GÖMÜLÜ SİSTEMLER İÇİN MODEL TABANLI YAZILIM TASARIM VE GELİŞTİRME ARACI

MODEL BASED SOFTWARE DESIGN AND DEVELOPMENT TOOL FOR EMBEDDED SYSTEMS

İBRAHİM ARDIÇ

YRD. DOÇ. DR. MEHMET DEMİRER
Tez Danışmanı

Hacettepe Üniversitesi
Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin
Elektrik ve Elektronik Mühendisliği Anabilim Dalı için Öngördüğü
YÜKSEK LİSANS TEZİ olarak hazırlanmıştır.

2015

İBRAHİM ARDIÇ' ın hazırladığı "**Gömülü Sistemler için Model Tabanlı Yazılım Tasarım ve Geliştirme Aracı**" adlı bu çalışma aşağıdaki jüri tarafından **ELEKTRİK ve ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI** 'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Prof. Dr. Selçuk GEÇİM
Başkan ....

Yrd. Doç. Dr. Mehmet DEMİRER
Danışman ....

Doç. Dr. Ali Ziya ALKAR
Üye ....

Yrd. Doç. Dr. Umut SEZEN
Üye ....

Yrd. Doç. Dr. Harun ARTUNER
Üye ....

Bu tez Hacettepe Üniversitesi Fen Bilimleri Enstitüsü tarafından **YÜKSEK LİSANS TEZİ** olarak onaylanmıştır.

Prof. Dr. Fatma SEVİN DÜZ
Fen Bilimleri Enstitüsü Müdürü

Babamın anısına…

ETİK

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü, tez yazım kurallarına uygun olarak hazırladığım bu tez çalışmasında,

- tez içindeki bütün bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- görsel, işitsel ve yazılı tüm bilgi ve sonuçları bilimsel ahlak kurallarına uygun olarak sunduğumu,
- başkalarının eserlerinden yararlanılması durumunda ilgili eserlere bilimsel normlara uygun olarak atıfta bulunduğumu,
- atıfta bulunduğum eserlerin tümünü kaynak olarak gösterdiğimi,
- kullanılan verilerde herhangi bir bölümünü bu üniversite veya başka bir üniversitede başka bir tez çalışması olarak sunmadığımı

beyan ederim.

21/01/15

İbrahim ARDIÇ

ÖZET

GÖMÜLÜ SİSTEMLER İÇİN MODEL TABANLI YAZILIM TASARIM VE GELİŞTİRME ARACI

İbrahim ARDIÇ

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Danışmanı: Yrd. Doç. Dr. Mehmet DEMİRER

Ocak 2015, 110 sayfa

Yazılım projelerinde, iyi tanımlanmış gereksinimlerden ve mimari tasarımlardan satır bazında kod geliştirmek kolaydır. Bu durum, daha karmaşık, milyonlarca satır kod içeren sistemlerde bir ekip tarafından geliştirilebilir. Bu noktada tasarım ve gereksinim arasındaki tutarlılık ön plana çıkmaktadır. Yapılacak olan değişikliklerdeki herhangi bir yanlış anlama tasarımın değişmesine ve sonunda kodun da binlerce satırında değişikliğine sebep olabilir. Bu değişim sırasında geliştirici, doğru olan satırlarda da yanlışlıkla değişiklik yapabilir. Yapılacak olan değişiklik, değişiklik gerektirmeyen yerlere de etkisi sonucunda beklenmeyen hatalara sebep olabilir ve testlerde bu hatanın farkına varılamayabilir. Sonuçta, bütün bu durumlar, üretilecek olan yazılımın maliyetini beklenene göre daha fazla arttırmaktadır. Bu tip karmaşaların önüne geçmek ve maliyeti düşürebilmek amacıyla yazılım sektöründe çeşitli yazılım araçları geliştirilmiştir. Tasarımı daha kolay ifade etmek ve yazılımla eşleştirmek bu problem için bir çözüm olabilir, ancak sistemin büyüklüğü ve birçok alt sistemler mevcut ise bu araçlar bazı noktalarda yetersiz kalabilmektedir.

Bu tez kapsamında, belirtilmiş olan problemlere çözüm bulabilmek için yeni bir model dili ve bu dili şekillendiren model tabanlı yazılım geliştirme ana çatısı geliştirilmiştir. Bu ana çatıda, bu tür yapılmış çalışmalarda olmayan grafiksel ifade edilmiş hem grafik arayüzü hem de mantıksal modellerden, belirlenmiş olan programlama diline özgü kod üretim işlemi gerçekleştirilmektedir. Üretilen bu kod ile birlikte bu yöntemle uçuş ekranları tasarımının geliştirilmesi açısından büyük kolaylıklar sağlayacaktır. Geliştiriciler için sadece gereksinim ve tasarım arasındaki tutarlılık ön plandadır ve bu şekilde herhangi bir tasarım değişikliğinden kaynaklı kodlamada yapılacak olan hataların önüne geçilmiş olacaktır.

Anahtar Kelimeler: Grafik, Tasarım, Model, Kod Üretimi, Geliştirme Aracı

ABSTRACT

MODEL BASED SOFTWARE DESIGN AND DEVELOPMENT TOOL FOR EMBEDDED SYSTEMS

İbrahim ARDIÇ

Master of Science, Department of Electrical and Electronics Engineering

Supervisor: Asst. Prof. Mehmet DEMİRER

January 2015, 110 pages

Developing manual codes in compliance with well-defined requirements and architectural design is an easy task as software projects. For complex systems, a billion lines of code can even be developed by teams. At this point, the requirement and design consistency become crucial. Even little misunderstandings in the amendments to be made leads to change in design and eventually resulting in changes thousands of lines of code. During the operation of changes, the developer may accidentally modify correct lines. Amendments to be made, the places that do not require changes as a result of the impact may cause unexpected errors and these errors may not be detected during the tests. After all, the cost of the software to be produced shall be higher than expected. To avoid this type of complexities and in order to reduce the cost of software, various software tools have been developed in software industry. More understandable and easier expression of design and reasonable matching of this design to the software can be a good solution to the

problem; however in the case that the system size is large and various sub-systems exist, these tools may fail to satisfy requirements adequately.

In this thesis, to find solutions to the problems described, a new model language and the model-based software development framework for this language have been developed. At this framework, the code generation process has been executed to model specified programming language from the models expressed graphically. For developers only, the consistency between requirements and design is at the forefront, and arising from any design changes to be made in coding errors will be prevented accordingly.

Keywords: Graphics, Design, Model, Code Generation, Development Tool

TEŞEKKÜR

Ders dönemi ve tez çalışmam süresince bilgi ve tecrübelerinden faydalandığım, yardım ve katkılarını hiçbir zaman esirgemeyen değerli tez danışmanım Sayın Yard. Doç. Dr. Mehmet DEMİRER' e teşekkürlerimi sunarım.

Öğrenim hayatım boyunca maddi ve manevi destekleriyle daima yanımda olan, başta üzerimde çok emeği olan ve hakkını hiçbir zaman ödeyemeyeceğim annem olmak üzere bütün aileme teşekkürü bir borç bilirim.

Son olarak, tez çalışmam süresince gösterdikleri desteklerden dolayı tüm arkadaşlarıma teşekkür ederim.

İÇİNDEKİLER

Sayfa

KABUL VE ONAY SAYFASI……………………………………………………………i
İTHAF SAYFASI…………………………………………………………………..ii
ETİK……………….………………………………………………………………iii
ÖZET……………………………………………..…………………………..……iv
ABSTRACT…………………………………………………………..………………vi
TEŞEKKÜR…………………………………………………………….………..viii
İÇİNDEKİLER…………………………………………………………………….ix
KISALTMALAR………………..….……………………………………..………xi
ŞEKİLLER……………………………………………………………………….xiii
ÇİZELGELER………………..………………………………..……….…………………xvi
1. GİRİŞ 1
1.1. Model Tabanlı Yazılım Geliştirme Tanımı ve Tarihsel Gelişimi 3
1.2. Model Tabanlı Yazılım Aracı Kullanımının Avantajları ve Dezavantajları...................5
1.2.1. Kullanım Avantajları 5
1.2.2. Kullanım Dezavantajları 6
1.3. Model Tabanlı Yazılım Aracı ve Kod Üreteci Örnekleri 6
1.4. Tez Amaç ve Kapsamı 9
1.5. Tez İçeriği 10
2. MODEL TABANLI YAZILIM YAKLAŞIMLARI 12
2.1. Model Güdümlü Mimari 12
2.2. Üretken Programlama 14
3. KOD ÜRETİM TEKNİKLERİ 17
3.1. Şablon Temelli Üretim 17
3.2. Şablon ve Meta-Model 18
3.3. Satır-içi Kod Üretim 19
3.4. Satır-içi Kod Genişleticisi...21
3.5. Çerçeve İşleyiciler 22
3.6. API Tabanlı Üretim 25
4. GELİŞTİRİLEN MODEL TABANLI YAZILIM VE TASARIM ARACI............................27
4.1. Model Dili..29
4.1.1. Giriş 29
4.1.2. Model Dili Tasarım Prensipleri...29

4.1.3. Arayüz Modeli 30
4.1.4. Grafik Modeli 32
4.1.5. Mantıksal Modeli 40
4.2. Tasarım Editörü 46
4.2.1. Tasarım Prensipleri 46
4.2.2. Editör Arayüz Tasarımı 48
4.3. Kod Üreteci 54
4.3.1. Kod Üretim Algoritmaları 55
5. UYGULAMA 64
5.1. Giriş 64
5.2. Ön Tasarım 64
5.3. Örnek Tasarım Çalışması 66
6. SONUÇLAR VE DEĞERLENDİRME 69
KAYNAKLAR 71
EKLER 75
E.1. Tasarım Örüntüleri 75
E.1.1. Bileşik Tasarım Örüntüsü 75
E.1.2. Yapıcı Tasarım Örüntüsü 76
E.2. Şablon Dosyaları 77
E.2.1. Proje Dosyası 77
E.2.2. Arayüz 79
E.2.3. Grafik 80
E.2.4. Mantıksal 83
E.3. MOBASO ile Model Tasarımı ve Üretilmiş Kod 86
E.3.1. Grafiksel Tasarım 87
E.3.2. Algoritma 1 88
E.3.3. Algorithma 1 için Mantıksal Tasarım 88
E.3.4. Algorithma 2 89
E.3.5. Algoritma 2 için Mantıksal Tasarım 89
E.3.6. Üretilen Kodlar 90
ÖZGEÇMİŞ 110

KISALTMALAR

API	Uygulama Programı Arabirimi (Application Programming Interface)
AST	Soyut Sözdizimi Ağacı (Abstract Syntax Tree)
CORBA	Ortak Nesne İstem Aracısı Mimarisi (Common Object Request Broker)
DSL	Alana Özgü Dil (Domain Specific Language)
DSM	Alana Özgü Modelleme (Domain Specific Modelling)
EMF	Eclipse Modelleme Anaçatısı (Eclipse Modeling Framework)
GMF	Grafiksel Modelleme Anaçatısı (Graphical Modeling Framework)
HMI	İnsan Makine Arayüzü (Human Machine Interface)
IDE	Entegre Geliştirme Ortamı (Integrated Development Environment)
MBSD	Model Tabanlı Yazılım Geliştirme (Model Based Software Development)
MDA	Model Güdümlü Mimari (Model Driven Architecture)
MDSD	Model Güdümlü Yazılım Geliştirme (Model Driven Software Development)
MOBASO	Model Tabanlı Yazılım (Model Based Software)
OCL	Nesne Kısıtlama Dili (Object Constraint Language)
OMG	Nesne Yönetim Grubu (Object Management Group)
OMT	Nesne Modelleme Teknikleri (Object Modeling Techniques)
PIM	Platform Bağımsız Model (Platform Independent Model)
PSM	Platforma Özgü Model (Platform Specific Model)
SCADE	Güvenlik-Kritik Uygulama Geliştirme Ortamı (Safety-Critical Application Devekopment Environment)
SQL	Yapılandırılmış Sorgu Dili (Structured Query Language)
UML	Birleşik Model Dili (Unified Modeling Language)
VAPS	Görsel Uygulama Prototipleme Yazılımı (Visual Application Prototyping Software)
XML	Genişletilebilir İşaretleme Dili (Extensible Markup Language)

XMI	Genişletilebilir İşartetleme Dili Meta Veri Değişimi (Extensible Markup Language Metadata Interchange)

ŞEKİLLER

Şekil 1.1 Şelale Süreci 1
Şekil 1.2 V-Model Süreci 2
Şekil 1.3 Model Güdümlü Mimari İşleyişi [7] 5
Şekil 2.1 MDA Yazılım Geliştirme Yaşam Döngüsü [11] 13
Şekil 2.2 Modelleme Seviyeleri 14
Şekil 3.1 Şablon Temelli Kod Üretimi İşleyişi 17
Şekil 3.2 T4 Örnek Şablon Dosyası 18
Şekil 3.3 T4 Üretilmiş Örnek kod 18
Şekil 3.4 Şablon ve Meta-Model Üretim modeli 19
Şekil 3.5 Satır-içi Üretim modeli 19
Şekil 3.6 C++ Ön İşlemci Komutları 20
Şekil 3.7 Karmaşık Ön İşlemci Komutları 20
Şekil 3.8 C++ Şablonları Örnek Kullanım 21
Şekil 3.9 Satır-içi kod genişleticisi 22
Şekil 3.10 Çerçeve İşleyicisi Modeli 23
Şekil 3.11 Örnek Çerçeve Hiyerarşisi 23
Şekil 3.12 ANGIE Kod Üretim Çalışması 24
Şekil 3.13 ANGIE Çerçeve Tanımı 24
Şekil 3.14 ANGIE Çerçeve Oluşturma 24
Şekil 3.15 ANGIE Çerçeve Yayınlama 24
Şekil 3.16 ANGIE Çerçeve Sınıfı Oluşturma 25
Şekil 3.17 ANGIE Çerçeve Sınıfı Örnekleme 25
Şekil 3.18 API Tabanlı kod üretimi modeli 26
Şekil 3.19 API Tabanlı C# Dili Kod Örneği 26
Şekil 3.20 API Tabanlı C# Dili Kod Üretimi 26
Şekil 4.1 Model Tabanlı Kod Geliştirme Süreci 27
Şekil 4.2 MOBASO Kullanım Senaryosu 28
Şekil 4.3 Model Dili Kütüphanesi 29
Şekil 4.4 Interface XML Örneği 31
Şekil 4.5 Arayüz Sınıfı Diyagramı 32
Şekil 4.6 Grafik nesneleri hiyerarşisi 33
Şekil 4.7 Grafik Sınıfları ve İlişkileri 34

Şekil 4.8 Grafik Modeli Örneği 40
Şekil 4.9 Mantıksal XML Modeli Toplama Örneği 44
Şekil 4.10 Mantıksal Sınıfları ve İlişkileri 45
Şekil 4.11 Model Editörü Kullanım Senaryosu 47
Şekil 4.12 Kullanım Sıralı Şeması 48
Şekil 4.13 Proje Sihirbaz Arayüzü 49
Şekil 4.14 Ana Tasarım Ekranı 49
Şekil 4.15 Proje Paneli 50
Şekil 4.16 Grafik Blokları 50
Şekil 4.17 Mantık Blokları 50
Şekil 4.18 Grafik Tasarım Örneği 51
Şekil 4.19 Mantıksal Tasarım Örneği 51
Şekil 4.20 Nesne Özellikleri Ekranı 52
Şekil 4.21 Arayüz Tanımlama Ekranı 52
Şekil 4.22 Grafik Tasarımı ve Arayüz Ataması 53
Şekil 4.23 Yeni Model Ekleme 53
Şekil 4.24 Proje Penceresi 54
Şekil 4.25 Kod Üreteci Sınıf Şeması 55
Şekil 4.26 Kod Üretimi ana Akış Şeması 56
Şekil 4.27 Arayüz Şablon Örneği 57
Şekil 4.28 Kütle Çekim Formülü Mantıksal tasarım 61
Şekil 4.29 Mantıksal Model Ana Kod Üretimi 62
Şekil 4.30 Mantıksal Model Alt Blok Kod Üretimi 63
Şekil 5.1 Hız Göstergesi Veri Akış Diyagramı 64
Şekil 5.2 Hız Göstergesi – Mantıksal Arabirim Akış Diyagramı 65
Şekil 5.3 Hız Göstergesi – Grafiksel Arabirim Akış Diyagramı 66
Şekil 5.4 Hız Göstergesi - Örnek Grafiksel Tasarım 67
Şekil 5.5 Hız Göstergesi - Örnek Mantıksal Tasarım 68
Şekil E.1 Bileşik Tasarım Örüntüsü Diyagramı [38] 76
Şekil E.2 Yapıcı Tasarım Örüntüsü Diyagramı [35] 76
Şekil E.3 Speed Indicator 86
Şekil E.4 Hız göstergesi grafiksel tasarım 87
Şekil E.5 Sayaç Tasarımı 88
Şekil E.6 Sayaç Kontrol Tasarımı 89

Şekil E.7 İbre Değeri ve Hız Geçerliliği 90

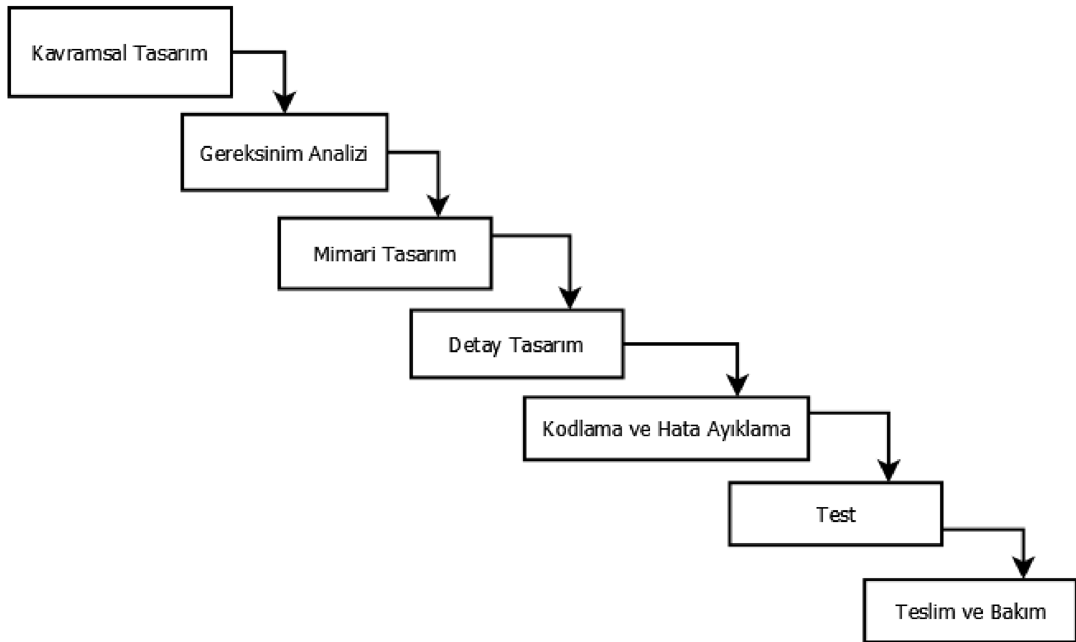
ÇİZELGELER

Çizelge 1.1 Model Tabanlı Yazılım Araçları Tipleri 7
Çizelge 4.1 Arayüz XML etiketleri 30
Çizelge 4.2 Grafik Modeli Ana Tipleri 33
Çizelge 4.3 Grafik Nesnesi Ortak Özellikler 35
Çizelge 4.4 Grafik Modeli Tip Özellikleri 36
Çizelge 4.5 Mantıksal Modeli XML Ana Tipleri 41
Çizelge 4.6 Mantıksal Modeli XML Blok Ana Tipleri 43
Çizelge 5.1 Örnek Grafiksel Tasarım Nesne Çizelgesi 67

1. GİRİŞ

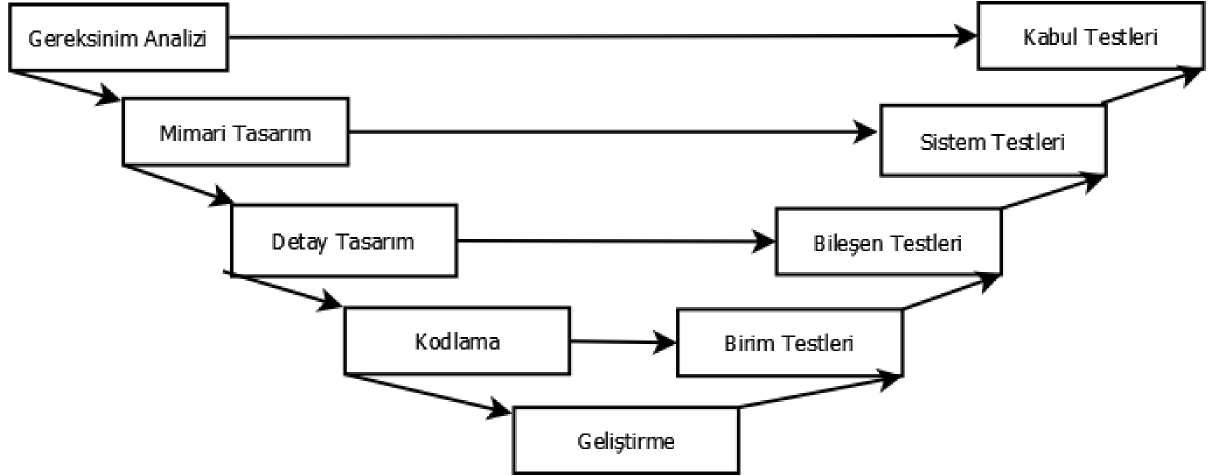
Yazılım, belirli bir müşteri ya da markete özel olarak işlevsel ve fonksiyonel gereksinimlerine göre geliştirilmiş çalıştırılabilir kod parçaları ve bu sırada üretilmiş olan dokümanların tümüdür. Yazılımsız modern bir dünya düşünmek imkânsızdır. Birçok elektrikle çalışan araçlar veya sistemler, bilgisayar tabanlı sistemler ile kontrol edilmektedir. Basit gömülü sistemlerden daha karmaşık sistemlere, dünya çapında çeşitli yazılım sistemleri mevcuttur. Müzik endüstrisi, bilgisayar oyunları sinema ve televizyon dâhil olmak üzere birçok eğlence sektöründe de yazılım yoğun bir şekilde kullanılmaktadır. Daha karmaşık yazılım sistemleri olarak uçak seyrüsefer yazılım sistemleri, nükleer santral kontrol ile güvenlik sistemleri ve banka sistemleri gibi örnekler verilebilir [1].

Her yazılımın bir yaşam döngüsü vardır. Ne kadar büyük ya da kaç kişi rol alırsa alsın tüm yazılımlar bu yaşam döngüsü içerisinde aslında belirli aşamalardan geçmektedir. Bu aşamalar kavramsal tasarım, gereksinim belirleme, tanıma, modelleme, tasarım, kodlama ve hata ayıklama, test, yayınlama, bakım ve gelişim gibi basamaklardan oluşmaktadır [2]. Yazılım geliştirecek olan sistem daha karmaşıklaştıkça geliştirmede kullanılacak olan yöntem ve kurallarında belirli bir sisteme göre geliştirilmesi önem taşımaktadır.



Şekil 1.1 Şelale Süreci

Yazılım yaşam süreci modelleri için geliştirilmiş süreçler şelale, artırımsal ve çevik programlama olarak adlandırılmaktadır. Gömülü sistemlerde ve güvenliğin ön planda tutulduğu alanlarda şelale süreç modeli kullanımı yaygındır. Şelale modeli, Şekil 1.1'de gösterildiği üzere sırasıyla devam eden süreçler bütünüdür. Bir sonraki safhaya geçmek için, içinde bulunulan safhanın tamamen bitmiş olması gerekmektedir. Gereksinimlerin büyük oranda olası değişikliklere uğraması bu modelin kullanımında olumsuz sonuçlar oluşturmaktadır [2]. Bu problemleri engelleyebilmek için bu süreç modeline geri dönüş beslemesi ve sürece paralellik katılarak V modeli geliştirilmiştir [3]. Böylelikle doğrulama ve geçerliliğini sağlama faaliyetlerinde ortaya çıkabilecek problemler daha önceden tespit edilmiş olmaktadır.



Şekil 1.2 V-Model Süreci

V-Model sürecine bakıldığında tüm safhalardaki test karşılığı sonucunda geri dönüşler yapılabilmekte ve hatalar erken safhalarda önlenebilmektedir. Gereksinim ve mimari tasarıma göre oluşturulan tasarımın, kodlama aşamasında tasarıma aykırı kodlama yapılabilmekte ve sonuçta hatanın kaynağının tespit edilmesinde büyük bir zaman kaybı yaşanabilmektedir. Gereksinime göre yapılan tasarımın yanlış kodlanması sonucu hatanın kodda veya tasarımda aranması maliyeti artıran faktörlerdir. Aynı şekilde bu durum tasarım için de olabilir. Gereksinimlerin yanlış anlaşılması ve bununla birlikte hatalı tasarımın sonucunda kodun da bu tasarıma göre geliştirilmesi sonucunda, yine aynı şekilde hatanın önce kodda daha sonra tasarımda aranması, maliyeti dört katına çıkarmaktadır.

Gömülü sistemler için geliştirilecek yazılımlar kritiklik seviyelerine göre bahsetmiş olduğumuz süreçler temel alınarak geliştirilmektedir. Ancak, bu süreç içerisindeki bu problemleri aşabilmek için fazlar arasındaki geçişleri daha kolay, anlaşılabilir şekilde ortaya koymak gerekmektedir. Bu bağlamda birleşik model dili (UML) geliştiricilere çeşitli kolaylıklar sağlayarak tasarımdan iskelet kodu üretmede kullanılmıştır [4].

UML, temelde geliştirilecek olan yazılımın davranışlarını, birbirleri ile ilişkilerini, dış dünya ile bağlantılarını ve kullanım senaryolarını anlatır. Bunu grafiksel çizimler kullanarak yapar. Bu çizimler tasarımı yapısal ya da davranışsal olarak ifade etmeye yarar. UML, geliştirilecek olan yazılımın idame ettirilebilmesi için gerekli olan tasarım prensiplerinin ortaya konulduğu ideal bir metot olarak günümüzde özellikle nesne tabanlı programlama dillerinin kullanıldığı yazılım geliştirme projelerinde kullanılmaktadır. UML kullanılan yazılım projelerinde gerçekleştirilecek olan her yeni güncelleme ile birlikte tasarımdaki değişikliklerin UML modellerine de yansıtılması daha sonraki güncelleme adımları için önemlidir. UML kullanımının avantajlarının yanı sıra uygulanmasında özellikle proje bitiş tarihinin yakınlığı sonucunda bu güncellemelerle ilgili bir takım problemler olmaktadır. Geliştirilecek olan kodun bu modellere uygunluğunun her güncellemeyle tekrar tekrar gözden geçirilmesi, geliştiriciler açısından fazladan bir iş olarak görülmeye ve sonunda bu işlemin eksik yapılması ya da hiç yapılmaması gibi sonuçlar doğurmaktadır.

Yazılım araç üretici firmaları, UML deki iskelet kod üretiminin de ötesine geçebilmek için çalışmalarda bulunmuş ve ürettikleri araçlarla detay tasarımın bire bir karşılığı kodun üretimini sağlayan araçlar geliştirmiştir. Bu sayede geliştiriciler daha çok tasarıma odaklanmış olacaklardır. Tasarımı ve değişikliklerini bu araçlarda gerçekleştirip kodun otomatik olarak üretilmesi sonucunda bahsedilmiş olan problemlerin ortadan kaldırılması amaçlanmıştır.

1.1. Model Tabanlı Yazılım Geliştirme Tanımı ve Tarihsel Gelişimi

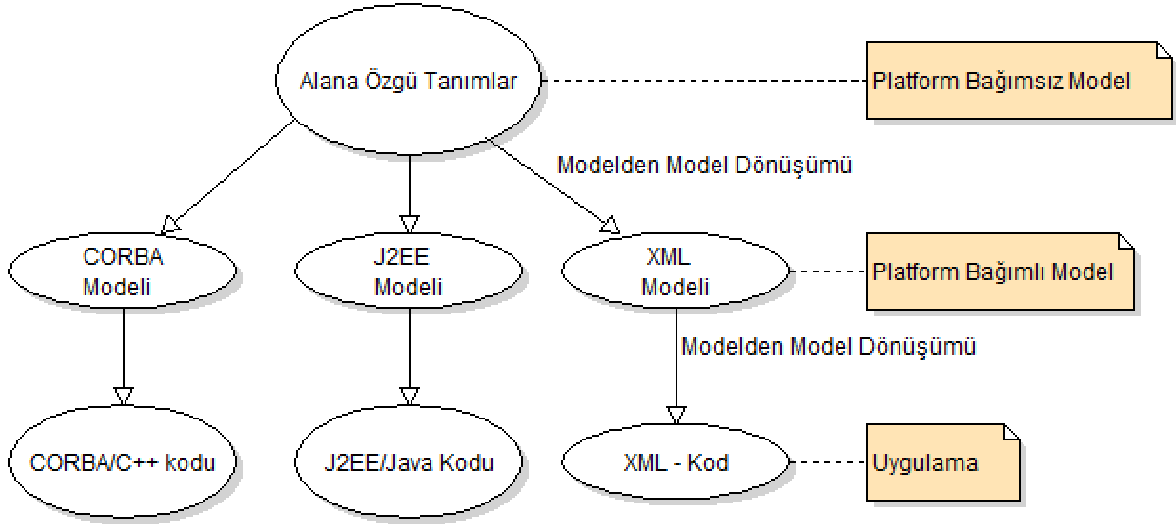
Birçok gömülü yazılım sistemleri hesapladığı ya da sensörlerden gelen sistem değerlerini, durum bilgisini kullanıcılara göstermesi önemli bir durumdur. Öyle ki bu gösterge yazılım sistemi uçağın hızını, yüksekliğini ve benzeri bilgileri içeriyorsa hayati önem taşımaktadır. Gösterilecek olan bilginin doğru bir şekilde elde edilip, kullanıcıya gösterilmesi gerekmektedir. Bu işin doğru bir şekilde gerçekleştirilmesi yazılım geliştirme süreçlerinin ne kadar doğru uygulandığına bağlıdır. Geleneksel yöntemlerde yazılım süreçlerinde yaşanan aksaklıklar, tasarım hatalarından

kaynaklanan değişimi idame ettireme gibi sebepler yazılımın yaşam döngüsünde düzeltilmesi zor problemlere yol açmaktadır.

Modeller, matematikte, fen bilimlerinde ve mühendisliğin tüm alanlarında bir problemin, yapısını, davranışını ve diğer özelliklerini ifade edebilmek için kullanılan güçlü araçlardır [5]. Yazılım mühendisliğindeki karşılığı olarak, çalışılmakta olan sistemi tanımlayan bir takım ifadelerdir. Herhangi bir modelin doğru olduğunu düşünebilmemiz için, çalışılmakta olan sistemi tanımladığı tüm ifadelerinin doğru olması gereklidir [6]. Modellerin yaygın kullanılmalarının yanında, bu modelleri ifade etmek için bilgisayar destekli yazılım mühendisliği araçları kullanılmaktadır [5]. Bu tip araçlarla yazılım alanındaki modelleme üzerinden yapılan geliştirme faaliyetleri model tabanlı yazılım geliştirme tanımına girer.

