İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

NESNEYE YÖNELİK PROGRAMLAMAYA ROL DESTEĞİNİN KAZANDIRILMASI

DOKTORA TEZİ

Y. Müh. Yunus Emre SELÇUK

Anabilim Dalı: BİLGİSAYAR MÜHENDİSLİĞİ

Programı: BİLGİSAYAR MÜHENDİSLİĞİ

TEMMUZ 2006

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

NESNEYE YÖNELİK PROGRAMLAMAYA ROL DESTEĞİNİN KAZANDIRILMASI

DOKTORA TEZİ

Y. Müh. Yunus Emre SELÇUK
(504992423)

Tezin Enstitüye Verildiği Tarih : 15 Aralık 2005
Tezin Savunulduğu Tarih : 6 Temmuz 2006

TEZ DANIŞMANI : **Prof.Dr. Nadia ERDOĞAN**

Diğer Jüri Üyeleri **Yrd.Doç.Dr. Feza BUZLUCA**

Prof.Dr. Oya KALIPSIZ (Y.T.Ü.)

Yrd.Doç.Dr. Ayşegül GENÇATA

Doç.Dr. Can ÖZTURAN (Boğaziçi.Ü.)

TEMMUZ 2006

ÖNSÖZ

Çalışmalarım sırasında bana yol gösteren ve destek olan sayın hocam Prof. Dr. Nadia Erdoğan'a, manevi desteklerini hiçbir şart altında esirgemeyen eşim Sena Günay Selçuk'a ve aileme en içten teşekkürlerimi sunarım.

Temmuz 2006 **Yunus Emre Selçuk**

İÇİNDEKİLER

KISALTMALAR **ix**
TABLO LİSTESİ **x**
ŞEKİL LİSTESİ **xi**
ÖZET **xi**
SUMMARY **xviii**

1. GİRİŞ **1**
1.1 Problemin İncelenmesi 1
1.2 Çözüm Önerisi: Nesne Düzeyinde Özelleştirme ve Rol Modelleri 2
1.3 Tezin Amacı ve Özgün Katkıları 3
1.3.1 Nesne Düzeyinde Çoklu Kalıtım 4
1.3.2 Rol Aktarımı 4
1.3.3 Ad ile Üye Erişimi 5
1.3.4 Rollerin Askıya Alınması ve Askıdan İndirilmesi 5
1.3.5 Baskın Roller 5
1.3.6 Danışma ve Vekalet Mekanizmalarının Birlikte Kullanımı 6
1.3.7 Rol Araması ve Geçişinde İyileştirme 6
1.3.8 Sonuç 7

2. NYP TEMEL KAVRAMLARI **8**
2.1 Nesne 8
2.2 Sınıf 8
2.3 Çok Anlam Yükleme (Overloading) 9
2.4 Kalıtım 9
2.5 Çoklu Kalıtım 10
2.6 Çok Biçimlilik 10
2.7 Soyut Sınıflar 11
2.8 Nesneler Arasındaki İlişkiler 11
2.8.1 Belirtme (IsDesignatedBy) 11
2.8.2 Örneğidir (IsInstanceOf) 11
2.8.3 Aynısıdır (IsA) 11
2.8.4 Etkileşir (IsAssociatedWith) 12
2.8.5 Kullanılır (IsUsedBy) 12
2.8.6 Bir Parçasıdır (IsPartOf) 12

3. DİNAMİK SİSTEMLERİN MODELLENMESİ İÇİN BAZI ÇALIŞMALAR 13
3.1 Nesne Arayüzleri 13
3.1.1 Nesne Modeli, Nesne Yüzleri ve Kullanım Şekilleri 14
3.1.2 Değerlendirme 19
3.2 Etmen Tabanlı Sistemler 19
3.2.1 Değerlendirme 20
3.3 Şema Evrimleştirme 20
3.3.1 ORION Sisteminde Şema Evrimleşmesi 20
3.3.2 ORION'da Bileşik Nesneler 21
3.3.3 Değerlendirme 22
3.4 Nesne Göç Ettirme 22
3.4.1 Nesne Göç Ettirme İle Rol Desteği 23
3.4.2 Değerlendirme 26
3.5 Bileşik Nesne Modelleri 26
3.5.1 Değerlendirme 27
3.6 Özne Tabanlı Programlama 27
3.6.1 Değerlendirme 30
3.7 Bakış Tabanlı Programlama 30
3.7.1 Değerlendirme 32

4. DİNAMİK SİSTEMLERİN MODELLENMESİ İÇİN ÖNERİLEN TASARIM KALIPLARI 33
4.1 Rol Alt Tipleri (Role Subtype) 34
4.1.1 İçsel Bayrak Yolu ile Rol Alt Tipleri (Role Subtype with Internal Flag) 35
4.1.2 Gizli İletim Yolu ile Alt Rol Tipleri (Role Subtype with Hidden Delegate) 37
4.1.3 Durum Nesnesi ile Alt Rol Tipleri (Role Subtype with State Object) 38
4.1.4 Hangi Alt Rol Tipi Kalıbı? 40
4.1.5 Açık Rol Tipi İşareti (Explicit Type Method) 40
4.2 Rol Nesneleri (Role Object) 41
4.2.1 Parametreli Rol Tipi İşareti (Parameterized Type Method) 41
4.2.2 Roller Arasındaki Kuralların Yürütülmesi 43
4.3 Rol İlişkileri (Role Relationship) 43
4.4 İncelenen Kalıpların Karşılaştırılması 45

5. ROL MODELLERİ VE ROL TABANLI PROGRAMLAMA 46
5.1 Giriş ve Tanımlar 46
5.2 Rollerin Temel ve İleri Düzey Özellikleri 47

5.2.1 Rollerin Temel Özellikleri 47
5.2.1.1 Bakış Açısı 47
5.2.1.2 Rol Hiyerarşisi Kavramı 47
5.2.1.3 Rol Kazanımı ve Terki 50
5.2.1.4 Birden Fazla Nesne ile Gösterim 50
5.2.1.5 Rol Geçişi 50
5.2.1.6 Sahibi Üzerinden Üye Erişimi 51
5.2.1.7 Çakışma Engeli 51
5.2.1.8 Sınıf Düzeyi Kalıtım ile Nesne Düzeyi Kalıtımın Birlikte Kullanımı 51
5.2.1.9 Rol Varlığı Kontrolü 52
5.2.1.10 Nitelikli Roller 52
5.2.2 Rollerin İleri Düzey Özellikleri 52
5.2.2.1 Aykırı Rol İlişkilerinin Önlenmesi 53
5.2.2.2 Kalıcılık 53
5.2.2.3 Rol Aktarımı 54
5.2.2.4 Nesne Düzeyinde Çoklu Kalıtımın Desteklenmesi 54
5.2.2.5 Ad ile Üye Erişimi 54
5.2.2.6 Baskın Roller 55
5.2.2.7 Rollerin Askıya Alınması ve Askıdan İndirilmesi 55
5.2.2.8 Danışma ve Vekalet Mekanizmalarının Birlikte Kullanımı 56
5.3 Neden Rol Modelleri? 56
5.4 Yazılım Kalite Kriterleri ve Rol Modellerinin Katkıları 57
6. LİTERATÜRDEKİ ROL MODELLERİ 60
6.1 Fibonacci 60
6.1.1 Fibonacci Nesne Mekanizması ve Kullanım Şekilleri 60
6.1.2 Değerlendirme 63
6.2 Smalltalk ile Rol Hiyerarşisi Yaklaşımı 63
6.2.1 SmallTalk'ta Rol Tanımı 63
6.2.2 Sınıf ve Rol Hiyerarşilerinin Birleştirilmesi 67
6.2.3 Değerlendirme 68
6.3 INADA 69
6.3.1 INADA'nın (ODMG ve C++'a) Eklentileri ve Tip Erişimi 69
6.3.2 Tip Tanımları 71
6.3.3 Değerlendirme 72
6.4 Chameleon 72

6.4.1 Gerçekleme Detayları ve Örnek Uygulamalar 73
6.4.2 Değerlendirme 76
6.5 DEC-JAVA 78
6.5.1 Dec-Java'nın Yapısı 78
6.5.2 Değerlendirme 80
6.6 Geliştirilmiş Rol Modeli 81
6.6.1 Rol Anormalliği Nedir? 82
6.6.2 GRM Yapısı 82
6.6.3 Değerlendirme 84
6.7 Java ile Rol Hiyerarşisi Yaklaşımı 85
6.7.1 Gerçekleme Detayları ve Kullanım Örnekleri 85
6.7.2 Değerlendirme 87

7. JAVA DİLİNE ÖZGÜ YETENEKLER 88
7.1 Dinamik Tipleme ve Tip Dönüşümü 88
7.2 Serileştirme (Serialization) 89
7.2.1 Genel Bilgiler 89
7.2.2 Ek Bilgiler 90
7.3 Yansıtma (Reflection) 91
7.3.1 Genel Bilgiler 92
7.3.2 Yansıtma Kullanımı İçin Bilgi Toplanması 93
7.3.3 `java.lang.reflect.Field` Sınıfı 95
7.3.4 `java.lang.reflect.Method` Sınıfı 96
7.3.5 `java.lang.reflect.Constructor` Sınıfı 97
7.4 Şifreleme 98
7.4.1 Ön Hazırlıklar 98
7.4.2 Açık Dosyanın Şifrelenmesi ve Şifrelenmiş Dosyanın Açılması 100
7.5 Uygulama Ayarlarının Saklanması 101

8. GELİŞTİRİLEN ROL MODELİ: JAWIRO 103
8.1 JAWIRO'nun Kullanıcı Arayüzü 103
8.1.1 JAWIRO ile Rollerin Temel Özelliklerinin Kullanımı 107
8.1.1.1 Rol Kazanımı ve Terki ile Rol Hiyerarşilerinin Kurulması 107
8.1.1.2 Nitelikli Roller 108
8.1.1.3 Rol Varlığı Kontrolü ve Rol Geçişi 109
8.1.1.4 Ebeveynlere Erişim 111
8.1.1.5 Sınıf ve Nesne Düzeyi Kalıtımın Birlikte Kullanımı 111
8.1.2 JAWIRO ile Rollerin İleri Düzey Özelliklerinin Kullanımı 112
8.1.2.1 Rollerin Askıya Alınması ve Askıdan İndirilmesi 112

8.1.2.2 Rol Aktarımı 113
8.1.2.3 Aykırı Rol İlişkilerinin Önlenmesi 114
8.1.2.4 Ad ile Üye Erişimi 117
8.1.2.5 Baskın Roller 120
8.1.2.6 Nesne Düzeyinde Çoklu Kalıtım 121
8.1.2.7 Danışma ve Vekalet Mekanizmalarının Birlikte Kullanımı 122
8.1.2.8 Kalıcılık 125
8.2 Gerçekleme Detayları 132
8.2.1 Hiyerarşi Yapısının Modellenmesi 133
8.2.2 `RoleInternals` Arayüzü 135
8.2.3 Rol Kazanımı, Terki ve Aktarımı 136
8.2.4 Hiyerarşide Arama İşlemleri 138
8.2.4.1 Özyineli Arama 138
8.2.4.2 Derinlemesine Arama 140
8.2.5 Ad ile Üye Erişimi 141
8.2.6 Rol Araması, Rol Geçişi ve Ad ile Üye Erişimi İçin İyileştirmeler 143
8.2.7 Rollerin Askıya Alınması ve Askıdan İndirilmesi 145
8.2.8 Danışma ve Vekalet Mekanizmaları 147
8.2.9 Kalıcılık 147
8.3 Başarım Ölçümleri 148
8.3.1 Genel Bilgiler 148
8.3.2 Rollerin Temel Özelliklerinin JAWIRO ile Başarımı 150
8.3.3 Rollerin İleri Düzey Özelliklerinin JAWIRO ile Başarımı 153
8.4 Örnek Uygulama 159
8.4.1 Modellenen Sistem 159
8.4.2 Kullanıcı Arayüzü 161
8.4.3 Gerçekleme Detayları 161
8.4.4 Rollerin Kullanımı 163
8.4.4.1 Rollerin Askıya Alınması ve Askıdan İndirilmesi 163
8.4.4.2 Aykırı Rol İlişkilerinin Önlenmesi 164
8.4.4.3 Rol Aktarımı, Nitelikli Roller ve Kısıtlama Yöneticisi 165
8.4.4.4 Nesne Düzeyinde Çoklu Kalıtım, Nitelikli Roller ve Ad İle Üye Erişimi 168
8.4.4.5 Kalıcılık 174
8.4.4.6 Rollerin Temel Özellikleri 175

9. JAWIRO'NUN GÜNCEL ROL MODELLERİ İLE KARŞILAŞTIRMASI **176**
9.1 Gerçeklenen Özellikler Açısından Karşılaştırmalar 176
9.2 Çalışma Zamanı Başarımı Açısından Karşılaştırmalar 179

10. SONUÇLAR VE TARTIŞMA **186**

KAYNAKLAR **191**

ÖZGEÇMİŞ **195**

KISALTMALAR

ÇES	**:** Çoklu etmen sistemleri
NYP	**:** Nesneye yönelik programlama.
RBAC	**:** Role based access control.
RTP	**:** Rol tabanlı programlama.

TABLO LİSTESİ

Sayfa No

Tablo 3.1 : Nesne yüzleri ile Java arayüzlerinin karşılaştırılması 14
Tablo 6.1 : Rol sınıfının işlevleri 64
Tablo 6.2 : Nitelikli rollerin desteklenmesi için ek metotlar 67
Tablo 8.1 : RoleInterface arayüzünün kullanıcıya yönelik metotları 104
Tablo 8.2 : Actor sınıfının kullanıcıya yönelik metotları 105
Tablo 8.3 : Role sınıfının kullanıcıya yönelik metotları 106
Tablo 8.4 : JAWIRO ile rollerin temel özelliklerinin AMD sistemdeki başarımı 150
Tablo 8.5 : JAWIRO ile rollerin temel özelliklerinin Intel sistemdeki başarımı 151
Tablo 8.6 : AMD ve Intel sistemdeki başarımların karşılaştırılması (temel özellikler)..... 152
Tablo 8.7 : Intel sistem üzerinde rollerin temel özelliklerinin başarımına hiyerarşi derinliğinin etkisi..... 152
Tablo 8.8 : AMD sistem üzerinde rollerin temel özelliklerinin başarımına hiyerarşi derinliğinin etkisi..... 153
Tablo 8.9 : JAWIRO ile rollerin ileri düzey özelliklerinin Intel sistemdeki başarımı 155
Tablo 8.10 : JAWIRO ile rollerin ileri düzey özelliklerinin AMD sistemdeki başarımı 156
Tablo 8.11 : AMD ve Intel sistemdeki başarımların karşılaştırılması (ileri özellikler)..... 157
Tablo 8.12 : Intel sistem üzerinde rollerin ileri düzey özelliklerinin başarımına hiyerarşi derinliğinin etkisi 157
Tablo 8.13 : AMD sistem üzerinde rollerin ileri düzey özelliklerinin başarımına hiyerarşi derinliğinin etkisi 158
Tablo 9.1 : Roller ile ilgili güncel çalışmaların ilk grubunun, rollerin desteklenen özellikleri açısından karşılaştırması..... 177
Tablo 9.2 : Roller ile ilgili güncel çalışmaların ikinci grubunun, rollerin desteklenen özellikleri açısından karşılaştırması..... 178
Tablo 9.3 : Intel sistem üzerinde yapılan başarım karşılaştırması 180
Tablo 9.4 : AMD sistem üzerinde yapılan başarım karşılaştırması 181
Tablo 9.5 : Schrefl'in rol modeline hiyerarşi derinliğinin etkisi 183
Tablo 9.6 : Rol ilişkileri tasarım kalıbına hiyerarşi derinliğinin etkisi 184

ŞEKİL LİSTESİ

Sayfa No

Şekil 3.1 : Person sınıfı için bir soyut tip ve gerçeklemesi 15
Şekil 4.1 : Tasarım kalıpları ile modellenecek örnek sistemin kavramsal şeması..... 33
Şekil 4.2 : Rol alt tiplerinin kullanımı ile modellemeyi örnekleyen UML şeması..... 34
Şekil 4.3 : Rol alt tipi kullanımı kalıplarına ait örnekler için ortak arayüz..... 35
Şekil 4.4 : Satıcı tipinin içsel bayraklar ile gerçeklenmesi 36
Şekil 4.5 : Çok biçimli metotların içsel bayraklar ile gerçeklenmesi 36
Şekil 4.6 : Yönetici tipinin gizli iletim yolu ile gerçeklenmesi..... 37
Şekil 4.7 : Satıcı arayüzünü gerçeklemek üzere bir şahıs sınıfının durum nesnesi kalıbı ile kodlanması 38
Şekil 4.8 : Durum nesnesi kalıbı ile satıcı rolü için gizli üst ve alt rol sınıflarının tanımlanması 39
Şekil 4.9 : Durum nesnesi kalıbı ile satıcı rolünün `payAmount` metodunun gerçeklenmesi 40
Şekil 4.10 : Rol nesnelerinin kullanımı ile modellemeyi örnekleyen UML şeması..... 41
Şekil 4.11 : Rol nesneleri kalıbı ve parametreli işaretleme kullanılan örnek kök ve rol sınıfları..... 42
Şekil 4.12 : Örnek kök ve rol nesnelerinin kullanımı 42
Şekil 4.13 : Rol nesneleri kalıbında roller arasındaki kurallarının korunmasına örnek. 43
Şekil 4.14 : Rol ilişkileri kullanımı ile modellemeyi örnekleyen UML şeması..... 44
Şekil 4.15 : Rol ilişkileri kalıbı kullanılan kök ve rol sınıfları..... 44
Şekil 5.1 : Örnek bir rol hiyerarşisi. 49
Şekil 6.1 : Bir Fibonacci nesnesinin iç yapısı 61
Şekil 6.3 : Nesnenin çalışma anında yeni rol kazanması 62
Şekil 6.4 : Eklenen tüm rollerle `john` nesnesinin davranışı 62
Şekil 6.5 : Rol sınıfları ve nesnelerinin tanımlanması 64
Şekil 6.6 : Rol nesneleri kullanımına örnekler. 65
Şekil 6.7 : Rol varlığı ve varlık eşitliği kontrolü. 65
Şekil 6.8 : Roller arasında geçişler. 66
Şekil 6.9 : Rol ve sınıf hiyerarşilerinin birlikteliği ile bir gerçek dünya sisteminin modellenmesi..... 66
Şekil 6.10 : Kalıcı bir gerçek dünya nesnesinin oluşturulması 69
Şekil 6.11 : Bir nesneye rol kazandırılması 70
Şekil 6.12 : Örnek sınıf tanımlamaları 71
Şekil 6.13 : Bir rol hiyerarşisi ve rol yöneticileri. 74
Şekil 6.14 : Bir Chameleon nesnesinde metot çağırımı. 75
Şekil 6.15 : Chameleon'da rol tanımlamasına örnekler 76

Şekil 6.16 : Chameleon'da rol kullanımına örnekler 76
Şekil 6.17 : (a) Chameleon ile modellenemeyen bir durum (b) Aynı durumun ilişkisel rol hiyerarşileri yaklaşımı ile modellenmesi mümkündür..... 77
Şekil 6.18 : İç içe donatım..... 79
Şekil 6.19 : GRM'nin diğer çalışmaların rol sistemlerine yaklaşımlarından farklılığı. 81
Şekil 6.20 : GRM'nin katmanları..... 83
Şekil 6.21 : Schrefl'in rol modelinin UML şeması 85
Şekil 6.22 : Schrefl'in rol modelini kullanan örnek sınıflar..... 86
Şekil 6.23 : Örnek sınıfları kullanan örnek kod 87
Şekil 7.1 : Çalışma anında tipleme ve tip dönüşümüne örnekler. 88
Şekil 7.2 : Bir nesneyi serileştirerek kalıcı ortama yazma ve kalıcı ortamdan okuma. Aykırı durum işleme kodları gösterilmemiştir. 90
Şekil 7.4 : Düz metin kaynak akısının şifrelenmesi 101
Şekil 7.5 : Şifreli kaynak akısının açılması 101
Şekil 8.1 : JAWIRO'nun kullanıcı arayüzü ve bazı önemli üyeleri..... 103
Şekil 8.2 : Örneklerde kullanılan iki rol hiyerarşisi 106
Şekil 8.3 : Rol hiyerarşisi kurulması işlemleri 107
Şekil 8.4 : Rol hiyerarşisi kurulması işlemleri 109
Şekil 8.5 : Rol varlığı kontrolü ve rol geçişi 110
Şekil 8.6 : Rol varlığı kontrolü ve kullanımının diğer yolu. 110
Şekil 8.7 : Sınıf ve nesne düzeyinde kalıtımın birlikte kullanımı 111
Şekil 8.8 : Rollerin askıya alınması ve askıdan indirilmesinin ilk yolu..... 112
Şekil 8.9 : Rol aktarımı 114
Şekil 8.10 : Kontrol mekanizmasına ait UML şeması 115
Şekil 8.11 : Kısıtlayıcı sınıfların gerçeklemesi gereken arayüz..... 116
Şekil 8.12 : Bir kısıtlayıcı sınıf örneği 116
Şekil 8.13 : Ad ile üye erişimi örneği 118
Şekil 8.14 : Bir rol hiyerarşisi üzerinde baskınlık mekanizmasının örneklenmesi..... 120
Şekil 8.15 : Nesne düzeyi çoklu kalıtım sonucu oluşan tipleme belirsizliğini çözmek amacıyla ad ile üye erişimi yeteneğinin kullanılması 121
Şekil 8.16 : Danışma ve vekalet mekanizmaları 122
Şekil 8.17 : Örnek rol hiyerarşisi ve iki olası rol geçişi (i. ve ii.) 123
Şekil 8.18 : Örnek rol hiyerarşisindeki sınıfların kodu ve `self` üyesinin kullanımı 124
Şekil 8.19 : Vekalet ve danışma mekanizmalarının birlikte kullanımı 125
Şekil 8.20 : Kalıcı ortama saklanacak bir rol hiyerarşisi ve ilgili bilgi dosyasının açık hali..... 128
Şekil 8.21 : Örnek hiyerarşiyi kalıcı ortama kaydeden kod 129
Şekil 8.22 : Kalıcılık eylemi ardından nesnelerin bellekteki ve kalıcı ortamdaki durumları ile aralarındaki bağlantılar 129
Şekil 8.23 : Kalıcılık eylemi sırasındaki mesaj akışları 130
Şekil 8.24 : Örnek hiyerarşiyi kalıcı ortamdan geri yükleyen kod 131
Şekil 8.25 : Gerçeklenen rol modelinin iç yapısı 132
Şekil 8.26 : Bir rol hiyerarşisi ve içsel gösterimi 134
Şekil 8.28 : Örnek uygulamanın kullanıcı arayüzü 160
Şekil 8.29 : Örnek uygulamada doktor ve hasta ilişkileri 161
Şekil 8.30 : Örnek uygulamada alım satım ilişkileri hakkında özet bilgi 162
Şekil 8.31 : Bir doktorun hastalanmasını modelleyen kod 163

Şekil 8.32 : Hasta doktorun iyileşmesini modelleyen kod 164
Şekil 8.33 : Doktor rolünün terkinin önlenmesi 165
Şekil 8.34 : Kısıtlama yöneticisi ile baş hekim rollerinin otomatik olarak yedek doktora aktarılması 166
Şekil 8.35 : Kısıtlama yöneticisi ile hastalıktan dönen doktora baş hekim rolünün geri aktarılması 167
Şekil 8.36 : Olağan dışı işlemlere yol açmamak için önceden rol geçişi yapılması 168
Şekil 8.37 : Örnek uygulamada alım satım ilişkileri hakkında detaylı bilgi 168
Şekil 8.38 : Mal oluşturma ekranı 170
Şekil 8.39 : Mal alım-satım ekranı 171
Şekil 8.40 : Alım-satım işlemi öncesi uygulamanın yaptığı işlemler 172
Şekil 8.41 : Müşteri rolünün satın alma metodu 173
Şekil 8.42 : Satıcı rolünün satış metodu 173
Şekil 9.1 : Rollerin temel özelliklerinin sabit hiyerarşi derinliğinde çalıştırılması ile elde edilen başarım sonuçları 182
Şekil 9.2 : Rollerin temel özelliklerinin sabit hiyerarşi derinliğinde çalıştırılması ile elde edilen başarım sonuçları 185

NESNEYE YÖNELİK PROGRAMLAMAYA ROL DESTEĞİNİN KAZANDIRILMASI

ÖZET

Gerçek dünyada karşılaşılan sistemler durağan ve dinamik olmak üzere iki kümeye ayrılabilir. Durağan sistemlerde modellenecek varlıklar ayrık sınıflara bölünebilir ve sınıflarını asla değiştirmezler. Nesneye yönelik programlamada da (NYP) bir nesne ve ait olduğu sınıf arasındaki ilişki kalıcı, durağan ve dışlamalı bir yapıda olduğu için; durağan sistemler NYP ile kolayca modellenebilir. Dinamik sistemler ise zamanla değişen ve evrimleşen varlıklardan oluşur. Böylesi nesneleri NYP'nin sınıf hiyerarşileri ile modellemek zor olacaktır. Bu zorluğun nedeni, dinamik yapıdaki gerçek dünya varlıklarının durağan yapıdaki sınıflar ile modellenmeye çalışılmasıdır. NYP durağan sınıf yapısına dinamizm katacak yeteneklere sahip olmasına rağmen dinamik sistemlerin modellenmesinde karşılaştığı güçlüklerden tam anlamıyla kurtulamaz. Bu tez çalışmasının amacı da, dinamik sistemlerin yazılım kalite ölçütlerine göre iyi bir şekilde modellenebilmesi için bir NYP diline rol desteğinin kazandırılmasıdır.

Dinamik olarak evrimleşen varlıkları modellerken nesne düzeyinde özelleştirme yapılması, sınıf düzeyinde özelleştirmenin dinamik sistemlerin modellenmesinde karşılaştığı güçlükleri çözecektir. Rol modelleri ise NYP'ye nesne düzeyinde özelleştirme yeteneği kazandıran bir yaklaşımdır. Bu durumda bir gerçek dünya varlığı, her biri yükümlü olduğu görevlerden birini yürüten bir rol nesnesi olmak üzere, birden fazla nesne ile temsil edilir. Bir nesnenin bir rolü, bu nesnenin diğer nesnelerin beklediği gibi davranabilmesi için gereken bir özellikler kümesi olarak tanımlanabilir. Rol kullanımı temeline dayanan programlama biçimine rol tabanlı programlama (RTP), rollerin tasarımı için bir tarz ve rollerin kullanımı için çeşitli işlevler sunan yazılımlara ise rol modeli adı verilir. Rol modelleri nesne düzeyinde özelleştirme için bir yol sunarken bir gerçek dünya varlığını modellemek için birden fazla nesne kullanıldığı gerçeğinin korunmasını sağlar.

Dinamik sistemlerin etkili ve kolay bir biçimde modellenmesi için önerilen yollar arasında rol modelleri; kullanışlı olmaları, NYP kavramları ile iyi uyuşmaları ve problemin çözümü için dolaysız bir yol sunmaları nedeniyle dikkati çekmektedir. Nesne düzeyinde özelleştirmeye dayanan rol modelleri, sınıf düzeyinde özelleştirmeye dayanan NYP'yi doğal bir biçimde genişletir. Bu nedenle tez çalışmasının çözmeyi hedeflediği problem olan dinamik sistemlerin saf NYP yaklaşımı ile sunulandan daha iyi modellenebilmesi gereksinimini karşılamak üzere, Java programlama diline rol desteği kazandıran bir rol modeli olan JAWIRO (Java with Roles) gerçeklenmiştir.

Nesne düzeyinde özelleştirme, sınıf düzeyinde özelleştirmeyi genişlettiğinden dolayı, rol modelleri tüm NYP dillerinin felsefesine uygundur ve herhangi bir NYP dili ile gerçeklenebilir. Tez çalışmasında Java dilinin seçilmesinin ise iki nedeni bulunmaktadır: Öncelikle Java dilinin Bölüm 7'de incelenen yeteneklerinin, özellikle de yansıtma yeteneğinin, bir rol modelinin gerçeklenmesinde önemli kolaylıklar sağlayacağı düşünülmüştür. İkinci neden ise, Java dilinin diğer NYP dillerine göre çok daha geniş bir sınıf kitaplığına sahip olmasına rağmen dinamik sistemlerin modellenmesi için gereçlere sahip olmamasıdır.

Değişik araştırmacılar, rollerin sahip olması gereken özellikleri birbirlerinden farklı şekilde tanımlamışlardır. Tez çalışması boyunca yapılan araştırmalar ve değerlendirmeler sonucunda, bir rol modelinin sağlaması gereken özellikler bu tez çalışmasında temel ve ileri düzey olmak üzere ikiye ayrılmıştır. Rollerin temel özellikleri şu şekilde sıralanabilir:

- Bakış açısı: Bir gerçek dünya varlığının oynadığı rollerin her biri, o varlığa bir bakış açısı oluşturur. Bir nesneye erişim o anda kullanılan rolü ile sınırlıdır. Ancak rol geçişi sayesinde, mevcut rol içerisinden başka bir rolün üyelerine erişim sağlanılabilir.

- Rol hiyerarşisi: Bir rol (üst rol) başka bir rolü (alt rol) oynayabilmelidir. Bu da rollerin çeşitli hiyerarşik düzenler oluşturacak şekilde birbirleri ile ilişkilendirilebilmeleri anlamına gelecektir.

- Rol kazanımı ve terki: Roller birbirlerinden bağımsız olarak kazanılabilmeli ve terk edilebilmelidir.

- Birden fazla nesne ile gösterim: Bir gerçek dünya nesnesinin sahip olduğu rollerin tümü ile tanımlandığı gerçeği korunmalıdır. Bu amaçla her rol nesnesi sahibinden, kendi rollerinden ve hiyerarşi kökünden haberdar olmalıdır.

- Rol geçişi: Bir varlık herhangi bir rolüne herhangi bir anda geçiş yapabilmelidir.

- Sahibi üzerinden üye erişimi: Bir nesnenin bir rolü, diğer rollerinin üyelerine, üstteki iki özellikten biri sayesinde erişebilmelidir.

- Çakışma engeli: Değişik roller bir çakışma olmaksızın aynı adlı üye metot ve değişkenlere sahip olabilmelidir.

- Sınıf düzeyi kalıtım ile nesne düzeyi kalıtımın birlikte kullanımı: Nesne düzeyi kalıtım sınıf düzeyi kalıtımın yerini almaz. Aksine, nesne düzeyi kalıtım sınıf düzeyi kalıtımı tamamlar. Gerçek dünyada birlikte görülebilen bu iki tür kalıtım ilişkisi rol modellerinde de birlikte kullanılabilmelidir.

- Rol varlığı kontrolü: Varlıklar herhangi bir rol tipini oynayıp oynamadıkları hakkında sorgulanabilmelidir.

- Nitelikli roller: Bir varlık aynı rol tipinin birden fazla örneğine sahip olabilmelidir. Böyle rollere bu çalışmada nitelikli rol adı verilmiştir. Nitelikli roller birbirlerinden bir niteleyici ile ayrılırlar.

Rollerin ileri düzey özellikleri de aşağıda listelenmiştir. Bu özelliklerden son altı tanesi tez çalışmasının özgün katkılarını oluşturmaktadır:

- Aykırı rol ilişkilerinin önlenmesi: Bir yazılımı hazırlayanlar ile kullananların farklı kişiler olmasından kaçınılamaz. Bu durumda kullanıcıların sistemin yapısına aykırı işler yapmaları olasılığı artacaktır. Anormal ilişkilerin önlenmesi bu nedenle bir zorunluluk olarak karşımıza çıkar.

- Kalıcılık: Kalıcılık yeteneğine sahip bir rol modeli bütün bir rol hiyerarşisini kalıcı bir ortama saklayabilir ve sonradan tekrar kullanmak için kalıcı ortamdan geri yükleyebilir.

- Rol aktarımı: Bir rol mevcut sahibinden bir başka sahibe, alt rollerini kaybetmeden aktarılabilmelidir.

- Nesne düzeyinde çoklu kalıtım: Bir rol örneği farklı sınıfların rol örnekleri tarafından oynanabilmelidir.

- Ad ile üye erişimi: Bir rol hiyerarşisinde bulunan nesnelerden herhangi birisinin istenilen bir üye değişkeni veya metoduna, sahibine doğrudan bir referans olmadan, sadece o üyenin adı belirtilerek erişilebilir. Aynı ada sahip birden fazla üye varsa, en çok özelleşmiş üyeye erişim sağlanmalıdır. Bu sayede ilgili role, o rolün tipi bilinmeden erişilebilir ve yazılımın tiplere olan bağımlılığı azaltılmış olur.

- Baskın roller: Önceki kural hiyerarşideki bazı katılımcıların baskın kılınması ile değiştirilebilmelidir.

- Rollerin askıya alınması ve askıdan indirilmesi: Geçici bir süre ihtiyaç duyulmayacak roller askıya alınabilmeli ve gerek duyulduğunda askıdan indirilerek durum bilgisinde kayıp olmadan tekrar kullanılabilmelidir.

- Danışma ve vekalet mekanizmalarının birlikte kullanımı: Rol geçişi sırasında mesajın ilk alıcısının saklanıp saklanmaması ile birbirinden ayrılan bu iki mekanizma gerçek dünya sistemlerinde bir arada görülebileceği için, bir rol modelinde de birlikte kullanılabilmelidirler.

Dinamik sistemlerin etkin olarak modellenmesi için rol modelleri tek çözüm değildir. NYP'nin dinamik sistemlerin modellenmesindeki zayıf noktalarının giderilmesinin farklı yolları önerilmiştir ve bu çalışmalar tez kitabında da incelenmiştir. Ancak yapılan karşılaştırmalar ile, rol modellerinin dinamik sistemlerin modellenmesi için literatürdeki diğer çalışmalardan daha yararlı olacağı ve rollerin kimi özelliklerinin diğer çalışmalar ile gerçeklenemeyeceği gösterilmiştir. Rol modellerinin yazılımın içsel kalite özelliklerini iyileştirerek yazılım maliyetlerini düşüreceği de tez kitabında anlatılmıştır. Tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO ise, sahip olduğu özgün yetenekler ve çalışma anı başarımına ek yük getirmemesi sayesinde dinamik sistemlerin modellenmesi için önemli bir gereç haline gelmiştir.

EXTENDING OBJECT ORIENTED PROGRAMMING WITH ROLE SUPPORT

SUMMARY

Real world systems can be classified either as static or dynamic. In static systems, entities to be modeled can be divided into distinct classes and they do not change their classes. Such systems can be modeled efficiently by using the object oriented programming (OOP) paradigm as the relationship between an object and its class is persistent, static and exclusive in OOP. On the contrary, dynamic systems contain entities which constantly change and evolve. Modeling such entities with the class structure of OOP is hard because objects of dynamic nature are required to be modeled by classes having static nature. Although OOP has features to add some scale of dynamism to the static nature of its classes, it cannot fully eliminate the obstacles it faces while modeling dynamic systems. This thesis aims to extend an OOP language with role support in order to model dynamic systems in a good way according to the software quality criteria.

When modeling dynamically evolving entities, using instance level specialization will eliminate the obstacles faced by the class level specialization. Role models provide a means for instance level specialization. In that case, a real world entity is represented by multiple role objects, each representing one of the responsibilities of the real world entity. A role of an entity can be defined as the set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects. The programming paradigm which is based on using roles is called role based programming (RBP), while a role model is a software which specifies a style of designing and implementing roles. Role models provide a mechanism for object level inheritance while preserving the fact that multiple objects are used to model a real world entity.

Role models receive much attention with their usefulness among the proposed ways of modeling dynamic systems as they match with the OOP paradigm well and represent a direct way for solving the problem at hand. Role models which are based on object level specialization naturally extends the OOP which is based on class level specialization. Therefore, the Java programming language is extended with role support in order to solve the problem of modeling dynamic systems better than the ways available with the pure OOP paradigm, the problem which is targeted by this dissertation. For that purpose, a role model named JAWIRO (Java with Roles) is implemented.

Role based programming is in harmony with the OOP paradigm as the instance level specialization extends the class level specialization. Likewise, a role model can be implemented by any OOP language. Java is chosen as the base language for two reasons: Firstly, Java has advanced features which can be useful for the implementation of a role model. These features are examined in Section 7. The second reason is the absence of tools in Java for modeling dynamic systems although it has an abdundant class library.

Different researchers define the features expected from the roles in slightly different ways. According to the results of the research and evaluations done during this thesis work, the features of roles which are expected to be provided by a role model are divided into two groups as the basic and extended features in this dissertation. The basic features of roles can be listed as below:

- Viewpoint: Each role played by a real world entity provides a viewpoint to that entity. The access scope of an object is restricted by the currently played role. However, access to members of a different role can be provided via role switching.

- Role hierarchies: A role (parent role) should be able to play another role (sub-role). This will lead to the ability to arrange role objects in various hierarchical relationships.

- Gaining and loosing roles: Roles can be gained and lost independently from each other.

- Representation with multiple objects: The fact that a real world entity is modeled with multiple objects should be preserved. Each role object should be aware of its owner, its subroles and the root of its hierarchy for this purpose.

- Role switching: An entity should be able to switch between its roles whenever required.

- Member access via the owner: A role of an entity should be able to reach the members of the other roles of this entity by using either of the previous two features.

- Conflict prevention: Different roles should be able to contain member fields and methods with the same name.

- Use of the object level and class level inheritance together: Object level inheritance does not replace class level inheritance. On the contrary, object level inheritance completes class level inheritance. As these two types of inheritance relationships can be found together in the real world, they should be used together in role models as well.

- Run time role checking: Entities can be queried whether they are playing a particular role type or not.

- Aggregate roles: An entity should be able to play multiple instances of the same role type. Such roles are called aggregate roles and they are distinguished from each other by an identifier.

The advanced features of roles are listed below as well. The last six of these features represent the unique contributions of this dissertation:

- Preventing abnormal role relationships: It's usually inevitable that the coders and users of a piece of software are different people. This will increase the probability of users to commence operations which contrast the nature of the modeled system. The necessity to prevent abnormal role relationships stems from this fact.

- Persistency: A persistent role model lets users to save an entire role hierarchy to persistent storage for later use.

- Role transfer: A role can be transferred to another owner without dropping its subroles.

- Object level multiple inheritance: A role type can be played by instances of different types.

- Member access by name: A desired member field or method of a participant of a role hierarchy can be accessed from any participant of that role hierarchy. This is achieved by solely mentioning the name of the desired member and without any direct reference to the owner of that desired member, reducing the dependency on role types. In case of multiple members with the same name, the member of the most evolved participant is returned.

- Dominant roles: The previous rule can be overridden by determining some participants of the role hierarchy as dominant.

- Suspending and resuming roles: Roles can be suspended when they become temporary unnecessary and resumed when they are needed again.

- Support for delegation and consultation mechanisms: These two mechanisms are distinguished from each other by whether the original recipient of a message is preserved or not. As both mechanisms exist in the real world together, they should coexist in the role models as well.

Role models are not the sole solution for efficient modeling of dynamic systems. Other ways for overcoming the difficulties faced by OOP while modeling dynamic systems are proposed as well. These alternatives are examined in this dissertation, too. However, the comparisons made in this dissertation show that role models are more valuable than the other proposals for modeling dynamic systems. Moreover, it is shown that some features of roles cannot be implemented with those proposals. It is also shown that role models increase the internal quality attributes of software and reduce the costs of producing software. Meanwhile, the role model JAWIRO implemented as a part of this thesis work is surfaced as an important tool for

modeling dynamic systems, thanks to its unique contributions to features of roles and its neglible overhead on runtime performance.

1. GİRİŞ

Bu bölümde öncelikle tez çalışmasının çözmeyi hedeflediği problem olan, dinamik sistemlerin saf NYP yaklaşımı ile sunulandan daha iyi modellenebilmesi gereksinimi incelenecektir. Çözüm önerisi olarak rol modellerinin kısaca tanıtılmasından sonra bu bölüm tez çalışmasının özgün katkılarının özetlenmesi ile sona erecektir.

1.1 Problemin İncelenmesi

Gerçek dünyada karşılaşılan sistemler durağan ve dinamik olmak üzere iki kümeye ayrılabilir. Durağan sistemlerde modellenecek varlıklar ayrık sınıflara bölünebilir ve sınıflarını asla değiştirmezler. Sınıf tabanlı NYP yaklaşımında da, bir sınıf ile o sınıftan türetilmiş bir nesne arasındaki ilişki kalıcı, değişmez ve dışlamalı bir yapıya sahiptir. NYP'nin temellerinden olan sınıf kavramı durağan bir yapıya sahip olduğundan, durağan sistemler NYP ile kolayca modellenebilir.

Durağan sistemlerin aksine, dinamik sistemlerde zamanla değişen ve evrimleşen varlıklar söz konusudur. Buna en güzel örnek insanlardır. Bir insan hayatı boyunca birbirleriyle örtüşen değişik roller alır: Öğrenciliğe başlayan şahıs aynı zamanda yarı zamanlı bir veya birkaç iş sahibi olabilir; aynı anda bir eş, evlat, ebeveyn olmak gibi çeşitli yükümlülükleri yerine getirir.

Bu şekilde zamanla değişen nesneleri NYP'nin sınıf hiyerarşileri ile modellemek zor olacaktır. Bu zorluğun nedeni, dinamik yapıdaki gerçek dünya varlıklarının durağan yapıdaki sınıflar ile modellenmeye çalışılmasıdır. NYP bu durağan yapısına dinamizm katacak yeteneklere sahip olmasına rağmen dinamik sistemlerin modellenmesinde karşılaştığı güçlüklerden tam anlamıyla kurtulamaz. Varlıkların birbirinden farklı rolleri bağımsız olarak alabilmesi güçlükleri daha da büyütmektedir.

Davranışsal sınıflandırma kalıtım ile modellenebilir, ancak birçok dil bir nesnenin kimliğini tek bir tipe kalıcı ve değişmez şekilde bağlar. Eğer dinamik varlıklar

böylesi durağan bir nesneler dizisi ile modellenirse kavramsal varlık ve nesneler arasında doğrudan bir bağlantı kurulamaz. Gerekli bağlantıyı kuracak ve koruyacak yolların bulunması ve gerçeklenmesi, programcıya düşen zor bir iş olacaktır.

Modellenen varlıklar her evrimleştiğinde yeniden sınıflandırılması bir çözüm olabilir ancak bu da hala zor bir iştir. Bu iş söz konusu varlığın eski sınıfına ait nesnenin yok edilip yeni sınıfına ait bir nesnenin oluşturulması şeklinde basitçe tanımlanabilir. Ancak bu eylem, eski nesneye eski sınıf türünden olan tüm işaretçilerin yok edilip yeni sınıf türünden işaretçilere çevrilmesini gerektirir. Büyük çaplı bir sistemde bu aşama çok büyük bir ek yük ortaya çıkaracaktır.

Dinamik sistemleri modellerken çoklu kalıtım kullanılabilir: Öncelikle modellenen sistemdeki her farklı davranış türünü belirleyen sınıflar ayrı ayrı tanımlanır. Bu sistemdeki nesneler birden fazla davranış türü kazanmak yoluyla evrimleşeceğinden, ilgili rollerin her olası birlikteliği için ilgili sınıflardan kalıtım yolu ile ek sınıflar oluşturulması gerekecektir. "Arakesit sınıfları" olarak adlandırılan bu sınıflar genellikle çoklu kalıtım ile elde edilir. Böylece üstel olarak büyüyen bir sınıf hiyerarşisi ağacı oluşur. Üstel olarak artan sınıf sayısına karşılık çoğunlukla hiyerarşinin göreli olarak az sayıda üyesinden örnekler belli bir anda etkindir. Üstelik Java dahil bazı nesneye yönelik dillerde çoklu kalıtım desteklenmemektedir. Böyle dillerde gerekli işlevselliği çok sayıda arakesit sınıfına kazandırmak gayet zor olacaktır.

Çok biçimlilik sadece aynı kalıtım zincirindeki sınıflar arasında gerçekleşebileceği için, kalıtımın yukarıda anlatılan tüm zayıflıklarını da beraberinde getirecektir. Çok biçimlilik, arayüz yapısı sayesinde, aynı kalıtım zincirinde bulunmayan sınıflar arasında da gerçekleştirilebilir fakat gerekli esnekliği tek başına sağlayamaz.

1.2 Çözüm Önerisi: Nesne Düzeyinde Özelleştirme ve Rol Modelleri

NYP'deki kalıtım ilişkisi ile varolan genel bir sınıftan daha özel yeni bir sınıf türetilir. Bu özelleştirme sınıf düzeyinde gerçekleşir; yeni sınıfa ait tüm nesneler, eski sınıfa ait tüm nesnelerden aynı biçimde farklılaşır. Sınıf tabanlı yaygın NYP dillerinin dışında SELF [1] gibi sınıfsız NYP dilleri de bulunmaktadır. Sınıfsız NYP dillerinde kalıtım ilişkisi nesneler arasında kurulur; bir nesne diğerinden kalıtım ile

türetilir. Bu durumda da özelleştirme nesne düzeyinde gerçekleşmektedir. Sınıf düzeyinde özelleştirme, nesne düzeyinde özelleştirmeye göre, aynı tipten tüm nesnelere eş biçimli ulaşım ve nesnelerin daha etkin saklanması avantajlarına sahiptir. Bununla birlikte; evrimleşen varlıkları modellerken nesne düzeyinde özelleştirme yapılması, sınıf düzeyinde özelleştirmeye göre daha iyi bir yaklaşımdır [2]. Dinamik sistemlerde aynı sınıfın farklı örnekleri, farklı varlıklar olarak zaman içinde farklı şekillerde evrimleşeceklerdir. Rol modelleri de tam bu yeteneği sağlayarak önceki bölümde incelenen probleme iyi bir çözüm sunmaktadır.

Dinamik sistemlerin etkili ve kolay bir biçimde modellenmesi için önerilen yollar arasında rol modelleri; kullanışlı olmaları, NYP kavramları ile iyi uyuşmaları ve problemin çözümü için dolaysız bir yol sunmaları nedeniyle dikkati çekmektedir. Nesne düzeyinde özelleştirmeye dayanan rol modelleri, sınıf düzeyinde özelleştirmeye dayanan nesneye yönelik programlamayı doğal bir biçimde genişletir. Bu nedenle bir çok başarılı rol modeli, mevcut bir NYP dilinin genişletilmesi ile elde edilmiştir.

Rol modelleri ile sağlanan nesne düzeyinde özelleşmenin temelinde, bir varlığın zaman içinde farklı roller kazanabilmesi ve kaybedebilmesi yer almaktadır. Bu durumda bir gerçek dünya varlığı belli bir anda çeşitli roller oynuyor olacaktır. Rol modellerinde çoğu zaman aynı türden roller, aynı sınıfla modellenir. Söz konusu sınıfa ilgili rolün sınıfı veya türü adı verilir. Bazı rol modelleri ise bir varlığın bazı rollerinin de başka rolleri oynayabilmesini mümkün kılar. Bir rol diğerini oynuyorsa, oynayan konumundaki role üst rol, oynanan konumundaki role ise alt rol denilmektedir.

1.3 Tezin Amacı ve Özgün Katkıları

Bu tez çalışmasının amacı, Java dilinin rol desteği ile genişletilmesi için bir rol modeli tasarlamak ve gerçeklemektir. Oluşturulacak rol modelinin, rollerden beklenen özelliklerin tümünü herhangi bir kısıtlama olmaksızın gerçeklemesi ana hedeftir. Diğer hedefler ise rol modelinin basit ve esnek kullanıma sahip olması, metot çalıştırma zamanına düşük ek yük getirmesi ve geliştirilmeye açık olmasıdır.

Tez çalışması kapsamında gerçeklenen rol modeline JAWIRO (Java with Roles) adı verilmiştir. Kullanılacak dil olarak Java'nın seçilmesinin nedeni, Java'nın geniş kullanım kazanmasını sağlayan ileri düzey yeteneklere sahip olmasına rağmen rollerin kullanımı için doğal bir desteğe sahip olmamasıdır. Ayrıca Java dilinin 7. Bölüm'de anlatılan yeteneklerinin rollerin ileri düzey özelliklerinin gerçeklenmesi için önemli bir destek sağlayacağı düşünülmüştür. Sonuçta ortaya çıkan rol modeli Java diline rol desteği kazandırarak dinamik sistemlerin etkin bir biçimde modellenmesini sağlarken, önceden belirlenen amaçlara da sadık kalmıştır.

Literatürde rollerin ileri düzey özelliklerinin tümünü birden gerçekleyen hiçbir rol modeli bulunmamaktadır. JAWIRO çeşitli diller için oluşturulmuş rol modellerinin bir kısmı diğerinde bulunmayan yeteneklerinin tümüne sahip olmanın yanısıra, hiçbir rol modelinin sahip olmadığı ek yeteneklere de sahiptir. Aşağıda kısaca özetlenen bu yetenekler daha sonra örneklerle ve ayrıntılı olarak incelenecektir.

1.3.1 Nesne Düzeyinde Çoklu Kalıtım

Tez çalışmasının ilk özgün katkısı önerilen rol modelinin nesne düzeyinde çoklu kalıtımı desteklemesidir. Bu sayede, modellenen sistem gerektirdiği takdirde, aynı rol sınıfına ait nesneler farklı tiplerden sahipler tarafından oynanabilmektedir. Bu gereksinimi karşılamak için zor veya masraflı çeşitli yöntemler kullanmak yerine JAWIRO ile doğrudan karşılamak kazançlı olacaktır.

Belirli bir rol örneği aynı anda yalnız bir sahibe ait olabileceği için, nesne düzeyinde çoklu kalıtım mantıksal bir belirsizlik ortaya çıkarmayacaktır. Alt rolün üst role ait bir üyeye erişmesi gereken durumlarda Java tiplemesi açısından ortaya çıkacak belirsizliği çözmek için, üst rollerin aynı arayüzü gerçeklemesi veya erişilecek üyenin ayrı bir role taşınarak o rol üzerinden kullanılması sağlanabilir. Alternatif bir yol olarak JAWIRO'nun bir sonraki bölümde değinilen yeteneği de etkili bir çözüm sunmaktadır.

1.3.2 Rol Aktarımı

Halihazırda bir sahibi olan bir rol nesnesi, alt rollerini kaybetmeden başka bir sahibe aktarılabilir. Eski ve yeni sahip aynı veya farklı rol hiyerarşisinde bulunabilir. Hatta,

nesne düzeyinde çoklu kalıtım desteklendiği için eski ve yeni sahip farklı sınıflardan bile olabilir.

1.3.3 Ad ile Üye Erişimi

JAWIRO'nun diğer bir özgün katkısı sahibinin adını ve tipini belirtmeden üye değişken veya metot erişimi yeteneğidir. Bu sayede bir rol hiyerarşisindeki elemanların herhangi birinin yalnızca adı bilinen üyesine, bu üyenin sahibini belirtmeden erişilebilir. Ad ile üye erişimi yeteneği, nesne düzeyinde çoklu kalıtım kullanıldığı durumlarda, alt rolün tipi ancak çalışma anında belirlenecek üst role erişmesi için kolay ve etkin bir yol sağlar.

