlem isimlendirme için isimleri büyük harfle başlatmak gibi bazı özel gelenekler kullanıyor ise, kod üretici bunlar için çevirmeleri kullanabilir.

2.2.3.2 Üretilen Kodun Kalitesi

Ne türden kodların hangi kalite düzeyinde üretilebileceği hakkında çeşitli görüşler vardır. Örnek olarak, arayüzler ve veritabanı şemaları gibi yaygın tasarımlardan statik belirtimsel tanımlamaların üretilmesinin otomasyonu uzun yıllardır yapılmaktadır. Bununla birlikte davranışsal, fonksiyonel kodun üretiminde durum farklıdır. AÖM'de kod üretimi hem statik kod hem de davranışsal, fonksiyonel kodu üretmektedir.

Üretilen Kodun Güvenilirliği: Geliştiricilerin birçoğu üçüncü parti kod üreticilerini kullanmayı yeğlemez. Zira bunların sahibi olan tedarikçi şirketler kod üretim yöntemini sabitleştirmiştir. Kodu yazmak için birden fazla yöntem bulunurken üçüncü parti kod üreticiler bunlardan yalnızca birini seçer. Bu seçilen yöntem çoğu zaman ihtiyacı tam olarak karşılamaz. Bunlar organizasyon için ideal olan kodu üretmek için organizasyonun özel gereksinimlerinin farkında olmaz. Üretilmiş kod üzerinde değişiklik yapmak gerçekçi bir yaklaşım olmadığından, üretilmiş kod genelde kullanım dışı olur. Bu şekilde üretilmiş kodun değeri prototip yapma ve gereksinim toplama şeklinde sınırlı olur.

Geliştiricilerin kötü deneyimleri, üretilmiş koda olan güvenin azalmasına neden olur. Bu güven eksikliği, geliştiricilerin kod üreticileri kendileri yazması halinde radikal bir biçimde değişir. Sadece kod üretim sürecinde değil, tasarımdan çıktıya kadar bütün kontrolün geliştiricide olması üretilmiş koda olan güveni artırır. AÖM'de şirket içindeki deneyimli bir geliştirici, takımdaki diğer geliştiriciler için otomasyon sürecini ve çıktıyı tanımlar.

Üretilen Kodun Verimliliği: Kod üreticilere karşı olanların en temel argümanı üretilmiş kodun boyut gereksinimleri ve çalışma zamanı verimliğini karşılayamaz olacağı iddiasıdır. Üretilmiş kod için de elle yazılmış kod için de nihayetinde derleyiciler daha ileri bir optimizasyon uygulamaktadır. Geleneksel kod üreticilerin ürettiği kodların verimliliği düşük olma olasılığı olsa da AÖM için durum farklıdır. AÖM'de kontrol geliştiricinin elindedir ve istenildiğinde kod üreticiyi değiştirebilir ve gerekli iyileştirmeleri yapabilir.

Deneyimli bir geliştirici tarafından hazırlanmış kod üretici, ortalama bir programcının elle yazdığı koddan daima daha iyi kod üretir. Bellek yönetimi, optimizasyon, programlama modeli ve kod stilleri tutarlı bir şekilde uygulanır. İdeal durumda, üretilmiş kod, kod üreticiyi tanımlayan deneyimli programcının elle yazdığı kod gibi görünür. Üretilmiş kod aynı zamanda şirket standartları gibi diğer gereksinimleri de karşılar.

2.2.3.3 Üreticilerin Diğer Kullanımları

Kod üreticiler program kodlarını üretmenin yanında diğer başka amaçlar için de kullanılabilir. Üretim süreci yaygın olarak kod inşa betiklerini oluşturmak, derleyiciyi çalıştırmak, üretilen kodu yaygınlaştırmak gibi işlemleri de içerecek şekilde genişletilir. Diğer üretici türleri aşağıda belirtilmiştir.

Model Kontrolü: AÖM'de üreticiler aynı zamanda tasarımların tutarlılığı ve tamlığını kontrol amacıyla da kullanılır. Bütün kuralları üst modelde tanımlamak ve her modelleme esnasında kontrol etmek imkan dahilinde olmadığı için bu gereklidir. Özellikle birden fazla model olduğunda kısmi modelleri kontrol ederken veya farklı geliştiriciler tarafından yapılmış modelleri entegre ederken bu durum geçerlidir. Modelleme işine yardımcı olması ve yapılması gerekenleri bildirmesi amacıyla model analizi için de üreticiler kullanılabilir. Modelin tam olup olmadığına bakılıp tma olması için ne yapılması gerektiğini bildirmek buna bir örnektir. Bu türden bir model kontrolü üreticilere de benzer şekilde uygulanabilir.

Metrikler: Modele tabanlı geliştirme yapıldığında kod tabanlı metrikler aynı şekilde uygulanabilir. Bunlar elle yazılan kod yerine üretilen koda bakılarak hesaplanır. Hedef

ortam kodu hali hazırda mevcut olduğu durumda metrikler aynı zamanda ortam fonksiyonları ve kütüphanelerine uygulanabilir.

AÖM'de kod tabanlı metrikler gerekli insan işi miktarını ölçmez. Program büyüklüğünü analiz eden ve gerekli geliştirme eforunu tahmin eden fonksiyona nokta analizi (FPA) [29] gibi metrikler geçerli değildir. Zira bu metrikler hesaplandığında uygulama zaten hazır olmaktadır. AÖM özel alana dayanan ve üst model verisini kullanan metrikleri kullanır. Sistem karmaşıklığı (cyclomatic complexity) [30] ya da program uzunluğu [31] gibi üst modelden bağımsız metrikler kullanılmaz.

Prototip ve Simülasyon: Üreticiler aynı zamanda son uygulama kodu yerine prototip kodu üretmek için de kullanılabilir. Prototipler genellikle erken geri bildirim almak için kullanılır ve üreticiler tamamiyle farklı bir hedef ortam için ve üretim kodundan farklı bir programlama dilinde kod üretebilir. Burada üreticiler kodu opitimize etmesi gerekli değildir, fonksiyonaliteyi, kullanılırlık gibi diğer karakteristikleri ortaya çıkarması yeterlidir. Bununla birlikte modelleme dili hem prototip üretmek hem de son üretim kodunu üretmek için aynı olabilir. Model aynı zamanda simülasyon için de kullanılabilir. Kod üreticiler simülatör tarafından ihtiyaç duyulan sentakstaki çıktıyı sağlar.

Konfigürasyon, Paketleme ve Yaygınlaştırma: Üreticiler mevcut bileşenleri bir araya getirmek üzere entegrasyon kodları üretmekte limitlidir. Daha iyi bir otomasyon arayışında, daha tipik bir senaryo konfigürasyon verilerini ve uygulama yaygınlaştırma ve kurulum için paketleme bilgisi üretmektir. Diğer bir ifade ile uygulama ve kurulum konfigürasyonu aynı modellerden eş zamanlı üretilebilir. Bu tekrar edilen bu süreci daha güvenli ve kolay yapmak için gerekli iş ihtiyacını düşürür.

Dokümantasyon: Gerçek kodun üretildiği kaynak modellerden üreticiler aynı zamanda dokümantasyon, teftiş raporları, yönetim durum raporları ve bunlar gibi belgeleri üretmek için de kullanılabilir. Bunun açık bir yararı şudur ki dokümantasyonu geliştirme ile birlikte güncel tutmak için herhangi bir elle güncelleme ihtiyacı yoktur. Şunu da belirtmek gerekir ki üretilmiş dokümantasyon sadece gerçekleme ile ilgili değil bunun yanında alan terimleri ile tanımlanmış çözümü de ortaya koyar. Hepsinden sonra, AÖM'de modeller çözümü değil daha ziyade problem alanını tanımlar.

Test: Modeller test etme için de kullanılabilir, özellikle AÖM gibi formal modeller; bunlar diğer modellerin tutarsız bilgi ya da ürünün hatalı tanımlanması problemlerine nadiren sahiptir. Dildeki alan kuralları normal birçok testi basit bir şekilde gereksiz kılar, çünkü modellerde zaten test edilmiştirler. Benzer şekilde otomatik kod üretimi elle yapılan kodlama karşılaşılan dizgi hatası, eksik değerler, geçersiz referanslar gibi birçok tipik hatayı ortadan kaldırır. Kod üretiminde kullanılan modeller genellikle testleri üretmek için zayıf kaynaktırlar. Modellerden elde edilen veriler bununla birlikte geliştirilen sistem için uygun olan testleri seçmek için kullanılabilir. Bu özellikle binlerce testin bulunduğu büyük sistemler için geçerlidir. AÖM özel olarak test etme için de kullanılabilir; test mühendisleri test etmeyi ve test senaryoları ve testleri çalıştıracak uygulamaları üretmek için ürün kavramlarını kullanan testleri tanımlayabilir.