Model tabanlı yazılım geliştirme yaklaşımı ile birlikte tasarımın net bir şekilde ortaya konulması ile tasarımsal problemlerin birçoğunun önüne geçilmiş oldu. Bu geliştirme yöntemi başka isimlerle de "alana-özgü modelleme", "üretken programlama", "ürün-hattı mühendisliği" ve "kod üretimi" uzun bir süredir kullanımda idi [7]. Model tabanlı yazılım geliştirmede ilk standartlaştırılmış yazılım teknolojisi UML dili olmuştur. 1994 yılında, Grady Booch ve Jim Rumbaugh önderliğinde UML in ilk adımları atılmış ve kullanılmakta olan popüler çizim notasyonlarının birleştirilmesine çalışılmıştır. Bu çizim methodları Booch ve nesne modelleme teknikleri (OMT) olarak isimlendirilmiştir. Daha sonra Ivar Jacobson' unda bu ekibe katılması ile bu grup *üç amigo* olarak anılmıştır. 1997 yılında nesne modelleme grubu (OMG) tarafından standartları belirtilmiş ve günümüzdeki 2.0 versiyonu ortaya çıkmıştır [8]. Ayrıca, UML çizim notasyonlarına ek olarak tasarımların metinsel ifade edildiği nesne kısıtlama dili (OCL) geliştirilmiştir [9].

OMG grubu daha sonra model güdümlü mimari yaklaşımını ortaya koymuştur. Bu yaklaşıma göre öncelikli olarak modeller, herhangi bir platforma bağlılık kurmadan gereksinimlere göre tasarımları gerçekleştirilir. Tasarlanan modeller daha sonra, platforma yönelik değişimlerden geçerler. Bu platformlara örnek olarak J2EE, CORBA, XML verilebilir. Son olarak platforma özgü dönüşümü yapılmış modellerden kod üretilmiş olur. Şekil 1.3'te bu işleyiş verilmiştir.



Şekil 1.3 Model Güdümlü Mimari İşleyişi [7]

Model Tabanlı yazılım geliştirme alanında, OMG standardı haricinde, araca özelleştirilmiş bir yazılım yaklaşımı da mevcuttur. Bu yaklaşıma göre, özel tanımlanmış bir modelin modelleyici bir araç üzerinden tasarımının gerçekleştirilip hedef bir programlama diline göre kod üretiminin yapılmasıdır. Bu yaklaşımda da model güdümlü mimari (MDA) yaklaşımıyla benzerlikleri vardır. Ancak UML dilinin kullanılma zorunluluğu yoktur. Temel farklılık olarak, bu yaklaşımdaki temel modelleme dilinin UML dilinden bağımsız bir şekilde tanımlanmış olmasıdır. Bu farklılık, 3. seviye programlama dillerinin alana özgü tanımlanmış bir dil ile daha fazla soyutlanması ile elde edilir. Bu yaklaşımda geliştirilecek olan sisteme özgü modelleme dili kullanılarak alt seviyedeki prorgramlama dili kodu otomatik olarak üretilir. Böylelikle üretilecek olan yazılım herhangi bir programlama dilinden soyutlanarak, kodlama hatalarından arıdırılmış ve belirli bir kalite seviyesinde geliştirilmiş olur.

1.2. Model Tabanlı Yazılım Aracı Kullanımının Avantajları ve Dezavantajları

1.2.1. Kullanım Avantajları

Model tabanlı yazılım geliştirmenin geliştirilecek olan yazılımın daha kolay idame ettirilebilmesi ve test edilebilmesi avantajların başlıcalarıdır. İlk bakışta model tabanlı yazılım geliştirme yöntemi, geliştirme hızını artırmaktadır. Bu artış kodun tasarlanmış modellerin bir ya da birkaç dönüşümü ile birlikte otomatik üretilmesi ile sağlanmaktadır. Aynı zamanda geliştiricilerin, grafiksel ifade edilmiş modelleri

metinsel kod parçalarına göre daha kolay ve hızlı anlayabilmesi geliştirme hızına önemli bir katkı sağlamaktadır [10].

Otomatik dönüşüm işlemleri ve biçimsel olarak tanımlanmış modelleme dilleri, geliştirilen kodun kalitesini artırır ve bu şekilde tasarlanan mimariler başka projelerde tekrar kullanılabilir. Tasarlanan modeller platformdan bağımsız olacağı için, hedef platformlar için dönüşüm kurallarının tanımlandığı tek bir yer mevcuttur. Bu sayede herhangi bir yeni geliştirme, eklentiler veya yeni teknoloji ile ilgili değişikliklerin bu şekilde yapılması daha kolay olacaktır [7]. Geliştirilmiş modellerin başka platformlara taşınabilirliği model tabanlı yazılım geliştirme yöntemi sayesinde daha kolaylaşmış olacaktır [11].

Model tabanlı yazılım araçlarındaki kod üreteçleri kodun üretilmesini sağlamasının yanı sıra aynı zamanda test kodunun da otomatik üretme kabiliyetinin bulunması, yazılımın test edilme çabalarını önemli miktarda azaltır [12].

Sonuç olarak özetle, model tabanlı yazılım geliştirme araçları ve kodun otomatik olarak üretilmesi, geliştirme maliyetlerinin azalmasına, doğru, kaliteli, tekrar kullanılabilir ve test edilebilir kod üretilmesinde büyük kolaylıklar sağlar.

1.2.2. Kullanım Dezavantajları

Model tabanlı yazılım aracı kullanımıyla ilgili dezavantajlar aşağıdaki şekilde listelenmiştir.

- Tasarımdaki sürümler arası değişikliğin model dosyasından anlaşılması daha zordur.
- Üretilecek olan kodun, kod üretim tekniklerine göre normal metinsel koddan fazla satır üretmesi potansiyel bir problemdir. Böyle bir durumda yazılımın çalışma performansının daha düşük olması beklenilmelidir.
- Ticari model tabanlı yazılım geliştirme araçlarının lisans ücretlerinin yüksek olması, ortalama maliyetli projelerde kullanılmasını engellemektedir.

1.3. Model Tabanlı Yazılım Aracı ve Kod Üreteci Örnekleri

Yazılım endüstrisinde, gömülü sistemler için yazılım geliştirmede kullanılmakta olan birçok ticari model tabanlı yazılım geliştirme aracı mevcuttur. Bu araçlar genellikle kendine özel tanımladığı modellerden kod üretme amacıyla geliştirilmiş yazılımlardır. Üretken programlama yazılım geliştirme sürecine uygundurlar. OMG

grubunun tanımladığı MDA yapısı ile bazı noktalarda paralellikler gösterirler. Genel bir geliştirme yaklaşımı olarak tüm bu araçlar bir grafik editörüne sahiptir. Böylelikle geliştirilecek olan yazılımın parçaları bu editöre çizilecek olan grafik parçalarıyla ifade edilmektedir. Üretilen koda müdahale edilmeden yapılan bu çalışmalarla birlikte geliştirme için harcanacak çaba da düşüşler olmaktadır. Bu tip araçlar kendi aralarında grafiksel ve mantıksal modelleme olarak iki tipe ayrılmaktadır.

Bu araçlara örnek olarak Mathworks Matlab/Simulink, SCADE Suite [13], SCADE Display [14], VAPS [15] ve Rational Rose [16] gibi yazılımlar örnek verilebilir.

Çizelge 1.1 Model Tabanlı Yazılım Araçları Tipleri

Araç	**Kategori**	**Model Dosya Tipi**
SCADE Suite	Mantıksal	XML
SCADE Display	Grafiksel	XML
Matlab Simulink	Mantıksal	Metinsel/XML
Rational Rose	Mantıksal	Metinsel
VAPS	Grafiksel	Metinsel

Çizelge 1.1'de listelenen yazılım araçları, kontrol mantığı geliştirilmesi ya da gömülü sistemlere grafik arabirimi geliştirilmesine yönelik olarak tek bir alana göre kategorileri belirtilmiştir. Tez uygulamamızın bu araçların aksine, geliştirilen uygulama modeline göre belirtilen her iki kategorideki modellerin tek bir araç üzerinden geliştirilebilmesidir. Her iki model tipinin tek bir araçta sağlanmış olması, geliştirme ortamının daha anlaşılır, gerçekleştirilecek olan gereksinimlerin daha kolay takip edilebilir olmasını ve erişilecek olan sonuçlara tek bir araç üzerinden ulaşılabilmesini sağlamaktadır. Ayrıca, uygulamamızda diğer araçlarda kullanılmayan şablon temelli kod üretimi yöntemi ile kod üretimi gerçekleştirilmektedir. Bu sayede şablon temelli dosyaların değişimi ile birlikte üretilecek olan kodun herhangi bir dilin kurallarına göre geliştirilebilmesi sağlanmış olacaktır. Uygulamamızda geliştirilen model dosya tipi okunabilirliğinden ve sistematik bir şekilde işlenebilmesinden dolayı XML olarak belirlenmiştir.

Matlab Simulink gömülü sistemlere kontrol yazılımlarını görsel modeller kullanarak geliştirmek için kullanılan güçlü bir araçtır. Ancak, geliştirme süreçleri boyunca sisteminin tam bir görünümünü sunma yeteneğinde eksiklikler mevcuttur. Sonuçta, aynı sistemi farklı alanlara göre geliştirebilmek için kayda değer bir çalışma yapmak gerekmektedir [17].

Scade Suite ve Scade Display tamamen farklı ana çatılardır. Her iki yapının da model çıktı dosyası XML olarak belirlenmiş olmasına rağmen, ilk bakışta anlaşılabilmesi için daha basit bir yapıda olmaları gerekmektedir.

VAPS yazılım aracının model dosyası ikili sisteme göre oluşturulmaktadır. Bu dosya tipi için sürümler arası farklılıkları tespit etmek mümkün değildir. Bu durum için ayrıca VMF uzantılı metin dosyası üretimi yapmaktadır. VMF metin tabanlı bir dosya olmasına rağmen, herhangi bir betik programlama veya programlama dili ile hiyerarşisine ve yapısına ulaşmak, XML dosya tipine göre daha zor olmaktadır.

Hedef platform için bir grafik ara yüzü yapılacaksa genellikle gömülü sistemler için OpenGL kütüphanesini kullanan yazılımlar geliştirilmektedir. Buna göre bu araçlarla yapılacak olan tasarımın kodu OpenGL desteklemektedir. VAPS ve Scade Display bu tip te kod üretmektedirler.

OMG'nin MDA standardına yönelik olarak geliştirilmiş model tabanlı yazılım geliştirme araçları mevcuttur. Buna örnek olarak Eclipse Modelleme Ana Çatısı (EMF) verilebilir. Bu ana çatı yapısı, Eclipse modelleme aracının çekirdeğini oluşturur. Ayrıca bu ana çatı, bu ana çatıya dayanan geniş yelpazedeki hizmetlerin ve araçların devamlılığını, bağlı modellerin düzenlenmesi, işlenmesi ve soyut sözdizimlerinin tanımlanmasını sağlar [18]. ECore, EMF'in çekirdek modeli (meta-meta model) olarak tanımlanmıştır [19].

Ayrıca akademik alanda kod üretim alanında yapılmış çalışmalar bulunmaktadır. Bu çalışmalar iki ana kategoride incelenmiştir. Bunlardan ilki durum diyagramları ya da özdevinir kullanılarak geliştirilmiş olan kod üretimidir. Genellikle donanım ve yazılımın birlikte değerlendirildiği gömülü sistemlerde kullanılmaktadır. Diğeri ise daha çok dağıtık mimariye sahip sistemler için geliştirilmiş olan kod üretim teknikleridir [20]. PYROSS [21] çalışmasında, uygulamaların dağıtık hafızalı mimarilerde paralel bir şekilde çalışması hedeflenmiştir. Kod üretiminde, gelişigüzel görevlerin zamanlanmasını çizge kuramına dayalı asenkron iletim modelini

kullanarak oluşturmaktadır. Görevler birbirleri ile paylaşılmış hafıza alanı üzerinden mesaj iletimi ile haberleşmektedir [20]. Ayrıca [22]'de, hibrit otomat ana çatısından kod üretimi ile ilgili bir yöntem önerilmiştir. Bu yöntemde, her bir basit bölümlerin (partition) toplanarak koda dönüştürüldüğü ve bu oluşturulan kodların çalışma zamanlamasını düzenleyerek bölümler arasındaki veri bağımlılığını tutarlı kılmıştır [20].

1.4. Tez Amaç ve Kapsamı

Model tabanlı yazılım geliştirme modelinin, yazılım geliştirme süreçlerine birçok katkısı bulunmaktadır. Geliştirilecek olan ürünü kod satırı yazmadan, izlenilebilirliği ve değiştirilmeye daha yatkın olması bu katkıların en önemlilerindendir.

Tezde, model tabanlı yazılım tasarımı ve geliştirme mimarisi için yeni bir model dili, tasarım editörü ve kod üreteci geliştirilmiştir. Bu dildeki temel hedef, benzer endüstri araçlarında olmayan hem grafiksel hem de mantıksal yazılım geliştirme altyapısını aynı platform üzerinde barındırmak, oluşturulan model dili çıktısının daha kolay anlaşılabilir olmasını sağlamaktır. Tez uygulamasının temel aldığı alan özellikle gömülü sistemler olması ve bu sistemler için hedef programlama dili, programlama dili fonksiyonel programlama dili olan ve birçok gömülü sistemde derleyicisinin bulunmasından dolayı C dili seçilmiştir. Geliştirmiş olduğumuz modelleme yapısının OMG standartları ile tanımlanmış olan modelleme yapısından farklı olup, bire bir uyumlu geliştirme amacı güdülmemiştir. Ancak, tezin geliştirme safhasında bu standartlardan faydalanılmıştır.

Model dilinde üç ana temel esas alınmıştır. Bunlar ara yüz, grafik ve mantıksal modelleridir. Grafiksel modeller herhangi bir bilginin veya grafiğin gömülü bir sistemde OpenGL uygulama programlama arabirimi (API) kullanılarak grafiksel olarak gösterilebilmesi için kullanılmaktadır. OpenGL API'leri kullanılması ile üretilen grafiksel kodlar platformdan bağımsız hale gelmiş olmaktadır ve yeni platformda aynı şekilde çalışması için sadece yeni platform için tekrar derlenmesi yeterli olacaktır. Aynı şekilde mantıksal modellerde gösterilecek olan bilginin ya da herhangi bir hesaplama modelinin algoritmasının grafiksel olarak oluşturulabilmesi için geliştirilmiş modelleme tipidir. Arayüz, her iki model tipinin birbirleri ile ya da dışarıdan veri paylaşımı yapabilmesi için ortaya konulmuş bir veri yolu tanımıdır.

Model oluşturmak için belirlenmiş kurallar dâhilinde oluşturulacak olan yazılımın model tanımları XML dosyalarında saklanmaktadır. XML'i seçmekteki amacımız, birçok programlama dili tarafından desteklenmesi, metin tabanlı olması, kendine özgü etiketleri tanımlanabilir, okunabilir olmasıdır. Benzer ticari yazılımlarda grafiğin ve mantıksal model alanlarından sadece birini seçtiklerini belirtmiştik.

Sonrasında, ara yüz, grafik ve mantıksal modellerin tek bir yerden geliştirilebilmesi için belirlenmiş olan model kurallarıyla tasarımları oluşturabilmek ve tasarım çıktılarını model dili çerçevesinde XML tabanlı dosyalarda saklayabilmek ve daha sonra tekrar kullanabilmek amacıyla bir tasarım editörü yazılımı geliştirilmiştir.

Tasarım editöründen üretilmiş olan XML tabanlı model dosyalarından platforma özgü kod üretebilmek için tamamen bağımsız model kod üreteci geliştirilmiştir. Bu dile göre tanımlanmış modeller herhangi bir kod yazımı olmadan C diline özgü kod üretmektedir. Kullanılan şablon dosyalarına göre herhangi bir fonksiyonel programlama dillerine dönüşüm gerçekleştirilebilir. Son olarak ise geliştirmiş olduğumuz bu ana çatıyı kullanarak örnek bir uygulama geliştirilmiş ve sonuçları ortaya konulmuştur.

Tez uygulaması, Microsoft Visual Studio 2010 geliştirme ortamında C# dili kullanılarak gerçekleştirilmiştir. Ayrıca kod üretimi için şablon dosyaları kullanımından kod üretimini sağlayabilen StringTemplate [23] kütüphanesi kullanılmıştır. Üretilecek olan dil ansi-C ile uyumlu olacağından, nesne kodu MinGW platformu ve kütüphaneleri ile birlikte kullanılmıştır.

1.5. Tez İçeriği

Bölüm 2'de, model tabanlı yazılım yaklaşımı ile ilgili bilgilere yer verilecektir. Bu bölümde model tabanlı yazılım ile ilgili standartlar ile ilgili daha ayrıntılı bilgiler ve geliştirme yöntemleri anlatılacaktır.

Bölüm 3'te, kod üretim teknikleri listelenerek, bu teknikler hakkında ayrıntılı anlatımlar verilmiştir.

Bölüm 4'te, tez çalışmasının ayrıntılı incelenmesi yapılmıştır. İncelenen modüllere göre geliştirilen model dilinin yapısı ve içeriği, model dilinin uygulayan modelleme editörü ve son olarak geliştirilmiş modellerden kod üretimini sağlayan kod üreteci

ayrıntılı bir şekilde verilmiştir. Ayrıca geliştirilen yazılım parçalarının standart karşılıkları ile ilgili eşleştirmeler gösterilmiştir.

Bölüm 5'te, tez çalışmasının sonucu olarak geliştirilmiş olan yazılım ile örnek bir uygulama çalışması ve sonuçları verilmiştir.

Bölüm 6'de ise bu çalışmada elde edilen sonuçlar değerlendirilmiş, ileride yapılacak çalışmalarda tezin iyileştirilmesi ve genişletilmesi amacıyla yapılması önerilen çalışmalara değinilmiştir.

2. MODEL TABANLI YAZILIM YAKLAŞIMLARI

Geleneksel yazılım geliştirmede UML geliştirilecek olan yazılımın tasarım prensiplerini ve dokümantasyonunu sağlayan bir tasarım modeli olarak kullanılır. Üretilen dokümantasyonda sistemin tasarımın izlediği yazılımın gereksinimlerine göre detaylı ya da öz bilgisi sunulabilir. Diğer yandan, gerçekte yazılım projeleri dinamiktir ve yaşam döngüsünün erken safhalarındaki önemli sayılabilecek değişikliklere meyillidir. Bu yaşam döngüsünde bu değişiklikler UML modelleri içinde değişikliğe gidilmesine yol açar. Büyük çaplı karmaşık sistemlerde bu UML modellerin de değişime gidilmesi yazılım geliştiriciler tarafından büyük bir yük olarak görülmektedir. Zamanla yerine getirilmeyen bu değişiklikler, UML modellerin kod ile olan eşitliğinin kaybolmasına ve zamanla kullanışsız hale gelmesine neden olur [7]. Geleneksel yazılım süreçlerinin analiz ve tasarım fazlarında modeller karşılaşılan problemleri anlamak ve çözümü tasarlamak için kullanılırken, model tabanlı yazılım geliştirmede modeller üretilmiş kodun karşılığıdır. Herhangi bir değişik kodun da aynı anda otomatik güncellenmesini sağlar. Bu noktada model tanımının kalitesi kod üretecinin ürettiği kodun kalitesi ile doğru orantılı olarak değişmektedir [24], [25].

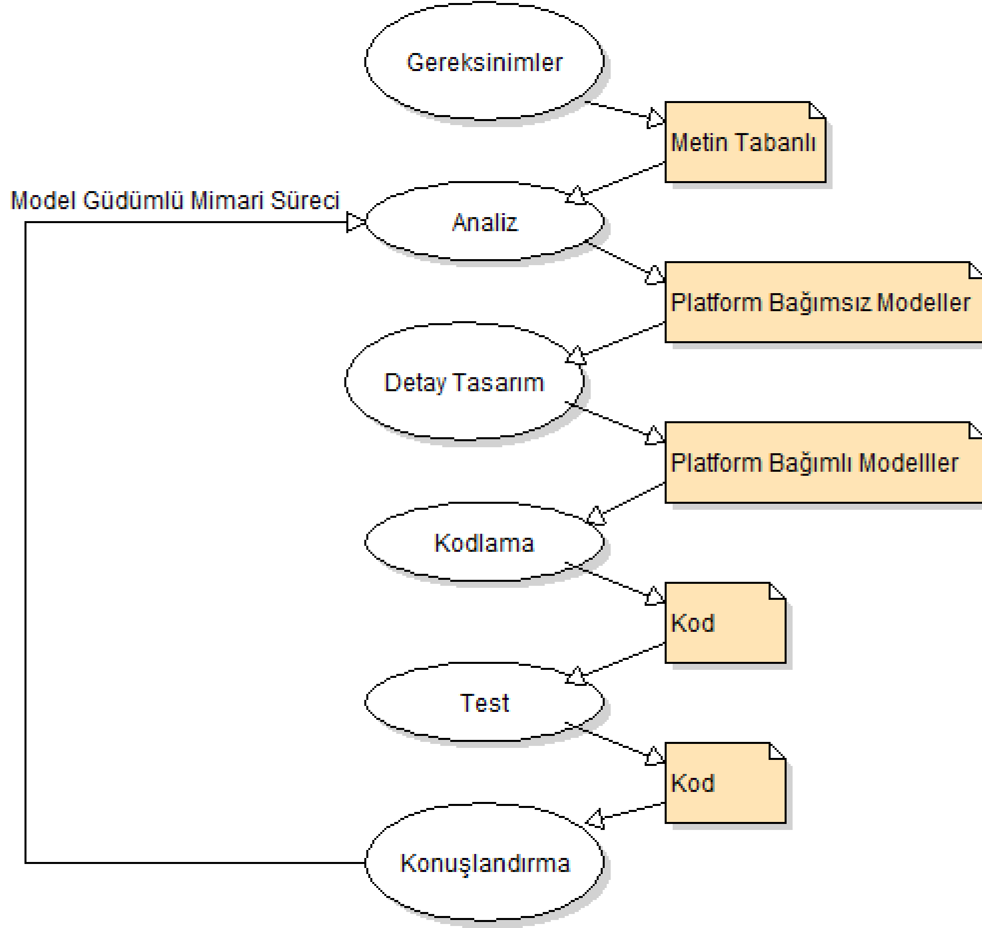
Model tabanlı yazılım yaklaşımları üç ana başlık altında incelenecektir. Bunlar Model Güdümlü Mimari ve Üretken Programlama'dır. Bu yaklaşımlarda, modeller ve sürecin işleyişi belirtilecektir.

2.1. Model Güdümlü Mimari

Model güdümlü mimari standardını nesne yönetim grubu (OMG) ismi altında bir organizasyon tarafından ortaya konulmuştur. Bu yaklaşıma göre yazılım geliştirme süreci platformdan bağımsız modellerle (PIM) başlar. Tanımlanan bu modeller, teknolojinin değişmesine rağmen herhangi bir değişime uğramazlar fakat platform bağımlı modellere (PSM) XML, CORBA gibi modellere dönüştürülebilirler. Sonuçta, bu modellerden kod üretilmiş olur [26].

Şekil 2.1'de MDA yazılım geliştirme sürecinin işleyişi verilmiştir. Süreç, gereksinimlerin ortaya çıkmasından sonra analiz safhasıyla başlar. Bu kısımda gereksinimlere göre yapılan analizler sonucunda tasarımın temelleri olan platform bağımsız modeller ortaya çıkar. Analiz safhası sonrası ortaya çıkan modellerden detay tasarım çalışması gerçekleştirilir. Bu çalışmada gerçek sistemin çalışma prensiplerinden, kısıtlamalarına kadar tüm kalemler tek tek detay tasarımda

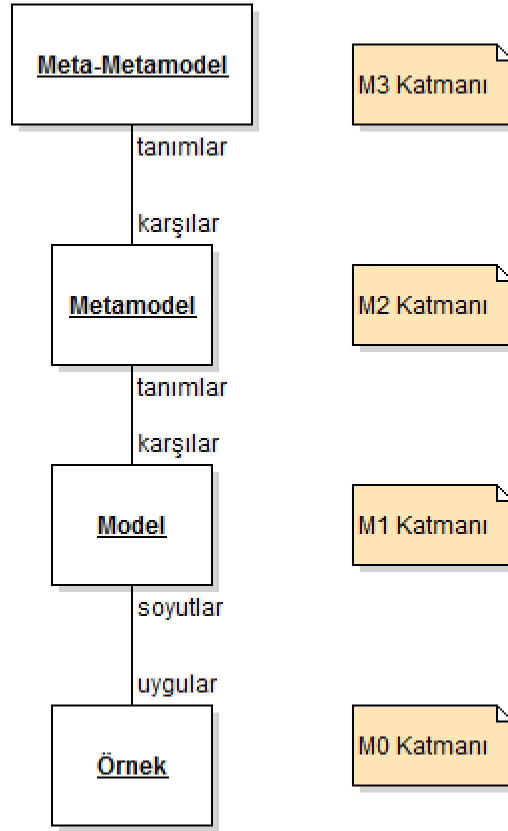
değerlendirilir. Detay tasarım sonucunda, sisteme özgü tanımlanmış tasarımlar ortaya çıkmış olur. Üretilecek olan kod bu safhadan itibaren platforma özgü tanımlanmış olan modeller ile üretilmesi gerçekleştirilir.



Şekil 2.1 MDA Yazılım Geliştirme Yaşam Döngüsü [11]

Platform bağımsız modeller, tanımlayacağı modelin fonksiyonlarını ve davranışlarını ifade eder, fakat herhangi bir teknik altyapı içermez. Burada, UML (Unified Modeling Language) platform bağımlı ve bağımsız kısımlar için bir temel oluşturur.

MDA sürecinde model ve modelin oluşturduğu tanımlar model alanına göre dört ana soyutlamada belirlenir. Bu soyutlamalar arasındaki ilişkiler tanımlanır. Modelin belirlenmesinde etkin rol oynayan kurallar dizisi meta model olarak Şekil 2.2'de görülmektedir. Meta model, modelin belirli ilişkiler ve tanımlar altında tanımlanmasında kullanılır. Şekil 2.2'de gösterilmiş her seviye için benzer bir işleyiş bulunmaktadır. M3 katmanı M2 katmanındaki meta modeli tanımlar. M0 katmanı modelden üretilmiş olan kodu ifade etmektedir [27].



Şekil 2.2 Modelleme Seviyeleri

Eclipse Modelleme Anaçatısı (EMF) Eclipse geliştirme aracı üzerinde, meta model oluşturabilmek için yaygın olarak kullanılan ve açık kaynak kodlu ücretsiz bir anaçatıdır. Bununla birlikte ek olarak modelleme arabirimi olan EMF, Eclipse geliştirme ortamı üzerinde devam ettirilebilen uygulamalar ve araçlar geliştirilmesine olanak tanır. EMF, UML dilindeki kısıtlanmış alt sınıf tanımlarını ve birbirleri ile olan ilişkilerini destekler. EMF'nin UML tanımlanmasında kullanılan XML'in özelleştirilmiş hali olan XMI'ı desteklemesi, Eclipse platformunun kabul görmesinde etkili olmuştur. Ayrıca, EMF tanımlarının görselleştirilmesi için grafiksel modelleme anaçatısı (GMF) geliştirilmiştir. GMF, EMF ile geliştirilmiş olan modellerin ya da yeni geliştirilecek olan model alt yapısının görsel olarak tasarlanmasında kullanılır [5].

2.2. Üretken Programlama

Üretken programlama, yazılım geliştirmede otomasyonun faydalarını ortaya çıkarmakla ilgilidir. Son zamanlarda, yazılım geliştirmede sıklıkla çeşitli araçlar ve methodlar kullanılarak yazılım modüllerinin amacından fazla genellenmeye çalışılması ile geliştirme işlemi gerçekleştirilmektedir. Bu yaklaşım genellikle kaçırılan bitiş tarihleri, hatalar, istenmeyen fonksiyonalite ve tamamen projeyi

kaybetme olarak sonuçlanabilir. Birçok organizasyon için yazılım geliştirme süreçleri halen olgun değildir. Alana özgü modelleme (DSM) yeni bir yaklaşım olarak, üretken programlama yöntemini yazılım geliştirme süreçlerinin ve geliştiricinin üretkenliğindeki kaliteyi arttırmak için kullanır [28].

Yazılım geliştiricilerin üretkenliğindeki en büyük artış Assembler dilinden C ve Fortran gibi üçüncü seviye programlama dillerine geçiş ile olmuştur. Bundan sonraki yeni programlama dillerinin, geliştirme kalitesine ve hızında küçük etkileri olmuştur. Üçüncü seviye programlama dillerinin otomasyona geçişi %400'lük bir üretkenlik artışına sebep olmaktadır. Bu yeni diller geliştiricilerin kendi çözümlerini daha yüksek seviyede ifade edebilmesine imkân sağlar. Bir derleyici otomatik olarak daha düşük seviyeli Assembler kodunu üretir. Alana özgü modelleme de benzer şekilde bir yaklaşımla yazılım ya da sistem geliştirme sürecini geliştirir. Normalde, bir uygulama ya da sistem geliştirmedeki ilk adım, çözümün problem alanı ile direk bağlantılı kavramların kullanımı ve düşünülmesi ile gerçekleştirilien çizimsel tanımlamalardır. Çözümün UML ile daha detaylandırılması ile görsel modelleme dilinde genel bir taslağı ortaya çıkarılmış olur. Programcıların daha sonra bu tanımlamaları kullanarak uygulamayı bir kez daha tanımlar, ancak bunu programlama dili kullanarak gerçekleştirir. Gerçekleştirilen bu iş aynı iş olarak görülebilir, daha da önemlisi daha zor bir iştir çünkü programlama dilleri bu işin daha iyi gerçekleştirilmesinde herhangi bir katkıda bulunmaz. Sadece limitli ve katı söz dizim kuralları sunar. Problemi veya tasarlanacak sistemi programlama dillerinden daha fazla soyutlayarak, başka bir deyimle çözümü alana özgü olan kavramlarla modelleyerek geliştirmek, çözüme daha hızlı ve anlaşılır bir şekilde ulaşılmasını sağlar. Bu yaklaşım, alana özgü kavramların programlama dili kavramları gibi kullanılmasıyla alana özgü modelleme olarak isimlendirilir. Alana özgü modellemede düşük seviyedeki kod parçaları ya da yapılandırma dosyaları, bu yaklaşımla otomatik olarak üretilir. Bu sayede gerçekleştirilecek olan iki aşama olan tasarım ve kod yazma safhası, tek safhaya (tasarım) indirilmiş olur. Ayrıca alana özgü bir tasarım olacağı için herhangi bir programlama dillerini bilmekten çok, belirlenmiş alana ya da sisteme hâkim olan uzman kişiler gereklidir. Diğer yandan yetenekli yazılımcılar ise rutin işlemlerden çok bu modelleme anaçatısının idame ettirilmesi kısımlarında görev alırlar [28].

Günümüzde kullanımda olan birçok model tabanlı yazılım aracı mevcuttur. Bunların çoğunluğu UML desteği vermektedir. Sağlanan bu destek diyagramlardan program kodu çevrimi ve eşzamanlamasıdır. Genellikle üretilen bu kod tam değildir ve belirlli bir kalite seviyesinden yoksundur. Çünkü UML dokumentasyon amaçlı geliştirilmiş bir dildir, kod üretimi için geliştirilmiş bir dil değildir. Yüksek seviyeli modellerden belirli bir kalite de tam bir kod üretimi için problem iki taraflı ele alınmalıdır. Alana özgü modelleme araçları, alan uzmanları eşliğinde hem model dili tasarımını hemde kod üretecini parçasını ayrı ayrı gerçekleştirir. Geliştiriciler de tasarlanmış olan modellerin alan içerisinde doğru çalışmasını sağlayabilmesi için gerekli ortamı oluşturur. Alana özgü kod üreteçleri yüksek kalidete, okunabilirliği fazla ve gerektiğinde gözden geçirilebilen kod üretme yeteneğine sahip araçlardır. Hedef, model dilinin güncellemeleri ile üretilen kodun satır olarak uygun değerde kalmasını sağlamaktır. Üretilen koda elle müdahele ve sonrasında ters mühendislik çalışılabilir bir durum değildir. Çalışılan alanla ilgili herhangi bir güncelleme ya da değişiklik, model dilinde de bir değişiklik ile sonuçlanır ve son olarak kod üreteci de değişen model dili için güncellenir [28].