Aynı hiyerarşideki birden fazla nesnede aynı adlı üye bulunuyorsa, en çok özelleşmiş (hiyerarşinin en derinindeki) katılımcının üyesine erişim sağlanacaktır. Bu özellik sayesinde, modellenen varlık, sadece aynı metotlara sahip farklı rollerin kazanılıp terk edilmesi ile, aynı olaya değişen şartların gerektirdiği biçimde farklı davranabilecektir.

1.3.4 Rollerin Askıya Alınması ve Askıdan İndirilmesi

Diğer rol modellerinde bulunmayan ancak JAWIRO ile desteklenen bir başka özellik ise, istenilen rollerin geçici olarak askıya alınabilmesi ve şartlar değiştiğinde askıdan indirilebilmesidir. Bir rolün askıya alınması ve askıdan indirilmesi, tüm alt rolleri ile birlikte gerçekleşir. Rolün askıya alınması yerine terk edilmesi, o rolde saklanan bilginin kaybedilmesi anlamına gelecektir. Söz konusu tüm bilginin geçici değişkenlerde saklanmasına dayalı çözümler yerine JAWIRO'nun bu yeteneğinin kullanılması programcıya kolaylık sağlayacaktır.

1.3.5 Baskın Roller

JAWIRO ad ile üye erişimi sırasında en çok özelleşmiş üyeye erişim sağlamakla birlikte, programcı hiyerarşinin bazı üyelerini baskın kılarak bu davranışı değiştirebilir. Böylece değişen koşullara uyum için rol terki ve kazanımına alternatif bir yol daha sunulmuştur. Modellenen sistemin o andaki durumunun rol terki, rol kazanımı ve rolün askıya alınması işlemlerinden biri veya daha fazlasına izin vermediği anlarda, baskın rollerin kullanımı ile değişen şartlara uyum sağlanılabilir.

1.3.6 Danışma ve Vekalet Mekanizmalarının Birlikte Kullanımı

Tez çalışmasının özgün katkılarından biri de, danışma ve vekalet mekanizmalarının ikisinin birlikte kullanımının desteklenmesidir. Bu sayede programcı istediği anda danışma, istediği anda vekalet mekanizmasına geçebilir. İleride ayrıntılı olarak incelenecek bu iki mekanizma, rol geçişi sırasında mesajın ilk alıcısının saklanıp saklanmaması konusunda birbirinden ayrılır: Vekalet mekanizmasında mesajın ilk alıcısı önem taşır ve durum değişiklikleri mesajı ilk alan role yansıtılır. Danışma mekanizmasında ise durum değişiklikleri mesajın son alıcısına yansıtılır. Gerçek dünyada her iki çalışma biçiminin birden kullanımını gerektiren durumlarda, kullanılan gereçlerin de bu esnekliğe sahip olması önemli bir kolaylık sağlayacaktır.

1.3.7 Rol Araması ve Geçişinde İyileştirme

Dinamik sistemlerin doğası gereği, bir varlığın çalışma anında hangi rollere sahip olacağı derleme anında bilinemez. Bu nedenle bir role geçiş yapmadan önce o role sahip olunup olunmadığının sorgulanması yerinde bir davranış olacaktır. Dolayısıyla bir rol sorgusunun ardından çoğu kez o role geçiş isteği gelecektir. Diğer rol modellerinin aksine JAWIRO sorgulanan son role ait bir işaretçi saklar. Takip eden geçiş isteği az önce aranan role aitse, rolün tekrar aranması yerine saklanan işaretçi sayesinde doğrudan rol geçişi yapılır. Tez çalışmasının bu özgün katkısının getirdiği başarım iyileştirmesi ileride ölçümlerle gösterilecektir.

Tez çalışmasının özgün katkılarının yanı sıra, diğer rol çalışmalarında yer alan özgün katkıların tümü de JAWIRO ile kullanılabilmektedir. Söz konusu yetenekler çalışma anında rol varlığı kontrolü, aykırı rol ilişkilendirmelerini önleme ve kalıcılıktır.

Tez çalışmasının özgün katkıları, gerçeklenen rol modelinin tüm yetenekleri ile birlikte daha ileride ayrıntılı olarak incelenecektir. Tez kitabının geri kalanında önce ikinci bölümde nesneye yönelik programlama hakkında temel bilgiler özetlenecek, ardından üçüncü bölümde dinamik sistemlerin modellenmesine yönelik bazı çalışmalar incelenecektir. Dördüncü bölümde rollerin etkin kullanımı için öne sürülen bazı tasarım kalıpları incelenecektir. Beşinci bölümde rol modelleri hakkında genel bilgi verilecek, altıncı bölümde ise literatürdeki rol modelleri incelenecektir. JAWIRO'nun gerçeklenmesinde kullanılan Java dilinin ileri düzey yetenekleri

yedinci bölümde örneklerle anlatılacaktır. Gerçeklenen rol modeli sekizinci bölümde detaylı olarak anlatıldıktan sonra sonuçlar bölümü ile tez kitabı son bulacaktır.

1.3.8 Sonuç

Dinamik sistemlerin etkin olarak modellenmesi için rol modelleri tek çözüm değildir. Çeşitli tasarım kalıpları da dahil olmak üzere, NYP'nin dinamik sistemlerin modellenmesindeki zayıf noktalarının giderilmesinin farklı yolları önerilmiştir. Bölüm 9'daki karşılaştırmalar incelendiği zaman, rol modellerinin dinamik sistemlerin modellenmesi için literatürdeki diğer çalışmalardan daha yararlı olacağı anlaşılacaktır. Tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO ise, sahip olduğu özgün yetenekler ve çalışma anı başarımına ek yük getirmemesi sayesinde dinamik sistemlerin modellenmesi için önemli bir gereç haline gelmiştir.

2. NYP TEMEL KAVRAMLARI

Rol modelleri nesneye yönelik programlamayı rol desteği ile genişleten çalışmalar olduğu için, tez çalışmasının bütünlüğü açısından NYP kavramlarının çok iyi anlaşılması zorunludur. Değinilecek NYP kavramlarının amaçları olarak şunlar verilebilir:

- Programcıya tutarlı bir arayüz sağlayarak gerçeklemeyi kolaylaştırmak,
- Daha kolay anlaşılabilir, hata ayıklaması yapılabilir ve geliştirilebilir programlar yazmak,
- Aynı sınıfları birden fazla programda kullanarak yazılım geliştirme maliyetlerini azaltmak.

2.1 Nesne

Kendini tanımlayan veriden ve yapabileceği eylemlerden oluşan bir programlama varlığıdır. Bir nesne kendi verisini nasıl kullanacağını bilir [3].

Her nesne tekil tanımlayıcı olan bir kimliğe sahiptir. Bu kimlik nesnenin adından ve durumundan bağımsızdır.

Bir nesne diğer nesnelerden gelen mesajlara yanıt verir. Bir mesaj nesnenin yapacağı eylem(ler) için bir tanımlayıcı ve eylemin gerçekleştirilmesi için gerekli ek bilgileri içerir. Bu nedenle mesajlar için "nesnenin dış dünyaya açılan penceresini oluşturur" diyebiliriz [3].

2.2 Sınıf

Nesneleri oluşturmakta kullanılan şablonlara sınıf adı verilir. Bir sınıf; nesneyi tanımlayan verileri ve nesnenin nasıl yerine getireceğini bildiği eylemlerin tanımlarını içerir. Söz konusu eylemlere sınıf metotları veya üye fonksiyonlar adı verilir [3]. Bir sınıfa ait nesne, ya da bir başka deyişle o sınıfın bir örneği ilk

oluşturulduğu anda çalıştırılan metoda kurucu adı verilir. Söz konusu örnek yok edileceği zaman çalışan metoda ise sonlandırıcı adı verilir.

2.3 Çok Anlam Yükleme (Overloading)

Üye metotları birden fazla şekilde tanımlamak yoluyla aynı adlı metodun nasıl kullanıldığına göre farklı biçimlerde davranmasını sağlama tekniğidir [3]. Bu tekniği kullanmanın yolu aynı adlı üye metodu farklı parametre listeleriyle birden fazla kez tanımlamaktan geçer. Derleyiciler bu tekniğin kullanımını sağlamak için metotları sadece adları ile değil, tüm imzası ile ve sınıf bazında ayırt eder. Sadece metotlara değil, operatörlere de birden çok anlam yüklenebilir. Ancak bunun için kullanılan dile özel kurallar bulunabilir.

2.4 Kalıtım

Kalıtım benzer nesnelerin üye değişken ve metotları paylaşmasını sağlar ve çok biçimliliğe olanak verir. Kalıtım ile varolan genel bir sınıftan daha özel başka bir sınıf türetilir. Kalıtım ilişkisinde genel sınıfa üst, taban veya ebeveyn sınıf, özel sınıfa ise alt veya çocuk sınıf adı da verilmektedir. Derleyici, üst sınıfın üyelerinin alt sınıfta da bulunmasını sağlar[3]. Kalıtım ilişkisinin bir adı da "aynısıdır" (IS-A) ilişkisidir. Bir alt sınıf, üst sınıfın beklendiği tüm bağlamlarda kullanılabilirken bunun tersi söz konusu değildir. Bir başka deyişle, aynısıdır ilişkisi tek yönlüdür.

Kalıtım kullanımı için en uygun yer, aynısıdır ilişkisinin geçerli olduğu gerçek dünya problemleridir. Aynısıdır ilişkisi alt sınıftan bir nesnenin üst sınıftan bir nesne gibi kullanılmasına izin verir. Bir başka deyişle, alt sınıftan olan bir nesne, üst sınıftan bir nesnenin beklendiği her bağlamda kullanılabilir.

Kalıtım ilişkisinde dile özel kısıtlamalar olabilir. Örneğin C++ dilinde şu metotlar kalıtımla alınmaz ve gerektiğinde programcı tarafından yeniden tanımlanmalıdır:

- Kurucular
- Sonlandırıcılar
- Birden çok anlam yüklenmiş metotlar
- "friend" olarak tanımlanmış metotlar [3].

Türetilmiş bir sınıftan da başka sınıflar türetilmesi mümkündür. Bu durumda kalıtım ilişkilerinden bir hiyerarşi oluşacaktır.

Kalıtım mekanizmasının fazla derin bir zincir oluşturmak gibi aşırı uçlarda kullanımından kaçınılması gereklidir. Aksi halde kırılgan üst sınıf problemi [4] gibi sorunlarla karşılaşılacaktır.

2.5 Çoklu Kalıtım

Bazı diller bir sınıfın kalıtım ile birden fazla türetilmesine, başka bir deyişle bir alt sınıfın birden fazla üst sınıfı olmasına izin verir. Örneğin, C++ çoklu kalıtımı desteklerken Java desteklemez. Çoklu kalıtımın desteklenmediği dillerde tek bir sınıftan türetilme yapılır ve diğer üst sınıfların üyelerinin alt sınıfta tanıtılması işi programcıya düşer. Java dilinde aynısıdır ilişkisinin tip dönüşümü için kullanılabilmesini sağlamak amacıya arayüz kavramı geliştirilmiştir [5].

2.6 Çok Biçimlilik

Aynı kalıtım hiyerarşisindeki sınıfların aynı mesaja farklı şekilde yanıt vermesine çok biçimlilik adı verilir. Derleyiciler aynı kalıtım zincirindeki farklı sınıflarda, aynı imzaya sahip üye metodun gövdesinin farklı biçimlerde tanımlanmasına izin vererek çok biçimliliğe olanak verir. Bu durumda belli bir türdeki sınıfa gönderilen herhangi bir mesajın bu sınıfa ait olan metodu mu, yoksa aynı kalıtım zinciri içerisindeki başka bir sınıfın metodunu mu yürüteceği ancak çalışma anında belli olacaktır.

Çok anlam yükleme ile çok biçimlilik arasındaki farkın anlaşılması önemlidir. Çok anlam yükleme aynı sınıf içinde aynı ada ancak farklı imzalara sahip üye metotlar tanımlanması ile olur. Çok biçimlilik ise aynı kalıtım hiyerarşisindeki farklı sınıflarda aynı imzalı ancak farklı gövdeli üye fonksiyonlar tanımlanması ile olur.

C++ dilinde çok biçimliliği sağlayacak olan metotlar **`virtual`** anahtar kelimesi ile tanımlanır. Bir metot **`virtual`** olarak tanımlandığında tüm hiyerarşi boyunca bu özelliğini korur [3], ancak programcı kodlamada kolay anlaşılabilirlik için türetilmiş sınıflarda da bu metodu özellikle **`virtual`** etiketiyle tanımlamayı tercih edebilir.

Çok biçimliliğin sağladığı en büyük katkı, tiplere olan bağımlılığın azalmasına yardımcı olmasıdır. Tip bağımlılığı azaldıkça, yazılımda sonradan yapılacak değişiklikler kolaylaşacaktır. Ancak tip bağımlılığı gereğinden fazla azaltılırsa bu kez de kodun anlaşılabilirliği azalacaktır.

2.7 Soyut Sınıflar

Bazı durumlarda bir üst sınıfı kodlamanın tek amacı, kendisinden türetilecek sınıflar için bir temel oluşturmak olabilir. Kendisinden herhangi bir nesne oluşturulmayacak ancak kalıtım hiyerarşisinde yer alması gereken sınıflara soyut sınıf adı verilir.

Bir soyut sınıftan herhangi bir nesne oluşturulmasını engellemek için C++ dilinde bir soyut sınıf `virtual` anahtar kelimesiyle tanımlanır. Bir soyut sınıf, `virtual` etiketiyle tanımlanmış ancak gövdesi tanımlanmamış en azından bir üye metot içermelidir.

2.8 Nesneler Arasındaki İlişkiler

Bu bölümde nesneler ve sınıflar arasında kurulabilecek ilişkilerden en önemlileri anlatılmıştır [4].

2.8.1 Belirtme (IsDesignatedBy)

Uygulama alanındaki soyut veya somut varlıklar ile bunların temsilinde kullanılan nesneler arasındaki ilişkidir. NYP'de sadece nesneler gerçek dünya varlıklarını modelleyebilir [4].

2.8.2 Örneğidir (IsInstanceOf)

Bir sınıf ve o sınıftan oluşturulan tüm nesneler arasındaki ilişkidir [4] (birebir olmak üzere).

2.8.3 Aynısıdır (IsA)

Kalıtım yolu ile sınıf hiyerarşileri oluşturulduğunda ortaya çıkan ilişkidir [4]. Bu bağıntı gereğince, alt sınıftan olan bir nesne üst sınıftan bir nesnenin beklendiği tüm bağlamlarda üst sınıftan bir nesne olarak kullanılabilir.

2.8.4 Etkileşir (IsAssociatedWith)

Sınıflar arasında iki yönlü etkileşim anlamında olan bu ilişki, farklı veya aynı sınıftan nesnelerin birbirlerine mesaj gönderebilmelerini sağlar [4].

2.8.5 Kullanılır (IsUsedBy)

Sınıflar ve/veya nesneler arasında istemci/sunucu anlamında tek yönlü bir ilişkidir. A nesnesi B tarafından kullanılıyor ise, B'nin en az bir metodu doğru çalışabilmek için A'ya gönderdiği mesaj veya mesajların sonuçlarına ihtiyaç duyuyor demektir [4].

2.8.6 Bir Parçasıdır (IsPartOf)

Sınıflar veya nesneler arasında birleşme, bütünlük bağlantısı kuran bu ilişki, bir başka deyişle parça/bütün ilişkisidir. Bileşik nesnelerin oluşturulmasına olanak sağlayan bu ilişkiye sahip tüm nesne ve sınıflar, birlikte bir bütünün parçasını oluştururlar [4]. İlişkiye katılanlar arasında anlamsal ve işlevsel kesişme bulunmaz.

3. DİNAMİK SİSTEMLERİN MODELLENMESİ İÇİN BAZI ÇALIŞMALAR

Dinamik sistemlerin modellenmesinde ortaya çıkan ve giriş bölümünde değinilen güçlükler, bazı araştırmacıları tüm NYP dillerinde kullanılabilecek tasarım kalıpları oluşturmaya, bazılarını da alternatif çözümler aramaya yöneltmiştir. Bu bölümün geri kalanında bazı alternatif çözümler incelenecektir. Tasarım kalıplarının incelenmesi ise daha detaylı olarak ele alınacağından 4. Bölüm'de verilmiştir. Söz konusu ayrıntılar, neden elimizde halihazırda bulunan bu kalıplara rağmen rol modellerinin kullanımına gereksinim duyulacağını açıkça gösterebilmek için gereklidir.

3.1 Nesne Arayüzleri

Skarra ve Zdonik tarafından yapılan bir çalışmada [6] problemlerin çözümü için önerilen mekanizma nesne yüzü kavramıdır. Bir yüz, mevcut bir nesneye ek durum ve davranış kazandırırken kimliğini değiştirmeden korur. Bir nesne zamanla gelen ve giden yüzlere sahip olabilir. Birçok yüz sahibi olan nesne; çoklu kalıtım ile birden fazla sınıfa ait bir nesne olmaktansa basitçe arayüzünü aldığı her sınıfa ait bir nesne olur. Bu sınıflara olan her nesne referansı aslında bir arayüze yöneliktir ve nesnenin davranışı hangi yüzün referans edildiğine göre değişir.

Nesne arayüzlerini önerilen şekilde gerçekleyen bir programlama dili henüz yazılmamıştır. Ancak önerilen nesne arayüzleri, Java arayüzlerine benzer. Dolayısıyla bunların birbirlerine karıştırılmaması için nesne yüzü veya kısaca yüz olarak adlandırılacaktır. Tablo 3.1'de Java arayüzleri ile nesne yüzlerinin karşılaştırması görülebilir. Orijinal çalışmada yüz yerine *aspect* kelimesi kullanılmıştır ancak bu kelime bakış tabanlı programlama kavramındaki [7] anlamında kullanılmamıştır.

Tablo 3.1: Nesne yüzleri ile Java arayüzlerinin karşılaştırılması

Nesne yüzleri	**Java arayüzleri**
Dinamik bağlantı	Statik tanımlama. En önemli farklılık budur.
Bir nesne birden fazla yüzü gerçekleyebilir	Bir nesne birden fazla arayüzü gerçekleyebilir
Bir yüz birden fazla diğer yüzü gerçekleyebilir	Bir arayüz diğerini gerçekleyemez
İki yüz birbirini gerçekleyebilir	İki arayüz birbirini gerçekleyemez
Miras alma söz konusu değil	Miras alma söz konusu değil
Aynı yüzü gerçekleyen farklı sınıflardan nesneler birbirinin yerine kullanılabilir	Aynı arayüzü gerçekleyen farklı sınıflardan nesneler birbirinin yerine kullanılabilir
Çeşitli eşitlik kontrolü operatörleri var	Bu operatörler Java'da bulunmamaktadır.
Bir yüzü sadece belirtilen sınıftan nesneler genişletebilir.	Statik dahi olsa böyle bir dil yapısı yoktur.
Role yalnız izin verilen nesneler sahip olabilir. (Önceki maddenin bir sonucu)	Java'da doğal rol desteği yoktur.
Bir nesne aynı tür yüzü birden fazla kez alabilir, bu yüzlerin her biri de ayrı duruma sahip olabilir.	Java'da böyle bir özellik yoktur.

3.1.1 Nesne Modeli, Nesne Yüzleri ve Kullanım Şekilleri

Önerilen model Emerald [8] dilinden esinlenerek oluşturulmuştur. Soyut veri tipleri ve gerçeklemeler (nesne) arasında kesin bir ayrım mevcuttur. Bir soyut tip metot imzaları kümesinden ibarettir ve bir çeşit arayüz tanımlar. Nesneler ise kendisinin bir sunumunu ve metotları için kodu belirten bir gerçekleme (*implementation*) ile tanımlanır. Nesne yüzü bir metot imzaları dizisidir ve her nesne gerçeklediği soyut tip(ler)de belirtilmiş tüm metotlara, aynı imzalarla sahip olmalıdır. Bir nesnenin bir soyut tipi gerçeklemiş olması için en azından soyut tipteki arayüzü sağlaması gerekir.

Soyut tip bu aşamada henüz nesne yüzü olarak adlandırılmamaktadır. Önce nesne modelinden bahsetmek gereklidir. Şekil 3.1'de bir soyut tip olan **`Person`** ve bunun gerçeklemesi olan **`personImpl`** sınıfı örneklenmiştir. Burada **`personImpl`**'nin bir **`Person`** gerçeklemesi olmasının sebebinin tanımlanmış bir ilişki olmadığını anlamak

önemlidir. Sebep, **personImpl** sınıfının **Person** için yeterli metotları aynı imzalarla gerçeklemesidir. Bu önemli bir ayrımdır çünkü bir tipin birçok gerçeklemesi olabileceği gibi bir nesne birçok tipi gerçekleyebilir.

```
/* an abstract type specification for Person */
type Person
{
       String name();
       int age();
       String phone();
};
/* an implementation of Person */
implementation personlmpl
{
       /* representation */
       String my Name;
       int myAge,
       String my Phone;
       public:
    /* operations */
    String name( return my Name; };
    int age( return myAge; };
    String phone( return my Phone; };
    /* constructor */
    personlmpl( String n, int a, String p )
    { myName = n; myAge = a; my Phone = p; };
};

/* Example usage. Declare an identifier. */
Person joeP; /* initially nil */
/* Create an object representing Joe. */
joeP = personlmpl( "Joe", 72, "555-1234");
/* Print "Joe" */
stdout.putstring( joeP.name() );
```

Şekil 3.1: Person sınıfı için bir soyut tip ve gerçeklemesi

Bu çalışmada her ne kadar Şekil 3.1'deki **Person** tipinin tanımını kaldırıp gerçekleme başlığı olarak **implementation personlmpl with type Person** yazılması ile bir kod kısaltması önerilmişse de bu kısaltmanın özel bir ilişkilendirme komutu olmadığı tekrar edilmelidir.

Soyut tipler birbirleriyle ve gerçeklemelere uyumluluk bağı ile ilişkilidir. Özetle, A ve B soyut tipler olmak üzere B, A'da belirtilen tüm metot imzalarını (ve belki daha fazlasını) sağlıyorsa "B A'ya uyar" denir. A bir soyut tip ve B de bir sınıf olursa bu tanım yine doğrudur. B ile A arasında uyumluluk şartları sağlandığı sürece B, A'yı

gerçekler denilir. Soyut tipler karşılıklı olarak birbirini gerçekleyebilir. Bir gerçekleme ilişkisinin kodda önceden deklare edilmesine gerek yoktur.

Uyma kuralı programcının genel bir tip tanımlamasını ve bu tipin ortak davranışı paylaşan daha özelleşmiş tiplerini gerçeklemesini sağlar. Daha genel tipe uyan tüm daha özel nesneler, bu genel tipin beklendiği her yerde sorunsuzca genel tipin yerine geçirilerek kullanılabilir. Burada hiçbir şekilde sınıf seviyesinde miras alma bağıntısı söz konusu değildir. Yeni alt tipler tanımlamanın yanı sıra, mevcut tiplerin üzerinde ve mevcut tiplerin tanımını değiştirmeden üst tipler tanımlanması da uyma kuralının getirdiği güçlü bir özelliktir.

Soyut tipler, gerçeklemeler ve uyum kuralının kombinasyonu ile elde edilen nesne yüzleri, rollerin modellenmesini kolayca desteklemek için güçlü bir mekanizma sağlar. Sözdizimsel açıdan nesne yüzünün tanımı, bir gerçeklemenin genişleteceği taban nesnenin (uyduğu soyut tipinin) bildirilmesi ile kuvvetlendirilmesidir (Nesne yüzü = gerçekleme + rolü alabilecek soyut tipin tanımı). Her nesne yüzü aslında bir roldür. Şekil 3.2'de bir memur yüzü (rolü) tanımlanmıştır:

```
/* an employee aspect definition */
implementation emplmpl //sınıf adı
       with type Employee   //aspect (yüz) adı
       extends Person       //base object türü (sınıfı)
{
       /* added employee state */
       int myEid,
       String myPhone, myDept;
       public:
     /* constructor */
     emplmpl( int e, String d, String p )
     { myEid = e, myDept = d, myPhone = p; };
     /* operations */
     int eid() { return myEid; };
     String dept() { return myDept; };
     String phone()
       {
       if( my Phone != nil )   return myPhone;
       else                    return base.phone();
     };
       /*operations imported from Person */
       Person::name; //Kod tanımlanmadığından Person'daki ile
     aynı
       Person::phone as homePhone;
};
```

Şekil 3.2: Bir **`Person`** nesnesinin memur yüzü.

Gerçekleme başlığındaki **extends** yönergesi verilen soyut tipe uyan her nesnenin (temel nesne adı alır) bu yüze sahip olabileceğini belirtir. Yüz adı ile bir tip tanımlanmış olur. Örnekte **empImpl** tarafından gerçeklenmiş **Employee** yüzünü **Person** soyut tipine uyan her nesnenin alabileceği tanımlanmıştır.

Bir yüz yeni durum ve davranış ekleyebilir ve temel nesnenin arayüzünün istenilen kısımlarını ithal edebilir. Yeniden tanımlanmayan veya ithal edilmeyen metotlara erişilemez. Bu özellik sisteme veri gizleme yeteneği kazandırır ancak bu özellik bir ek yük olarak da görülebilir. Öte yandan temel nesne ile nesne yüzünün arasındaki uyma bağıntısını bozabileceği için bir yüzün temel nesnesinin yerine geçebilme özelliğini kaybetmesine de neden olabilir.

Bir yüz temel nesnenin metot ve üyelerine de ayrıca erişebilir. Bunun için dilde **base** adı verilen bir referans tanımlıdır. Şekil 3.2'deki **phone()** metodunda bunun bir örneği görülmektedir. Kullanımı C++'daki **this** işaretçisine benzer. Önerilen sistemde **this** işaretçisi metodun çağrıldığı yüze işaret etmektedir.

Yeniden tanımlanan metotlar bir isim çakışmasına yol açmaz. Ancak anlamsal karışıklığı önlemek için bazı metotların yeniden adlandırılması gerekebilir. Şekil 3.2'deki memur örneğinde hem temel nesnenin hem de yeni yüzün **phone** üyesi vardır. Bu üyeleri döndüren iki metot da aynı adlı olamayacağına göre iş telefonunu döndüren metot **phone** olarak adlandırılmışken ev telefonunu döndüren metoda **homePhone** adı verilmiştir. Şekilde **homePhone** metodunun tanıtılması için kısa gösterim önerilmiştir, açık gösterimi **String homePhone(){ return base.Phone();}** şeklindedir.

Şekil 3.2'de gösterilen memur yüzü gerçeklemesinin bir **Person** nesnesine memur rolü kazandırmak için nasıl kullanılacağı Şekil 3.3'te gösterilmiştir. Önce **joeP** değişkeniyle referans edilen "Joe" adlı bir şahıs tanımlanıyor. Ardından onu oyuncak departmanına atayan bir yüz veriliyor. Nesne ile rolün ilişkilendirmesi yüzün tanımlanması anında yapılıyor. Yeni yüz artık temel nesnenin bir uzantısı olmuştur ve her ikisinin de nesne kimliği hem mantıksal açıdan hem de derleyici için aynıdır. Yine de temel nesne ile yüzü farklı referanslardır. Buradan nesnenin karşılaştırılması için iki tür eşitlik operatörünün tanımlanması gereği doğar: **==** operatörü referans

eşitliği için kullanılırken nesne kimliği eşitliği testi için önerilen yeni operatör `@=` ile simgelenir.

```
/* example usage. Create a Person, Joe. */
Person joeP = personlmpl( "Joe", 72, "555-1234");

/* extend Joe with Employee information */
Employee joeE = emplmpl(10139, "Toy", "555-5678" ) extends
    joeP;

/* joeE and joeP are same oid, but not identical refs, */
Bool bl = ( joeE @= joeP ), /* TRUE */
Bool b2 = ( joeE == joeP ); /* FALSE */

/* Employee does not conform to Person */
Person p = joeE; /* ERROR */

/* Person and Employee both conform to PhoneOwner */
PhoneOwner pol = joeP; /* OK */
PhoneOwner po2 = joeE; /* OK */

/* Employee exports Person::name method */
stdout putstring( joeP.name() ); /* "Joe" */
stdout.putstring( joeE.name() ), /* "Joe" */

/* Employee and Person have different phone methods */
stdout.putstring( joeP.phone() ); /* "555-1234" */
stdout putstring( joeE.phone() ); /* "555-5678" */
stdout putstring( joeE.homePhone() ); /* "555-1234" */
```

Şekil 3.3: Bir memur yüzü oluşturmak ve kullanmak

Bundan sonraki satır nesne yüzü ile uyum kuralı arasındaki yukarıda da belirtilen ilişkiyi simgeler. Örneğimizde memur yüzü ile şahıs nesnesi arasında uyum kuralı bozulmuştur ve bu ikisi birbirinin yerine kullanılamaz. Geri kalan satırlarda ise çeşitli kullanım örnekleri verilmiştir.

Temel nesneye eklenen bir yüz, nesnede tanımlanan metotların yerine geçmez. Bu özellik temel nesneyle halen süren ilişkilerin bozulmamasını sağlar. Nesne yüzleri çoklu miras almada karşılaşılan sorunları da çözer: Yüzler sayesinde mevcut tüm rollerin bütün kombinasyonları için birer arakesit sınıfı oluşturma zorunluluğu ortadan kalkar, sadece gerekli rol sayısı kadar arayüz tanımlanır.

Bir nesneye birden fazla yüz eklenebilir. Eklenen yeni yüz, eski yüzlerin kaybedilmesine neden olmaz, çünkü eski yüzlerle olan uyum kuralı yeni yüzler eklenmesiyle bozulmaz.

Bir nesne arayüzünden olan rol nesnesini taban alan yeni bir yüz tanımlanması mümkün olduğu için, bir nesne tüm rolleri ile birlikte bir ağaç yapısı veya çok seviyeli bir hiyerarşi oluşturabilir.

Bir temel nesne aynı tip yüzden, yani aynı kurucu metot ile üretilen nesne yüzlerinden birden fazlasına aynı anda sahip olabilir. Örneğin kurucu metoduna bölüm adını parametre olarak alan bir öğrenci yüzü tanımlanırsa şahıs yüzüne iki farklı bölümde öğrencilik rolü kazandırılabilir.

3.1.2 Değerlendirme

Bu çalışma henüz tamamlanmamış olsa da rollerin gereksinimlerini eksik bırakmadan gerçekleyebilecek araçlara sahiptir ve eksik kalan noktalar çalışma ile kapatılabilir. Bunlara örnek olarak bir nesnenin sahip olduğu yüzlerin keşfedilmesi ve yüzlerin gerektiğinde terk edilmesi mekanizmaları verilebilir. Daha da önemlisi bu çalışmada önerilen model kalıtım konusunu ele almamıştır ve değişik rollere erişim için açık operatörler tanımlamamıştır.

Bu yaklaşım sıfırdan başlayarak yeniden oluşturulacak bir dil için daha uygun olacaktır. Mevcut bir NYP diline bu yaklaşımla rol desteği kazandırmakta en önemli sorunlardan biri, dilin sağladığı işlevler ile nesne yüzleri arasındaki farklılıkları ortadan kaldıran mekanizmalar tasarlamak olacaktır.

3.2 Etmen Tabanlı Sistemler

Çoklu etmen sistemleri (ÇES), etmenlerin bir arada çalışarak yeteneklerini birleştirme yolu ile büyük bir işi çözmek için işbirliği yaptıkları sistemlerdir. Bu tip çalışmalarda rol kavramına değinilse de roller etmen bazında bir yetenekler topluluğu olarak ele alınır. Rollere çoğu kez rol modellerinin aksine zamanla kazanılan ve terk edilen özellikler olarak değil, bir etmenin önceden ve kesin hatlarla tanımlanmış görevleri olarak bakılır. Görev dağıtım şemaları gibi veri yapıları ile yetki atamaları yapılır. Örnek olarak ROPE [9] ve ÇES bazında rol modelleme konulu [10] çalışması incelenebilir. Diğer taraftan rol modelleri etmenlerin hem davranışlarını hem de eşgüdümünü sağlamak için kullanılabilir. Bu tür yaklaşıma sahip olan çalışmalara örnek olarak [11] ve [12] verilebilir.

3.2.1 Değerlendirme

Çoklu etmen sistemlerin rol yaklaşımı ile rol modellerinin rol yaklaşımı birbirinden farklıdır. ÇES'lerin odak noktası dinamik sistemlerin modellenmesi olmayıp, ÇES'lerin gerçeklenmesinde rol modelleri yardımcı olabilir [13].

3.3 Şema Evrimleştirme

Nesnelerin ait olduğu veya türetildiği sınıf veya sınıfların dinamik olarak değişmesi, şema evrimleşmesi olarak adlandırılır ve sınıfların kalıtım zincirlerinin değişmesini de beraberinde getirir. Şema evrimleştirme iki gruba ayrılabilir. İlk grup evrimleşme sınıf tanımında yapılan değişiklikleri, ikinci grup evrimleşme ise sınıf hiyerarşisindeki değişiklikleri kapsar. Bu bölümde şema evrimleştirme sadece ilk grup olan sınıf hiyerarşisinde yapılan değişiklikler açısından ele alınacaktır. Diğer tür olan sınıf hiyerarşisindeki değişiklikler ise Bölüm 3.4'te incelenmiştir.

Şema evrimleştirme çalışmaları arasında bileşik nesneler ile yapılanlar mevcuttur. ORION [14] bunlardan en önemlilerinden biridir. Bölüm 3.5'te değinileceği üzere bileşik nesnelerin tez konusu ile ilgili olabilmesine rağmen bu çalışmada rol kavramından bahsedilmemiş, şema evriminin semantiği ve gerçeklemesi üzerinde durulmuştur.

3.3.1 ORION Sisteminde Şema Evrimleşmesi

ORION'un nesneye yönelimli veri modelinde 20'den fazla türde şema değişimi tanımlanmıştır. Bu açıdan ORION'daki şema değişimi statiktir, yani sadece önceden tanımlanmış olan bu değişim yolları izlenerek şema evrimleştirme yapılabilir. Halihazırda tanımlanmış evrimleşme yolları yeterli gözükmekle birlikte, yenilerinin keşfedilip sisteme eklenmesi gerektiğinde tüm sistemde revizyona gidilmesi gereklidir.

Evrimleşme yolları tanımlandıktan sonra ikinci adım olarak bunların anlamsal tanımlamaları yapılmıştır. Bunun için öncelikle bir şemanın tüm değişiklikler süresince koruması gereken değişmez özellikleri belirlenmiştir. Bu değişmez özellikler, nesnenin zamanla değişmeyen işlevleri, yani rol hiyerarşisinin kökü anlamında değildir. Sadece kalıtım zincirlerinin hiçbir örnek için değişmeyecek

özellikleri anlamındadır. ORION'da çoklu kalıtım desteklenmektedir. Böylece sınıf kafesi ağaç veya çizge (graph) yapısı şeklinde olabilir. İkinci adımın devamında, istenen sonucu verecek birden fazla evrimleştirme sonucu bulunduğu takdirde bunlardan en anlamlısının ve değişmez özellikleri ihlal etmeyecek olanının seçimi için bir dizi kurallar tanımlanmıştır.

Üçüncü ve son adım olarak tüm veri tabanının yeniden organizasyonunu gerektirmeyecek veya sistemi kapatmak zorunda kalınmayacak şekilde bir gerçekleme stratejisi kararlaştırılmıştır. Ek olarak son kullanıcılara yardımcı olmak ve kurulan şema evrimleştirme alt yapısının doğruluğunu ispat etmek amaçlarıyla ORION üzerinde çalışan grafik tabanlı bir şema editörü gerçeklenmiştir.

Şemanın evrimleşebilmesi, miras alınan ve sınıf bünyesinde tanımlanan üyelerle metotlar arasında olabilecek adlandırma çakışmalarına özellikle dikkat edilmesini gerektirmektedir. Çakışmaların çözülmesi miras alma hiyerarşisindeki bir adımın baskın olarak ilan edilmesiyle çözülür. NYP'de en son tanımlanan ad geçerlidir, bir üst seviyedeki veya hiyerarşinin kökündeki ad veya adlara erişim dile özel operatörlerle olur. Baskın adımın seçilmesi sonuçta miras zincirindeki bir permutasyona dayanmaktadır. ORION'un güçlü bir yönü, bu çakışmaları önleyecek kuralların sistem tarafından otomatik olarak uygulanmasının yanı sıra kullanıcının söz konusu permutasyonlardan istediği birini dikte edebilmesidir.

ORION'da gerçeklenen şema evrimleştirme alt yapısının herhangi bir NYP diline uyarlanabileceği belirtilmektedir. Bu yapı daha önce değinilen değişmezlerden ve birden fazla seçenek olduğunda en uygununun tercihini sağlayan kurallardan oluşmaktadır.

3.3.2 ORION'da Bileşik Nesneler

Bir ORION bileşik nesnesi birbiriyle ilişkili nesnelerin oluşturduğu hiyerarşik bir yapıdır. Bileşik nesneyi oluşturan nesneler arasında ise "bir parçasıdır" ilişkisi vardır. Bir bileşik nesne ağaç yapısındadır: Tek bir kökü mevcuttur ve kök (veya ara düğümler) birden fazla çocuk nesneye işaret eder. Her işaret, bileşik örnek değişkeni adı verilen ve göstereceği nesneye ait olan tipteki bir üye değişken üzerinden yapılır. Bu işarete bileşik bağ adı verilmiştir.

Bir ebeveyn, çocuklarını dışlamalı olarak sahiplenir. Çocuk nesneler bu bakımdan bağımlı nesnelerdir. Sonuçta oluşan ağaca bileşik nesne şeması adı verilir. Bu yapının tanımı rol hiyerarşisini anımsatmakla birlikte verilen örneğin rol kavramıyla benzerliği bulunmamaktadır.

Bir bileşik bağ NYP veri modelinin bütünlük özelliklerine iki yeni özellik ekler. Bunlardan birincisi, bileşik bağın gösterdiği nesneye sadece o bağ üzerinden erişilebilmesidir. Örneğin A sınıfından B sınıfına VA adlı bir bileşik bağ varsa, A tipi bir nesne olan a'nın va adlı bileşik bağı simgeleyen üyesi B tipindeki b'ye sadece a nesnesi va üzerinden erişebilir. C sınıfından olan c de b'ye erişebilir, ancak bu erişim va üzerinden olamaz. İkinci özellik ise işaret edilen nesnenin varlığının işaret eden nesnenin varlığına bağlı olmasıdır. Aynı örnek üzerinden gidersek, b ancak va ve a yaşadığı sürece var olabilir.

3.3.3 Değerlendirme

Tek başına şema evrimleştirme, rol desteği için yeterli değildir. Şema evrimleştirme güçlü bir araçtır, ancak rol desteği için gerekliliği özellikle hiyerarşik yaklaşımlar için tartışmaya açıktır. İncelenen yaklaşımlarda ve tez konusu olarak önerilecek yapıda roller zaten istenildiğinde kazanılıp terk edilebilmektedir. Şema evrimleştirme ancak Java arayüzlerinin dinamik yapıya sahip olmamasından kaynaklanan eksiklikleri gidermek için kullanılabilir. Bu kez de çok karmaşık ve gerçeklemesi zor bir sistem ortaya çıkacaktır.

3.4 Nesne Göç Ettirme

Nesne göç ettirme terimi ile, bir nesnenin daha önceden ait olduğu sınıftan yeni bir sınıfa taşınarak biçim değiştirmesi; bir başka deyişle çalışma anında tip değiştirmesi kastedilmektedir. Bir tipten diğerine geçen nesne, geçiş sırasında eski tipinin üyelerini kaybetmeyip bu üyelerin yeni tipe uyarlanması sayesinde evrimleşmiş olacaktır.

Nesne göç ettirme yaklaşımına örnek olarak PKM [15] adlı bir NYVT motoru incelenebilir. Bir motor görevi gördüğünden PKM işlemleri son kullanıcı tarafından doğrudan kullanılmak yerine, daha üst düzeyli veri tabanı veya uzman sistemler için bilgi tabanı uygulamaları hazırlayacakların kullanacakları temel yapı taşları olarak

önerilmiştir. Bu çalışmada rol kavramının desteklendiği açıkça belirtilmiştir. Rol desteği, bu çalışmada nesnenin biçim değiştirmesi ile sağlanmaktadır.

Nesne göç ettirmeye, bir başka deyişle tip değişimine dayanan tüm yaklaşımların ortak bir zayıflığı zamanla oluşabilecek tutarsızlıklardır. Bir nesne evrimleşince, bu nesnenin önceden iletişim halinde olduğu diğer nesnelerle arasındaki ilişkiler bozulabilir. Eski tiple çalışan diğer nesnelerin metotları yeni nesneyle uyuşamayabilir. Rol hiyerarşisi ve nesne arayüzlerinde bu sakıncalar en aza indirilmiştir, çünkü değişen rol gerçek dünya nesnesinin geri kalan rollerini etkilememektedir. Tek istisna diğer arayüz veya alt rollerin eriştiği bir üyenin değişikliğe uğrayan rolden silinmesi durumudur.

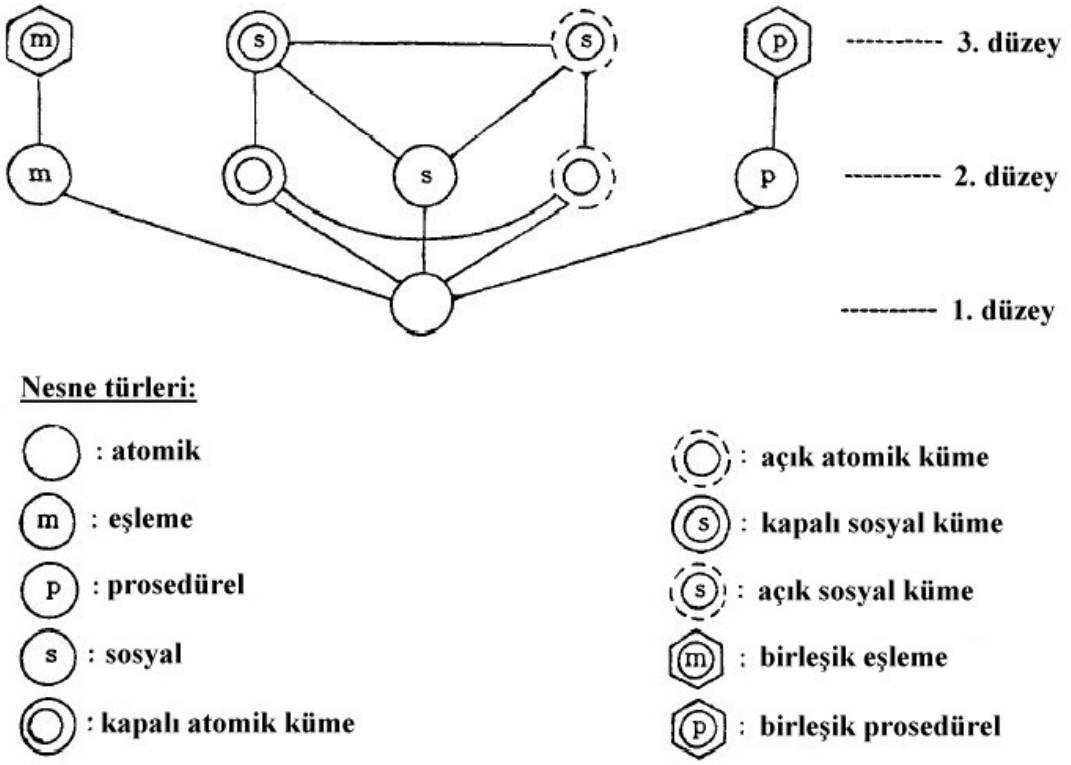
Bu yaklaşımda karşılaşılabilecek bir diğer sorun da isim çakışması olasılığıdır. Değişen bir tip, alt veya üst sınıflarında bulunan bir metot veya üye değişken adı alabilir. Bu durumda tip uyuşmazlığı, mantıksal hatalar gibi sorunlar ortaya çıkacaktır. Rol hiyerarşisi ve nesne arayüzü yaklaşımlarında bu sorun bulunmamaktadır, çünkü alt sınıflar üst sınıfın metot ve üyeleri ile aynı ada sahip metot ve üyelere sahip olabilir. Bu durumda o adla erişilen özellik alt role ait olacaktır, üst tipin ilgili özelliğine tanımlanan özel bir operatör yardımıyla erişilmektedir.

3.4.1 Nesne Göç Ettirme İle Rol Desteği

Bir PKM veri tabanı nesneler ve nesneler arası eşlemelerden oluşur. Eşlemeler de birer nesnedir. Veri tabanının kayıtları, d ve r nesne, m ise eşleme nesnesi ise (d, m, r) şeklinde üçlüler olarak düşünülebilir. O nesneler uzayını, M de eşleme nesneleri uzayını simgelerse PKM veri tabanı (O x M x O) kümesinin bir alt kümesi olacaktır. Örneğin **`(John Smith,has-gpa,"3.75")`** ve **`(John Smith,is-member-of, Applicants)`** gibi.

PKM'de 10 adet hazır ve sabit nesne biçimi bulunmaktadır. Bu 10 biçim 3 seviyeye ayrılmıştır. Sisteme girilen her yeni nesne ilk seviyeden başlar ve her seferinde birer seviye ileride veya gerideki bir biçime dönüşmek kaydıyla zamanla biçimini değiştirebilir. Bir nesnenin yok edilmesi için ise önce 1. seviyeye çekilmesi gerekir. Bu işlem sistemde nesneler arasında kırık bağlantılar (ilişkiler) olmasını engeller. 1.

seviyedeki tek biçim Atomik biçimdir. Bu biçime sahip nesneler oluşturma aşamasında parçalarına ayrılamayacak bilgi birimleridir. Üst seviyelere çıkıldıkça veri yapısı karmaşıklaşmaktadır. Önceden hazırlanmış bu şema Şekil 5.1'de görülebilir.



Şekil 3.4: PKM sınıf şeması [15]

Atomik biçim dışındaki diğer biçimler şunlardır:

- Açık atomik nesne kümesi biçimi: Belli bir kritere göre ilişkili atomik nesneler kümesidir.

- Kapalı atomik nesne kümesi biçimi: Belli bir kritere göre ilişkili atomik nesneler kümesidir. Ancak bu küme bir kere tanımlandıktan sonra kümeye yeni eleman eklenemez veya mevcut bir elemanı çıkarılamaz.

- Sosyal nesne biçimi: Nitelikleri olan, yani diğer nesnelerle olan ilişkileri ile tarif edilen varlıklardır.

- Eşleme biçimi: Bir nesne kümesinden diğerine bir eşleme şablonu oluşturur. Bir şablon eşleme adı, eşleme metodu, ters eşleme metodu, eşlemenin hedef ve kaynak uzayı ve nesneler arası ilişkiler için basit kısıtlama kuralları içerir.

- Prosedürel biçim: Çalıştırılabilecek işlemler ve prosedürler içerir. Nesne oluşturmak, güncellemek ve yok etmek için önceden tanımlanmış ilkel prosedürel nesnelere ek olarak, kullanıcı yeni prosedürel nesneler oluşturabilir ve mevcutların kodunu değiştirebilir.

- Açık sosyal nesne kümesi biçimi: Belli bir kritere göre ilişkili sosyal nesneler kümesidir. Açık küme olduğu için kümeye yeni elemanlar eklenebilir veya mevcut elemanlar çıkarılabilir.

- Kapalı sosyal nesne kümesi biçimi: Belli bir kritere göre ilişkili sosyal nesneler kümesidir. Kapalı küme olduğu için bir kere tanımlandıktan sonra kümeye yeni eleman eklenemez veya mevcut bir elemanı çıkarılamaz. Ancak elemanlar bir seçme ifadesi ile eklenmişlerse sorgunun sonucunun değişimine göre kümeye eklenip çıkarılabilir.

- Bileşik eşleme nesneleri biçimi: Çeşitli eşleme nesnelerinin bir gruplanması ile elde edilen sanal nesnelerdir. 2 seviyedeki eşleme nesnesine çevrilerek gerçek eşleme nesnesi yapılabilirler.

- Bileşik prosedürel nesne biçimi çeşitli prosedürel nesnelerin bir gruplanması ile elde edilen sanal nesnelerdir. Önceki biçim gibi kullanıcı talebi üzerine gerçek prosedürel nesneye dönüşebilirler.

Şekilde görüldüğü üzere biçimler arasında 13 bağlantı mevcuttur. İki yönde de geçiş olabileceğinden 13 x 2 = 26 değişik biçim değiştirme işlemi mümkündür. Her geçiş yani evrimleşme yolu çeşitli nedenlerden ötürü izlenmiş olabilir. Bu nedenler genel nesne tasarım aktiviteleri, modellenen nesnenin gerçek dünyada evrimleşmesi, nesnenin kullanıcı bakış açısına göre evrimleşmesi gibi tahmin edilebilecek nedenlerdir. Daha ileri düzey iki neden ise şunlar olabilir:

- Sorgu başarısızlığı: Bazen sonuç kümesi boş gelen bir sorgu çalıştırılabilir. Eğer sorgunun cevabının veri tabanının derinliklerinde bir yerlerde gizlendiği biliniyorsa uygun biçimde nesnelerin uygun biçimde evrimleşmesi ile sonuç saklandığı yerden bulunup çıkartılabilir.

- Aykırı durumlar: Gerçek dünya düzensiz olduğu ve öngörülemediği için veri tabanı sistemleri veri yapısından kaynaklanan kısıtlamalardan kimi zaman sapmalara tahammül edebilmelidir. Bu amaçla uygun evrimleşmeler yapılabilir.

3.4.2 Değerlendirme

Çalışmanın altında yatan fikir orijinal ve ilginç olmasına rağmen, temel yaklaşımlarla karşılaştırıldığında uygulama kuralları açısından karmaşıktır. Nesne göçü sonrasında zamanla oluşabileceği belirtilen tutarsızlıkların giderilmesi için çeşitli çalışmalar yapılmıştır. Bunlara bir örnek olan [16] çalışmasında bu tutarsızlıkların çözümü için bir yaklaşım önerilse de tutarsızlıkların tümünün çözülemeyeceği belirtilmiştir. Dahası, bütün bir nesnenin göç ettirilmesinin getireceği ek yükün rol hiyerarşisinde değişiklikler yapmanın getireceği ek yükten daha fazla olacağı tahmin edilir. Ne yazık ki incelenen çalışmalarda başarım ölçüm sonuçlarına değinilmemiştir.

3.5 Bileşik Nesne Modelleri

Bileşik nesne modellerinde (BNM), bir dağıtılmış ve paylaşılan nesne tekil bir adres alanı üzerinde bir veya daha fazla sayıda alt nesnenin bir kabuk nesne altında birleştirilmesi ile oluşturulan bileşik nesneler kullanılır. Nesnelerin dağıtılmış olmasını bir kenara bırakırsak, küçük nesnelerin bir tek büyük nesne çatısı altında birleştirilmesi bize doğal rol desteği sağlar. Bir gerçek dünya nesnesini zaman içinde hiç değişmeyecek özellikleri içeren ana alt nesne ve bunun zaman içersinde alacağı rolleri gerçekleyen diğer alt nesneler olarak bir bileşik nesne şeklinde modelleyebiliriz. Buradan anlaşılabileceği gibi BNM'ler tek başına rol desteği için yeterli olmamakla birlikte, yapısı dağıtılmış çalışmaya uygun olmayan rol modellerine dağıtılmışlık özelliği kazandırmak için kullanılabilir. Bu gerçeğe rağmen bileşik nesnelerin rol desteği için kullanılabileceğinin açıkça belirtildiği bir çalışmaya rastlanılamamıştır. Ancak rol desteğini birleşik nesneler ile sağlayan sistemler mevcuttur ve bazıları da Bölüm 6'da incelenecektir. Önceki bölümde incelendiği gibi Orion'da [14] da bileşik nesne yapısı önerilmiş ancak şema evrimi rol gerçeklemesinde kullanılabilecekken rol kavramı üzerinde durulmamıştır.