Dil Kullanımı ve İyileştirme: Kod üreticiler dilin ve kod üreticinin nasıl kullanıldığı bilgisini de üretebilir. Dilin kullanım kalıplarını ortaya çıkarmaya yardımcı olur, hangi kavramlar kullanılmıyor, hangi modeller dilin eski sürümünü kullanıyor gibi. Modelleme dilleri aynı zamanda modelleyiciler için mevcut dil ile yakalanamayacak şeyleri ifade etmek için imkanlar sağlayan açık semantikler ile birlikte bazı elemanlar içerebilir. Bir üretici dilin kullanıcılarının nerede eksikler bulduklarını ve geliştirilebilecek bölgeleri anlamak için bu kavramların olası kullanımlarını raporlayabilir.

Birden fazla üreticiler ile tek bir kaynağa sahip olunup birden fazla hedefe sahip olunabilir; geliştiriciler sadece bir yerde değişikliğe ihtiyaç duyarlar ve üreticiler geriye kalan işi yapar. Üreticiler aynı zamanda tek başına çalışmak yerine diğerleri ile ilişki içinde olacak şekilde birleştirilebilir. Örneğin, bir model kontrol raporu kod üretiminden evvel otomatik olarak çalıştırılabilir. Eğer hata bulunursa, kod üretim işlemi de durdurulur. Benzer şekilde üretilmiş dokümantasyon bir metrik raporu tarafından üretilmiş bilgiyi içerebilir. Başka tür bir üretici, pratikte nadir görülse de, diğer üreticileri üretmek için kullanılan bir üretici de olabilir.

2.3 AÖM Organizasyon ve Süreci

AÖM'de geliştirme organizasyonu, AÖM ile uygulamaları geliştirenler ve AÖM çözümünü geliştirenler olmak üzere iki farklı role ayrılır. Bu ayrım hâlihazırda geliştirme işi yapan birçok şirkette aynı şekilde uygulanmaktadır. Örnek olarak bileşen tabanlı

geliştirmede bazı elemanlar bileşenleri geliştirir ya da tüm organizasyonel birimler bileşen fabrikaları şeklinde hareket eder, diğer elemanlar da bu bileşenleri kullanarak uygulamaları geliştirir. Benzer şekilde, ürün hattı mühendisliğinde, bazı kişiler tüm projeler için ortak platformu geliştirirken, bazı kişiler de bu ortak platformu kullanarak ürünler geliştirir. Yazılım geliştirme diğer geliştirme organizasyonu ve endüstrilerine benzer şekilde organizasyonda her alanda bilgili kişilere sahip olma fikrinden belirli bir alanda uzmanlaşan kişilere sahip olma fikrine kaymaktadır.

2.3.1 Organizasyon ve Roller

AÖM'de iki rolün bulunması her rolde farklı kişilerin olacağı anlamına gelmemektedir. Genellikle AÖM çözümünü geliştirenler en azından iyileştirme ve genişletme amacıyla da olsa aynı zamanda bu çözümü kullanırlar. Burada kritik olan nokta daha deneyimli geliştiricilerin AÖM çözümünü oluşturuyor olmasıdır. Bunlar daha az deneyimli geliştiricilere nazaran otomasyonu daha iyi tanımlarlar ve diğerleri üzerinde gerekli otoriteye de sahiptirler.

AÖM'de aşağıdaki roller tanımlanabilir:

- **Alan uzmanları** AÖM'nin uygulanacağı problem alanı hakkında bilgi sahibi olan kişilerdir. Bunlar alanın terminolojisini, kavramlarını ve kurallarını bilirler. Uygulama geliştiriciler de eğer geçmişte birçok benzer uygulamar geliştirmiş ise alan uzmanı olabilirler. Az sayıda uygulama geliştirmek daha yüksek soyutlama bulmak üzere genelleştirme yapabilmek için yeterince alan uzmanlığı kazandırmaz. Örnek olarak sigorta uygulaması geliştirirken alan uzmanları sigorta uzmanları ve ürün yöneticileridir.
- **AÖM kullanıcıları** modelleme dillerini kullanan kişilerdir. Bunlar genelde organizasyondaki en geniş gruptur. AÖM kullanıcılarının en yaygın alt grubu uygulamaları üretmek için model oluşturanlar olmakla birlikte diğer alt gruplar da mevcuttur. Modeller test mühendisleri, ürün yöneticileri, yaygınlaştırma mühendisleri, satışlar ve müşteriler ile iletişimi desteklemek için kullanılabilirler. İletişim desteğine ek olarak, test mühendisleri test senaryolarını planlamak ve test setleri üretmek için modelleri kullanabilir. Yaygınlaştırma mühendisleri AÖM modellerini kullanarak kurulum programları ve ürün konfigürasyonları

üretebilir. Bu kullanıcılar birbirinden farklı üreticiler kullansalar da kullandıkları modeller aynıdır. Yöneticiler de geliştirme durumu ile ilgili raporları ve ölçütleri modeller üzerinden alabilirler.

- **Dil geliştiriciler** modelleme dilini tanımlar. Bir üst modeli biçimlendirirler ve kullanımı ile ilgili yardım dokümanları ve örnek modeller hazırlarlar. Dil geliştiriciler alan uzmanları ve önemli AÖM kullanıcıları ile birlikte çalışırlar. AÖM çözümünü oluşturmak için üst model tabanlı araçlar kullanıldığında genelde organizasyonda dil tanımlama işini bir ya da iki kişi yapar. Bu tür araçlar gerekli modelleme araçlarını otomatik olarak sağlar. Dolayısıyla araç geliştirme için geleneksel programlama gerekmez. Bu tür araçlar aynı zamanda alan uzmanlarının modelleme dillerini kullanarak AÖM oluşturmaya kolayca katılımlarını sağlar.
- **Ergonomist** dil geliştiricilere dilin kullanılabilirliğini artırmakta yardımcı olur. Bu özel rol bazı durumlarda çok önemli olabilmektedir. Örneğin kullanıcı arayüzü veya insan makine arayüzü uygulamaları için bir AÖM çözümü oluşturulduğunda, modellerin gerçek ürüne iyi bir biçimde karşılık gelmesi önem taşır. Arayüz stillerini tanımlayan kişi daha sonra dil geliştiricilere de yardımcı olabilir. Eğer dil programcı olmayan veya farklı kültürdeki kişiler tarafından kullanıyor ise ergonomistin rolü oldukça önemli olacaktır.
- **Üretici geliştiriciler** modellerden koda dönüşümleri tanımlar. Bu işlemi yaparken mimarlar ve çerçeve kodunun geliştiricileri tarafından verilen formatları ve referans gerçeklemeleri takip ederler. Üretici geliştiriciler genelde alan çerçevesini tanımlayanlar ile aynı kişilerdir.
- **Alan çerçevesi geliştiricileri** genelde deneyimli geliştiriciler ve uygulama mimarlarıdır. Bunlar referans gerçeklemeler sağlar ve hedef ortamın nasıl kullanılacağını belirlerler. Bunlar genelde hâlihazırda bileşen çerçeveleri ve kütüphaneleri gibi yeniden kullanılabilir varlıklar yapan kişilerdir.
- **Araç geliştiriciler** modelleme dillerini ve kod üreticileri gerçeklerler. Kullanılan araçlara bağlı olarak bu grubun çok büyük olması gerekmeyebilir. Zira modern üst modelleme araçları dil ve üretici tanımlamalarından otomatik olarak modelleme editörleri ve üreticiler sağlayabilmektedirler. Otomasyon ve bu

türden araçlar kullanılmadığında araç üretimi için organizasyonda fazla eleman bulundurmak gerekebilir. Bu nedenle AÖM çözümünü oluşturanlar kendi işleri için de bir otomasyon mekanizması kullanmalıdırlar.

Yukarıdaki liste hedef ortam ve yeniden kullanılabilir bileşenlerin geliştiricilerini ya da müşterileri ve yöneticileri ayırmaz. Bunlar AÖM yaklaşımı kullanımından bağımsız olarak zaten mevcuttur. Benzer şekilde uygulama mühendisleri ve mimarlar da zaten mevcuttur fakat bunların işleri kısmen değişir. AÖM'de uygulama mühendislerinin çoğu uygulamaları oluşturmak ve bakım yapmak için yüksek seviye modeller kullanır. Mimarlar da bir AÖM çözümünde takip edilecek kuralları tanımlarlar.

2.3.2 AÖM Tanımlama Süreci

AÖM süreci dört temel fazdan oluşur: ilk geliştirme, yaygınlaştırma, kullanım ve bakım.

2.3.2.1 Konsept Kanıtı

Alana özgü yaklaşım şirkette ilk uygulanacağı zaman, AÖM'nin yaklaşım olarak uygunluğunun bir ön gösterimi gerekli olabilir. Böyle bir ön gösterimde, konsept kanıtı hızlıca yapılabilsin diye genellikle dar ve iyi bilinen bir alan seçilir. Burada seçilen AÖM çözümü büyük olmak zorunda değildir. Alan içinde somut bir fikir verebilecek büyüklükte olması yeterlidir. Kavram ön gösterimi aday alanları belirlemeye ve AÖM çözümünü gerçekleyen gerçek proje için gereksinimleri ortaya çıkarmaya da yardımcı olur.