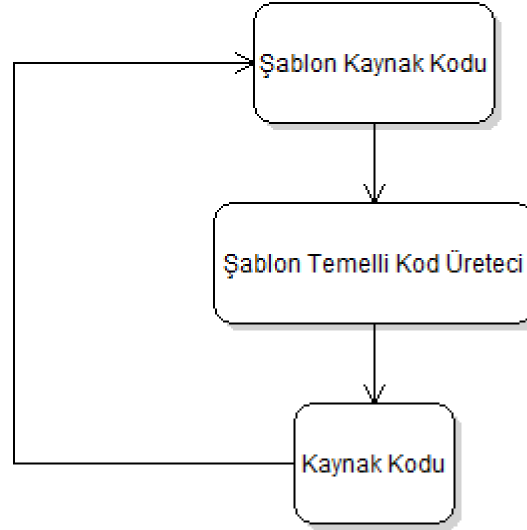
OMG’nin MDA yaklaşımında UML modeli, otomatik olarak başka modellere dönüşüm gerçekleştirilir. Herbir dönüşüm safhasında modeller daha detaylı tanımlanarak kodun üretimi sağlanır. Üretken model yaklaşımında ise alan uzmanlığı gerekmektedir kod üretimi belirlenmiş alanlara göre gerçekleştirilir. Bu tez kapsamında belirlenen alanlar görsel tanımlanabilen grafik arabirimleri ve mantıksal modellerdir. Belirlemiş olduğumuz alan çoğu uygulamalar için genel olduğu için birçok alan için kullanım ve geliştirme imkânı sunmaktadır.

3. KOD ÜRETİM TEKNİKLERİ

Model tabanlı yazılım geliştirme sürecinde kaynak kod, tasarlanan modellerin çeşitli yöntemler kullanılarak dönüştürülmesi ile gerçekleştirilir. Bu bölümde yaygın olarak kullanılan kod üretim teknikleri ve içerikleri verilmiştir.

3.1. Şablon Temelli Üretim

Bu teknik en çok kullanılan kod üretim tekniğidir. Şablon girdi dosyası olarak daha önceden belirli kurallara göre hazırlanmış alana özgü kaynak kod dosyaları kullanılır. Çıkış kaynak kod dosyasının üretilmesinde, bu şablon dosyasındaki özel işaretlenmiş alanların bulunup yerlerine uygun değerlerin yerleştirilmesi ile gerçekleştirilir [29]. Şekil 3.1'de bu yöntemin işleyişi verilmiştir.



Şekil 3.1 Şablon Temelli Kod Üretimi İşleyişi

Bu yöntem için geliştirilmiş ANTLR kütüphanesinin alt modülü olan StringTemplate Kütüphanesi yaygın olarak kullanılmaktadır. StringTemplate kütüphanesi özelleştirilmiş şablon dosyalarını, Java, C#, Python, Ruby ve Scala gibi dilleri kullanarak çıktı dosyalarının üretimini sağlamaktadır [30].

Ayrıca, Microsoft firmasının .NET platformu için geliştirdiği T4 Kod üretimi çözümü de benzer modeli kullanmaktadır. Ancak, T4'ün kod üretimi işlemi T4 motorunun, kontrol blokları, sınıf tanımları ve programlama dil komutlarını içeren şablon dosyalarını kullanması ile gerçekleşir. Bu şablon dosyasıyla birlikte T4 motoru *TextTransformation* sınıfından türetilmiş geçici bir sınıf yaratır. Uygulama alanındaki üretilen bu türetilmiş sınıf, daha sonradan derlenerek çalıştırıldığında, herhangi bir

çıktı kodu (HTML sayfası, C# kodu vb.) üretebilir. Aşağıda, bu yapıdaki şablon ve üretilmiş kod örnek olarak verilmiştir [31].

```
<#@ template language="VB" #>
public partial class ConnectionManager
{
<#
  For Each conName As String in Connections
#>
  private void <#= conName #>(){}
<#
  Next
#>
<#+
  Private Function GetFormattedDate() As String
    Return DateTime.Now.ToShortDateString()
  End Function
#>
```

Şekil 3.2 T4 Örnek Şablon Dosyası

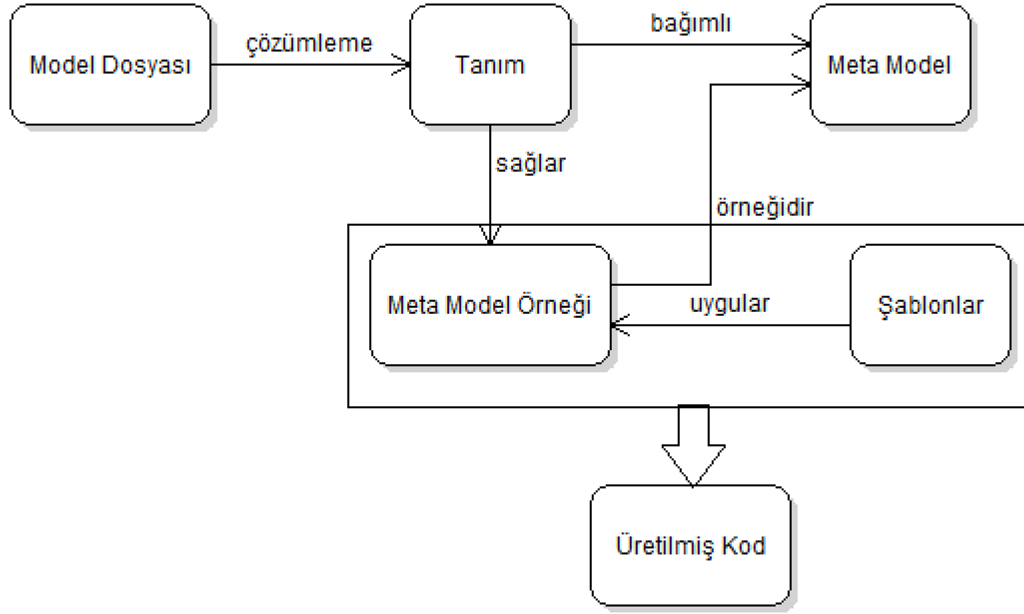
Şekil 3.2'de verilen şablon dosyası ile üretilmiş olan kod dosyası Şekil 3.3'te gösterilmiştir [31].

```
Public Class GeneratedTextTransformation
  Inherits Microsoft.VisualStudio.TextTemplating.TextTransformation
  Public Overrides Function TransformText() As String
    Me.Write("public partial class ConnectionManager{")
    For Each conName As String in Connections
      Me.Write("private void ")
      Me.Write(Me.ToStringHelper.ToStringWithCulture(conName))
      Me.Write("(){}")
    Next
  End Function
  Private Function GetFormattedDate() As String
    Return DateTime.Now.ToShortDateString()
  End Fuction
End Class
```

Şekil 3.3 T4 Üretilmiş Örnek kod

3.2. Şablon ve Meta-Model

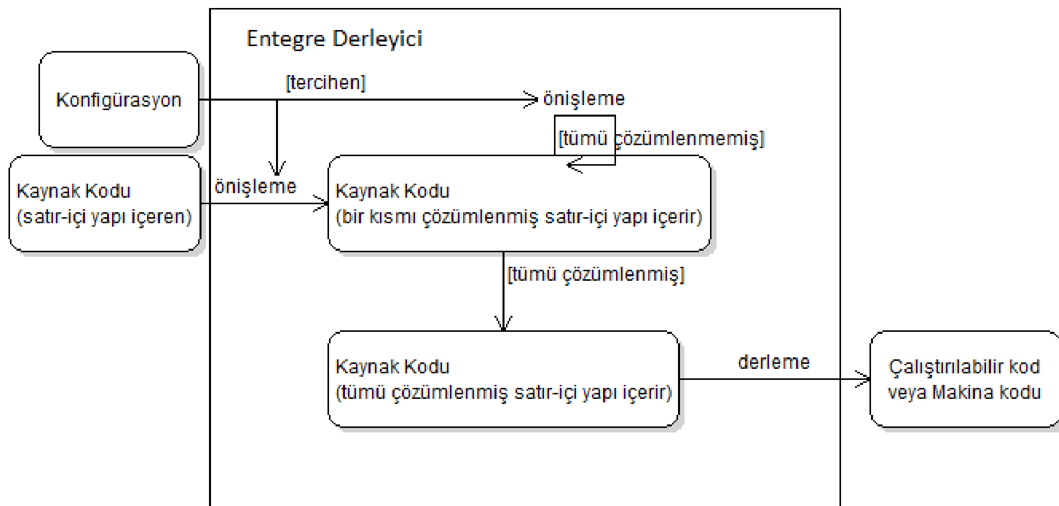
Bu yöntem de şablon temelli üretim yöntemi ile benzer yapıdadır. Bu kısım daha çok şablon temelli kod üretecine gönderilecek olan değerlerin nasıl oluşturulacağı ile ilgilidir. Üretim süreci, belirlenmiş olan model kurallarına göre (meta-model) daha önceden oluşturulmuş XML model dosyaları ile başlar. Bu dosyalar, çözümlenerek meta-model oluşturulur. Oluşturulan meta-model kod üretecine gönderilerek şablon temelli kod üretimi gerçekleştirilir [7]. Sürecin işleyişi Şekil 3.4'de verilmiştir.



Şekil 3.4 Şablon ve Meta-Model Üretim modeli

3.3. Satır-içi Kod Üretim

Bu üretim tekniğinde, herhangi bir dilde yazılmış bir kaynak kodun içerdiği birtakım yapılarla derleme sırasında veya önişleme işlemleri sonucunda daha fazla kaynak kodunun veya makine kodunun üretilmesi ile gerçekleştirilen tekniktir. C++ dilindeki ön işlemci komutları veya C++ dilindeki şablon yapıları bu üretim tekniğine örnek verilebilir [7]. Bu üretim modelinin işleyişi Şekil 3.5'te verilmiştir.



Şekil 3.5 Satır-içi Üretim modeli

C++ için, Şekil 3.6'daki kod örneğinde C++ ön işlemci komutlarının kullanımı gösterilmiştir.

```
#if defined (ACE_HAS_TLI)
static ssize_t t_snd_n (ACE_HANDLE handle,
        const void *buf, size_t len, int flags,
        const ACE_Time_Value *timeout = 0,
        size_t *bytes_transferred = 0);
#endif /*ACE_HAS_TLI*/
```

Şekil 3.6 C++ Ön İşlemci Komutları

Belirtilen *#if, #endif* arasındaki kod parçası ancak *ACE_HAS_TLI* etiketinin tanımlanmış olmasıyla derlenir. Ayrıca parametre alabilen daha karmaşık ifadelerde Şekil 3.7'deki gibi kullanılabilir.

```
#define MAX(x, y) (x < y ? y : x)
#define square (x) x* x
```

Şekil 3.7 Karmaşık Ön İşlemci Komutları

Uygulama kodu içerisinde bulunan *MAX(v1, v2)* gibi ifadeler derleyici tarafından tespit edilerek tanımlanmış olan kurala göre *v1 < v2 ? v2: 1* ifadesi *MAX(v1, v2)* ifadesinin yerine konulur. Tüm bu işlemler sadece metin yer değiştirmesi ile gerçekleştirilen, herhangi bir tip kısıtlaması veya öncelik kuralları olmayan uygulamalardır. Sonuçta, bu yaklaşım sadece basit durumlar için kullanışlıdır [7].

Kıyaslama yapılırsa şablon meta programlama daha yapısal bir yaklaşıma izin verir. Çünkü bu durum derleyicinin Turing fonksiyonel programlama dili olan C++'in C++ tipleri ve sabitleri şablonların derleyici tarafından işlenmesi ile olmaktadır. Böylelikle derleme sürecinde herhangi bir geliştirici bu tip yapılarla tam bir kaynak kodu Şekil 3.8'deki gibi yazabilir [7].

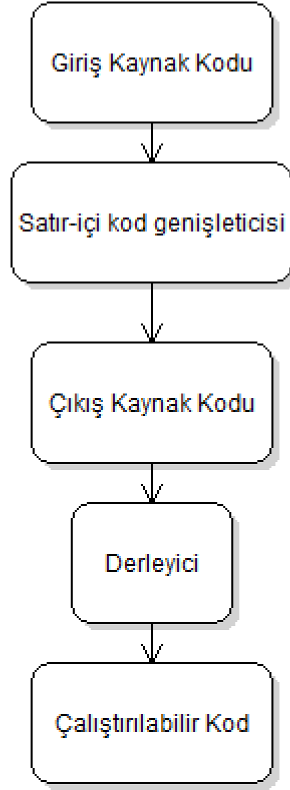
```
#include <iostream>
using namespace std;
#include "../meta/meta.h"
using namespace meta;
struct Stop
{       enum {RET = 1};
};
template<int n>
struct Factorial
{       typedef IF<n==0, Stop, Factorial<n-1> >::RET
        PreviousFactorial;
enum {      RET = (n==0) ? PreviousFactorial:: RET: PreviousFactorial:: RET*n);
};
void  main()
{       cout <<Factorial<3>::RET<<endl;
}
```

Şekil 3.8 C++ Şablonları Örnek Kullanım

3.4. Satır-içi Kod Genişleticisi

Bu yöntemin satır-içi kod üreteci ve şablon temelli modelleri ile benzer bir kullanımı vardır. Ancak satır-içi kod üretecinden farkı programlama dili desteği yerine kaynak kodda kullanılan özel işaretçiler ile gerçekleştirilmesidir. Satır-içi kod genişleticileri kaynak kodda kullanılan özel işaretçileri değiştirerek çıktı kaynak kodunu üretmektedir. Şekil 3.9'daki model bu işleyişi göstermektedir.

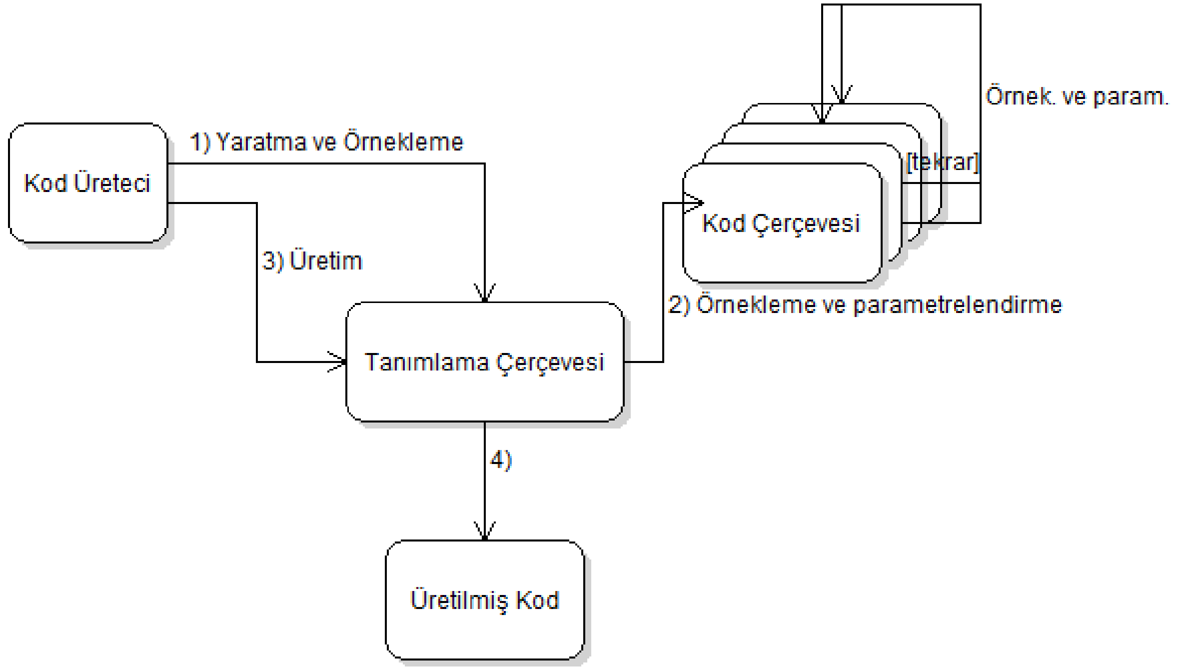
Satır-içi kod genişleticileri şablon temelli kod üretme modelinin aksine özel işaretçiler olarak yaygın olarak yapısal sorgu dili (SQL) komutları kullanılmaktadır. Geliştiriciler bu özel işaretçileri kod genişleticisine gösterebilmek için farklı bir işaretle gösterirler. Tamamı okunan kaynak kodun içindeki özel alanlardaki SQL sorgusunun veya komutunun sonucu tespit edilen yere konulur. Buradaki amaç geliştirilecek olan kodu altyapıdan bağımsız bir şekilde sorguların gözetiminde tutmaktır [29].



Şekil 3.9 Satır-içi kod genişleticisi

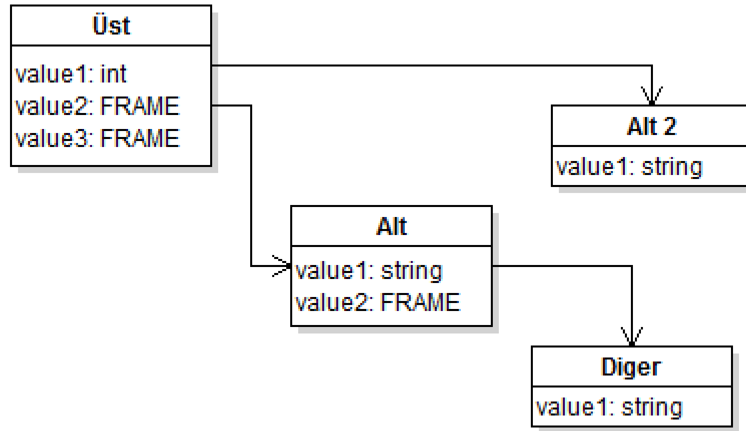
3.5. Çerçeve İşleyiciler

Çerçeveler, çerçeve işleyicilerin merkez elemanı olan ve kod üretimi gerçekleştirilebilecek basit kod tanımlamalarıdır. Nesne tabanlı programlama dillerindeki sınıflar gibi çerçeveler birçok kez örneklenebilir. Örnekleme sürecinde, değişkenler belirli değerlere bağlı kalır. Her bir örneği, barındırdığı değişkenler için kendi içinde kullanacağı değerlerine sahiptir. Bir alt basamak ilerisinde, bu çerçeve örnekleri üretilerek bu şekilde gerçek kaynak kodu üretilmiş olur [7]. Şekil 3.10'da bu modelin işleyişi verilmiştir.



Şekil 3.10 Çerçeve İşleyicisi Modeli

Değişkenlere atanan değerler, metinsel ifadelerden diğer çerçeve örneklerine kadar her şey olabilir. Çalışma anında, çerçeve örneklerinden oluşturulan bu ağaç şeklindeki yapılar sonuç olarak üretilecek olan programın yapısını ifade eder [7]. Şekil 3.11'de bu dönüşümün bir örneği verilmiştir.



Şekil 3.11 Örnek Çerçeve Hiyerarşisi

Çerçeve modeline uygun kod üretimi yapabilen çalışmalara örnek olarak ANGIE [32] ve TGEN [33] verilebilir. ANGIE çalışmasında, Şekil 3.12'deki belirtilen kodun üretim aşamaları verilecektir.

short int aShortNumber = 100;

Şekil 3.12 ANGIE Kod Üretim Çalışması

Verilen kod parçasının değişken yönleri oldukça fazladır. Bunlar değişkenin adı, tipi ve isteğe bağlı ilklendirme parametresidir.

```
.Frame GenNumberElement(Name, MaxValue)
  .Dim vIntQual = (MaxValue > 32767) ? "long" : "short"
  .Dim sNumbersInitVal
  <!vIntQual!> int <Name!> <? = <!sNumbersInitVal!>?>;
.End Frame
```

Şekil 3.13 ANGIE Çerçeve Tanımı

Şekil 3.13'te çerçeve tanımı ilk satırda iki parametre *Name* ve *MaxValue* kullanılarak yapılmıştır. İkinci satırda *MaxValue* değerine göre kullanılacak olan tip kararı *short int*, *long int* yada sadece *int* verilir. Dördüncü satırda üretilecek olan kodun son hali belirlenmiş olur. *<! !>* sözdizimi tanımlanmış olan değişkene ulaşabilmek için kullanılır. *<? ?>* arasında kullanılanlar sadece tanımlanmış ise üretilir. Şekil 3.14'te çerçevenin oluşturulması verilmiştir [7], [32].

```
.myNumbElm = CreateFrame("GenNumberElement", "aShortNumber", 100)
```

Şekil 3.14 ANGIE Çerçeve Oluşturma

Bu aşamaya kadar olan kısımda henüz kod üretimi yapılmamış olup sadece oluşturulmuş olan çerçeve örneği *.myNumElm* değişkenine atanmıştır. Çerçeve örneği üretecin dâhili örnekleme alanında tutulur. Şekil 3.15'te ki komutla birlikte belirlenmiş olan kodun üretimi yapılmış olur [7], [32].

```
.Export(myNumbElm)
```

Şekil 3.15 ANGIE Çerçeve Yayınlama

Çerçeve örneğini bu şekilde üretmek yerine, farklı bir çerçeve örneğine değer olarak verilmesi ile daha karmaşık yapılar oluşturulabilir. Şekil 3.16'da bir *Java* sınıfı örneği verilmiştir [7].

```
.Frame ClassGenerator(fvClassName)
  .Dim fvMembers = CreateCollection()
  public class <!fvClassName!> {
    <!fvMembers!>
  }
.End Frame
```

Şekil 3.16 ANGIE Çerçeve Sınıfı Oluşturma

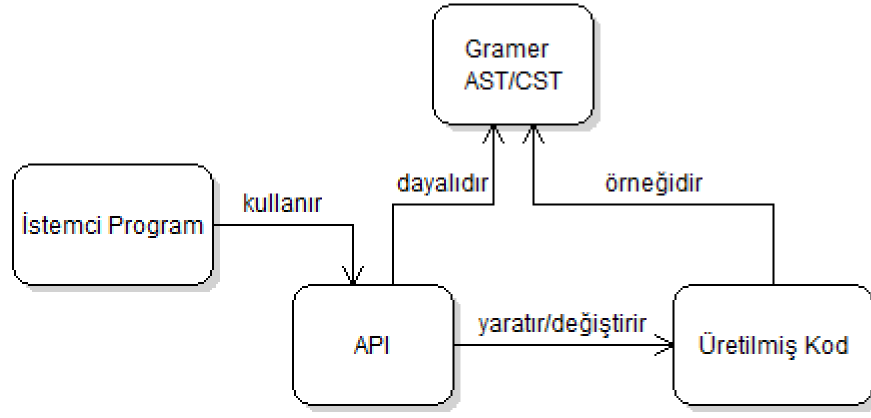
Örnekte, çerçeve üretilecek olan sınıfın ismini parametre olarak almaktadır. Bununla beraber çoklu değişken kullanımı tanımlanmıştır (*fvMembers*). Bu değişkenler harici bir yorumlayıcı dili veya başka bir çerçeve ile değerleri atanabilir. Şekil 3.17'de *i* ve *j* değişkenleri Şekil 3.16'daki çoklu değişken kullanımına örnek olarak verilmiştir [7].

```
.Static myGeneratedClass As ClassGenerator
.Function Main(className)
  .myGeneratedClass =
        CreateFrame("ClassGenerator", className)
    .Add(myGeneratedClass.fvMembers,
      CreateFrame("GenNumberElement", "i", 1000))
.Add(myGeneratedClass.fvMembers,
        CreateFrame("GenNumberElement", "j", 1000000))
.End Function
```

Şekil 3.17 ANGIE Çerçeve Sınıfı Örnekleme

3.6. API Tabanlı Üretim

Uygulama programlama arayüzü (API) tabanlı kod üretiminde, hedef platformun veya dilin elemanlarının üretimi, sağlanan bir API ile yapılmaktadır. Kavramsal olarak bu üreticiler hedef dilinin meta diline özgü üretim gerçekleştirirler, bundan dolayı üretici her zaman belirli tek bir dile özelleştirilmiştir. Şekil 3.18'e bu modelin fonksiyonel işleyişi verilmiştir [7].



Şekil 3.18 API Tabanlı kod üretimi modeli

Şekil 3.19'da .NET platformunda bu modele özgü örnek kodu üretimi verilmiştir.

```
public class Vehicle: object {
}
```

Şekil 3.19 API Tabanlı C# Dili Kod Örneği

Şekil 3.19'daki C# dilindeki kodu üreten kod parçası Şekil 3.20'de verilmiştir.

```
CodeNamespace n = ...
CodeTypeDeclration c = new CodeTypeDeclaration ("Vehicle");
c.IsClass = true;
c.BaseTypes.Add (typeof (System.Object));
c.TypeAttributes = TypeAttributes.Public;
n.Types.Add(c);
```

Şekil 3.20 API Tabanlı C# Dili Kod Üretimi

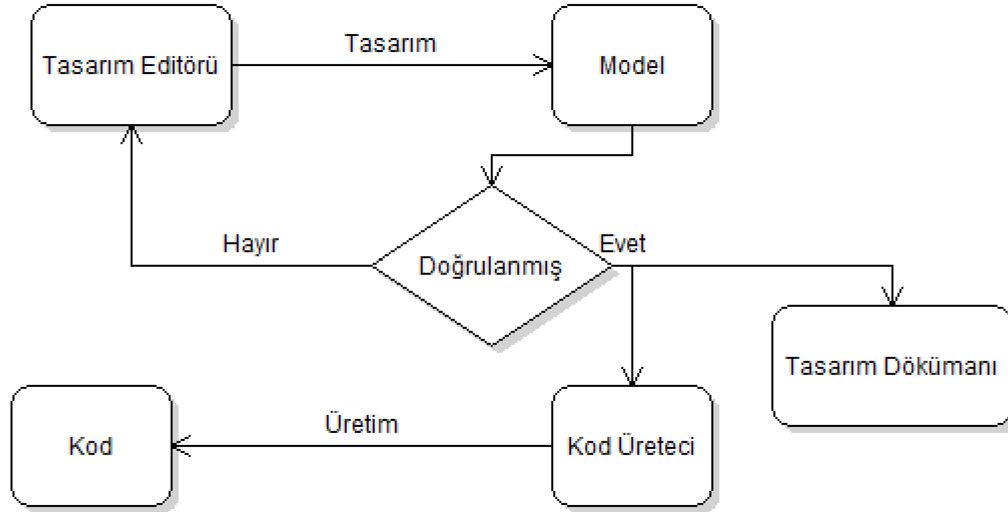
Şekil 3.20'de gösterilen kod parçası, kodun içyapısını temsil eder. Genellikle soyut sözdizimi ağacı (AST) ile ifade edilir [7].

Soyut sözdizimi ağacı, veya sadece sözdizimi ağacı, bir programlama dilinde yazılmış kaynak kodunun soyut sözdizimi yapısının bir ağaç temsilidir. Ağacın her düğümü kaynak koduna karşılık gelen bir yapıdadır [34]. AST bu yönü ile girdilerin yapısını ağaç formunda tutar. Soyut sözdizimi ağaçları da program analizi ve program dönüşüm sistemlerinde kullanılmaktadır.

4. GELİŞTİRİLEN MODEL TABANLI YAZILIM VE TASARIM ARACI

Model tabanlı yazılım sistemlerindeki uygulama alanlarına bakıldığında daha çok gömülü sistemlerde kullanıldığı, ancak bu kullanım yönteminin giderek diğer alanlarda da kendini gösterdiği ortaya çıkmıştır. Gömülü sistemlerde kullanılmasından kaynaklı, geliştirilecek olan modellerde ve kodda nesne tabanlı yaklaşım yerine fonksiyonel akış ve programlama dillerini hedef alan bir bakış tercih edilmiştir. Tezin geliştirme alt yapısında model tabanlı yazılım yaklaşımlarından üretken programlama yaklaşımı hedef alınmıştır. Buna göre geliştirilen yöntemde OMG standartları ile ilgili bir takip söz konusu değildir.

Tez uygulaması üç ana parçadan oluşmaktadır. Şekil 4.1'den görüleceği üzere bunlar Model, Tasarım Editörü ve Kod Üreteci olarak gösterilmiştir. Bu bölümde, tezin konusu olan modelleme dilinin, model tabanlı yazılım aracının ve kod üretecinin tasarım yöntemleri ve aşamaları gösterilecektir.

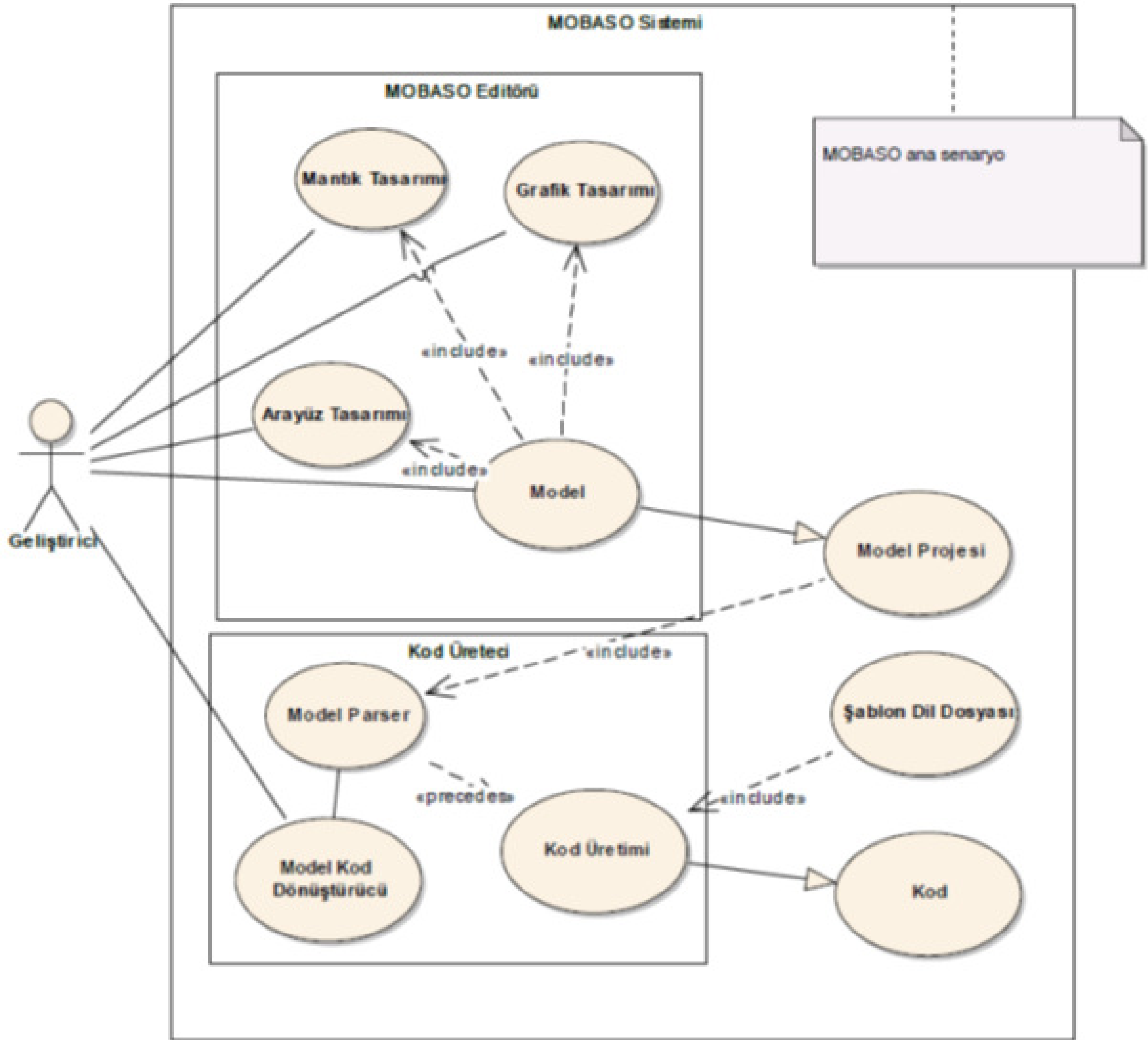


Şekil 4.1 Model Tabanlı Kod Geliştirme Süreci

Belirtilen bu süreçteki ana çatının tasarım editörü, model ve kod üreteci gibi yapıların kullanıcı/geliştirici tarafından kullanım sıralaması Şekil 4.2'de gösterilmiştir. Buna göre kullanıcı/geliştirici öncelikle hedeflenen modeli tasarım editöründe oluşturması gereklidir. Model geliştirimi arayüz, grafiksel ve mantıksal alanlarından herhangi birini içerebilir. Oluşturulacak olan modeller bir veya birden fazla modelden

oluşabilir. Tüm oluşturulan modeller doğruluğun kontrol edildiği model projesi içerisinde saklanarak editör dışına gönderilir. Editör dışında oluşturulan dosyalar model dilinin kurallarına göre oluşturulmuş XML tabanlı dosyalardır. Tasarım editörü daha sonradan oluşturulmuş Model Projesini tekrar görüntüleyip değişiklikleri proje dosyasına tekrar yansıtabilir.