Bileşik nesne modellerine örnek olarak DBNTO [17] incelenebilir. Dağıtılmış yapının ilk paragrafta belirtilen avantajına ek olarak, sistemin sağladığı dinamik uyarlama desteği rol kavramının desteklenmesi için ikinci önemli özelliktir. Yeni bir alt nesnenin bir kabuk nesnenin bir bileşeni olarak eklenebilmesi veya var olanlardan birinin çıkarılabilmesi anlamına gelen bu özellik sayesinde kullanılan bileşik nesneye yeni bir rolü simgeleyen alt nesnenin eklenmesi mevcut çalışma düzenini bozmaz. Bir rolün terk edilmesi amacıyla alt nesnelerden biri olan rol nesnesinin bileşik nesneden çıkartılması da aynı şekildedir.

Her ne kadar rol desteği için elzem olmasa da, kullanıcının DBN'yi oluşturma aşamasında tutarlılık kontrolünü *read-invalidate* veya *write-update* olarak seçebilmesi sistemin diğer bir cazip özelliğidir.

3.5.1 Değerlendirme

DBNTO sisteminin rol tabanlı programlama aracı olabilmesi için önemli bir eksiği bulunmaktadır: Parçalı nesneler tek başına rol desteği veremez. Parçalar arasındaki rol hiyerarşisini gerçekleyen mekanizmalar eklenmelidir, ki bu da tez çalışmasının ilk amacıdır.

DBNTO'nun dağıtılmış işlemlerde başarımı tatmin edicidir. Tek dezavantaj, yerel işlemlerde ilk kez yürütülmesi durumunda, DBNTO çağrısının normal Java çağrısına göre bir misli daha fazla zaman almasıdır.

Özetle, Java nesnelerine rol desteği sağladıktan sonra dağıtılmış rol desteğinin verilmesi için DBNTO çok değerli bir araç olabilir. Belirtilen eksiğin ve başarım sorununun giderilmesi gibi iyileştirmelerin yapılması da gerekecektir.

3.6 Özne Tabanlı Programlama

ÖTP'de her uygulama bir özne (*subject*) veya öznelerin bir kompozisyonudur. Bir özne sadece uygulamanın kendisine ait durum ve davranışların yanı sıra, ilgili sınıfların durum ve davranışlarından bir kısmını da belirler. Bu kavramın ilk önerildiği çalışma Harrison ve Ossher tarafından yapılmıştır [18].

Özne terimi ile, belirli bir amacı yansıtan, gerçek dünyaya geniş bir açıdan (sistemdeki bir çok uygulamadan belli birisinin bakacağı gibi) bakan bir durum ve davranış koleksiyonu anlatılmaktadır. Bildiride kimi yerlerde öznelerden birer birey gibi bahsedilse de aslında bunlar kendileri birer birey değil, bazı bireylerin ortak bakış açısının genellenmiş halidir. Özneler genellikle birçok sınıftan nesnelerin durum ve davranışını tarif etmekle birlikte sisteme yeni sınıflar da ekleyebilirler.

ÖTP metodolojisinin hedeflerinden biri, NYP'nin nesnelere yüklediği önemi gevşetmektir. Sınıf bazında kalıtım, nesnelerin zaman içinde değişmeyeceğini varsayarak, nesnelere yerellik anlamı yükler. Bir sistemdeki birden fazla bulunması kaçınılmaz olan öznel bakış açılarının nesne kavramının yerelliğini azalttığı gerçeği bu yaklaşımda vurgulanır. ÖTP'nin ana amacı birbirleriyle işbirliği yapacak ayrı uygulamaların gerçekleme ve evrimleşme aşamalarının NYP'nin dezavantajlarını ortadan kaldırarak tamamlanmalarını sağlamaktır. Bu bağlamda şu noktalara odaklanılır:

- Uygulamaları bağımsız olarak gerçekleyip sonra bunları birleştirebilmek mümkün olmalı

- Bağımsız olarak gerçeklenen uygulamalar birleştirilecekleri diğer uygulamalarla açıkça bağımlı olmak zorunda olmamalıdır

- Birleştirilen uygulamalar sıkı veya gevşek bir işbirliği içinde bulunabilmeli, sıkça ve hızlı etkileşim için birbirlerine sıkıca bağlı veya geniş bir dağıtılmış sistem oluşturabilmelidir.

- Bir birlikteliğe yeni bir uygulama ekleneceği zaman mevcut uygulamaların veya bunların oluşturduğu kalıcı nesnelerin değiştirilmesi ya da geçersiz kılınması gerekmemelidir. Ancak iyileştirme için yeniden düzenleme düşünülebilir.

- Daha önceden beklenilmeyen yeni uygulamalar veya mevcut uygulamalara yeni hizmetler sunacak olan beklenmedik uygulamaların sisteme katılabilmesi desteklenmelidir.

Bu temel ilkeler göz önüne alınınca ÖTP'nin ilgi alanlarının ayrılma ilkesine önem verdiği anlaşılmaktadır.

ÖTP modelinde içsel özelliklere herhangi bir özel önem atfedilmez. Eğer isterse bir programcı bir veya daha çok sınıftan nesnelerin içsel özelliklerini gerçekleyen bir özne yazabilir ve bu özelliklerle yapılan okuma ve yazma işlemlerinin bu özne tarafından gerçeklenmesini sağlayabilir.

ÖTP'nin temel özelliği değişik öznelerin birbirinden bağımsız olarak paylaşılan nesneler tanımlayabilmeleri ve bunlar üzerinde işlemler yapabilmeleridir. Özneler bu aşamada bu nesnelerin diğer öznelerle ilişkili olan özelliklerini bilmek zorunda değildir. Özneler arasında ortak nesnelerin sadece tekil tanımlayıcıları paylaşılır.

Özne tanımsal veya şematik olabilir; geleneksel sınıf hiyerarşisini (genellikle eksik olmakla birlikte) andırsa da, öznenin kendisi herhangi bir durum bilgisi içermez; kendisinin bildiği sınıf ve arayüzleri bir çeşit şablon sunarak tanımlamakla yetinir. Arayüz, bir nesneler grubunun desteklediği işlemlerin soyut bir tarifini yapar.

Nasıl ki bir nesne kendi içinde yapısal programlama gereçlerini kullanır ve başka nesnelerle etkileşimi NYP kuralları ile belirlenirse, benzer şekilde bir özne herhangi bir nesneyi NYP perspektifinden görürken, nesneye müdahalesi ve diğer öznelerle etkileşimi ÖTP kuralları ile belirlenir.

İşbirliği yapan özneler, kompozisyon olarak adlandırılan gruplarda toplanabilir. Her kompozisyon için bu kompozisyondaki bileşenlerin nasıl bir araya getirilebileceğini belirleyen bir kompozisyon kuralı mevcuttur. Bir ÖTP programı, özne veya özne kompozisyonlarının istenilen sırada etkinleştirilmesine dayanır. Her ikisi de özne aktivasyonu adı altında toplanmıştır. Dağıtılmış sistemlerde aktivasyonlar da uzayda ve zamanda ayrık olabilirler. Böylece herhangi bir nesnenin tüm sistem çapında bir genel durumundan bahsedilemez. Önemli olan yeni özne aktivasyonlarını diğerlerini bozmadan sisteme dahil edebilmektir. ÖTP kodu nesnelerin genel durumuna veya biçimine dayanmamalıdır.

3.6.1 Değerlendirme

ÖTP yaklaşımı IBM tarafından geliştirilmektedir ve günümüzde pratik uygulama gereçlerine sahiptir. IBM'in ÖTP sitesine (http://www.research.ibm.com/sop/) girildiğinde oturmuş bir alt yapının kurulduğu görülecektir. C++ için IBM'in VisualAge 4 derleyicisi ve Java için Hyper/J eklentisi ÖTP desteği sağlamaktadır. ÖTP'nin getirdiği kazanç ve kayıpların gerçek projeler ile ölçümü çalışmaları sürmektedir.

ÖTP'nin ilgi alanlarının ayrılma ilkesine önem verdiği anlatılmıştı. Bu yetenek her tür sistemin modellenmesinde avantajlar sağlayacaktır, ancak dinamik sistemlerin modellenmesindeki tek gereksinim ilgi alanlarının ayrılması değildir.

3.7 Bakış Tabanlı Programlama

Bakış (*aspect*) tabanlı programlamanın özünü oluşturan, ilgi alanlarının ayrılması problemini çözmek için önerdiği mekanizmalardır. İlgi alanları en uygun şekilde ayrılsa da halen ortak bir görevin parçaları farklı modüllere dağılmış ve/veya farklı modüllerde tekrarlanmış olabilir. Bu tip görevlere "kesişimli ilgi alanı" adı verilir. Bakış tabanlı programlama, bu kesişimli ilgi alanlarını tek bir bakış altında toplayarak "arakesit modülerliği" sağlar. Bir başka deyişle, bakışlar, arakesitlerin kesişimlerinde yer almaktadır.

İlgi alanlarının birbirinden ayrılması açısından bu yaklaşımlar üç temel gruba ayrılmıştır [19]:

1. Programlama dili (*linguistic*) yaklaşımı: Bakışları ve etkileşimlerini ifade etmek için kullanılan bazı yazılım dili yapılarına dayanır. İlişkili görevler tespit edilir, bu görevler ortak bakışsal dil yapılarına çevrilir ve ana programa dahil edilir. AspectJ [20] bu yaklaşıma bir örnektir. Bu yaklaşımın dezavantajı bakışların zamanla gelişmesini desteklemenin zorluğu ve sağladığı işlevselliğin daha kısıtlı olmasıdır. Ayrıca BTP dillerinin veya araçlarının mevcut araç ve dillerle entegrasyonu başlı başına bir meseledir.

2. Saf nesneye yönelim yaklaşımları.

3. Mimariye yönelik yaklaşım: Mimari yapı modelleri kullanarak görevleri önceden tespit etmeyi ve sonra çeşitli gerçekleme teknolojileriyle bunları bakışlara eşlemeyi amaçlar. Örneğin NYP, söz konusu gerçekleme teknolojilerinden biri olabilir.

BTP'nin güçlü ve zayıf noktalarını analiz etmek için bir örnek olay incelemesinde [19] ısı akışı benzetimi problemi bu üç yaklaşımın her biriyle ve ayrı takımlar tarafından çözülmüştür. Aynı problem geleneksel NYP yaklaşımı ile de çözülerek bir karşılaştırma yapılmıştır. Elde edilen sonuçlara göre saf nesneye yönelimli yaklaşım NYP yaklaşımından en az üç kat daha kötü başarım vermiştir. Diğer bakış yaklaşımları ise NYP uygulaması ile hemen hemen aynı başarım sonuçları vermiştir. Takım üyelerinin yorumları şunlardır:

- BTP yaklaşımları çözümün karmaşıklığını arttırıcı ya da azaltıcı yönde büyük bir etki yapmamakta.
- BTP yaklaşımları yeniden kullanılabilirliği arttırmaktan çok, yazılım bakımı işlemlerini kolaylaştırma yönünde fayda sağlamakta.
- BTP yaklaşımlarının getirdiği ek yük göz ardı edilebilir seviyede azdır.

BTP teknolojilerinin temel problemi sadece görevlerin ayrılması ve kesişmesi değildir, yazılımı uygun ve ayrı parçalar halinde ele alıp bunları tutarlı bir sistem olarak birleştirmeyi de içerir.

BTP belli türlerdeki görevleri yazmayı ve değiştirmeyi kolaylaştırır. Diğer türden görevler ile kastedilenin, bazı problem bileşenlerini bakışlara ayırmanın çok zor olduğu ve dolayısıyla tasarım kararlarının net olarak verilemediği görevlerdir.

BTP teknolojilerin sağladığı mekanizmalardan çok, programcının bakışlara yönelik tasarımı nasıl kurduğu önemlidir. Bununla birlikte, görevlerin ayrılmasında etkinliği belirleyen eninde sonunda BTP teknolojisinin sağladığı mekanizmalar olmaktadır.

Çeşitli bakış tabanlı programlama araçları mevcut olmakla birlikte, çoğu halen geliştirilme aşamasındadır. Örnek olarak AspectJ [20] verilebilir. AspectJ şu önemli tanımlamaları yapar:

- Birleşme Noktası (*Join Points*): Program akışı içersindeki önemli noktalar
- Kesme Noktası (*Pointcuts*): Bir birleşme noktası kümesine ve bu noktalardaki çeşitli değerlere referans verebilmeyi sağlar.
- Tavsiye (*Advice*): Kesme noktalarına iliştirilebilen metot benzeri bir yapı.
- Bakış: Arakesit modülerliği sağlayan; Kesme noktaları, tavsiye ve sıradan üye tanımlarından oluşan modüler yapı. Bir bakışın birleşme noktasını içermesi, bakışın içerdiği kesme noktasının birleşme noktasını içermesinden kaynaklanır.

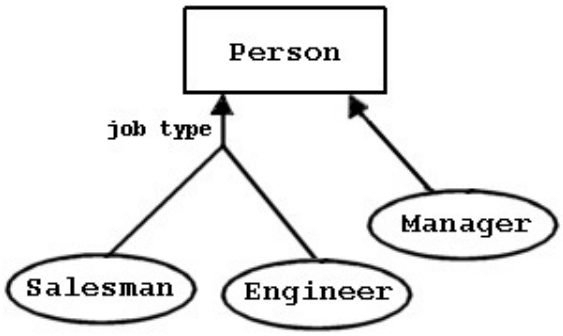
3.7.1 Değerlendirme

Değişik BTP yaklaşımlarının pratik vakalara uygulaması çalışmaları sürmektedir [19]. BTP ile gelen kazançlar her tür sistemin modellenmesinde avantajlar sağlayacaktır, ancak dinamik sistemlerin modellenmesindeki gereksinimlerin tümünü BTP tek başına karşılayamaz.

4. DİNAMİK SİSTEMLERİN MODELLENMESİ İÇİN ÖNERİLEN TASARIM KALIPLARI

Sınıf düzeyinde çoklu kalıtımın Bölüm 1.1'de anlatılan biçimde kullanımının, dinamik sistemlerin NYP ile modellenmesinde seçilecek en iyi yol olmadığı açıktır. NYP gibi güçlü bir yaklaşım, kendine yakışacak daha iyi çözümler elde etmek için kullanılabilmelidir. Nitekim, Fowler [21] henüz yayınlanmamış bir makalesinde dinamik sistemlerin modellenmesine yönelik çeşitli tasarım kalıplarını incelemiştir. Bu tasarım kalıplarından önemli olanlarının ayrıntılı olarak anlatılması, "Neden rol modelleri daha iyi bir çözümdür?" sorusuna cevap verebilmek için gereklidir. Fowler'ın çalışmasının geniş bir özeti bu bölümde sunulmuştur. Bölüm 9'da tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO; hem bu bölümde incelenen kalıplardan en özelleşmişi olan rol ilişkileri kalıbı ile, hem de literatürdeki diğer güncel rol modelleri ile başarım ve sahip olunan özellikler açısından karşılaştırılacaktır.

Tasarım kalıpları ile modellenecek örnek sistem Şekil 4.1'de görülebilir. Şekil 4.1'deki bileşenlerin açıklaması şu an için gerekli değildir, ilişkilerin bir bakışta anlaşılan görsel ifadesini kavramak bu bölümde anlatılanlar için yeterli olacaktır. Modellenen sistemde kişiler satış veya mühendislik bölümlerinde çalışmakla birlikte müdürlüğe de terfi edebileceklerdir.

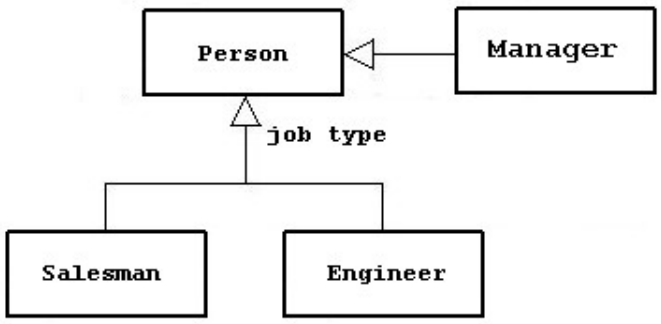


Şekil 4.1: Tasarım kalıpları ile modellenecek örnek sistemin kavramsal şeması

Tez çalışmasında kullanılan dil Java olarak seçildiği için tasarım kalıplarının anlatımında kullanılan kodlar Java dilinde olacaktır.

4.1 Rol Alt Tipleri (Role Subtype)

Modellenecek sistemin bileşenleri arasında hem bir takım benzerlikler hem de çeşitli farklılıklar bulunuyorsa kalıtımla alt tipler oluşturma en belirgin çözüm olacaktır. Şekil 4.1'deki örnek sistemin bu yolla modellenmesi sonucu Şekil 4.2'de görülen UML şeması elde edilecektir. Her tipin ortak davranışı **`Person`** üst tipinde gerçeklenir, her ek davranış ise alt tiplerden uygun olanına kodlanır. Bu alt tiplere rol alt tipi adı verilir. Bu kalıp alışılmış sınıf düzeyinde kalıtım kavramı ile de örtüşür: Her mühendis aynı zamanda bir insandır ancak ek özellikler kazanarak özelleşmiştir; bir uygulama bu özellikler ile ilgileniyorsa nesneleri birer mühendis, ilgilenmiyorsa kişi olarak ele alır. Bir insan hem bir mühendis hem de müdür ise her iki alt tipe birden aittir.



Şekil 4.2: Rol alt tiplerinin kullanımı ile modellemeyi örnekleyen UML şeması.

Bu noktada nesneye yönelim kavramları ile çelişkiye düşülmektedir. Sorun önceki paragrafın son cümlesinden kaynaklanır: Çoğu NYP dilinde tekli ve statik sınıflandırma desteklenirken söz konusu cümle çoklu ve dinamik sınıflandırma gerektirir. Tekli ve durağan sınıflandırmada bir nesne tek bir sınıfa aittir ve diğer sınıflardan kalıtım yoluyla çeşitli özellikler kazanabilir. Fakat rol alt tipleri tasarım kalıbı, üç yoldan birini kullanarak bu engeli aşabilir: İçsel bayrak, gizli iletim ve durum nesnesi. Bunların ayrıntılarına girmeden önce, tümünün gerçekleyeceği ve Şekil 4.2'nin gerektirdiği arayüzün verilmesi yerinde olacaktır. Şekil 4.2'den türetilen arayüz Şekil 4.3'te görülebilir.

```
interface Person {
  public String name();
  public void name(String newName);
  public Money salary ();
  public void salary (Money newSalary);
  public Money payAmount ();
  public void makeManager (); }
interface Engineer extends Person{
  public void numberOfPatents (int value);
  public int numberOfPatents ();   }
interface Salesman extends Person{
  public void numberOfSales (int numberOfSales);
  public int numberOfSales ();  }
interface Manager extends Person{
  public void budget (Money value);
  public Money budget (); }
```

Şekil 4.3: Rol alt tipi kullanımı kalıplarına ait örnekler için ortak arayüz.

4.1.1 İçsel Bayrak Yolu ile Rol Alt Tipleri (Role Subtype with Internal Flag)

İçsel bayrak çözümünde Şekil 4.3'teki dört arayüzü birden gerçekleyen bir tek **`PersonImpl`** sınıfı yazılır. Alt tipler arasında geçiş yapabilmek için ek komutlar gerekecektir, örneğimizde bunlar **`newSalesman`**, **`makeSalesman`** ve **`isSalesman`** olacaktır. Bu metotlar **`PersonImpl`** sınıfına eklenebilir. Satıcıya ait metotların sadece satıcı nesneleri üzerinde çalışmasını sağlamak için de **`requireIsSalesman`** metodu yazılmıştır. Böylelikle satıcı olmayan bir nesne için çalıştırılmak istenen satıcı metotları bir çalışma anı hatası ortaya çıkaracaktır. Detayları verilen bu çözümün sadece satıcı rolü için gerçeklenmesi örnek olarak Şekil 4.4'te verilmiştir.

Üst tipte tanımlanmakla birlikte alt tiplerde farklı gerçeklenen çok biçimli metotlar olabilir. **`Person`** arayüzünde tanımlanan **`payAmount`** metodu böyle bir örnektir. Çok biçimli metotların gerçeklemesi, tip tanımlayıcısına göre bir **`case`** bloğu içerisinden uygun metodu çağırılarak veya strateji tasarım kalıbı [22] kullanılarak yapılabilir. Böylesi türe göre dallanma yapılması NYP'de teşvik edilmese de, bu kullanım üst tip içine gizlenmiş olacağı için çok fazla sakınca oluşturmayacaktır. Anlatılan çözümün gerçeklemesi Şekil 4.5'te görülebilir.

İçsel bayrak çözümün güçlü yanı açık bir arayüz sağlamasıdır. Ancak sisteme yeni bir rol ekleneceği zaman üst sınıfın arayüzünün değiştirilmesi gereksinimi çözümün zayıf noktasıdır. Bu çözümün çok biçimli metotlara sağladığı destek sayesinde çoklu ve dinamik sınıflandırmanın desteklenmesi diğer güçlü yön olarak görülmektedir.

Ancak tüm rollerin arayüzlerinin tek gerçekleme sınıfında toplanarak git gide karmaşıklaşan bir sınıf oluşturması önemli bir dezavantaj oluşturur.

```
public class PersonImpFlag implements Person, Salesman,
    Engineer, Manager { // Implementing Salesman
  public static Salesman newSalesman (String name) {
    PersonImpFlag result;
    result = new PersonImpFlag (name);
    result.makeSalesman();
    return result;
  };
  public void makeSalesman () {
    _jobTitle = 1;
  };
  public boolean isSalesman () {
    return _jobTitle == 1;
  };
  public void numberOfSales (int value){
    requireIsSalesman () ;
    _numberOfSales = value;
  };
  public int numberOfSales () {
    requireIsSalesman ();
    return _numberOfSales;
  };
  private void requireIsSalesman () {
    if (! isSalesman())
    throw new PreconditionViolation ("Not a Salesman") ;
  };
  private int _numberOfSales;
  private int _jobTitle;
}
```

Şekil 4.4: Satıcı tipinin içsel bayraklar ile gerçeklenmesi.

```
public Money payAmount (){
  if (isSalesman()) return payAmountSalesman();
  if (isEngineer()) return payAmountEngineer();
  throw new PreconditionViolation ("Invalid Person");
};
private Money payAmountSalesman () {
  return _salary.add (Money.dollars(5).multiply
    (_numberOfSales));
};
private Money payAmountEngineer () {
  return _salary.add (Money.dollars(2).multiply
    (_numberOfPatents));
};
```

Şekil 4.5: Çok biçimli metotların içsel bayraklar ile gerçeklenmesi.

4.1.2 Gizli İletim Yolu ile Alt Rol Tipleri (Role Subtype with Hidden Delegate)

Gizli iletim yolu ile gerçeklemede alt tiplerin gerektirdiği ek veri ve davranış ayrı bir sınıf içinde gerçeklenir. Bu ayrı sınıf kullanıcıdan gizlenir: İstemci sınıf hiyerarşinin köküne ait metotları çağırır, kök sınıf ise gereken çağrıları gizli sınıfa iletir. Dolayısıyla rol arayüzünün tüm **`public`** metotları kök sınıfta da gerçeklenmelidir. Şekil 4.6'da bu gerçeklemenin **`Manager`** tipi için nasıl yapıldığı görülebilir. **`Manager`** tipinin **`public`** metotlarının **`Person`** tipindeki eşlenikleri önce mesajın alıcısının doğru rol tipi olup olmadığını kontrol eder ve tip doğru ise metodu gizli **`Manager`** tipine iletir.

```
public class PersonImpHD implements Person, Salesman,
     Engineer, Manager {
//implementing manager
  public void makeManager () {
    _manager = new ManagerImpHD();
  };
  public boolean isManager (){
    return (_manager != null);
  };
  private void requireIsManager () {
    if (! isManager())
      throw new PreconditionViolation ("Not a Manager") ;
  };
  public void budget (Money value) {
    requireIsManager();
    _manager.budget(value);
  };
  public Money budget () {
    requireIsManager ();
    return _manager.budget();
  };
  private ManagerImpHD _manager;
}
class ManagerImpHD {
  public ManagerImpHD () {
  };
  public void budget (Money value){
    _budget = value;
  };
  public Money budget (){
    return _budget;
  };
  private Money _budget;
}
```

Şekil 4.6: Yönetici tipinin gizli iletim yolu ile gerçeklenmesi.

Gizli iletim kullanıldığında alt rol tiplerine ait davranış kök sınıftan alt rol sınıfına taşınmış olur. Böylece kök sınıfın karmaşıklığı içsel bayraklar kullanımına göre azalmış olur. Eğer alt rol tiplerinin getirdiği ek özellik ve davranışlar fazla ise bu azalma çok önemli bir kazanç getirecektir. Fakat kök sınıf, hala, alt rol tiplerinin tümünün arayüzünü gerçeklemek ve mesajın gizli nesneye iletip iletilmemesine karar vermek zorundadır.

4.1.3 Durum Nesnesi ile Alt Rol Tipleri (Role Subtype with State Object)

Kök sınıfın karmaşıklığını daha da azaltmak için alt rol tipleri kalıbı durum nesnesi kalıbı ile [22] birlikte kullanılabilir. Bu durumda gizli rol sınıflarının tümünün birden üst sınıfı olan yeni bir gizli sınıf oluşturulur. Bu gizli rol üst sınıfı tüm tip testleri ve metot seçimlerinden sorumlu olacaktır. Kök sınıf hala alt rollerin arayüzüne sahip olmak zorundadır ancak söz konusu metotlar kökte çok basit olacaktır: Tek yapılması gereken rollere olan tüm çağrıları gizli üst rol sınıfına iletmektir. Metot seçimi için gizli üst rol sınıfında kalıtım ve çok biçimlilik kullanılabilir.

```
public class PersonImpHD implements Person, Salesman,
     Engineer, Manager {
  public static Salesman newSalesman (String name){
    PersonImpHD result;
    result = new PersonImpHD (name);
    result.makeSalesman();
    return result;
  };
  public void makeSalesman () {
    _job = new SalesmanJobHD();
  };
  public boolean isSalesman () {
    return _job.isSalesman();
  };
  public void numberOfSales (int value){
    _job.numberOfSales(value);
  };
  public int numberOfSales () {
    return _job.numberOfSales();
  };
  private JobHD _job;
}
```

Şekil 4.7: Satıcı arayüzünü gerçeklemek üzere bir şahıs sınıfının durum nesnesi kalıbı ile kodlanması

Şekil 4.7 ve 4.8'de satıcı rolü için durum nesnelerinin nasıl kullanılabileceği görülmektedir. Gizli üst rol sınıfı olan **`JobHD`** varsayılan davranışı sağlar. Örneğimizde bu davranış olası hataları bildirilmektir.

```
abstract public class JobHD {
  private void incorrectTypeError() {
    throw new PreconditionViolation("Incorrect Job Type");
  };
  public boolean isSalesman () {
    return false;
  };
  public void numberOfSales (int value) {
    incorrectTypeError();
  };
  public int numberOfSales (){
    incorrectTypeError();
    return 0; /*value not returned since exception is thrown,
    compiler needs return*/
  };
}
public class SalesmanJobHD extends JobHD {
  public boolean isSalesman () {
    return true;
  };
  public void numberOfSales (int value){
    _numberOfSales = value;
  };
  public int numberOfSales () {
    return _numberOfSales;
  };
  private int _numberOfSales = 0;
}
```

Şekil 4.8: Durum nesnesi kalıbı ile satıcı rolü için gizli üst ve alt rol sınıflarının tanımlanması

Durum nesnesi kalıbı özellikle Şekil 4.9'da verilen **`payAmount`** metodu için çok iyi çalışmaktadır. Bu özel durumda, durum nesnesi kök nesnenin bir üyesine erişmek zorundadır. İletim yapılırken kök nesnenin de ek bir parametre ile iletilmesi gerekli erişime olanak tanır.

Durum nesnesi kalıbı karşılıklı dışlamalı rollerin sayısı arttıkça daha yararlı olmaktadır. Örnek sistemimizde mühendis ve satıcı rolleri karşılıklı dışlamalı rollerdir: Bir insan bu görevlerden ya birine ya da ötekine sahip olabilir.

```
public class PersonImpHD implements Person, Salesman,
     Engineer, Manager {
  public Money payAmount (){
    return _job.payAmount(this).add(managerBonus());
  };
  //code for the rest of the class
}
abstract public class JobHD {
  abstract public Money payAmount (Person thePerson);
  //code for the rest of the class
}
public class SalesmanJobHD extends JobHD {
  public Money payAmount (Person thePerson) {
    return thePerson.salary().add (Money.dollars(5).multiply
        (_numberOfSales));
};
```

Şekil 4.9: Durum nesnesi kalıbı ile satıcı rolünün `payAmount` metodunun gerçeklenmesi

Alt rol tipleri kalıpları arasında durum nesneleri ile kullanım, çoklu ve dinamik sınıflandırmayı en iyi destekleyen kalıptır. Bu kalıpta kök sınıfın karmaşıklığı yok denilebilecek düzeye iner. Ancak gizli üst rol sınıfı, diğer kalıpların kök sınıfı kadar olmasa da, hala önemli bir karmaşıklık içermekte ve bu kalıbı kurmak için diğerlerinden daha fazla bir çaba harcamak gerekmektedir.

4.1.4 Hangi Alt Rol Tipi Kalıbı?

Alt rol tipleri kalıbının önceki bölümlerde anlatılan üç çeşidinin ortak yönü, son kullanıcı tarafında aynı arayüze sahip olmalarıdır. Böylelikle, son kullanıcılar etkilenmeden, bir kalıptan diğerine geçmek mümkün olacaktır. Eğer roller çok fazla özelleşmiyorsa içsel bayrak metodu tercih edilir. Aksi halde gizli iletim kalıbı ya da karşılıklı dışlamalı rollerin çokluğu halinde durum nesneleri kalıbı seçilebilir. Programcı arayüzün değişmemesi avantajını kullanarak en basit çözümle başlayabilir, modellenen sistem karmaşıklaştıkça daha karmaşık kalıplara bir sorun olmadan geçebilir.

4.1.5 Açık Rol Tipi İşareti (Explicit Type Method)

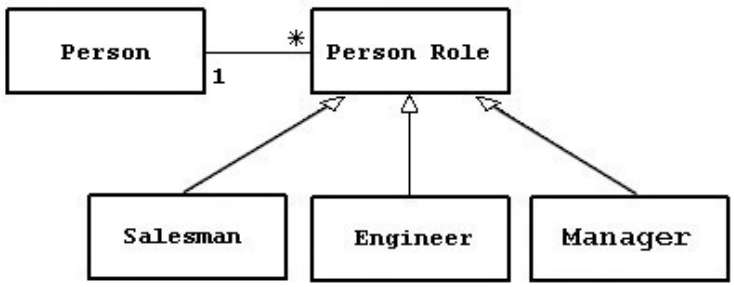
Alt rol tipleri kalıbının önceki bölümlerde anlatılan üç çeşidinde de rol tiplerine açık şekilde işaret edilmiştir. Örnek kodlar hep bu şekilde yazılmıştır. Bu yaklaşım açık bir arayüz sunar, ancak yeni rol tipi eklendiğinde kök sınıfın arayüzünün de

değişmesini gerektirir. Bu yaklaşım bir nesnenin istediğimiz rol nesnesi olup olmadığının farklı rol tiplerini anımsatan birden fazla metotla sınanmasını gerektirir. Kök sınıf bu sınama metotlarının tümünü içermek zorunda iken rol sınıflarının sadece kendine ait sınama metoduna sahip olması yetecektir. Şekil 4.2'deki örneğimizde **`Person`** sınıfının **`isSalesman`**, **`isEngineer`** ve **`isManager`** metotlarına sahip olması gerekir. Örneğin **`Salesman`** sınıfının ise adı geçen üç metottan sadece **`isSalesman`** metodunu içermesi yeterli olacaktır.

Açık rol tipi işaretinin alternatifi parametreli rol tipi işaretidir. Bu yaklaşım ise rol nesneleri tasarım kalıbında örneklenerek anlatılacaktır.

4.2 Rol Nesneleri (Role Object)

Rol nesneleri tasarım kalıbı ile yapılacak gerçekleme durum nesneleri yaklaşımına benzer. Durum nesneleri kullanıcıdan tamamen gizlenirken rol alt tipinde durum tam tersidir. Durum nesneleri kullanılırken bir role erişmek için hiyerarşi köküne mesaj gönderilir, kök de mesajı ilgili role iletir. Rol nesneleri kullanıldığında ise hiyerarşi köküne mesaj gönderilerek gerekli rol nesnesine bir işaretçi elde edilir, ardından rol nesnesinin istenilen metodu kullanılır. Şekil 4.1'deki örnek sistemin bu yolla modellenmesi sonucu Şekil 4.10'da görülen UML şeması elde edilecektir.



Şekil 4.10: Rol nesnelerinin kullanımı ile modellemeyi örnekleyen UML şeması

4.2.1 Parametreli Rol Tipi İşareti (Parameterized Type Method)

Bu bölümde rol nesneleri tasarım kalıbı, parametreli rol tipi işareti yaklaşımı ile kullanılacaktır. Bu sefer rol tipleri için ayrı ayrı sınama metotları yazmak yerine bir tek **`hasType(String)`** metodu kullanılacaktır. Roller arasında geçiş ise **`beType(String)`** metodu ile yapılacaktır. Bu yaklaşımın avantajı yeni roller

eklenince kök sınıfın arayüzünün değişmesine gerek kalmamasıdır. Ancak bunun karşılığında arayüzün açıklığı azalmış olur ve derleyicinin tip kontrolü yapması mümkün olmaz. Örnek sistemimizdeki kök sınıfın satıcı rolünün rol nesneleri tasarım kalıbı ve parametreli rol tipi işareti kullanılarak gerçeklenmesi Şekil 4.11'de görülebilir. Bu sınıflardan nesnelerin kullanımı ise Şekil 4.12'de örneklenmiştir.

```
class Person {
  public void addRole(PersonRole value) {
    _roles.addElement(value);
  };
  public PersonRole roleOf(String roleName) {
    Enumeration e = _roles.elements();
    while (e.hasMoreElements()) {
      PersonRole each = (PersonRole) e.nextElement();
      if (each.hasType(roleName)) return each;
    };
    return null;
  };
  private Vector _roles = new Vector();
  //code for the rest of the class
}
public class PersonRole{
  public boolean hasType (String value) {
    return false;
  };
  //code for the rest of the role
}
public class Salesman extends PersonRole{
  public boolean hasType (String value) {
    if (value.equalsIgnoreCase("Salesman")) return true;
    if (value.equalsIgnoreCase("JobRole")) return true;
    else return super.hasType(value);
  };
  public void numberOfSales (int value){
    _numberOfSales = value;
  };
  public int numberOfSales () {
    return _numberOfSales;
  };
  private int _numberOfSales = 0;
}
```

Şekil 4.11: Rol nesneleri kalıbı ve parametreli işaretleme kullanılan örnek kök ve rol sınıfları

```
Person subject;
Salesman subjectSalesman=(Salesman)subject.roleOf("Salesman");
//Person.roleOf ile ilgili rol nesnesine erişildi.
subjectSalesman.numberOfSales(50);
//Erişilen rol nesnesi kullanıldı.
```

Şekil 4.12: Örnek kök ve rol nesnelerinin kullanımı

4.2.2 Roller Arasındaki Kuralların Yürütülmesi

Rol nesnelerinin bu şekilde kullanımı daha esnek olmakla beraber, roller arasında var olabilecek kuralları yürütmekte rol alt tipleri kullanımına göre daha zayıf kalmaktadır. Örneğin Şekil 4.1 ve Şekil 4.2'de verilen şemalarda bir kişinin aynı anda ya satıcı ya mühendis olabileceği görülmektedir. Ayrıca bir kişi bu iki rolden hangisine sahip olursa olsun, ek olarak bir de yönetici rolüne sahip olabilir. Bu gibi kuralları korumak için ek önlemler alınmalıdır. Bu önlemlerden biri, **`addRole`** metodunu yeni kazanılan bir rolün kurallara uygun olup olmadığını kontrol edecek şekilde değiştirmektir. Bu yaklaşım Şekil 4.13'te gösterilmiştir.

```
class Person {
  public void addRole(PersonRole value)throws CannotAddRole {
    if (! canAddRole(value)) throw new CannotAddRole();
    _roles.addElement(value);
  };
  private boolean canAddRole(PersonRole value){
    if (value.hasType("JobRole")){
      Enumeration e = _roles.elements();
      while (e.hasMoreElements()) {
        PersonRole each = (PersonRole) e.nextElement();
        if (each.hasType("JobRole")) return false;
      };
    };
    return true;
  };
  //code for the rest of the class
}
```

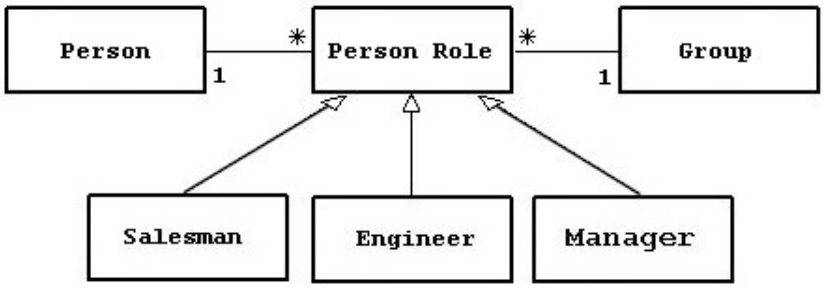
Şekil 4.13: Rol nesneleri kalıbında roller arasındaki kurallarının korunmasına örnek.

Roller arasındaki kuralları korumak için başka yollar da düşünülebilir. Örneğin kuralların her rol eklemesinde kök sınıf tarafından kontrol edilmesi yerine, kurallar strateji tasarım kalıbı [22] kullanılarak diğer bir sınıf tarafından da kontrol edilebilir. Tez çalışması kapsamında gerçeklenen JAWIRO rol modelinde bu yaklaşım kullanılmıştır.

4.3 Rol İlişkileri (Role Relationship)

Modellenen sistemde birden fazla grup olduğu durumlarda, roller de farklı gruplara ait olabilir ve bir nesne birden fazla gruba ait rollere sahip olabilir. Aynı tip rolün farklı gruplara ait birden fazla örneğinin aynı kök nesne ile ilişkilendirilmesi, böyle sistemler için bir gereksinim olacaktır. Şekil 4.1'deki örnek sistem üzerinden

anlatılırsa bu kalıptaki model, şirket içinde farklı gruplar olduğu ve bir kişinin aynı anda birden fazla gruba ait rolleri oynayabileceği şekilde genişletilmiş olacaktır. Bu durumda Şekil 4.14'te gösterilen sistem elde edilecektir.



Şekil 4.14: Rol ilişkileri kullanımı ile modellemeyi örnekleyen UML şeması

```
class Person {
  public void addRole(PersonRole value)throws CannotAddRole {
    _roles.addElement(value);
  };
  public PersonRole roleOf(String roleName, Group group) {
    Enumeration e = _roles.elements();
    while (e.hasMoreElements()) {
      PersonRole each = (PersonRole) e.nextElement();
      if ((each.hasType(roleName))&&(each.group()==group))
        return each;
    };
    return null;
  };
  private Vector _roles = new Vector();
  //code for the rest of the class
}
public class PersonRole{
  protected PersonRole (Group group){
    _group = group;
  };
  public Group group(){
    return _group;
  };
  protected Group _group;
  //code for the rest of the role
}
public class Salesman extends PersonRole{
  public Salesman (Group group){
    super (group);
  };
  //code for the rest of the role
}
```

Şekil 4.15: Rol ilişkileri kalıbı kullanılan kök ve rol sınıfları

Rol ilişkileri kalıbının gerçeklenmesi ve kullanımı rol nesneleri kalıbına çok benzer. Aradaki temel fark rolleri oluşturmanın ve rollere erişimin, rolün ait olduğu grubu gösteren bir parametre ile yapılmasıdır. Şekil 4.15'te satıcı rolü için rol ilişkileri kalıbı kullanılarak yapılan gerçekleme görülebilir.

4.4 İncelenen Kalıpların Karşılaştırılması

Rol alt tipleri kalıbının en önemli avantajı basit ve açık bir arayüz sunmasıdır. Sistemde çok fazla rol yoksa, yeni roller sıkça ortaya çıkmıyorsa, rollerin sahipliği için önemli ve belirgin kurallar varsa rol alt tipleri kullanılabilir. Bu noktada NYP'nin tekli ve durağan sınıflandırmasını aşmanın şu üç yolundan birine karar vermek gerekir: İçsel bayrak, gizli iletim ve durum nesnesi tasarım kalıpları. Bu kararın nasıl verileceği incelenmişti.

Rol nesneleri kalıbı, rollerin daha çok olduğu veya daha sık yeni rollerin ortaya çıktığı sistemlerde, rol alt tipleri kalıbına tercih edilir. Böyle sistemlerde rol alt tiplerini kullanmayı sürdürmek çok fazla çaba gerektirecektir.

Rol ilişkileri kalıbı rol nesneleri kalıbının bir üst kümesi olmakla birlikte sadece bu gözle görülmemelidir. Bu kalıpta rol nesnelerinin grup ayırt edicisine göre ele alınması, kök nesnelerin zaman içerisinde oynadığı rollerin kaydının tutulması gerektiği durumlarda avantaj sağlayacaktır.

Rollerin hangi kalıpla modelleneceği kararından bağımsız olarak, rol tiplerine nasıl işaret edileceğine karar vermek gerekir. Bu noktada açık rol tipi işaretleri veya parametreli rol tipi işaretleri kalıplarından biri kullanılabilir. Açık rol tipi işaretleri kolay anlaşılır bir arayüz sunmakla birlikte eklenen her yeni rol arayüzün değişmesini gerektirir. Parametreli rol tipi işaretleri ise tam tersine anlaşılması daha zor ve derleyici kontrolüne olanak vermeyen bir arayüz sunar ancak arayüz sisteme yeni eklenecek rollerden bağımsızdır.

Özetle her durum için en iyi çözüm olarak nitelendirilebilecek bir tasarım kalıbı bulunmamaktadır. Ayrıca dinamik sistemlerin bu kalıplarla karşılanamayacak gereksinimleri mevcuttur: Bölüm 5.3 ve Bölüm 9.1'de görüleceği üzere, incelenen tasarım kalıpları rollerin temel özelliklerinin tümünü birden desteklemez.

5. ROL MODELLERİ VE ROL TABANLI PROGRAMLAMA

Bu bölümde dinamik sistemlerin etkili ve kolay biçimde modellenebilmesi için önerilen bir çözüm olan rol modelleri, genel bilgiler ve rollerin özellikleri ile birlikte tanıtılacaktır. Rollerin hiyerarşik düzeninin yapısına ve rol modellerinin diğer alternatif çözümlere göre avantajlarına da bu bölümde yer verilmiştir.

5.1 Giriş ve Tanımlar

NYP dilleri nesnenin kimliğini tek bir tipe kalıcı ve değişmez şekilde bağlar. NYP'nin bu özelliği sınıflar ve nesneleri arasında eş biçimli ve tutarlı bir yapı sağlama avantajını getirse de, dinamik sistemlerin modellenmesinde giriş bölümünde incelenen güçlükleri de ortaya çıkarır. Dinamik sistemlerin etkin biçimde modellenebilmesinin yolu, zaman içerisinde değişip evrimleşerek sisteme dinamizm kazandıran nesnelerin iyi bir şekilde modellenmesinden geçer.

Bölüm 1.2'de nedenleri anlatıldığı gibi, evrimleşen varlıkları modellerken nesne düzeyinde özelleştirme yapılması, sınıf düzeyinde özelleştirmeye göre daha iyi bir yaklaşımdır. Bu durumda bir gerçek dünya varlığı, her biri yükümlü olduğu görevlerden birini yürüten bir rol nesnesi olmak üzere, birden fazla nesne ile temsil edilecektir. Roller, Kristensen tarafından şu şekilde tanımlanır: "Bir nesnenin bir rolü, bu nesnenin diğer nesnelerin beklediği gibi davranabilmesi için gereken bir özellikler kümesidir [23]".

Rol tabanlı programlamanın (RTP) temeli rol kullanımına dayanmaktadır. RTP'de nesneler yeni roller alarak evrimleşir. Örnek düzeyinde gerçekleşen bu özelleşmeye "nesne düzeyinde kalıtım" adı verilir. Nesne düzeyinde kalıtım ile sınıf düzeyinde kalıtım birbirini tamamlar.

Sınıf düzeyinde kalıtım güçlü bir biçimde "aynısıdır" ilişkisini modellerken nesne düzeyinde kalıtım "bir parçasıdır" ilişkisini başarıyla modeller [4]. Gerçek dünya sistemlerini modellerken her iki ilişkinin kullanılması da gerekli olduğundan, bir

nesne yönelimli ortamda iki tip kalıtım da bir arada kullanılabilmelidir. Bu nedenle bir çok başarılı rol modeli mevcut bir NYP dilini genişletecek şekilde gerçeklenmiştir. Böylelikle sınıf düzeyinde özelleşmeye dayalı olan NYP dillerine rol modelleri sayesinde nesne düzeyinde özelleştirme yeteneği kazandırılır.

Rollerin tasarımı için bir tarz ve rollerin kullanımı için çeşitli işlevler sunan yazılımlara rol modeli adı verilir. Literatürdeki önemli rol modelleri bir sonraki bölümde ayrıntılı olarak incelenecektir.

5.2 Rollerin Temel ve İleri Düzey Özellikleri

Değişik araştırmacılar, rollerin sahip olması gereken özellikleri birbirlerinden farklı şekilde tanımlamışlardır [22, 42]. Tez çalışması boyunca yapılan araştırmalar ve değerlendirmeler sonucunda, bir rol modelinin sağlaması gereken özellikler bu tez çalışmasında temel ve ileri düzey olmak üzere ikiye ayrılmıştır. Bu bölümde rollerin sahip olması gereken özellikler örneklerle incelenecektir.

5.2.1 Rollerin Temel Özellikleri

Rollerin temel özellikleri, değişik araştırmacıların üzerinde genel olarak anlaştıkları ve 6. Bölüm'de de incelenen literatürdeki rol modellerinin hemen hepsi tarafından gerçeklenen özelliklerdir.

5.2.1.1 Bakış Açısı

Bir gerçek dünya varlığının oynadığı rollerin her biri, o varlığa bir bakış açısı oluşturur. Bir nesneye erişim mevcut bakış açısı, bir başka deyişle o anda kullanılan rolü ile sınırlıdır. Ancak rollerin diğer bir temel özelliği olan rol geçişi sayesinde, mevcut rol içerisinden başka bir rolün üyelerine erişim sağlanılabilir.

5.2.1.2 Rol Hiyerarşisi Kavramı

Rollerin ikinci temel özelliği, bir rolün başka bir rolü oynayabilmesidir. Bu özellik, rollerin çeşitli hiyerarşik düzenler oluşturacak şekilde ilişkilendirilebilmesini gerektirir.

Rol hiyerarşisi, yaprak ve ara düğümlerinde rol tipi (sınıfı) adı verilen özel bir tipten (sınıftan) nesneleri içeren bir ağaçtır. Bu ağacın kökünü ise nesnenin zamanla değişmeyecek özellikleri oluşturur. Diğer düğümler nesnenin zaman içersinde kazanıp kaybedebileceği özellikleri simgeler. Bir varlık yeni bir rol kazandığı zaman uygun rol sınıfından bir nesne oluşturulup ağaca eklenir. Bir rol terk edildiğinde ise basitçe ilgili nesne yok edilir. Rol kazanımı ve terki de rollerin diğer temel özellikleri arasındadır.

Sınıf hiyerarşisinin evrimleşen nesneleri modellemede yetersiz kaldığı Bölüm 1.1'de gösterilmişti. Nesnelerin dinamik olarak tip değiştirdiği durumlarda kalıtım ve uzmanlaşmanın sınıf yerine nesne düzeyinde uygulanması daha uygundur [2]. Bir rol hiyerarşisi rol tiplerinden (sınıflarından) oluşan bir ağaçtır ve ağacın kökü olan bir varlık tipinin nasıl evrimleşeceğini belirler. Şema olarak gösterimde rol hiyerarşisi ile sınıf hiyerarşisi birbirine benzer. Her rol tipi kendine ait üyeleri ve metotları tanımlar. Sınıf hiyerarşisinden farklı olarak bir alt rol tipi üst rolün üye ve metotlarını miras almaz. Kalıtım sınıf yerine nesne düzeyinde yapılır. Nesne düzeyinde kalıtım sayesinde bir alt tip, üst tipin gereksinim duymayacağı özelliklerini almak zorunda kalmaz.

Rol hiyerarşisi sayesinde nesneye yeni bir tip eklenmesi eski tip üzerinden erişimi değiştirmez. Ancak rollerin diğer özellikleri arasında görüleceği üzere, bir rol terk edileceği veya başka bir sahibe aktarılacağı zaman, hiyerarşi ağacında bu rolün çocukları da terk edilmek veya aktarılmak zorunda kalınır. Doğrusu da budur; terk edilen rolün çocukları da o rolle mantıksal açıdan bağlantılı olacağından çocuk rollerin de terk edilmesinin zorunlu olduğu ortaya çıkar.

Bir gerçek dünya varlığı ağacın kökünde bulunan nesne ile diğer düğümlerinde bulunan ve o anda sahip olduğu tüm rolleri içeren rol tipleri ile temsil edilir. Rol tipleri bir varlığı tek başına temsil edemez. Tüm roller en azından bir üst rolünden ve hiyerarşi kökünden doğrudan haberdar olmalıdır.

Rol hiyerarşisinin sağladığı avantajlar şunlardır:

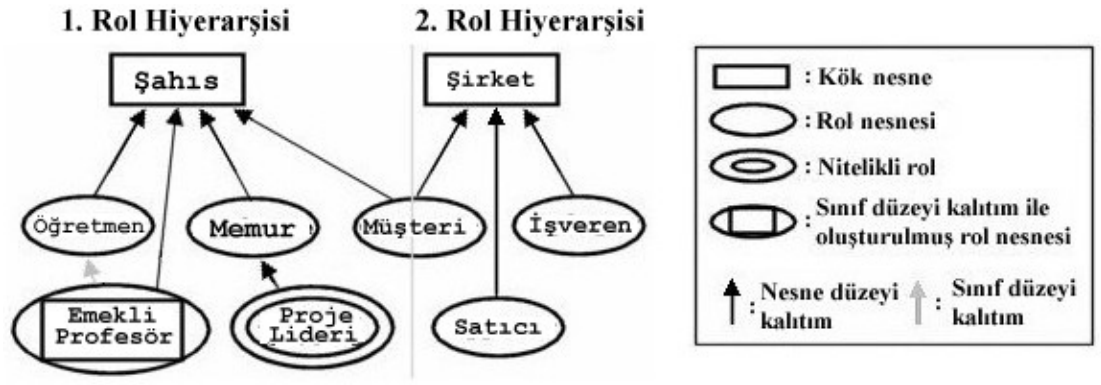
- Değişik roller arasında bilgi paylaşımı sağlanır. Birden fazla rol tarafından paylaşılan özellikler bunların hiyerarşideki ortak atasında temsil edilir. Sınıf hiyerarşilerinin aksine bu paylaşım sınıf seviyesi yerine nesne seviyesindedir.