2.3.2.2 Pilot Proje

AÖM yaklaşımının fizibilitesi doğrulandıktan sonra bir pilot proje yapılır. Pilot projede AÖM çözümünün ilk versiyonu tanımlanır, gerçeklenir ve test edilir. Bu aşamada AÖM kullanıcıları işin içine daha fazla girer ve bazıları gerçek bir projenin temsili bir örneğinde AÖM çözümünü kullanır. Pilot AÖM'nin etkilerini tahmin etmeye yardımcı olur ve organizasyonu yardımcı dokümanlar, eğitim materyalleri ve örnek tasarımlar yapmak suretiyle AÖM'ye giriş için hazırlar.

2.3.2.3 AÖM Çözümünün Kullanımı

AÖM'nin yaygınlaştırılması üretim ortamına tam olarak geçildiğinde gerçekleşir. Bu andan itibaren şirket daha yüsek seviyede bir soyutlamanın ve daha geniş çapta bir otomasyon sürecinin faydalarını toplamaya başlar. Aynı zamanda modellerin ve üretilen kodun kullanımı arttıkça AÖM'nin rolü de artar.

2.3.2.4 AÖM Bakımı

Geliştirilen bir AÖM çözümü genelde ilk geliştirildiği şekilde sabit kalmaz. Gereksinimler değiştiğinde ve organizasyon AÖM çözümünü kullanmak için daha iyi yöntemler bulduğunda AÖM çözümünde güncelleme ihtiyacı doğar. Kullanım olarak yaygın olan büyük boyutlu bir kod üretmek ya da kod üreticileri ölçütler, simülasyon gibi diğer türden çıktılar üretmek için kullanmaktır. AÖM çözümünün geliştiricileri ilgili dilleri, üreticileri ve alan çerçevelerini günceller ve modelleyiciler ile paylaşır. Bir müddet sonra AÖM çözümü yaşam döngüsünün sonuna ulaşır ve artık çok fazla değişiklik yapılmaz. Çünkü genelde uygulamalar aktif olarak sürekli geliştirilmez, sadece bulunan hatalar için düzeltmeler yapılır. Çoğunlukla yeni bir alan ortaya çıkar ya da uygulamanın hedef ortamı değişir.

BÖLÜM 3

AÖM ÇÖZÜMÜ GELİŞTİRME

3.1 Problem Tanımı

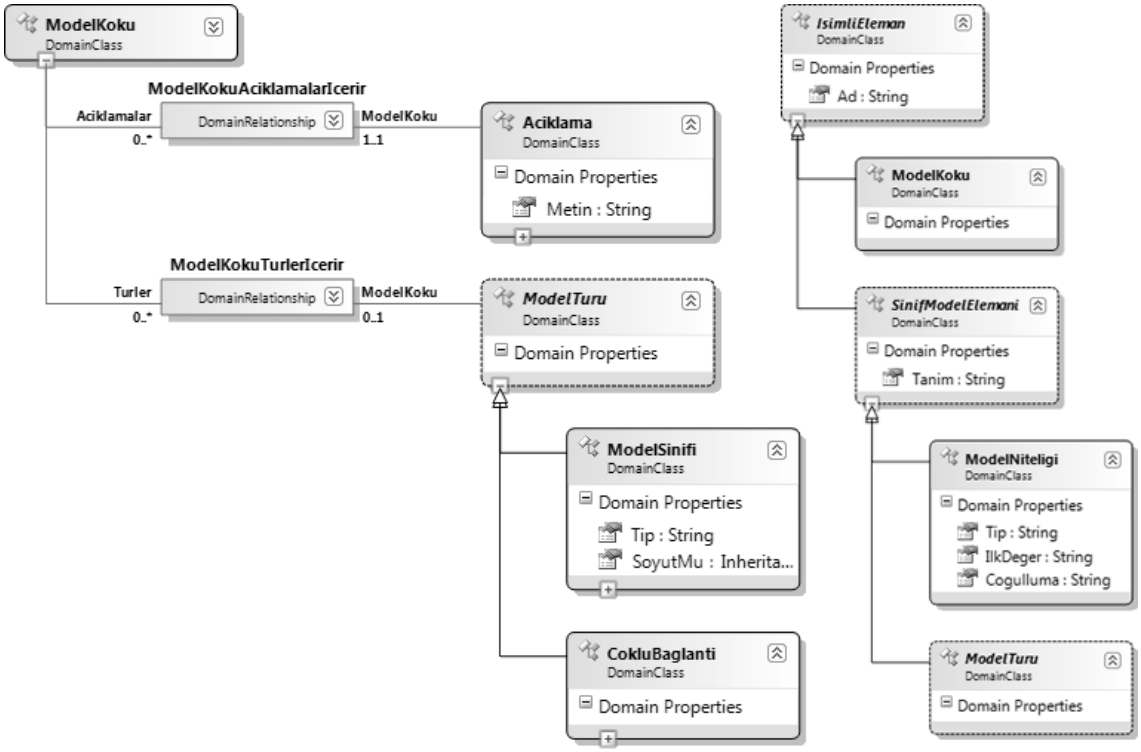
Web tabanlı veri yönetim uygulamaları bir sistemde saklanan bilgilerin izlenmesi, eklenmesi, güncellenmesi ve silinmesi gibi bilgi yönetim işlemlerini sistemi kullanan veya yönetenlere bir kullanıcı arabirimi sağlamak amacıyla kullanılmaktadırlar. Sektörde yaygın olarak kullanılan bu tür uygulamalar genelde katmanlı mimari ile geliştirilmekte ve üretiminde otomatik olmayan veya yarı otomatik süreçler kullanılmaktadır.

Yazılım geliştiricilerin bu tür uygulamaları geliştirirken izlediği klasik temel yöntem önce veritabanının oluşturulması, daha sonra veri erişim katmanın yazılması ve son olarak da bu katmanı kullanarak uygulamanın görsel ara yüzünün hazırlanması şeklindedir. Diğer bir yöntem bir ORM aracı ile veri katmanından kısmen bağımsız şekilde kodun bir kısmını üretmek suretiyle uygulamayı geliştirmektir. Genel olarak izlenen süreçlerde genel amaçlı programlama dilleri kullanılır ve elle kodlama yapılır. Sisteme yeni bir bilgi eklendiğinde yeniden bu süreçler tekrar ettirilir. Mevcut bir bilginin niteliklerinde veya geçerlilik kurallarında bir değişiklik olduğunda bu değişiklikler ve geliştirmeler elle yapılır ve sistemin yapılan değişiklikle ilişkili olan kısımları yeniden test edilir.

Bu geliştirme süreçlerinin maliyetli ve hataya açık olmaları nedeniyle en azından uzun vadede geliştirme ve bakım maliyetlerini düşürmek adına böyle bir sistemin alana özgü modelleme ile geliştirilmesi düşünülmüştür. Bu amaçla çalışmada böyle bir sistemin alana özgü modelleme yaklaşımı ile geliştirilmesi gerçeklenmiştir.

3.2 Üst Model Tanımı

Web tabanlı veri yönetim sistemi üst modelini tanımlarken temel olarak bir veri veya varlık modelinden yola çıkılarak alan modeli oluşturulmuştur [35]. Üst model seviyesinde gerçek dünya nesneleri için alan sınıfları, bu nesneler arasındaki bağlar için alan ilişkileri, nesnelerin özellikleri için alan özellikleri ve alan kuralları tanımlanmıştır. Üst modeldeki varlıklar arasındaki temel ilişkiler Şekil 3.1'de görülmektedir.