Oluşturulan model projesi harici olarak model kod üreteci tarafından belirlenmiş dilde kod üretimi gerçekleştirmektedir. Bunu yaparken öncelikle XML tabanlı model dosyasının model kurallarına göre ayrıştırıp, ayrıştırılan anlamlı parametrelerin değerleri şablon dosyalarına iletilmektedir. Şablon dosyaları C diline göre özelleştirilmiştir. Bu sayede üretilen tasarım kodu C dilinde olmaktadır.



Şekil 4.2 MOBASO Kullanım Senaryosu

4.1. Model Dili

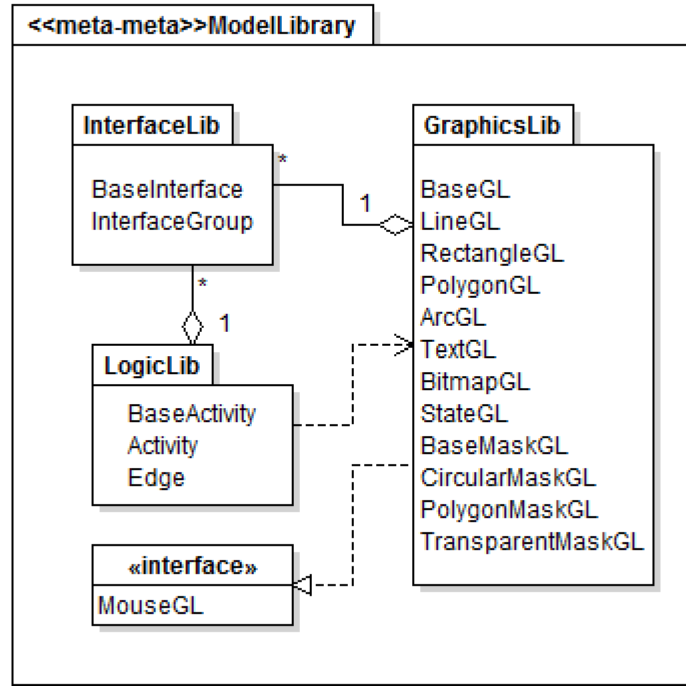
Model dilindeki amaç hem mantıksal hem de grafik arayüz tasarımı destekleyebilen, uygulanması ve okunması kolay bir model dili geliştirme çalışmasını içermektedir. Tanımladığımız bu model dili aynı zamanda geliştirdiğimi bir editör tarafından grafiksel olarak tanımlanarak bu dile dönüşümü sağlanmaktadır. Bu çalışmada kullanmış olduğumuz teknoloji, metin tabanlı, okunması ve uygulanması kolay olduğu için XML olmuştur.

4.1.1. Giriş

Tez projesinde modeller, hem resmi bir tasarım dokümanını hem de kodun birebir eşleniği olarak kabul edilebilir. Bu kısımda belirtilecek olan paketler ve XML etiketleri oluşturulacak olan modelin meta modelini tanımlamaktadır.

4.1.2. Model Dili Tasarım Prensipleri

Model yapısında üç ana sınıf tanımı içermektedir. Model kütüphanesi paketi içerisindeki sınıfların birbirleri ile ilişkisini ve bu sınıflara ek olarak fare eylemlerini gerçekleştirebilmek amacı ile eklenmiş bir arayüz *MouseGL* Şekil 4.3'te verilmiştir.



Şekil 4.3 Model Dili Kütüphanesi

Bu kütüphane birbirinden bağımsız olarak tasarlanıp geliştirilmiş olan tasarım editörü ve kod üretecinde beraber kullanılmasından dolayı bağımsız bir paket olarak geliştirilmiştir.

4.1.3. Arayüz Modeli

Model dosyasında tanımlanan ilk ana bölüm arayüz tanımıdır. Bu kısım geliştirilecek olan grafiksel veya mantıksal tasarımının dış parçalarla veya birbirleri arasındaki köprü olarak yer alır. Diğer bir anlatımla, arayüzler diğer tipteki modellerin giriş çıkış veri yolunu tanımlamaktadır. Çizelge 4.1'e göre arayüzler için kullanılan XML etiketleri ve tanımları verilmiştir.

Çizelge 4.1 Arayüz XML etiketleri

Arayüz XML Etiketleri	Tanım
InterfaceName	Tanımlanan arayüzün isminin bulunduğu XML özelliğidir. Interface etiketiyle birlikte kullanılmaktadır.
var	Bir elemanı ifade eder. Eğer "external" özelliği (attribute) içeriyor ise özellikte belirtilmiş olan harici model dosyasındaki tipleri ifade eder.
name	Eleman veri ismi.
type	Eleman veri tipi.
char	"char" veri tipi.
int	"int" veri tipi.
float	"float" veri tipi.
double	"double" veri tipi.
structure	"structure" veri tipi.
elements	"structure" veri tipinde bulunan eleman listesi
size	Eleman boyutu, kullanılmamışsa tek olarak kabul edilir.

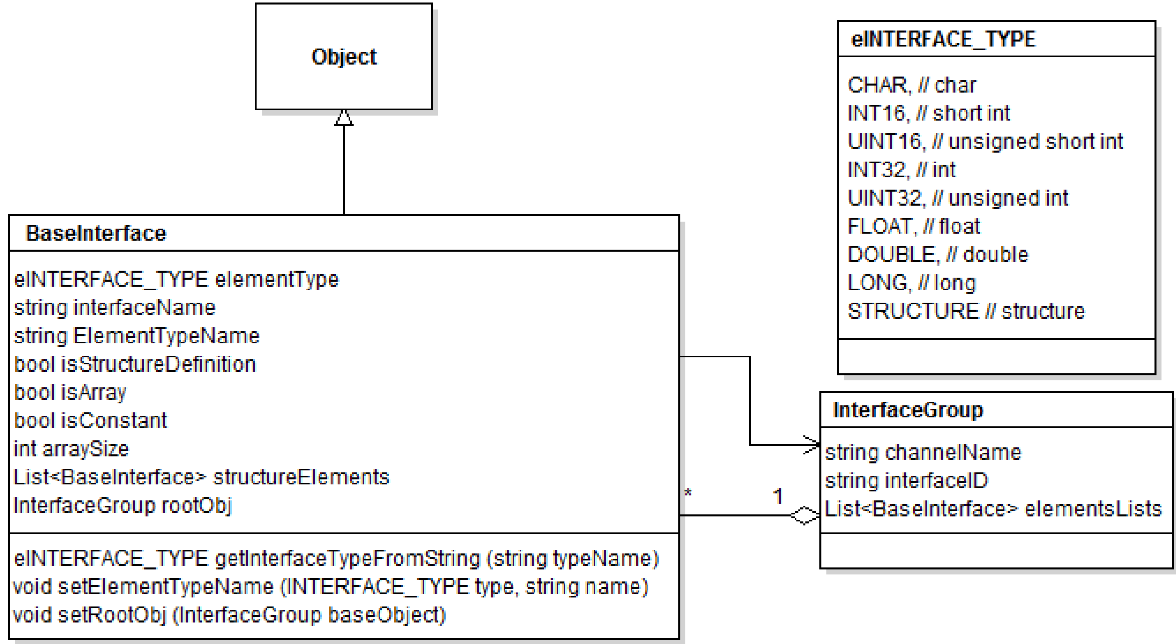
Arayüz tanımı *interface* etiketinin altında tanımlanır. *Interface* veri tipi içermelidir. Her *var* etiketi *interface* grubunun altında tanımlanmış tipleri belirli olan verilerdir. Şekil 4.4'de Çizelge örneğinin bir kullanımı gösterilmiştir.

```
<interface ID="1" InterfaceName="deneme_interface_1">
  <var>
    <name>PointPosition</name>
    <type>structure</type>
    <elements>
        <var>
           <name>PositionX</name>
           <type>float</type>
        </var>
        <var>
           <name>PositionX</name>
           <type>float</type>
        </var>
    </elements>
  </var>
  <var>
     <name>PolygonPoints</name>
     <type>PointPosition</type>
     <size>6</size>
  </var>
  <var external= "RectangleModel.xml"></var>
</interface>
```

Şekil 4.4 Interface XML Örneği

Şekil 4.4'teki örnekte tanımlanan *float* tipinde olan iki elemanlı (*PositionX, PositionY*) yeni bir yapının (*PointPosition*), başka bir verinin tanımlanmasında altı elemanlı bir dizi olarak kullanımı görülmektedir (*PolygonPoints*).

XML arayüz dosyasının oluşturulabilmesi için gerekli olan sınıf yapısı Şekil 4.5'te verilmiştir. Buna göre XML *interface* etiketi *InterfaceGroup* adlı sınıfı işaret eder. *interface* etiketinin altından tanımlanmış olan her *var* veri tanımı ise *BaseInterface* adlı sınıfı işaret etmektedir. *InterfaceGroup, BaseInterface* sınıfına sahip elemanları elementLists listesinde tutar. Sınıfları bu şekilde tutmamızdaki amaç *InterfaceGroup* sınıfının *interface* etiketini uygulaması, tüm veri tiplerini barındırdığı *BaseInterface* sınıfları ile birlikte tek bir *interface*'te tutabilmesidir. Ayrıca *InterfaceGroup, BaseInterface* sınıfından türetilmiş bir sınıftır.



Şekil 4.5 Arayüz Sınıfı Diyagramı

4.1.4. Grafik Modeli

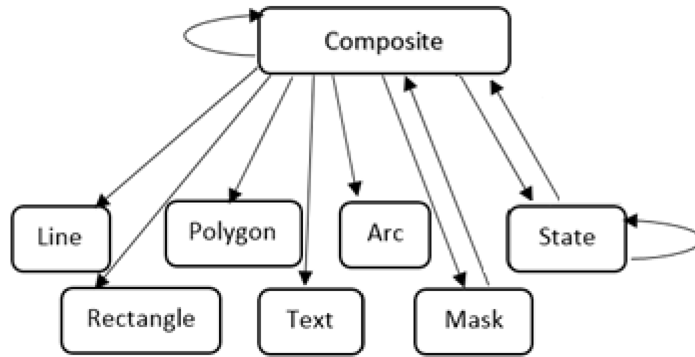
Birçok gömülü sistemde hesaplanmış olan verilerin grafiksel bir şekilde OpenGL Kütüphanesi kullanılarak kullanıcılara gösterilmesi gerekmektedir. Bu durum geliştirme sürecinde önemli bir yere sahiptir. Müşteriler grafik arabirimi ile ilgili isteklerini çok çabuk değiştirebilmekte ve elle yapılan kodlamada bu değişikliklerin ilk örneklerinin hazırlanması çok verimli olmamaktadır. Önerilen modelleme yöntemi ile ilk örneklemeler daha çabuk geliştirilecek ve karşılaşılan değişik istekleri ile kodlama kısmına müdahale etmeden istekler esnek bir şekilde yapılabilecektir.

Sunulan grafik modeli tanımlanan model dosyası içerisinde *graphical* XML etiketi altında tanımlanmaktadır. Aynı şekilde *interface* etiketinde olduğu gibi editör üzerinde tasarlanmış görsel nesnelerin daha sonra XML dosyasında grafik modeline göre özelleştirilmiş XML etiketleri ile gösterilir. Grafik modelini oluşturan elemanlar çizim yapılabilecek basit grafik nesnelerinden oluşmaktadır. Bu grafik tipleri ile daha karmaşık grafik arabirimleri hazırlanabilir ve tekrar daha sonra kullanılabilir. Bu tipler Çizelge 4.2'de gösterilmiştir.

Çizelge 4.2 Grafik Modeli Ana Tipleri

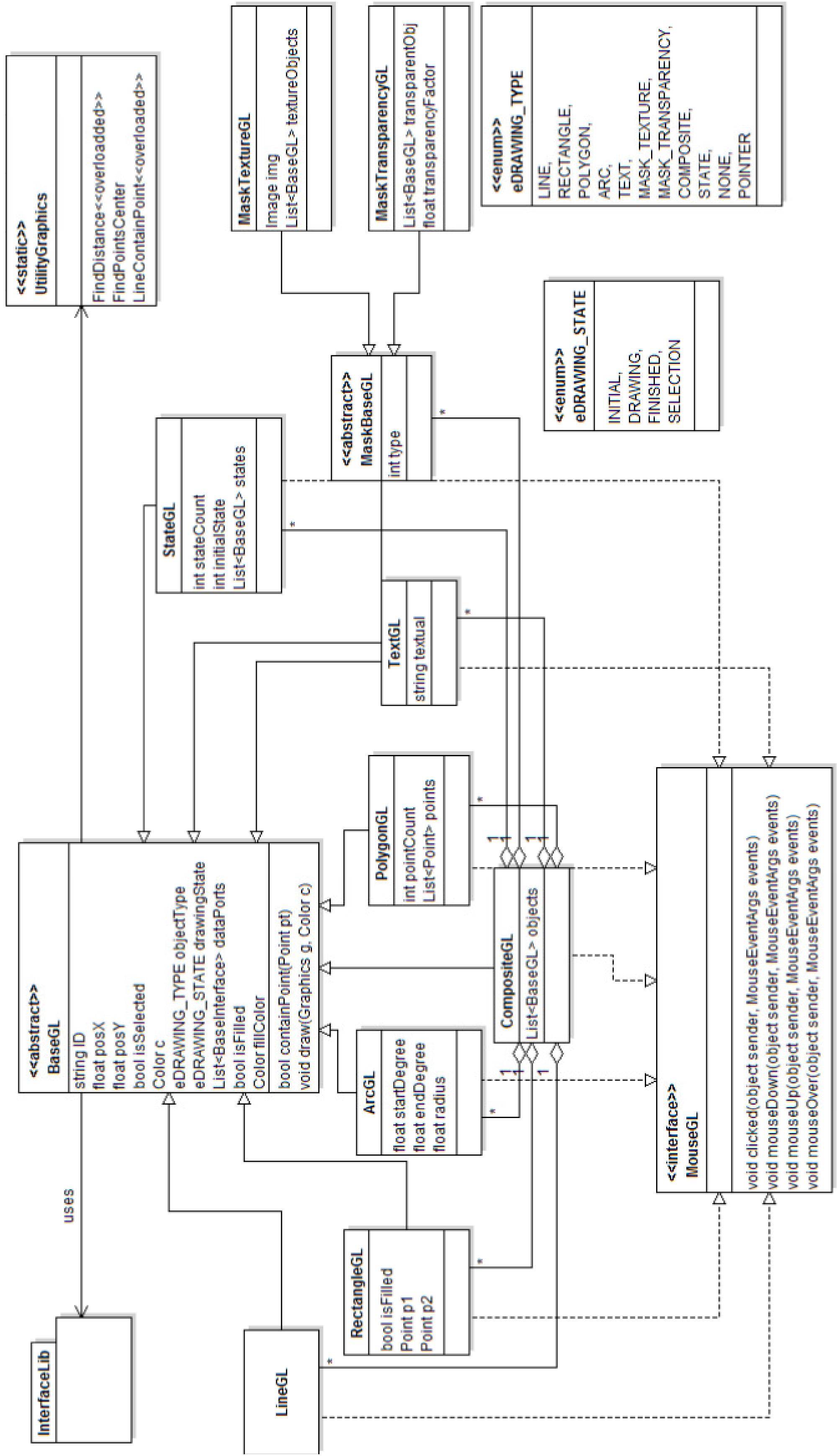
Grafik Tipleri	Tanım
Line,	Doğru nesnesi
Rectangle,	Dikdörtgen nesnesi
Polygon,	Poligon nesnesi
Arc,	Eğri nesnesi
Text,	Metin nesnesi
Mask,	Maske nesnesi
State,	Koşullu nesne
Composite	Bileşim nesnesi

Desteklenen grafik nesnelerinin birbirleri ile olan ilişkileri Şekil 4.6'da gösterilmiştir. Buna göre ana nesne Bileşim nesnesi olarak görülmektedir. Herhangi bir Bileşim nesnesi, diğer tipli nesnelerin tutulduğu bir nesnedir. Bu tasarım ayrıca tasarım örüntülerinde yaygın kullanım şekli olan bileşim örüntüsünün [35] bir kullanım örneğidir.



Şekil 4.6 Grafik nesneleri hiyerarşisi

Bu hiyerarşiye göre oluşturulan sınıflar ve birbirleri ile ilişkileri Şekil 4.7'de gösterilmiştir.



Şekil 4.7 Grafik Sınıfları ve İlişkileri

Koşul nesnesi iki veya daha fazla Bileşim nesnesinin tutulduğu Bileşim nesnesinin özel bir tanımıdır. Hangi Bileşim nesnesinin aktif olarak gösterileceği özelleştirilmiş veri yolu bağlantısının değerine göre çalışma anında belirlenir. Bu nesne genellikle herhangi bir gösterimin veri bağlantısının değerine göre diğer hallerinin gösterileceği durumlarda kullanılır. Örnek olarak sayfa şeklinde tasarlanmış bir grafikte hangi sayfanın aktif olarak görüneceği bu tip grafik nesnesi ile sağlanmaktadır.

Maske nesnesi, kapsadığı nesnelerde özel işlemler gerçekleştirebilmek için kullanılan bir nesnedir. Bu işlemler kapsadığı nesnenin geçirgenliği, dokusu ve maskeleme işlemlerdir. Geçirgenlik özelliği üst üste gelen grafik parçalarının birbirleri ile ilgili renklerinin kaynaşması ile altta kalan nesnenin belirginleşmesidir. Doku özelliği bağlanan grafik nesnesinin sınırlarına bağlı olarak üstüne belirlenmiş bir resmin bağlanmasıdır. Maskeleme özelliği bağlı olan grafik nesnesinin bir boşluk gibi davranarak altta kalan grafik nesnelerinin gösterilmesini sağlayan bir işlemdir. Bu özellik bağlanılan nesnenin tipine göre poligon veya dairesel özellikte olabilir.

Tüm grafik nesnelerinde ortak olan özellikler mevcuttur. Bu özellikler grafiklerin ana sınıfı. Bu özellikler Çizelge 4.3'te verilmiştir.

Çizelge 4.3 Grafik Nesnesi Ortak Özellikler

Özellik	Tanım	İlk Değer
Tanımlama,	Grafik Nesnesinin adı	GrafikTipAdı_(artan_sayı)
Pozisyon,	Grafik Nesnesinin pozisyonu	(0,0) noktası
Döndürme,	Grafik Nesnesi açısı, (Z eksenine göre)	0
Ölçekleme,	Grafik Nesnesi esnetme bilgisi (X ve Y eksenlerine göre)	0, 0
Renk,	Grafik Nesnesi rengi	Beyaz
Doldurma,	Grafik Nesnesi doluluk bilgisi (Çizgi Nesnesi hariç)	Yok

Ortak tanımlanmış özelliklerin ilk değerleri tanımlıdır, örneğin Pozisyon ilk tanım bilgisi (0,0) noktasıdır. Tanımlanmış ortak özellikler, kullanıcı/geliştirici tarafından özel bir değere atanmamış ise, belirlenmiş olan ilk değerler kabul edilir. Modeldeki bu yaklaşım üretilen model dosyasının boyutunun daha az olmasına ve böylelikle geliştirme editörü tarafından daha kolay yüklenmesine yol açar.

Diğer yandan, diğer grafik nesnelerinin kendine özgü farklı özellikleri mevcuttur. Örneğin Arc nesnesinde açı özelliği, Bitmap nesnesinde ise resim özelliği gibi. Çizelge 4.4'te tüm nesneler için tip özellikleri verilmiştir. Tüm belirtilen özellikler harici olarak bir Arayüz modeline bağlanarak, grafiğin davranışları belirlenebilir.

Çizelge 4.4 Grafik Modeli Tip Özellikleri

XML	**Line**	**Rectangle**	**Polygon**	**Arc**	**Text**	**Mask**	**State**	**Composite**
line	*							*
rectangle		*						*
polygon			*					*
arc				*				*
text					*			*
bitmap								*

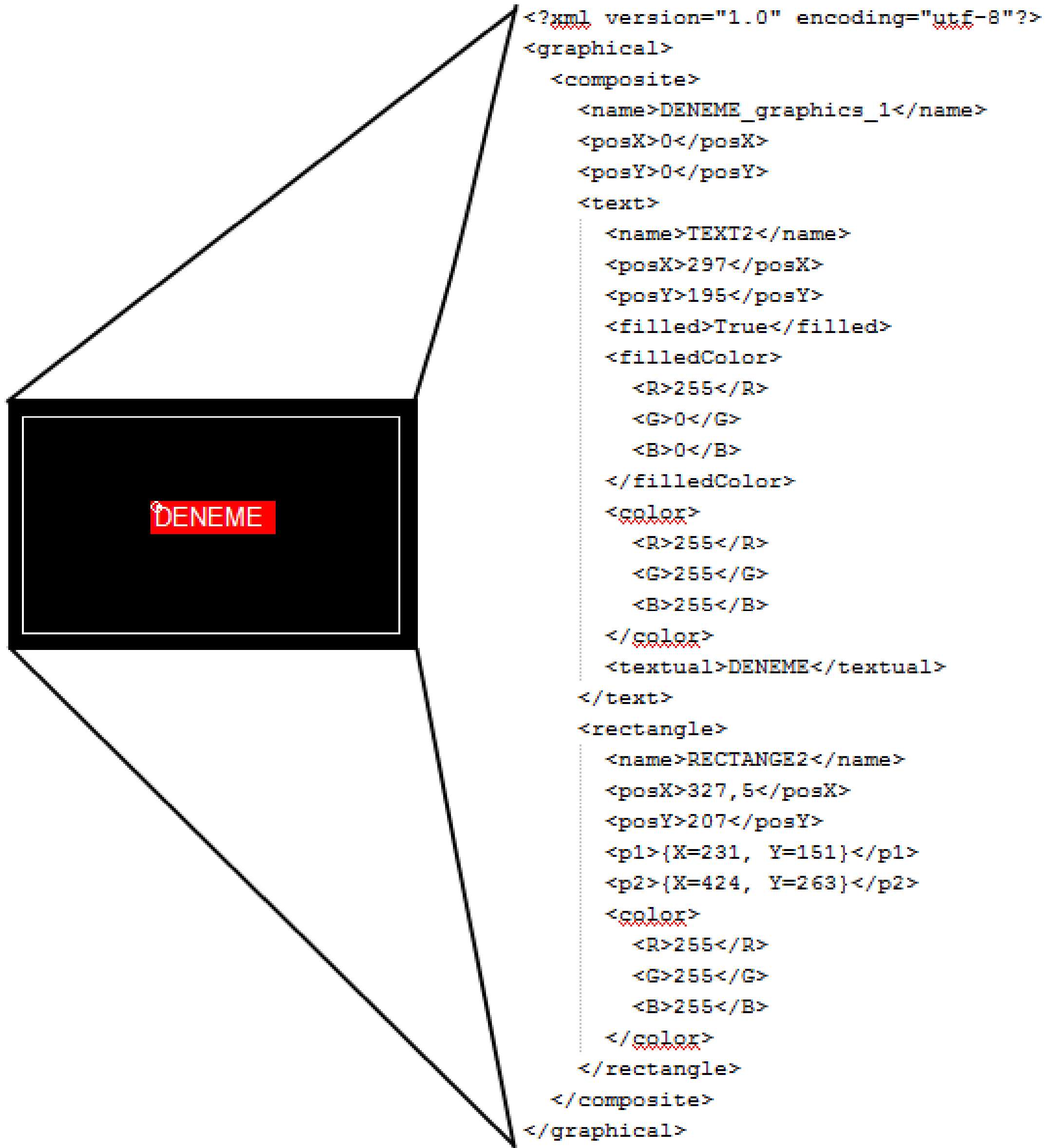
XML	mask	state	composite	name	posX	posY	pointCnt	p1…pn	rotation	scale
Line				*	*	*		*	*	*
Rectangle				*	*	*		*	*	*
Polygon				*	*	*	*	*	*	*
Arc				*	*	*		*	*	*
Text				*	*	*				
Mask	*			*	*	*	*	*	*	*
State		*	*	*	*	*		*	*	*
Composite	*	*	*	*	*	*		*	*	*

XML	Line	Rectangle	Polygon	Arc	Text	Mask	State	Composite
color	*	*	*	*		*	*	*
filled		*	*	*				*
fillColor		*	*	*				*
degree				*				
textual					*			
image								
type						*		
initial							*	
stateCnt							*	
lineThick	*	*	*	*			*	*

XML	Line	Rectangle	Polygon	Arc	Text	Mask	State	Composite
linePattern	*	*	*	*			*	*

Grafik modelleri alt yapısı tanımlarının yanında oluşturulacak olan çıktı modelinin yapısına bakıldığında tüm grafik modelleri <graphical> XML etiketi altında tanımlanmaktadır. “graphical” etiketinin altında en az bir tane C*omposite* kök nesnesi bulunması gerekmektedir. “graphical” etiketi altında tanımlanmış olan herbir *Composite* kök nesnesi birbirinden bağımsız tasarlanmış olan grafik nesnelerini ifade eder. Bunun yanısıra *Composite* nesnesi başka bir *Composite* nesnesi barındırabildiğinden dolayı “graphical” etiketi altında tanımlanmış olan diğer grafik nesnelerini referans vererek kendi içinde tutabilir. Bu sayede geliştirme aşamasında tasarım daha küçük parçalara bölünüp bağımsız ve modüler bir şekilde geliştirilmesi tasarımın dağıtılmasında ve ekip halinde geliştirilmesinde kolaylıklar sağlar. Şekil 4.8’de örnek bir grafik tasarımı verilmiştir.

Şekil 4.8’de, belirli bir tasarımın model dosyasının örneği olarak bir dikdörtgen içerisinde tanımlanmış bir metin verilmiştir. Tasarımın kök nesnesi olarak DENEME_graphics_1 içerisinde TEXT2 tanımlı DENEME metnini gösteren bir metin nesnesi ve RECTANGLE2 tanımlı bir dikdörtgen nesnesi bulunmaktadır. Metin nesnesinde ek özellik olarak doluluğu ve doluluk rengi belirtilmiştir.



Şekil 4.8 Grafik Modeli Örneği

4.1.5. Mantıksal Modeli

Model dilimizde ayrıca mantıksal modellerin de ifade edilebileceği bir blok kümesi tanımlanmıştır. Mantıksal modelin görsel bir şekilde tanımlanması, fonksiyonel programlamanın daha üst seviyelere soyutlanmasını ifade etmektedir. Fonksiyonel programlama dilleri genellikle gömülü sistemlerde, güvenliğin üst düzeyde olduğu projelerde kullanılmaktadır. Çalışma sırasının açıkça belirgin olduğu bu programlama şeklinin görsel bir şekilde tasarımının gerçekleştirmek belirlenen modelleme kurallarının esnekliğine bağlıdır.

Mantıksal modellerde *Activity* nesnesi, Grafik modellerindeki *Composite* nesnesinde olduğu gibi XML mantıksal modelin kök nesnesini oluşturmaktadır. *Activity* nesnesi prosedürel programlamadaki fonksiyonun karşılığına denk gelmektedir. Herhangi bir prosedürel programlamada tanımlanan fonksiyonun girdileri ve çıktıları olan bir ön tanımı mevcuttur. Ayrıca fonksiyon içerisinde geçici olarak kullandığı veri elemanları bulunabilir. Bu elemanlar ile birtakım hesaplamalar sonucu elde edilerek fonksiyonun çıktısının hesaplanmasında kullanılabilir.

Aynı şekilde mantıksal model olan *Activity* içerisinde tanımlanmış olan mantıksal bloklar da prosedürel programlamadaki fonksiyon tanımındaki gibi girdi ve çıktı parametrelerine ve içerisinde ayrıca geçici veri elemanlarına sahiptir. Ortaya konulacak olan mantıksal işlemler mantık bloklarını ve bu veri elemanlarını kullanarak sonuca ulaşır.

Çizelge 4.5 Mantıksal Modeli XML Ana Tipleri

Mantıksal XML Etiketleri	**Tanım**
var	Bir tip elemanını ifade eder. Eğer "external" özelliği (attribute) içeriyor ise özellikte belirtilmiş olan harici model dosyasındaki tipleri ifade eder.
name	Eleman veri ismi.
type	- Eleman veri tipi. - Mantıksal modellerin blok tipi (Çizelge 4.4)
char	"char" veri tipi.
int	"int" veri tipi.
float	"float" veri tipi.
double	"double" veri tipi.
structure	"structure" veri tipi.
elements	- Veri tipleri için "structure" veri tipinde bulunan eleman listesi - Mantıksal ilkel modellerin eleman listesi

size	Eleman boyutu, kullanılmamışsa tek olarak kabul edilir.
inputList	Modelin girdi elemanlarının tutulduğu liste
outputList	Modelin çıktı elemanlarının tutulduğu liste
localList	Model içi tip elemanlarının tutulduğu liste
activity	Mantıksal modelin kök nesnesi
logic	Mantıksal modelin içindeki kullanılan blokların tutulduğu kök nesnesi
edge	Mantıksal Blok çiftlerini ifade eden bağlantı birimi
operation	Mantıksal Blok nesnesinin tamamını kapsayan kök nesnesi
inputsOperation	Blok nesnesinin girdilerin tutulduğu liste
outputsOperation	Blok nesnesinin çıktılarının tutulduğu liste

XML modelindeki mantıksal modelin kök elemanı *Activity* ile başlar. *Activity*'yi *name, inputList, outputList, localList* ve *logic* takip eder. *Logic* XML düğümü hariç geri kalan XML düğümleri Çizelge 4.1'de tanımlanmış olan XML düğümlerini kapsar. Bu noktada tanımlanan *inputList* ve *outputList* Aktivitenin giriş çıkış veri parametrelerini ifade etmektedir. *localList* Aktivite içerisinde kullanılacak olan geçici veri elemanlarını ifade etmektedir. Çizelge 4.5'te mantıksal modelde kullanılan XML düğümleri gösterilmiştir.

Logic XML düğümü altına tanımlanabilen birincil mantıksal bloklar Çizelge 4.6'te listelenmişlerdir.