- Evrimleşen nesneleri modellemek ve farklılaşmalarını izlemek kolaydır. Bir varlık yeni bir rol kazandığında basitçe o rol sınıfının bir örneği oluşturulur, varlık rolü terk ettiğinde de oluşturulan örnek silinir.

- Çoklu kalıtıma gerek kalmadan bir varlığın birbirinden bağımsız çok sayıda rolü modellenebilir.

- Rollerin diğer temel gereksinimlerini en esnek biçimde destekler. Örneğin bir rol bir başka sahibe aktarılacağı zaman alt rolleri de kolaylıkla yeni sahibe taşınabilir.

Şekil 5.1'de bileşenlerinin açıklaması ile birlikte iki örnek rol hiyerarşisi görülebilir. 1. hiyerarşide bir şahıs nesnesi dört farklı tip alt role sahip olabilir. Bir şahıs önce memur rolünü üstlenmeden proje lideri rolünü üstlenemez. Bununla birlikte, bir memur birden fazla projenin lideri rolünü üstlenebilir. Emekli profesör rolü ise öğretmen rolünden sınıf düzeyinde kalıtım ile türetilmiştir. Bir üst rol (veya kök) ile alt rol arasında rol sahipliği veya başka bir deyişle nesne düzeyi kalıtım ilişkisi vardır. Müşteri rolü ise hem şahıs hem de şirket örnekleri tarafından oynanılabilir. Şekil 5.1'de verilen hiyerarşiler rollerin ileride anlatılacak diğer temel ve ileri düzey özelliklerinin çoğuna örnek oluşturmaktadır.



Şekil 5.1: Örnek bir rol hiyerarşisi.

Bazı rol modelleri, rollerin hiyerarşik bir yapıda gösterimi yerine küme şeklinde gösterimini kullanmaktadır. Başka bir deyişle bir aktör ve bunun rolleri olmak üzere sadece iki düzeyden oluşmuş bir rol hiyerarşisi ağacı kullanımı söz konusudur. Hiyerarşik gösterim, küme gösterimine göre daha güçlü ve esnek bir yapı sağlar. Hiyerarşik gösterimin gücü, içinde barındırdığı ek bilgiden kaynaklanır. Örneğin

Şekil 5.1'de bir insanın bir proje yöneticisi olmadan önce bir işte çalışıyor olması gerektiği, bu bilginin ayrıca anlatılmasına gerek kalmadan anlaşılabilmektedir.

Bir rol hiyerarşisinin en çok özelleşmiş katılımcısı, hiyerarşinin en derinindeki katılımcıdır. Rollerin gösteriminde hiyerarşik düzen yerine küme düzeni kullanılıyorsa en çok özelleşmiş katılımcı en son kazanılan rol olacaktır.

JAWIRO rol modeli gerçeklenirken, rollerin temel ve ileri düzey özelliklerinin bu tür bir hiyerarşik düzen ile daha kolay ve esnek bir biçimde gerçeklenebileceği görülmüştür. Üstelik altıncı bölümde incelenen rol modellerinden bazılarında kullanılan küme biçiminde gösterimin diğer bazı tasarım ayrıntıları ile birleşerek rollerin kimi temel özelliklerinin kullanılamamasına neden olduğu belirlenmiştir.

5.2.1.3 Rol Kazanımı ve Terki

Roller birbirlerinden bağımsız olarak kazanılabilmeli ve terk edilebilmelidir. Örneğin bir şahıs, memur olup olmamasından bağımsız olarak müşteri rolü edinebilmelidir. Hizmet süresi dolan bir memur da emekli olabilmelidir. Bir rol terk edildiği zaman alt rolleri de terk edilmelidir, çünkü terk edilen rolün çocukları da o rolle mantıksal açıdan bağlantılı olacaktır. Terk edilmiş rollere, rol modelinin sağladığı komutlar ile erişilememelidir.

5.2.1.4 Birden Fazla Nesne ile Gösterim

Bir gerçek dünya nesnesinin sahip olduğu rollerin tümü ile tanımlandığı gerçeği korunmalıdır. Bu amaçla her rol nesnesi sahibinden, kendi rollerinden ve hiyerarşi kökünden haberdar olmalıdır. Bir rol, sahibi olmadan bir anlam taşımaz. Örneğin bir öğretmen rolü tek başına, o rolü oynayan şahıs olmadan anlamsızdır.

5.2.1.5 Rol Geçişi

Bir varlık herhangi bir rolüne herhangi bir anda geçiş yapabilmelidir. Bir nesnenin rollerinin birine olan referanstan nesnenin diğer rollerine de erişimin mümkün olması söz konusudur. Bir rol nesnesi katılımcısı olduğu hiyerarşide bulunan ve önceki maddede anlatılan şekilde doğrudan haberdar olduğu katılımcıların dışındaki rollere de erişebilmelidir. Örneğin elimizde bir şahsın öğretmen rolünü gösteren işaretçi bulunuyorsa, bu işaretçi üzerinden varsa proje lideri rolüne erişilebilmelidir.

Rol geçişi için geçiş yapılacak rolün tipinin bilinmesi gerekmektedir. Rol geçişi, bir varlığın o anda sadece geçiş yapılan rolü oynaması ve diğer rollerinin etkinliklerini kaybetmesi değildir. Bir varlık herhangi bir anda, kendisini simgeleyen rol hiyerarşisinde o anda bulunan tüm rolleri oynamaktadır. Rol geçişinin ardından kullanıcının elinde, gerçek dünya varlığının geçiş yapılan rolünü gösteren bir işaretçi olacaktır. Bu işaretçi kullanılarak varlığın geçiş yapılan role göre davranması sağlanır. Başka bir deyişle; yapılması gereken eylemi yürütecek mesajlar, o mesajı anlayabilecek olan rol nesnesine kullanıcı tarafından gönderilir. Bunun için kullanıcı hangi rollerin hangi eylemleri yürütebileceğini önceden bilmelidir. Bir gerçek dünya varlığına o anda oynadığı rollerin tiplerinden bağımsız olarak mesajların gönderilmesi, rollerin ileri düzey özelliklerinden olan ve Bölüm 5.2.2.5'te anlatılan ad ile üye erişimi yeteneği kullanılarak yapılabilir.

5.2.1.6 Sahibi Üzerinden Üye Erişimi

Bir nesnenin bir rolü, diğer rollerinin üyelerine, üstteki iki özellikten biri sayesinde erişebilmelidir. Bir başka deyişle; belli bir rol aynı hiyerarşideki bilinen başka bir rolün, istenilen üye değişkenine erişebilmeli veya istenilen üye metodunu çalıştırabilmelidir. Örneğin elimizde bir şahsın öğretmen rolünü gösteren işaretçi bulunuyorsa, bu işaretçi üzerinden müşteri rolünün satın al komutu çalıştırılabilir.

5.2.1.7 Çakışma Engeli

Değişik roller bir çakışma olmaksızın aynı adlı üye metot ve değişkenlere sahip olabilmelidir. Şekil 5.1'deki örnekte, şahıs ve memur sınıflarının ikisi de telefon numarası üyesine sahip olabilir. Bu durumda şahıs sınıfının telefon numarası üyesi kişinin ev telefonunu simgelerken, memur sınıfının telefon numarası üyesi ise kişinin iş telefonunu simgeler. Rollerin önceki iki temel özelliği olan rol geçişi ve sahibi üzerinden üye erişimi aracılığı ile, doğru bağlamda doğru üyeye erişim her zaman için mümkün olabilir.

5.2.1.8 Sınıf Düzeyi Kalıtım ile Nesne Düzeyi Kalıtımın Birlikte Kullanımı

Rol modelleri söz konusu olduğunda, nesne düzeyi kalıtım sınıf düzeyi kalıtımın yerini almaz. Aksine, nesne düzeyi kalıtım sınıf düzeyi kalıtımı tamamlar: Gerçek dünyada birlikte görülebilen bu iki tür kalıtım ilişkisi rol modellerinde de birlikte

kullanılabilmelidir. Şekil 5.1'de öğretmen rolünden sınıf düzeyi kalıtımla türetilen emekli profesör rolü aynı anda bir şahsın rolü olarak nesne düzeyi kalıtım ilişkisine girebilmektedir. Böylece emekli profesör rolü rollerin bu temel özelliğinin bir örneğini oluşturur.

5.2.1.9 Rol Varlığı Kontrolü

Varlıklar herhangi bir rol tipini oynayıp oynamadıkları hakkında sorgulanabilmelidir. Bu sorgulama çalışma anında yapılabilmelidir. Rol kazanımı ve terki ile dinamik olarak evrimleşen bir sistemde, bir varlığın belirli bir anda belirli bir rolü oynayıp oynamadığının bilinmesi her an gerekebilir. Bir rol geçişi işlemi yapmadan önce, geçilmek istenen rolün gerçekten oynanıp oynanmadığını sınamak yerinde bir davranış olacaktır.

5.2.1.10 Nitelikli Roller

Bir varlık aynı rol tipinin birden fazla örneğine, bir başka deyişle aynı rol sınıfından türetilmiş birden fazla rol nesnesine sahip olabilmelidir. Böyle rollere bu çalışmada nitelikli rol adı verilir. Nitelikli roller birbirlerinden bir niteleyici ile ayrılırlar. Bir rol hiyerarşisinde aynı tür rolden yalnızca bir katılımcı bulunabilecekken, niteleyicileri farklı olmak koşulu ile aynı rol hiyerarşisinde aynı nitelikli rol türünden birden fazla örnek bulunabilir.

Her ne kadar isteyen sıradan bir rolün birden fazla örneğini oluşturup aynı varlığa ilişkilendirebilse de bu durumda benzer roller arasında ortak bir ayırt edici özelliğin ne olduğu bilinemez. Bu belirsizlik de roller arasında biçimsel bir şekilde geçiş yapmayı zorlaştırır. Bu nedenle bir niteleyici ile türdeşlerinden ayrılabilen ve normal rollerden farklı olan özel bir rol türüne gereksinim vardır. Nitelikli roller de bu gereksinimi yanıtlar.

5.2.2 Rollerin İleri Düzey Özellikleri

Roller, dinamik sistemlerin modellenmesinde çok kullanışlı olan ve JAWIRO ile kullanılabilen ileri düzey özelliklere de sahiptir. Bu ileri düzey özellikler, gereklilikleri ve kullanım şekilleri daha detaylı inceleme gerektirdiği için, Bölüm 8.1.2'de örnekler yardımıyla ve JAWIRO ile kullanım şekilleri gösterilerek

anlatılacaktır. Yine de rollerin ileri düzey özelliklerine bu bölümde kısaca değinilmiştir. Rollerin bu bölümde listelenen ileri düzey özelliklerinin son altı tanesi başka hiçbir çalışmada tanımlanmamıştır ve Bölüm 1.3.7 ile Bölüm 8.2.6'da anlatılan başarım iyileştirmesiyle birlikte tez çalışmasının özgün katkılarını oluşturmaktadır.

5.2.2.1 Aykırı Rol İlişkilerinin Önlenmesi

Rol ilişkilerindeki aykırılıklar, veya bir başka deyişle anormallikler, aynı rol örneğinin aynı anda birden fazla sahibinin bulunması gibi olağandışı ilişkiler ve modellenen sistemin izin vermediği ilişkiler olmak üzere iki gruba ayrılabilir. Örneğin bir projenin birden fazla lideri olamaz; bir başka deyişle aynı proje lideri rolünü birden fazla memur oynayamaz. Bu olağandışı ilişki, rollerin bir sonraki ileri düzey özelliği olan nesne düzeyinde çoklu kalıtım ile karıştırılmamalıdır.

Olağandışı ilişkilerin yanı sıra, modellenen sistemin kurallarına ters düşen ilişkiler de kurulmaya çalışılabilir. Örneğin bir memur rolüne işveren rolü eklenmesi Şekil 5.1'de verilen sisteme ters düşecektir. Bir yazılımı hazırlayanlar ile kullananların farklı kişiler olmasından kaçınılamaz. Kullanıcıların zamanla sistemin yapısına aykırı işler yapmamaları için anormal ilişkilerin önlenmesi bu nedenle bir zorunluluk olarak karşımıza çıkar.

5.2.2.2 Kalıcılık

Kalıcılık özelliğine sahip olmayan programlama dillerinde tanımlanan nesnelerin yaşam süreci, yazılan programın çalışma süreci ile sınırlıdır. Kalıcılık özelliğine sahip dillerde ise tanımlanan nesnelerin durum bilgisi program sonlanmadan önce veya programcının istediği anlarda kalıcı bir ortama kaydedilerek saklanır. Uygulama programı yeniden çalıştırıldığında veya programcının istediği anda nesnelerin en son durumu kalıcı ortamdan yüklenir ve nesneler yaşam sürecine kaldıkları yerden devam ederler. Kalıcılık yeteneğine sahip bir rol modeli bütün bir rol hiyerarşisini kalıcı ortama saklayabilir ve sonradan tekrar kullanmak için kalıcı ortamdan geri yükleyebilir.

5.2.2.3 Rol Aktarımı

Bir rol mevcut sahibinden bir başka sahibe, alt rollerini kaybetmeden aktarılabilmelidir. Örneğin bir memur, liderliğini üstlendiği bir projenin sorumluluğunu bir başka meslektaşına devredebilir.

Rol aktarımı, eski ve yeni sahiplerin aynı veya farklı rol hiyerarşilerinde olup olmaması ile kısıtlanmamalıdır. Bu kısıtlama programcı tarafından başka yöntemlerle de yürütülebilir. Rolllerin bir sonraki bölümde anlatılan ileri düzey özelliği olan anormal ilişkilerin önlenmesi yeteneği kullanılarak da istenen kısıtlama yürütülebilir.

5.2.2.4 Nesne Düzeyinde Çoklu Kalıtımın Desteklenmesi

Nesne düzeyi çoklu kalıtımın desteklenmesi sayesinde, bir rol örneği farklı sınıfların rol örnekleri tarafından oynanabilir. Modellenen gerçek dünya sistemi bu yeteneğe gereksinim duyabilir. Şekil 5.1'deki sistemde hem bir şahıs hem de bir şirket, bir satıcının müşterisi olabilir.

Bir rolün bir anda yalnız bir sahibi olabileceğinden nesne düzeyinde çoklu kalıtım mantıksal bir belirsizliğe yol açmayacaktır. Ancak Smalltalk gibi dillerin aksine Java gibi kuvvetli tiplemeye sahip dillerde, bazı durumlarda tipleme belirsizliği ortaya çıkabilir. Söz konusu belirsizlik, alt rolün üst role ait bir üyeye erişmesi gereken durumlarda oluşacaktır. Çalışma anında üst rolün tipi bilinemeyeceği için istenilen üyeye erişim mümkün olmayacaktır. Bu sorun üst rollerin aynı arayüzü gerçeklemesi veya erişilecek üyenin ayrı bir role taşınarak o rol üzerinden kullanılması gibi yollarla çözülebilir. Rollerin bir sonraki bölümde anlatılacak ileri düzey özelliği olan ad ile üye erişimi de bu belirsizliği çözmenin yollarından birini oluşturur.

5.2.2.5 Ad ile Üye Erişimi

Bir rol hiyerarşisinde bulunan nesnelerden herhangi birisinin istenilen bir üye değişkeni veya metoduna, sahibine doğrudan bir referans olmadan, sadece o üyenin adı belirtilerek erişilebilir. Aynı ada sahip birden fazla üye varsa, en çok özelleşmiş üyeye erişim sağlanmalıdır. Bir rol hiyerarşisinin en çok özelleşmiş katılımcısının, hiyerarşinin en derinindeki üyesi olduğu hatırlatılır.

Rollerin temel özelliklerinden olan rol geçişi sayesinde sahibi üzerinden üye erişiminden farklı olarak; ad ile üye erişiminde, kullanılmak istenen üye değişken veya metodun rol hiyerarşisinin hangi katılımcısında olduğunu bilmek gereksizdir. Bu esneklik, bir önceki bölümde anlatılan nesne düzeyinde çoklu kalıtımın doğurabileceği tipleme belirsizliğini ortadan kaldırır. Şekil 5.1'deki sistemde; müşteri rolünün satın al metodu, hiyerarşi kökünün bütçe üyesinin değerini değiştiren harcama metoduna erişmek zorundadır. Fakat çalışma anında o müşterinin bir şahıs mı yoksa bir şirket mi olduğu belli olmayabilir. Bu durumda müşteri nesnesine "hiyerarşideki harcama metodu her neredeyse onu bul ve çalıştır" gibi bir komut vermek, oluşan tipleme sorununu çözecektir.

Bir gerçek dünya varlığının o anda sahip olduğu rollere göre davranmasının iki yolu vardır. Bunlardan birincisi, rollerin temel özelliklerinden olan ve Bölüm 5.2.1.5'te anlatılan rol geçişidir. Rol geçişi tip bilgisine bağımlıdır; kullanıcı hangi rolün hangi mesajları anlayabildiğini önceden bilmek zorundadır. İkinci yol olan ad ile üye erişimi özelliği ise gerçek dünya varlığına tipten bağımsız erişim olanağı sunar.

5.2.2.6 Baskın Roller

Ad ile üye erişimindeki "en çok özelleşmiş üyeye erişim" kuralı, hiyerarşideki bazı katılımcıların baskın kılınması ile değiştirilebilmelidir. Bir varlığın o anda sahip olduğu rollere göre davranmasının rol tiplerinden bağımsız olarak sağlanırken; bazı rollerin baskın olarak atanabilmesi, varlığın koşullara uygun olarak davranabilmesi için gereklidir. Koşullar aynı mesajı anlayabilecek rollerden en çok özelleşmiş olanın değil de başka bir rolün mesajı yanıtlandırmasını gerektiriyorsa; kullanıcı ya tip adını vererek doğru role geçiş yapar ve onu kullanır, ya da doğru rol baskın kılınarak varlık rol tiplerinden bağımsız olarak kullanılmış olur.

5.2.2.7 Rollerin Askıya Alınması ve Askıdan İndirilmesi

Geçici bir süre ihtiyaç duyulmayacak roller askıya alınabilmeli ve gerek duyulduğunda askıdan indirilerek durum bilgisinde kayıp olmadan tekrar kullanılabilmelidir. Örneğin bir doktor bulaşıcı bir hastalığa yakalandığında doktor rolü askıya alınmalı ve şahıs iyileşince askıdan indirilmelidir.

Rollerin bu ileri düzey özelliği, sadece rolün terki ve yeniden kazanımı ile kolaylıkla gerçeklenemez. Bir yetkinin geçici olarak dondurulması ile tamamen geri alınması farklı anlamlar taşır. Rollerin askıya alınması da aynen böyledir. Terk edilen rol nesnesi kullanılan dilin atık toplayıcısı tarafından durum bilgisi geri alınamayacak şekilde silinebilir. Atık toplayıcısı yoksa bile, bozulan nesne düzeyi kalıtım ilişkilerin rolün tekrar kazanımı ile yeniden kurulması gereksiz yere başarımı düşürecektir. Üstelik Bölüm 5.2.1.2'de anlatılan rollerin hiyerarşik yapısı nedeniyle, bir rol askıya alındığı veya askıdan indirildiği zaman o rolün tüm alt rolleri için de aynı işlem yapılmalıdır.

5.2.2.8 Danışma ve Vekalet Mekanizmalarının Birlikte Kullanımı

Bölüm 8.1.2.7'de ayrıntılı olarak incelenecek bu iki mekanizma, rol geçişi sırasında mesajın ilk alıcısının saklanıp saklanmaması ile birbirinden ayrılır. Gerçek dünya sistemlerinde bu iki mekanizma birlikte görülebileceğinden bir rol modelinde de danışma ve vekalet mekanizmaları birlikte kullanılabilmelidir.

5.3 Neden Rol Modelleri?

Önceki bölümlerde incelenen çalışmaların dinamik sistemlerin etkin modellenmesi için yeterli olmadığı, dezavantajlarının ve eksiklerinin vurgulanması sayesinde açığa çıkmıştır. Dolayısıyla bu alternatif çözümler yerine NYP'nin güçlü özelliklerini daha iyi kullanan tasarım kalıpları tercih edilecektir. Dördüncü bölümde incelenen tasarım kalıplarının ayrıntıları bir yana bırakılırsa, şu çıkarımlarda bulunmak mümkündür:

- Rol modelleri, incelenen kalıpların kullanımı için gerekli olan ek çabayı programcının üzerinden almaktadır. Kalıpların gerçeklenmesi için harcanacak çabaya ek olarak, kalıpları modellenen farklı sistemlere göre iyileştirme çabaları da gerekli olabilir.
- İncelenen kalıpların tümü, rollerin küme şeklinde gösterimini kullanmaktadır. İncelenen kalıpları rol hiyerarşileri şeklinde kullanmak için gereken çaba üstel olarak artacak, bunun sonucunda kod aşırı karmaşık bir hale gelecektir.
- İncelenen kalıplar, rollerin temel ve ileri düzey özellikleri kullanılarak gerçeklenebilecek işlemlerin ancak bir kısmını kullanıcıya sağlayabilecektir.

Sonuç olarak; rollerin temel ve ileri düzey özelliklerinin mümkün olduğunca büyük bölümünü çalışma anında başarım kaybı olmadan destekleyen bir rol modelinin, dinamik sistemlerin modellenmesinde literatürdeki diğer çalışmalardan daha yararlı olacağı görülmektedir. Ancak gerçeklenmek istenen projede rol modellerinin sağlayacağı yeteneklere gereksinim duyulmuyorsa, gereksinimlere en uygun olan tasarım kalıbının kullanılması yanlış olmayacaktır.

5.4 Yazılım Kalite Kriterleri ve Rol Modellerinin Katkıları

Yazılımın kalite özellikleri iç ve dış özellikler olarak iki gruba ayrılabilir. Dış özellikler yazılımı kullananları, iç özellikler ise yazılımı gerçekleyenleri doğrudan ilgilendirir. Dış kalite özellikleri arasında şunlar sayılabilir [35]:

- Doğruluk: Yazılımın analiz, tasarım ve gerçekleme aşamalarında hatalar içermemesi.
- Etkinlik: Bellek ve işlemci gibi sistem kaynaklarının en az oranda kullanımı.
- Güvenilirlik: Sistemin her koşulda istenildiği gibi çalışması, hatalar arasındaki ortalama zaman aralığının (MTBF) yüksek olması.
- Bütünlük: Veri ve işlemlere gerektiği gibi ve istenen güvenlik kuralları çerçevesinde erişilmesi.
- Uyarlanabilirlik: Sistemin değişik uygulamalar veya ortamlarda kullanılabilmesi için mümkün olduğunca az değişiklik gerektirmesi.
- Hassaslık (accuracy): Sistemin kendisinden beklenen işi mümkün olduğunca iyi yapabilmesi.
- Sağlamlık: Aykırı girişlere veya güç çalışma ortamlarına karşılık sistemin çalışmayı sürdürebilmesi.
- Kolaylık: Yazılım kolay kullanılabilir olmalıdır.

Programcıların iç kalite özelliklerine gösterdiği özen dış kalite özelliklerini olumlu yönde etkilemekle birlikte, programcılar gerçekleme sırasında dış kalite özelliklerini

de göz önünde bulundurmalıdırlar. İç kalite özellikleri arasında şunlar sayılabilir [35]:

- Bakım kolaylığı: Yazılıma yeni yetenekler eklemenin, yazılımdaki hataları gidermenin veya yazılımın başarımını attırmanın mümkün olduğunca kolay olması.

- Esneklik: Yazılımın orijinal olarak tasarlandığı uygulamanın dışında çalışabilmesi için gerekli olan değişikliklerin olduğunca az olması.

- Taşınabilirlik: Yazılımın farklı donanım ve işletim sistemleri gibi değişik çalışma ortamlarına kolaylıkla aktarılabilmesi.

- Yeniden kullanılabilirlik: Sistemin parçalarının başka sistemlerde kullanılabilmesinin kolaylığı.

- Okunabilirlik: Kodun kaynak kodunun incelenmesinin kolay olması.

- Sınanabilirlik: Sistemin istenen gereksinimleri karşılayıp karşılamadığının sınanabilmesinin bileşen ve tüm sistem çapında mümkün olduğunca kolay olması.

- Anlaşılabilirlik: Yazılımın sistem, bileşen ve kod düzeylerinde anlaşılabilirliğinin mümkün olduğunca kolay olması. Okunabilirlik sadece kod düzeyinde anlaşılabilirliği sağlar.

Tez çalışmasının hedefi, dinamik sistemlerin etkin bir biçimde modellenebilmesi için Java dilinin rol desteği ile genişletilmesi olarak belirlenmiştir. Saptanan hedefin sağlayacağı yarar ise; daha iyi modellemenin yazılım kalitesini iyileştirmesi, iyileşen kalitenin de yazılım projelerindeki yüksek başarısızlık oranını [36] azaltarak yazılım maliyetini düşürmesi olacaktır. Rollerin temel ve ileri düzey özelliklerinin bu yönde yapacağı katkıların temelleri şunlardır:

- Rollerin temel özelliklerinin kullanımı, tasarım sırasında *ilgi alanlarının ayrılması* ilkesine uyulmasını kolaylaştırır. İlgi alanlarının ayrılması ilkesi; yazılımın sadece belirli bir amaç, kavram veya hedef ile ilgili kısımlarının tanımlanabilmesi, birleştirilebilmesi ve değiştirilebilmesi yeteneğine işaret eder.

Roller ise belirli görevleri yapmak için gerekli yeteneklerin dinamik olarak kazanılmasına olanak tanıyarak, ilgi alanlarının ayrılması ilkesine uyulmasını kolaylaştıracaktır [37].

- Rollerin ileri düzey özelliklerinin, özellikle de ad ile üye erişimi ve baskın rollerin kullanılması ise yazılımın tiplere olan bağımlılığını azaltır. Bölüm 5.2.2.5 ve Bölüm 5.2.2.6'da anlatılan bu iki özellik kullanılarak, nesnelere ait oldukları tipten bağımsız olarak erişilebilir. Bir gerçek dünya varlığını simgeleyen rol hiyerarşisine bu şekilde gönderilen mesaj, o mesajı anlayabilecek rollerden en çok özelleşmiş olanı tarafından işlenir. Modellenen ortamın değişen şartlarına göre aynı mesajın sürekli en çok özelleşmiş rol tarafından değil de, bir başka rol tarafından cevaplanması gerekebilir. Bu durumda da gerçek dünya varlığının ilgili rolleri değişen koşullara göre baskın kılınır.

Bu temel doğrultusunda, rollerin yazılım kalite kriterlerine yapacağı katkılar şunlardır:

- Yeniden kullanılabilirlik: Rollerin ileri düzey özellikleri sayesinde tiplere olan bağımlılığın azalması, yeniden kullanılabilirliği arttırır [38]. Yeniden kullanılabilirlik ise yazılım projelerinin maliyetini ve tamamlanma süresini azaltır [39]. Yeniden kullanılabilirliğin artması, uyarlanabilirliği ve bakım kolaylığını da arttıracaktır.

- Çok biçimlilik: Tiplere olan bağımlılığın azaltılması, çokbiçimlilik derecesini de arttıracaktır. Çokbiçimlilik ise yazılım karmaşıklığının azaltılmasına yardımcı olarak okunabilirliği ve bakım kolaylığını iyileştirecektir [40]. Çokbiçimliliğin uygun kullanımı değinilen yararları sağlarken, çok yüksek veya çok düşük düzeylerde kullanımı ise yarardan çok zarar sağlayacaktır [41].

- İlgi alanlarının ayrılması: Rollerin temel özellikleri kullanılarak ilgi alanlarının ayrılması ilkesine uyulduğu takdirde kod karmaşıklığı azaltılarak anlaşılabilirlik arttırılmış, ayrı amaçlara sahip bileşenlerin yeniden kullanılabilirliği arttırılmış ve bakım kolaylığı yükseltilmiş olacaktır [42].

6. LİTERATÜRDEKİ ROL MODELLERİ

Bu bölümde literatürdeki rol modelleri incelenerek, güçlü ve zayıf yönleri belirtilmiştir.

6.1 Fibonacci

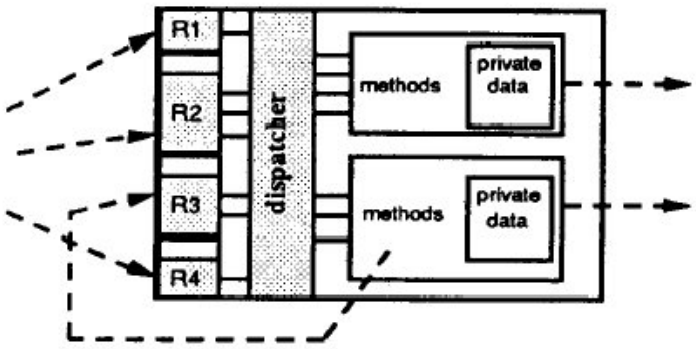
Fibonacci [24], Albano ve arkadaşları tarafından, Galileo dilinden [25] türetilmiş bir NYVT programlama dili gerçeklemesidir. Dilin kendi veri modeli vardır ve miras alma özelliği ile kalıcılığı destekler. http://www.di.unipi.it/~albano/galileo/ adresinden Galileo yorumlayıcısına erişilebilir. Her ifade statik olarak kontrol edilen en azından bir tipe sahip olduğundan Fibonacci kuvvetli tiplemeye sahip bir dildir.

6.1.1 Fibonacci Nesne Mekanizması ve Kullanım Şekilleri

Fibonacci'de nesneler alışılmış NYP dillerinde olduğu gibi bir iç duruma (üye değişkenler ve değerleri) ve bu durumu değiştirmek için kullanılan metotlara sahiptir. Metot çağrısına mesaj adı da verilmiştir. Bir nesneye doğrudan müdahale edilemeyeceğinden tüm işlemler nesnenin rolleri üzerinden yapılmaktadır. Bu kuralla "rollere mesajlar yollanır" diyebiliriz. Bir nesnenin bir mesaja verdiği yanıt tamamen mesajı alan role bağlıdır. Fibonacci'de en çok evrimleşmiş, yani rol hiyerarşisinin daha alt seviyesinde bulunan rol baskındır. Dolayısıyla mesaj iletimi yapraklardan köke doğrudur. Eğer kardeş roller aynı mesaja cevap verebileceklerse bu kez en yakın zamanda yani en son alınan rol mesaja cevap verir. Fibonacci aynı zamanda mesaj iletiminin kökten yapraklara doğru olmasını da destekler.

Fibonacci'de bir gerçek dünya nesnesi kullanıcı için sadece rolleri aracılığı ile erişilen bir kapalı kutudan ibarettir. Rollere ait metotları bu kutunun bağlantı noktaları olarak görebiliriz. Gerçeklemede tüm mesajların doğrudan rol nesnesine gönderilmesi yerine bağlantı noktaları ile rol nesneleri arasında bir mesaj dağıtıcı (*dispatcher*) bulunmaktadır. Mesaj dağıtıcı mesajlar ile roller arasındaki eşlemeden sorumludur. Bu yapı Şekil 6.1'de görülebilir. Fibonacci'de nesnelere doğrudan

müdahale edilmez; nesne ve nesne tiplerine ilişkin tüm işlemler roller ve rol tipleri kullanılarak yapılır.



Şekil 6.1: Bir Fibonacci nesnesinin iç yapısı

Yeni bir tip tanımlanması ve tanımlanan tipin gerçeklenmesi için önce **`Let PersonObject = NewObject;`** şeklinde bir gerçek dünya nesnesi oluşturulur ve **`Let`** ... **`IsA`** ... **`With`** ... **`End`** kurucusu içersinde üye ve metot tanımlamaları yapılır. Ardından da nesne **`let`** ... **`role`** ... **`methods`** ... **`end`** yapısı içersinde gerçeklenir. Bu işlemler Şekil 6.2'de görülebilir. Fibonacci etkileşimli bir dil olduğundan her tanım ve komut sonunda sistem başında >>> işareti koyarak bir cevap verecektir.

```
Let Person = IsA PersonObject With
     Name, Address, Introduce: String;
     BirthYear, Age: Int;
     ModAddress( newAddress: String ): Null;
End;
>>>Let Person <: PersonObject = <Role>
Let john = role Person
     private
     let Name = "John Daniels";
     let BirthYear = 1967;
     let Address = var( "123, Darwin Road, London" )
     methods
     Name = Name;
     BirthYear = BirthYear;
     Age = CurrentYear() - BirthYear;
     Address = at( Address );
     ModAddress( newAddress: String ) =
       Address:= newAddress;
     Introduce = "My name's " & Name & " and I was born in " &
   intToString( BirthYear );
end;
>>>Let john: PersonObject = <role>
```

Şekil 6.2: Şahıs rolünün oluşturulması ve bir nesneye atanması

Fibonacci'de rol tipleri arasında miras alımı desteklenmektedir. Alt tip üst tipin tüm üye değişkenlerini ve metotlarını miras alır. Miras alınan metotlar yeniden ve farklı şekilde tanımlanarak değiştirilebilir (*overriding*). Çoklu miras alma durumunda ad çakışmaları "en son miras alınan rol baskın çıkar" kuralıyla önlenmiştir.

Nesne ve rollerin karşılaştırılması, rol varlığının kontrolü ve gerçek dünya nesnesinin sahip olduğu roller arasında geçiş yapmak (*role casting*) işlemleri her rol desteği veren sistemde zorunlu olarak bulunmalıdır. Fibonacci'de bu özellikleri eksiksiz olarak sağlamaktadır. İki nesnenin rol tiplerine bakmaksızın eşitliği = operatörü ile, bir nesnenin bir role sahip olup olmadığı **`isAlso`** deyimi ile, iki rol nesnenin aynı tipten olup olmadığı **`isExactly`** deyimi ile kontrol edilir. Bir nesneyi mevcut olan rollerinden biri ile kullanma ise **`as`** deyimi ile mümkündür.

```
let johnAsStudent = ext john to Student
    private
      let Faculty = var "Science";
      let StudentNumber = newStudentNumber();
    methods
      Introduce = (me as Person) ! Introduce & ". I am a
        Science student";
end;
>>> let johnAsStudent: Student = <role>
john = johnAsStudent;
>>> true: Bool
```

Şekil 6.3: Nesnenin çalışma anında yeni rol kazanması

Bir nesne, gerekli olduğu üzere, çalışma zamanında da yeni roller kazanabilir. **`ext`** deyimi bu amaçla kullanılır. Şekil 6.3'te bu kullanıma bir örnek görülebilir. **`John`** nesnesine şimdiye dek eklenen bütün rollerin ardından nasıl davranacağı ise Şekil 6.4'te görülebilir.

```
john.Introduce;
>>> "My name is John Daniels and I was born in 1967. I am a
    Science student": String
johnAsStudent.Introduce;
>>> "My name is John Daniels and I was born in 1967. I am a
    Science student": String
(johnAsStudent as Person)!Introduce:
>>> "My name is John Daniels and I was born in 1967": String
```

Şekil 6.4: Eklenen tüm rollerle **`john`** nesnesinin davranışı

6.1.2 Değerlendirme

Kullanılan nesne yapısı, dağıtılmış uygulamalar için uygundur. Fibonacci'nin en önemli eksiği, rollerin temel özelliklerinden olan nitelikli rolleri desteklememesidir. Bir başka deyişle, Fibonacci aynı tip rolden birden fazlasına sahip olunması özelliğini desteklemez.

6.2 Smalltalk ile Rol Hiyerarşisi Yaklaşımı

Gottlob, Schrefl ve Röck tarafından yapılan çalışma [2], mevcut NYP dillerine birkaç ek sınıfın yardımıyla nasıl rol desteği kazandırılabileceğini göstermesi nedeniyle büyük bir öneme sahiptir. Ortaya çıkarılan sisteme bir ad verilmemiştir, ancak kullandıkları yaklaşım nesne bazında miras alan rol nesnelerinin hiyerarşisinden ibaret olduğu için rol hiyerarşisi yaklaşımı olarak adlandırılabilir. Önerilen sistem sınıf bazında miras almayı dışlamamakta, aksine en uygun olduğu yerlerde kullanılmasını tavsiye etmekte ve örneklemektedir.

Yazarlar iddialarının doğruluğunu göstermek amacı ile Smalltalk için yazdıkları birkaç sınıf sayesinde sıradan nesneleri rolleri destekler hale getirmişlerdir. Tez çalışmasının amacı Java diline rol desteği kazandırmak olduğundan orijinal çalışmada verilen Smalltalk kaynak kodları mümkün olduğunca Java'ya çevrilmiştir. Mesaj yayılımı ve global sınıf değişkenleri gibi dil özelliklerinin Java'da bulunmaması birebir çevrimi imkansız kılar. Ayrıca Java kuvvetli tiplemeye sahip bir dil iken Smalltalk'ta geç bağlama (*late binding*) mevcuttur, yani bir değişkenin tipi atama yapılıncaya kadar belli değildir.

6.2.1 SmallTalk'ta Rol Tanımı

Rol hiyerarşisi Tablo 6.1'de işlevleri verilen **`ObjectWithRoles`** ve **`RoleType`** olmak üzere sadece iki sınıf ile gerçeklenebilir. Rol hiyerarşisinin kökü olacak ve çeşitli rollere sahip olabilecek her nesne, **`ObjectWithRoles`** sınıfının bir alt sınıfına ait olmalıdır. Yeni rol tipleri tanımlamak ve bunlardan rol nesneleri oluşturmak içinse Şekil 6.5'te özetlenen **`RoleType`** sınıfı kullanılır. Yeni bir rol tipi tanımlamak için **`RoleType`** sınıfının **`defRoleType`** metodu çalıştırılır. **`defRoleType`** metodunun parametreleri arasında bu rolün üst rolünü simgeleyen sınıf ve rolün

gerektirdiği üye değişkenler vardır. C dilinde fonksiyona işaretçi tanımlamak mümkündür. Java'da ise bir sınıftan herhangi değişken üretilmeksizin sınıfın metodu kullanılabilir. Örneğin **`java.lang.Integer`** sınıfı **`Integer.parseInt("245")`** şeklinde kod içersinde herhangi bir yerde kullanılabilir.

Tablo 6.1: Rol sınıfının işlevleri

ObjectWithRoles ve RoleType sınıf metotları:	**İşlev:**
root	Rol hiyerarşisinin kökünü döndür
roleOf	Rol hiyerarşisindeki ebeveyni döndür
as("aRoleType")	aRoleType rolüne geç
existsAs("aRoleType")	aRoleType rolünün olup olmadığının kontrolü
entityEquiv(anObject)	anObject ile aynı gerçek dünya nesnesinin temsil edilip edilmediğinin kontrolü
RoleType sınıfı metodu	**İşlev:**
abandon	Bu rolü terket

```
class RoleType
{
     Object roleSuperType;
     defRoleType(StringroleTypeName,String[]
       stringOfInstVarNames, String[] stringOfClassVarNames,
       String[] stringOfPoolNames,
       String categoryNameString, Object nameOfRoleSuperType )
     {
       ... [yeni bir rol tipi oluşturma için kod]
     }
     newRoleOf( Object *anObject )
     {
       ... [bu rolü başka bir nesneye ekleme için kod]
       roleSuperType = anObject;
     }
}
```

Şekil 6.5: Rol sınıfları ve nesnelerinin tanımlanması

Bir rolü terk etmek için ise o rolü temsil eden **`RoleType`** sınıfından ilgili nesnenin **`abandon`** metodu çağrılır. Bu metot ilgili nesneyi yok eder. Bu rol hiyerarşi ağacındaki ara düğümlerden biri ise tüm çocukları da silinmiş olur. Bu özellik

SmallTalk'ın yapısından kaynaklanır. Yeni rol tipleri tanımlama ve terk etme Şekil 6.6'da gösterildiği gibi kodlanır.

```
//Hiyerarşinin kökünü oluştur:
personSmith = new Person( );
personSmith.setName( "Smith, Anne" );
personSmith.setBirthDate( 30, "jan", 1966 );
personSmith.setPhoneNo( "312-4534" );

//çalışan rolünü tanımla:
empSmith = new Employee( );
empSmith.newRoleOf( personSmith );
empSmith.setSalary( 1500 );
empSmith.setPhoneNo( "304-5601" );

//öğrenci rolünü tanımla:
studentSmith = new Student( );
studentSmith.newRoleOf( personSmith );

//...
//Buraya Şekil 6.7 ve 6.8'deki komutlar gelebilir

//Smith mezun olunca:
studentSmith.abandon( );
```

Şekil 6.6: Rol nesneleri kullanımına örnekler.

Bir nesnenin istenilen bir role sahip olup olmadığını anlamak için o nesnenin **`existsAs`** metodu çağrılır. İki rol nesnesinin aynı gerçek dünya varlığını temsil edip etmediğini anlamak için ise **`anObject1.entityEquiv (anObject2)`** metodu kullanılır. Şekil 6.7'de bu işlemler örneklenmiştir: Şekil 6.7'deki kod Şekil 6.6'da üç nokta ile işaretlenen yere konulduğu takdirde ne gibi sonuçlar alınacağı da Şekil 6.7'de satır sonlarında gösterilmiştir.

```
anEmployee = empSmith;
aStudent = studentSmith;
if( aStudent == studentSmith ) { /*TRUE*/ }
if( aStudent == anEmployee ) { /*FALSE*/ }
if( aStudent.entityEquiv( studentSmith ) ) { /*TRUE*/ }
if( aStudent.entityEquiv( anEmployee ) ) { /*TRUE*/ }
```

Şekil 6.7: Rol varlığı ve varlık eşitliği kontrolü.

Roller arasında geçiş yapmak, yani bir nesneyi istenilen bir rolünde kullanmak için ilgili rol nesnesinin (**`RoleType`** veya **`ObjectWithRoles`** sınıfından) metotlarına ulaşmak yeterlidir. Bir rol nesnesinin üzerinden başka bir role ulaşmak da mümkündür ve üç şekilde mümkündür:

1. Rol nesnesinin (**RoleType** sınıfından) **roleOf** metodu hiyerarşideki bir üst role erişim sağlar.

2. Rol nesnesinin (**RoleType** sınıfından) **root** metodu hiyerarşinin **ObjectWithRoles** sınıfından olan köküne erişim sağlar.

3. Bir **anObject** nesnesininin **aRoleType** rolüne erişim için **anObject as: aRoleType** metodu kullanılır. Java örneğiyle **anObject->as ("RoleType")** yazılır veya Java'nın yansıtma yetenekleri kullanılır. SmallTalk'ta fonksiyona parametre olarak bir sınıf adı verilebiliyorken Java'da ancak bir etiket bilgisini simgeleyen bir katar verebiliriz. Gerçekleme aşamasında her rol nesnesine bir etiket eklenirse, **as** metodu bir etiket kabul etmiş olur ve nesnenin rolleri arasında bu etikete göre tarama yapılarak uygun olanı kullandırılır.

Şekil 6.8'de bu işlemler örneklenmiştir: Şekil 6.8'deki kod Şekil 6.6'da üç nokta ile işaretlenen yere konulduğu takdirde ne gibi sonuçlar alınacağı gösterilmiştir.

Mevcut olmayan bir role geçiş yapmaya çalışmak aykırı durum oluşturacaktır. Bunu önlemek için önce geçiş yapılmak istenen rolün var olup olmadığı yukarıda anlatıldığı şekilde kontrol edilmek istenebilir.

```
System.out.println( empSmith.getPhoneNo() );
  //304-5601 yazdırır
System.out.println(empSmith.roleOf().getPhoneNo());
  //312-4534 yazdırır
personSmith = new Person( studentSmith.root() );
System.out.println(depMgSmith.root().getPhoneNo());
  //312-4534 yazdırır
if( personSmith.existsAs("Employee") ) /*TRUE*/
  System.out.println(personSmith.as("Employee").getPhoneNo());
  //304-5601 yazdırır
System.out.println( empSmith.getName( ) );
  //Smith, Anne yazdırır.
```

Şekil 6.8: Roller arasında geçişler.

Rollerin temel özellikleri arasında Bölüm 5.2.1.10'da anlatılan nitelikli rollerin modellenmesi için ise, **RoleType** sınıfından türetilen **QualifiedRoleType** sınıfı kullanılır. Aynı rol varlığının birden fazla örneğinin birbirinden ayırt edilmesini sağlayan niteleyiciler herhangi bir tipten nesneler olabilir. Nitelikli rollerin normal

rollerle eşbiçimli olarak ele alınabilmesi için **`ObjectWithRoles`** ve **`RoleType`** sınıflarında rollerin varlığının kontrolü ve nitelikli rolleri de destekleyecek şekilde rol geçişleri için, Tablo 6.2'de gösterilen eklemeler yapılmıştır.

Tablo 6.2: Nitelikli rollerin desteklenmesi için ek metotlar

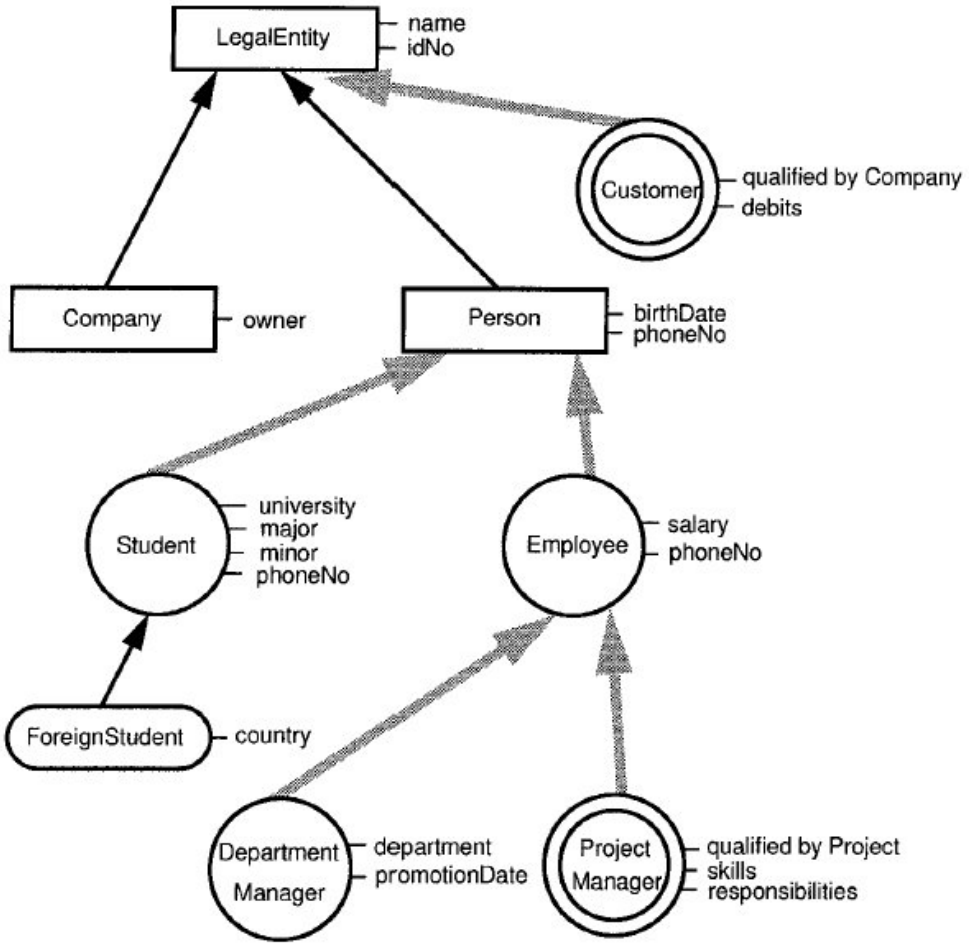
ObjectWithRoles ve RoleType için	**Amaç**
as(aQualifiedRoleType, qualifyingObj)	QualifyingObj tarafından nitelenen aQualifiedRoleType rolüne geçiş
existsAs(aQualifiedRoleType, qualifyingObj)	Metodu çağrılan nesne QualifyingObj tarafından nitelenen aQualifiedRoleType rolüne sahip mi?
QualifiedRoleType için	**Amaç**
qualifier( )	Niteleyen nesneyi döndür

6.2.2 Sınıf ve Rol Hiyerarşilerinin Birleştirilmesi

Şekil 6.9'da [2] bir sınıf ve iki rol hiyerarşisinin birleşmesi gösterilmiştir. Şekilde koyu oklar sınıf bazında, açık oklar ise nesne bazında miras alma ilişkisini göstermektedir. Dikdörtgenler nesneleri, daireler rolleri, çift daireler ise nitelikli rolleri gösterir. Sınıf hiyerarşisini oluşturanlar kökü **`LegalEntity`** sınıfı olmak üzere **`LegalEntity`**, **`Company`** ve **`Person`** sınıflarıdır. **`Company`** sınıfı sıradan bir sınıf olabilir, ancak diğerleri **`ObjectWithRoles`** sınıfından türetilmiş olmalıdır. 1. rol hiyerarşisi kökünde **`LegalEntity`** ve tek düğümünde **`Customer`** nitelikli rolünden ibarettir. İkinci rol hiyerarşisinin kökü ise **`Person`** nesnesidir. Burada dikkat edilmesi gereken nokta şudur: **`Person`** sınıfı, **`LegalEntity`** sınıfında tanımlanan **`Customer`** nitelikli rolünü miras alır. Dolayısıyla **`Person`** sınıfından her nesne hem **`Customer`** rolüne, hem de 2. rol hiyerarşisinde tanımlanmış rollerden herhangi birine sahip olabilir.

Sınıflar ve roller dikey olarak birleştirilebilir. Bir sınıfın alt sınıfları bir rol hiyerarşisinin kökü olabilir. Ayrıca rollerin de alt rolleri ve alt sınıfları olabilir. Bu ikisi arasındaki fark, alt rol ile üst rol arasında **`roleOf`** ilişkisinin (nesne seviyesinde miras), üst rol ve bunun alt sınıfı arasında ise (sınıf seviyesinde) kalıtım ilişkisinin mevcut olmasıdır. Örnekte bir insanın öğrenci rolü ya normal öğrenci (**`Student`**) ya da yabancı öğrenci (**`ForeignStudent`**) rolü sınıfına girer. Eğer öğrenci rolü

nitelikli rol olsaydı birden fazla okulda normal ve/veya yabancı öğrenci olabilmek mümkün olacaktı. Roller arasında sınıf seviyesi kalıtım ilişkisi, rollerin özelleştirilebilmesi için gereklidir. Böylece ortak özellikleri olmakla birlikte farklı işlevselliğe sahip roller modellenebilir. Örneğin hem bir mühendislik öğrencisi hem de tıp öğrencisi stajyer olabilir ancak bu ikisinin yapması gereken görevler birbirinden çok farklıdır. Şekil 6.9'da **`ForeignStudent`** rolü **`Student`** rolünden sınıf bazında miras alma ile türetildiği için kenarları yuvarlak dikdörtgen ile gösterilmiştir.