Şekil 3. 1 Temel model ilişkileri

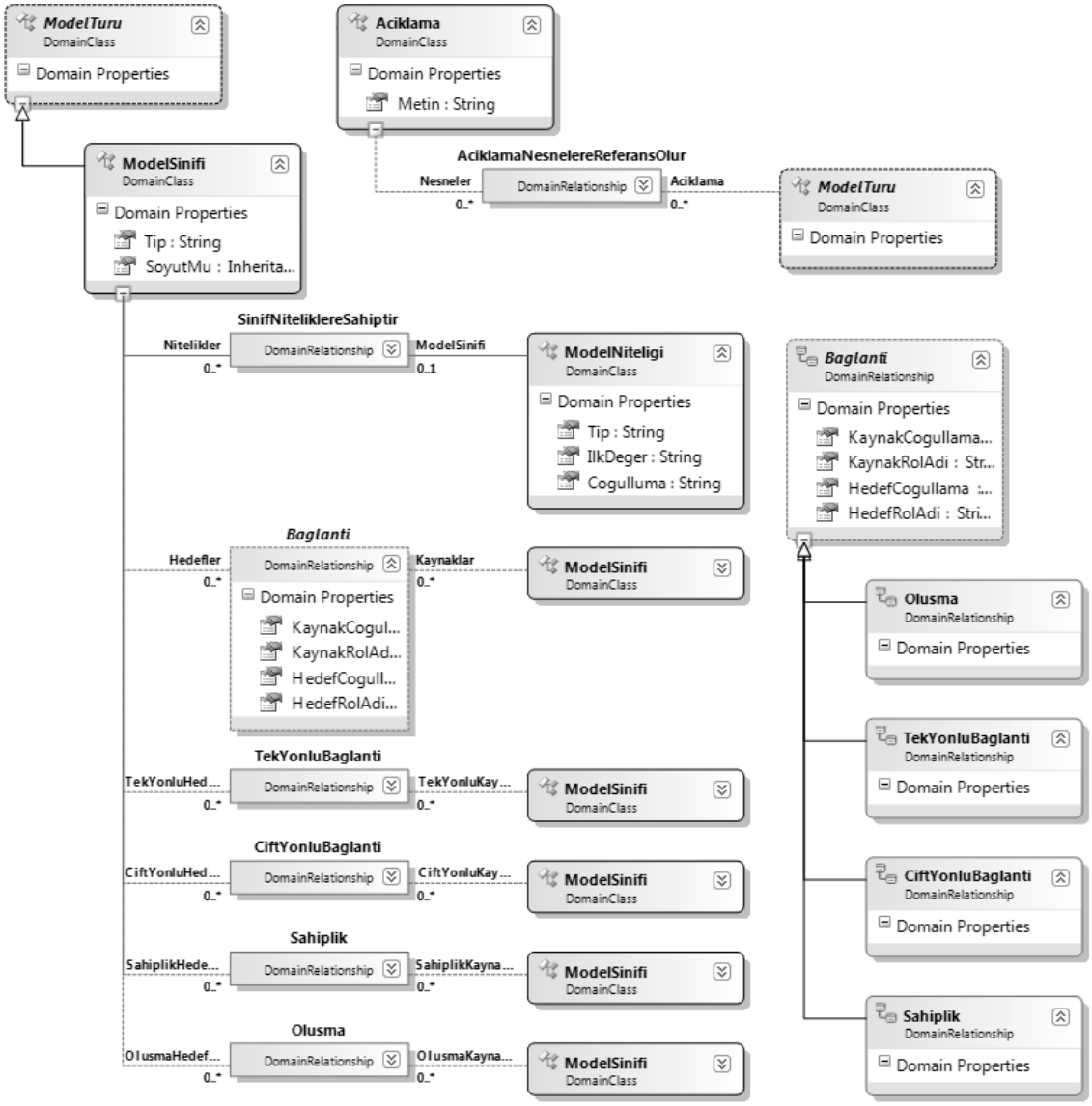
Modelde ilk olarak diğer bütün model elemanlarının bağlı olduğu bir model kökü tanımlanmıştır. Model kökü iki alan sınıfı ile ilişkilendirilmiştir. Bunlardan birincisi modeli veya modeldeki elemanları açıklamak için *Aciklama* isimli alan sınıfıdır. Bu alan sınıfı sadece *Metin* özelliği içermektedir. Model kökü ile açıklama sınıfları arasındaki alan ilişkisi *ModelKokuAciklamaIcerir* olarak adlandırılmıştır. Model kullanıcısı modele açıklama ekleyebilir veya eklemeyebilir. Bu nedenle aradaki ilişkinin model kökü tarafındaki rol çoğullaması sıfır veya daha fazla, açıklama tarafında ise sadece birdir. Model kökü ile ilişkisi olan diğer eleman soyut bir sınıf olan *ModelTuru* isimli sınıftır. Bu

sınıf alan modelinde varlıkları temsil eden *ModelSinifi*'na temel teşkil eder. Model kökü ile model türü arasında *ModelKokuTurlerIcerir* ilişkisi bulunmaktadır.

Model elemanları arasındaki hiyerarşi Şekil 3.1'deki gibidir. Modeldeki her bir eleman *IsimliEleman* temel sınıfından türemektedir. Bunlardan ilki model köküdür ve daha önce bahsedildiği gibi modelin ilk elemanıdır. Diğer bir isimli eleman ise soyut bir sınıf olan *SinifModelElemani*'dır. Bu sınıf model sınıfındaki model niteliklerinin ve model türünün türediği temel sınıftır. *ModelNiteligi* sınıfı varlıkların niteliklerini temsil etmektedir. Model niteliklerinin tip, ilk değer ve çoğullama şeklinde alan özellikleri mevcuttur.

Model sınıfları arasındaki ilişkiler de Şekil 3.2'deki gibi tanımlanmıştır. Bunlar nesne modellemede de kullanılan temel sınıf ilişkileridir. İki model sınıfı arasındaki ilişki türlerinin türediği temel alan ilişkisi *Baglanti* ismiyle tanımlanmıştır. Bu ilişkiden *Olusma, TekYonluBaglanti, CiftYonluBaglanti* ve *Sahiplik* ilişkileri türetilmiştir.

Şekile 3.2'de üst model tanımı ilişkiler ve ilişki türleri ile birlikte gösterilmektedir. Burada açıklama ve model sınıfının ilişkileri belirtilmiştir. *ModelTuru* cinsinden elemanlar ile *Aciklama* arasındaki ilişki *AciklamaNesnelereReferansOlur* ilişkisidir. *ModelTuru* temel sıfından türetilen *ModelSinifi* ile *ModelNiteligi* arasında da *SinifNiteliklereSahiptir* ilişkisi bulunmaktadır. *ModelSinifi* ile bir diğer *ModelSinifi* arasında da Şekil 3.2'de tanımları belirtilen bağlantı ilişkileri bulunmaktadır. İlk olarak diğer bağlantıların temel sınıf ilişkilerini belirten *Baglanti* ilişkisi sağlanmıştır. Bu temel ilişkinin sağlanmasıyla ilişkilerdeki temel özellikler bu alan ilişkisi ile belirtilmiş olmaktadır. Böylece ortak özellikleri içeren tek bir bağlantı sağlanmış olur. Diğer bağlantı türleri de ayrıca tanımlanmıştır.

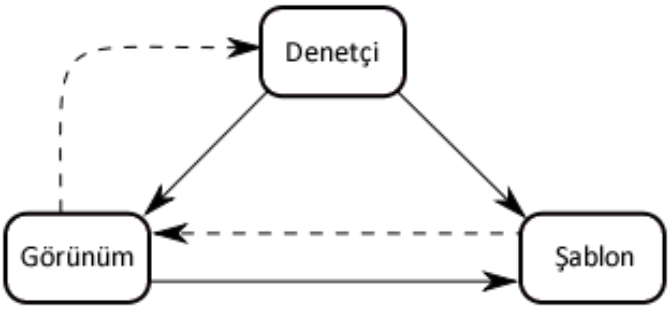


Şekil 3. 2 Model eleman ilişkileri ve ilişki türleri

3.3 Model Dönüşümü ve Kod Üretimi

Çalışmada modelin dönüştürüldüğü kodlar şablon-görünüm-denetçi (model-view-controller) kalıbını [36] gerçekleyen bir web uygulama çerçevesi üzerinde çalışmaktadır. Şablon-görünüm-denetçi kalıbında Şekil 3.3'teki gibi üç bileşen bulunmaktadır. Şablon bileşeni uygulamanın temel bilgilerini içerir. Uygulamanın kullandığı veri, geçerlilik kuralları, veri erişim ve ilişki mantığı şablon tarafında bulunmaktadır. Görünüm bileşeni uygulamanın gösterim kısmıdır ve kullanıcının uygulama ile etkileşimini sağlar. Denetçi bileşeni de kontrol-akış mantığını içerir ve

uygulamanın bilgi akışını ve çalışmasını sağlamak üzere şablon ve görünüm ile etkileşimde bulunur.



Şekil 3. 3 Şablon-görünüm-denetçi kalıbı

Model sınıfı ile oluşturulan her bir varlık için kod tarafında şablon-görünüm-denetçi kalıbına uygun olarak bir şablon, bir denetçi ve verinin listelenmesi, eklenmesi, güncellenmesi, silinmesi ve izlenmesi için görünüm kodları üretilir.

Şablon üretilirken varlığın nitelikleri, bu niteliklerin tipleri ve niteliklerin geçerlilik kuralları model sınıfı için üretilen kod sınıfına eklenir. Bunlara ek olarak varlıklar arasındaki ilişki türlerine göre kod sınıfı içinde ilişkili olduğu diğer sınıf nesneleri de bulunur.

Modeldeki her sınıf için yine kod tarafında bir denetçi kod sınıfı üretilir. Denetçi kod sınıfı içinde listeleme, ekleme, silme, güncelleme, detay bilgileri getirme gibi veritabanı işlemleri bulunmaktadır.

Görünüm için modeldeki her varlığın kod tarafında karşılığı olan şablonu kullanarak kullanıcı arayüzü sağlayan html kodları üretilir. Bunlar listeleme, ekleme, güncelleme, silme ve detay sayfaları olarak ayrı ayrı üretilmektedir.