Çizelge 4.6 Mantıksal Modeli XML Blok Ana Tipleri

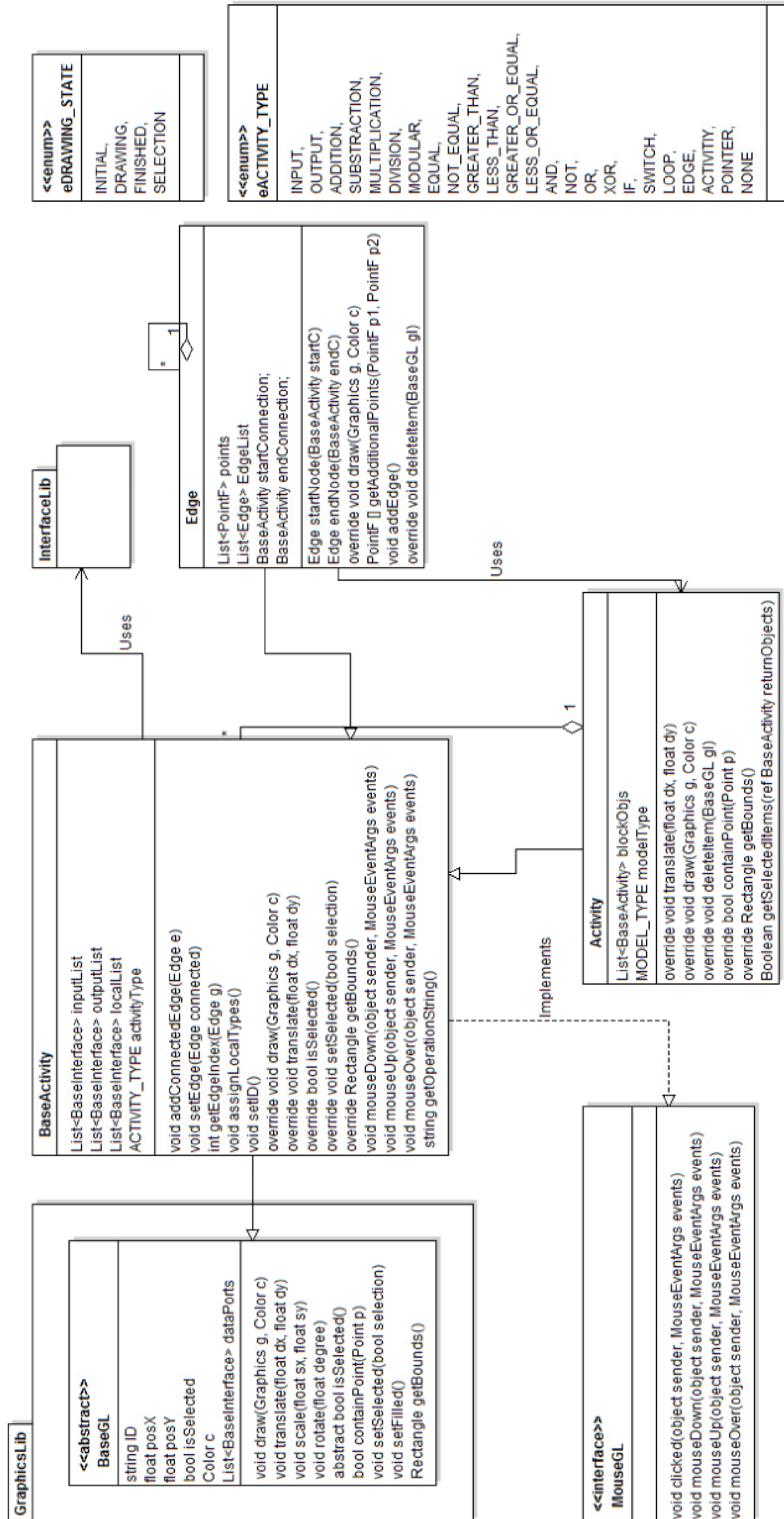
Blok Tipleri	**Tanım**
input	girdi elemanı
output	çıktı elemanı
addition	toplama bloğu
substraction	çıkarma bloğu
multiplication	çarpma bloğu
division	bölme bloğu
modular	mod bloğu
equal	eşittir bloğu
not_equal	eşitsizlik bloğu
greater_than	büyüktür bloğu
less_than	düşüktür bloğu
greater_or_equal	büyüktür eşittir bloğu
less_or_equal	düşüktür eşittir bloğu
and	ve bloğu
not	değil bloğu
or	veya bloğu
xor	özel veya bloğu
if	koşul bloğu
switch	koşullar bloğu
loop	döngü bloğu
edge	bağlantı
state	durum bloğu
activity	aktivite bloğu

Belirlenmiş olan blok tiplerinden olan toplamanın basit bir örneği aşağıdaki şekilde Şekil 4.9'da gösterilmiştir.

```
<operation>
  <name>addition1</name>
  <type>addition</type>
  <posX>100</posX>
  <posY>150</posY>
  <inputsOperation>
        <input>X</input>
        <input>tempX</input>
  </inputsOperation>
  <outputsOperation>
        <output>Y</output>
  </outputsOperation>
</operation>
```

Şekil 4.9 Mantıksal XML Modeli Toplama Örneği

Şekil 4.9'da *addition1* isimli mantıksal toplama bloğunun iki adet *X* ve *tempX* girdilerini alarak, blok sonucunun çıktı verisinde *Y* tutulmasını sağlamaktadır. Mantıksal blokların tüm girdi ve çıktıları *inputList, outputList* ve geçici veri elemanları *localList* düğümlerinde kullanıma sunulmaktadır. Mantıksal bloklarında editör tarafından şekilsel olarak ifade edilebilmesi için ekrandaki koordinatlarının bilinmesi gereklidir. Bu yüzden *posX* ve *posY* XML etiketleri mantıksal blokların BaseGL grafik kütüphanesinden türetilmiş olmasından dolayı ortak özellik olarak bu mantıksal sınıflara aktarılması ile kullanılmaktadır.



Şekil 4.10 Mantıksal Sınıfları ve İlişkileri

4.2. Tasarım Editörü

Model dosya tanımının açık ve kolay anlaşılabilir olması, modellerin tasarım editörü olmadan geliştirilmesine imkân sağlasa da, bu yöntem model tabanlı yazılım geliştirme sürecinde kabul edilemez. Bu yöntemin herhangi bir dördüncü seviye programlama dilleri ile yapılan geliştirme sürecinden farkı yoktur. Tasarım editörü, belirlenmiş olan XML alana özgü dilin modellerini görsel olarak ortaya koyabilmek için gerçekleştirilmiş bir geliştirme aracıdır. Bu kısımda tasarım editörü ve tasarımları ile ilgili kısımlar verilecektir.

4.2.1. Tasarım Prensipleri

Tasarım editörünün tasarlanmasında birkaç husus dikkate alınmıştır. Bu oluşturulan gereksinimler aracılığı ile gerekli olan model dilinin işlenmesi sağlanmıştır. Bu kurallara göre;

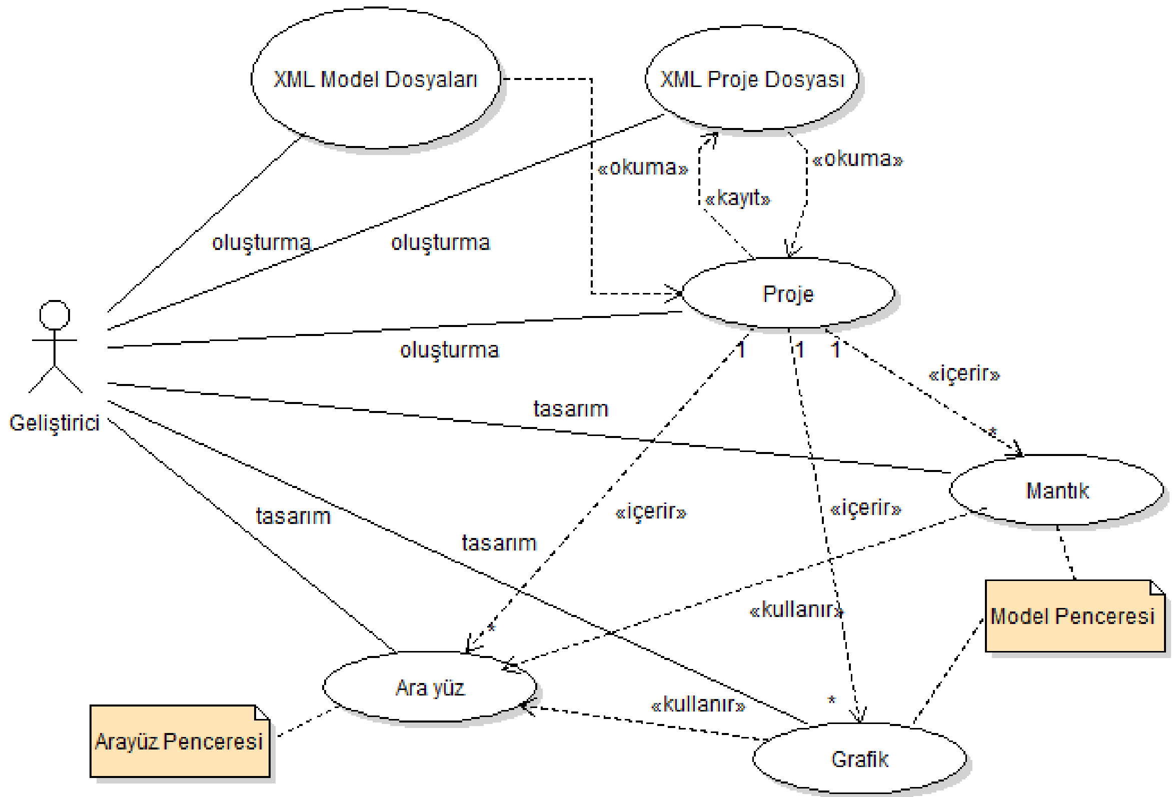
1) Tasarımı bir çizim ekranı vasıtası ile gerçekleştirmek,
2) Oluşturulmuş olan model ana çatısını kullanarak bu ana çatıya uygun bir tasarım dosyasını daha kolay bir şekilde üretmek,
3) Oluşturulmuş olan model dosyalarının içeriğini görüntüleyebilmek,
4) Hatadan arındırılmış bir modeli oluşturularak, kod üretecisin işleyişinin sorunsuz bir şekilde devam etmesini sağlayabilmek,
5) Tasarlanan modelin doğruluğunu kontrol etmek (mantıksal modeller için),
6) Arayüz modellerine yeni eklenecek olan veri tiplerinin kolay bir şekilde eklenebilmesini ve diğer tipli modellerle bağlantısını oluşturmak,
7) Çizimi gerçekleştirilecek olan modellerin içerdiği birincil nesneleri bir menüden seçilebilmesini, eklenmiş olan birincil nesnelerin isim, renk koordinat gibi ortak özelliklerinin bir pencere üzerinden kontrolünü sağlamak,
8) Çizim ekranına uygulanmış olan birincil nesnelerin seçilme, kaydırma (pozisyon değişimi) gibi işlemleri gerçekleştirmek,
9) Modellerin kaydını ve modellerin kod üretimini kod üreteci parçasına gönderilmesini tasarım editörü üzerinden sağlamak,
10) Kod üretimi sonrasında oluşturulan C kodlarından çalıştırılabilir programı harici bir betik programlama fonksiyonu ile tasarım editörü üzerinden gerçekleştirmek.

Belirlenen kıstaslara göre geliştirilen model tasarım editöründe olması gereken ana tasarım ekranları; çizim, çizim özellikleri, arayüz tanımlama değiştirme, birincil blokların bulunduğu araç çubuğu, model kaydetme, kod dönüştürme, yapılandırma araç çubuğu gibi arayüz ekranlarıdır.

Ana ekranların belirlenmesi ile birlikte, ara ekranlarında daha net ortaya çıkabilmesi için tasarım editörünün kullanımına yönelik ana kullanım senaryoları şu şekilde ortaya çıkmıştır.

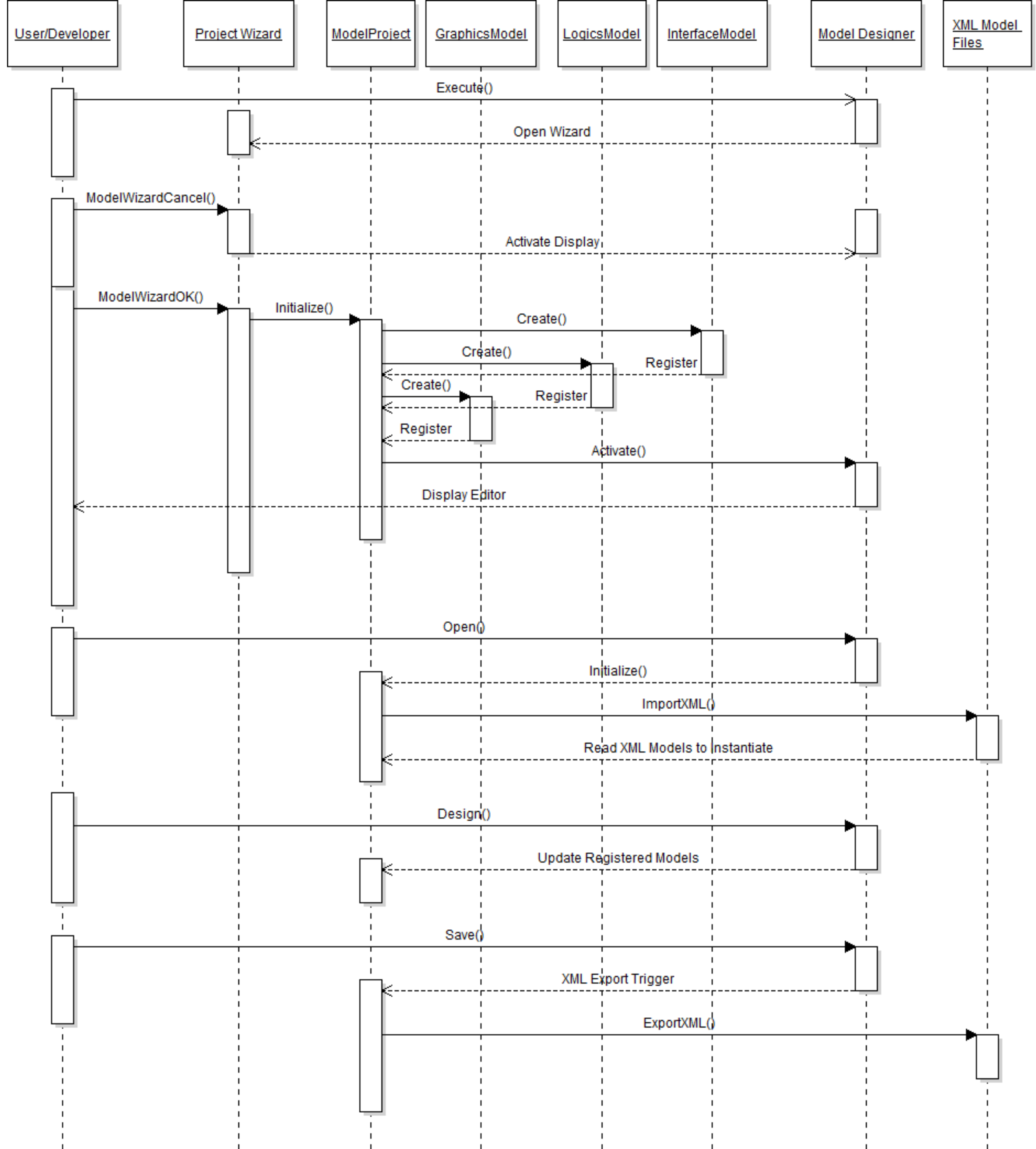
1) Geliştirici, model editöründe bir proje oluşturur ve bu projeye bir veya birden fazla grafik, mantık ve/veya arayüz tasarımını dâhil edebilir.
2) Geliştirici, oluşturduğu projenin kaydını gerçekleştirdiğinde tanımlanmış meta modelleme kütüphanesine uygun XML dosyalarını üretir.
3) Geliştirici, daha önceden kaydedilmiş olan XML dosyaların olduğu projeyi model editörüne yükleyebilir ve modellerin tasarımını gösterebilir.

Ana kullanım senaryosu Şekil 4.11’de gösterilmiştir.



Şekil 4.11 Model Editörü Kullanım Senaryosu

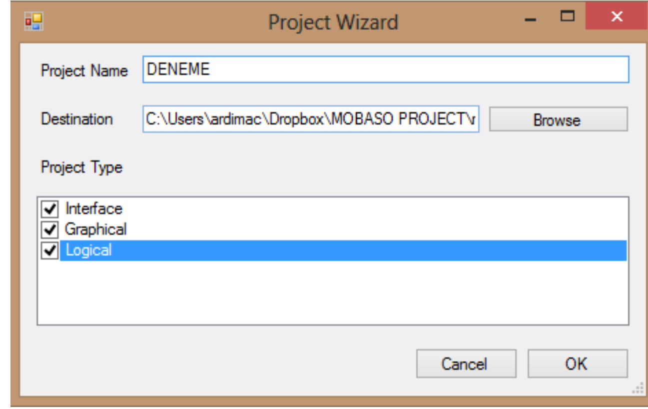
Tasarım editörü, model alt yapısını etkili bir şekilde kullanmaktadır. Kullanım senaryosunda, sınıfların kullanımı ve birbirleri ile olan etkileşimleri Şekil 4.12'de Sıralı şemada gösterilmiştir.



Şekil 4.12 Kullanım Sıralı Şeması

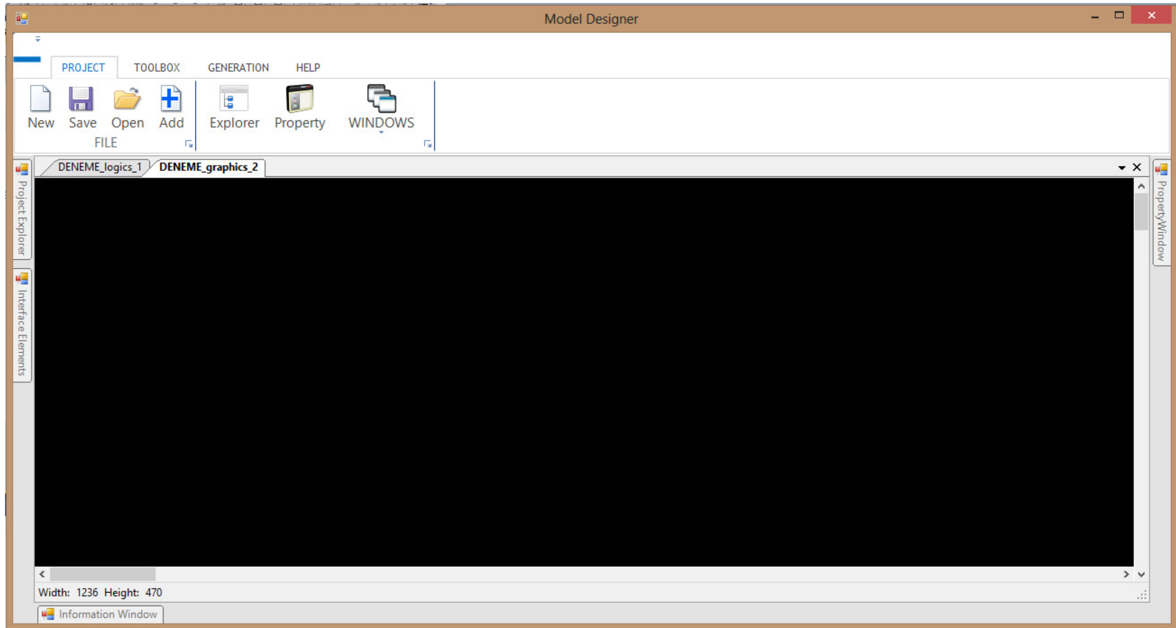
4.2.2. Editör Arayüz Tasarımı

Editör tasarım prensiplerinden ortaya çıkan kullanım senaryolarına göre belirlenen ekranların ilki olarak Proje Sihirbazı Arayüzü Şekil 4.13'de gösterilmiştir.



Şekil 4.13 Proje Sihirbaz Arayüzü

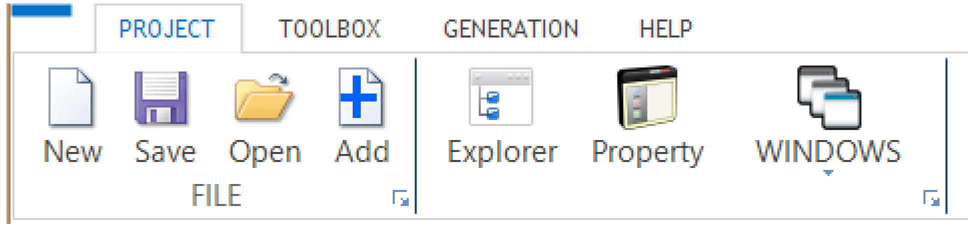
Proje Sihirbaz Arayüzü penceresi, geliştirilecek olan tasarımın hangi tipte (grafiksel, mantıksal) tasarım içereceği, proje adını ve yolu ile ilgili bilgileri kullanıcıdan alarak Model Proje sınıfının ilklendirilmesi kısmında rol almaktadır.



Şekil 4.14 Ana Tasarım Ekranı

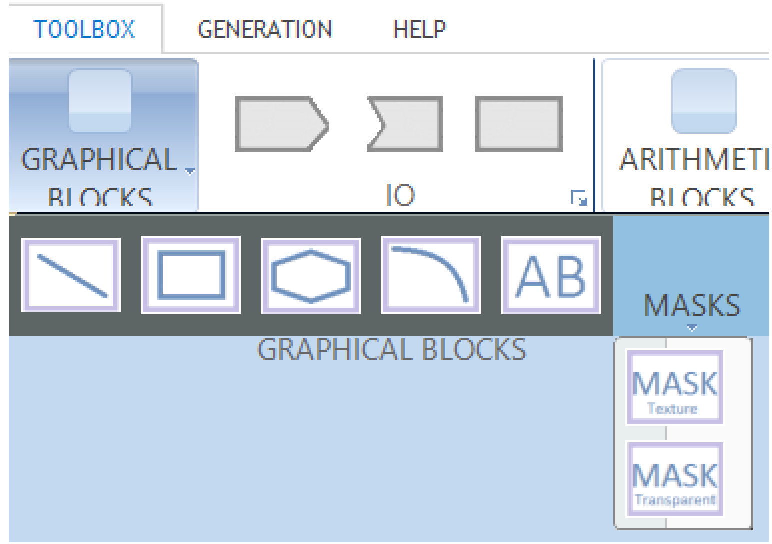
Proje Sihirbaz ekranı geçildikten sonra, Şekil 4.14'te olduğu gibi Ana Tasarım Penceresi açılmaktadır. Bu pencere belirlenmiş olan tüm pencereleri kapsayan, yönlendiren ve kullanıcının tanımlanmış isteklerine karşılık veren bir tasarım editörüdür.

Ana tasarım ekranı, proje paneli, tasarım çizim ekranları, arayüz tanımlama ekranları, nesne özellikler ekranı ve proje ekranından oluşmaktadır.

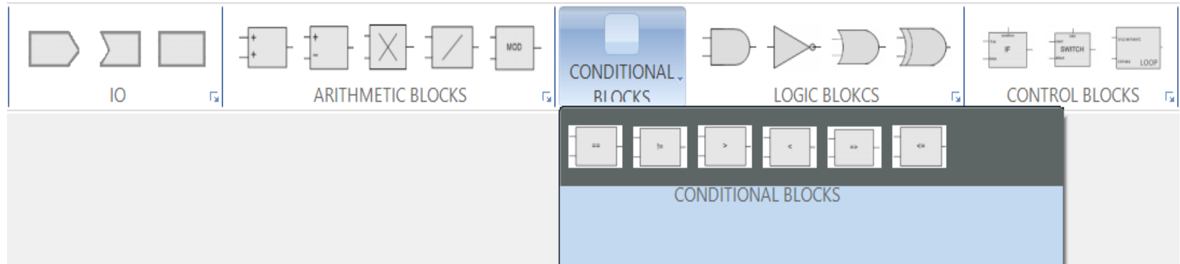


Şekil 4.15 Proje Paneli

Proje paneli, Şekil 4.14'teki gibi tasarlanmıştır. Bu panel üç ana grubun alt fonksiyonlarını taşıyan alt panellerden oluşmaktadır. Alt panellerde proje, modellerde kullanılan birincil blokların listelendiği araç çubuğu (Şekil 4.16 ve Şekil 4.17) ve kod üretimi ile ilgili fonksiyonlardan oluşan panellerdir.

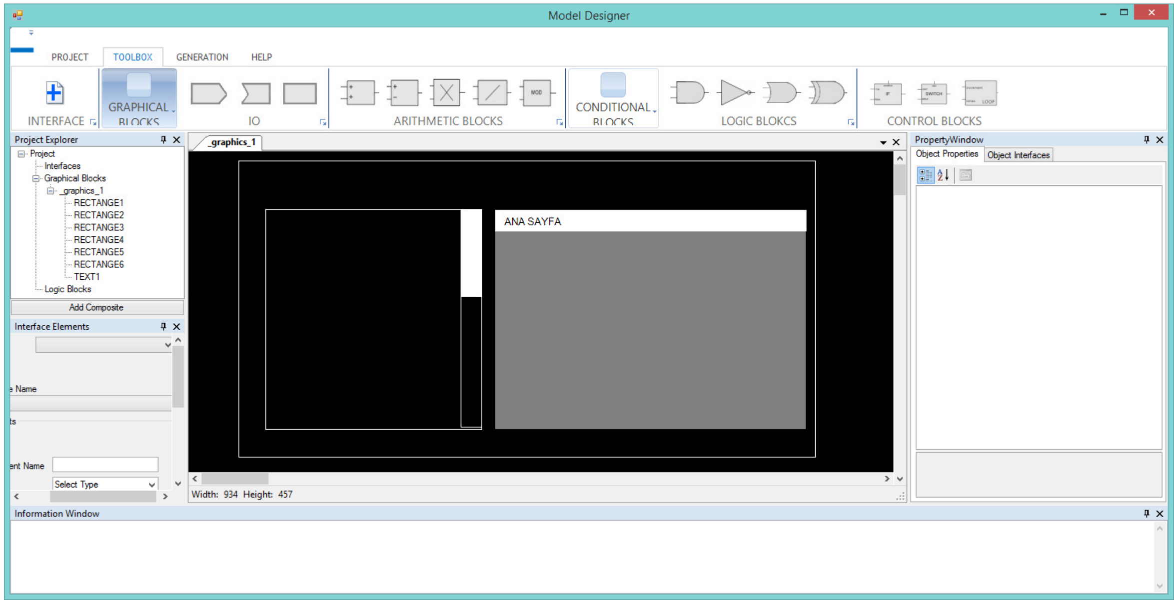


Şekil 4.16 Grafik Blokları



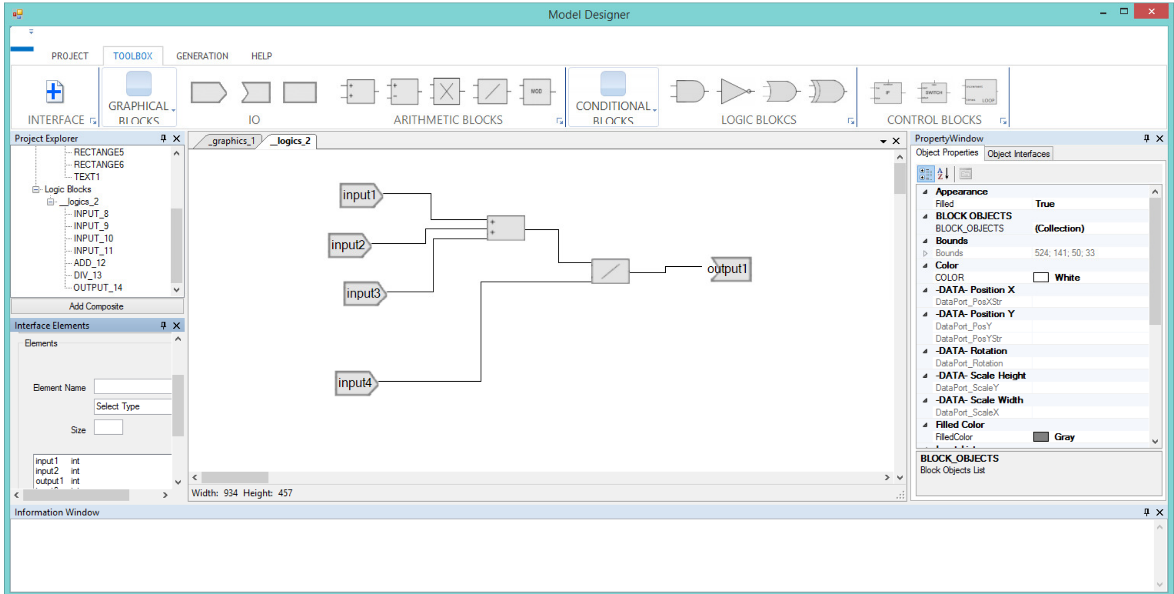
Şekil 4.17 Mantık Blokları

Grafik modelleri olarak belirlenmiş tasarım çizim penceresinde sadece grafik blokları kullanılmaktadır. Şekil 4.18'de Grafik ile ilgili tasarım örneği gösterilmiştir.



Şekil 4.18 Grafik Tasarım Örneği

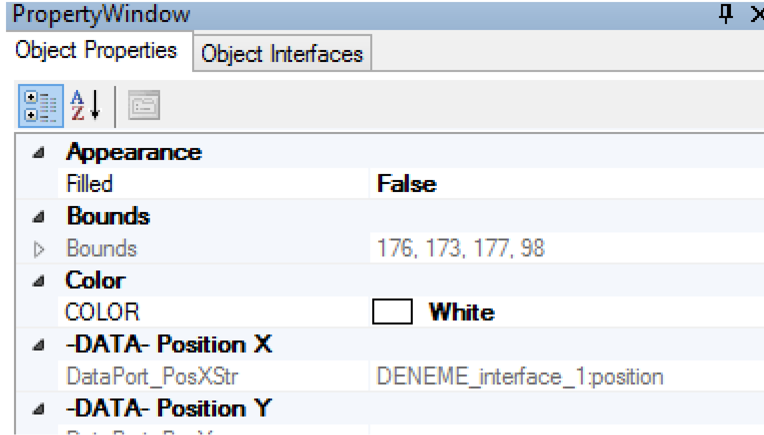
Diğer yandan Mantıksal model olarak belirlenmiş tasarım çizim penceresinde ise sadece mantık blokları kullanılabilmektedir. Şekil 4.19'da mantıksal tasarım örneği gösterilmiştir.



Şekil 4.19 Mantıksal Tasarım Örneği

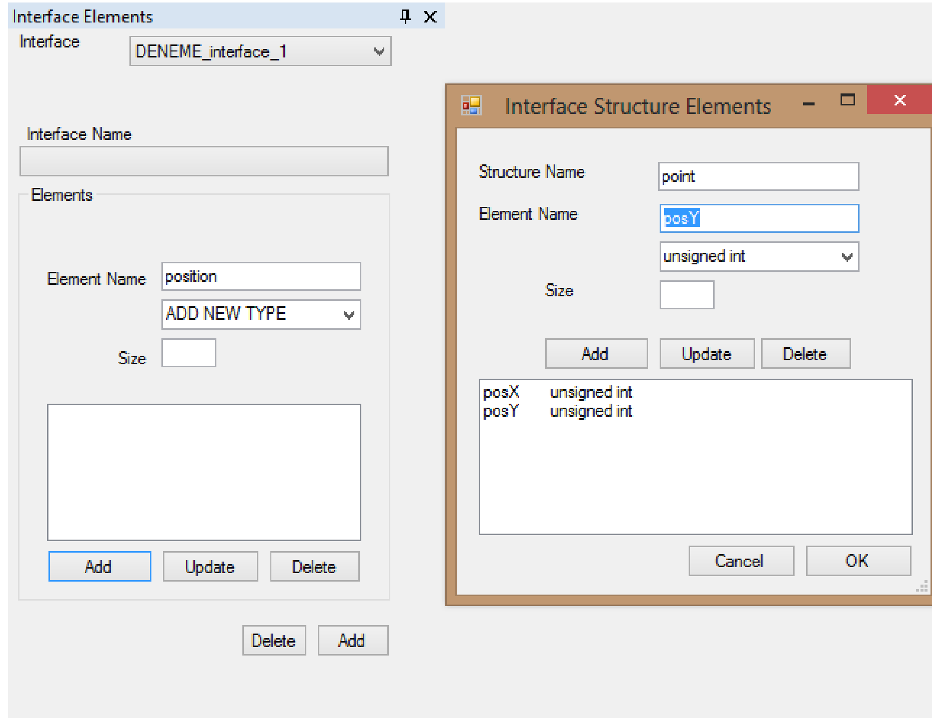
Tasarım editörü üzerinde çizimsel ifade edilen tasarım nesnesinin aktif olduğunda özelliklerini (pozisyon, renk, isim vb.) gösterebilmek, değişikliklerini yapabilmek ve

herhangi bir arayüzle bağlantısını kurabilmek için nesne özellikler ekranı, Şekil 4.20'deki gibi tasarlanmıştır.