Şekil 6.9: Rol ve sınıf hiyerarşilerinin birlikteliği ile bir gerçek dünya sisteminin modellenmesi [2]

6.2.3 Değerlendirme

İncelenen çalışma mevcut NYP dillerine rol desteğinin eklenmesinin mümkün olduğunu ispat ederek bunun için rol hiyerarşisi yaklaşımını önermektedir. Bu rol modeli Smalltalk dilinin sunduğu **`doesNotUnderstand:`** mekanizmasına

dayanmaktadır. Anlamadığı bir mesaj iletilen her nesne; anlaşılmayan mesajın yerine, eğer tanımlanmışsa **doesNotUnderstand:** adlı metodunu çalıştırır. **RoleType** sınıfında bu metot değiştirilerek roller arasında mesaj iletimi sağlanmıştır; alt rolün anlayamadığı her mesaj üst role aktarılır. Bir tek kısıtlama ile birlikte bu rol modeli, rollerin temel özelliklerin tümünü destekler. Söz konusu kısıtlama nitelikli rollerin alt rollerinin de nitelikli roller olması zorunluluğudur.

Gottlob'un [2] çalışmasının temel önerileri olan rollere hiyerarşik yaklaşımın kullanılması ve rollerin temel özelliklerinin desteklenmesi için sadece üç sınıfın kullanması önerileri, tez çalışmasında kullanılarak gerçeklenen rol modeli JAWIRO'ya katkıda bulunmuştur.

6.3 INADA

INADA [26], M. Aritsugi tarafından doktora tezi olarak temelleri atılmış ve Japon Kyushu üniversitesinde geliştirilen, C++ üzerine kurulmuş bir dildir. Üzerindeki çalışmalar halen devam eden INADA, WAKASHI [27] adlı bir dağıtılmış nesne depolama sunucusu üzerine kurulduğundan kalıcı nesneleri destekler. INADA, ODMG (Object Database Management Group) standartlarına uyumludur. Her INADA tipi bir role karşılık gelmektedir.

6.3.1 INADA'nın (ODMG ve C++'a) Eklentileri ve Tip Erişimi

T adlı kalıcı sınıftan bir nesne ODMG ile benzer olarak, **d_Ref<T>** ***nesne_adı*** şeklinde tanımlanır. Kalıcı nesneler için **new** operatörü kurucu fonksiyonundan önce gelen iki ek parametre alır. Bunlardan birincisi bir ODMG veritabanına referans, ikincisi sınıfın adıdır. Şekil 6.10'daki örnek bir şahısı modellemek üzere **Person** sınıfına ait bir p nesnesinin oluşturulmasını göstermektedir. Kalıcı nesneler için C++ standart tip özellikleri ve hataları kullanılır.

```
d_Database dbobj;
d_Database *database = &dbobj;
main() {
  database -> open("dbfile");
  d_Ref<Person> p = new(database,"Person")Person("ari",30,…);
}
```

Şekil 6.10: Kalıcı bir gerçek dünya nesnesinin oluşturulması

Birden fazla tipe, yani role sahip nesneleri desteklemek için **transforms** ve **as** olmak üzere iki yeni dil öğesi bulunur. Kalıcı nesnelere yeni bir tip eklerken, kalıcı nesneye sahip olduğu bir tip üzerinden erişirken ve bir tipi kalıcı nesneden silerken bu komutlar kullanılır.

Kalıcı nesneye veri tabanında bulunan bir tip, yani bir rol ekleneceği zaman **transforms** komutu **new** operatörü ile birlikte kullanılır. Şekil 6.11'de örnek kullanım görülebilir. Bu örnekteki e işaretçisi önceki örnekte tanımlanan p işaretçisi ile dil açısından karşılaştırıldığında, aynı kalıcı nesneyi gösteren eşit işaretçilerdir. Ancak aynı varlığın değişik rollerini simgeledikleri için aynı sonuçları vermemeleri doğaldır. Ayrıca INADA'nın mevcut halini aynı nesnenin aynı tipten birden fazla role sahip olmasına, başka bir deyişle nitelikli rollerin kullanımına izin veremez.

```
d_Ref<Employee> e = new(database, "Employee")
Employee ("sale", 1853)
transforms p;
```

Şekil 6.11: Bir nesneye rol kazandırılması

Kalıcı nesnelere sahip olduğu tiplerden herhangi biri ile erişilebilir, ancak bunlar birbirlerinden farklı sonuç verir. Şekil 6.11'deki örnekten gidersek **p->TelNo()** metodu ev telefonunu verirken **e->TelNo()** metodu iş telefonunu döndürür. **Person** sınıfının **p()** metodu olsun ancak **Employee** sınıfının olmasın. Bu durumda **e->p()** erişimi hata verir, ancak **e as Person->p()** erişimi hata vermez. Bu kural tersine de geçerlidir: **Employee** sınıfının **e()** metodu var ama **Person** sınıfının yoksa **p->e()** hatalıdır, ancak **e as Person->p()** doğrudur. İşte bu tür erişimi mümkün kılmak için yeni bir tip sistemi eklemek gereklidir.

as'in diğer kullanımı **this as *tip_adı*** şeklindedir. Bu deyim tipi verilen tip adından olan bir nesnenin sanal adresini döndürür, tabi eğer **this** ile anlatılmak istenen nesne o tipe (role) sahipse. Şekil 6.12'de bu tip kullanıma bir örnek görülebilir.

Nesneler ve sahip olabileceği (ya da olmayacağı) tiplerin tanımlanması ile, o tipten olan nesnelerle yapılan nesne oluşturma, o nesneye tip eklenmesi veya o nesneden tip çıkarılması işlemlerinin sırası birbirinden tamamen bağımsızdır. Dolayısıyla burada dikkat edilmesi gereken nokta, Şekil 6.12'de (3) ile işaretlenen deyimin (1) ve (2)

deyimlerinden önce çalışması gerektiğidir. Aksi halde çalışma anı hatası olur. **`delete tip_adı of nesne_adı`** komutu ile ise istenilen nesneden istenilen rol geri alınabilir.

```
class P: public d_Object
{
  int a;
  char b[12];
  public:
    P(int, char*); // constructor
    int A(){ return a; }
    char* B(){ return b; }
};
class Q: public d_Object
{
  public:
    Q() {} // constructor
    int A() { return this as P -> A(); } //(1)
    char* B() { return this as P -> B(); } //(2)
};
main()
{
  database -> open("dbfile");
  d_Ref<P> p = new(database, "P")P(4, "ari");
  d_Ref<Q> q = new(database, "Q")Q() transforms p; //(3)
}
```

Şekil 6.12: Örnek sınıf tanımlamaları

6.3.2 Tip Tanımları

INADA'da geleneksel NYP miras mekanizmasının aksine istenmeyen metotlar atılabilir. Bu yaklaşım nesne arayüzleri [6] çalışmasınınki ile aynıdır. Temel ilke "sadece tanıtılan metotlar kalıtım ile alınır" şeklinde özetlenebilir. Bu sayede tipler arasında kalıtım işlemleri aşağıda verilen amaçlarla gerçeklenebilir. Bu amaçlardan biri, Şekil 6.12'de örneklenen klonlama işlemidir. Diğer örnekler için orijinal çalışma incelenebilir.

Şekil 6.12'de tanıtılan Q sınıfı bir gerçek dünya problemini çözmeye yaramayacaktır. Bir tür isim değiştirme, ya da P'ye yeni bir Q arayüzü ekleyip onun üzerinden P'nin üyelerine ulaşma işlevi gördüğü söylenebilir. Sistemin tüm özellikleri düşünülürse bu işlem veri tabanı görüntüsü (*database view*) oluşturmakta kullanılabilir. Görüntüye dahil olacak elemanlar klonlanarak bir nesne kümesi oluşturulur ve bu kümede yapılan işlemler gerçek veri tabanına da aktarılmış olur.

INADA'da tip tanımları ile ilgili diğer ayrıntılar şunlardır:

- Bir özelliğe (üye veya metot) diğer bir (alt) tipin ilgili özelliği üzerinden erişilebilir.
- Bir (alt) tip yeni üye ve/veya metotlara sahip olabilir.
- Tiplerin eklenmesinin mevcut nesneler üzerinde herhangi bir etkisi yoktur. Yani nesneye yeni bir tip eklenmesi eski tip üzerinden erişimi değiştirmez.
- INADA'da klasik NYP anlamındaki miras alma ilişkisi de geçerliliğini korur.

INADA'nın üzerine kurulduğu WAKASHI bir dönüşümsüz (non-swizzling) [28] sistemdir, yani nesnelerin hafızadaki yerlerine olan işaretçiler ile kalıcı ortamdaki adresleri arasındaki eşleme önceden hesaplanmaz. Dönüşümlü yaklaşımlar genelde daha iyi başarım gösterse de yazarlar daha önceki deneyimlerine dayanarak arada fazla başarım farkı olmadığını ve çok tipli nesnelerle çalışıldığında dönüşümsüz yaklaşımın daha fazla esneklik sağladığını öne sürmektedir.

6.3.3 Değerlendirme

INADA'nın önemli eksiklikleri mevcut gerçekleme tekniğinin bir nesnenin aynı tipten birden fazla role sahip olmasına izin vermemesi ve çalışma anında bir kalıcı nesnenin istenen bir tipe sahip olup olmadığını kontrol edecek bir komutun gerçeklenmemiş olmasıdır. Ayrıca roller ağaç yerine küme yapısı ile gerçeklenmiştir; bir rol alt rollere sahip olamaz.

6.4 Chameleon

Rol kavramını ilişkisel olarak ele alan Chameleon [29] adlı çalışma Java diline rol desteği kazandırmayı amaçlamaktadır. Rol desteği seçici (constituent) metotlar ile sağlanmaktadır. Seçici metotların özellikleri şunlardır:

- Doğrudan çağırılamazlar,
- Birebir olarak ilişkilendirildikleri normal metodun öncesinde veya sonrasında çalıştırılabileceği gibi normal metodun yerini de alabilirler.

- Giriş argümanlarını ve dönüş değerini değiştirebilirler.

İncelenen diğer yaklaşımların aksine, Chameleon nesne ve rolleri arasındaki ilişkiyi referanslar ile değil, seçici metotlar ile gerçeklemektedir. Ancak herhangi bir rol yine programcı tarafından o rol nesnesine olan bir referans ile kullanılır. Bir metota birden fazla seçici metot ilişkilendirildiğinde bunlar önceden belirlenmiş önceliklere göre çalışacaktır. Seçici metotların öncelikleri çalışma anında da değiştirilebilir.

Seçici metotlar gibi bir yapı, kullanılan dilin modifikasyonunu gerektirir. Bu amaçla Chameleon standart Java yerine OpenJava [30] kullanmaktadır. OpenJava ile yazılan Chameleon kodu standart Java koduna dönüştürülmektedir. OpenJava şu an için ancak Java 1.1 uyumlu kod üretmektedir ve OpenJava'nın en son sürümü olan 1.1 versiyonu Ağustos 2002 tarihinde derlenmiştir. Bu tarihten beri herhangi bir ilerleme olmaması ve Sun Microsystems'in Java sitesinde OpenJava ile ilgili herhangi bir bilgi veya bağlantı olmaması dikkat çekicidir.

İncelenen diğer yaklaşımlarda olduğu gibi, Chameleon'da da sınıf bazında kalıtım ile nesne bazında kalıtım birlikte kullanılabilir. Burada nesne bazında kalıtım baskındır. Bir alt sınıfın üst sınıftan kalıtım ile aldığı metotla sahip olduğu rollerden birinin metodu aynı imzaya sahipse, role ait metot kullanılır. Bu durum çeşitli metotların birbirlerini çağıracak şekilde yazılması halinde gerçekleşebilir, kullanıcının nesnenin kendisi veya rolü ile ilgili olan ve aynı imzalı metotlardan dilediğini çalıştırabilir (doğrudan seçici metotları çalıştırmak hariç).

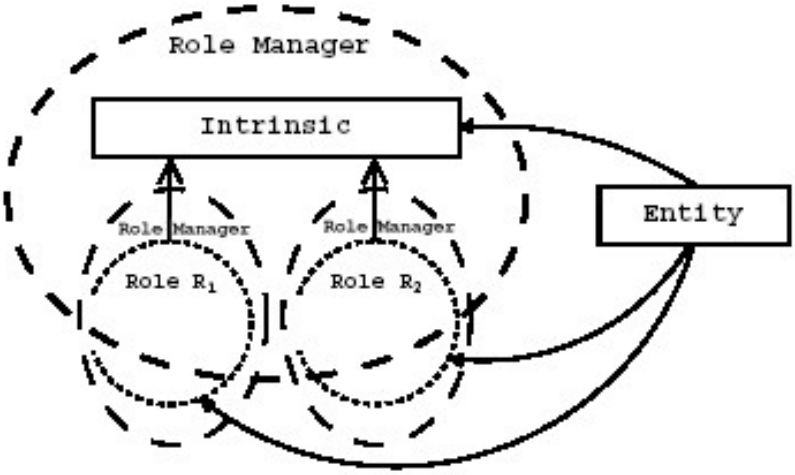
6.4.1 Gerçekleme Detayları ve Örnek Uygulamalar

En az bir role sahip her nesnenin "rol yöneticisi" adı verilen bir üst-nesnesi vardır. Rol yöneticisi seçici şu işlevleri destekler:

- Metotların gerekli sırada çalıştırılması,
- Nesnenin sahip olduğu rollerin sorgulanabilmesi,
- İki rolün aynı nesnede barınıp barınamayacağı kısıtlamalarının oluşturulup yürütülmesi.

Rol yöneticisi ait olduğu nesneye erişimi kontrol eden ve onu saran bir zar olarak düşünülebilir. Rol alabilecek bir nesne ilk rolünü aldığı anda bir de rol yöneticisine

sahip olur. Chameleon bu yönüyle Fibonacci[24] ve bileşik nesne sistemlerini[17] andırır, ancak bunlardan farklıdır: Bütün bir rol hiyerarşisini kontrol eden merkezi bir yönetici yoktur, her rol sahibi nesne veya alt rolleri olan rol için ayrı bir yönetici vardır. Bu yapı Şekil 6.13'te görülebilir. Rol yöneticisi bu görevleri yerine getirebilmek için kontrol akışına müdahale eder. Bu müdahale, Chameleon kodunun OpenJava'dan düz Java'ya çevrilmesi sırasında yapılan işlemler sayesinde mümkün olabilmektedir.



Şekil 6.13: Bir rol hiyerarşisi ve rol yöneticileri.

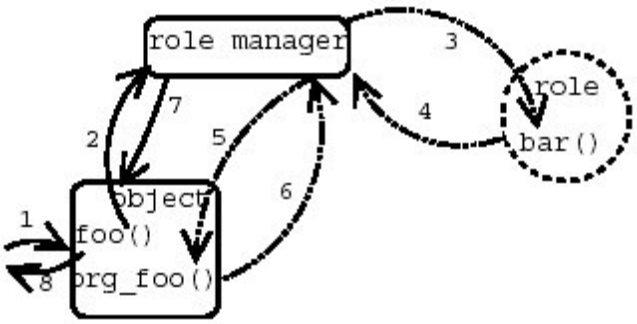
Kod dönüştürmesi sırasında bir X metodu üstünde yapılan işlemler şunlardır:

1. X'in aldığı parametrelere ek olarak bir bayrak değişkeni ve **`self`** adlı bir işaretçi olmak üzere iki yeni parametre ekleyerek yeni bir metot oluşturulur. Bu metot kontrol akışını rol yöneticisine devreder. X'in döndürdüğü bir parametre varsa rol yöneticisi de bu parametreyi döndürecek şekilde hazırlanır. Bayrak değişkeni metot çağrısının nesnenin içinden mi yoksa dışından mı geldiğini belirler. Çağrı içsel ise yayılım geçici olarak devre dışı bırakılır. **`self`** işaretçisi mesajın orijinal alıcısını gösterirken Java'nın **`this`** işaretçisi o anda çalışan nesneyi gösterir.
2. X'in adı **`org_X`** olarak değiştirilir, böylece bu metoda rol yöneticisi dışında erişilemez.
3. **`org_X`** ile aynı imzaya sahip bir başka X metodu oluşturulur. Tüm metot çağrılarının alıcısı olan bu metodun tek işlevi ilk adımda oluşturulan metodu çağırmak ve **`self`** işaretçisini sağlamaktır.

4. Nesnenin kendi arayüzünde ve kendisinden miras alan (sınıf veya nesne bazında) nesnelerin arayüzlerinde bulunmayan, yani kendi iç çalışması ile ilgili tüm metotlara olan çağrılar, parametrelerine (1. adımla uygun olacak şekilde) **`self`** işarteçisi ve bayrak değişkeni eklenerek genişletilir. Bu tip metotlara kapalı metotlar adı verilir.
5. Açık metotlara olan çağrılar genişletilmez, çünkü bu bir Chameleon çağrısı ise o nesnenin 3. adımda üretilen ilgili metodu tarafından yakalanacaktır. Aksi takdirde bu sıradan bir metot çağrısıdır ve zaten genişletilmesi gerekmemektedir.

Bu dönüşümler sonucunda:

- Seçici metotların çalışmasını sağlamak üzere kontrol akışı rol yöneticisine geçer,
- İletimler sırasında **`self`** referansının hatırlanması sağlanır,
- Chameleon derleyicisi ile derlenmemiş sıradan Java nesneleri ile uyumluluk sağlanır.



Şekil 6.14: Bir Chameleon nesnesinde metot çağırımı.

Dönüşüm yapısı karmaşıktır ve kontrol akışı bir çok kez fazladan el değiştirir. Şekil 6.14'te bir Chameleon nesnesinin bir metodunun ve onunla ilişkili olan rolün işletilmesi gösterilmiştir. Rol yöneticisi bir nesnenin **`foo`** adlı metodu çağrıldığında (adım 1) kontrolü ele alır (adım 2), eğer ilişkili bir rol varsa çalıştırıp (adım 3) sonucunu alır (adım 4), gerekirse metodun orijinal hali olan **`org_foo`** metodunu da çalıştırır (adım 5) ve sonucunu alarak (adım 6) nesnede gerekli işlem ve değişiklikler yapılır (adım 7). Bu örnekte seçici metodun ilgili metottan önce çalışması gerekliydi.

Seçici metodun sonradan çalışması halinde 3. adım ile 5. adım ve 4. adım ile 6. adımın sıraları değişecekti. Zaten karmaşık bir yapı ortaya çıkan Chameleon sisteminde hem metottan önce hem de metottan sonra bir seçici metot çalışması gerekseydi araya iki adım daha eklenecekti.

Gerçeklemedeki karmaşıklığa rağmen, Chameleon ile rollerin tanımlanması ve kullanılması, Şekil 6.15 ve Şekil 6.16'da gösterildiği gibi anlaşılabilir bir biçimdedir.

```
class Person { //sınıf tanımı
  int height;
  String name;
  void say(String s){System.out.print(name+s);}
}
role Teacher roleifies Person { //rol tanımı
  String getname(){return "Teacher " + name;}
  void teach(){...}
}
role Supervisor roleifies Teacher { ... }
```

Şekil 6.15: Chameleon'da rol tanımlamasına örnekler

```
Person p1 = new Person("Windy");
Person p2 = new Person("Bert");
Teacher t = new Teacher(p1);       //rol atanması
t.say("says hello");               //rol kullanımı
void accounting(Teacher t) {
  Supervisor s = t as Supervisor; //rol değiştirme
  ...
}
```

Şekil 6.16: Chameleon'da rol kullanımına örnekler

6.4.2 Değerlendirme

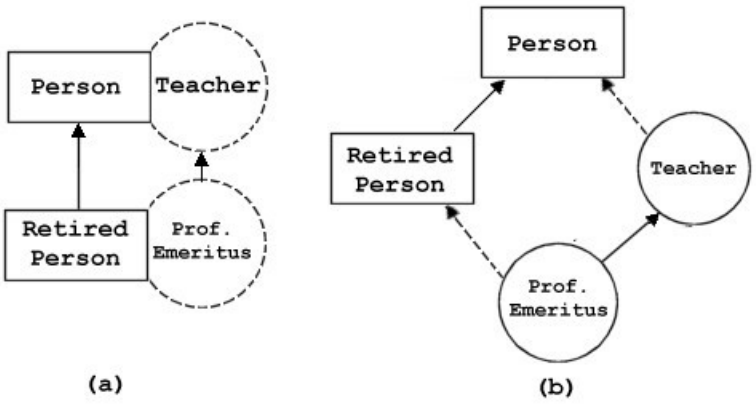
Chameleon rol modeli ile bakış tabanlı programlama yaklaşımı arasında önemli benzerlikler vardır. Chameleon'un temel taşı olan seçici metotlar ve bunların kullanımının BTP yaklaşımındaki tavsiye ve kesme noktası kavramları ile benzeşmesi nedeniyle bu benzetme rahatlıkla yapılabilir. Bakış tabanlı programlama ile rol desteğinin kazandırılabileceği fikri yeni değildir, Kendall'ın çalışması [10] örnek olarak verilebilir.

Chameleon'un rol modelinin ilk zayıf yönü, bir rolün ait olduğu nesnenin tüm alanlarını gereksiz yere miras almasıdır. Roller aslında rolün sahibinin sınıfından sınıf düzeyinde kalıtımla gerçeklenmiştir. Bir rolün sahibinde tanımlı olup da rolde

tanımlı olmayan tüm metotlar için rol sınıfında tek görevi çağrıyı rolün ait olduğu nesneye iletmeye yarayan metotlar dönüşüm sırasında tanımlanır.

Nesne düzeyinde kalıtımın kullanılmaması, rollerin farklı sınıflardan olmalarına rağmen kendisi ile uyumlu nesneler arasında çalışma anında aktarılmasını imkansız kılacak ya da en azından zorlaştıracaktır. Hangi durumun söz konusu olduğu belirtilmemiştir, ancak bu konuda hiçbir örnek verilmediği için ve aşağıda anlatılan kısıtlama nedeniyle rol sınıflarının statik olarak belli sınıf veya roller ile ilişkilendirdiği kabul edilmeli ve bu ikinci zayıf yön olarak ele alınmalıdır.

Chameleon'un üçüncü zayıf yönü de ilk iki zayıflığından kaynaklanmaktadır: Java dilinde çoklu kalıtım desteklenmediği için bir rol aynı anda hem başka bir rolden türetilip (sınıf düzeyinde kalıtım ile) hem de türetildiği rol sınıfı ile ilişkilendirilmiş sınıfın dışındaki sınıflardan nesnelerin rolü olamaz. Bu kısıtlama Şekil 6.17a'da gösterilen durumun modellenmesini imkansız hale getirmektedir. Halbuki bu durum ilişkisel rol hiyerarşileri yaklaşımı ile Şekil 6.17b'de görüldüğü gibi modellenebilir.



Şekil 6.17: (a) Chameleon ile modellenemeyen bir durum. **(b)** Aynı durumun ilişkisel rol hiyerarşileri yaklaşımı ile modellenmesi mümkündür.

Chameleon rol tabanlı programlamaya değişik bir bakış açısı getirse de oluşturulan yapının karmaşıklığı ve zayıf noktalar sistemin verimli kullanımını önemli ölçüde azaltmaktadır. Sistemin standart Java yerine sadece Java 1.1 desteğine sahip OpenJava üzerine kurulması ise önemli sorun oluşturur. 1.2 ve 1.5 sürümleri, Java dili için önemli dönüm noktaları olup bir çok yenilikler getirmiştir. OpenJava ile bu yeniliklerin hiçbiri kullanılamayacaktır. Üstelik, Chameleon Java'nın yansıtma yeteneklerini yoğun olarak kullanır ki bu yeteneklerin başarımı Java'nın 1.4

sürümünde önemli ölçüde iyileştirilmiştir [31]. OpenJava gereksinimi nedeniyle Chameleon'un başarımı kısıtlı kalacaktır.

Chameleon'un vekalet mekanizmasının gerçeklenmesi için önerdiği yol, rol geçişi sırasında çağrılan metodun ilk alıcısının self adlı bir üye nesnede saklanması mekanizmasıdır. Bu öneri tez çalışmasında kullanılarak gerçeklenen rol modeli JAWIRO'ya katkıda bulunmuştur.

6.5 DEC-JAVA

Bettini ve arkadaşları tarafından önerilen bir Java dili uzantısı olan Dec-Java [32], nesnelerin uygulama süresince değişmeyecek temel özelliklerinin, nesnelerin davranışlarından ayırt edilebilmesini sağlar. Böylece nesnelerin çalışma anında değişebilecek davranışları dinamik olarak düzenlenebilir. Bu çalışmanın temelinde de sınıf düzeyinde kalıtımın durağan yapısı ve arakesit sınıflarının üstel olarak artması sorunlarını giderme arzusu yatmaktadır.

Dec-Java donatıcı tasarım kalıbı adı verilen özel bir tasarım kalıbından [22] esinlenerek oluşturulmuştur. Böylece sınıf düzeyinde kalıtıma bir alternatif olarak metotların dinamik evrimleştirilebilmesine olanak sağlanmıştır. Dec-Java'da kalıtımla alınan metot yeniden yazılarak işlevi değiştirilmez, bunun yerine orijinal metodun sonuna eklemeler ile metot özelleştirilir.

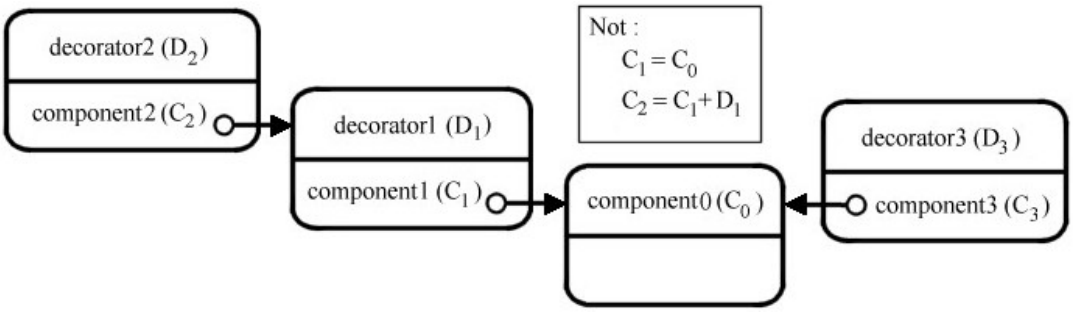
Dec-Java sadece bir nesnenin mevcut metotlarını özelleştirmekle kalmaz, aynı zamanda nesneye yeni metotlar da kazandırabilir. Dec-Java'yı donatıcı tasarım dokusundan ayıran en büyük özellik iletim mekanizmasıdır: Donatıcı tasarım dokusunda danışma mekanizması kullanılırken Dec-Java'da vekalet mekanizması kullanılmaktadır.

6.5.1 Dec-Java'nın Yapısı

Önerilen yaklaşımın temelinde bileşen (*component*) olarak adlandırılan ve bir durumu temsil eden nesnenin çalışma anında donatıcı (*decorator*) adı verilen bir başka nesne içine gömülmesidir. Donatıcı, bileşen nesnesinin dinamik değişimine göre nesnenin anlık durumunu nesnenin davranışlarına yansıtır.

Donatıcı ile bileşen aynı arayüzü gerçeklediğinden çok biçimliliğin yararları kullanılabilir. Gerçeklenen arayüzün adı **`Durum`** olsun. Bir donatıcı nesne **`Durum`** tipinden bir nesne ile ilişkilendirildiğinde, **`Durum`** tipinden herhangi bir **`d`** değişkenine bu donatıcı atanabilir. Bu değişkene **`Durum`** arayüzünün bir metodu olan **`m`** mesajı gönderildiğinde donatıcı bu mesajı ilişiğindeki bileşene iletim mekanizması ile gönderir. Gerektiği takdirde bu metot çağrımının sonunda donatıcı ek komutlar da ekleyebilir. İletim mekanizmasının vekalet olduğu tekrar hatırlatılır; **`this`** işaretçisi iletimden sonra da bileşen yerine donatıcıyı göstermeye devam eder.

Dec-Java iç içe geçmiş donatımları destekler. Bir başka deyişle bir bileşenin donatıcısını da donatan bir başka nesne olabilir. Bu durum Şekil 6.18'deki UML şeması ile örneklenmiştir.



Şekil 6.18: İç içe donatım

Şekil 6.18'de dikkat edilmesi gereken üç nokta vardır:

1. İç içe donatımın ikinci aşamasında bileşen sadece en dipteki bileşenden ibaret değildir, ilk donatıcıyı da içerir. ($C_2 = C_1 + D_1$)

2. Bir bileşen nesnesinin aynı anda birden fazla iç içe geçmemiş donatıcısı olabilir.

3. Bir bileşen aynı türden birden fazla donatıcı tarafından donatılabilir.

Orijinal çalışmada [32] tez konusu olan rol kavramına değinilmemekle birlikte, üstteki iki madde sayesinde Dec-Java'nın rol desteği amaçlı kullanımı mümkündür. İlk madde rol geçişini mümkün kılar; $C_2 = C_1 + D_1$ olduğu için decorator2 ile temsil edilecek rolden decorator1 ile simgelenen role geçiş yapılabilir. İkinci madde ise bir ilişkisel rol hiyerarşisinin kurulabilmesine imkan verir. Üçüncü madde ise Dec-Java'nın nitelikli rolleri desteklemesine izin verir.

Dec-Java, donatım yapısının gerçeklenmesi için Java diline bazı anahtar kelimeler eklemektedir. Bu nedenle Dec-Java kodu koşturulmadan önce bir dönüştürücü program ile Java koduna çevrilmektedir. Söz konusu eklentiler ve dönüşüm mekanizmasının detayları orijinal çalışmada [32] incelenebilir. Dec-Java'nın dönüşüm mekanizması hakkında burada şu noktalar vurgulanabilir:

- Donatıcı, bileşenin metotlarından sadece istediklerini gerçekleyebilir. Donatıcıda gerçeklenmeyen metotlar, vekalet mekanizmasının işletilmesi dışında bir değişikliğe uğramazlar. Bu yaklaşım Bölüm 3.1'de incelenen nesne yüzleri çalışması ile [5] aynıdır.

- Herhangi bir kurucu fonksiyon gerçeklenmez. Gerekli kurucu Dec-Java kodunun Java koduna dönüşümü sırasında oluşturulur. Kurucunun bir bileşen nesnesi veya iç içe donatımlar için başka bir donatıcı olabilecek tek bir parametresi vardır ve yeni donatıcı örnekleri sadece bu kurucu fonksiyon ile üretilir.

6.5.2 Değerlendirme

Dec-Java, nesnelerin uygulama süresince değişmeyecek temel özelliklerinin nesnelerin davranışlarından ayırt edilmesini sağlar, böylece nesnelerin çalışma anında değişebilecek davranışlarının dinamik olarak düzenlenebilmesine olanak verir. Bu açıdan rollerin kazandırmaya çalıştığı tüm esneklikleri sağladığı ve donatım ilişkisinin nesne düzeyinde kalıtım ilişkisine denk olduğu söylenebilir, ancak Dec-Java'nın bazı eksiklikleri bulunmaktadır:

- Bir değer geri döndüren metotların donatımı gerçeklenmemiştir.

- Donatıcı, ilişkili bileşenin metotlarına erişebilir ancak üyelerine erişemez.

- Programcı sadece Dec-Java tarafından oluşturulan kurucu metodu kullanmak zorundadır.

Dec-Java ile çalışırken dikkate alınması gereken en önemli nokta, donatım ilişkisi ile sınıf düzeyinde kalıtımın bir arada kullanılmasının olanaksızlığıdır. Java çoklu kalıtımı desteklemez. Java'da herhangi bir **A** sınıfı bir donatıcı sınıf ile

ilişkilendirildiği zaman gerçekleşen **`class A extends` D_I `implements I {bodyA'}`** dönüşümü, **`A`** sınıfı için Java'nın olanak verdiği tek kalıtım ilişkisini harcamış olacaktır. Halbuki bu tez çalışmasının temel dayanaklarından biri, sınıf düzeyinde kalıtımla nesne düzeyinde kalıtımın birlikte kullanılmasının daha iyi sonuç vereceğidir. Bu açıdan bakıldığı takdirde söz konusu olanaksızlık bir kısıtlama olarak göze çarpmaktadır.

6.6 Geliştirilmiş Rol Modeli

Lee ve Bae yaptıkları çalışmaya [33] geliştirilmiş rol modeli (GRM) adı vermişlerdir. Roller ve gerçek dünya nesneleri arasında önceden beklenilmeyen ilişkiler kurularak kuralların ihlal edilmesinin anormal durumlar oluşabileceğini belirten yazarlar, olası anormallikleri önleyecek yeni bir rol modeli geliştirmişlerdir.

Tez çalışması dahil olmak üzere, rol modellerinin birincil amacı dinamik evrimleşmenin sağlanmasıdır. Rollerin varlıklara atanması bu yaklaşımın sonucudur. Bu düşünce zinciri sonucunda gerçek dünya nesneleri programlamanın temel taşı olarak ele alınmıştır. GRM'yi diğer çalışmalardan ayıran nokta, modelin odak noktasını gerçek dünya nesnelerinden uzaklaştırıp rol nesnelerine kaydırmasıdır. Bu fark Şekil 6.19'da özetlenmiştir.

Diğer çalışmalarda rol tanımı → Amaç: dinamik evrimleşme → Roller oyunculara atanır → Odak noktası (programlama temeli) oyuncular → Ortak çalışmayı oyuncular yönetir.

GRM'de amaç: Ortak çalışma ve anormalliklerin önlenmesi → Nesnelerin iç davranışları (hesaplamalar) nesneler arası (dış) davranışlardan (etkileşimler ve birlikte çalışmalar) ayrılmalı → İç davranışı oyuncular dış davranışı roller modeller → Odak noktası (programlama temeli) roller → Oyuncular rollere bağlanır.

Şekil 6.19: GRM'nin diğer çalışmaların rol sistemlerine yaklaşımlarından farklılığı.

6.6.1 Rol Anormalliği Nedir?

Bir rol sistemi tasarlanırken kararlaştırılan kurallar, roller ile çekirdek nesneler arasındaki ilişkiler kurulurken ihlal edilebilir. Rol sistemini tasarlayanlar ile kullananların farklı kişiler olmasından kaçınılamaz. Özellikle kalıcılık desteği sağlayan uzun ömürlü sistemlerde zamanla ilişki kuralları unutulabilir. Rol sisteminin tasarım kurallarının çiğnenmesi, rol anormalliklerini doğurur. Sıradan rol sistemlerinin aksine, GRM kendi tasarım kurallarının çiğnenmesine kendisi engel olabilmektedir.

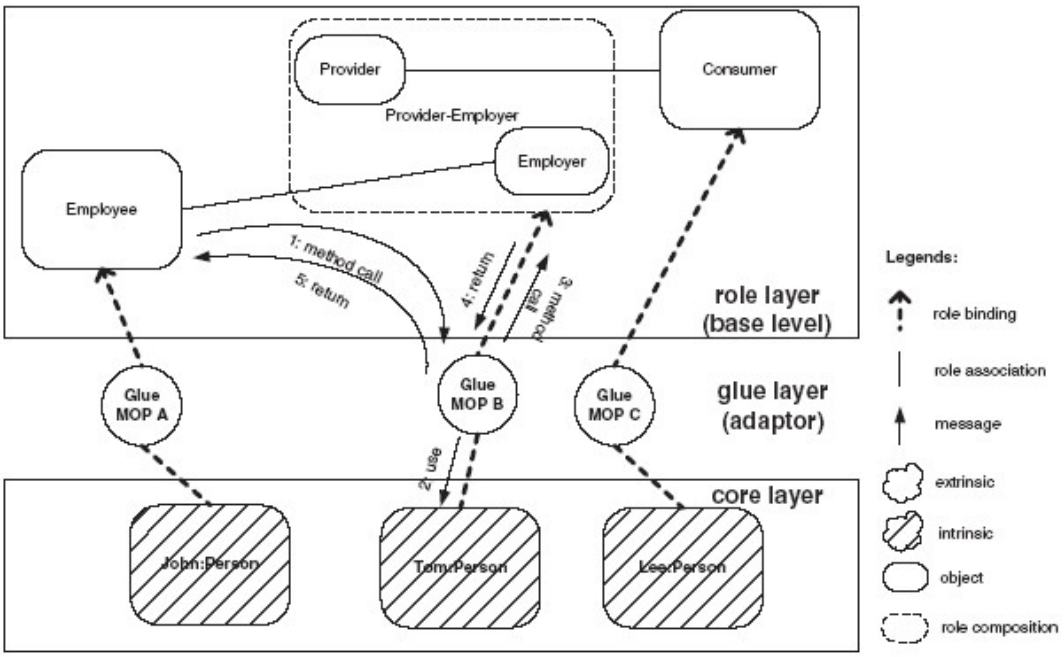
Rol anormallikleri yapısal kuralların çiğnenmesinden veya anormal rol ilişkilerinden kaynaklanabilir. Yapısal kurallar ilişki türlerini belirler: Bir işveren rolünün geçerli olması için bir veya daha fazla çalışan rolünün işveren ile bağlantılı olmasının gerektiği gibi. Anormal rol ilişkileri ise birbiri ile eşleşmemesi gereken rollerin eşleştirilmelerinden doğar. Örneğin yedekleme sunucusu rolü ile depolama sunucusu rolleri aynı nesneye verilmemelidir. Ancak anormal rol ilişkilerinin önlenmesi için gerekli kuralların sisteme nasıl tanıtıldığına orijinal çalışmada [33] değinilmemiştir.

6.6.2 GRM Yapısı

GRM ile her sistem iki alt sistem şeklinde modellenir: Çekirdek sistem ve rol sistemi. GRM bir de bu iki katman arasında ilişkiyi sağlayan üçüncü katman olan tutkal katmanı içerir.

Çekirdek sistemin zaman içinde en az değişiklik gerekebilecek şekilde tasarlanması amaçlanır. Ancak bir sınıfın uzun vadede alabileceği tüm rollerinin kısa vadede tahmini mümkün olmayabilir. Zaman içinde sisteme eklenen roller; yazılımın geliştirilebilirlik, anlaşılırlık ve yeniden kullanılabilirlik gibi niteliklerini azaltabilir. GRM'de rol ve çekirdek sınıfların ayrı katmanlar içinde ele alınması, bu problemin çözümünü sağlar. GRM aynı zamanda rol sistemlerinde oluşabilecek anormal durumların önlenmesini de sağlar.

GRM'nin rol, çekirdek ve tutkal katmanlarından oluşmuş üç katmanlı yapısı Şekil 6.20'de görülebilir. Tutkal katmanı, roller ve çekirdek nesneler arasındaki bağlantıyı kurmakla yükümlüdür.



Şekil 6.20: GRM'nin katmanları

Ortak çalışma işlevlerinin tümünün rollere yüklenmesinin ve aynı katman içindeki nesnelerin etkileşimine sadece rol katmanında izin verilmesinin sonucunda, GRM'de çekirdek nesneler arasında tam bir izolasyon oluşur. Roller dahi doğrudan sahipleri ile ilişkili değildir, ancak atanmış tutkal nesneleri ile ilişkilidirler. Bir varlık birden fazla role sahip olacağı zaman bunlar için ayrı ayrı tutkal nesneleri oluşturulmaz, sadece hepsi ilgili çekirdek nesnenin tutkal nesnesi ile ilişkilendirilir. Bir varlığın oynadığı rollerin tümü bu şekilde bir hiyerarşi olarak değil, bir küme gibi ele alınmış olur. Bir başka deyişle birkaç rol tek bir büyük rol altında birleştirilmiş olur. Örneğin bir kişi hem işveren hem de satıcı rollerine sahip olacaksa, bu rollerin birleşimi ile *işveren-satıcı* rolü oluşturulur. GRM'nin rollere Şekil 6.20'de örneklenen bu yaklaşımı, INADA'nın [26] rollere yaklaşımına benzer. GRM'de roller doğrudan kullanılamaz, sadece sorumlu oldukları ortak çalışma bağlamında görev yapmak için kullanılabilirler.

GRM gerçeklemesinde Java'nın sağlayabileceği yansıtma yeteneklerinden daha fazlasına ihtiyaç duyulduğu için Javassist [34] kullanılmıştır. Javassist, OpenJava [30] gibi bir derleme-zamanı yansıtma gereci olmayıp, ikili kaynak kodu seviyesinde çalışır. Javassist ek bir derleme veya JVM'nin modifikasyonunu gerektirmez, ayrıca sınıfın örneği JVM'ye yüklendiği andan itibaren Java'ya eklediği yansıtma özelliklerinin kullanılmasına izin vermez. Sınıf ve nesnelerin yansıtma işlerini UNIX

dosya hakları olan RWX (sırasıyla okuma, yazma ve çalıştırma) şeklinde tanımlarsak, Java'nın sağladığı davranışsal yansıtma RX, Javassist'in kısmen sağladığı yapısal yansıtma W hakkı gibidir. Yapısal yansıtma ile arayüzler, nesnelerin sınıf tipleri, sınıf kodu içinde istenen kısımlar, vb. değiştirilebilir.

GRM'de aynı çekirdek nesnenin rolleri arasında bir hiyerarşik yapı olmadığı için, aynı mesaja cevap verebilecek roller arasında seçim yapmak mümkün değildir. Bu nedenle; ad çakışmalarının çözümü için baskınlık, birleştirme veya kullanıcı tarafından seçme gibi öncelik kuralları kullanılabilir. Orijinal çalışmada [33] bu kuralların kullanım detaylarına değinilmediği gibi, GRM'nin henüz çakışmaların kullanıcı tarafından çözülmesinin desteklenmediği belirtilmiştir.

6.6.3 Değerlendirme

GRM, rol ilişkilendirme anormalliklerini başarıyla önlemekte ve rol sistemlerinin tasarlanmasına yeni bir bakış açısı getirmektedir. Üç katmanlı yapısı, çekirdek nesnelerin rol nesneleri tarafından üretilmesi, rol nesnelerinin çekirdek nesneden bağımsız olarak test edilebilmesi ve programlama tabanının rol seviyesinde olması; GRM'nin ayırt edici özellikleridir.

GRM belirgin ve kısıtlayıcı eksiklere de sahiptir. Çakışan roller arasında öncelik belirleme kurallarından en önemlisi olan, istenilen durumda istenilen rolün istenilen üyesinin seçimi henüz gerçeklenmemiştir. GRM'nin çalışma anında rol terk edebilme desteği de belirsizdir. Bir rol nesnesinin bağlı olduğu çekirdek nesnesini değiştirme olanağı sunulmakta, ancak bu olanak rol terkinin yerini tutmamaktadır. GRM'nin rol anormalliklerini önlemeye odaklanmış gerçeklemesi, nitelikli rollerin kullanımına olanak vermemektedir. Ayrıca tutkal tabakası sistemde bir darboğaz oluşturmaktadır.

GRM'de programlama tabanının rol seviyesinde olması rol terkini zorlaştıracaktır: Tüm ortak çalışmayı kotaran bir rolün terki kolay olmayacaktır. Ayrıca odak noktası ister rollerde ister oyunculardа olsun, nesnelerin iç ve dış davranışlarının birbirinden ayrılması sağlanabilir. Bu nedenle rol modellerinde odak noktasının oyuncular değil de roller olması bir zorunluluk değil, bir tasarım kararı olarak ele alınmalıdır.

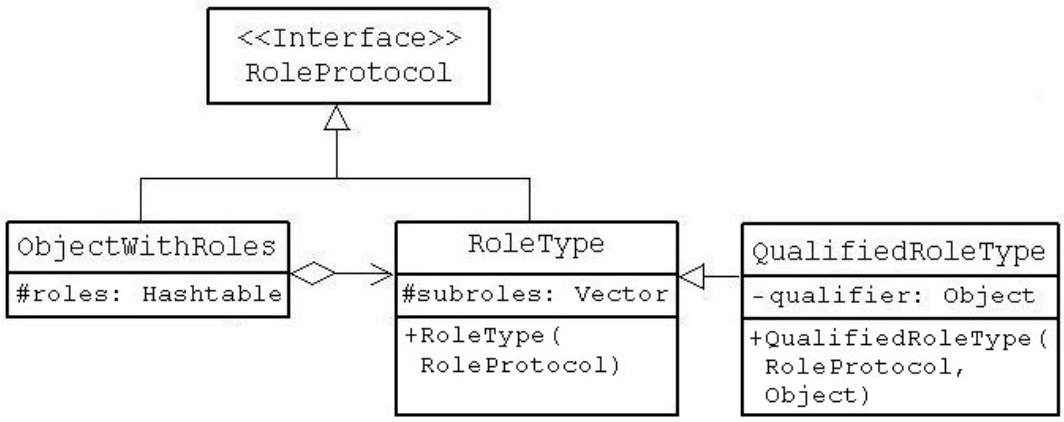
GRM önemli eksiklerine rağmen, anormal rol ilişkilerinin önlenmesi gereksinimini inceleyerek literatüre katkıda bulunmuştur. GRM bu yeteneğe dikkat çekerek tez çalışmasına katkıda bulunmuştur.

6.7 Java ile Rol Hiyerarşisi Yaklaşımı

Schrefl ve Thalhammer'ın çalışması [43], Gottlob, Schrefl ve Röck'ün [2] Smalltalk için yaptıkları ve Bölüm 6.2'de incelenen çalışmanın Java diline uyarlanması ile elde edilmiştir.

6.7.1 Gerçekleme Detayları ve Kullanım Örnekleri

Schrefl'in rol modelinde [43] bir rol hiyerarşisinin kökü olabilecek gerçek dünya nesneleri **`ObjectWithRoles`** sınıfı ile, rol nesneleri ise **`RoleType`** sınıfı ile modellenir. Her iki sınıf, rol kontrolü ve rol geçişi gibi ortak davranışların tanımlandığı **`RoleProtocol`** arayüzünü gerçekler. Nitelikli rollerin modellenmesi için ise **`RoleType`** sınıfından kalıtım ile türetilen **`QualifiedRoleType`** sınıfı kullanılır. Şekil 6.21'de rol modelinin gerçekleme detayları verilmiştir.



Şekil 6.21: Schrefl'in rol modelinin UML şeması

Bir gerçek dünya nesnesini modellemek için, önce rol hiyerarşisinin kökünü oluşturacak olan sınıf **`ObjectWithRoles`** sınıfından (sınıf düzeyinde) kalıtım ile türetilir. Ardından bu varlığın oynayabileceği normal roller **`RoleType`**, nitelikli roller ise **`QualifiedRoleType`** sınıfından kalıtım ile türetilir. Rol hiyerarşisinin kurulumuna erkenden, daha henüz örnek oluşturma aşamasında başlanır: Bir rol nesnesi oluşturulurken kurucu metoduna o rolün sahibi parametre olarak verilir. Bir

nitelikli rol oluştururken ise kurucusuna sahibini gösteren parametreye ek olarak rolün niteleyicisi olan bir nesne de verilir.

Schrefl'in Thalhammer ile yaptığı çalışmada da [43], Gottlob ile çalışmasında [2] olduğu gibi sınıf ve nesne düzeyinde kalıtım ilişkileri bir arada kullanılabilir. Örneğin kullanıcı kendi tanımladığı bir rol sınıfından bir başka rol tipini daha sınıf düzeyinde kalıtım ile türetip bu yeni tip rolün örneklerini başka nesnelere atayarak nesne düzeyinde kalıtım ilişkisi kurabilir.

Rol varlığı kontrolü için **`RoleProtocol.existsAs(String roleTypeName)`**, rol geçişi için ise **`RoleProtocol.as(String roleTypeName)`** veya nitelikli roller için **`RoleProtocol.as(String roleTypeName, Object qualifier)`** metodu kullanılır. Şekil 6.22'de tanımlanan sınıfları kullanan basit kullanım örnekleri Şekil 6.23'te görülebilir.

```
class LegalEntity extends ObjectWithRoles {
  String name;
  /* LegalEntity can take on role Customer;
  this is defined at class Customer */
... }
class Person extends LegalEntity {
  String phoneNo;
  Date birthdate;
... }
class Employee extends RoleType {
  String phoneNo;
  int salary;
  public Employee (Person p, ...) { super (p);
  ... }
... }
class DepartmentManager extends RoleType {
  String department;
  Date promotionDate;
  public DepartmentManager (Employee s, ...) { super (e);
  ... }
... }
class Customer extends QualifiedRoleType {
  int debits;
  public Customer (LegalEntity 1, Company c, ...) {
    super (1,c);
  ... }
... }
class Company extends LegalEntity {
  String phoneNo;
  Date birthdate;
... }
```

Şekil 6.22: Schrefl'in rol modelini kullanan örnek sınıflar

```
/* define root instance representing Mrs. Smith */
Person persSmith = new Person();
perSmith.setName("Smith, Anne");
...
try {
  /* define employee role for Mrs. Smith */
  Employee empSmith = new Employee(perSmith);
  empSmith.setPhoneNo("304-5618");
  empSmith.setSalary(1500);
  /* define department manager role for Mrs. Smith */
  DepartmentManager depMgrSmith = New
    DepartmentManager(empSmith);
  /* role checking and switching */
  if(perSmith.existsAs("Employee")
    ((Employee)perSmith.as("Employee")).setPromotionDate(
      new Date() );
} catch (DuplicateRoleException ex) {...}
catch (NoSuchRoleException ex) {...}
```

Şekil 6.23: Örnek sınıfları kullanan örnek kod

6.7.2 Değerlendirme

Schrefl ve Thalhammer'ın rol modeli[43], API dokümanı ile birlikte http://www.dke.uni-linz.ac.at/research/projects/roles/index.html adresinden temin edilebilir. Rollerin temel özelliklerinin tümünü destekleyen bu çalışma, tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO ile bazı ortak noktalar içermektedir. Her iki çalışma da rollerin gösterimi için en güçlü ve esnek yaklaşım olan hiyerarşik yaklaşımı kullanarak Java diline rol desteği kazandırmaktadır. Rol geçişi komutlarında geri döndürülen nesnelerde tip dönüşümünü yapma zorunluluğu iki çalışmada da ortaktır. Benzerlikler bu noktada sona ermektedir. JAWIRO rollerin temel özelliklerine ek olarak yeni katkılarla birlikte rollerin ileri düzey yeteneklerini gerçeklemektedir. İlk bakışta **`Person`** ve **`Company`** sınıflarından örneklerin **`Customer`** rolünü oynayabilmesi nedeniyle farklı türden nesnelerin aynı türden role sahip olabilecekleri, dolayısıyla nesne düzeyinde çoklu kalıtımın desteklendiği düşünülebilir. Ancak Şekil 6.22 incelendiğinde aslında **`Person`** ve **`Company`** sınıflarının ortak bir sınıftan, **`LegalEntity`** sınıfından kalıtımla türetildiği görülür. Bu bölümde incelenen çalışmanın JAWIRO ile daha ayrıntılı karşılaştırmaları için Bölüm 9 incelenebilir.

7. JAVA DİLİNE ÖZGÜ YETENEKLER

Bu bölümde Java dilinin tez çalışmasına katkıda bulunan özellikleri incelenecektir.