3.3.1 Şablon Üretimi

Şablon tarafında her bir model sınıfı için bir kod sınıfı üretilir. Kod sınıfı model sınıfının adıyla isimlendirilir. Model nitelikleri kod sınıfına nitelik adları ve tanımları ile birlikte

kod sınıfının birer üyesi olarak eklenir. Model niteliklerinin kod tarafındaki adı niteliğin *Ad* özelliği, görünüm tarafındaki adı *Tanim* özelliği kullanılarak oluşturulur (Şekil 3.4).

```
public class <#= Sinif.Ad #>
{
        public int ID { get; set; }
<#+
foreach (ModelNiteligi nitelik in Sinif.Nitelikler)
{
#>
        [DisplayName("<#= nitelik.Tanim #>")]
        public string <#= nitelik.Ad #> { get; set; }
<#+
}
#>
}
```

Şekil 3. 4 Model sınıfından kod sınıfına dönüşüm

Model sınıfları arasında çift yönlü bağlantı olduğu durumda bağlantının hedef sınıfı kod sınıfına Şekil 3.5'teki gibi eklenir.

```
<#+
if (Sinif.CiftYonluHedefler.Count > 0)
{
   foreach (ModelSinifi hedefSinif in Sinif.CiftYonluHedefler)
   {
#>
   [DisplayName("<#= hedefSinif.Tanim #>")]
   public int <#= hedefSinif.Ad #>ID { get; set; }
   [DisplayName("<#= hedefSinif.Tanim #>")]
   public virtual <#= hedefSinif.Ad #> <#= hedefSinif.Ad #> { get; set; }
<#+
   }
}
#>
```

Şekil 3. 5 Çift yönlü bağlantı hedef sınıfının kod sınıfına eklenmesi

Model sınıfları arasında çift yönlü bağlantı olduğu durumda bağlantının kaynak sınıfı da kod sınıfına Şekil 3,6'daki gibi eklenir.

```
<#+
if (Sinif.CiftYonluKaynaklar.Count > 0)
{
   foreach (ModelSinifi kaynakSinif in Sinif.CiftYonluKaynaklar)
   {
#>
[DisplayName("<#= kaynakSinif.Tanim #>")]
public List<<#= kaynakSinif.Ad #>> <#= kaynakSinif.Ad #>Liste { get; set;
}
<#+
   }
}

#>
```

Şekil 3. 6 Çift yönlü bağlantı kaynak sınıfının kod sınıfına eklenmesi

İki model sınıfı arasında tek yönlü bağlantı olduğu durumda bağlantının hedef sınıfı kod sınıfına Şekil 3.7'deki gibi eklenir.

```
<#+
if (Sinif.TekYonluHedefler.Count > 0)
{
  foreach (ModelSinifi hedefSinif in Sinif.TekYonluHedefler.Distinct())
  {
    foreach (Baglanti baglanti in Baglanti.GetLinks(Sinif,hedefSinif))
    {
      if(baglanti.GetType().Name == "TekYonluBaglanti")
      {
#>
[DisplayName("<#= hedefSinif.Tanim #>")]
public int <#= baglanti.HedefRolAdi #>ID { get; set; }
[DisplayName("<#= hedefSinif.Tanim #>")]
public virtual <#= hedefSinif.Ad #> <#= baglanti.HedefRolAdi #> { get;
set; }
<#+
      }
    }
  }
}

#>
```

Şekil 3. 7 Tek yönlü bağlantı hedef sınıfının kod sınıfına eklenmesi

Model sınıfı ile başka bir model sınıfı arasında oluşma ilişkisinden kaynaklanan bir bağlantı olduğu durumda bağlantının kaynak sınıfı kod sınıfına Şekil 3.8'deki gibi eklenir.

```
<#+
if (Sinif.OlusmaKaynaklari.Count > 0)
{
  foreach (ModelSinifi kaynakSinif in Sinif.OlusmaKaynaklari)
  {
#>
[DisplayName("<#= kaynakSinif.Tanim #>")]
public int <#= kaynakSinif.Ad #>ID { get; set; }
[DisplayName("<#= kaynakSinif.Tanim #>")]
public virtual <#= kaynakSinif.Ad #> <#= kaynakSinif.Ad #> { get; set;
}
<#+
  }
}
#>
```

Şekil 3. 8 Oluşma kaynak sınıfının kod sınıfına eklenmesi

İki model sınıfı arasında oluşma ilişkisinden kaynaklanan bir bağlantı olduğu durumda bağlantının hedef sınıfı kod sınıfına Şekil 3.9'daki gibi eklenir.

```
<#+
if (Sinif.OlusmaHedefleri.Count > 0)
{
  foreach (ModelSinifi hedefSinif in Sinif.OlusmaHedefleri)
  {
#>
[DisplayName("<#= hedefSinif.Tanim #>")]
public List<<#= hedefSinif.Ad #>> <#= hedefSinif.Ad #>Liste { get; set;
}
<#+
  }
}
#>
```

Şekil 3. 9 Oluşma hedef sınıfının kod sınıfına eklenmesi

3.3.2 Denetçi Üretimi

ŞGD kalıbının denetçi ayağında her model sınıfı için bir denetçi kod sınıfı Şekil 3.10'daki gibi üretilir. Denetçi, şablon nesnelerini ve veritabanı bağlamını kullanarak veri ekleme, silme, güncelleme ve detay izleme ekranlarının beslenmesi vazifesini yerine getirir.

```
public class <#= Sinif.Ad #>Controller : Controller
{
    ModelDBContext db = new ModelDBContext();
    public ActionResult Index()
    {
        ...
    }

    public ActionResult Details(int id)
    {
        ...
    }

    public ActionResult Create()
    {
        ...
    }

    [HttpPost]
    public ActionResult Create(<#= Sinif.Ad #> p<#= Sinif.Ad #>)
    {
        ...
    }

    public ActionResult Edit(int id)
    {
        ...
    }

    [HttpPost]
    public ActionResult Edit(int id, FormCollection collection)
    {
        ...
    }

    public ActionResult Delete(int id)
    {
        ...
    }

    [HttpPost]
    public ActionResult Delete(int id, FormCollection collection)
    {
        ...
    }
}
```

Şekil 3. 10 Denetçi kod sınıfının oluşturulması

3.3.3 Görünüm Üretimi

Görünüm kodları listeleme, oluşturma, silme, güncelleme ve detay izleme ekranları olarak cshtml kodları şeklinde üretilir. Şekil 3.11'de oluşturma görünüm kodunun üretilmesi görünmektedir. Burada <<Nitelikler>> kısmında model sınıfındaki her bir nitelik için gerekli değer giriş alanı koda eklenir. <<Çift yönlü hedefler>> kısmında model sınıfının çift yönlü bağlantıları için bir seçim alanı eklenir, <<Tek yönlü hedefler>> kısmında tek yönlü bağlantılar için bir seçim alanı eklenir ve <<Oluşma kaynakları>> kısmında da oluşma bağlantıları için bir seçim alanı eklenir.

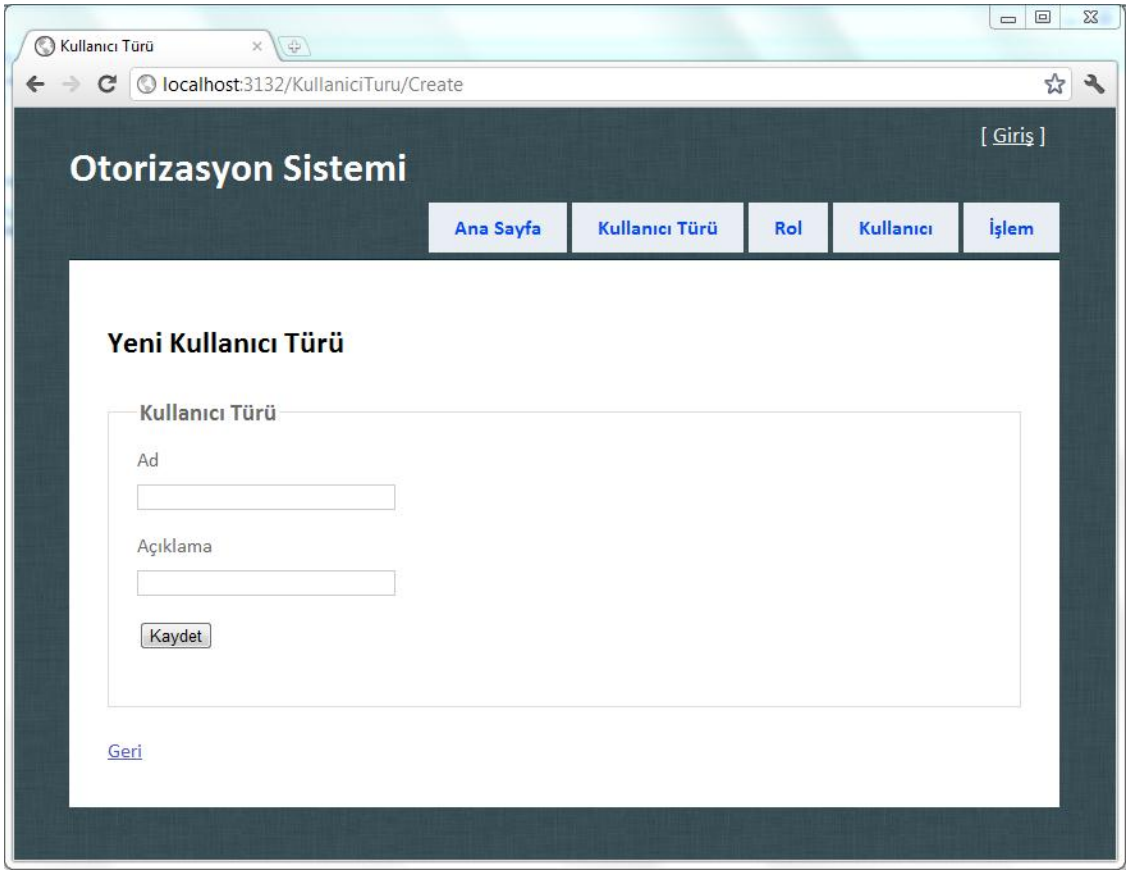
```
@model ModelMVC.Models.<#= Sinif.Ad #>
@{
    ViewBag.Title = "<#= Sinif.Tanim #>";
}

<h2>Yeni <#= Sinif.Tanim #></h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend><#= Sinif.Tanim #></legend>
        <<Nitelikler>>
        <<Çift yönlü hedefler>>
        <<Tek yönlü hedefler>>
        <<Oluşma kaynakları>>
        <p>
            <input type="submit" value="Kaydet" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Geri", "Index")
</div>
```