Şekil 4.20 Nesne Özellikleri Ekranı

Arayüz tanımlama ekranında, tasarlanacak olan birincil blokların veriyolu bağlantılarını tanımlanır. Bu ekranda tanımlanmış olan değişkenler grafik ve mantıksal modeller arasında kullanılabilmektedir. Şekil 4.21'de arayüzler için önerilen tasarım görülmektedir.

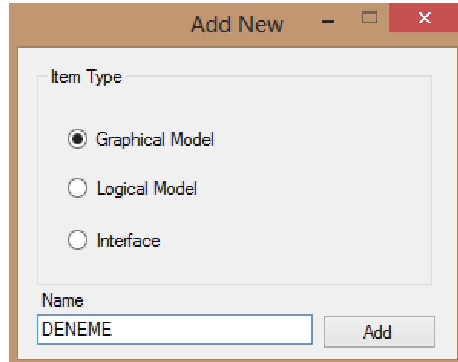


Şekil 4.21 Arayüz Tanımlama Ekranı

Tanımlanan verilerle daha sonradan tanımlanmış birincil blokun X eksenindeki pozisyonuna karşılık gelen bağlantı noktası ile eşleştirilme örneği Şekil 4.25'te verilmiştir. Tüm nesneler için ortak olan özellikler "Common" panelinde, nesneye özgü olan diğer özel bağlantı noktaları "Special" panelinde tanımlanmaktadır. Örneğin seçili nesne bir metin bloku için, Özel bağlantı noktasınta "text" bağlantı noktası görülecektir.

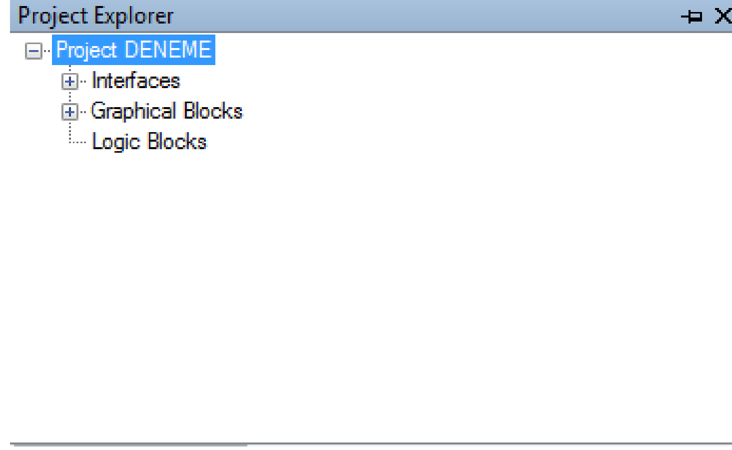
Şekil 4.22 Grafik Tasarımı ve Arayüz Ataması

Tasarım editörü, aynı anda birden fazla tasarım penceresi sunmaktadır. Başlangıçta gelen modellere ek olarak daha sonradan, yeni modellerin eklenebileceği pencere Şekil 4.23'te gösterilmiştir.



Şekil 4.23 Yeni Model Ekleme

Eklenen yeni tüm modellerin takip edilebildiği ve seçilebildiği proje yöneticisi ekranı Şekil 4.24'te gösterilmiştir. Bu ekranda gösterilen modeller tiplerine göre kendi kategorisi içerisinde yer alır. Kod üretim aşamasında her model ayrı bir C kaynak dosyası (*.c), Arayüzler C başlık dosyası (*.h) olarak oluşur.



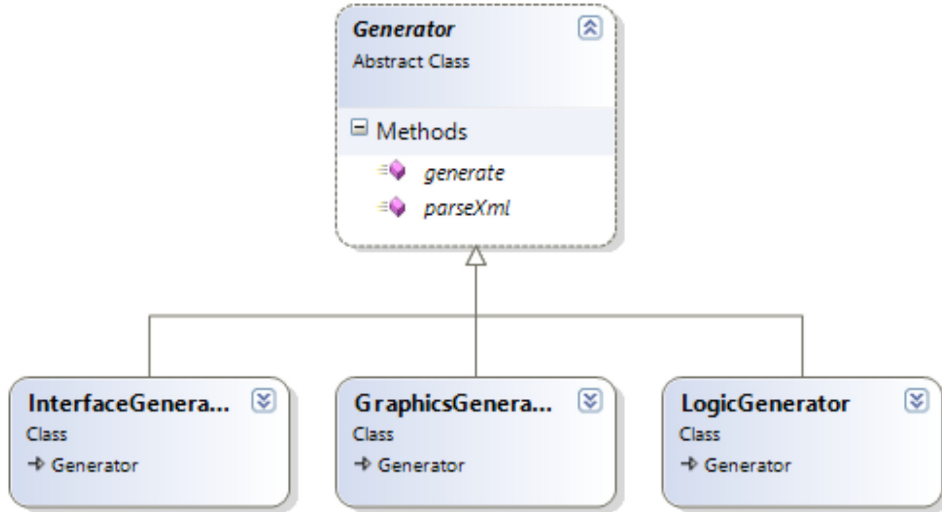
Şekil 4.24 Proje Penceresi

4.3. Kod Üreteci

Kod Üreteci tasarlanmış olan modellerden belirlenmiş olan şablona göre kodun üretilmesini sağlayan bir uygulamadır.

Kod üreteci harici geliştirilmiş bir uygulama olup tasarım editöründen bağımsızdır. Bu yaklaşımın çeşitli faydaları mevcuttur. Bunlardan ilki, kodun üretilmesi tasarım editörü olmadan sadece XML tanım dosyaları ile gerçekleştirilir. Bunun yanında üretecin bağımsız olmasıyla birlikte herhangi bir başka geliştirme ortamı içerisine entegre edilebilir. Böylelikle manuel programlama ve model tabanlı geliştirmenin birlikte kullanımına olanak sağlanır. Manuel programlama genellikle modellerin girdi ve çıktı arayüzlerinin harici arayüzlerle olan kısımlarının eşleştirilmesinde kullanılır. Son olarak, kod üretecinin tasarım editöründen bağımsızlığı geliştirme ortamının hafıza alanının kullanılması ve performansının belirli ölçüler altında kalması sağlanır.

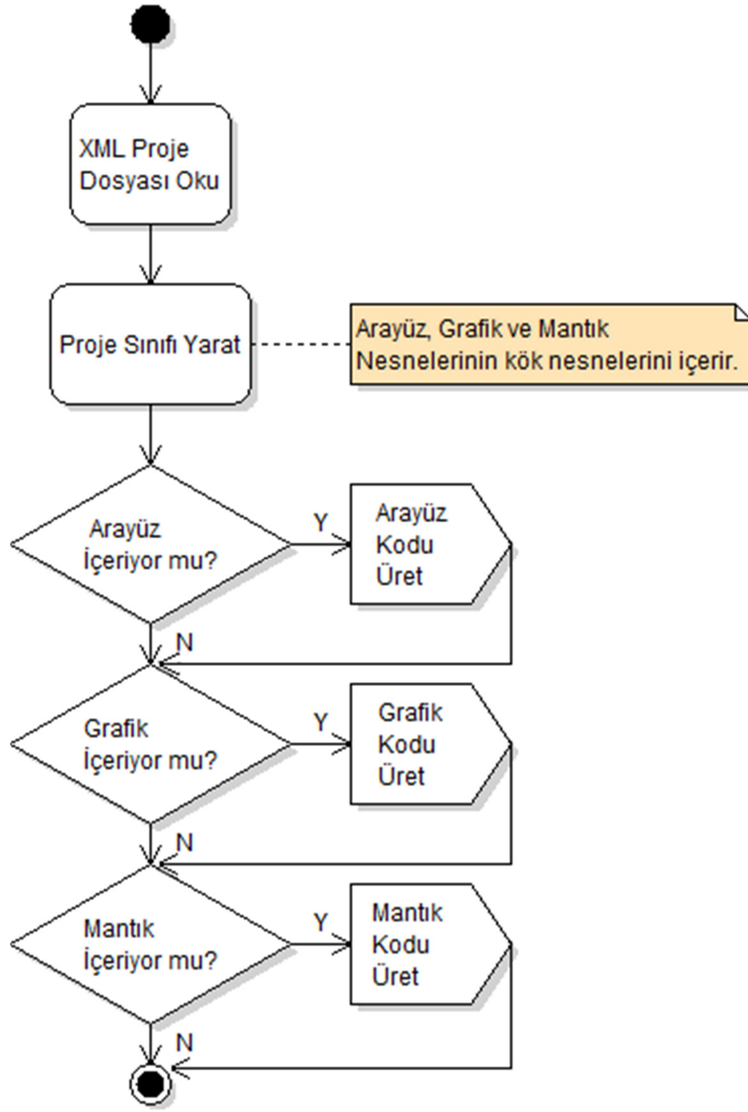
Kod üreteci uygulaması, grafik ve mantıksal alanlarına özgü modeller ve ayrıca bu modellerin birbirleri ile ilgili arayüzleri ile uyumlu olarak çalışmaktadır. Kod üreteci Arayüz, Grafik ve Mantık üreteçleri olmak üzere üç ana sınıftan oluşmaktadır. Tüm sınıflar soyut sınıf olan üreteç sınıfından türetilmişlerdir. Soyut üreteç sınıfında XML model dosyasını tarama ve üretme olarak iki ana fonksiyon mevcuttur. Sınıf şemaları Şekil 4.25'te gösterilmiştir.



Şekil 4.25 Kod Üreteci Sınıf Şeması

4.3.1. Kod Üretim Algoritmaları

Kod üreteci bileşeni işlemlerine proje dosyasını okuması ile başlar ve sonrasında okunan değerler ile model proje sınıfını yaratarak kod üretimi için kullanır. Model projesi sınıfı arayüz, grafik ve mantıksal kök nesneleri içermektedir. Ana işleyiş Şekil 4.28'de belirtilmiştir. Her bir model tipi için kod üretimi Şekil 4.26'da gösterildiği gibi kod üreteci soyut sınıfından türetilmiş ayrı birer görev parçalarıdır.



Şekil 4.26 Kod Üretimi ana Akış Şeması

Kod üretimde kullanılan şablon dosyaları temelde C programlama diline özgü geliştirilmiştir. Uygulamada StringTemplate [23] kütüphanesinden faydalanılmıştır. StringTemplate kütüphanesi, C# programlama dili ile kullanılarak şablon dosyalarındaki özel olarak işaretlenmiş alanların model tasarımına özgü bir şekilde uyarlanır ve daha sonrasında bir araya getirilerek kaynak kod gerçekleştirilir. Örneğin arayüz model tipi C programlama dilindeki "header" dosyasına karşılık gelmektedir. Şekil 4.27'de arayüz için geliştirilmiş basit bir şablon dosyası verilmiştir. Bu dosya içerisinde özel olarak işaretlenmiş veri isimleri *typeT, varName* ve *varType* olarak gösterilmiştir.

```
InterfaceTemplate.st
        $if(typeT)$
        typedef $varName$ {
        $elements:{it|$it. varType$
        $it.varName$};separator=";\n\r"$;
        };
        $else$
        $varType$ $varName$;
        $endif$
```

Şekil 4.27 Arayüz Şablon Örneği

Şekil 4.27'deki InterfaceTemplate.st dosyasının geri kalan kısımlarında kullanılmış olan yazı düzeni StringTemplate kütüphanesinin işleyebilmesi açısından, kütüphanenin işleyebileceği belirli bir biçime uygun bir şekilde yazılmıştır. Kütüphanenin işleyebildiği özel alanlar '$' sembolü arasında belirtilen veri tanımı ya da basit kod parçalarıdır. Bu dosyada ilk olarak belirlenen tipin birincil bir tiplemi (char, int, float, double vb.) yoksa karmaşık tiplemi (structure) olduğuna göre bir kıyaslama yapılır. *typeT* değişken adı üretilecek olan arayüzün birincilmi yoksa karmaşık tip mi olduğunun bilgisini taşır. Bu karara göre karmaşık tip tanımı ya da birincil tip tanımı gerçekleştirilir. Özel sembol alanı haricinde yazılmış olan metinler birebir çıktı dosyasında yer alır (";", "typedef" gibi).

Grafik modelleri de benzer bir yaklaşımla üretilmekle beraber, şablonlarda daha fazla belirlenmiş özel alanlar ve OpenGL [36] kütüphanesi fonksiyon çağrımlarından oluşmaktadır. Grafik modelindeki kök nesne olan Composite nesnesi C programlama dilindeki fonksiyon olarak ifade edilir. Grafik Model kısmında tanıtılmış olan diğer alt grafik nesneleri OpenGL fonksiyon çağrıları ile tanımlanmaktadır. Grafik modelinin kod üretimi ile ilgili algoritma aşağıda verilmiştir.

Algoritma: Grafik Model Kod Üretimi

```
01 function GraphicsGenerator (Composite item) as Composite
02      n = lenght(item) // Get Elements Count
03      for each objects in item.Childs // Check If any Interface Used
```

```
04            if item contains Interface
05                    substitudeTemplate(ReplaceUnique(IncludeHeader))
06            end if
08      next item
09      // Define function name of Composite
10      substituteTemplate(FunctionDefinition)
11      // For each object in Composite
12      for i = 0 to n-1 do
13            if (item.Child[i] is Composite)
14                    // If Sub Composite item is Referenced type
15                    // Call only its prototype
16                    if (item.Child[i].referencedComposite)
17                            substituteTemplate(FunctionCall)
18                    else
19                    // Otherwise, Recurse it
20                            return GraphicGenerator(item.Child[i])
21                    end if
22             else if (item.Child[i] is State)
23                    // Use Interface for State Change
24                    substituteTemplate(StateInterface)
25                    // Iterate for each composite in state object
26                    for each composite as Composite in item.Child[i].States
27                            // If Sub Composite item is Referenced type
28                            // Call only its prototype
```

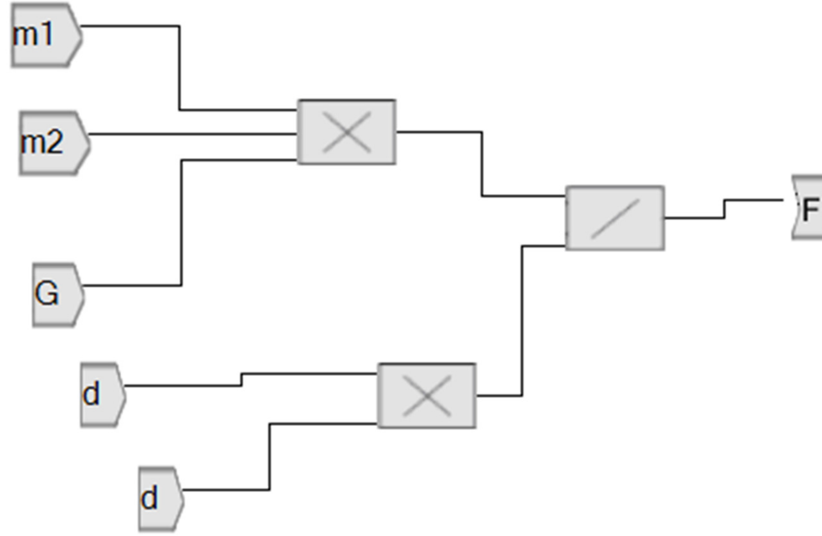
```
29                  if (composite.referencedComposite)
30                          substituteTemplate(FunctionCall)
31                  else
32                          // Otherwise, Recurse it
33                          returnGraphicGenerator(composite)
34                  end if
35          next CompositeItem
36      else if (item.Child[i] is Mask)
37          substituteTemplate(MaskType)
38          // Iterate for each object in Mask object
39          for each object in item.Child[i].Childs
40                  if (object is Composite)
41                          // If Sub Composite item is Referenced type
42                          // Call only its prototype
43                          if (object.referencedComposite)
44                                  substituteTemplate(FunctionCall)
45                          else
46                                  // Otherwise, Recurse it
47                                  return GraphicGenerator(object)
48                          end if
49                  else
50                          substituteTemplate(Primitive)
51                  end if
52          next object
```

```
53              else
54                      substituteTemplate(Primitive)
55              end if
56      next i
57      return item
58 end procedure
```

Algoritmada, "*SubstituteTemplate*" ismi ile gösterilmiş olan sembolik fonksiyonda herhangi bir grafik nesnesinin, belirli olan şablon dosyasının çağrılmasında etkili olmasında rol alan bir fonksiyondur. Örneğin nesne listesi sırasıyla dolaşıldığında karşılaşılan herhangi bir "line" nesnesi için gerekli olan şablon dosyasının çağrılmasını sağlar. Benzer şekilde *"FunctionDefinition", "FunctionCall", "StateInterface", "MaskType"* gibi özel nesnelere özgü kullanımlar da bulunmaktadır.

Composite Nesnesi tüm nesneleri barındırabilen bir nesne olduğu için, alt grafik nesnelerinde tanımlanmış bu tip nesnelerinde bütünden teke doğru kodunun üretilmesi gerekmektedir. Grafik Algoritması, bu tip taşıyıcı nesneleri kullanabilmek için kendini yineleyen bir algoritma olarak tanımlanmıştır.

Mantık kod üretme algoritması temelde ağaç üzerinde yol bulma algoritmasına dayalıdır. Bu yol herbir model çıktısı için bir ağaç modeli oluşturularak, model girdisine ulaşıncaya kadar gezdiği blokların tespitidir. Bu blokların tespiti ile birlikte blokların girdi ve çıktılarının sırasıyla hangi işlemlerden geçeceği blok tipine göre karar verilir. Örneğin, Şekil 4.29'da kütle çekim formulunun ($\mathrm{F} = G\frac{m1m2}{d^2}$) , tasarım editöründeki mantıksal model karşılığı verilmiştir. Tasarımda F model çıktısı, m1, m2, G ve d model girdileri, çarpım blokları ve bölme bloğu kullanılmıştır. Kod üretim algoritması, model çıktısı olarak kullanılan F'i oluşturan yolları bulmak için geriye doğru bağlantı yollarını takip eder. Bağlantı yollarını bulabilmek için Model tanımında daha önceden bahsedilmiş olan "Edge" sınıflarını kullanır. "Egde" sınıfında bağlantı listeleri tutulmaktadır.



Şekil 4.28 Kütle Çekim Formülü Mantıksal tasarım

Şekil 4.28'deki tasarımda model çıktısı olan F ile bağlanan blok arasındaki yol kullanılarak bölme bloğuna ulaşılır. Bölme bloğunun herbir girdileri için öncelikle ilk çarpma bloğununa, sonrasında çarpma bloğunun girdileri olan m1, m2 ve G'ye ulaşıldığında ilk yol tespit edilmiş olur. Diğer yandan bölme bloğunun ikinci girdisini oluşturan ikinci çarpma bloğunun girdileri olan d'ye ulaşıldığında ikinci yol ortaya çıkmış olacaktır.

Belirlenen yollar tespit edilirken, kod üretmek için yığın yapısı kullanılır. Yol bulma işleminde tespit edilen her blok arayüzleri ile birlikte yığına eklenir. Model tarama tamamlanınca dolu olan yığın kod üretimi için hazırdır. Kod üretim algoritması iki aşamada tamamlanmaktadır. İlk aşamada kök nesne olan "Activity" nesnesinin çıktı listesindeki her eleman için bağlı olan bloklar öz yinelemeli olan fonksiyona aktarılır. Öz yinelemeli fonksiyon "Input" girdi tipindeki bloğa gelinceye kadar yığını doldurur. Son olarak, bir yol tamamlandığında yani input nesnesine ulaşıldığında, dolu olan yığın kümesi kullanılarak kod üretim aşamaları gerçekleştirilir. Şekil 4.29'daki algoritma, mantık kod üretim algoritmasının ilk kısmını göstermektedir.

```
01 // Main pseudo code of Logic generator.
02 procedure LogicsGenerator(Activity rootItem)
03     set n = rootItem.Outputs.Count
04     for i = 0 to n-1 // Find All output paths
05         set OutputActiviyu = rootItem.Outputs(i)
06         // Find the edge connected to the output activity
07         set EdgeItem = EdgeList.FindEndNode(rootItem.OutputActivity)
08         set InputActivity = EdgeList.StartNode(EdgeItem)
09         // Get the Local Interface of current object
10         set OutputInterface = rootItem.Outputs(i).Input
11         // Get the connected Blocks output
12         set InputInterface =
SubLogicsGenerator(InputActivity).OutputInterface
13         set substituteEqualParam = {InputInterface, OutputInterface}
14         substituteTemplate(substituteEqualParam)
15     next i
16 end procedure
```

Şekil 4.29 Mantıksal Model Ana Kod Üretimi

Mantıksal algoritmanın temeldeki görevi çıktıyı girdiye eşitlemektir. Algoritmanın ilk kısmında belirtilen sembolik "*substituteTemplate*" fonksiyonunda verilen eşitlik parametresinde çıktıyı alt özyinelemeli fonksiyon olan "*SubLogicsGenerator"* fonksiyonunun çıktısını kullanarak işlemde olan model çıktısına eşitler. Buradaki önemli husus modelin tüm çıktıları için girdiye uzanan yolu tespit etmektir.

```
01 // Recursive Function - traverse path
02 function SubLogicsGenerator (Activity rootItem) as Activity
03     set TypeOfActivity = rootitem.Type
04     // The Input of Activity i s reached
05     if (TypeOfActivity.Equals ("Input"))
06             return rootitem
07     else
08             set n = rootitem.InputList.count
09             for i = 0 to n-1 // Find Inputs
10                     set Inputinterface = rootitem.InputList (i)
11                     // Find the edge connected to the this activity.
12                     set Edgeitem = EdgeList.FindEndinterface (Inputinterface)
13                     set InputActivity = EdgeList.StartNode(Edgeitem)
14                     // Continue traverse and collect the interfaces.
15                     set ChildOutputinterface =
16                             SubLogicsGenerator (InputActivity).outputinterface
17                     set substituteEqualParam =
18                             {Inputinterface, ChildOutputinterface }
19                     substituteTemplate(substituteEqualParam)
20             end for
21             set substituteOperateParam =
22                             {rootitem.Type, rootitem.InputList}
23             substituteTemplate (substituteOperateParam)
24 end if
```

Şekil 4.30 Mantıksal Model Alt Blok Kod Üretimi

Şekil 4.30'daki Alt Blok kod üretimi algoritmasında 19. ve 23. satırlarda geçen "*substituteTemplate"* fonksiyonu 19. satırda eşitlik, 23. satırda ise blok tipine göre çağrılmaktadır.

5. UYGULAMA

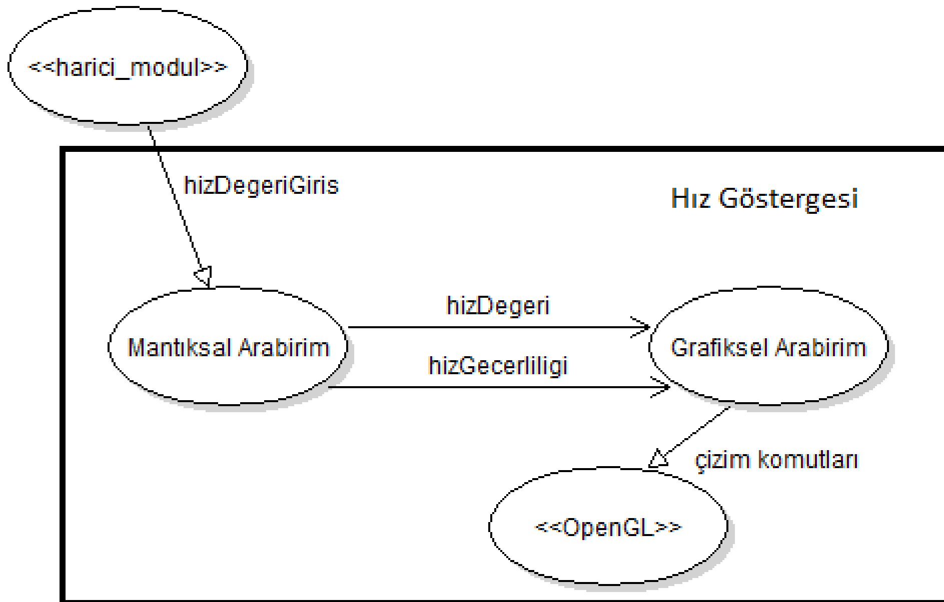
5.1. Giriş

Geliştirilen MOBASO ana çatısı ile herhangi bir arayüz ya da kontrol içeren bir uygulama yapmak mümkündür. Bu kısımda geliştirilen platform ile birlikte örnek bir hız göstergesi tasarımı geliştirilecektir.

5.2. Ön Tasarım

Geliştirilecek olan hız göstergesi ön tasarımı için aşağıdaki belirlenmiş olan gereksinimleri temel alacaktır.

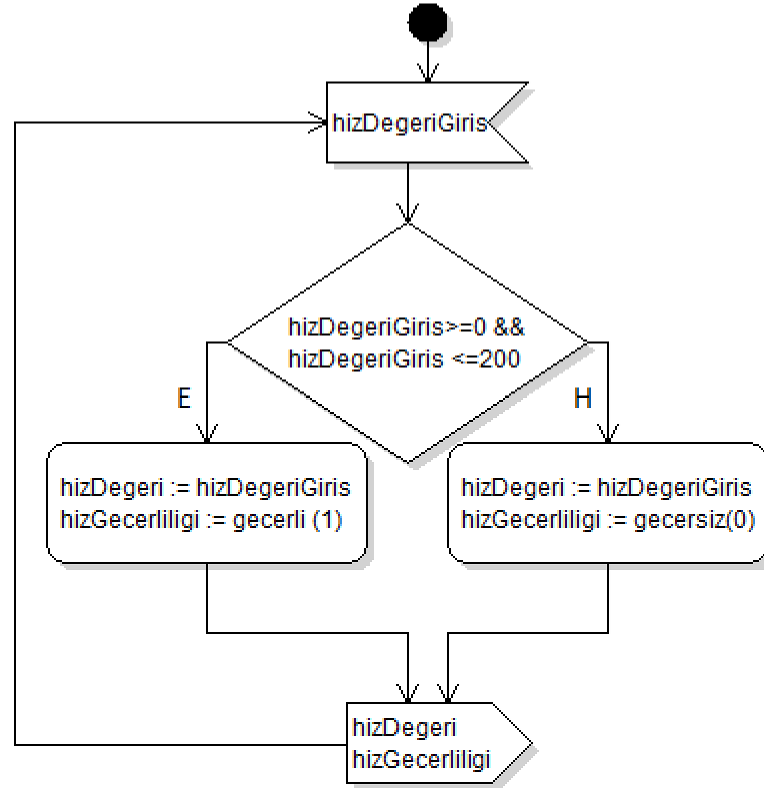
1) Hız göstergesi geçerliliği *hizGecerliligi* tanımına göre geçerli(1) ise, hız göstergesi *hizDegeri*'ni gösterecektir.
2) Hız göstergesi geçerliliği *hizGecerliligi* tanımına göre geçersiz(0) ise, hız göstergesi üç adet kırmızı '*X' gösterecektir.*
3) Hız gösterge değeri olan *hizDegeri* [0, 200] aralığında ise hız geçerliliği *hizGecerliligi* geçerli(1) olacaktır.



Şekil 5.1 Hız Göstergesi Veri Akış Diyagramı

MOBASO ana çatısında geliştireceğimiz bu modelin, üç modelleme tipini de *Arayüz, grafiksel* ve *mantıksal* modellemeyi içermektedir. Geliştirilen bu modelin başka bir model içerisinde kullanılacağını, sadece *hizDegeri*'nin bu modele sağlandığı

varsayılmaktadır. Ayrıca, *hizDegeri* ve *hizGecerliligi* mantıksal arabirim ve grafiksel arabirim arasında paylaşılan arayüz bilgisi olarak ortaya çıkmaktadır. Gereksinimlere göre ortaya çıkan veri akış diyagramı Şekil 5.1'de gösterilmiştir. Öncelikli olarak dışarıdan gelen verinin önce işlenip daha sonra gösterilecek olan bilginin oluşturulması sağlanmaktadır. Buna göre mantıksal arabirim verinin alınıp işlenmesi ile ilgili olan kısmını, grafiksel arabirim de mantıksal kısımdan alınan verilerin gösterilmesi ile ilgili kısımlarını üstlenmişlerdir.

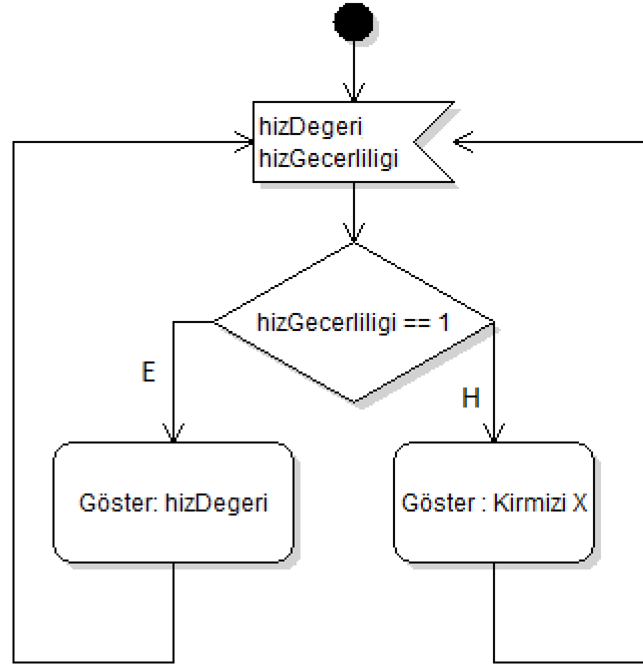


Şekil 5.2 Hız Göstergesi – Mantıksal Arabirim Akış Diyagramı

Gereksinimlere göre oluşturulacak olan mantıksal arabirim algoritmasının akış diyagramı Şekil 5.2'deki gibi olacaktır. Mantıksal arabirim sadece birinci ve üçüncü gereksinimleri karşılamaktadır. Birinci gereksinime göre gösterilecek olan verinin grafiksel arabirime ulaştırılması ile ilgili olandır. Üçüncü gereksinime, birinci gereksinimde de kullanılması gereken verinin geçerliliği ile ilgili olan gereksinimdir. Mantıksal arabirimin gereksinimin işleyiş sırası önce üçüncü daha sonra birinci gereksinim olarak belirlenmiş olmaktadır.

Grafiksel arabirim gösterim ile ilgili olan gereksinimlerden sorumludur. Buna göre sadece birinci ve ikinci gereksinimleri karşılamaktadır. Sağlanan veri geçerliliğine göre geçerli bilginin veya herhangi bir grafiksel gösterim gerçekleştirilecektir.

Grafiksel arabirim algoritmasının akış diyagramı Şekil 5.3'te verilmiştir. Oluşturulan veri, mantıksal ve grafiksel akış diyagramlarına göre tüm gereksinimler sağlanmış durumdadır.



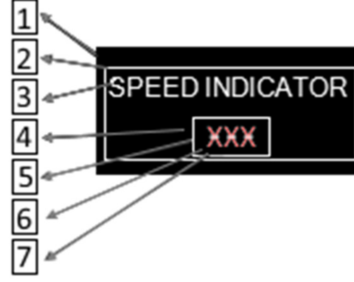
Şekil 5.3 Hız Göstergesi – Grafiksel Arabirim Akış Diyagramı

Verilen grafiksel ve mantıksal akış diyagramları geleneksel yöntemlerle yazılım geliştirme çalışmalarında gerçekleştirilen diyagramlardır. Bir sonraki kısımda MOBASO aracı üzerinden verilmiş olan diyagramları oluşturmaya gerek kalmadan, bu diyagramları aracın kendi diline göre tasarımı gerçekleştirip kodu otomatik olarak üretilecektir.

5.3. Örnek Tasarım Çalışması

Ön tasarımdaki tanımlanmış olan gereksinimlerden yola çıkarak gerçekleştirilecek olan tasarımın grafiksel ve mantıksal modelleri MOBASO aracı ile gerçekleştirilecektir.