7.1 Dinamik Tipleme ve Tip Dönüşümü

Kuvvetli tiplemeli olarak nitelendirilen dillerde bir değişkenin tipi asla değiştirilemez. Ancak kalıtım ilişkisindeki "üst sınıfın beklendiği her bağlamda alt sınıf kullanılabilir" kuralı geçerliliğini korur. Örneğin bir **A** türünden değişken bir işaretçi olarak tanımlanırsa, bu değişkene **A**'nın alt sınıfı olan **B** türünden nesneler atanabilir. Lisp gibi *dinamik tiplemeli* olarak nitelendirilen dillerde ise bir değişkene çalışma anında değişik tiplere ait değişken ve sınıflar atanılabilir.

```
Interface I {
    void methodI();
}
Class A {
    void methodA() {
    ...}
}
Class B extends A implements I
    void methodB() {
    ...}
    void methodI() {
    ...}
}
Class C implements I {
    void methodI() {
    ...}
}
...
A Adata;
B Bdata = new B();
C Cdata = new C();
I Idata;
B temp;
Adata = Bdata;                    //DOĞRU: Dinamik tipleme örn.
Idata = Bdata; Idata.methodI(); //DOĞRU: Dinamik tipleme örn.
Idata = Cdata; Idata.methodI(); //DOĞRU: Dinamik tipleme örn.
//Adata.methodB();                //YANLIŞ: Tip dönüşümü gerekli
((B)Adata).methodB();             //DOĞRU: Tip dönüşümü (1.yol)
temp = (B)Adata; temp.methodB();//DOĞRU: Tip dönüşümü (2.yol)
```

Şekil 7.1: Çalışma anında tipleme ve tip dönüşümüne örnekler.

Java kuvvetli tiplemeli bir dildir. Java dilinin arayüzleri de kuvvetli tipleme kuralına bağlıdır. Java arayüzlerinin birbirlerini gerçekleyemez, ancak I tipindeki bir değişken ile tutulan bir arayüz aslında I arayüzünü gerçekleyen farklı sınıflara ait nesnelerden herhangi biri olabilir. Çalışma anında bir değişkene bu kurallara uyan farklı sınıflara ait nesneler atanabilir. Bu kurallara uyan nesneler, çalışma anında geçici olarak birbirlerine dönüştürülerek kullanılabilir. Tip dönüşümünün kullanımı Şekil 7.1'de verilen örnek kod ile gösterilmiştir.

7.2 Serileştirme (Serialization)

Java dilinin desteklediği serileştirme özelliği, nesnelerin kalıcı ortama saklanmak veya ağ üzerinden iletilmek üzere sekizliler dizisi haline getirilmesini ve daha sonra nesnelerin bu diziden durum bilgileri kaybedilmeden yeniden oluşturulmasını sağlar. Serileştirme özelliği, Java diline temel kalıcılık yeteneği kazandırmaktadır.

7.2.1 Genel Bilgiler

Serileştirme özelliği, **`java.io.Serializable`** veya **`java.io.Externalizable`** arayüzünü destekleyen sınıflar tarafından kullanabilir. **`Serializable`** arayüzünü gerçekleyen sınıfların ek kod barındırmasına gerek yoktur, Java'nın serileştirme mekanizması sınıfın tüm üyelerinin durumunun saklanılabilmesi için gerekli kodu oluşturur. Ancak kullanıcı bu mekanizmanın istediği aşamalarına müdahale edebilir. **`Externalizable`** arayüzünü gerçekleyen sınıflar ise kendi içeriklerinin sekizli dizisinde saklanış biçiminden tamamen kendileri sorumludur. Bu yüzden **`Serializable`** arayüzünün üstünde durulacaktır.

Bir nesneyi serileştirip kalıcı ortama yazmak ve geri okumak **`FileOutputStream`** ile ilişkilendirilmiş **`ObjectOutputStream`** ve **`FileInputStream`** ile ilişkilendirilmiş **`ObjectInputStream`** nesneleriyle yapılan, karmaşık olmayan işlemlerdir. Aynı dosyaya birden fazla nesne ardışık olarak yazılabilir, bu durumda nesneler yazıldıkları sıra ile dosyadan yüklenerek geri kazanılır. Tüm bu sınıflar **`java.io`** paketinde bulunmaktadır.

Bir nesne serileştirilip saklandığında, o nesneden ulaşılabilen tüm diğer nesneler de saklanmaktadır. Bir başka deyişle, nesnenin, kendileri de başka nesneler olan üyeleri

de saklanır. Varsayılan düzende **`static`** ve **`transient`** anahtar kelimeleri ile tanımlanmış üyeler saklanmaz, ancak bu davranış saklanılması istenen alanlar **`serialPersistentFields`** adlı özel bir alanda belirtilebilir. Buradaki tek kısıtlama, iç sınıfların[1] sadece **`final static`** üyeler içermesi zorunluluğudur.

```
// Serialize today's date to a file.
    FileOutputStream f = new FileOutputStream("dosya_adi");
    ObjectOutputStream s = new ObjectOutputStream(f);
    s.writeObject("Bugünün tarihi: ");
    s.writeObject(new Date());
    s.flush();
    s.close();
// Deserialize a string and date from a file.
    FileInputStream in = new FileInputStream("dosya_adi");
    ObjectInputStream so = new ObjectInputStream(in);
    String today = (String)so.readObject();
    Date date = (Date)so.readObject();
    so.close();
```

Şekil 7.2: Bir nesneyi serileştirerek kalıcı ortama yazma ve kalıcı ortamdan okuma. Aykırı durum işleme kodları gösterilmemiştir.

Java dilinin serileştirme özelliğinin bir zayıf noktası bulunmaktadır: Bir a nesnesinin b üyesine c nesnesinde referans varsa, a ve c geri yüklenince b'ye olan referans kopup a.b ve c.b aynı değerde iki farklı nesne haline doner. Programcı a.b ve c.b'nin aynı nesneyi göstermesini istiyorsa ya bu nesneleri aynı fiziksel dosya içerisine serileştirmelidir. Aksi halde programcı geri yükleme sonrasında işaretçi eşitliğini yeniden kurmalıdır.

7.2.2 Ek Bilgiler

Bir nesne aynı dosya akısına **`writeObject`** komutu ile birden fazla kez yazılırsa JVM değişim dosyası algoritmasını kullanır: Nesnenin akıdaki ilk örneğine referans verip sadece yapılan yeni değişiklikleri akıya yazar. Bu davranışın istenmediği durumlarda **`writeObject`** yerine **`writeUnshared`** metodu kullanılır. Benzer şekilde kalıcı ortamdan okuma yaparken **`readUnshared`** metodu kullanıldığında

---

[1] Java terminolojisinde esas sınıfın kaynak kod dosyası içersinde ancak esas sınıfın kodunun dışında tanımlanan sınıflara iç sınıf adı verilir, üye sınıflarla karıştırılmamalıdır. Üye sınıf ise esas sınıfın kodu içerisinde tanımlanır ve tıpkı bir üye değişken veya metod gibidir. A sınıfının B adlı iç sınıfı ve C adlı üye sınıfı varsa bunların sınıf adları sırasıyla **`"A.B"`** ve **`"A$C"`** olacaktır.

artık o nesne akıda birden fazla yerde bulunsa dahi o nesneyi gösteren işaretçi geçersiz hale gelmiş olur. Bu durumda ardışık **`readUnshared`** kullanımları aykırı durum oluşturur (**`ObjectStreamException`**). **`writeUnshared`** metodu ile yazılan nesneler okuma anında **`readUnshared`** metodu ile okunmak zorunda değillerdir.

`ObjectOutputStream` sınıfı bir nesneyi akıya yazmadan önce o nesne için **`ObjectStreamClass`** sınıfından bir tekil tanımlayıcı üreterek bunu kalıcı ortama yazar. **`ObjectInputStream`** sınıfının **`ObjectStreamClass`** üyesinin **`lookup`** metodu kullanılarak akıda istenilen bir sınıftan nesne veya nesnelerin bulunup bulunmadığı öğrenilebilir. **`ObjectStreamClass`** üyesi ile akıdaki nesnenin tekil tanımlayıcısı, üyeleri hakkında yansıtma bilgileri (akıdaki nesnenin tüm üyelerinin tipleri ve değerleri, vb.) gibi özelliklerine erişilebilir.

Serileştirme işlemi sırasında dikkat edilmesi gereken noktalar bulunmaktadır. Bunlardan birincisi güvenlik konusudur. Nesnelerin varsayılan saklanış biçimi kalıcı ve geçici verilerinin bütünlüğünü sağlasa da verileri şifreleme yapmadan açık metin şeklinde saklamaktadır. Kullanıcı nesneyi kalıcı ortamdan geri yüklerken bir başkası tarafından kurcalanmış bir dosyadan okuma yapılmadığından emin olmalı ve gerekli doğrulama önlemlerini almalıdır. Bunlardan biri nesnenin tekil tanımlayıcısının akıdaki veri ile uyuşup uyuşmadığını kontrol etmektir. Tekil tanımlayıcı üretilirken NIST'nin SHA-1 algoritması kullanılır.

Serileştirmede dikkat edilmesi gereken ikinci önemli nokta ölümcül kilitlenmelerdir. **`ObjectInputStream`** kurucusu engellemeli bir metottur. Bu metodun kullanacağı dosya akısı henüz ilgili **`ObjectOutputStream`** kurucusunun tamamlanmasını bekliyorsa bir ölümcül kilitlenme oluşacaktır.

Serileştirilen nesnelerin versiyonlandırılması ve serileştirme protokolünün detayları tez çalışmasında kullanılmadığından bu bölümde incelenmemiştir.

7.3 Yansıtma (Reflection)

Bu bölümde Java'nın yansıtma ile ilgili sınıfları ve bunların **`public`** metotları incelenecektir. Özetle, yansıtma yeteneği yüklenen sınıfların üye metodları ve değişkenleri hakkında bilgi toplamaya ve bu üyelere nesne düzeyinde erişmeye yarar.

7.3.1 Genel Bilgiler

Java'nın **`java.lang.reflect`** paketi ile kullanım imkanı bulan yansıtma yeteneği, JVM'ye yüklenen sınıfların iç yapısı hakkında incelemeler yapılmasına ve inceleme sonucu keşfedilen üye ve metotların kullanımına olanak tanır. Yansıtma yeteneği kullanılarak erişilen sınıf ve nesnelere *yansıtılan sınıf* ve *yansıtılan nesne* adları verilir. Yansıtma API'si üç ana sınıf üzerine kurulmuştur:

- **`Field`** sınıfı: Bu sınıftan olan bir nesne, yansıtılan nesnenin bir üyesini temsil eder. Temsil edilen üye üzerinde okuma ve yazma işlemleri yapılabilir. Hem sınıf üyeleri (**`static`** anahtar kelimesi ile tanımlanmış üyeler) hem de örnek üyeleri (**`static`** olmayan üyeler) **`Field`** sınıfından nesnelerce temsil edilebilir.

- **`Method`** sınıfı: Bu sınıftan olan bir nesne, yansıtılan nesnenin soyut, sınıf veya örnek metotlarından herhangi birini temsil edebilir. **`Method`** sınıfının metotları ile yansıtılan metodun parametrelerine, dönüş tipine ve kontrol ettiği aykırı durumlara erişilebilir. Ayrıca **`Method.Invoke`** metodu ile yansıtılmış metodun çalıştırılması da mümkündür.

- **`Constructor`** sınıfı: Yansıtılan sınıfın kurucu fonksiyonları **`Method`** sınıfı yerine **`Constructor`** sınıfı ile temsil edilir. Yansıtılan sınıftan örnekler de bu sınıf kullanılarak oluşturulur.

Bu üç sınıf da **`java.lang.reflect.Member`** arayüzünü gerçekler. Bu arayüzün metotları yansıtılan üyenin temel tanımlama bilgilerinin sorgulanmasını sağlar. Tanımlama bilgileri üyenin adı, üyeyi tanımlayan sınıf ve üyeye erişim biçiminden (**`public`**, **`protected`**, vb.) oluşur.

Yansıtma API'sinin 3 temel sınıfı kullanılmadan önce, bunlara değer atanması gerekir. Değer atama işlemi ise **`java.lang.Class`** sınıfının metotları kullanılarak yapılır. Bu yüzden öncelikle **`java.lang.Class`** sınıfının yansıtma yetenekleri anlatılacak, ardından yansıtmanın temel sınıfları incelenecektir.

7.3.2 Yansıtma Kullanımı İçin Bilgi Toplanması

Her Java sınıfın üst sınıfı olan **`Class`** sınıfının hemen hemen tüm metotları yansıtma ile ilişkili yetenekler sağlar. Bunlardan yansıtma API'sinin önceki bölümde kısaca değinilen üç sınıfından olan nesnelere değer atanmasını sağlayan metotlar şunlardır:

- **`Class getSuperClass():`** Çalıştırıldığı sınıfın (veya arayüzün) kalıtım hiyerarşisindeki ebeveynini, başka bir deyişle üst sınıfını (veya arayüzünü) döndürür.

- **`Class[] getClasses():`** Çalıştırıldığı sınıfın kendileri de birer sınıf veya arayüz olan **`public`** üyelerini içeren bir dizi döndürür. Bu diziye kalıtım ile alınmış üyeler de dahildir.

- **`Constructor getConstructor(Class[] paramTypes):`** Çalıştırıldığı sınıfın aldığı parametre listesi ile aynı olan **`public`** kurucu fonksiyonunu döndürür.

- **`Constructor[] getConstructors():`** Çalıştırıldığı sınıfın tüm **`public`** kurucu fonksiyonlarını içeren bir dizi döndürür.

- **`Field getField(String name):`** Çalıştırıldığı sınıfın verilen adlı public üyesini temsil eden Field nesnesini döndürür. İlgili üye aranırken önce sınıfın kendisinde tanımlanmış aynı adlı **`public`** üye aranır. Uygun üye bulunmazsa kalıtım ile sahip olunmuş **`public`** üyeler rekürsif olarak aranır. Rekürsif aramada öncelikle arayüzler sayesinde sahip olunan üyelere bakılır, üst arayüzlerde uygun üye bulunamazsa üst sınıflar araştırılır. Araştırma yönü kalıtım hiyerarşisinde aşağıdan yukarı doğrudur.

- **`Field[] getFields():`** Çalıştırıldığı sınıfın tüm **`public`** üyelerini içeren bir dizi döndürür. Kalıtım yolu ile alınan **`public`** üyeler bu diziye dahildir.

- **`Method getMethod(String name, Class[] parameterTypes):`** Çalıştırıldığı sınıfın metotları arasında **`name`** parametresi ile belirtilen ada ve **`parameterTypes`** dizisi ile belirtilen parametrelere sahip olan metodu temsil eden **`Method`** nesnesini döndürür. Arama algoritması **`getField`** ile aynıdır.

- **`Method[] getMethods():`** Çalıştırıldığı sınıfta tanımlanmış veya kalıtımla alınmış tüm **`public`** metotlarını içeren bir diz döndürür.

Bu metotların **`Declared`** kelimesi içeren adlara sahip eşlenikleri de bulunmaktadır. Bu eşleniklerin yukarıda tanıtılanlardan farkı, geri döndürülen dizi veya nesneler arasında kalıtım ile kazanılan bileşenlerin bulunmayıp, **`public`** dışında bileşenlerin de bulunmasıdır. Alınan parametreler ve döndürülen nesneler her eşlenik çift için aynı olduğundan söz konusu metotların sadece adını vermekle yetinilecektir:

- **`getDeclaredClasses`**,
- **`getDeclaredConstructor`**,
- **`getDeclaredConstructors`**,
- **`getDeclaredField`**,
- **`getDeclaredFields,`**
- **`getDeclaredMethod`**,
- **`getDeclaredMethods.`**

Örneğin **`getDeclaredClasses`** metodu çalıştırıldığı sınıfta tanımlanan tüm üyeleri içeren bir dizi döndürür. **`getClasses`** metodundan farklı olarak döndürülen diziye kalıtım ile alınmış üyeler dahil değildir, ancak **`public`**, **`protected`** ve **`private`** üyeler dahildir. Ancak **`protected`** ve **`private`** üyelere erişmeden önce, eğer JVM için geçerli olan güvenlik yöneticisi izin veriyorsa, bu üyelerin **`setAccessible`** adlı metodu **`true`** parametresi ile çalıştırılmalıdır.

`Class` sınıfının yansıtma ile ilgili diğer metotlarından önemli görülenleri şunlardır:

- **`static Class forName(String className):`** Parametre olarak aldığı ad ile ilişkili sınıf veya arayüzü temsil eden bir **`Class`** nesnesi geri döndürür.

- **`String getName():`** Çalıştırıldığı nesnenin ait olduğu sınıfın adını içeren bir karakter katarı döndürür.

- **`Class getDeclaringClass():`** Çalıştırıldığı nesnenin ait olduğu sınıf bir başka sınıf içinde bildirilmiş bir üye ise, kendisini tanımlayan bu sınıfı simgeleyen Class nesnesini döndürür.

- **`boolean isInstance(Object obj):`** **`instanceOf`** operatörünün dinamik versiyonudur. **`obj`** nesnesi metodu çalıştırılan **`Class`** nesnesinin temsil ettiği sınıfa çevrilebiliyorsa **`true`** döndürür. Bir başka deyişle parametre olarak alınan nesnenin, temsil edilen sınıfın kendisinden ya da alt sınıflarından bir örnek olup olmadığını kontrol eder.

- **`boolean isAssignableFrom(Class cls):`** Temsil edilen sınıf parametre olarak alınan sınıf ile aynı ise ya da temsil edilen sınıf parametre olarak alınan sınıfın üst sınıflarından biri ise **`true`** döndürür. "Alt sınıf mıdır?" kontrolü yapılan **`isInstance`** metodunun aksine burada "Üst sınıf mıdır?" kontrolü yapılmaktadır.

7.3.3 `java.lang.reflect.Field` Sınıfı

Bu sınıftan olan bir nesne, yansıtılan nesnenin bir üyesini temsil ederek bu üyeye dinamik erişim sağlar. Bir **`Field`** nesnesi ile temsil edilen bir üyenin değeri okunabilir ve güncellenebilir. **`Field`** sınıfının önemli metotları şunlardır:

- **`Object get(Object obj):`** Metodu çalıştırılan **`Field`** nesnesinin temsil ettiği alanı **`obj`** parametresi ile alınan nesnede arayarak bulduğu değeri geri döndürür. Java ilkelleri ilgili nesnesi ile sarılır: Örneğin tamsayı ilkeli üye için **`Integer`** sınıfından bir nesne döner.

- **`void set(Object obj, Object value):`** **`obj`** parametresi ile bildirilen nesnesinde metodu çalıştırılan **`Field`** nesnesinin temsil ettiği alanın değerini **`value`** parametresi olarak değiştirir.

- **`boolean equals(Object obj):`** Metodun çağrıldığı **`Field`** nesnesi ile parametre olarak alınan **`Field`** nesnesinin aynı sınıfın aynı üyesini temsil ediyorsa **`true`** döndürür. Parametre olarak **`Field`** nesnesi alması gerektiği açıkça belirtilmemesine rağmen, parametre olarak alanın tanımlandığı nesne verilirse dahi geri dönen değer **`false`** olacaktır.

- **`Class getDeclaringClass():`** Temsil edilen üyenin ait olduğu sınıfı temsil eden bir **`Class`** nesnesi döndürür.

- **`String getName():`** Field nesnesinin temsil ettiği üyenin adını döndürür.

7.3.4 `java.lang.reflect.Method` Sınıfı

`Method` sınıfı ile yansıtılan metodun parametreleri, dönüş tipi ve kontrol ettiği aykırı durumlara erişilebileceği gibi temsil edilen metot çalıştırılabilir. **`Method`** sınıfının önemli metotları şunlardır:

- **`int getModifiers():`** Temsil edilen metodun sıfatlarının tümü hakkında bilgi içeren, ikili düzene göre 12 basamağı olan bir tamsayı döndürür. Bu sayı **`java.lang.reflect.Modifier`** sınıfının tümü **`static`** olan, **`boolean`** değer döndüren ve bir tamsayı parametre alan **`isAbstract`**, **`isPublic`** gibi metotları ile test edilebilir.

- **`Class getReturnType():`** Temsil edilen metodun döndürdüğü değerin tipini yansıtan bir **`Class`** nesnesi döndürür.

- **`String getName():`** Yansıtılan metodun adını döndürür.

- **`Class[] getParameterTypes():`** Yansıtılan metodun parametrelerini temsil eden **`Class`** tipi nesneleri, tanımlama sırasına göre içeren bir dizi döndürür.

- **`Class[] getExceptionTypes():`** Yansıtılan metot tanımlanırken başlığında yakalandığı belirtilen aykırı durumları temsil eden **`Class`** tipi nesneleri, tanımlama sırasına göre içeren bir dizi döndürür.

- **`Object invoke(Object obj, Object[] args):`** Yansıtılan metodu **`obj`** parametresi ile alınan nesne için çalıştır. Yansıtılan metodun döndürdüğü değeri **`invoke`** metodu **`Object`** nesnesi olarak döndürür. Eğer yansıtılan metodun geri döndürdüğü hiçbir değer yoksa **`invoke`** metodu **`null`** değeri döndürebilir. Yansıtılan metodun parametreleri ise **`args`** dizisi ile sağlanır. Yansıtılan metot hiçbir parametre almıyorsa **`args`** parametresi de **`null`** olabilir.

- **`Class getDeclaringClass():`** Temsil edilen metodun ait olduğu sınıfı temsil eden bir **`Class`** nesnesi döndürür.

7.3.5 `java.lang.reflect.Constructor` Sınıfı

Yansıtılan sınıfın kurucu fonksiyonlarını temsil eder. **`Constructor`** sınıfının önemli metotları şunlardır:

- **`int getModifiers():`** Temsil edilen kurucu fonksiyonun sıfatlarının tümü hakkında bilgi içeren, ikili düzene göre 12 basamağı olan bir tamsayı döndürür. Bu sayı **`java.lang.reflect.Modifier`** sınıfının tümü **`static`** olan, **`boolean`** değer döndüren ve bir tamsayı parametre alan **`isAbstract`**, **`isPublic`** gibi metotları ile test edilebilir.

- **`boolean equals(Object obj):`** Metodun çağrıldığı **`Constructor`** nesnesi ile parametre olarak alınan **`Constructor`** nesnesinin aynı sınıfın aynı üyesini temsil ediyorsa **`true`** döndürür.

- **`String getName():`** Yansıtılan kurucu metodun adı sınıf adı ile aynı olacağı bariz olmakla birlikte, bu metodun bulunması **`Constructor`** ve **`Method`** nesnelerinin benzer şekilde kullanımını kolaylaştıran bir özelliktir.

- **`Class[] getParameterTypes():`** Yansıtılan kurucu fonksiyonun parametrelerini temsil eden **`Class`** tipi nesneleri, tanımlama sırasına göre içeren bir dizi döndürür.

- **`Class[] getExceptionTypes():`** Yansıtılan kurucu fonksiyon tanımlanırken başlığında yakalandığı belirtilen aykırı durumları temsil eden **`Class`** tipi nesneleri, tanımlama sırasına göre içeren bir dizi döndürür.

- **`Object newInstance(Object[] initargs):`** Yansıtılan kurucu fonksiyon kullanılarak temsil edilen sınıftan yeni bir nesne oluşturulmasına yarar. Kurucu fonksiyonun alması gereken parametreler **`initargs`** dizisi ile sağlanır. Yansıtılan metot hiçbir parametre almıyorsa **`initargs`** parametresi de **`null`** olabilir.

- **`Class getDeclaringClass():`** Temsil edilen kurucu fonksiyonun ait olduğu sınıfı temsil eden bir **`Class`** nesnesi döndürür.

Özetlersek **`Constructor`** sınıfı **`Method`** sınıfına benzemekle birlikte, **`invoke`** metodu yerine **`newInstance`** metodu sağlayarak yansıtılan sınıftan yeni bir nesne türetilmesini sağlar.

7.4 Şifreleme

JDK 1.4 sürümünden itibaren şifreleme yeteneği Java dilinin doğal bir parçası haline getirilmiştir. Simetrik veya açık anahtar şifreleme işlemleri için Java çalışma anı ortamı (JRE) ile hazır gelen algoritmalar veya bir şifreleme servisi sağlayıcısının sağladığı algoritmalar kullanılabilir. Üstelik programcılar kendi şifreleme algoritmalarını hazırlayıp bunlardan Java şifreleme mimarisi için servisler hazırlayabilir. Bu bölümde ikincil depolama ortamındaki bir açık dosyanın nasıl şifreleneceğini ve ardından şifrelenmiş dosyanın nasıl açılacağının adımları incelenecektir.

7.4.1 Ön Hazırlıklar

Bir çok işletim sisteminde hazır gelmesi nedeni ile örnek şifreleme algoritması olarak 64-bit DES kullanılacaktır. Şifreleme kodu için gereksinimler şunlardır:

- İlk değer atama vektörü olarak kullanılmak üzere 8 sekizliden oluşan bir dizi.
- Anahtar olarak kullanılacak istenilen uzunlukta bir karakter katarı.
- Kaynak dosyadan okunacak veri için tampon olmak üzere bir sekizli dizisi.
- Algoritmayı yürütecek olan, **`javax.crypto.Cipher`** sınıfından bir nesne.

`Cipher` nesneleri kullanılmadan önce bunlara ilk değer ataması işlemleri yapılmalıdır. **`Cipher`** nesneleri sistemde mevcut olan herhangi bir algoritma ile çalışabilir ve ister şifreleme ister şifre çözme için kullanılabilir. Programcı ilk değer atama işlemleri sırasında bu konularda karar verir. Gerekli adımlar şunlardır:

- **`java.security.spec.KeySpec`** arayüzüne uyan bir anahtar gerçeklemesi elde edilmelidir. Anahtar gerçeklemesinin kullanılacak algoritmadan bağımsız şekilde ve programcıya şeffaf olarak elde edilmesi için, bir sonraki adımda algoritmaya bağımlı anahtar elde etmeden önce, bu adımda sıradan bir karakter katarı olan anahtardan **`KeySpec`** türünde bir anahtar tanımlaması elde edilir.

- **`javax.crypto.SecretKeyFactory`** sınıfının statik **`getInstance( String algorithm)`** metodu veya türevleri ile **`javax.crypto. SecretKeyFactory`** türünden bir gizli anahtar üreteci elde edilir. Parametre olarak verilen karakter katarında kullanılacak algoritmanın adı verilir.

- Önceki adımda elde edilen gizli anahtar üretecinden ilk adımda elde edilen anahtar tanımlaması kullanılarak, **`javax.Crypto.SecretKey`** türünden algoritmaya bağımlı bir gizli anahtar örneği elde edilir. Bu iş için kullanılan **`SecretKeyFactory.generateSecret(KeySpec spec)`** metodu ile **`getInstance`** metodu birbirine zincirlenebilir.

- **`javax.crypto.Cipher`** örnekleri **`Cipher.getInstance`** sınıf metodu türevlerinden biri ile elde edilir. **`getInstance`** metodu için gerekli olan parametre **`SecretKey.getAlgorithm`** metodu ile elde edilebilir.

- **`Cipher`** örneklerine ilk değer atanması da gizli anahtarın elde edilmesi gibi algoritmaya bağımlı olacaktır. Bu bağımlılığın da programcıya şeffaf olarak kotarılması için önce **`java.security.spec.AlgorithmParameterSpec`** arayüzüne uyan bir tanımlayıcı elde edilmelidir. Hazır gelen her algoritmanın hem ilk adımdaki **`KeySpec`** arayüzüne uyumlu bir örnek elde etmek için hem de bu adımda **`AlgorithmParameterSpec`** arayüzüne uyumlu bir örnek elde etmek için kullanılan sınıfları mevcuttur. Örneğin DES algoritmasını MD5 karıştırma algoritması ile birlikte kullanmak için ilk adımda kullanılacak sınıf **`javax.crypto.spec.PBEKeySpec`**, ikinci adımda kullanılacak algoritma adı **`"PBEWithMD5AndDES"`**, bu adımda kullanılacak sınıf adı ise **`javax.crypto.spec.PBEParameterSpec`** olacaktır.

- Nihayet **`Cipher`** örneklerine ilk değer atamak için bu son adımda **`Cipher.init`** metodu türevlerinden biri kullanılabilir. Bu türevlerin tümünde ortak olan ilk parametre çalışma biçimini belirler. **`Cipher`** örneğini şifreleme amaçlı kullanmak için **`Cipher.ENCRYPT_MODE`**, şifre çözme kullanmak için ise **`Cipher.DECRYPT_MODE`** sabitleri kullanılabilir.

Şimdiye dek anlatılan tüm bu adımlar Şekil 7.3'te verilen kod ile örneklenmiştir. İşlemler sırasında oluşabilecek aykırı durumlar da Şekil 7.3'te görülebilir.

```
private Cipher enCipher, deCipher;
private byte[] salt = { (byte)0x13,(byte)0xFC,(byte)0x59,
    (byte)0xA7,(byte)0x27,(byte)0x76,(byte)0x73,(byte)0x42 };
private String passPhrase = "ACILSUSAMACIL";
private int iterations = 19;
private void initCiphers( ) {
  try {
    KeySpec keySpec = new PBEKeySpec(
       passPhrase.toCharArray(), salt, iterations);
    SecretKey key = SecretKeyFactory.getInstance(
       "PBEWithMD5AndDES").generateSecret(keySpec);
    enCipher = Cipher.getInstance(key.getAlgorithm());
    deCipher = Cipher.getInstance(key.getAlgorithm());
    AlgorithmParameterSpec paramSpec = new
       PBEParameterSpec(salt, iterations);
    enCipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
    deCipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
  }
  catch(java.security.InvalidAlgorithmParameterException e)
    { e.printStackTrace(); }
  catch( java.security.spec.InvalidKeySpecException e )
    { e.printStackTrace(); }
  catch( java.security.NoSuchAlgorithmException e )
    { e.printStackTrace(); }
  catch( java.security.InvalidKeyException e )
    { e.printStackTrace(); }
  catch( javax.crypto.NoSuchPaddingException e )
    { e.printStackTrace(); }
}
```

Şekil 7.3: Şifreleme ve şifre açma işlemleri için gerekli ön hazırlıklar

7.4.2 Açık Dosyanın Şifrelenmesi ve Şifrelenmiş Dosyanın Açılması

Gerekli hazırlık aşamaları tamamlandıktan sonra, ikincil depolama ortamındaki bir dosyayı şifrelemek veya şifrelenmiş bir dosyayı açmak çok kolaydır. Bu amaçlarla **`javax.crypto`** paketindeki **`CipherInputStream`** ve **`CipherOutputStream`** sınıfları kullanılır. Bir **`CipherInputStream`** örneği **`java.io.InputStream`** gibi

bir giriş akısı ile bir **`Cipher`** nesnesinden elde edilir. Benzer şekilde bir **`CipherOutputStream`** örneği **`java.io.OutputStream`** gibi bir çıkış akısı ile bir **`Cipher`** nesnesinden elde edilir. **`Cipher`** nesnesinin şifreleme veya şifre çözme seçeneklerinden hangisi ile oluşturulmuşsa ilgili akı nesnesi şifreleme veya şifre çözme işi için kullanılır. Şekil 7.4'te bir düz metin akısı olan kaynağı şifreleyerek kalıcı ortama yazan bir metod örneği, Şekil 7.5'te ise şifreli bir kaynak akısını açan bir metod örneği görülebilir. Örneklerde kullanılan üyelerin tanımlanması ve ön hazırlıklar ise Şekil 7.3'te verilmişti.

```
void encrypt( InputStream in, OutputStream out ) {
  try {
    out = new CipherOutputStream( out, enCipher );
    int numRead = 0;
    while ((numRead = in.read(buf)) >= 0) {
      out.write(buf, 0, numRead);
    }
    in.close();
    out.close();
  }
  catch( IOException e ) { e.printStackTrace(); }
}
```

Şekil 7.4: Düz metin kaynak akısının şifrelenmesi

```
void decrypt( InputStream in, OutputStream out ) {
  try {
    in = new CipherInputStream(in, deCipher);
    int numRead = 0;
    while ((numRead = in.read(buf)) >= 0) {
      out.write(buf, 0, numRead);
    }
    in.close();
    out.close();
  }
  catch( java.io.IOException e ) { e.printStackTrace(); }
}
```

Şekil 7.5: Şifreli kaynak akısının açılması

7.5 Uygulama Ayarlarının Saklanması

Java dili ile birlikte gelen **`java.util.Properties`** sınıfı, bir uygulama ile ilgili küçük detayların kalıcı ortama kaydedilmesi ve kalıcı ortamdan geri yüklenmesi için basit ve kullanışlı bir arayüz sunar. Bir çok uygulama; çalışması ve durumu ile ilgili çeşitli bilgilerin kalıcı olmasına, program kapatılıp yeniden çalıştırıldığında söz konusu bilgilere yeniden ulaşmaya gerek duyar. Örneğin bir e-posta istemcisi e-posta

hesabına ulaşmak için gerekli hesap bilgilerinin kalıcı olmasına gerek duyar ve hiçbir kullanıcı da her oturumda e-posta sunucu adresi, doğrulama metodu, kullanıcı adı gibi yapılandırma bilgilerini tekrar tekrar girmek istemez.

Uygulama ayarlarının saklanması için en basit yol, bu ayarların bir metin dosyasında "ayar adı = değer" şeklinde satırlar oluşturacak şekilde kaydedilmesidir. Güncel işletim sistemleri kayıt defteri adı verilen diğer bir seçenek sunar: İşletim sisteminin kendisi de dahil olmak üzere, uygulamaların yapılandırması ile ilgili bilgilerin hiyerarşik bir düzen içerisinde saklanabileceği bir ortam sunar. Java dili her iki seçeneğin de kullanılabilmesi için hazır araçlar sunar. Ne var ki kayıt defterleri zaman içerisinde aşırı büyüyerek hem uygulamanın hem de işletim sisteminin başarımını azaltabilir. Üstelik çoğu uygulama için ayarların metin dosyasında saklanması yeterli olacaktır. Bu bölümde **`java.util.Properties`** sınıfı kısaca tanıtılacaktır.

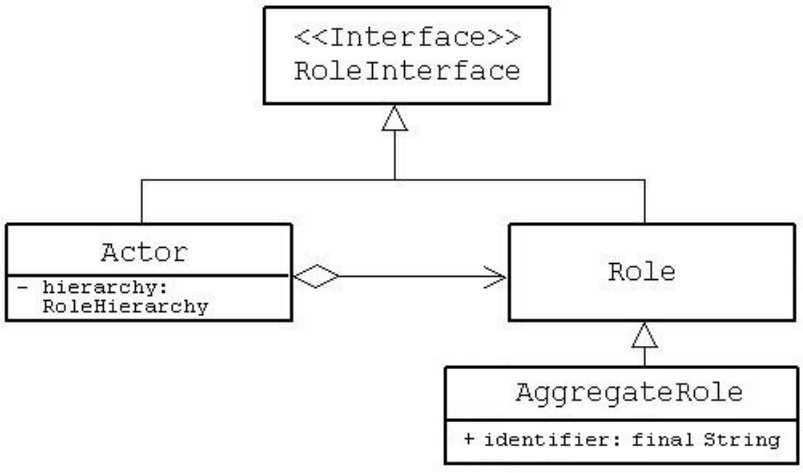
Metin dosyası kullanılarak uygulama ayarlarının (ayar adı,değer) ikilileri şeklinde işlenmesi için **`java.util.Properties`** sınıfı programcıya kolay ve esnek bir yol sunar. **`Properties`** sınıfı aslında **`java.util.Hashtable`** sınıfından kalıtım ile türetilmiştir ancak sağladığı yeni metotlarla programcıya amacına uygun işlevler sunar. Bir ayar ikilisi, **`public Object setProperty(String key, String value)`** metodu ile belleğe alınır. Bir ayar adı ile ilişkili değer ise bellekten **`public String getProperty(String key)`** metodu ile elde edilebilir. Ayar tablosunu kalıcı ortama kaydetmek için **`public void store(OutputStream out, String comments)`** metodu, ayar tablosunu kalıcı ortamdan yüklemek için ise **`public void load (InputStream inStream)`** metodu kullanılır.

8. GELİŞTİRİLEN ROL MODELİ: JAWIRO

Tez çalışmasının amacının, dinamik sistemlerin daha etkin modellenebilmesi için Java dilini rol desteği ile genişletilen bir rol modeli hazırlamak olduğu ilk bölümde anlatılmıştı. Gerçeklenen rol modeline JAWIRO (Java with Roles) adı verilmiştir. Bu bölümde JAWIRO'nun kullanım ve gerçekleme detayları verilecektir. JAWIRO'nun program başarımına ek yük getirip getirmediğini araştırmak üzere detaylı başarım ölçümleri ve JAWIRO kullanılarak gerçeklenmiş örnek bir uygulamanın açıklanmasına da bu bölümde yer verilmiştir.

8.1 JAWIRO'nun Kullanıcı Arayüzü

JAWIRO'nun kullanıcı arayüzünü ve bazı önemli üyelerini gösteren UML şeması Şekil 8.1'de verilmiştir. Bir rol hiyerarşisinin kökü olabilecek gerçek dünya nesneleri **`Actor`** sınıfından türetilir. Rol nesneleri ise **`Role`** sınıfı ile modellenmiştir. Nitelikli rollerin modellenmesinde kullanılan **`AggregateRole`** sınıfı, **`Role`** sınıfından kalıtım yolu ile türetilmiştir. Hem rol hem de aktör nesnelerinin benzer davranışları olabileceği için, **`Actor`** ve **`Role`** sınıfları, **`RoleInterface`** adlı ortak bir arayüzü gerçekler. Tablo 8.1'de **`RoleInterface`**, Tablo 8.2'de **`Actor`** ve Tablo 8.3'te **`Role`** sınıflarının kullanıcı arayüzünü oluşturan komutlar kısa açıklamaları ile verilmiştir.



Şekil 8.1: JAWIRO'nun kullanıcı arayüzü ve bazı önemli üyeleri

Tablo 8.1: RoleInterface arayüzünün kullanıcıya yönelik metotları

Arayüz: `RoleInterface` **Metot İmzası**	**Üye Metotlar: `public`** **İşlevi**
`public boolean addRole( Role r )`	Bu nesneye yeni bir rol ekler.
`public Object as( Class roleClass )`	Rol geçişi komutu.
`public Object as( Class roleClass, String identifier )`	Nitelikli roller için rol geçişi komutu.
`public Object as( String className )`	Rol geçişi komutu.
`public Object as( String className, String identifier )`	Nitelikli roller için rol geçişi komutu.
`public Object bringLocalMember (String name)`	Bu nesneye ait ve adı bilinen üye değişkene erişim sağlar.
`public Object bringMember (String name)`	Ad ile üye değişken erişimi. Adı verilen üye, rol hiyerarşisindeki tüm nesnelerde aranır.
`public boolean canDelegate(Role r)`	r nesnesinin bu nesne ile aynı hiyerarşide bulunup bulunmadığını araştırır.
`public boolean canSwitch( Class roleClass )`	Bu nesnenin aranan tipte bir role sahip olup olmadığını tüm hiyerarşide araştırır.
`public boolean canSwitch( Class roleClass, String identifier )`	Bu nesnenin aranan tipte bir nitelikli role sahip olup olmadığını tüm hiyerarşide araştırır.
`public boolean canSwitch( String className )`	Bu nesnenin aranan tipte bir role sahip olup olmadığını tüm hiyerarşide araştırır.
`public boolean canSwitch( String className, String identifier )`	Bu nesnenin aranan tipte bir nitelikli role sahip olup olmadığını tüm hiyerarşide araştırır.
`public void dominateSearch ( boolean dominate )`	Rol hiyerarşisinin bu üyesini baskın olarak atar.
`public Object executeLocalMethod( String name, Object... parameters )`	Bu nesneye ait ve adı bilinen üye metoda erişim sağlar.
`public Object executeMethod ( String name, Object... parameters )`	Ad ile üye metot erişimi. Adı verilen üye, rol hiyerarşisindeki tüm nesnelerde aranır.
`public String getClassName ( )`	Bu nesnenin sınıfını gösteren bir karakter katarı döndürür.
`public int getDepth( )`	Nesnenin rol hiyerarşisindeki derinliği.
`public Object getSelf( )`	Vekalet veya iletim mekanizmasına göre mesajın ilk veya son alıcısını döndürür.
`public boolean isDominant()`	Bu nesnenin baskın olup olmadığını döndürür.

Nitelikli rollerin birbirlerinden ayırt edilmesini sağlayan niteleyici olarak bir karakter katarı olan **`public final AggregateRole.identifier`** kullanılmıştır. **`AggregateRole`** sınıfı bir varsayılan kurucu metoda sahip değildir; sahip olduğu tek kurucunun aldığı **`String`** parametresi ise **`identifier`** üyesine değer atamak için kullanılır. Bunun dışında **`Role`** ve **`AggregateRole`** sınıfı arasında fark bulunmayıp, aksi belirtilmediği takdirde **`Role`** sınıfı için anlatılan her şey **`AggregateRole`** sınıfı için de geçerlidir.

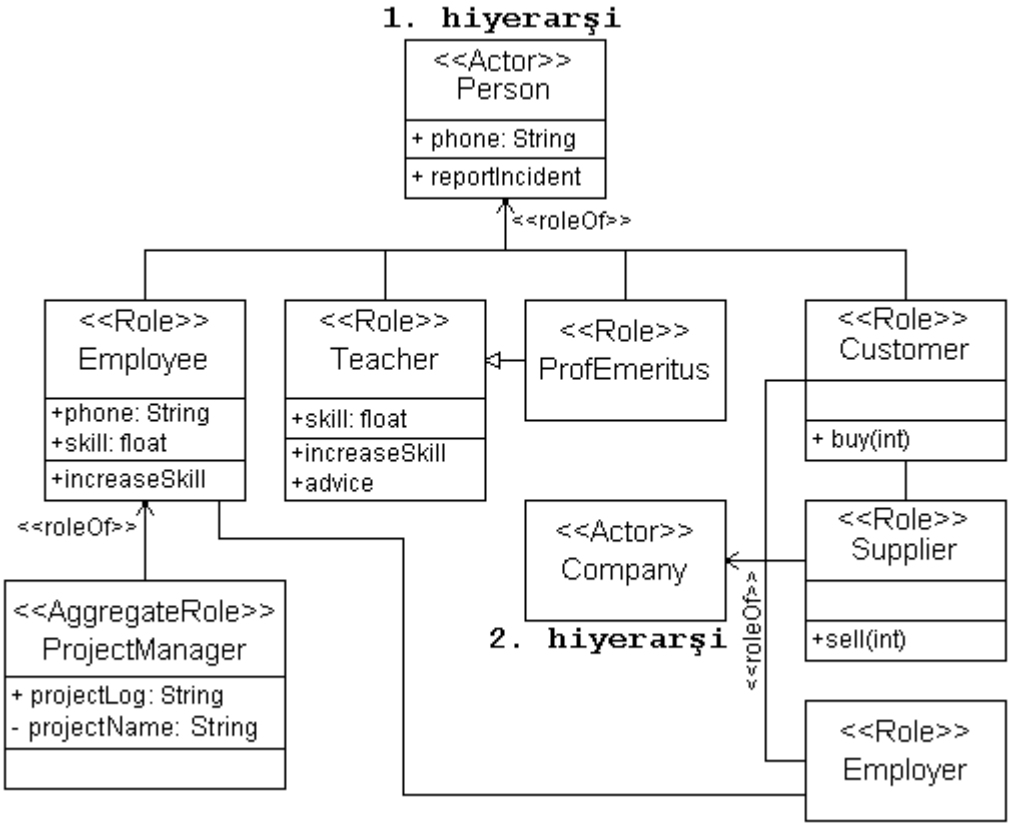
Tablo 8.2: Actor sınıfının kullanıcıya yönelik metotları

Sınıf: `Actor`	**Üye Metotlar: `public`**
Metot İmzası	**İşlevi**
`public Actor()`	Varsayılan kurucu.
`public void enableDelegation( boolean use )`	Vekalet mekanizmasının kullanımına izin vermek.
`public ConstraintStrategy getConstraintManager( )`	Kısıtlama yöneticisini döndürür.
`public boolean isDelegationEnabled( )`	Vekalet mekanizmasına izin verilip verilmediğini döndürür.
`public boolean isDelegationUsed( )`	Vekalet mekanizmasının kullanımda olup olmadığını döndürür.
`public void prePopulateReflectionTables ( boolean prePopulate )`	Nesnenin üyelerinin önceden hesaplanması davranışını belirler.
`public void removeAllDominance( )`	Hiyerarşinin tüm üyelerinin baskınlığını kaldırır.
`public boolean resumeRole( String className )`	Askıdaki belli bir tipteki rolün askıdan kaldırılması.
`public boolean resumeRole( String className, String identifier )`	Askıdaki belli bir tipteki nitelikli rolün askıdan kaldırılması.
`public void setConstraintManager( ConstraintStrategy cstr )`	Hiyerarşiye bir kısıtlama yöneticisi atar.
`public boolean suspendRole( String className )`	Belli bir tipteki rolün askıya alınması.
`public boolean suspendRole( String className, String identifier )`	Belli bir tipteki nitelikli rolün askıya alınması.
`public void useConsultation()`	Danışma mekanizmasının kullanımı.
`public void useDelegation( )`	Vekalet mekanizmasının kullanımı.

Tablo 8.3: Role sınıfının kullanıcıya yönelik metotları

Sınıf: `Role`	**Üye Metotlar: `public`**
Metot İmzası	**İşlevi**
`public Role()`	Varsayılan kurucu.
`public Actor getActor()`	Hiyerarşi kökünü döndürür.
`public Object playedBy( )`	Rolün sahibini döndürür.
`public boolean resign( )`	Bu rolün terk edilmesi.
`public boolean resume( )`	Bu rolün askıdan geri alınması.
`public boolean suspend( )`	Bu rolün askıya alınması.
`public boolean transfer( RoleInterface newOwner )`	Bu rolün alt rolleri ile birlikte başka bir sahibe aktarılması.

`RoleHierarchy` sınıfı rol modelinin omurgasını oluşturur ve her **`Actor`** örneğinin bu türden **`hierarchy`** adlı bir üyesi bulunur. Bu üye tez metninde *hiyerarşi yöneticisi* olarak adlandırılacaktır. **`RoleHierarchy`** sınıfı kullanıcı arayüzünün bir parçası olmayıp metotlarına sadece **`jawiro`** paketinden erişilebilir. **`RoleHierarchy`** sınıfı rol hiyerarşisini oluşturan ağaç yapısının düğümlerinde dolaşarak kullanıcı arayüzünü oluşturan metotların tüm gereksinimlerini karşılar.



Şekil 8.2: Örneklerde kullanılan iki rol hiyerarşisi

Tablo 8.1, Tablo 8.2 ve Tablo 8.3'te sırasıyla **`RoleInterface`** arayüzünün, **`Actor`** sınıfının ve **`Role`** sınıfının kullanıcı arayüzleri kısa açıklamalarla verilmiştir. Adları verilen tüm bu metotlar, rollerin temel ve ileri düzey özelliklerinin JAWIRO ile kullanımı örneklerle anlatılırken daha detaylı incelenecektir. Örneklerde kullanılan iki rol hiyerarşisine ait UML şeması ise Şekil 8.2'de görülebilir.

8.1.1 JAWIRO ile Rollerin Temel Özelliklerinin Kullanımı

Bu bölümde rollerin Bölüm 5.2'de yapılan gruplamaya göre temel özelliklerinin JAWIRO ile nasıl kullanılabileceği örneklerle anlatılacaktır.

8.1.1.1 Rol Kazanımı ve Terki ile Rol Hiyerarşilerinin Kurulması

Hareketli sistemlerde nesne düzeyinde özelleştirmenin daha doğru bir yaklaşım olduğundan bahsedilmişti. Bir gerçek dünya varlığı, zaman içerisinde çeşitli roller kazanarak ve terk ederek aynı tipten diğer nesnelere göre farklı biçimde uzmanlaşır. Bu durumda bir gerçek dünya nesnesi, sahip olduğu rollerin tümü ile tanımlanır. Bir rol hiyerarşisi, tüm üyeleri ile birlikte bir gerçek dünya nesnesini simgeler.

```
package test_destek;
import jawiro.*;
class Person extends Actor {
  String name, phone;
  public Person( String n, String p )
    { name = n; phone = p; }
}
class Teacher extends Role {
  String course;
  public Teacher( String c )
  { course = c; }
}
class ProjectManager extends AggregateRole {
  String projectName;
  public ProjectManager(String id) {
    super(id)
    projectName = id;
  }
}
Person peTom;
Teacher teTom;
peTom = new Person("Thomas Anderson", "843-663");
teTom = new Teacher("Physics");
peTom.addRole(teTom);
teTom.introduce(); //verdiği dersi tanıtır
teTom.resign(); //Tom emekli oldu
```

Şekil 8.3: Rol hiyerarşisi kurulması işlemleri

Bir rol hiyerarşisini kurmak için, kökü oluşturacak **`Actor`** sınıfından kalıtımla türetilmiş bir tipten bir nesneye ve **`Role`** sınıfından türetilmiş tiplerden gerekli rol nesnelerine gereksinim vardır. Tüm nesneler ister önceden, ister gerektiği anda oluşturulabilir. Nesneler oluşturulduktan sonra, modellenecek hiyerarşinin gerektirdiği şekilde rol ekleme işlemleri **`public boolean RoleInterface.addRole(Role aNewRole)`** metodu ile yapılır. Gereksinim kalmayan roller ise **`public boolean Role.resign()`** komutu ile terk edilebilir. **`addRole`** ve **`resign`** komutları başarılı olurlarsa **`true`**, aksi halde **`false`** değer döndürür. Bu komutların başarılı olamayacağı durumlar Bölüm 8.1.2.3'te anlatılmıştır. Şekil 8.3'te bu bölümde anlatılan işlemleri yapan örnek kod görülebilir.

Şekil 8.3'te verilen örnek kodda, rollerin yürütülmesi için, doğrudan rol nesnesinin **`introduce`** metodu çağrılmıştır. Rollerin bu şekilde doğrudan kullanımı modelleme açısından herhangi bir avantaj sağlamamaktadır, bu yüzden rollerin ilerideki bölümlerde incelenecek olan rol geçişi (Bölüm 8.1.1.3) ve ad ile üye erişimi (Bölüm 8.1.2.4) özellikleri aracılığıyla kullanılması tercih edilmelidir.

8.1.1.2 Nitelikli Roller

Bir rol hiyerarşisinde birden fazla örneği bulunabilecek nitelikli rol sınıflarını kodlarken, hiç parametre almayan varsayılan kurucu metot kullanılamaz. Bu tip rollerin birbirinden ayırt edilmesine yarayan niteleyiciyi atayacak şekilde **`AggregateRole(String identifier)`** kurucusunun çalışmasını sağlamak zorunludur. Ayrıca belirli bir niteleyiciye sahip bir nitelikli role ulaşabilmek için kullanıcı arayüzünün gerekli komutlarının ek bir **`String`** parametresine sahip biçimleri tanımlanmıştır. Bu detaylar dışında nitelikli rollerin kullanımı rol sınıfının kullanımı ile aynıdır. Şekil 8.4'te bir nitelikli rol gerçeklemesi görülebilir.