Şekil 3. 11 Oluşturma görünüm kodunun üretilmesi

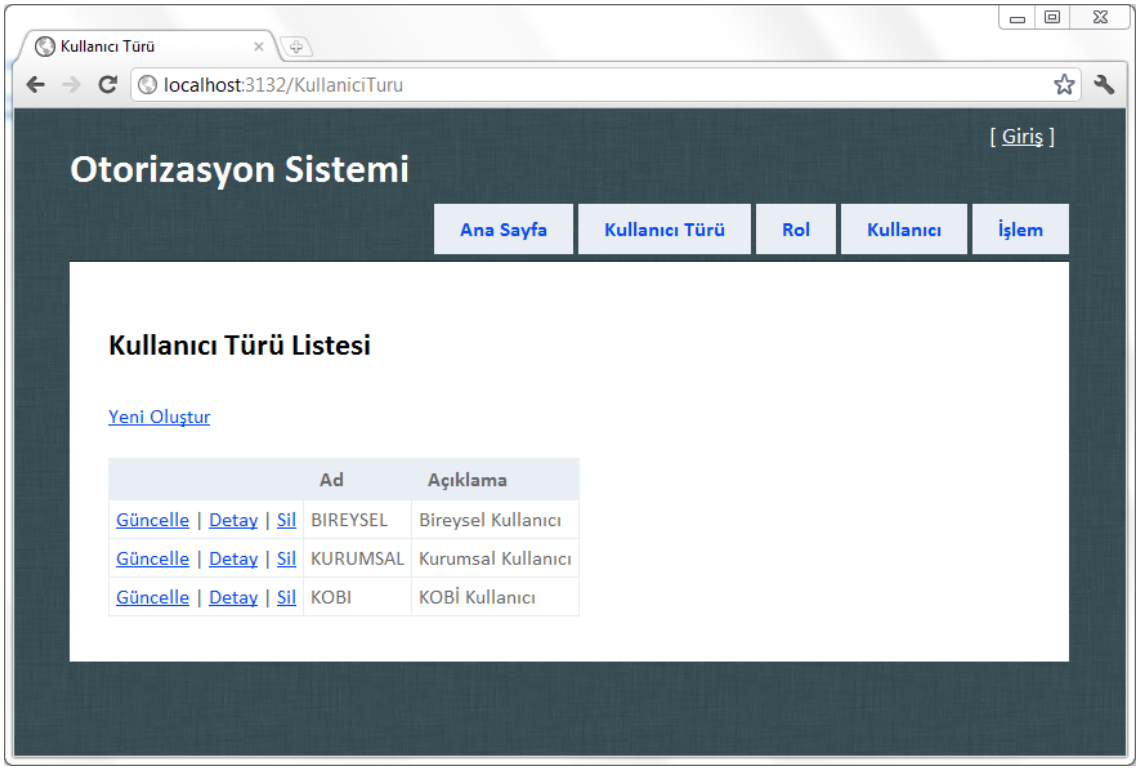
Şekil 3.12'de listeleme görünüm kodunun üretilmesi görülmektedir. Bu ekranda da bir tablo yapısı oluşturularak başlıklara <<Nitelikler>>, <<Çift yönlü hedefler>>, <<Tek yönlü hedefler>> ve <<Oluşma kaynakları>>'nın tanımları, satırlara da şablon modelindeki değerleri eklenir. Örnek bir model sınıfı oluşturma ekranı Şekil 3.13'te, listeleme ekranı da Şekil 3.14'te görülmektedir.

```
@model IEnumerable<ASModel.Models.<#= Sinif.Ad #>>

@{
    ViewBag.Title = "<#= Sinif.Tanim #>";
}

<h2><#= Sinif.Tanim #> Listesi</h2>

<p>
    @Html.ActionLink("Yeni Oluştur", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>
            <<Nitelikler>>
        </th>
        <th>
            <<Çift yönlü hedefler>>
        </th>
        <th>
            <<Tek yönlü hedefler>>
        </th>
        <th>
             <<Oluşma kaynakları>>
        </th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Güncelle", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Detay", "Details", new { id=item.ID }) |
            @Html.ActionLink("Sil", "Delete", new { id=item.ID })
        </td>
        <td>
            <<Nitelikler>>
        </td>
        <td>
            <<Çift yönlü hedefler>>
        </td>
        <td>
            <<Tek yönlü hedefler>>
        </td>
        <td>
             <<Oluşma kaynakları>>
        </td>
    </tr>
}

</table>
```

Şekil 3. 12 Listeleme görünüm kodunun üretilmesi



Şekil 3. 13 Model sınıfı oluşturma ekranı



Şekil 3. 14 Model sınıfı listeleme ekranı

3.4 Teknik Detay

Alan modelinin oluşturulmasında Visual Studio 2010 üzerinde çalışan Visualization and Modeling SDK (VMSDK) [37] kullanılmıştır. Bu platform kullanılarak Visual Studio'ya entegre bir biçimde çalışabilen model tabanlı geliştirme araçları üretilebilmektedir.

Çalışmada ilk olarak model tanımları yapılmış ve modeldeki elemanların model diyagramında nasıl görüneceği belirlenmiştir. Model araç seti ve model ağacı oluşturulmuştur. Daha sonra oluşturulan bu proje derlenerek Visual Studio'ya eklenebilecek bir bileşen oluşturulmuştur. Bu bileşen kurulduğunda Visual Studio içinde herhangi bir proje içinde model oluşturulabilmesi sağlanmaktadır. Model oluşturulduktan sonra bu modeli kullanarak kod üreten T4 metin şablonu (T4 Text Template) dosyaları yazılmıştır. T4 metin şablonları modelden kod ve diğer dokümanları üretebilen program parçaları içermektedir.

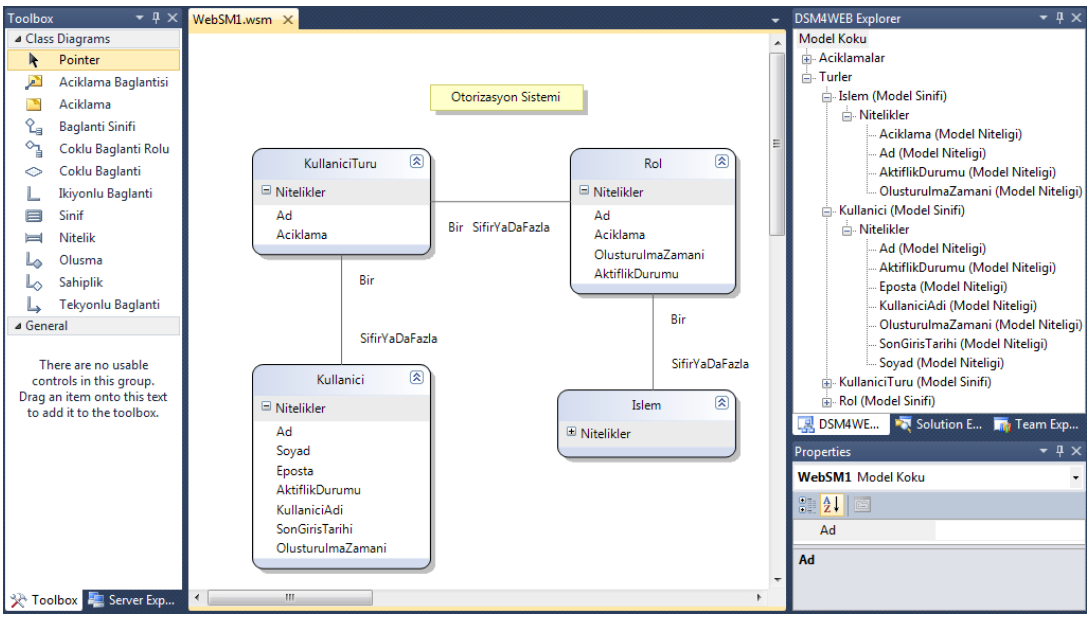
Web tabanlı veri yönetim uygulaması ASP.NET MVC uygulama çerçevesi kullanılarak geliştirilmiştir. Bu uygulama çerçevesinde şablonlar ve denetçiler için C# dilinde yazılmış sınıflar üretilir. Görünüm sağlayan kodlar için de C# program parçaları ve html içeren kodlar üretilir. Uygulamanın veritabanı olarak SQL Server, ORM aracı olarak ADO.NET Entity Framework kullanılmıştır. Entity Framework veri-yoğun uygulamalar geliştirirken ilişkisel bir saklama şeması yerine kavramsal bir uygulama modeli üzerinde kod geliştirmeyi sağlamaktadır.