Grafik tasarımı ile ilgili olarak gereksinimlerde, hız göstergesinin geçerli değerinin ve geçersiz değerinin nasıl gösterebileceğiyle ilgili tanımlamalar mevcuttur. Gereksinimlerde grafik ile ilgili herhangi bir özel bir tanım olmadığından Şekil 5.4'teki gibi tasarımı, MOBASO aracı üzerinden gerçekleştirilmiştir.



Şekil 5.4 Hız Göstergesi - Örnek Grafiksel Tasarım

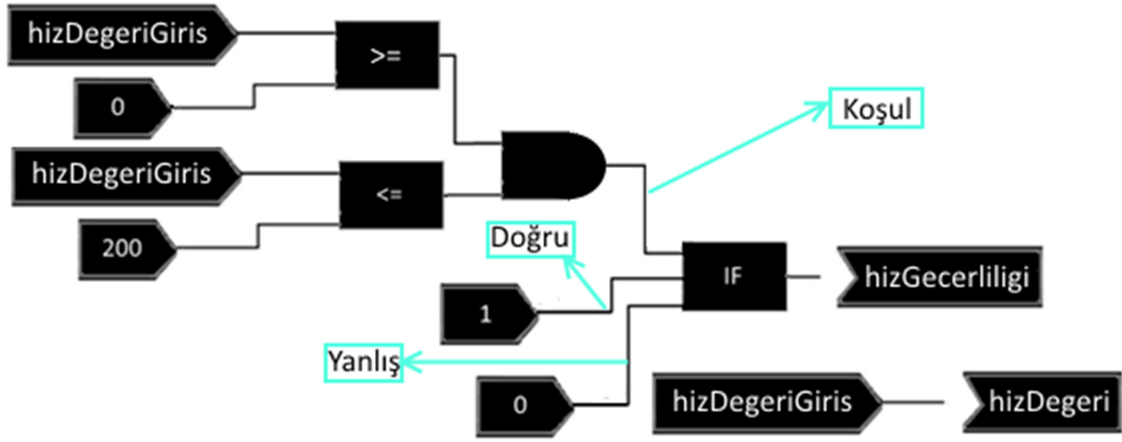
Tasarım yedi elemandan oluşmaktadır. Şekil 5.4'te bu nesneler işaret edilmiştir. Belirlenen nesnelerin oluşturduğu seviyeler ve özellikleri Çizelge 5.1'de verilmiştir.

Çizelge 5.1 Örnek Grafiksel Tasarım Nesne Çizelgesi

Numara	Seviye	Nesne Tipi	İsim	Bilgi
1	1	Composite	speedInd_graph	**-**
2	2	Rectangle	rect_1	**-**
3	2	Text	text_1	Gösterilen Metin: **"SPEED INDICATOR"**
4	2	State	state_1	State Sayısı: **2** Arayüz: ***hizGecerliligi***
5	3	Rectangle	rect_2	-
6	3	State 0	state_1_0	Birinci state nesnesi Gösterilen Metin: **"XXX"**
	4	Text	txtInvalid	
7	3	State 1	state_1_1	İkinci state nesnesi, Gösterilen Metin: "**---**" Arayüz: ***hizDegeri***
	4	Text	txtValid	

Çizelge 5.1’de tanımlanmış olan *speedInd_graph* ayrıca grafik model ismi olarak tanımlanmıştır. *State* nesnesi çalışma anında bağlanan Arayüzün değerine göre içerdiği alt çizim ekranlarından sadece bir tanesini gösterebildiği için Şekil 5.4’te tasarlanmış olan grafiksel modelde *state_1* nesnesinin metinlerinin üst üste geldiği görülmektedir.

Grafik modeline bağlı olan *hizGecerliligi* ve *hizDegeri* arayüzleri Şekil 5.5’te tasarlanmış olan mantıksal modelin oluşturduğu değerlerdir.



Şekil 5.5 Hız Göstergesi - Örnek Mantıksal Tasarım

Şekil 5.5’teki Örnek mantıksal tasarım, *speedInd_logic* ile isimlendirilmiştir ve bu isim kök *activity* nesnesininde aynı zamanda adıdır. Kök nesne dört tane blok içermektedir. Bu bloklar; *greater_than*, *less_than, and* ve *if* bloklarıdır. Ayrıca üç tane *input* arayüz bağlantısı ve dört tanede yerel *input* bağlantısı kullanılmıştır. Tüm bu bloklar ve arayüzleri birbirine bağlayan *edge* tipindeki bağlantı çizgileri mevcuttur.

6. SONUÇLAR VE DEĞERLENDİRME

Model Tabanlı Yazılım geliştirmedeki amaç hedef gereksinimler ve tasarım arasındaki tutarlılığı devam ettirmektir. Karmaşık sistemlerde ise bu durum daha büyük bir önem taşır. Gereksinimlerdeki değişikliklerin yönetimi piyasada sunulan bazı araçlarla kolay bir şekilde yapılabilmektedir, ancak bu araçlar tasarım aşaması için aynı kolaylığı sağlamamaktadır. Kod gözden geçirme safhasında yapılan bu gereksinim değişikliklerinin, koddaki etkisini bulmak zor olabilmektedir. Ayrıca, bu değişen gereksinimlerin doğru bir şekilde koda uyarlanıp uyarlanmadığını tespit etmek de oldukça zaman ve çaba gerektirmektedir. Programlama dillerinin seviyesi arttıkça, programlama dilinin gereksinimleri daha anlaşılır ifade etmesi de kolaylaşmaktadır. Bununla birlikte elle kodlamada, geliştiriciler birçok kodlama hatası ile karşı karşıya kalabilmektedirler. Endüstride kullanılan ve bu tip kod hatalarının önüne geçen birçok yazılım kodlama editörü olmasına rağmen hala kodlamadaki çeşitli hatalar yapılmaya devam etmektedir. Bu noktada, tüm bu sorunlara çözüm sunan model tabanlı yazılım geliştirme araçları ortaya çıkmaktadır. Bu tezde, yeni bir modelleme dili, bu dili kullanan model geliştirme editörü ve bu modellerden 4. Seviye programlama dillerine dönüşüm yapabilen bir kod üreteci geliştirilmiştir. Model tabanlı yazılım geliştirme aracımızda bulunan modelleme editöründe, görsel olarak tanımlanmış hem mantıksal blok diyagramları ve hemde grafiksel arabirim tasarımları kullanılarak kod üretimi yapmak, gömülü sistemler için elle kodlamada karşılaşılan sorunlara çözüm olarak gösterilebilir. Modelleme dili daha önceden de belirtildiği gibi kolay uygulanabilir bir dildir ve bu sayede gereksinimlere karşılık gelen izlenebilirlik daha doğru ve kolay bir şekilde sağlanmaktadır. İzlenebilirliğin net olması sayesinde gereksinimlerdeki herhangi bir değişikliğin tasarım tarafındaki etkileri belirgin olmakta ve bunun maliyet araştırmaları daha kolay yapılabilmektedir. Modeller belirlenen dil kuralları ile birlikte doğrulanarak belirlenmiş şablon dosyaları kullanılarak hedef programlama diline (C Dili) uygun kod üretimi sağlanmıştır.

Gelecekte, yaklaşımımıza ek olarak, modeller için test modelleme çerçevesi geliştirilerek modelleme editörü ile bütünleştirilebilir. Başlangıçta, tasarlanan modellerinin test ana çatısına uygun kodunun üretilmesi sağlanır. Bu sayede modeller arası alıp verilen arayüzlerin, model içi kullanılan giriş çıkışların dışa aktarımı sağlanarak, tasarlanmış yeni test ekranlarında bu değerler gözlemlenebilir.

Modellerin sonuçları ve farklı test senaryolarına göre olan davranışları bu ekranlarda tasarımın her aşamasında görülebildiği için mantıksal hataların erken safhalarda önüne geçilebilir. Tasarlamış olduğumuz modelleme ana çatısında bulunan birincil bloklarla, birçok kabiliyeti gerçekleştirebilen tasarımları oluşturmak mümkündür. Ancak, özel amaçlara yönelik, sık kullanılması gereken birincil bloklar ile tasarlanmış blok grubunun, yeni bir birincil blok olarak kullanılmasına ihtiyaç duyulabilir. Bu amaçla ikinci olarak, eklenti mekanizması bu tip birincil blokların tanımlanması, modelleme editörünün gelişmesi açısından uygun bir uygulama yöntemi olabilir. Üç temel parça; model kuralı, XML tanımları ve kod üretiminin yapılacağı şablon dosyalar yeni birincil blokların eklenmesinde yeterli olacaktır. Bu yapılacak iyileştirmelerle birlikte MOBASO ana çatısı modülerlik ve esneklik kazanmış olacaktır.

KAYNAKLAR

[1] Sommerville, I., *Software Engineering*, Addison Wesley, 9th Baskı. **2010**.

[2] Dooley, J., Software Development and Professional Practice. APRESS, **2011**.

[3] Pressman, R. S., *Software Engineering A Practitioner's Approach*, 7th ed. McGraw-Hill, **2010**.

[4] Graichen, C., D'Amato F., "Adding code generation to develop a simulation platform," *2011 IEEE Long Island Systems, Applications and Technology Conference*,1–6, Mayıs **2011**.

[5] Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G., "3 Metamodelling," in *Model-Based Engineering of Embedded Real-Time Systems*, Springer Berlin Heidelberg, 57–76, **2010**.

[6] Seidewitz, E., "What Models Mean," *IEEE Software*, cilt. 20, no. 5,26–32, **2003**.

[7] Stahl, T., Völter, M., *Model-Driven Software Development: Technology, Engineering, Management*, John Wiley & Sons, **2006**.

[8] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed. Prentice Hall Inc, **2001**.

[9] A. K. Jos Warmer, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, **2003**.

[10] Selic, B., Model-Driven Development of Real-Time Software Using OMG Standards, 1–2, **2003**.

[11] Kleppe, A., J. Warmer, Bast, W., *MDA Explained: The Model Driven Architecture™: Practice and Promise*. Addison Wesley, sayfa 192, **2003**.

[12] Kleidermacher, D., Kleidermacher, M., "Secure Embedded Software Development," *Embedded Systems Security*, 93–208, **2012**.

[13] Anonim, S. Suite®, Control and Logic Application Development, http://www.esterel-technologies.com/products/scade-suite/ (Erişim Tarihi: Ocak, **2015**).

[14] Anonim, S. Suite®, Display and HMI Development, http://www.esterel-technologies.com/products/scade-display/ (Erişim Tarihi: Ocak, **2015**).

[15] Anonim, Presagis, HMI VAPS QCG, http://www.presagis.com/files/product_brochures/2008-03-BR-HMI-VAPSQCG-web.pdf (Erişim Tarihi: Ocak, **2015**).

[16] Sendall, S., Kozaczynski, W., Model Transformation : "The Heart and Soul of Model-Driven Software Development," *IEEE Software*, cilt. 20, no. 5,42–45, **2003**.

[17] Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S., "Developing Applications Using Model-Driven Design Environments," *IEEE Computer*, vol. 39, no. 2,33–40, Şubat, **2006**.

[18] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G., *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*, **2013**.

[19] Burak, H., Geylani, S., "Java Card Yazılımlarının Model Güdümlü Geliştirilmesi," Türkiye Bilişim Vakfı Bilgisayar Bilimleri ve Mühendisliği Dergisi, cilt. 4, no. 4,1–10, **2013**.

[20] Sairaman, V., Ranganathan, N., Singh, N. S., "An automatic code generation tool for partitioned software in distributed systems," *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*,4, **2006**.

[21] Yang, T., Gerasoulis, A., "PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors," *ICS '92 Proceedings of the 6th international conference on Supercomputing*, 428–437, **1992**.

[22] Kim, J., Lee, I., "Modular code generation from hybrid automata based on data dependency," *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, 160–168, **2003**.

[23] Parr, T., StringTemplate 4 Documentation, http://www.stringtemplate.org/about.html (Erişim Tarihi: Ocak, **2015**).

[24] Miles, R., Hamilton, K., *Learning UML 2.0*. O'Reilly Media, **2006**.

[25] Mohagheghi, P., Dehlen, V., "*Developing a Quality Framework for Model-Driven Engineering,"* Springer-Verlag Berlin Heidelberg, **2008**.

[26] Anonim, MDA® Specifications, http://www.omg.org/mda/specs.htm (Ocak, **2015**).

[27] Koray, T., Yakın, İ., "Eclipse Modelleme Çerçevesi ile Model Güdümlü Yazılım Geliştirme Deneyimleri," *5. Ulusal Yazılım Mühendisliği Sempozyumu- UYMS*,263–266, **2011**.

[28] Iseger, M., Domain-specific modeling for generative software development, http://www.developerfusion.com/article/84844/domainspecific-modeling-for-generative-software-development/ (Erişim Tarihi: Ocak, **2015**).

[29] Herrington, J., *Code Generation In Action*. Manning Publications Co., **2003**.

[30] Parr, T., *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, LLC., **2012**.

[31] Vogel, P., T4 Templates - Managing Complexity in T4 Code-Generation Solutions, MSDN Magazine, http://msdn.microsoft.com/en-us/magazine/hh975350.aspx (Erişim Tarihi: Ocak, **2015**).

[32] DSTG, *ANGIE – An Introduction*, MA 21'474. Delta Software Technology GmbH, **2005**.

[33] Le, H. T., "A Frame-based Approach to Text Generation," *The 21st Pacific Asia Conference on Language, Information and Computation*, cilt. 21,192–201, **2007**.

[34] Anonim, Abstract syntax tree, http://en.wikipedia.org/wiki/Abstract_syntax_tree (Erişim Tarihi: Ocak, **2014**).

[35] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, **1995**.

[36] Anonim, OpenGL - The Industry’s Foundation for High Performance Graphics, https://www.opengl.org/ (Erişim Tarihi: Ocak, **2015**).

[37] Horstmann, C., *Object-Oriented Design & Patterns*, Second Edi. John Wiley & Sons, 194–357, **2006**.

[38] Anonim, Composite pattern, http://en.wikipedia.org/wiki/Composite_pattern (Erişim Tarihi: Ocak, **2015**).

EKLER

E.1. Tasarım Örüntüleri

Tasarım örüntüsü, yazılım mühendisliğinde birçok kez karşılaşılmış olan problemlere yönelik ortak karara varılmış olan en iyi çözümlerdir. Tez uygulamasının tasarım ve geliştirme aşamasındaki tasarım örneklerinde aşağıda tanımlanmış olan tasarım örüntülerinden faydalanılmıştır.

E.1.1. Bileşik Tasarım Örüntüsü

Bileşik tasarım örüntüsü, üst sınıf içerisinde tutulan farklı tipteki birincil sınıflar ile birlikte aynı şekilde kendisinin de dâhil edilen sınıflar gibi davrandığı bir tasarım örüntüsüdür [37]. Bileşim tasarım örüntüsü modelleme mimarisindeki grafik modelinde bulunan "Composite" sınıfı ve mantıksal modelin "Activity" sınıfının tasarımında kullanılmıştır.

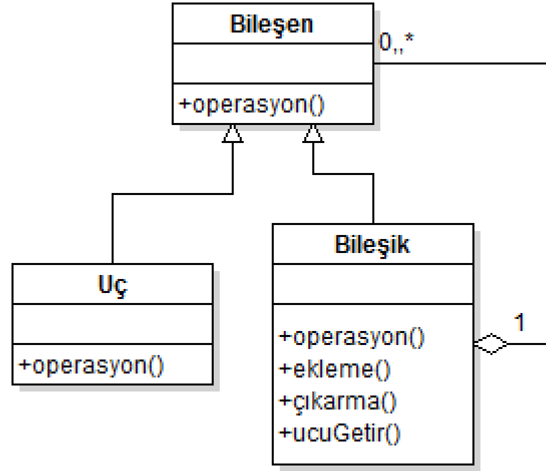
Bileşik Tasarım Örüntüsü içerik, çözüm adı altında özetlenmiş ve Şekil E.1'de gösterilmiştir.

İçerik:

1. Birincil sınıflardan oluşmuş nesneler bileşik sınıftan türemiş nesneler içerisinde bulunabilir.
2. İstemciler bileşik nesneye birincil nesne gibi davranabilir.

Çözüm:

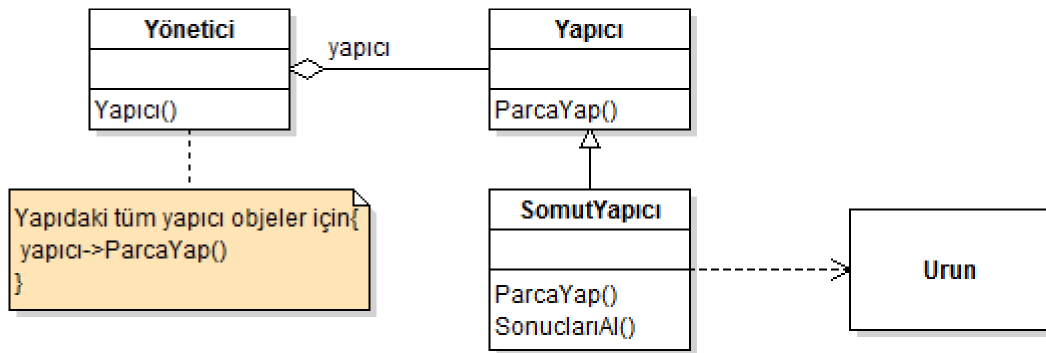
1. Arayüz tipi tanımlanarak birincil sınıflar için soyutlama sağlanır.
2. Bileşik objenin birincil nesneleri saklaması sağlanır.
3. Hem Bileşik nesnesinin hem de barındırdığı birincil nesnelerin belirli arayüzü uygulaması sağlanır.
4. Arayüzdeki bir metodu uygularken, bileşim sınıfı bu metodu içerdiği tüm birincil nesnelere de uygular ve sonuçlarını toplar [37].



Şekil E.1 Bileşik Tasarım Örüntüsü Diyagramı [38]

E.1.2. Yapıcı Tasarım Örüntüsü

Yapıcı tasarım örüntüsü, çok biçimliliği kullanarak nesne yaratılmasında kullanılan bir tasarım örüntüsüdür. Bu tasarım örüntüsü model tasarım editöründeki araç çubuğunda uygulanmıştır. Tasarlanacak olan model için seçilen blok(mantıksal) ya da çizim (grafiksel) elemanlarının ulaştırdığı tip numarasına göre Fare tuş basımı ile tetiklenen olayla birlikte model kütüphanesinden ilgili nesne yaratılarak bu pattern gerçekleştirilmiştir. Şekil E.2'de bu örüntünün çalışması özetlenmiştir.



Şekil E.2 Yapıcı Tasarım Örüntüsü Diyagramı [35]

E.2. Şablon Dosyaları

Şablon dosyaları, Kod Üreteci bölümü (4.3) altında belirtilmiş olan şablon dosyalarını ifade etmektedir. Şablon dosyaları kullanımındaki ana senaryoya göre StringTemplate Kütüphanesi kullanılarak dosyalardan okunan şablon metin verileri, parametre verileri ile işlenir, işleme sonucunda ortaya çıkan veriler, farklı işlenmiş veriler ile birleştirilerek, belirlenmiş dosyaya nihai kod olarak kaydedilir. Şablon dosyaları proje dosyası hariç C programlama diline özgü yazılmıştır. Bu kısımda tez uygulamasında geliştirilmiş olan kod şablon dosyalarına yer verilmiştir.

E.2.1. Proje Dosyası

MakeFile dosyası MinGW kütüphaneleri ve derleyicisi için gerekli olan yapı dosyasıdır. Bu dosyada şablon verisi olarak proje adının tutulduğu "*projectNam"e* ve üretilecek olan kod dosyalarının listesi olan "*sources"* veri tiplerinde saklanmaktadır.

MakeFile:

```
01  # Project: %projectName%
02  # Makefile created by ModelCodeGenerator
03
04  CPP      = g++.exe
05  CC       = gcc.exe
06  WINDRES  = windres.exe
07  OBJ      = main.o %sources:{it|%it%.o}; separator="
    "%
08  LINKOBJ  = main.o %sources:{it|%it%.o};
    separator=" "%
09  LIBS     = -L"C:/Program Files (x86)/Dev-
    Cpp/MinGW64/lib" -L"C:/Program Files (x86)/Dev-
    Cpp/MinGW64/x86_64-w64-mingw32/lib" -static-
    libgcc -mwindows -lopengl32 -lglu32
```

```
10    INCS     = -I"C:/Program Files (x86)/Dev-Cpp/MinGW64/include" -I"C:/Program Files (x86)/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.8.1/include"
11    CXXINCS  = -I"C:/Program Files (x86)/Dev-Cpp/MinGW64/include" -I"C:/Program Files (x86)/Dev-Cpp/MinGW64/x86_64-w64-mingw32/include" -I"C:/Program Files (x86)/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.8.1/include" -I"C:/Program Files (x86)/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/4.8.1/include/c++"
12    BIN      = %projectName%.exe
13    CXXFLAGS = $(CXXINCS)
14    CFLAGS   = $(INCS)
15    RM       = rm.exe -f
16
17    .PHONY: all all-before all-after clean clean-custom
18
19    all: all-before $(BIN) all-after
20
21    clean: clean-custom
22                  ${RM} $(OBJ) $(BIN)     // Todo
23
24
```

25	$(BIN): $(OBJ)	
26		$(CC) $(LINKOBJ) -o $(BIN) $(LIBS)
27		
28	main.o: main.c	
29		$(CC) -c main.c -o main.o $(CFLAGS)
30	%sources:{it| %it%.o: %it%.c	
31		$(CC) -c %it%.c -o %it%.o $(CFLAGS)
32		*}; separator="\r\n\r\n"%*

E.2.2. Arayüz

Arayüz şablon (InterfaceTemplate.st) dosyası C dili ".h" uzantılı başlık dosyası üretimi için kullanılmaktadır. Üretilen arayüz dosyası geliştirilecek olan yazılım parçasının dış parçalara bağlanabilmesi için üretilmektedir. Şablonda kullanılan "*typeT*" değerine göre normal ya da yapısal veri tiplerini tanımlar.

<u>InterfaceTemplate.st:</u>

```
00: $if(typeT)$
01: typedef $varName$ {
02:     $elements:{it|$it.varType$ $it.varName$};separator=";\n\r"$;
03:     };
04: $else$
05:     $varType$ $varName$;
06: $endif$
```

E.2.3. Grafik

Grafik arabirim kodunun üretilmesinde birden fazla şablon dosyası kullanılmaktadır. İlk olarak grafik arabiriminde üretilecek olan kodun ilk kısımları ayrı bir şablon dosyasında (GraphicsTemplate_Header.st) üretilmektedir. Bu dosyada ilgili başlık dosyaların dâhil edildiği, grafik fonksiyon tiplerinin tanımlandığı kısımlar mevcuttur.

GraphicsTemplate_Header.st

```
00	// Include files
01	#include<windows.h>
02	#include<GL\gl.h>
03	$if(anyInterfaceUsed)$
04	$inclusions: {it|#include "$it$.h"}; separator="\n\r"$
05	$endif$
06	extern void glPrint(const char *fmt, ...);
07
08
09	 $if(anyInterfaceUsed)$
10	 /* Initialize the used Interface Values */
11	void $compositeName$_init()
12	{
13		$elements:{it|$it.varType$ $it.varName$};separator=";\n\r"$;
14	}
15	$endif$
16	/* Draws the root object */
17	void $compositeName$_draw()
```

```
18      {
19              glLoadIdentity();
20              glTranslatef($posX$,$posY$,-1);
21              glRotatef(0, 0,0, 1);
22              glPushMatrix();
```

"GraphicsTemplate.st" dosyası herhangi bir birim grafik nesnesi için kullanılacak olan kod üretim şablon dosyasıdır. Bu dosya grafik tasarımında oluşturulmuş her bir birincil nesnenin kodunun üretilmesinde çağırılır. Bu dosyada ayrıca üretilecek olan grafik nesnesinde tanımlanmış olan saydamlık ve resim kaplama kısımlarının da kod üretimi yapılabilmektedir.

GraphicsTemplate.st

```
01   // Check for texture
02   $if(texture)&
03     glEnable(GL_TEXTURE_2D);
04   $endif$
05
06     glPushMatrix();
07             // Enabling transprency
08     $if(transparency)$
09             glEnable (GL_BLEND);
10             glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
11             glColor4f($color.R$,$color.G$,$color.B$, $transFactor$);
12     $else$
13             glColor3f($color.R$,$color.G$,$color.B$);
14     $endif$
15             glTranslatef($moveX$,$moveY$,$moveZ$);
16     $if(isRotateExist)$
17             glRotatef($rotateX$,$rotateY$,$rotateZ$);
```

```
18      $endif$
19
20      $if(graphicalDraw)$
21                      // object id $objectName$
22              $if(isText)$
23              glRasterPos2f($textPosX$,$textPosY$);
24              glPrint("$textualData$");
25              $else$
26                      // Check is filled ?
27                      $if(isFilled)$
28              glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
29                      $else$
30              glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
31                      $endif$
32
33                      $if(texture)$
34              // Texture bind operation
35              glBindTexture(GL_TEXTURE_2D, $textId$);
36              glBegin(GL_QUADS);
37                      glVertex3f($pX1$, $pY1$, 0);    glTexCoord2f (0, 0);
38                      glVertex3f($pX1$, $pY2$, 0);     glTexCoord2f (1, 0);
39                      glVertex3f($pX2$, $pY2$, 0);      glTexCoord2f (1, 1);
40                      glVertex3f($pX2$, $pY1$, 0);     glTexCoord2f (0, 1);41
        glEnd();
41                      $else$
42              // Drawing Element
43              glBegin($objectType$);
44                      $vertexes:{it|glVertex3f($it.X$, $it.Y$,
$it.Z$)};separator=";\n\r"$;
45              glEnd();
46                      $endif$
```

```
47              $endif$
48      $else$
49                      $innerCompositeName$_draw();
50      $endif$
51      $if(masking)$
52              glDisable(GL_BLEND);
53      $endif$
54      glPopMatrix();
55    $if(texture)&
56      glDisable(GL_TEXTURE_2D);
57    $endif$
```

"*GraphicsTemplate_Footer.st*" dosyası genel bir cizim fonksiyonu tanımlar. Bu fonksiyonun içinde modelde tanımlanmış olan grafik modelinin isminin bulunduğu çizim fonksiyonunu çağıran alt fonksiyon mevcuttur. Bu fonksiyon grafik arabirim kodunun son kısımlarını içermektedir.

GraphicsTemplate_Footer.st

```
00              glPopMatrix();
01      }
02      /* Main draw function */
03      __declspec(dllexport) void draw()
04      {
05              $compositeName$_draw();
06      }
```

E.2.4. Mantıksal

Mantıksal şablon dosyaları da yapı olarak grafiksel şablon dosyaları gibi başlık, ana kod ve son kısım parçalarından oluşmaktadır. Başlık kısımlarında tasarlanmış modellerin C programlama dilindeki fonksiyon tip karşılığının belirlendiği başlık dosyaları üretilir. Bu fonksiyon tanımları üretilirken modelin girdi ve çıktılarıda ayrı şablon dosyalarında üretilir (*LogicsTemplate_FuncInputList.st,*

LogicsTemplate_FuncOutputList.st) ve daha sonra bu ana başlık dosyasında (*LogicsTemplate_FuncProto.st*) kullanılır. Ana başlık dosyası ile birlikte, kullanılan arayüz dosyalarının içerildiği şablon dosyasının (*LogicsTemplate_Header.st*) ürettiği veriler birleştirilerek nihai başlık dosyası üretilir.

LogicsTemplate_Header.st

```
01 // Include files
02 $inclusions: {it|#include "$it$.h"}; separator="\n\r"$
```

LogicsTemplate_FuncInputList.st

```
01 $inTypes:{inT| $inT$ /* input */ }; separator=",\n\r\t"$
```

LogicsTemplate_FuncOutputList.st

```
01 $outTypes:{outT| *$outT$ /* output */ }; separator=",\n\r"$
```

LogicsTemplate_FuncProto.st

```
01 // Activity $protoName$ protoype
02 void $protoName$
03      ($inputOutputList$);
```

LogicsTemplate_FuncHeader.st şablon dosyasında tanımlanmış olan fonksiyonun kaynak kodunun oluşturulacağı C uzantılı dosya, benzer şekilde tanımlanan fonksiyon tipi ve mantıksal modelde kullanılmış olan kısmi değişkenlerin tanımlanması ile başlar.

LogicsTemplate_FuncHeader.st

```
01 // Activity $protoName$ function
02 void $protoName$
03      ($inputOutputList$)
04 {
05      // Definition of local variables
06      $localVariables:{it|$it$ = 0;};separator="\n\r"$
```

Kısmı değişkenlerin tanımlanmasından sonra, ana işlem şablon dosyaları mantıksal model tasarımda kullanılmış olan birincil blokların işlemlerine, sıralamalarına, fonksiyon girdi ve çıktılarının kullanımlarına göre yinelemeli olarak çağrılmaktadır. Aşağıda sırasıyla eşitlik, işlem tanımlı operasyon, koşul ve döngü şablon dosyaları verilmiştir.

LogicsTemplate_FuncEquOpr.st

```
01 $output$ = $input$;
```

LogicsTemplate_FuncOpr.st

```
01 $output$ $opr$ $input$;
```

LogicsTemplate_FuncIFOpr.st

```
01 if ($condition$)
02      {
03              $outputs:{output| $output$ /* input */ }; separator="=\n\r\t\t"$ = $trueAction$;
04      }
05      else
06      {
07              $outputs:{output| $output$ /* input */ }; separator="=\n\r\t\t"$ = $falseAction$;
08      }
```

LogicsTemplate_FuncLoopOpr.st

```
01   {
02              int loopInc = 0;
03              for (loopInc = $loopStart$; loopInc <= $loopFinish$; loopInc = loopInc + $loopIncrement$)
```

```
04            {
05                  $operation$
06            }
07      }
```

E.3. MOBASO ile Model Tasarımı ve Üretilmiş Kod

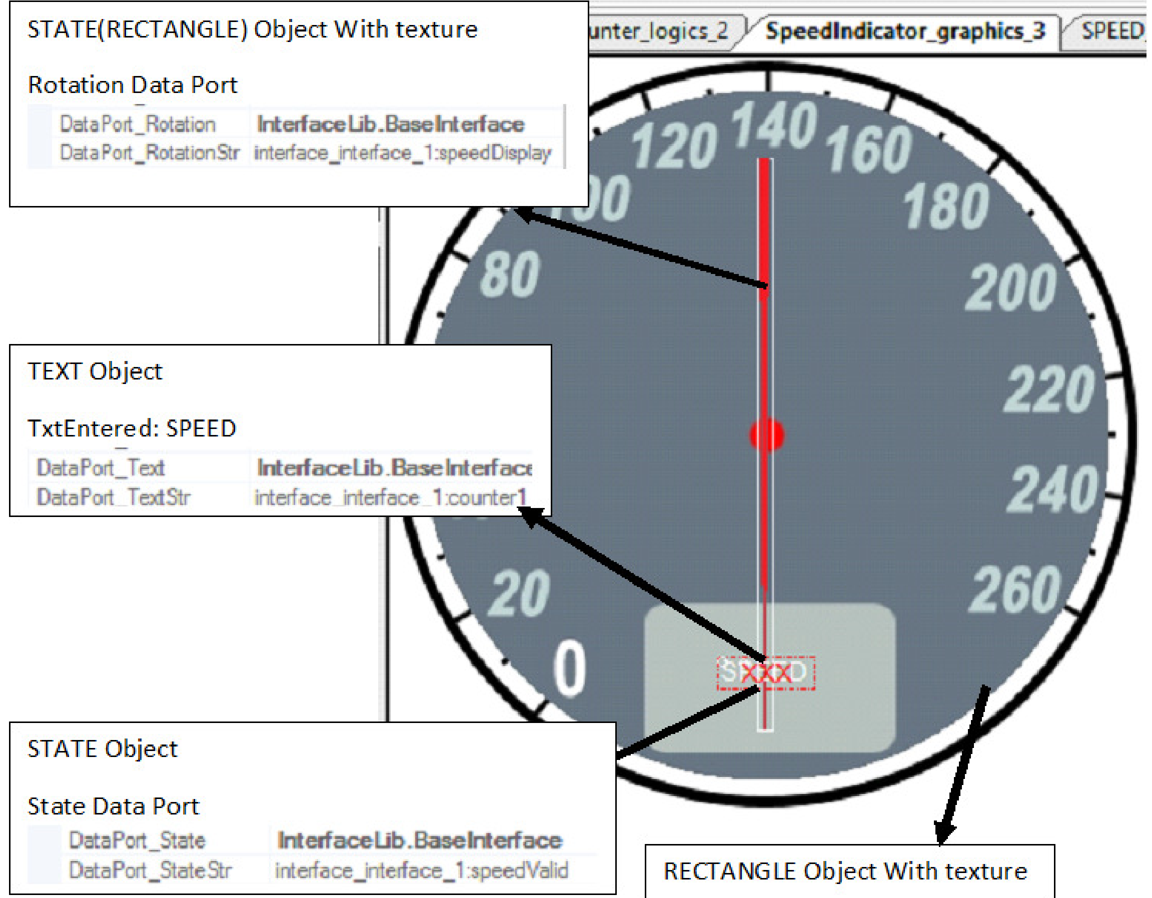
Bu kısımda MOBASO anaçatısının grafiksel ve mantıksal tasarımlarını tanıtmak ve bu tasarımlardan üretilmiş olan kod için eklenmiştir. Şekil E.3'te verilen gösterge tarzında olan hız göstergesinin MOBASO anaçatısı ile tasarımı verilecektir.