Aynı türdeki nitelikli rol örneklerinin niteleyicileri birbirlerinden farklı olmalıdır. Gerekli olan bu tekilliği sağlamaya yönelik önlemlerin alınması kullanıcıya bırakılmıştır. Tekillik söz konusu olmadığında Bölüm 8.1.1.3'te anlatılacak olan işlemler, aynı niteleyiciye sahip ve aynı türden olan nitelikli rollerden ilk olarak kazanılana erişim sağlayacaktır.

```
class ProjectManager extends AggregateRole {
  String projectName;
  public ProjectManager(String name,String id) {
    super(id); projectName = name;
  }
}
//Diğer rol sınıfları
Person peTom;
ProjectManager pmAI, pmVR;
peTom = new Person("Thomas Anderson", "843-663");
pmAI = new ProjectManager("Artificial Intelligence","AI");
pmVR = new ProjectManager("Virtual Reality","VR");
peTom.addRole(pmAI); //Tom yapay zeka projesini yönetiyor.
peTom.addRole(pmVR); //Tom sanal gerçeklik projesini de aldı.
```

Şekil 8.4: Rol hiyerarşisi kurulması işlemleri

8.1.1.3 Rol Varlığı Kontrolü ve Rol Geçişi

Bir gerçek dünya varlığının, sahip olduğu rollerin tümü ile tanımlandığı anlatılmıştı. Bu varlığın rollerinden birini gösteren bir işaretçi yardımıyla, gereksinim duyulan bir başka rolüne işaretçi elde edilerek, gerekli rolü kullanılabilir. Bu işlem rol geçişi olarak adlandırılır.

JAWIRO'da **`public Object RoleInterface.as`** metodu rol geçiş komutunu oluşturur ve birden fazla farklı şekli vardır. Bir hiyerarşideki farklı tipten bir kök ve bir veya daha fazla rol ile eş biçimli olarak çalışabilmek için, **`as`** metodu **`Object`** türünde işaretçi döndürür. Bu işaretçiyi kullanmadan önce uygun rol veya kök tipine dönüştürmek gerekecektir.

Rol tabanlı programlamada derleyici çalışma anında bir rol hiyerarşisinde hangi rollerin olduğunu bilemeyecektir. Bu nedenle bir rol geçişi işleminden önce, bu hiyerarşide aranan rolün olup olmadığı sorgulanabilmelidir. JAWIRO'da rol varlığı kontrolü için **`public boolean RoleInterface.canSwitch`** metodu kullanılır ve birden fazla farklı şekli vardır. Eğer istenen rol bulunursa **`true`**, aksi halde **`false`** değeri döndürülür.

Rol varlığı kontrolünün en basit şekli **`canSwitch(String className)`**, rol geçişinin en basit şekli ise **`as(String className)`** şeklindedir. İki komutta da istenen rolün tipi bir karakter katarı ile anlatılır. Tip adı verirken paketin adını da

içeren tam sınıf adı kullanılmalıdır. Şekil 8.5'te rol varlığı kontrolü ve rol geçişi örneklenmiştir. Örnekte bir insanın kişi rolünden öğretmen rolüne geçiş görülebilir.

```
package test_destek;
//Gerekli sınıf tanımları buraya gelecek
Person peTom;
Teacher teTom;
//Hiyerarşiyi kuran komutlar buraya gelecek
if( peTom.canSwitch("test_destek.Teacher") )
    ((Teacher)peTom.as("test_destek.Teacher")).introduce();
```

Şekil 8.5: Rol varlığı kontrolü ve rol geçişi

Rol varlığı kontrolü ve rol geçişinde tip adı yerine Java'nın yansıtma yetenekleri sayesinde elde edilebilecek ve tipi simgeleyen bir **`Class`** nesnesi de kullanılabilir. **`canSwitch(Class roleClass)`** ve **`as(Class roleClass)`** metotları bu amaçla tanımlanmıştır ve kullanımları Şekil 8.5'te gösterilen biçimlerininki ile tamamen aynıdır.

Tip adı kullanılarak rol varlığı kontrolü ve rol geçişi komutlarının **`canSwitch (String className, String identifier)`** ve **`as( String className, String identifier )`** biçimleri nitelikli roller için kullanılır. Tip adı yerine tip işaretçisi kullanılması gerekiyorsa **`canSwitch(Class roleClass, String identifier)`** ve **`as(Class roleClass, String identifier)`** metotlarından yararlanılır. Bu komutlar da Şekil 8.5'teki gibi kullanılır.

Eğer varlığı kontrol edilmek istenen rol nesnesine halihazırda bir işaretçi varsa **`public boolean RoleInterface.canDelegate(Role r)`** metodu diğer bir seçenek sunar. Bu durumda rol geçişi yapmak gereksizdir; doğrudan rol nesnesine olan işaretçi kullanılabilir. Şekil 8.6'te bu komutlar örneklenmiştir.

```
package test_destek;
//Gerekli sınıf tanımları buraya gelecek
Person peTom;
Teacher teTom;
//Hiyerarşiyi kuran komutlar buraya gelecek
if( peTom.canDelegate("teTom") )
    teTom.introduce();
```

Şekil 8.6: Rol varlığı kontrolü ve kullanımının diğer yolu.

Bir rol varlığı kontrolünü bir rol geçişinin takip etmesi büyük bir olasılıktır. Bu nedenle JAWIRO'da rol geçişi için bir iyileştirme mekanizması kurulmuştur: Son

kontrol edilen role ait işaretçi bir değişkende saklanır. Takip eden geçiş istekleri saklanan rol için yapılırsa; rol hiyerarşide yeni baştan aranmayıp, saklanan işaretçi doğrudan geri döndürülür. Bu iyileştirme tezin özgün katkılarından biridir.

8.1.1.4 Ebeveynlere Erişim

Her rol nesnesi, bulunduğu hiyerarşinin kökünden ve kendisini doğrudan oynayan üst rolünden haberdardır. Böylece her rol nesnesinin, sahip olduğu rollerin tümü ile tanımlanan bir gerçek dünya varlığının bir parçası olduğu gerçeği korunur. **`public Actor Role.getActor()`** metodu bir rolün ait olduğu hiyerarşinin köküne, **`public Object Role.playedBy()`** metodu ise bu rolün üst rolüne erişim sağlar. Bir rol askıda iken **`playedBy`** metodu sahibi de askıda bulunabileceğinden geriye **`null`** değeri döndürür ancak **`getActor`** metodu hiyerarşi kökünün askıya alınması söz konusu olmadığından düzgün çalışmasını sürdürür.

8.1.1.5 Sınıf ve Nesne Düzeyi Kalıtımın Birlikte Kullanımı

Bölüm 5.1'de sınıf ve nesne düzeyinde kalıtımın birbirinden farklı ancak birbirini bütünleyen iki ilişki olduğu anlatılmıştı. JAWIRO'da önceden tanımlanmış bir rol tipinden yeni rol tipleri sınıf düzeyinde kalıtım yolu ile tanımlanabilir. Ardından bu yeni rol tiplerinden örnekler çeşitli rol hiyerarşilerinin bir parçası olarak nesne düzeyi kalıtım ilişkisi içinde bulunabilir. Şekil 8.7'de verilen **`Teacher`** ve **`ProfEmeritus`** sınıfları sınıf ve nesne düzeyinde kalıtımın birlikte kullanımına örnek oluşturur.

```
class Teacher extends Role {
  String course;
  public Teacher( String c ) { course = c; }
}
class ProfEmeritus extends Teacher {
  public ProfEmeritus( Teacher t ) {
    super( t.course );
  }
}
Person peTom; Teacher teTom; ProfEmeritus preTom;
peTom = new Person("Thomas Anderson", "843-663");
teTom = new Teacher("Physics");
peTom.addRole(teTom);
teTom.resign(); //Tom emekli oldu ...
preTom = new ProfEmeritus(teTom);
peTom.addRole(preTom); /*... fakat emekli profesör olarak
ders vermeye devam ediyor. */
```

Şekil 8.7: Sınıf ve nesne düzeyinde kalıtımın birlikte kullanımı

8.1.2 JAWIRO ile Rollerin İleri Düzey Özelliklerinin Kullanımı

Bu bölümde rollerin Bölüm 5.2'de yapılan gruplamaya göre ileri düzey özelliklerinin JAWIRO ile nasıl kullanılabileceği örneklerle anlatılacaktır.

8.1.2.1 Rollerin Askıya Alınması ve Askıdan İndirilmesi

Geçici bir süre ihtiyaç duyulmayacak roller askıya alınabilir ve gerek duyulduğunda askıdan indirilerek durum bilgisinde kayıp olmadan tekrar kullanılabilir. Tez çalışmasının özgün katkılarından biri olan bu özelliğin, sadece rolün terki ve yeniden kazanımı ile gerçeklenmesi zor olacaktır. Bu zorluk hem terk edilen rolün hem de onun alt rollerinin durum bilgisinin saklanması ve yeniden kazanım öncesinde bu bilgilerin geri yüklenmesi gereğinden kaynaklanmaktadır.

Bir rolü askıya almak için o rol nesnesinin **`public boolean suspend()`** metodu çalıştırılır. Askıya alınmış bir rol ve bunun alt rolleri, JAWIRO kullanıcı arayüzünü oluşturan komutlarla kullanılamaz. Bu role tekrar gereksinim duyulduğu zaman aynı nesnenin **`public boolean resume()`** metodu çağrılır. Bir rolün askıda olup olmadığı ise **`public boolean Role.isSuspended()`** metodu ile öğrenilebilir. Bu biçimde kullanım Şekil 8.8'de örneklenmiştir.

```
Person peTom;
Teacher teTom;
peTom = new Person("Thomas Anderson","843-663");
teTom = new Teacher("Physics");
peTom.addRole(teTom);
teTom.suspend();
if( peTom.canSwitch("test_destek.Teacher") ) /* false döner */
    System.out.println("This is not supposed to happpen!");
teTom.resume();
if( peTom.canDelegate(teTom) )
    teTom.introduce();
```

Şekil 8.8: Rollerin askıya alınması ve askıdan indirilmesinin ilk yolu

Diğer bir seçenek olarak roller hiyerarşi kökünün sayesinde askıya alınabilir veya askıdan indirilebilir. Hiyerarşide bulunan belli bir rol tipi **`public boolean Actor.suspendRole(String className)`** metodu ile askıya alınabilir ve **`public boolean Actor.resumeRole(String className)`** metodu ile askıdan indirilebilir. Nitelikli roller için de bu metodların **`suspendRole(String`**

`className,String identifier)` ve **`resumeRole(String className, String identifier)`** biçimleri mevcuttur.

`suspendRole` ve **`resumeRole`** komutları, **`Role`** sınıfı yerine **`Actor`** sınıfına ait oldukları için, elimizde istenen rol nesnesine bir işaretçi olmadan kullanılabilirler. Kurulmuş bir rol hiyerarşisinde askıya alınmış bir rol bulunurken bu hiyerarşinin Bölüm 8.1.2.8'de anlatılacağı gibi kalıcı ortama kaydedildiğini ele alalım. Uygulama çalıştığı sürece, programcının elinde bir hiyerarşideki çeşitli rol nesnelerini gösteren işaretçiler olabilir. Bu işaretçiler arasında askıya alınan rolleri gösterenler de olabilir. Bu sayede ilgili rol nesnesinin **`resume`** komutunu kullanarak, askıdaki bir rol askıdan indirilebilir. Fakat uygulama sonlandırıldığında söz konusu işaretçiler de yok olacaktır. Söz konusu hiyerarşi kalıcı ortamdan geri yüklendiği zaman, hiyerarşi kalıcı ortama kayıt edildiğinde askıda olan roller askıda kalmaya devam edecektir. Geri yüklemenin ardından bu rolleri askıdan indirmek istenildiğinde, için ilgili rol nesnelerinin **`resume`** komutunu çalıştırmak gerekecektir. Halbuki uygulama daha önce sonlandırıldığı için, programcının elinde istenen rolü gösteren bir işaretçi kalmamıştır. Gerekli işaretçi rol geçişi ile de elde edilemez, çünkü askıdaki bir role geçiş yapılması bir aykırı durum olduğu için yasaklanmıştır. İşte bu durumda rol nesnesini gösteren bir işaretçi gerektiren **`suspend`** ve **`resume`** komutları yetersiz kalacağından, **`suspendRole`** ve **`resumeRole`** komutları kullanılmalıdır.

Bu bölümde anlatılan tüm komutlar başarılı oldukları takdirde **`true`**, aksi halde **`false`** değeri döndürür. Bu komutların başarılı olamayacağı durumlar Bölüm 8.1.2.3'te anlatılmıştır.

8.1.2.2 Rol Aktarımı

Gerçek dünya sistemlerinde bir varlık üstündeki sorumluluklardan birini bir başka varlığa devredebilir. Tez çalışmasının özgün katkılarından biri olan rol aktarımı yeteneği bu gereksinimi modeller. **`public boolean Role.transfer( RoleInterface newOwner)`** metodu ile bir rol alt rollerini kaybetmeden **`newOwner`** parametresi ile gösterilen başka bir sahibe aktarılabilir. Rol aktarımı Şekil 8.9'da örneklenmiştir.

```
Person peTom, peGordon; ProjectManager pmAI;
peTom = new Person("Thomas Anderson","843-663");
peGordon = new Person("Gordon Freeman","712-257");
pmAI = new ProjectManager("Artificial Intelligence","AI");
peTom.addRole(pmAI); //Tom projeyi önce üstlenir ama...
pmAI.transfer(peGordon);//...sonra yetersiz kalınca devreder.
```

Şekil 8.9: Rol aktarımı

8.1.2.3 Aykırı Rol İlişkilerinin Önlenmesi

Bir sistem hangi yolla ve ne kadar iyi modellenirse modellensin, uzun süren kodlama aşamasında veya daha uzun süren kullanım aşamasında çeşitli ilişkilendirme hatalarının yapılması olasılığı her zaman bulunur. Olası ilişkilendirme hataları, olağan dışı ilişkiler ve izin verilmeyen ilişkiler şeklinde iki grupta incelenebilir. JAWIRO her iki hata türünü de önleyebilecek mekanizmalara sahiptir.

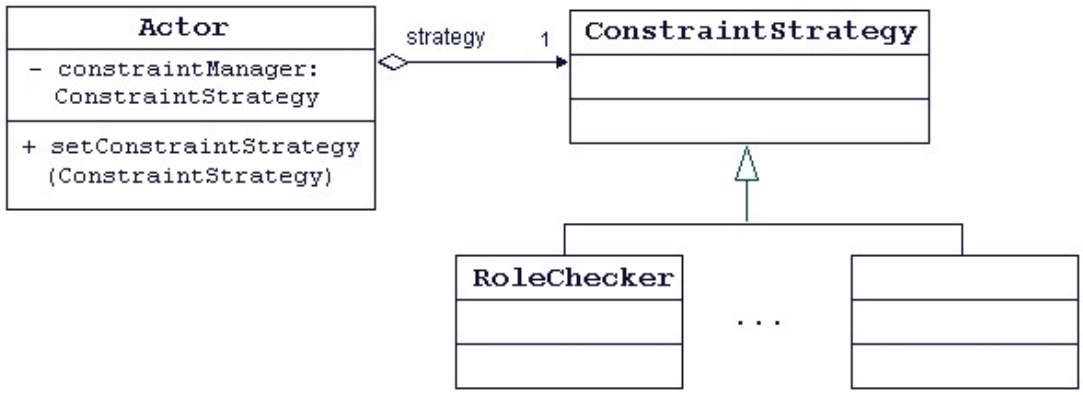
Olağan dışı ilişkiler, rol modelinin temel kavramlarına zıt düşen ilişkilerdir. Bu tür ilişkiler bütün gerçek dünya sistemlerine de ters düşer. Dolayısıyla olağan dışı ilişkileri önleyen şu kurallar rol modeli içine değiştirilemez biçimde kodlanmıştır:

- Bir rol örneği halihazırda bulunduğu bir hiyerarşiye eklenmez.
- Bir rol örneği aynı anda yalnız bir hiyerarşiye dahil olabilir.
- Askıya alınmış bir rol JAWIRO uygulama programlama arabiriminin komutları ile kullanılamaz. Örneğin başka bir sahibe aktarılamaz.

Askıya alınan role rol modelinin komutlarıyla ulaşılamaması bir gerekliliktir. Bir sorumluluğun askıya alınmasının nedeni, gerçek dünyanın o anki koşullarında o sorumluluğun kullanılması için bir engelin ortaya çıkmasıdır. Gerçek dünyada engellenen bir sorumluluğun uygulama programında serbestçe çalıştırılabilmesi ise bir tutarsızlık ortaya çıkartacaktır. Ancak askıda olan bir rolün durum bilgisine erişmenin kullanışlı olacağı haller ortaya çıkabilir. Bu durumda ilgili rol nesnesinin **`public void Role.setReadableWhileSuspended(boolean isReadable)`** metodu **`true`** parametresi ile çağırılarak bu role askıda olduğu süre içerisinde bile geçiş yapılması sağlanabilir. Bu çalışma biçiminde bile askıda olan rolün Bölüm 8.1.2.3'te anlatılan ad ile üye erişimi komutlarına ulaşılamaz ve Bölüm 8.1.1.3'te anlatılan rol varlığı kontrolü komutları da geriye **`false`** değeri döndürmeyi sürdürür.

Varsayılan çalışma biçimi ise askıda olan rollere geçiş yapılamamasıdır. Söz konusu kuralı rahatlık için esnetmek yerine tutarlılık için bu kuralın sürdürülmesi seçildiği halde ise, rol askıda iken ulaşılması gerekli bilgiler geçici değişkenlerde programcı tarafından saklanılabilir. Bölüm 8.4.4.1'de böylesi bir durum örneklenmiştir. **`public boolean Role.isReadableWhileSuspended()`** metodu ile de bir role askıdayken geçiş yapılıp yapılamayacağı öğrenilebilir.

İzin verilmeyen ilişkilendirmeler ise, modellenen sistemin kurallarına göre belirlenir. Her sistem için farklı kurallar geçerli olabileceğine göre, bir rol modeline programcı tarafından çeşitli kurallar esnek bir biçimde tanıtılabilmeli ve rol modeli de bu kurallara uyulmasını sağlamalıdır. JAWIRO, strateji tasarım dokusuna göre [22] gerçeklenen bir mekanizma ile kullanıcıya bu olanağı sağlar. Söz konusu mekanizma Şekil 8.10'da görülebilir.



Şekil 8.10: Kontrol mekanizmasına ait UML şeması

Kullanıcının **`public void Actor.setConstraintManager(ConstraintStrategy cstr)`** metodu ile atayacağı bir kısıtlayıcı nesne her **`addRole`**, **`resign`**, **`suspend`** ve **`resume`** komutu öncesinde JAWIRO tarafından çalıştırılacak ve kısıtlayıcı nesneye işlemi onaylama şansı verilecektir. Kısıtlayıcı sınıfların gerçeklemesi gereken arayüz Şekil 8.11'de verilmiştir. Şekil 8.11'den anlaşılabileceği gibi, JAWIRO kısıtlayıcı nesneye onaylayacağı işlem ile ilgili olan alt ve üst rolün tipini bildirir. Kısıtlayıcı da kendisine kodlanmış kurallara göre işlemi onaylar veya engeller. Denetlenen işlemin adı ile **`ConstraintStrategy`** arayüzünün denetlemeyi yapan metodunun adı birbirini çağrıştıracak şekilde verilmiştir. Örneğin askıya alma komutları olan **`Role.suspend`** ve **`Actor.suspendRole`** komutlarını **`ConstraintStrategy.approveSuspend`** metodu denetler. **`ConstraintStrategy.setActor(Actor`**

anActor) metodu ise **Actor.setConstraintManager** tarafından çağrılır ve hiyerarşinin kökü **anActor** parametresi ile rol kısıtlayıcısına bildirilir. Böylelikle kısıtlayıcı nesne hiyerarşide rol varlığı kontrolü veya askıya alma gibi eylemler yapma olanağı bulur. Hiyerarşinin kısıtlama yöneticisini elde etmek için ise **public ConstraintStrategy Actor.getConstraintManager** metodu kullanılır.

```
public interface ConstraintStrategy {
  public boolean approveAddRole(String parentClassName, String
     childClassName);
  public boolean approveResign(String parentClassName, String
     childClassName);
  public boolean approveSuspend(String parentClassName, String
     childClassName);
  public boolean approveResume(String parentClassName, String
     childClassName);
  public void setActor( Actor anActor ); }
```

Şekil 8.11: Kısıtlayıcı sınıfların gerçeklemesi gereken arayüz

Programcı rol modeline eklenen **RoleChecker** adlı hazır kısıtlayıcı sınıfı kullanabilir ya da isterse kendisi **ConstraintStrategy** arayüzünü gerçekleyen yeni bir kısıtlayıcı sınıf hazırlayabilir. Aykırı rol ilişkilerinin önlenmesi ile ilgili bilgileri örneklemek için Şekil 8.12'de doktor ve hasta rollerinin aynı anda bulunmasını ve eczacı rolünü kazanmayı engelleyen bir kısıtlama yöneticisi gerçeklemesi verilmiştir.

```
class CheckerExample implements jawiro.ConstraintStrategy {
  private Actor myActor;
  public void setActor(Actor anActor)
  { myActor = anActor; }
  public boolean approveAddRole(String parentClassName, String
     childClassName) {
    if(childClassName.equals("demoApp.RolePharmacist"))
      return false;
    if(childClassName.equals("demoApp.RoleDoctor"))
      if(anActor.canSwitch("demoApp.RolePatient"))
        return false;
    if(childClassName.equals("demoApp.RolePatient "))
      if(anActor.canSwitch("demoApp.RoleDoctor "))
        return false;
    return true; }
  public boolean approveResign(String parentClassName, String
     childClassName) { return true; }
  public boolean approveSuspend(String parentClassName, String
     childClassName) { return true; }
  public boolean approveResume(String parentClassName, String
     childClassName) { return true; } }
```

Şekil 8.12: Bir kısıtlayıcı sınıf örneği

Aykırı rol ilişkilerinin önlenmesinde kullanılan kısıtlama yöneticisi, hiyerarşi köküne erişebilmesi sayesinde, onaylayabileceği işlemlerden önce çeşitli eylemlerin otomatik olarak çalıştırılması için de kullanılabilir. Dolayısıyla kısıtlama yöneticisi sadece bir işlemi onaylamak veya reddetmek için değil, onaylayabileceği işlemlerin öncesinde modellenen sistemin gereksinimleri ve kuralları doğrultusunda eylemlerde bulunmak için de kullanılabilir. Ancak bu eylemler olağan dışı eylemlerin değişmez kuralları veya izin verilmeyen eylemlerin herhangi bir kısıtlama yöneticisinde tanımlanan kuralları ile çakışmamalıdır. Böylesi bir duruma Bölüm 8.4.4.3'ün sonunda yer verilmiştir.

8.1.2.4 Ad ile Üye Erişimi

JAWIRO ile bir rol hiyerarşisindeki elemanların herhangi birinin yalnızca adı bilinen üye değişkeni veya metoduna, bu üyenin sahibini belirtmeden erişilebilir. Bir başka deyişle; bir gerçek dünya varlığına istenilen bir iş, o işi hangi rolün üstlendiğini bilmeye gerek kalmadan yaptırılabilir. Örnek vermek gerekirse, bir rol hiyerarşisindeki herhangi bir A nesnesinin herhangi bir B üye değişkeni veya metoduna, B üyesinin adını o hiyerarşideki herhangi bir C nesnesine vererek ve B nesnesine hiçbir işaretçi kullanmadan erişilebilir. Tez çalışmasının özgün katkılarından bir olan ad ile üye erişimi yeteneğinin gerçeklenmesinde Java'nın yansıtma yetenekleri kullanılmıştır.

Üye değişkenlere yalnız ad ile erişim için **`public Object RoleInterface.bringMember(String name)`** metodu kullanılır. Bu komutun ardından istenilen üye değişken rol hiyerarşisindeki tüm nesnelerde aranır ve bulunduğu takdirde geri döndürülür. Aranan üye değişken bulunamazsa geriye **`null`** değeri döndürülür. İlkel tiplere erişim ancak ilgili paketleyici ile mümkündür; örneğin **`int`** ilkeli için **`Integer`** sınıfı kullanılır. Java dilinin yansıtma yeteneklerinden kaynaklanan kısıtlamalar nedeniyle, üye değişkenlere işaretçi ile (call-by-reference) değil değer ile (call-by-value) erişilir.

Üye metotlara yalnız ad ile erişim için **`public Object RoleInterface.executeMethod(String name,Object... parameters)`** metodu kullanılır. Bu komut istenilen üye metodu rol hiyerarşisindeki tüm nesnelerde arar ve buluduğu takdirde çalıştırır. Aranan metodun geri döndürdüğü bir değer varsa o değeri

executeMethod metodu programcıya iletir. Üye metot erişiminde herhangi bir kısıtlama söz konusu değildir; metoda referans ile erişim sağlanır ve üye metot çalışınca ait olduğu nesnenin iç durumunu değiştirir. Şekil 8.13'te ad ile üye erişimi özelliği için bir kullanım örneği görülebilir. Öğretmen ve emekli profesör sınıfının bir acil durumu yetkililere bildirmeyi sağlayan **reportIncident** metodu olduğunu ele alalım. Bu metoda ulaşmak için öğretmen ve emekli profesör rollerinin varlığını ayrı ayrı test edip hangi rol bulunursa ona geçiş yaptıktan sonra istenilen metodu çalıştırmak yerine, tek bir komutla hiyerarşinin kökünden istenilen metodu çalıştırması istenilebilir.

```
Person p;
Incident anIncident = new Incident("Fire");
Authority anAuthority = new Authority("Fire Brigade");
p = new Person("Yunus Selçuk","212-2891990");
p.add_Teacher_or_ProfEmeritus_Role(); /* Öğretmen veya emekli
profesör rolü ekleyen bir metot */
p.executeMethod("reportIncident",anIncident,anAuthority);
```

Şekil 8.13: Ad ile üye erişimi örneği

Ad ile üye erişiminde istenilen üyenin sahibine bir işaretçi elde edilmesinin mümkün olduğu, ancak bu üyenin tipinin belli olmayacağı durumlar ortaya çıkabilir. Bölüm 8.1.2.6'da anlatılan nesne düzeyinde çoklu kalıtım böylesi bir durumdur. Bu durumda **public Object RoleInterface.bringLocalMember (String name)** komutu ile bir rol hiyerarşisinin bilinen bir katılımcısının istenilen üye değişkeninin değerine erişilebilir. **public Object executeLocalMethod(String name, Object... parameters)** komutu ise bir rol hiyerarşisinin bilinen bir katılımcısının istenilen üye metodunu çalıştırmaya yarar. **bringMember** ile **bringLocalMember** komutları arasındaki tek fark; öncekinde istenen üye tüm hiyerarşide aranırken sonrakinde ise hiyerarşinin sadece komutun çalıştırıldığı katılımcısının kontrol edilmesidir. Aynı fark **executeLocalMethod** ve **executeMethod** komutları arasında da geçerlidir.

Aynı hiyerarşinin birden fazla katılımcısı aynı adlı üyeye sahipse, en çok özelleşmiş (hiyerarşinin en derinindeki) katılımcının üyesine erişim sağlanacaktır. Bu davranış, Bölüm 8.1.2.5'te anlatılacağı şekilde, hiyerarşinin bazı üyelerini baskın kılarak değiştirilebilir. Bir nesnenin hiyerarşide bulunduğu derinlik **public int RoleInterface.getDepth( )** metodu ile öğrenilebilir.

JAWIRO'nun aranan üyeyi bulabilmesi için, rol hiyerarşisinin tüm katılımcılarının tüm üyeleri hakkında gerekli bilgilerin Java'nın yansıtma yetenekleri kullanılarak toplanması gerekir. Hem yansıtma işlemleri hem de tüm hiyerarşinin dolaşılması büyük sistemlerde önemli bir ek yük getirecektir. Bu gibi durumlarda gerekli bilgilerin mümkün olduğunca önceden hesaplanması uygun düşecektir. Öte yandan, ad ile üye erişimi özelliğine gereksinim olmayan durumlarda gerekli bilgilerin önceden hesaplanması boş yere başarımı düşürecektir. Bu nedenle **`public void Actor.prePopulateReflectionTables(boolean prePopulate)`** metodu ile bir rol hiyerarşisinde yansıtma bilgilerinin önceden elde edilmesi davranışı açılabilir veya kapatılabilir. Bu davranış açıldığı takdirde JAWIRO her rol ekleme işlemi sırasında yansıtma bilgilerinin hesaplanıp hesaplanmadığını kontrol eder ve eğer hesaplanmamışsa hesaplar. Aksi halde gerekli bilgiler ilk ad ile üye erişimi isteğinde hesaplanır. Bir rol hiyerarşisinin önceden hesaplama davranışının ne olduğu **`public boolean Actor.getReflectionPrePopulationMode()`** metodu ile öğrenilebilir. Önceden hesaplama davranışının varsayılan değeri kapalıdır. Bundan bağımsız olarak istenildiği anda **`public String RoleInterface.getClassName()`** metodu ile nesnenin tip adı öğrenilebilir.

JAWIRO bir rol hiyerarşisinin katılımcıları hakkındaki yansıtma bilgilerini hesaplarken metot imzalarını elde etmek için **`Class.getName`** metodunu kullanır. Bu metot ilkel veri tiplerini olduğu gibi ele alır. Buna karşılık metotları çalıştırmak için kullanılan **`Method.Invoke`** metodu, ilkel tipler yerine paketleyici sınıfların kullanılmasını gerektirir. Sonuç olarak JAWIRO aranan metot imzasını hesaplanan metot imzalarının arasında bulunamaz ve istenen üye metoda erişim sağlayamaz. Bu nedenle ad ile üye metod erişimi yeteneği kullanılarak çalıştırılacak metotlar parametre olarak ilkel veri tipleri içermemeli, bunun yerine ilgili paketleyici sınıftan parametreler içermelidir.

Özetle, ad ile üye erişiminin kullanım amaçları bir rol hiyerarşisindeki en çok evrimleşmiş üyeye erişim ve nesne düzeyi çoklu kalıtımın ortaya çıkarabileceği tipleme belirsizliğini çözmektir. Tipleme belirsizliği çözümü için kullanımda, erişilmek istenen üyenin tam olarak hiyerarşinin hangi katılımcısında bulunduğunun bilindiği durumlar olabilir. Bu durumlarda rol hiyerarşisinin tümünün derinlemesine

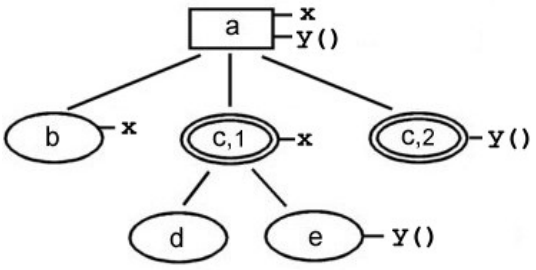
taranması yerine **`RoleInterface`** arayüzünde bulunan **`bringLocalMember`** ve **`executeLocalMethod`** metodlarının kullanılması daha doğru olacaktır.

8.1.2.5 Baskın Roller

JAWIRO'nun önceki bölümde anlatılan ad ile üye erişimi yeteneğinin en evrimleşmiş üyeyi getirme davranışı, bazı hiyerarşi üyelerinin baskın kılınması ile değiştirilebilir. Baskın roller kavramı da tez çalışmasının özgün katkılarından biridir.

Bir rol hiyerarşisindeki nesneler **`public void RoleInterface.dominateSearch(boolean dominate)`** metodu ile baskın kılınabilir. Bir **`Role`** nesnesi baskın kılındığında en evrimleşmiş üye yerine bu nesnenin üyesine erişilecektir. Aynı hiyerarşide birden fazla **`Role`** nesnesi baskınsa, bunlardan en evrimleşmiş olanının üyesine erişilecektir. Fakat bir **`Actor`** nesnesi baskınsa her zaman onun üyesine erişilir, baskın olsalar dahi diğer **`Role`** nesnelerine bakılmaz.

Baskınlık kurallarının örneklerle daha iyi anlaşılması için Şekil 8.14 incelenebilir. Şekil 8.14'te örnek bir rol hiyerarşisi verilmiş, hiyerarşideki bazı nesnelerin üyeleri işaretlenmiş ve çeşitli komutlar sonrasında sistemin nasıl davranacağı gösterilmiştir. Şekil 8.14'te hiyerarşi kökü dörtgen, roller elips ve nitelikli roller çift elips içinde niteleyicisi ile birlikte gösterilmiş; üyeler ise nesnelerin yanında gösterilmiştir. Hangi üyeye erişim sağlanılacağı ise çalıştırılan komutların ardından gösterilmiştir.




```
o=d.bringMember("x");//o=c1.x
b.dominateSearch(true);
o=c1.bringMember("x");//o=b.x
b.executeMethod("y",null);//e.y()
c2.dominateSearch(true);
b.executeMethod("y",null);//c2.y()
a.dominateSearch(true);
b.executeMethod("y",null);//a.y()
c2.executeLocalMethod("y",null);
//c2.y()
b.executeLocalMethod("y",null);
//HATA
```

Şekil 8.14: Bir rol hiyerarşisi üzerinde baskınlık mekanizmasının örneklenmesi

Bir rol hiyerarşisi katılımcısının baskın olup olmadığı **`public boolean RoleInterface.isDominant()`** metodu ile öğrenilebilir. Hiyerarşideki tüm

baskınlık işaretlerini tek adımda kaldırmak için ise hiyerarşi kökünün **`public void Actor.removeAllDominance()`** metodu kullanılır.

8.1.2.6 Nesne Düzeyinde Çoklu Kalıtım

Tez çalışmasının diğer bir özgün katkısı olarak nesne düzeyinde çoklu kalıtım desteklenmiştir. Bu sayede modellenen sistem gerektirdiği takdirde bir rol sınıfına ait nesneler farklı tiplerden sahipler tarafından oynanabilir. Bir satıcının hem kurumsal hem de bireysel müşterileri olan ve Şekil 8.2'de verilen örnek sistemde **`Customer`** rolü, hem **`Person`** hem de **`Company`** örnekleri tarafından oynanabilir.

Belli bir rol örneği aynı anda yalnız bir sahibe ait olabileceği için nesne düzeyinde çoklu kalıtım mantıksal bir belirsizlik ortaya çıkarmaz. Alt rolün üst role ait bir üyeye erişmesi gereken durumlarda ise Java tiplemesi açısından bir belirsizlik söz konusu olacaktır: Bir rol hiyerarşisinde hangi rollerin bulunacağı ancak çalışma anında belli olur. Derleme anında ortaya çıkacak tipleme belirsizliğini çözmek için üst rollerin aynı arayüzü gerçeklemesi veya erişilecek üyenin ayrı bir role taşınarak o rol üzerinden kullanılması sağlanabilir. Başka bir seçenek de, JAWIRO'nun Bölüm 8.1.2.4'te değinilen ad ile üye erişimi yeteneğinin kullanılmasıdır.

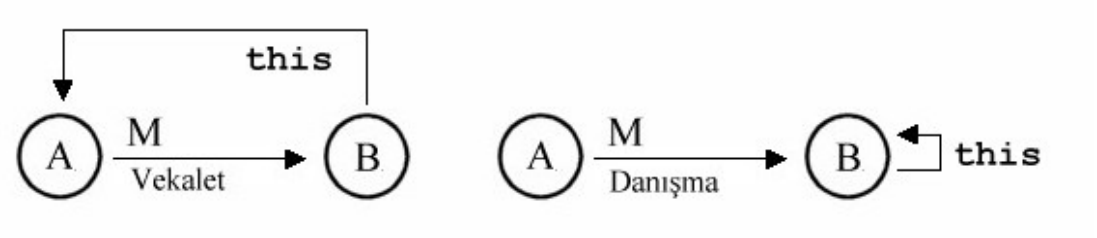
```
class Customer extends Role {
  Supplier supplier;
  public Customer(Supplier supplier )
  { this.supplier = supplier; }
  public void buy( Goods aGood, int amount ) {
    if( supplier.sell(aGood,amount) )
      getActor().executeLocalMethod( "increaseFunds",
        -myGood.getUnitPrice()*amount ); }}
class Supplier extends Role {
  public boolean sell( Goods aGood, int amount ) {
    if( satis_mumkun() ) {
      getActor().executeLocalMethod( "increaseFunds",
        +myGood.getUnitPrice()*amount );
      return true;
    } else return false; }}
class Person extends Actor { //sınıf kodunun bir kısmı
  private double funds;
  public void increaseFunds(Double incr) { funds += incr; } }
class Company extends Actor { //sınıf kodunun bir kısmı
  private double funds;
  public void increaseFunds(Double incr) { funds += incr; } }
```

Şekil 8.15: Nesne düzeyi çoklu kalıtım sonucu oluşan tipleme belirsizliğini çözmek amacıyla ad ile üye erişimi yeteneğinin kullanılması

Şekil 8.2'de verilen örnek sistemde şirketlerin ve bireylerin bütçeleri sırasıyla **`Company.funds`** ve **`Person.funds`** üyelerinde tutulmaktadır. **`Customer.buy`** metodu ile harekete geçen bir müşteri ile bir tedarikçi arasındaki alışverişin sonunda, ilgili varlıkların bütçelerinde değişiklik yapılmalıdır. Müşteri rolünün hem şirketler hem de bireyler tarafından oynanılabilmesi, müşteri rolünün hiyerarşi kökündeki bütçe üyesine erişmesi sırasında tipleme belirsizliği ortaya çıkartacaktır. Şekil 8.15'te ad ile üye erişimi sayesinde bu belirsizliğin nasıl üstesinden gelindiği görülebilir. Bu örnekte hiyerarşi kökünün bütçe üyesinin değerini değiştiren **`increaseFunds (Double incr)`** metoduna, hiyerarşi kökünün tipi bilinmeden erişilmektedir.

8.1.2.7 Danışma ve Vekalet Mekanizmalarının Birlikte Kullanımı

Bir A rolünden B rolüne geçiş yapıldığında, çalıştırılacak metodun ilk alıcısının saklanılması veya saklanılmaması tercih edilebilir. Danışma ve vekalet mekanizmaları birbirinden bu yolla, Şekil 8.16'da görüldüğü gibi ayrılır. Vekalet türü çalışmada rol geçişi nedeniyle M metot çağrısı A rolünden B rolüne iletildiğinde, **`this`** işaretçisi mesajın orijinal alıcısı olan A rol nesnesini gösterir. Bu mekanizmanın alternatifi olan danışma türü çalışmada **`this`** işaretçisi mesajın son alıcısı olan B nesnesini gösterir.



Şekil 8.16: Danışma ve vekalet mekanizmaları

Mevcut rol modelleri bu mekanizmalardan yalnız biri ile çalışırken tez çalışmasının diğer bir özgün katkısı olarak JAWIRO her iki mekanizmanın da kullanılmasına ve çalışma zamanı süresince istenilen anda istenilen hiyerarşi için istenilen mekanizmanın seçilmesine izin verir. Her **`Actor`** ve **`Role`** nesnesinin **`Object`** sınıfından **`self`** adlı bir üyesi bulunur. **`self`** üyesi her rol geçişi komutu o anda kullanılan mekanizmaya göre mesajın ilk veya son alıcısını tutar. **`public Object RoleInterface.getSelf( )`** metodu ile **`self`** üyesine erişilebilir.

JAWIRO'nun varsayılan çalışma şekli danışmadır. Bu çalışma şekli için herhangi bir kodlama yapılması gereksizdir, çünkü **`self`** üyesine gereksinim olmadan sadece **`this`** işaretçisi ile her zaman mesajın son alıcısına erişilebilir. Mesajın ilk alıcısını tutmak için ise ek işlemler yapılmalıdır. Bu işlemlerin sisteme ek yük getireceği düşünülerek, sadece vekalet mekanizmasının gerekebileceği anlarda ek işlerin yapılmasını sağlamak için, **`public void Actor.enableDelegation( boolean use)`** komutu eklenmiştir. Gerekli ek işlere bu komutla izin verdikten sonra programcı istediği anda istediği hiyerarşi için **`public void Actor. useDelegation()`** çağrısını kullanarak vekalet veya **`public void Actor. useConsultation()`** çağrısını kullanarak danışma türü çalışmaya geçebilir.

Belirli bir anda bir hiyerarşi için vekalet mekanizmasına izin verilip verilmediğini öğrenmek için **`public boolean Actor.isDelegationEnabled( )`** metodu, vekalet mekanizmasının kullanımda olup olmadığını öğrenmek için ise **`public boolean Actor.isDelegationUsed( )`** metodu kullanılır.

Vekalet mekanizmasına izin verilmesinin ve vekalet mekanizmasının kullanılmasının ayrı ayrı ek yükler getireceği düşünülmesine rağmen, rol modelinin gerçeklenmesi tamamlandıktan sonra yapılan ve Bölüm 8.3.3'te verilen başarım ölçümlerinde bu iki çalışma biçimi arasında fark edilebilir bir başarım farkı olmadığı görülmüştür.

Gerçek dünya sistemlerinde danışma ve vekalet mekanizmaları bir arada bulunabileceği için, bu iki mekanizma birbirini tamamlayan çalışma biçimleri olarak ele alınmalıdır. İki mekanizmanın da gerekliliğine örnek oluşturmak üzere, Şekil 8.2'de verilen birinci rol hiyerarşisinin bir kısmı Şekil 8.17'de tekrarlanmış ve iki olası rol geçişi gösterilmiştir.

Şekil 8.17: Örnek rol hiyerarşisi ve iki olası rol geçişi (i. ve ii.)

Şekil 8.17'de gösterilen iki olası rol geçişinden birincisi, **Employee** rolünden **ProfEmeritus** rolüne geçerek **advice** metodunun çalıştırılmasını simgeler. Her iki rolün de **skill** adlı bir üyesi bulunmaktadır. **Employee.skill** üyesi kişinin iş yeteneğini tutarken **ProfEmeritus.skill** üyesi kişinin akademik yeteneğini tutar. İki rolün de **increaseSkill** metodunun her çağrılışında, o rolün **skill** üyesinin değeri artar. **Employee** rolünden **ProfEmeritus** rolüne geçilip profesörün bir akademik tavsiye verilmesi sağlandığında, kişinin akademik yeteneğinin artmasını sağlamak için danışma mekanizmasının kullanılması gerekir. Aksi halde yanlış olarak **Employee.skill** üyesi artmış olacaktır.

Diğer yandan, **Person** sınıfına bir acil durumu yetkililere haber verme yeteneği kodlanmıştır ve bu iş arayan kişinin telefonunu yetkililere bildirmesini gerektirir. Eğer olay anında kişi ofiste ise, **Employee** rolünden **Person** rolüne geçiş yapmak gerekecektir. Bu durumda ev telefonu yerine iş telefonunun verilmesi için danışma yerine vekalet mekanizması kullanılmalıdır. Şekil 8.17'de gösterilen ikinci olası rol geçişi de bu şekildedir. Örnek sınıfların ilgili bölümlerinin gerçeklemesi ise Şekil 8.18'de, her iki rol geçişini sağlayan kod ise Şekil 8.19'da görülebilir.

```
public class Employee extends Role {
   double skill; String phone;
   public Employee( String p ) { phone = p; skill = 1.0;  }
   public void increaseSkill(Double d) { skill += d.doubleValue(); }
   public String givePhone( ) { return phone; }
}
public class ProfEmeritus extends Teacher {
   public double skill; Object[] params;
   public ProfEmeritus( Teacher t ) {
      super( t.course );
      skill = 1.0;   }
   public void advice( ) {
      //give academic advice
      params = new Object[2];
      params[0] = new Double(0.1);
      ((RoleInterface)self).executeMethod("increaseSkill",params);  }
   public void increaseSkill(Double d) {
      skill += delta.doubleValue(); } }
public class Person extends Actor {
   String name, phone;
   public Person( String n, String p )
   { name = n; phone = p;  }
   public void reportIncident( ) {
      //report the incident
      String currentPhone = (String) ((RoleInterface)self).
           ExecuteMethod ("givePhone",null);
      System.out.println("You can call me from" + currentPhone );}  }
```

Şekil 8.18: Örnek rol hiyerarşisindeki sınıfların kodu ve **self** üyesinin kullanımı

```
Person p; Employee e; Teacher t; ProfEmeritus pre;
p = new Person("Yunus Selçuk","212-2891990");
e = new Employee( "212-2853300" );
p.addRole( e );
pre = new ProfEmeritus( t );
p.addRole( pre );
p.enableDelegation( true );
p.useConsultation();
((ProfEmeritus)e.as("test.ProfEmeritus")).advice();
p.useDelegation();
((Person)e.as("test.Person")).reportIncident();
```

Şekil 8.19: Vekalet ve danışma mekanizmalarının birlikte kullanımına örnek

8.1.2.8 Kalıcılık

Kalıcılık desteği olmayan programlama dillerinde tanımlanan nesnelerin yaşam süreci, yazılan programın çalışma süreci ile sınırlıdır. Kalıcılık özelliğine sahip dillerde ise tanımlanan nesnelerin durum bilgisi program sonlanmadan önce veya programcının istediği anlarda kalıcı ortama kaydedilerek saklanır. Uygulama programı yeniden çalıştırıldığında veya programcının istediği anda nesnelerin en son durumu kalıcı ortamdan yüklenir ve nesneler yaşam sürecine kaldıkları yerden devam ederler.

JAWIRO'nun kalıcılık yeteneği, Java'nın serileştirme yeteneğini kullanır. Bu sayede rol hiyerarşileri istenildiği anda bütün olarak kalıcı ortama kaydedilebilir ve kalıcı ortamdan geri yüklenebilir. JAWIRO bu işlemleri güvenli bir biçimde yürütür, böylece yetkisi olmayan nesneler kendilerine ait olmayan hiyerarşilere erişemez.

Kalıcılık işlemleri, rol modeline eklenen **`PersistenceManager`** sınıfından türetilmiş nesneler üzerinden yapılır. Kalıcılık işlemlerinin ana sorumlusu olan bu sınıfın örneklerine *kalıcılık yöneticisi* adı verilmiştir. Rol hiyerarşileri, kalıcılık yöneticisi üzerinden kalıcı ortama kaydedilir ve kalıcı ortamdan geri yüklenir. Bu işlemler sırasında rol hiyerarşisini, hiyerarşinin kökü olan **`Actor`** örneği temsil eder.

Bir rol hiyerarşisi kalıcı ortama kaydedilmeden önce, bu hiyerarşinin kökü bir kalıcılık yöneticisine kayıt ettirilmelidir. Her hiyerarşi bir kalıcılık yöneticisine katakter katarı biçiminde bir tekil tanımlayıcı olan *kalıcılık anahtarı* ile kayıt ettirilir. Kalıcılık yöneticisi, *kalıcılık tablosu* adı verilen bir **`Hashtable`** örneğine sahiptir ve bu tabloda kendisine kayıt ettirilen her **`Actor`** nesnesi hakkında bazı bilgiler saklar.

Her kalıcılık yöneticisi çalışabilmek için bir kalıcılık tablosuna gereksinim duyar. **`PersistenceManager`** sınıfının tek kurucusu olan **`public PersistenceManager. PersistenceManager( String path, String name )`** metodundaki ilk parametre birlikte çalışılacak kalıcılık tablosunun yolunu, ikinci parametre ise kalıcılık tablosunun saklandığı dosyanın adını içerir. Verilen yolda verilen adlı ve geçerli bir kalıcılık tablosunu simgeleyen bir dosya yoksa, sistem aynı yerleşkede boş bir kalıcılık tablosu oluşturur. İstenilen bir anda kalıcılık yöneticisinin **`public boolean PersistenceManager.saveTable( )`** metodu kullanılarak kalıcılık tablosu kalıcı ortama kayıt edilebilir. Kalıcılık yöneticisi bir hiyerarşiyi kalıcılık tablosuna veya kalıcı ortama kaydettikten sonra kalıcılık tablosunun kalıcı ortamdaki içeriğini güncellediği için, kullanıcıların aslında **`saveTable()`** metodunu kendilerinin çalıştırmaları gerekmez.

Kalıcılık tablosu kalıcı ortamda şifrelenmiş bir dosya olarak saklanır. Her rol hiyerarşisini simgeleyen tekil tanımlayıcı gizli tutulduğu ve kullanılan şifreleme kırılmadığı sürece, bir rol hiyerarşisinin bilgileri yetkisiz erişimlerden korunacaktır.

Her bir rol hiyerarşisi için kalıcılık tablosunda tutulan bilgiler, bu hiyerarşinin kalıcı ortamdan geri yüklenmesini sağlayacak olan bilgilerdir. Söz konusu bilgiler şunlardır:

- Actor örneğinin sınıf adı.
- Bu Actor nesnesini ve sahip olduğu tüm rolleri kalıcı ortamda barındıracak dosyanın, diğer bir deyişle *veri dosyasının* adı.
- Rol hiyerarşisi hakkında bilginin yazılacağı *bilgi dosyasının* adı.

JAWIRO'nun kalıcılık yeteneğini sağlayan kalıcılık yöneticisi ve hakkındaki önemli ayrıntılardan sonra, kalıcı olması istenen bir rol hiyerarşisi için kullanıcının yapması gereken üç işlem bulunmaktadır:

1. Bir kalıcılık yöneticisi, mevcut veya yeni bir kalıcılık tablosu ile oluşturulur.
2. Hiyerarşi kökünü simgeleyen **`Actor`** nesnesi için bir kayıt anahtarı seçilir ve hiyerarşi kökü kalıcılık yöneticisine kayıt ettirilir. Bu amaçla **`public`**

`boolean PersistenceManager.register(Object anActor, String key)` metodu kullanılır. İlk parametre **`Actor`** nesnesini, ikinci parametre kayıt anahtarını simgeler.

3. Hiyerarşi kökü, doğru kayıt anahtarı ile ve **`public boolean PersistenceManager.upload(Object anActor, String key)`** metodu kullanılarak kalıcı ortama kaydedilir. İlk parametre hiyerarşi kökünü, ikinci parametre ise kayıt anahtarını simgeler.

Hiyerarşi kökünü kalıcılık yöneticisine kayıt ettirmek için ikinci adımda **`public String PersistenceManager.register(Object anActor)`** metodu da kullanılabilir. Bu durumda kalıcılık yöneticisi geriye uygun bir kayıt anahtarı döndürecektir. Hiyerarşiye daha sonra tekrar erişebilmek için kullanıcı bu kayıt anahtarını bir şekilde saklamalıdır. Kullanılan kayıt metodu hangisi olursa olsun, veri dosyasının ve bilgi dosyasının adları kayıt aşamasında kalıcılık yöneticisi tarafından belirlenir. Veri dosyasının adı ilgili **`Actor`** örneğinin, bilgi dosyasının adı ise her **`Actor`** örneğinde bulunan **`RoleHierarchy`** üyesinin **`Object.hashCode()`** metodu kullanılmıştır. Bu metot nesnelerin bellek adreslerinin bir fonksiyonunu döndürür ve JVM kodun yaşam süresi boyunca **`hashCode()`** metodunun tüm farklı nesneler için birbirinden farklı tamsayılar geri döndüreceğini garanti eder. Eğer bu tamsayının karakter şeklinde gösterimi tekil bir dosya adı sağlayamazsa, çakışmaya yol açan dosya adının sonuna bir basamak daha eklenir ve farklı bir dosya adına ulaşılıncaya dek bu basamağın değeri bir arttırılır. Örneğin 8452719 adlı dosya zaten varsa 84527191 adlı dosya denenir, bu isimli bir dosya da varsa bu kez 84527192 denenir. Veri ve bilgi dosyalarının bulunduğu dizin ise kalıcılık tablosunun bulunduğu dizin ile aynı olacaktır.