BÖLÜM 4

DENEYSEL ÇALIŞMA

Alana özgü modelleme yaklaşımının üretkenliğe katkısını anlamak üzere deneysel çalışmada bir otorizasyon sistemi hem klasik geliştirme yöntemi ile hem de çalışmada ortaya konan alana özgü modelleme dili ile geliştirilmiştir.

Otorizasyon sistemleri uygulamaların giriş ve yetkilendirme birimlerinde önemli bir role sahiptir ve sık kullanılan bir modeldir. Otorizasyon sisteminin modeli Şekil 4.1'deki gibidir. Kullanıcı sisteme giriş yapacak ve sistemi kullanacak kişi ya da uygulamalardır. Her kullanıcının bir kullanıcı türü tanımı bulunmaktadır. Bir kullanıcı türüne ait birden fazla kullanıcı bulunabilmektedir. Kullanıcı türünün rolleri bulunmaktadır ve bu rollere bağlı işlemler mevcuttur.



Şekil 4. 1 Otorizasyon sistemi modeli

Deneyde önce klasik geliştirme yöntemi ile uygulama elle geliştirilmiştir ve geliştirme süresi ve üretilen kod satırı sayısı ölçülmüştür. Daha sonra modelleme dili ile sistem modellenmiş ve yazılan kod üretici ile modelden uygulama kodu üretilmiştir. Burada da modelleme ve gerçekleme süreleri, üretilen kod satırı sayıları ve hata sayıları deneyde geliştirilen sistem için ölçülmüştür.

Çizelge 4. 1 Alana özgü modelleme ve klasik yöntemin geliştirme süreleri

	Alana Özgü Modelleme	Klasik Yöntem
Geliştirme süresi (Saat)	1 (modelleme) +12 (gerçekleme) =13 (toplam)	15 (kod geliştirme) =15 (toplam)

Çizelge 4.1'de her iki yöntemin geliştirme süresi sonuçları bulunmaktadır. Klasik yöntemde uygulamanın geliştirme süresi alana özgü modelleme kullanarak geliştirilmesinden bir miktar daha fazla sürmüştür. Modelleme dili ile geliştirmede modelleme süresi oldukça kısa olmaktadır. Buna karşın gerçekleme kısmı daha uzun sürmektedir. Gerçekleme kod üreticinin geliştirilmesini de içermektedir.

Çizelge 4.2'deki toplam kod satırı sayısı karşılaştırmasına bakıldığında da modelleme yaklaşımında ortaya çıkan kod sayısı klasik yönteme nazaran daha fazla olduğu görülmektedir. Bunun nedeni alana özgü modelleme yaklaşımında diğer yöntemde olmayan kod üreticinin geliştirilmesidir. Klasik yöntemde elle yazılan kodla modelleme yaklaşımında otomatik üretilen kod miktarı birbirine yakındır. Uygulama çerçevesinin kendi ürettiği kodlar da yine iki yaklaşımda da aynı olmaktadır.

Çizelge 4. 2 İki yöntemin kod satır sayısı karşılaştırması

	Alana Özgü Modelleme	Klasik Yöntem
Kod satırı sayısı (LOC)	1390 (üretilen) +1240 (kod üretici) +510 (framework) =3140 (toplam)	1350 (elle yazılan) +510 (framework) =1860 (toplam)

Uygulama geliştirilirken ve daha sonra kullanım esnasında ortaya çıkan hata sonuçlarına bakıldığında alana özgü modelleme yönteminde daha az hata oluştuğu görülmüştür. Klasik yöntemde ortaya çıkan hatalar elle kodlamadan kaynaklanan hatalardır. Bunun yanı sıra AÖM'de ortaya çıkan hatalar çoğunlukla kod üretici geliştirimi aşamasında ortaya çıkan hatalardır. Bu tür hataların düzeltilmesi de uygulamada sonradan çıkacak hatalara göre daha az maliyetli olmaktadır.

Son olarak AÖM yönteminde elde edilen önemli bir getiri de geliştirmenin klasik yönteme nazaran daha esnek ve kolay hale gelmesidir. Bu esneklik ve basitliğin de yazılım üretkenliğine olumlu etkisi olduğu görülmüştür.

Çizelge 4. 3 Ortaya çıkan hata miktarı karşılaştırması

	Alana Özgü Modelleme	Klasik Yöntem
Hata sayısı	5 (kod üretici) +2 (model) =7 (toplam)	12 (elle yazılan) =12 (toplam)

BÖLÜM 5

SONUÇ VE ÖNERİLER

Yazılım sistemlerinin geliştirilmesi işi yüksek karmaşıklıkta olabilmektedir. Bu sürecin karmaşıklığının azaltılması ve hızlandırılması amacıyla model tabanlı yazılım geliştirme yöntemleri kullanılmaktadır. Alana özgü modelleme yaklaşımı da model tabanlı bir yazılım geliştirme yöntemidir ve doğrudan alan kavramlarını kullanan bir model üzerinden çalışabilen yazılımın kodlarını üretmeyi hedeflemektedir.

Bu çalışmada alana özgü modelleme yaklaşımı detaylı olarak incelenmiş ve bu yaklaşım kullanılarak web tabanlı veri yönetim uygulamaları için alana özgü modelleme dili geliştirilmiştir. Modelleme dili deneysel olarak uygulanarak yaklaşımın üretkenliğe katkısı klasik geliştirme yöntemi ile karşılaştırılmıştır.

Alana özgü modelleme çözümü oluşturulurken ilk olarak web tabanlı veri yönetim uygulamasının alan model sınıfları ve bu sınıflar arasındaki ilişkileri içeren bir üst model tanımı yapılmıştır. Daha sonra bu modelden kod üretimi yapacak olan alana özgü kod üretici yazılmıştır. Deneysel çalışmada bu modelleme dilini kullanarak bir otorizasyon sistemi geliştirilmiştir. Bu sistemin geliştirimi aynı zamanda klasik yaklaşım olan elle kodlama ile de yapılmıştır.

Alana özgü modelleme yönteminin geliştirme süresine olumlu etkisi ilk geliştirmede çok fazla görülmese de benzer sistemler geliştirildiğinde veya geliştirilen sistemde değişiklik veya ekleme yapıldığında daha belirgin bir şekilde ortaya çıktığı görülmüştür. AÖM çözümünde kod üreticinin her ne kadar ilk geliştirme maliyeti yüksek gibi görünse de tekrar eden geliştirme faaliyetleri olduğu durumda maliyetinin ihmal edilebilir olduğu düşünülmektedir.

Klasik yöntem olan elle kodlamada hem hataya açıklığın daha fazla olduğu hem de uzun vadede sistemin geliştirilmesi ve bakımının zor olacağı anlaşılmıştır. AÖM çözümü kullanılarak yapılan geliştirmede, uygulamada yapılan değişiklik ve bakım faaliyetlerinin klasik yönteme göre daha kolay ve maliyetinin oldukça düşük olduğu görülmüştür.

Alana özgü modelleme yaklaşımının getirileri yanında bir takım dezavantajları da bulunmaktadır. Bunlardan en önemlisi AÖM çözümünün ilk geliştirme aşamasının zor ve maliyetli olmasıdır. Burada AÖM yaklaşımı uygulanırken kullanılan araç setlerinin önemi büyüktür. AÖM çözümünün yazılım geliştirme sürecinde etkin bir şekilde kullanılabilmesi için üst modelleme ve kod üretme araçlarının kullanımının kolay olması ve bunların geliştirme ortamına entegre bir biçimde çalışması gerektiği düşünülmektedir.

Yazılımda üretkenliği ve kaliteyi arttırmak, geliştirme ve bakım maliyetlerini azaltabilmek için alana özgü modelleme yaklaşımının kullanılması fayda sağlayacaktır. Ancak AÖM çözümü geliştirme maliyetinin kısa sürede geri dönmemesi ve yazılım geliştiricilerin alana özgü dil ve modeller yerine doğrudan kodlar ile çalışma istekleri nedeniyle alana özgü yaklaşımlar için yeterli yatırım yapılmamaktadır. Bu çalışmanın alana özgü modelleme yaklaşımının yazılım geliştirmede kullanılması konusunda özendirici olması umulmaktadır.