Şekil E.3 Speed Indicator

Bu gösterge için dört adet gereksinim verilmiş olsun;

1. Hız göstergesi ibresi *speedValue* olan hız değerini 0-200 aralığında dönüş yaparak gösterecektir.
2. Hız değerinin geçerliliği olan *speedValidity* geçerli (doğru: 1) olduğunda, Hız göstergesi *speedValue* olan hız değerini hız göstergesinin alt tarafında rakamsal olarak gösterecektir.
3. Hız değerinin geçerliliği olan *speedValidity* geçersiz(yanlış: 2) olduğunda, Hız göstergesi *speedValue* olan hız değerini hız göstergesinin alt tarafında üç adet kırmızı "X" olarak gösterecektir.
4. Hız değeri olan *speedValue* 180 değerinden büyük ise, hız geçerliliği olan *speedValidity* geçersiz(yanlış: 2) olacaktır.

E.3.1. Grafiksel Tasarım



Şekil E.4 Hız göstergesi grafiksel tasarım

Şekil E.4'te verilen *SpeedIndicator_graphics_3* isimli grafiksel tasarım, arka plan ve gösterge ibresinin doku resimleri gösterdiği iki adet *Rectangle*(bir tanesi *State* nesnesi içerisinde), ayrıca başka bir State nesnesi içinde bulunan iki *Text* nesnesinden oluşmaktadır. Veri bağlantısı olan nesneler *interface_interface_1* isimli arayüzün elemanlarına bağlanmıştır. Grafiksel tasarımda kullanılan arayüz verileri mantıksal tasarımda çıktı olarak kullanılmaktadırlar. Bu veriler;

- *Text* nesnesi *counter1* değerini hız değeri olarak gösterir.
- *speedValid* verisi, *Text* nesneleri barındıran *State* nesnesinin çalışma anındaki aktif olan *Text* nesnesini belirlemek için kullanılır.
- Hız ibresinin *speedDisplay* değerine göre ilgili değerin posizyonunu gösterebilmek için döndürülmesinde kullanılır. (Başlangıç değeri olan 0

değeri -40 derece daha ilerdedir, bu sepeble -40 kayma değeri olarak kullanılacaktır).

Grafiksel tasarım verilmiş olan 1, 2 ve 3 gereksinimlerini parçalı olarak kapsar.

E.3.2. Algoritma 1

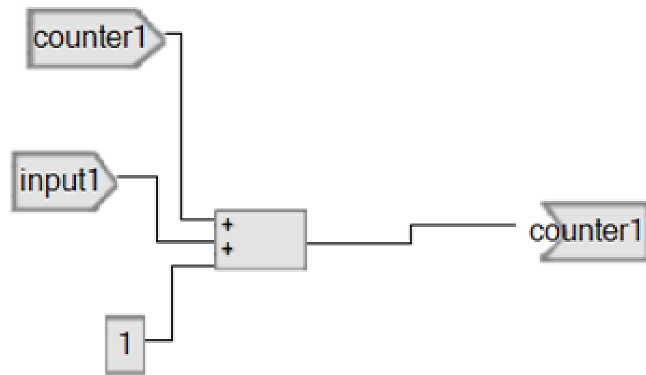
Belirtilen bir numaralı gereksinimde bir sayaca ihtiyaç vardır. Sıralı olarak [0-200] ve [200-0] aralıklarında sayım yapan bir sayaç algoritması ve mantıksal tasarımı gösterilecektir.

Basamaklar:

1. *counter1* değerini *counter1*, *input1* ve 1 toplamına eşitle.
2. *counter1*+1 değeri 200’ü aştı mı veya *counter1*-1 0’dan küçük mü?
   a. Doğru ise *input1* değerini (*input1*+2)*(-1) hesabının sonucuna eşitle.
   b. Yanlış ise *input1* değerini kendine eşitle.

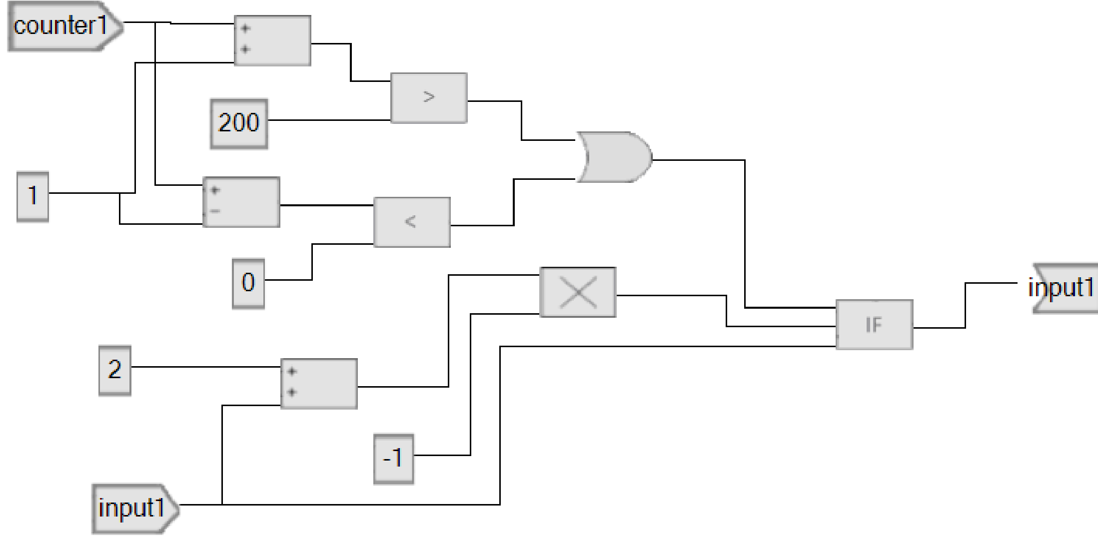
E.3.3. Algorithma 1 için Mantıksal Tasarım

Mantıksal tasarımda sayaç için geliştirilen *counter_logics_2* diyagramı Şekil E.5’te gösterilmiştir. Buna göre *counter1* değeri *input1* değerine göre artış (*input1* = 0) veya azalış (*input1* = -2) gösterecektir.



Şekil E.5 Sayaç Tasarımı

counter1 değerinin artış veya azalış durumunu kontrol eden *input1* değerinin hesaplanmasında tasarlanan mantıksal tasarım olan *_logics_1* Şekil E.6'da gösterilmiştir.



Şekil E.6 Sayaç Kontrol Tasarımı

Algoritma 1, [0,200] aralığı ile ilgili olan birinci gereksinimi karşılar.

E.3.4. Algorithma 2

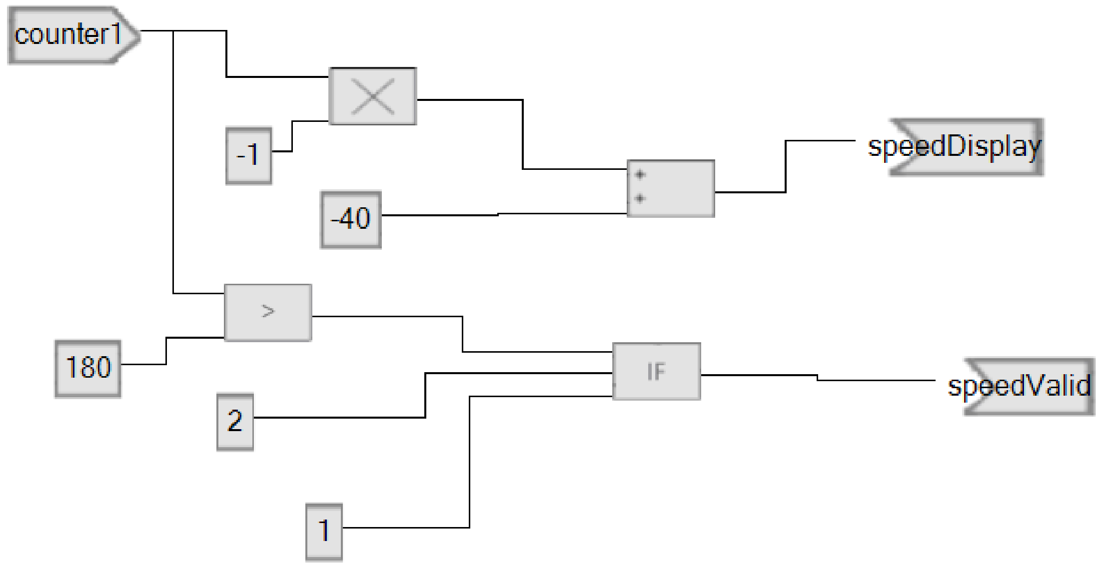
Bu algoritmada hız göstergesinin ibreye göre dönüşünü sağlayan değerin hesaplanması ve hız değerinin geçerliliğinin hesaplanması gösterilecektir.

Basamaklar:

1. *speedDisplay* değerini (-40 + (*counter1* * (-1))) hesabına eşitle
2. *counter1* değeri 180'i geçti mi ?
   a. Doğru ise *speedValid* değerini 2'ye eşitle.
   b. Yanlış ise *speedValid* değerini 1'e eşitle.

E.3.5. Algoritma 2 için Mantıksal Tasarım

Mantıksal tasarımda geliştirilen *SpeedLogics_4,* Şekil E.7'de gösterildiği gibi *speedDisplay* ve *speedValid* değerlerini algoritmada tanımlanan şekli ile ifade etmektedir.



Şekil E.7 İbre Değeri ve Hız Geçerliliği

Algoritma 2, başlangıçta bahsedilen gereksinimlerden ikinci ve üçüncü gereksinimleri kısmen, dördüncü gereksinimi ise tamamen karşılamaktadır.

E.3.6. Üretilen Kodlar

Grafiksel ve mantıksal tasarımlar sonrasında yapılan kod üretimi işlemi sonucunda 5 adet dosya (*Main.c, Noname_LogicHeader.h, interface_interface_1.h, counter_logics_2.c, _logics_1.c, SPEED_logics_4.c, SpeedIndicator_graphics_3.c)* ortaya çıkmaktadır. *Main.c* dosyası görsel tanımlanmış olan tasarımların oluşan kodlarını sıra ile çağıran ana dosyadır.

Main.c (Bir kısmı)

```
GLuinttexture[10];

int LoadGLTextures()                                        //
Load Bitmaps And Convert To Textures

{

        int Status = FALSE;                                  //
Status Indicator

        int Counter = 0;
```

```
char FileName[2][255] = {
        "E:/demoPic/SpeedoMeter.bmp",
        "E:/demoPic/needle.bmp"
        };
for (Counter = 0; Counter <  2; Counter++) {
// Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit

         texture[Counter] = SOIL_load_OGL_texture
(
FileName[Counter],
SOIL_LOAD_AUTO,
SOIL_CREATE_NEW_ID,
SOIL_FLAG_INVERT_Y
);
        if(texture[Counter] == 0)
Status = FALSE;
        // Typical Texture Generation Using Data From The Bitmap
        glBindTexture(GL_TEXTURE_2D, texture[Counter]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEA
REST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LIN
EAR);
    }
    return Status;
    // Return The Status
}
```

```
void Thread ( void *pParams)

{

	while(1)

	{

		SPEED_logics_4(counter1,

			&speedDisplay,

			&speedValid);

		counter_logics_2(counter1,

			input1,

			&counter1);

		_logics_1(counter1,

			input1,

			&input1);

		Sleep(50);

	}

}

int InitGL(GLvoid)                                                  //
All Setup For OpenGL Goes Here

{

	glShadeModel(GL_SMOOTH);                                        //
Enable Smooth Shading

	glClearColor(0.0f, 0.0f, 0.0f, 0.5f);                 //        Black
Background

	glClearDepth(1.0f);                                             //
Depth Buffer Setup
```

```
	glEnable(GL_DEPTH_TEST);					//
Enables Depth Testing

	glDepthFunc(GL_LEQUAL);
	// The Type Of Depth Testing To Do

	glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);	// Really
Nice Perspective Calculations

	_beginthread(Thread, 0, NULL);

	BuildFont();						//
Build The Font

	LoadGLTextures();

	return TRUE;						//
Initialization Went OK

}


int DrawGLScene(GLvoid)						//
Here's Where We Do All The Drawing

{

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //	Clear
Screen And Depth Buffer

	draw();

							// Increase The
First Counter

	return TRUE;						//
Everything Went OK

}
```

Noname_LogicHeader.h

```
/*
```

```
 * This file is generated by Model Based Generator
 * @name: Noname_LogicHeader.h
 * This is logic common header file that includes all logic model prototypes
 * Generated at: 15/01/2015 19:22:59
 */
#ifndef NONAME_LOGICS
#define NONAME_LOGICS
// Include files
#include "interface_interface_1.h"
// Activity _logics_1 protoype
void _logics_1
        (int counter1 /* input */ ,
        int input1_In /* input */ ,
        int *input1 /* output */ );
// Activity counter_logics_2 protoype
void counter_logics_2
        (int counter1_In /* input */ ,
        int input1 /* input */ ,
        int *counter1 /* output */ );
// Activity SPEED_logics_4 protoype
void SPEED_logics_4
        (int counter1 /* input */ ,
        int *speedDisplay /* output */ ,
        int *speedValid /* output */ );
```

```
#endif
```

interface_interface_1.h

```
/* This file is generated by Model Based Generator
 * @name: interface_interface_1.h
 * This is a interface which can be used from Graphical, Logical models
 * Generated at: 15.1.2015 19:22:59
 */
#ifndef INTERFACE_INTERFACE_1
#define INTERFACE_INTERFACE_1
	int counter1;
	int input1;
	int speedDisplay;
	int speedValid;
#endif
```

counter_logics_2.c

```
/*
 * This file is generated by Model Based Generator
 * @name:counter_logics_2.c
 * This is graphics source file
 * Generated at: 15/01/2015 19:22:59
 */
#include "Noname_LogicHeader.h"
// Activity counter_logics_2 function
void counter_logics_2
```

```
        (int counter1_In /* input */ ,
        int input1 /* input */ ,
        int *counter1 /* output */ )
{
        // Definition of local variables
        int INPUT_54_ADD_58_dataOut = 0;
        int INPUT_54_ADD_58_dataIn = 0;
        int INPUT_56_ADD_58_dataIn = 0;
        int CONST_57_ADD_58_dataIn = 0;
        int ADD_58_OUTPUT_55_dataOut = 0;
        int INPUT_56_ADD_58_dataOut = 0;
        int CONST_57_ADD_58_dataOut = 0;
        int ADD_58_OUTPUT_55_dataIn = 0;
 // Generating Logic operation of model
        INPUT_54_ADD_58_dataOut = counter1_In;
        INPUT_54_ADD_58_dataIn = INPUT_54_ADD_58_dataOut;
        INPUT_56_ADD_58_dataOut = input1;
        INPUT_56_ADD_58_dataIn = INPUT_56_ADD_58_dataOut;
        CONST_57_ADD_58_dataOut = 1;
        CONST_57_ADD_58_dataIn = CONST_57_ADD_58_dataOut;
        ADD_58_OUTPUT_55_dataOut = INPUT_54_ADD_58_dataIn +
INPUT_56_ADD_58_dataIn + CONST_57_ADD_58_dataIn;


        ADD_58_OUTPUT_55_dataIn = ADD_58_OUTPUT_55_dataOut;
        *counter1 = ADD_58_OUTPUT_55_dataIn;
```

```
}
```

_logics_1.c

```
/*
 * This file is generated by Model Based Generator
 * @name:_logics_1.c
 * This is graphics source file
 * Generated at: 15/01/2015 19:22:59
 */
#include "Noname_LogicHeader.h"
// Activity _logics_1 function
void _logics_1
	(int counter1 /* input */ ,
	int input1_In /* input */ ,
	int *input1 /* output */ )
{
	// Definition of local variables
	int INPUT_5_ADD_6_dataOut = 0;
	int INPUT_5_SUBS_7_dataOut = 0;
	int INPUT_5_ADD_6_dataIn = 0;
	int CONST_12_ADD_6_dataIn = 0;
	int ADD_6_GR_8_dataOut = 0;
	int ADD_6_GR_23_dataOut = 0;
	int INPUT_5_SUBS_7_dataIn = 0;
	int CONST_12_SUBS_7_dataIn = 0;
```

```
int SUBS_7_LE_9_dataOut = 0;
int CONST_12_ADD_6_dataOut = 0;
int CONST_12_SUBS_7_dataOut = 0;
int ADD_6_GR_23_dataIn = 0;
int CONST_24_GR_23_dataIn = 0;
int GR_23_OR_10_dataOut = 0;
int CONST_24_GR_23_dataOut = 0;
int GR_23_OR_10_dataIn = 0;
int LE_9_OR_10_dataIn = 0;
int OR_10_IF_11_dataOut = 0;
int SUBS_7_LE_9_dataIn = 0;
int CONST_19_LE_9_dataIn = 0;
int LE_9_OR_10_dataOut = 0;
int OR_10_IF_11_dataIn = 0;
int MULT_45_IF_11_dataIn = 0;
int INPUT_37_IF_11_dataIn = 0;
int IF_11_OUTPUT_35_dataOut = 0;
int CONST_19_LE_9_dataOut = 0;
int IF_11_OUTPUT_35_dataIn = 0;
int INPUT_37_ADD_38_dataOut = 0;
int INPUT_37_IF_11_dataOut = 0;
int INPUT_37_ADD_38_dataIn = 0;
int CONST_40_ADD_38_dataIn = 0;
int ADD_38_MULT_45_dataOut = 0;
```

```
int CONST_40_ADD_38_dataOut = 0;
int CONST_44_MULT_45_dataIn = 0;
int ADD_38_MULT_45_dataIn = 0;
int MULT_45_IF_11_dataOut = 0;
int CONST_44_MULT_45_dataOut = 0;
// Generating Logic operation of model
INPUT_5_ADD_6_dataOut = counter1;
INPUT_5_ADD_6_dataIn = INPUT_5_ADD_6_dataOut;
CONST_12_ADD_6_dataOut = 1;
CONST_12_ADD_6_dataIn = CONST_12_ADD_6_dataOut;
ADD_6_GR_23_dataOut =
ADD_6_GR_8_dataOut = INPUT_5_ADD_6_dataIn + CONST_12_ADD_6_dataIn;
ADD_6_GR_23_dataIn = ADD_6_GR_23_dataOut;
CONST_24_GR_23_dataOut = 200;
CONST_24_GR_23_dataIn = CONST_24_GR_23_dataOut;
GR_23_OR_10_dataOut = ADD_6_GR_23_dataIn > CONST_24_GR_23_dataIn;
GR_23_OR_10_dataIn = GR_23_OR_10_dataOut;
INPUT_5_SUBS_7_dataOut = counter1;
INPUT_5_SUBS_7_dataIn = INPUT_5_SUBS_7_dataOut;
CONST_12_SUBS_7_dataOut = 1;
CONST_12_SUBS_7_dataIn = CONST_12_SUBS_7_dataOut;
```

```
SUBS_7_LE_9_dataOut = INPUT_5_SUBS_7_dataIn - CONST_12_SUBS_7_dataIn;
SUBS_7_LE_9_dataIn = SUBS_7_LE_9_dataOut;
CONST_19_LE_9_dataOut = 0;
CONST_19_LE_9_dataIn = CONST_19_LE_9_dataOut;
LE_9_OR_10_dataOut = SUBS_7_LE_9_dataIn < CONST_19_LE_9_dataIn;
LE_9_OR_10_dataIn = LE_9_OR_10_dataOut;
OR_10_IF_11_dataOut = GR_23_OR_10_dataIn | LE_9_OR_10_dataIn;
OR_10_IF_11_dataIn = OR_10_IF_11_dataOut;
CONST_44_MULT_45_dataOut = -1;
CONST_44_MULT_45_dataIn = CONST_44_MULT_45_dataOut;
INPUT_37_ADD_38_dataOut = input1_In;
INPUT_37_ADD_38_dataIn = INPUT_37_ADD_38_dataOut;
CONST_40_ADD_38_dataOut = 2;
CONST_40_ADD_38_dataIn = CONST_40_ADD_38_dataOut;
ADD_38_MULT_45_dataOut = INPUT_37_ADD_38_dataIn + CONST_40_ADD_38_dataIn;
ADD_38_MULT_45_dataIn = ADD_38_MULT_45_dataOut;
MULT_45_IF_11_dataOut = CONST_44_MULT_45_dataIn * ADD_38_MULT_45_dataIn;
MULT_45_IF_11_dataIn = MULT_45_IF_11_dataOut;
INPUT_37_IF_11_dataOut = input1_In;
INPUT_37_IF_11_dataIn = INPUT_37_IF_11_dataOut;
if (OR_10_IF_11_dataIn)
{
```

```
        IF_11_OUTPUT_35_dataOut /* input */  = MULT_45_IF_11_dataIn;
    }
    else
    {
        IF_11_OUTPUT_35_dataOut /* input */  = INPUT_37_IF_11_dataIn;
    }
    IF_11_OUTPUT_35_dataIn = IF_11_OUTPUT_35_dataOut;
    *input1 = IF_11_OUTPUT_35_dataIn;
}
```

SPEED_logics_4.c

```
/*
 * This file is generated by Model Based Generator
 * @name:SPEED_logics_4.c
 * This is graphics source file
 * Generated at: 15/01/2015 19:22:59
 */
#include "Noname_LogicHeader.h"
// Activity SPEED_logics_4 function
void SPEED_logics_4
    (int counter1 /* input */ ,
    int *speedDisplay /* output */ ,
    int *speedValid /* output */ )
{
    // Definition of local variables
```

```
int INPUT_70_MULT_71_dataOut = 0;
int INPUT_70_GR_81_dataOut = 0;
int INPUT_70_MULT_71_dataIn = 0;
int CONST_72_MULT_71_dataIn = 0;
int MULT_71_ADD_75_dataOut = 0;
int CONST_72_MULT_71_dataOut = 0;
int MULT_71_ADD_75_dataIn = 0;
int CONST_77_ADD_75_dataIn = 0;
int ADD_75_OUTPUT_79_dataOut = 0;
int CONST_77_ADD_75_dataOut = 0;
int ADD_75_OUTPUT_79_dataIn = 0;
int INPUT_70_GR_81_dataIn = 0;
int CONST_83_GR_81_dataIn = 0;
int GR_81_IF_86_dataOut = 0;
int CONST_83_GR_81_dataOut = 0;
int GR_81_IF_86_dataIn = 0;
int CONST_89_IF_86_dataIn = 0;
int CONST_91_IF_86_dataIn = 0;
int IF_86_OUTPUT_84_dataOut = 0;
int IF_86_OUTPUT_84_dataIn = 0;
int CONST_89_IF_86_dataOut = 0;
int CONST_91_IF_86_dataOut = 0;

// Generating Logic operation of model
```

```
INPUT_70_MULT_71_dataOut = counter1;
INPUT_70_MULT_71_dataIn = INPUT_70_MULT_71_dataOut;
CONST_72_MULT_71_dataOut = -1;
CONST_72_MULT_71_dataIn = CONST_72_MULT_71_dataOut;
MULT_71_ADD_75_dataOut = INPUT_70_MULT_71_dataIn * CONST_72_MULT_71_dataIn;
MULT_71_ADD_75_dataIn = MULT_71_ADD_75_dataOut;
CONST_77_ADD_75_dataOut = -40;
CONST_77_ADD_75_dataIn = CONST_77_ADD_75_dataOut;
ADD_75_OUTPUT_79_dataOut = MULT_71_ADD_75_dataIn + CONST_77_ADD_75_dataIn;
ADD_75_OUTPUT_79_dataIn = ADD_75_OUTPUT_79_dataOut;
*speedDisplay = ADD_75_OUTPUT_79_dataIn;
INPUT_70_GR_81_dataOut = counter1;
INPUT_70_GR_81_dataIn = INPUT_70_GR_81_dataOut;
CONST_83_GR_81_dataOut = 180;
CONST_83_GR_81_dataIn = CONST_83_GR_81_dataOut;
GR_81_IF_86_dataOut = INPUT_70_GR_81_dataIn > CONST_83_GR_81_dataIn;
GR_81_IF_86_dataIn = GR_81_IF_86_dataOut;
CONST_89_IF_86_dataOut = 2;
CONST_89_IF_86_dataIn = CONST_89_IF_86_dataOut;
CONST_91_IF_86_dataOut = 1;
CONST_91_IF_86_dataIn = CONST_91_IF_86_dataOut;
```

```
        if (GR_81_IF_86_dataIn)
        {
                IF_86_OUTPUT_84_dataOut /* input */  = CONST_89_IF_86_dataIn;
        }
        else
        {
                IF_86_OUTPUT_84_dataOut /* input */  = CONST_91_IF_86_dataIn;
        }
        IF_86_OUTPUT_84_dataIn = IF_86_OUTPUT_84_dataOut;
        *speedValid = IF_86_OUTPUT_84_dataIn;
}
```

SpeedIndicator_graphics_3.c

```
/*
 * This file is generated by Model Based Generator
 * @name:SpeedIndicator_graphics_3.c
 * This is graphics source file
 * Generated at: 15/01/2015 19:22:59
 */
// Include files
#include<windows.h>
#include<GL\gl.h>
#include <gl\glu.h>


#include "interface_interface_1.h"
```

```
extern void glPrint(const char *fmt, ...);

extern GLuint         texture[10];

static int once = 0;

 /* Initialize the used Interface Values */

void SpeedIndicator_graphics_3_init()

{

        if(once == 0){

        speedDisplay = 0;

        speedValid = 0;

        counter1 = 0;

        once = 1;

        }

}

/* Draws the root object */

void SpeedIndicator_graphics_3_draw()

{

        glLoadIdentity();

        glPushMatrix();

        glTranslatef(0, 0, -1);

        glPushMatrix();

                glColor3f(1,1,1);

                glEnable(GL_TEXTURE_2D);

                glBindTexture(GL_TEXTURE_2D, texture[0]);
```

```
                // object id RECTANGE5
                // glRotatef(180, 0, 1, 0);
                glBegin(GL_QUADS);
                glTexCoord2f(0.0, 1.0); glVertex3f(-1,1,0.0);
                glTexCoord2f(0.0, 0.0); glVertex3f(-1,-1,0.0);
                glTexCoord2f(1.0, 0.0); glVertex3f(1,-1,0.0);
                glTexCoord2f(1.0, 1.0); glVertex3f(1,1,0.0);
                glEnd();
        glDisable(GL_TEXTURE_2D);
glPopMatrix();
glPushMatrix();
        glColor3f(1,1,1);
                // object id RECTANGE8
                glBegin(GL_LINE_LOOP);
                        glVertex3f(-0.02255639, 0.7543859, 0);
                        glVertex3f(-0.02255639, 0.7543859, 0);
                        glVertex3f(-0.02255639, 0.7543859, 0);
                        glVertex3f(-0.02255639, 0.7543859, 0);
                glEnd();
glPopMatrix();
glPushMatrix();
        glColor3f(1,1,1);
                // object id RECTANGE9
                glBegin(GL_LINE_LOOP);
```

```
                    glVertex3f(-0.02756892, 0.5789474, 0);
                    glVertex3f(-0.02756892, 0.5789474, 0);
                    glVertex3f(-0.02756892, 0.5789474, 0);
                    glVertex3f(-0.02756892, 0.5789474, 0);
                glEnd();
        glPopMatrix();
        glPushMatrix();
                glRotatef(speedDisplay,0, 0, 1);
                glPushMatrix();
                    glColor3f(1,1,1);
                    glEnable(GL_TEXTURE_2D);
                    glBindTexture(GL_TEXTURE_2D, texture[1]);
                        // object id RECTANGE11
                        // glRotatef(180, 0, 1, 0);
                        glBegin(GL_QUADS);
                        glTexCoord2f(0.0,        1.0);        glVertex3f(-
0.02756892,0.7343358,0.0);
                        glTexCoord2f(0.0, 0.0); glVertex3f(-0.02756892,-
0.7794486,0.0);
                        glTexCoord2f(1.0,  0.0);  glVertex3f(0.01253133,-
0.7794486,0.0);
                        glTexCoord2f(1.0,                          1.0);
glVertex3f(0.01253133,0.7343358,0.0);
                        glEnd();
                    glDisable(GL_TEXTURE_2D);
```

```
                    glPopMatrix();
        glPopMatrix();
        switch(speedValid)
        {                       case 1:
                                glPushMatrix();
                                glColor3f(1,1,1);
                                        // object id TEXT6
                                glRasterPos2f(-0.1328321,-0.6340852);
                                glPrint("%d",counter1);
                                glPopMatrix();
                                break;
                                case 2:
                                glPushMatrix();
                                glColor3f(1,0,0);
                                        // object id TEXT5
                                glRasterPos2f(-0.07769424,-0.6390978);
                                glPrint("XXX");
                                glPopMatrix();
                                break;
        }
        glPopMatrix();
}
/* Main draw function */
void draw()
```

```
{
    SpeedIndicator_graphics_3_draw();
}
```

ÖZGEÇMİŞ

Kimlik Bilgileri

Adı Soyadı : İbrahim ARDIÇ

Doğum Yeri : Suluova

Medeni Hali : Bekar

E-posta : ardicib@gmail.com

Adresi : Göksu Mah. 90.Sokak Spor Sit. 4B/30 Eryaman/Etimesgut/Ankara

Eğitim

Lise : 1998-2002 Fatih Anadolu Lisesi, Eskişehir

Lisans : 2002-2007 Eskişehir Osmangazi Üniversitesi Bilgisayar Mühendisliği Bölümü, Eskişehir

Lisans : 2004-2007 Eskişehir Osmangazi Üniversitesi Elektrik-Elektronik Mühendisliği Bölümü, Eskişehir

Yüksek Lisans : 2007-2015 Hacettepe Üniversitesi Elektrik-Elektronik Mühendisliği Bölümü, Ankara

Yabancı Dil ve Düzeyi

İngilizce – İyi

İş Deneyimi

2007- Aviyonik Yazılım Mühendisi, TAI -TUSAŞ A.Ş.

Deneyim Alanları

Yazılım Mühendisliği, Gömülü Yazılım, Aviyonik, Sistem Mühendisliği.

Tezden Üretilmiş Projeler ve Bütçesi

-

Tezden Üretilmiş Yayınlar

1) *XML Based Model Language Design and Software Development Tool* - SEDE 2014 – ISCA 23rd INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND DATA ENGINEERING (SEDE), 13-15 Ekim 2014, New Orleans, Louisiana, ABD
2) *Template Based Code Generation Techniques and Algorithms for Specific XML Based Models* - CATA 2015 – ISCA 30th INTERNATIONAL CONFERENCE ON COMPUTERS AND THEIR APPLICATIONS (CATA), 9-11 Mart 2015, Honolulu, Hawaii, ABD