Belirli bir alandaki uygulamalara temel sağlayacak, genişletilebilir ve bakımı yapılabilir mimariye sahip bir yapının oluşturulması zor bir süreçtir. Alana özgü modelleme yaklaşımının kullanımı, bu süreç sonunda istenilen özelliklere sahip uygulamaların oluşturulmasına yardımcı olacaktır. Alana özgü modelleme yaklaşımını Türkçe olarak, gerçek bir uygulama üzerinde anlatan çalışmaların oldukça az olması, alana özgü modellemenin öğrenilmesini ve kullanımını yavaşlatmaktadır. Bu tezde anlatılanların, alana özgü modellemenin etkin bir şekilde kullanılmasına ve anlaşılmasına yardımcı olması çalışmanın bir diğer amacıdır.

KAYNAKLAR

[1] Kelly, S. ve Tolvanen, J.-P., (2008). Domain-Specific Modeling: Enabling Full Code Generation, John Wiley & Sons, New Jersey.

[2] Tolvanen, J.-P. ve Lyytinen, K., (1993). "Flexible Method Adaptation in CASE - The Metamodeling Approach", Scandinavian Journal of Information Systems, 5:51-77.

[3] Kelly, S., Lyytinen, K. ve Rossi, M., (1996). "MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment", Proceedings of the 8th International Conference on Advances Information System Engineering, 20-24 May 1996, Crete, 1-21.

[4] Hillegersberg van, J., Kumar, K. ve Welke, R.J., (1998). "Using Metamodeling to Analyze the Fit of Object-Oriented Methods to Languages", Proceedings of the 31st Hawaii International Conference on System Sciences, 6-9 January 1998, Kohala Coast.

[5] Harel, D., (1987). "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, 8(3):231-274.

[6] Harel, D. ve Gery, E., (1997). "Executable Object Modeling with Statecharts", IEEE Computer, 30:31-42.

[7] Selic, B., Gullekson, G. ve Ward, P. T., (1994). Real-Time Object-Oriented Modeling, John Wiley & Sons, New York.

[8] Burst, A., Spitzer, B., Wolff, M. ve Müller–Glaser, K.D., (1998). "On Code Generation for Rapid Prototyping Using CDIF", OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, 19 October 1998, Vancouver.

[9] Tolvanen, J.-P., Gray, J. ve Rossi, M., (2004). "Domain-Specific Modeling with Visual Languages", Special issue of Journal of Visual Languages and Computing, 15:3-4.

[10] Luoma, J., Kelly, S. ve Tolvanen, J.-P., (2004). "Defining Domain-Specific Modeling Languages: Collected Experiences", Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling, 24 October 2004, Vancouver.

[11] Tolvanen, J.-P., (2004). "Making Model-Based Code Generation Work", Embedded Systems Europe, 8(60):36-38.

[12] Tolvanen, J.-P., (2005). “Making Model-Based Code Generation Work - Practical Examples (Part 2)”, Embedded Systems Europe, 9(64):38-41.

[13] Kärnä, J., Tolvanen, J-P. ve Kelly, S., (2009). “Evaluating the Use of DSM in Practice”, Proceedings of 9th OOPSLA Workshop on Domain-Specific Modeling, 25-26 October 2009, Orlando.

[14] Tolvanen, J-P. ve Kelly, S., (2009). “MetaEdit+: Integrated Modeling and Metamodeling Environment for Domain-Specific Languages”, Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages and Applications, 22-26 October 2006, Orlando.

[15] Sprinkle, J., Mernik, M., Tolvanen, J-P. ve Spinellis, D., (2009). “What Kinds of Nails Need a Domain-Specific Hammer?”, IEEE Software, 26(4):15–18.

[16] Tolvanen, J-P. ve Kelly, S., (2010). “Integrating Models with Domain-Specific Modeling Languages”, Proceedings of 10th Workshop on Domain-Specific Modeling, 17-18 October 2010, Reno, Nevada.

[17] Wijers, G., (1991). Modeling Support in Information Systems Development, Thesis Publishers, Amsterdam.

[18] Batani, C., Lenzerini, M. ve Navathe, B., (1992). Conceptual Database Design: An Entity Relationship Approach, Benjamin-Cummings Publishing Company, Redwood City.

[19] Venable, J., (1993). CoCoA: A Conceptual Data Modeling Approach for Complex Problem Domains, Doktora Tezi, State University of New York at Bimghamton, USA.

[20] Weber, R. ve Zhang, Y., (1996). “An Analytical Evaluation of NIAM’s Grammar for Conceptual Schema Design”, Information Systems Journal, 6:147-170.

[21] Buede, D.M., (1999). The Engineering Design of Systems, John Wiley & Sons, Inc., Hoboken.

[22] Jarke, M., Pohl, K., Weidenhaupt, K., Lyytinen, K., Marttiin, P. ve Tolvanen, J.-P., (1998). Metamodeling: a formal basis for interoperability and adaptability, in: Information Systems Interoperability, John Wiley & Sons, Research Studies Press, Somerset.

[23] Kotteman, J. ve Konsynski, B., (1984). “Information Systems Planning and Development: Strategic Postures and Methodologies”, Journal of Management Information Systems, 1(2):45-63.

[24] ISO-IEC 10027, (1990). Information Technology-Information Resource Dictionary System (IRDS)-Framework, ISO/IEC International Standard, http://www.iso.org/.

[25] OMG Meta Object Facility (MOF) Core Specification, http://www.omg.org/spec/MOF/1.4/PDF, 7 Ağustos 2011.

[26] Be´zivin, J. ve Ploquin, N., (2001). “Tooling the MDA Framework: A New Software Maintenance and Evolution Scheme Proposal”, Journal of Object Oriented Programming, 2001.

[27] Kelly, S., Kalle, L. ve Matti, R., (1996). "MetaEdit+: A Fully Configurable Multi-user and Multi-tool CASE and CAME Environment", Advanced Information Systems Engineering, Lecture Notes in Computer Science, 1080:1–21.

[28] OMG Meta Object Facility (MOF) 2.0 Core Specification, http://www.omg.org/spec/MOF/2.0/PDF, 7 Ağustos 2011.

[29] Albrecht, A.J. ve Gaffney, J.E., (1983). "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, 9(6):639-648.

[30] McCabe, T.J., (1976). "A Complexity Measure", IEEE Transactions on Software Engineering, 2(4):308-320.

[31] Cao, L., Ramesh, B. ve Rossi M., (2009). "Are Domain-Specific Models Easier to Maintain Than UML Models?", IEEE Software, 26(4):19-21.

[32] Mernik, M., Heering, J., ve Sloane, A.-M., (2005). "When and How to Develop Domain-Specific Languages", ACM Comput. Surv., 37(4):316-344.

[33] Kelly, S. ve Pohjonen, R., (2009). "Worst Practices for Domain-Specific Modeling", IEEE Software, 26(4):22-29.

[34] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M. ve Völkel, S., (2009). "Design Guidelines for Domain Specific Languages", 9th OOPSLA Workshop on Domain-Specific Modeling, 25-26 October 2009, Orlando.

[35] Jackson, E. ve Sztipanovits, J., (2009). "Formalizing the Structural Semantics of Domain-Specific Modeling Languages", Software and Systems Modeling, 8(4):451-478.

[36] Reenskaug, T., (2003). "The Model-View-Controller (MVC). Its Past and Present", Java Zone, 18–19 September 2003, Oslo.

[37] Cook, S., Jones, G., Kent, S. ve Wills, A.-C., (2007). Domain Specific Development with Visual Studio DSL Tools (Microsoft .Net Development), Addison-Wesley Longman, Amsterdam.

[38] Greenfield, J., Short, K., Cook, S. ve Kent, S., (2004). Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, Indianapolis.

ÖZGEÇMİŞ

KİŞİSEL BİLGİLER

Adı Soyadı : Alper ÇİFTÇİ

Doğum Tarihi ve Yeri : 07.03.1983 / Arsin

Yabancı Dili : İngilizce

E-posta : alperciftci@gmail.com

ÖĞRENİM DURUMU

Derece	Alan	Okul/Üniversite	Mezuniyet Yılı
Lisans	Bilgisayar Müh.	İstanbul Teknik Üniversitesi	2006
Lise	Fen Bilimleri	N.A.Öğretmen Lisesi	2001

İŞ TECRÜBESİ

Yıl	Firma/Kurum	Görevi
2010-	Fintek A.Ş	Kıdemli Yazılım Uzmanı
2007-2010	HSBC Bank A.Ş.	Yazılım Uzmanı
2006-2007	Bizitek A.Ş.	Yazılım Mühendisi

YAYINLARI

Bildiri

1. Alper Çiftçi, Oya Kalıpsız, "Web Tabanlı Veri Yönetim Uygulamaları için Alana Özgü Modelleme Dili Geliştirilmesi", Ulusal Yazılım Mühendisliği Sempozyumu, pp.109-114, 2011.

2. Alper Çiftçi, Şima Etaner-Uyar, "Kalıcı Durum Evrimsel Algoritmalarda Yerine Koyma Tekniklerinin Deneysel İncelenmesi", Akıllı Sistemlerde Yenilikler ve Uygulamaları Sempozyumu, pp. 236-241, YTÜ Basım-Yayın Merkezi, 2006.