Kullanıcı bir rol hiyerarşisini kalıcı ortama kaydetmek amacıyla yukarıda verilen işlemleri yaptıktan sonra, rol modeli bu amacı gerçekleştirmek için aşağıdaki işlemleri yapar:

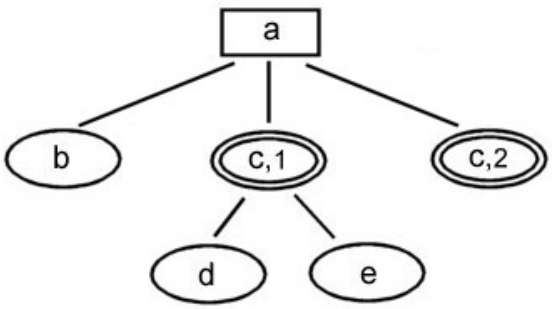
1. Kalıcılık yöneticisi **`Actor`** örneğini kalıcı ortama kaydeder.

2. **`Actor.hierarchy`** üyesi hiyerarşinin geri kalan nesnelerini kalıcı ortama kaydeder.

3. Kalıcılık yöneticisi oluşturulan tüm dosyaları 64-bit DES algoritmasını kullanarak şifreler.

Veri ve bilgi dosyalarının oluşturulmasından kalıcılık yöneticisi sorumlu olmakla birlikte, tüm hiyerarşinin dolaşılarak rol nesnelerinin veri dosyasına eklenmesinden hiyerarşi kökünün **`hierarchy`** üyesi sorumludur. **`hierarchy`** üyesi bu sırada bilgi dosyasının da içini doldurur. Bilgi dosyasında öncelikle hiyerarşi kökünün hangi rolleri oynadığı, ardından da diğer rollerin hangi rolleri oynadığı yazılır. Bilgi dosyasındaki gösterim, rol hiyerarşisinin ağaç yapısını yansıtacak şekildedir. **`hierarchy`** üyesi bu görevleri kalıcılık yöneticisinden aldığı dosya işaretçilerinin sayesinde yerine getirir.

Şekil 8.20'de kalıcı ortamda saklanılacak bir rol hiyerarşisi örneği ve bu hiyerarşi için oluşturulan bilgi dosyasının şifrelenmemiş hali görülebilir. Şekil 8.20'de hiyerarşi kökü dörtgen, roller elips ve nitelikli roller çift elips içinde niteleyicisi ile birlikte gösterilmiştir. Bilgi dosyasında roller sınıf adları ile gösterilmekte, eğer bu bir nitelikli rol ise ek olarak sınıf adının sonuna virgül konulup rolün niteleyicisi yazılmaktadır. Bilgi dosyasında # karakteri ile başlayan satırlar ise, listelenen rolleri kimin oynadığını belirtmek amacını taşır. Bu bilgiler ışığında Şekil 8.20 incelendiğinde, hiyerarşi ile bilgi dosyası arasındaki ilgi anlaşılabilir: Bilgi dosyasının ilk altı satırında hiyerarşi kökünün oynadığı rollerin "b", "c,1" ve "c,2" olduğu bilgisi yer almaktadır. Bilgi dosyasının son beş satırında ise benzer şekilde "c,1" niteleyicisine sahip rolün alt rolleri ve bunların saklandığı dosyaların adları yer almaktadır. Örnek hiyerarşinin kalıcı ortamda saklanması için programcının yazması gereken kod ise Şekil 8.21'de gösterilmiştir.



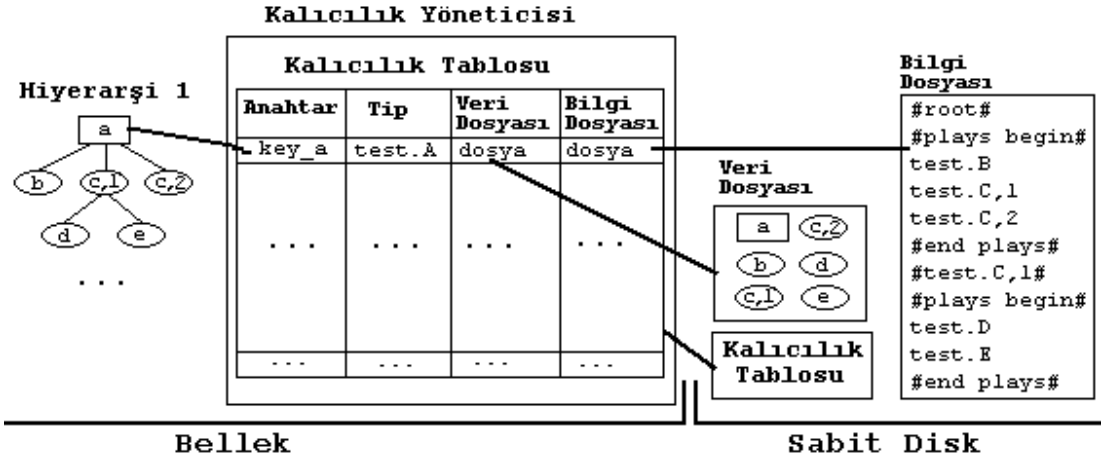

```
#root#
#plays begin#
test.B
test.C,1
test.C,2
#end plays#
#test.C,1#
#plays begin#
test.D
test.E
#end plays#
```

Şekil 8.20: Kalıcı ortamdasaklanacak bir rol hiyerarşisi ve ilgili bilgi dosyasının açık hali

```
PersistenceManager pm;
A a = new A(); //A extends Actor
construct_hierarchy(); //Şekil 4.13'teki hiyerarşiyi oluştur
pm = new PersistenceManager( "C:\\Temp\\", "test05" );
pm.register( a, "key_a" );
pm.upload( a, "key_a" );
```

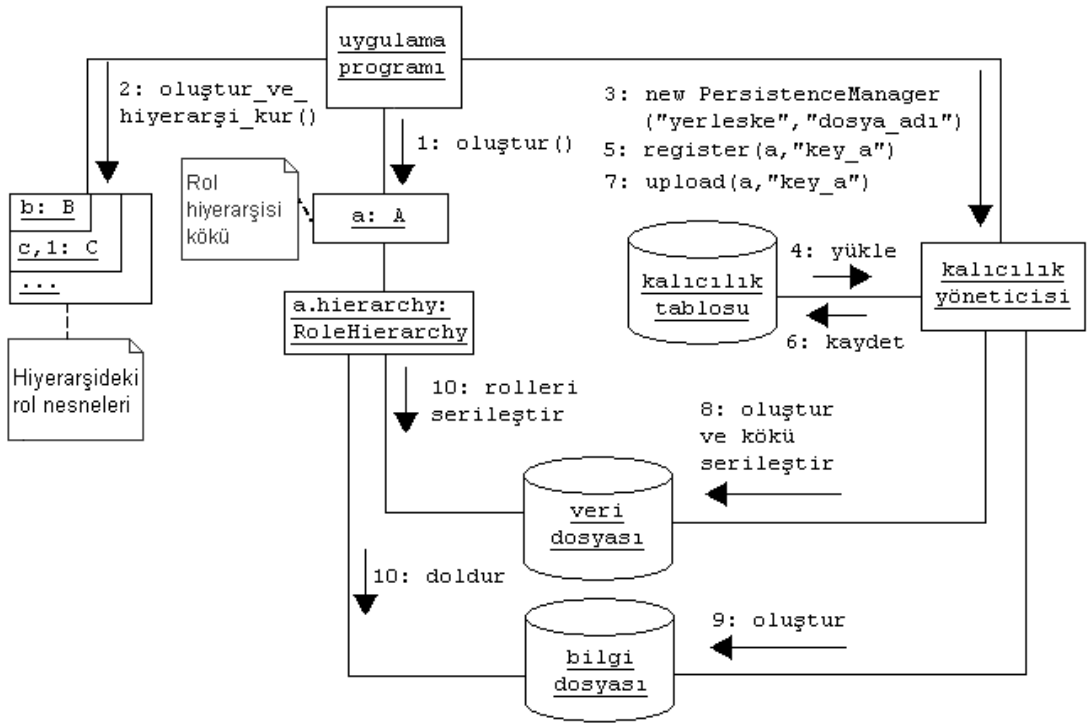
Şekil 8.21: Örnek hiyerarşiyi kalıcı ortama kaydeden kod

Şekil 8.21'de verilen örnek kod çalıştıktan sonra, nesnelerin bellek ve kalıcı ortamdaki durumları Şekil 8.22'de gösterildiği gibi olacaktır. Şekil 8.22'deki elemanlar arasında kurulan bağlantılar koyu çizgilerle gösterilmiştir. Örneğin söz konusu rol hiyerarşisinin kayıdının kalıcılık tablosunda bulunabilmesi, o hiyerarşinin anahtarı sayesinde olacaktır. Bu nedenle Şekil 8.22'deki koyu çizgilerden birisi hiyerarşi ile kalıcılık tablosundaki anahtar alan arasına çekilmiştir. Şekil 8.23'te ise örnek senaryo sırasında gerçekleşen mesaj akışlarını gösteren UML işbirliği şeması verilmiştir.



Şekil 8.22: Kalıcılık eylemi ardından nesnelerin bellekteki ve kalıcı ortamdaki durumları ile aralarındaki bağlantılar

Şekil 8.23'te görülen UML işbirliği şemasının kuralları, anlatım kolaylığı için esnetilmiştir: Kalıcı ortamdaki dosyalar silindir şeklinde gösterilmiştir, kalıcılık tablosu ve bilgi dosyalarının fiziksel adları verilmemiştir, dosyalar ile yapılan işlemlerde mesaj işlemi yapan nesnenin kendisine yönelik olmasına rağmen ok yönü veri akışının yönünü göstermektedir. Bu esnetme maddelerinden sonuncusunun bir örneği 4. mesajdır: Kalıcılık yöneticisi yükleme mesajını kendisine gönderir ancak kalıcılık tablosunun kalıcı ortamdan yüklenmesi söz konusu olduğu için kalıcılık tablosundan kalıcılık yöneticisine doğru bir ok çizilmiştir.



Şekil 8.23: Kalıcılık eylemi sırasındaki mesaj akışları

Bir hiyerarşinin kalıcı olması gerekmediği zaman **`public void PersistenceManager.unregister(String key)`** metodu kullanılarak, anahtarı doğru verilen **`Actor`** nesnesinin kaydı kalıcılık tablosundan silinir. Bu komut o hiyerarşi ile ilgili veri dosyasını ve bilgi dosyasını da kalıcı ortamdan siler.

Kalıcı ortama kaydedilmiş bir rol hiyerarşisinin geri yüklenmesi işlemi ise şu şekilde gerçekleşir:

- Bir kalıcılık yöneticisi, bölümün başında anlatılan kurucu fonksiyonuna bir kalıcılık tablosunun yerleşkesi verilerek oluşturulur. Geri yüklenecek hiyerarşi bu kalıcılık tablosunda kayıtlı olmalıdır.

- Rol hiyerarşisinin kökünün bir örneği oluşturulur. Hiyerarşideki diğer rol nesnelerinin oluşturulmasına gerek yoktur.

- Kalıcılık yöneticisine doğru anahtar sağlandığı takdirde, oluşturulan köke dosyadan yüklenilen durum bilgisi aktarılır. Bu amaçla **`public Object PersistenceManager.download( String key )`** metodu kullanılır.

- Kalıcılık yöneticisi hiyerarşinin geri kalanını oluşturan nesneleri yükler ve gerekli rol ekleme işlemlerini doğru sırada gerçekleştirir.

Rol nesnelerinin güvenliği, tüm bilgilerin 64-bit DES algoritması ile şifrelenmesi ve her **`Actor`** nesnesinin hem doğru anahtara hem doğru tipe sahip olmadan kalıcı ortamdaki rollere erişememesi ile sağlanmıştır. 64-bit DES algoritmasının seçilmesinin nedeni, bu algoritmanın Windows işletim sisteminin herhangi bir versiyonunun kurulu olduğu tüm PC'lerde bulunmasıdır.

Önceki örneklerde kalıcı ortama saklanan hiyerarşinin geri yüklenmesi için programcının yazması gereken kod Şekil 8.24'te verilmiştir. Şekilden anlaşılacağı gibi, programcının hiyerarşiyi yeniden oluşturmak için kod yazması gerekmez.

```
PersistenceManager pm;
A a = new A();
pm = new PersistenceManager("C:\\Temp\\","test05");
a = (A) pm.download("key_a");
```

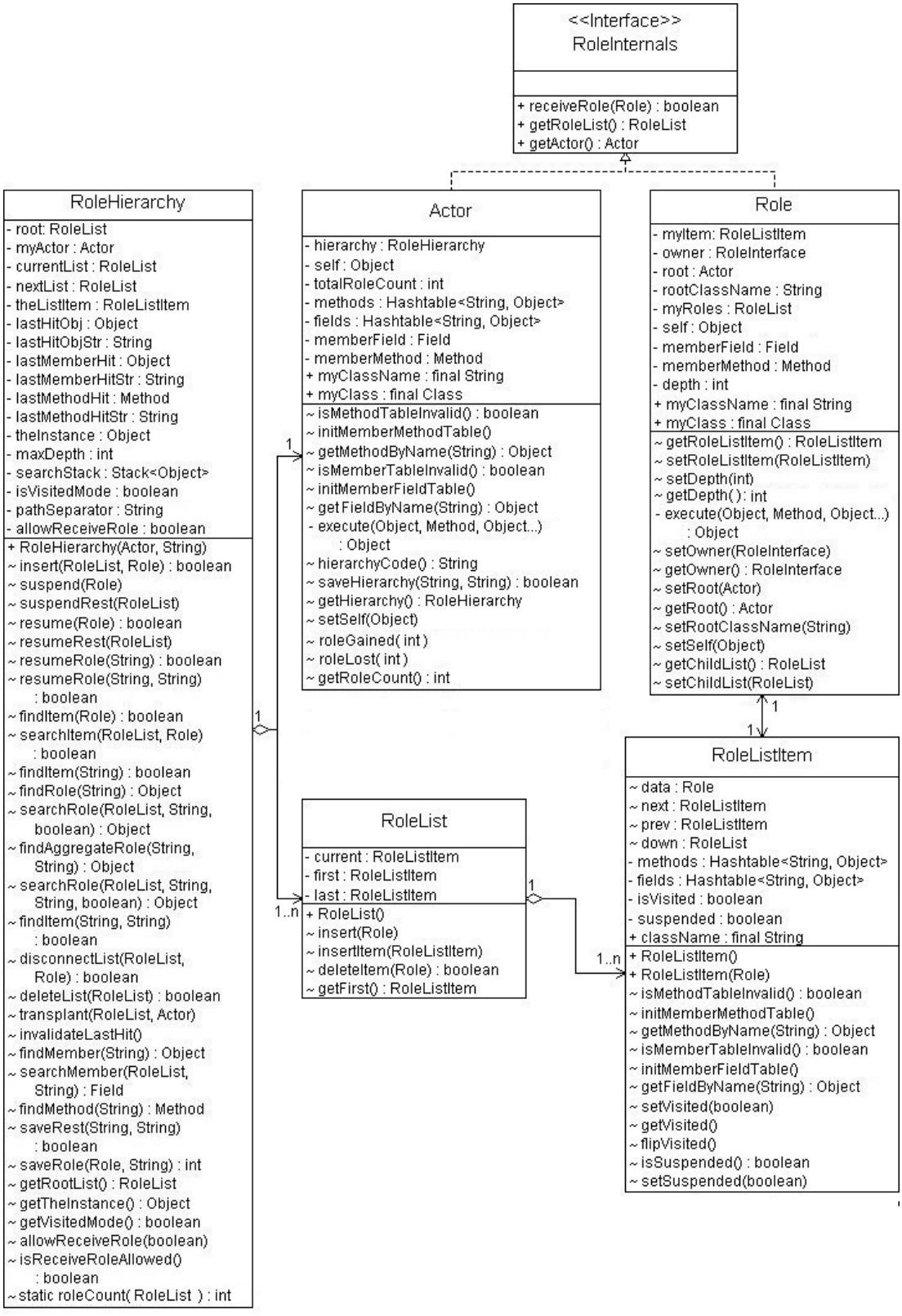
Şekil 8.24: Örnek hiyerarşiyi kalıcı ortamdan geri yükleyen kod

JAWIRO'da bir hiyerarşinin tüm katılımcılarının aynı dosya içerisine serileştirilmesi, bağlantı kopması olarak adlandırılabilecek bir sorunu ortadan kaldırmıştır. Bir hiyerarşinin katılımcılarından herhangi birinde, aynı hiyerarşideki başka bir nesneyi gösteren bir işaretçi bulunması durumunu ele alalım. İşaretçinin sahibi kaynak nesne, işaretçinin gösterdiği nesne ise hedef nesne olarak adlandırılsın. Eğer hiyerarşilerin katılımcıları ayrı ayrı dosyalara serileştirilirlerse, hiyerarşinin geri yüklenmesinin ardından artık kaynak nesnedeki işaretçi hedef nesneyi göstermeyecektir. Bunun yerine hedef nesnede kaynak nesnenin bir kopyası olacaktır.

Aynı hiyerarşinin üyeleri için bağlantı kopması sorununun çözülmesine rağmen, aynı sorun birden fazla hiyerarşi arasında da ortaya çıkabilir. Önceki paragraftaki adlandırma sürdürülürse, kaynak nesne ile hedef nesnenin farklı hiyerarşilerde bulunması durumunda, bağlantı kopması sorununu çözmek için söz konusu iki hiyerarşi aynı dosyaya serileştirilmelidir. Bu amaçla **`PersistenceManager`** sınıfına birden fazla hiyerarşiyi birlikte serileştiren **`public boolean upload( String[] keys, Actor... actorList)`** ve birden fazla hiyerarşiyi tek adımda geri yükleyen **`public boolean download( String[] keys, Actor... actorList )`** metotları eklenmiştir.

8.2 Gerçekleme Detayları

Bu bölümde gerçeklenen rol modelinin iç çalışması hakkında bilgi verilecektir.



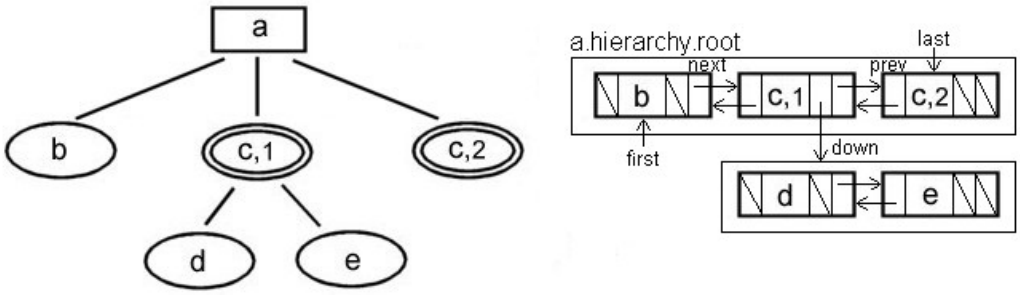
Şekil 8.25: Gerçeklenen rol modelinin iç yapısı

JAWIRO'nun gerçekleme detayları ile ilgili üyelerini gösteren UML şeması Şekil 8.25'te görülebilir. Özetle; **`RoleList`** ve **`RoleListItem`** sınıfları rol hiyerarşilerinin modellenmesinde, **`RoleHierarchy`** sınıfı hiyerarşinin idaresinde, **`RoleInternals`** arayüzü ise hiyerarşi katılımcılarının rol modelinin iç çalışması ile ilgili konularda eş biçimli olarak işlenmesinde kullanılmıştır.

8.2.1 Hiyerarşi Yapısının Modellenmesi

Rol hiyerarşilerini simgeleyen ağaç yapısı çift yönlü bağlı listeler kullanılarak gerçeklenmiştir. Her rol sahibinin alt rolleri bir **`RoleList`** örneğinin simgelediği listenin **`RoleListItem`** türünden düğümlerinin **`data`** üyesinde **`Role`** örnekleri olarak saklanmaktadır. JAWIRO'nun gerçeklenmeye başlandığı tarihlerdeki Java sürümü olan 1.4'te **`java.util`** paketi ile gelen hazır veri yapılarında yalnız **`Object`** türünden nesneler tutulabilmekte, bu da rol nesnelerinin her kullanımda **`Role`** tipine dönüştürülmesini gerektirmekteydi. Başarım kayıplarını önlemek için Java ile hazır gelen bağlı liste yapısı yerine özel bir bağlı liste yapısı kullanılmıştır. Java 1.5 sürümü ile gelen tip değişkenleri ve jenerik programlama (*generic programming*) özellikleri bu gereksinimi ortadan kaldırsa da, bağlı liste yapısı rol modelinin temeli olduğundan çok zaman alacağı düşünülerek hazır listelere geçilmemiştir.

Rol hiyerarşisinin kurulması, sürdürülmesi ve kullanılması ile ilgili işlemlerden büyük oranda **`RoleHierarchy`** sınıfı sorumludur. Hiyerarşi kökünün derinliği 1 kabul edilir. 2. düzey roller, başka bir deyişle kökün alt rollerinin listesi **`private RoleList RoleHierarchy.root`** üyesinde tutulur. Daha alt düzey rollerin saklandığı listelere erişmek için, o rollerin sahibinin saklandığı **`RoleListItem`** örneğinin **`RoleList`** türünden **`down`** üyesi kullanılır. Kardeş roller arasında dolaşmak için ise ilgili **`RoleListItem`** üyesinin **`RoleListItem`** türündeki **`next`** ve **`prev`** üyeleri kullanılır. **`RoleList`** sınıfı; sırasıyla listenin başını, sonunu ve dolaşma işlemleri sırasında bulunulan yeri gösteren **`RoleListItem`** türünden **`first`**, **`last`** ve **`current`** üyelerine sahiptir. **`RoleListItem RoleList.getFirst()`** metodu ile **`first`** üyesine erişilebilir. Şekil 8.26'da örnek bir rol hiyerarşisi, bellekteki yerleşim şekli ile birlikte görülebilir.



Şekil 8.26: Bir rol hiyerarşisi ve içsel gösterimi

Şekil 8.26'da hiyerarşi kökü dörtgen, rol nesneleri elips, nitelikli roller ise çift elips ile gösterilmiştir. Hiyerarşi yöneticisinin kurucu metodu **`Actor`** sınıfının kurucu metodunda çağırılırken parametre olarak **`private Actor RoleHierarchy.myActor`** üyesine atanmak üzere hiyerarşi kökünü gösteren bir işaretçi ve **`private String RoleHierarchy.pathSeparator`** üyesine atanmak üzere otomatik olarak öğrenilen yerleşke yolu ayırt edici karakteri kullanılır. Çalışma ortamına bağlı olarak düz veya ters bölü işareti olabilen ayırt edici karakter, Bölüm 8.1.2.8 ve Bölüm 8.2.9'da anlatılacak kalıcılık yeteneği için kullanılmaktadır. Hiyerarşi yöneticisine **`RoleHierarchy Actor.getHierarchy( )`** metodu ile erişilebilir.

Her rol nesnesi, kendisini rol hiyerarşisinde simgeleyen düğümden haberdardır. Bu bilgi **`private RoleListItem Role.myItem`** üyesinde saklanır. Gerekli durumlarda bu bilgiye **`RoleListItem Role.getRoleListItem( )`** metodu ile erişilmektedir. Benzer şekilde, her rol nesnesi rol hiyerarşisinin kökünden haberdardır. Hiyerarşi kökü **`private Actor Role.root`** üyesinde saklanır. Bu bilgi **`Actor Role.getRoot()`** metodu ile okunabilir ve **`void Role.setRoot (Actor newRoot)`** metodu ile değiştirilebilir. Hiyerarşi kökünün tip adı ise **`private String Role.rootClassName`** üyesinde saklanır ve bu üyeye **`void Role. setRootClassName(String rootName)`** metodu ile bilgi atanır.

Bir rol hiyerarşisinin her katılımcısı, çocuk rollerinin listesinden haberdardır. Rol nesneleri için bu bilgi **`private RoleList Role.myRoles`** üyesinde saklanır. Bu bilgi **`RoleList Role.getChildList()`** metodu ile okunabilir ve **`void Role. setChildList(RoleList newList)`** metodu ile değiştirilebilir. Hiyerarşi kökünün rol listesine ise **`RoleList RoleHierarchy.getRootList()`** metodu ile erişilir.

Her rol nesnesi kendi sahibinden haberdardır. Bu bilgi **`private RoleInterface Role.owner`** üyesinde saklanır, **`RoleInterface Role.getOwner( )`** metodu ile okunur ve **`void Role.setOwner(RoleInterface newOwner)`** metodu ile değiştirilir.

8.2.2 `RoleInternals` Arayüzü

`Actor` ve **`Role`** sınıfları **`RoleInternals`** arayüzünü rol modelinin iç işlevlerinden bazılarının yapılabilmesi amacıyla gerçekler. Bir Java arayüzünde tanımlı metodlar kendiliğinden **`public`** ilan edilirler. İç işlev için gerekli metodların **`public`** ilan edilmesi uygun değildir, bu metodlar bir üst sınıfta toplanıp alt sınıflarda yeniden tanımlanmalıdır. Buna rağmen JAWIRO'da **`RoleInternals`** bir sınıf değil arayüz olarak hazırlanmıştır. Bu seçimin nedeni **`Actor`** ve **`Role`** sınıflarının davranışlarının çok farklı olmasıdır. Rol modelinin iç işlevleri için gerekli komutların bazıları kritik öneme sahiptir. Böyle bir metot **`RoleInternals`** sınıfı ile çalıştığı takdirde, üst sınıf-alt sınıf ilişkilerinin kasıtlı veya kasıtsız olarak yanlış kullanımı ile bir **`Actor`** örneği yerine bir **`Role`** örneğinin geçirilmesi gibi sorunlar ortaya çıkabilirdi. Bu nedenle **`RoleInternals`** bir arayüz olarak gerçeklenmiştir.

`RoleInternals` arayüzünün **`RoleList getRoleList()`** metodu, rol hiyerarşisi üyelerini eş biçimli olarak işleyebilmek için bir rolün terk edilmesini sağlayan **`Role.resign`** ve bir rolün başka bir sahibe aktarılmasını sağlayan **`Role.transfer`** metotları tarafından çağırılır. **`getRoleList`** metodunun serbestçe kullanımı, geri döndürdüğü **`RoleList`** üyesinin iç yapıdaki önemi nedeniyle, ilk bakışta sakıncalı görülebilir. Ancak **`RoleList`** sınıfının bütün metodları sadece jawiro paketi içerisinden kullanılabilecek şekilde tanımlanmıştır. Bu nedenle **`getRoleList`** metodunun serbestçe kullanımı bir güvenlik açığı oluşturmamaktadır.

Bir rolün başka bir sahibe aktarılmasını sağlayan **`Role.transfer`** metodu tarafından kullanılan **`boolean RoleInternals.receiveRole(Role r)`** metodunun kullanıcılar tarafından çalıştırılması ise sorun çıkarabilir. Dolayısıyla **`receiveRole`** metodunun paket dışından kullanımını engelleyen bir mekanizma kurulmuştur: **`private boolean RoleHierarchy.allowReceiveRole`** bayrak değişkenini değiştiren **`void RoleHierarchy.allowReceiveRole(boolean`**

allow) metodu sadece jawiro paketi içinden çağrılabilir. Rol modelinde **receiveRole** metodunun çağrılması gereken yerlerden önce **allowReceiveRole** üyesine **true** değeri atanmış, işlem yapılmış ve hemen ardından **allowReceiveRole** bayrağı **false** yapılmıştır. **receiveRole** metodu ise bayrağın değerine **boolean RoleHierarchy.isReceiveRoleAllowed()** metodu ile bakmakta, bu değer **true** değilse işlemi yapmamaktadır. Rol hiyerarşisindeki herhangi bir katılımcıdan o hiyerarşinin bayrak değişkenine ulaşabilmek için önce hiyerarşi köküne, oradan hiyerarşiden sorumlu hierarchy üyesine ulaşmak gereklidir. Bu amaçla da **Actor RoleInternals.getActor()** metodu eklenmiştir.

8.2.3 Rol Kazanımı, Terki ve Aktarımı

Rol kazanım komutu olan ve Bölüm 8.1.1.1'de anlatılan **RoleInterface.addRole** metodunun yanı sıra, rol aktarım komutu olan ve Bölüm 8.1.2.2'de anlatılan **Role.transfer** metodu da bir rol hiyerarşisi katılımcısına bir veya daha fazla yeni rol kazandırabilir. **Role.transfer** metodu ilk olarak aktarım işlemini varsa hiyerarşinin kısıtlama yöneticisinin onayına sunar, ardından da aktarılan rolün yeni sahibinin **receiveRole** metodunu öncesinde ve sonrasında Bölüm 8.2.2'de anlatılan engelleme mekanizmasının gerektirdiği komutları vererek çalıştırır. Yeni sahibin **receiveRole** metodu da ilk olarak aktarılacak rolü kendine eklemek için kendisinin **addRole** metodunu çalıştırır. **receiveRole** metodu ikinci olarak **void RoleHierarchy.transplant(RoleList rl, Actor root)** metodunu çağırarak, aktarılacak rolün listesinde eski kök bilgisi yerine rolün yeni sahibinin bulunduğu hiyerarşinin kökünün özyineli olarak yazılmasını sağlar.

RoleInterface.addRole metodunun doğrudan kullanıcı tarafından mı, yoksa rol aktarımı işlemi sonucu dolaylı olarak rol modeli tarafından mı çalıştırıldığı; yeni eklenecek rolün hiyerarşi ağacındaki düğümünü simgeleyen ve Bölüm 8.2.1'de anlatılan **RoleListItem** türünden **myItem** üyesinin değerinin **null** olup olmadığının sınanması ile anlaşılır. Bu değerin **null** olması, eklenecek rolün herhangi bir hiyerarşide bulunmayıp **addRole** metodunun doğrudan kullanıcı tarafından çalıştırıldığını anlatır.

RoleInterface.addRole metodu, **Actor** ve **Role** sınıflarında bazı ufak farklarla birlikte benzer şekilde gerçeklenmiştir. Yeni rol sahibinin rol ekleme komutu, doğrudan veya dolaylı çalıştırılmasından bağımsız olarak; yeni rolün hiyerarşi kökü yerine kendi hiyerarşisinin kökünü ve yeni rolün sahibi olarak kendisini atar. Bunun ardından yeni rolün yeni hiyerarşideki derinliğini ve yeni rolün kökünün tip adını da atar. Bu noktadan sonra **addRole** metodunun çalıştırılma biçimi önem kazanır. Olası iki seçenek şunlardır:

- Rol ekleme komutu doğrudan çalıştırılmışsa hiyerarşi yöneticisinin **boolean RoleHierarchy.insert(RoleList level,Role r)** metodu ile yeni rol hiyerarşi ağacında gerekli yere yerleştirilir. **RoleHierarchy.insert** metodu da **void RoleList.insert(Role r)** metodundan yardım alır. **RoleList.insert** metodunun görevi yeni rol için **RoleListItem** türünden yeni bir nesne oluşturmak ve bunu yeni rolün **void Role.setRoleListItem(RoleListItem newItem)** metodunu çağırarak yeni role atamaktır. **RoleList.insert** metodu son olarak **void RoleList.insertItem(RoleListItem li)** metodunu çağırarak yeni rolü simgeleyen düğümün listeye yerleştirilmesini sağlar.

- Rol ekleme komutu dolaylı olarak çalışmışsa yeni bir **RoleListItem** nesnesi oluşturmaya gerek kalmayacaktır. Bu nedenle yalnızca **void RoleList.insertItem(RoleListItem li)** metodu çağrılır.

Rol aktarımı sırasında rolün yeni sahibi için rol ekleme komutu olan **RoleInterface.addRole** metodu çalıştırılmasına rağmen, rolün eski sahibi için rol terki komutu olan **Role.resign** metodu çalıştırılmamaktadır. Fakat her iki komutta da terk edilecek veya aktarılacak rol, eski hiyerarşisindeki listesinden **boolean RoleList.deleteItem(Role r)** metodu ile silinir. Silinmeden önce rolün sahibi ve dolayısıyla rolün bulunduğu liste bilindiğinden, bu metod listenin başından başlayıp silinecek rolün bulunduğu düğümü ardışıl olarak arar ve bulunca da siler.

Bir rol tamamen terk edileceği zaman, o rol nesnesinin Bölüm 8.1.1.1'de değinilen **resign** metodu çalıştırılır. Bu metot terk edilecek rolü hiyerarşiden ayırmak için önce hiyerarşi yöneticisinin **boolean RoleHierarchy.disconnectList**

(RoleList rl, Role r) metodunu çağırır. **disconnectList** metodu terk eidlecek rolün de alt rolleri bulunuyorsa bu alt rolleri simgeleyen listenin terk edilecek rolün sahibini simgeleyen düğüm ile bağını keser. Bunun ardından **disconnectList** metodu **boolean RoleHierarchy.deleteList (RoleList level)** metodunu çağırarak, terk edilecek role ait listeden başlayarak hiyerarşiyi oluşturan tüm bağları çözer. Çözme işi özyineli şekilde yapılır. Tüm bağlar koptuktan sonra **Role.resign** metodu **boolean RoleList. deleteItem(Role r)** metodunu çağırarak kendisini simgeleyen düğümü, içinde bulunduğu listeden siler.

8.2.4 Hiyerarşide Arama İşlemleri

Kullanıcı arayüzünün bazı metotları hiyerarşide özyineli aramayı, bazıları da yığın yapısı yardımıyla derinlemesine aramayı tetikler. Bölüm 8.1.1.3'te anlatılan rol varlığı kontrolü ve rol geçişi komutlarında özyineli arama yapıp ilk bulunan eşleşmede aramayı sonlandırmak yeterli olacaktır. Bir rol hiyerarşisinde herhangi bir tipten rol örneği veya istenen niteleyici ve tipe sahip nitelikli rol örneğinden sadece bir tane olabileceği için ilk eşleşme aynı zamanda tek eşleşme olacaktır. Ancak Bölüm 8.1.2.4'te anlatılan ad ile üye erişimi komutlarında hiyerarşideki farklı tipten katılımcılar aynı adlı üyeye sahip olabileceğinden tüm eşleşmelerin içerisinden en çok evrimleşmiş katılımcının üyesine erişim sağlamak gerekecektir. Bu tip çalışmaya da yığın ile arama uygundur. Hiyerarşide yapılan tüm aramalardan **RoleHierarchy** sınıfı sorumludur.

8.2.4.1 Özyineli Arama

Rol geçişi komutları, özyineli aramayı kullanan aynı metottan yararlanır. **RoleInterface.as** metotları **Object RoleHierarchy.findRole(String typeName)** metodunu, o da önce istenen rolün bizzat hiyerarşi kökü olup olmadığını kontrol ettikten sonra aranan kök değilse hiyerarşide özyineli aramayı başlatmak üzere **Object RoleHierarchy.searchRole(RoleList list, String typeName, boolean forResume)** metodunu çağırır. Arama kök listesini simgeleyen **RoleHierarchy.root** üyesinden başlatılır. **searchRole** metodu hem **findRole** metodu tarafından hem de bir rolün askıdan geri alınması için Bölüm

8.1.2.1'de anlatılan **`Actor.resumeRole`** tarafından çağırıldığından; **`searchRole`** metodunun son parametresi çağrının askıdan geri alma işlemi için yapılıp yapılmadığını simgeler. Bir nitelikli rol aranacaksa bu kez **`Object RoleHierarchy.searchRole(RoleList list, String typeName, String identifier, boolean forResume)`** metodu kullanılır.

Özyineli aramayı başlatmadan önce ve bitirdikten sonra **`searchRole`** metodu Bölüm 8.2.5'te detayları verilecek olan rol araması ve geçişindeki iyileştirme mekanizmasının gereklerini yerine getirir. Aranan rol bulunursa geri döndürülür, aksi halde **`null`** değeri geri döndürülür.

Nitelikli roller için aranan niteleyicinin de sınanması gerektiğinden, nitelikli roller için olan rol geçişi komutları **`Object RoleHierarchy.findAggregateRole( String typeName, String identifier)`** metodunu, o da **`Object RoleHierarchy.searchRole(RoleList list, String typeName, String identifier, boolean forResume)`** metodunu çalıştırır. Bu komutların nitelikli roller için olan biçimleri, niteleyici sınaması dışında normal roller için olan eşlenikleri ile aynı şekilde çalışır.

Rol varlığı kontrolünün ilk yolu olan **`canSwitch`** metotlarının ikinci adımı, rol geçişi komutları ile ortaktır. Normal roller için olan **`RoleInterface.canSwitch( String typeName)`** komutu **`boolean RoleHierarchy.findItem(String typeName)`** metodunu, o da yukarıda anlatılan aynı **`Object RoleHierarchy. searchRole(RoleList list, String typeName, boolean forResume)`** metodunu çağırır. **`findRole`** ile **`findItem`** metotları arasındaki benzerlik aynı özyineli arama metodunu çağırmaları; aralarındaki fark ise öncekinin doğrudan aranan nesneyi döndürürken sonrakinin aranan nesnenin bulunup bulunmadığını simgeleyen ikili değeri döndürmesidir. Nitelikli rollerin kontrolü ise aradaki tek farkın niteleyicinin de sınanması olan **`boolean RoleHierarchy.findItem( String typeName, String identifier)`** metodunu kullanır.

Rol varlığı kontrolünün ikinci yolu olan **`canDelegate(Role aRole)`** metodunun da özyineli aramayı tetiklemesine rağmen, bu iş için tip adını sınayan **`searchRole`** metodu kullanılamayacaktır. **`canDelegate`** metodu özyineli aramayı kökten başlatacak olan **`boolean RoleHierarchy.findItem(Role aRole)`** metodunu,

o da aramayı yapan **`boolean RoleHierarchy.searchItem(RoleList list, Role aRole)`** metodunu çalıştırır. Özyineli arama metotları olan **`searchRole`** ve **`searchItem`** arasındaki tek fark; öncekinin tip adını, sonrakinin ise nesne eşitliğini sınamasıdır.

İstenen rol aranırken her düğümün tip adının her seferinde yansıtma ile öğrenilmesi yerine, **`Actor()`** ve **`Role()`** kurucu metotlarında bu değer elde edilerek **`public final String Actor.myClassName`** ve **`public final String Role.myClassName`** üyelerinde saklanır. Düğümlerin tipleri ise **`public final Class Actor.myClass`** ve **`public final Class Role.myClass`** üyelerinde saklanır. **`private RoleList RoleHierarchy.currentList`** ve **`nextList`** üyeleri ise özyineli aramada hiyerarşiyi dolaşmak için kullanılan geçici değişkenlerdir.

8.2.4.2 Derinlemesine Arama

Ad ile üye erişimi komutları, **`RoleHierarchy`** sınıfının rol hiyerarşisinde derinlemesine arama yapan ad ile üye erişimi mekanizmasına ait metotlarını çalıştırır. Bu bölümde sadece derinlemesine aramanın detayları anlatılacak, ad ile üye erişimi mekanizmasının diğer gerçekleme detayları Bölüm 8.2.5'te verilecektir.

Derinlemesine aramada kullanılacak yığın veri yapısı olan **`private Stack<Object> RoleHierarchy.searchStack`** üyesinin kapasitesi, hiyerarşide bulunan tüm rol nesnelerini alabilecek kadar büyük olmalıdır. Bu nedenle **`private int Actor.totalRoleCount`** üyesinde her an için hiyerarşide bulunan rol nesnesi sayısı tutulur. Bu bilgiye **`int Actor.getRoleCount( )`** metodu ile erişilebilir. Rol modeli gerekli anlarda **`void Actor.roleGained(int count)`** ve **`void Actor.roleLost(int count)`** metotlarını çağırarak **`totalRoleCount`** üyesinin değerini arttırır ve azaltır. Bir rol nesnesi alt rolleri ile birlikte işlem görecektir. Bu yüzden toplam rol sayısının ne kadar değiştirileceğini öğrenmek için, bir alt ağaçta kaç düğüm olduğunu geri döndüren **`static int RoleHierarchy.roleCount (RoleList list)`** metodu kullanılır.

Aranan ada sahip birden fazla üye bulunması durumunda hiyerarşinin en derinindeki üyeye erişim sağlanması gerektiğinden, arama sırasında bulunan en derindeki adayın derinliği de **`private int RoleHierarchy.maxDepth`** üyesinde tutulur. Her rol

nesnesinin derinliği ise **`private int Role.depth`** üyesinde tutulur. Bu bilgi **`public int Role.getDepth()`** metodu ile okunabilir ve **`void Role.setDepth(int newDepth)`** metodu ile değiştirilebilir.

Hiyerarşi ağacındaki her bir düğümün gezilip gezilmediği bilgisi, derinlemesine arama sırasında gerekli olacaktır. Bu bilgi **`private boolean RoleListItem.isVisited`** üyesinde tutulur ve ikilinin değeri **`void RoleListItem.flipVisited()`** metodu ile evrilir. Eğer ikili bayrak değişkenlerin yaygın kullanım şekli izlenip her zaman için **`isVisited`** üyesine düğüm ziyaret edilmişse **`true`**, edilmemişse **`false`** değeri atansaydı; arama işleminden sonra tüm düğümlerin bayrağına **`false`** değeri atamak için ayrıca işlem yapmak gerekecekti. Bunun yerine hiyerarşi yöneticisi ziyaret ettiği düğümlerin **`isVisited`** üyesine kendisinin **`private boolean RoleHierarchy.isVisitedMode`** üyesi ile aynı değeri atar. Derinlemesine aramada her düğüm ziyaret edileceğinden; arama sonunda tüm düğümlerin **`isVisited`** üyesi hiyerarşi yöneticisinin **`isVisitedMode`** üyesi ile aynı değere sahip olacaktır. Aramanın ardından sadece tek bir işlemle hiyerarşi yöneticisinin **`isVisitedMode`** üyesi evrilerek tüm düğümlerin **`isVisited`** üyelerinin evrilmesi ile aynı etki sağlanır. Düğümlerin **`isVisited`** üyesine değer atamak için **`void RoleListItem.setVisited(boolean aBool)`** metodu, bu üyenin o anki değerini öğrenmek için **`boolean RoleListItem.getVisited()`** metodu kullanılır. Bir rol hiyerarşisinin ziyaret edilmiş düğümleri için belli bir anda hangi mantıksal değerin kullanıldığını öğrenmek amacıyla **`boolean RoleHierarchy.getVisitedMode()`** metodu kullanılabilir. Bölüm 8.2.3'te anlatıldığı üzere ister kullanıcı tarafından doğrudan ister rol aktarımı nedeniyle rol modelince çalıştırılan rol ekleme metodu, hiyerarşinin yeni katılımcısını simgeleyen düğümü **`getVisitedMode`** ve **`setVisited`** metotlarını kullanarak ziyaret edilmemiş olarak işaretlemek zorundadır.

8.2.5 Ad ile Üye Erişimi

Ad ile üye erişimi yeteneği, Java dilinin yansıtma yetenekleri kullanılarak gerçeklenmiştir. İstenen herhangi bir üyeye erişebilmek için, rol hiyerarşisinin tüm katılımcılarının üye değişken ve metotları önceden bilinmelidir. Hiyerarşi kökünün üye değişkenleri **`private transient Actor.Hashtable<String,Object>`**

fields, üye metotları ise **private transient Actor.Hashtable<String, Object> methods** tablosunda saklanır. Hiyerarşinin diğer katılımcıları için ise **private transient RoleListItem.Hashtable<String,Object> fields** ve **private transient RoleListItem.Hashtable<String,Object> methods** tabloları kullanılır. Tabloları üye bilgileri ile doldurmak için **Actor** ve **RoleListItem** sınıflarının **void initMemberFieldTable()** ve **void initMemberMethodTable()** metotları kullanılır. Tabloları üye varlığı için sorgulamak ise **Actor** ve **RoleListItem** sınıflarının **Object getFieldByName (String name)** ve **Object getMethodByName(String name)** metotları ile yapılır.

RoleInterface.bringMember metodu, Bölüm 8.1.2.5'te anlatılan baskınlık kurallarının gereğini yerine getirecek şekilde hiyerarşi kökünün ve nesnenin kendisinin baskın olup olmadığını kontrol ettikten sonra **Object RoleHierarchy. findMember(String name)** metodunu çağırır. Bu metot Bölüm 8.2.4.2'de anlatıldığı gibi aramada kullanılan yığıtın kapasitesini ayarlar ve aramayı başlatmak üzere **Field RoleHierarchy.searchMember(RoleList list, String name)** metodunu çağırır. **searchMember** metodu aranan üye değişkeni bulduğu takdirde bunu simgeleyen **Field** nesnesini döndürür ve bu üye değişkenin ait olduğu hiyerarşi katılımcısını da **private Object RoleHierarchy.theInstance** değişkeninde saklar. **findMember** metodu, Bölüm 7.3.3'te anlatıldığı şekilde, **searchMember** metodundan aldığı **Field** nesnesini ve **theInstance** değişkenini kullanarak istenen üyeyi elde edip geri döndürür. **Actor** ve **Role** sınıflarının **memberField** alanı bu işlemler sırasında **findMember** metodu tarafından geçici değişken olarak kullanılır.

RoleInterface.executeMethod metodu da önce baskınlık kurallarının gereğini yerine getirip ardından **Object RoleHierarchy.findMethod(String name)** metodunu çağırır. Buradaki ad değişkeni, istenen metodun imzasının tamamını içerir. **findMethod** metodu hiyerarşideki aramayı kendisi gerçekleştirir. Aranan metot bulunduğu takdirde geri döndürülür ve bu üye metodun ait olduğu hiyerarşi katılımcısını simgeleyen düğüm **private RoleListItem RoleHierarchy. theListItem** değişkeninde saklanır. **Actor** veya **Role** sınıfına ait olan **RoleInterface.executeMethod** metodu, sonraki işlem olarak bulunan üye

metodu çalıştırmak üzere **Actor** veya **Role** sınıfının **private Object execute(Object anInstance,Method aMethod,Object... parameters)** metodunu çağırır. Buradaki ilk parametre olan **anInstance** nesnesine, **RoleInterface. executeMethod** içerisinde **RoleHierarchy.theListItem** değişkeni **Object RoleInterface.getTheInstance( )** metodu ile elde edilerek atanır. **execute** metodu **Method.invoke** komutunu Bölüm 7.3.4'te anlatıldığı şekilde kullanarak bulunan üye metodu bulunan örnek için verilen parametrelerle çalıştırarak metodun geri döndüreceği sonucu kullanıcıya geri döndürür. **Actor** ve **Role** sınıflarının **memberMethod** alanı bu işlemler sırasında **findMethod** metodu tarafından geçici değişken olarak kullanılır.

Bölüm 8.1.2.4'te anlatılan ad ile üye erişimi komutlarından **bringLocalMember** ve **executeLocalMethod** metodlarının **bringMember** ve **executeMethod** metodlarından tek farkları, tüm rol hiyerarşisinin derinlemesine aranması yerine sadece metodu çalıştırılan **Actor** veya **Role** örneğinin üyelerinin aranmasıdır. Bu fark dışında **bringLocalMember** ve **bringMember** ile **executeMethod** ve **executeLocalMethod** metodlarının çalışma biçimleri tamamen aynıdır.

Ad ile üye erişimi için gerekli bilgilerin saklandığı yansıtma tabloları hakkında Bölüm 8.1.2.4'te detaylı bilgiler verilmişti. Hiyerarşi kökü için üye değişkenler ve metotlar için yansıtma tablolarının hesaplanıp hesaplanmadığı sırasıyla **Actor** sınıfının **isMemberTableInvalid** ve **isMethodTableInvalid** metotları ile öğrenilir. Eğer ilgili tablolar hesaplanmışsa bu metotlar **true** değeri geri döndürür. Bir rol nesnesinin yansıtma tablolarının hesaplanıp hesaplanmadığını öğrenmek için ise, hiyerarşide o rol nesnesini simgeleyen düğüm olan **RoleListItem** üyesinin aynı adlı metotları çalıştırılır.

8.2.6 Rol Araması, Rol Geçişi ve Ad ile Üye Erişimi İçin İyileştirmeler

Bir varlığın çalışma anında hangi rollere sahip olacağı derleme anında bilinemez. Bu nedenle bir role geçiş yapmadan önce o rolün oynanıp oynanmadığının sorgulanması yerinde olacaktır. Dolayısıyla bir rol sorgusunun ardından çoğu kez o role geçiş isteği gelecektir. Bölüm 8.2.4.1'de anlatıldığı gibi, rol varlığı kontrolü ve rol geçişi isteği geldiğinde gerçekleşen özyineli arama işlemi **RoleHierarchy.searchRole**

metodu tarafından yapılır. **searchRole** metodu ilk olarak **private String RoleHierarchy.lastHitObjStr** üyesinin içeriğinin aranan rol tipinin adı ile aynı olup olmadığını kontrol eder. **lastHitObjStr** üyesinde en son başarılı rol varlığı kontrolü veya rol geçişi isteğinin hedefi olan rolün tip adı tutulur. **private Object RoleHierarchy.lastHitObj** üyesinde ise hedef rol nesnesinin kendisi saklanır. Eğer aranan rol ile saklanan rolün tipleri aynı ise, hiyerarşide boş yere yeni bir özyineli arama işlemi yapmaktansa, doğrudan saklanan rol nesnesi geri döndürülür. Nitelikli roller söz konusu ise, aranan ve saklanan rolün niteleyicilerinin içeriklerinin de aynı olup olmadığı kontrol edilir. Eğer saklanan rol aranan rol değilse, özyineli arama gerçekleştirilir ve bulunan rolün tip adı ile rol nesnesi **lastHitObjStr** ve **lastHitObj** üyelerine atanır.

Rol varlığı kontrolü ve rol geçişi işlemleri için gerçeklenen iyileştirme mekanizmasının aynısı ad ile üye erişimi komutları için de gerçeklenmiştir. Son başarılı ad ile üye değişken erişimi isteğinin hedefi olan üyenin adı **private String RoleHierarchy.lastMemberHitStr**, üye değişkenin kendisi ise **private Object RoleHierarchy.lastMemberHit** üyesinde saklanır. Bölüm 8.2.4.2'de anlatılan derinlemesine arama işini başlatmadan önce, **RoleHierarchy.findMember** metodu istenen üyenin adı ile saklanan üye adınını karşılaştırır ve uyuşma varsa derinlemesine arama işlemi yapmayarak doğrudan saklanan üyeyi geri döndürülür. Uyuşmazlık halinde arama yapılır ve başarılı olursa bulunan üye değişkenin adı **lastMemberHitStr**, üye değişkenin kendisi ise **lastMemberHit** içerisinde saklanır. Benzer şekilde; son başarılı ad ile üye metod erişimi isteğinin hedefi olan metodun imzası **private String RoleHierarchy.lastMethodHitStr**, üye metodu simgeleyen **Method** nesnesi ise **private Method RoleHierarchy.lastMethodHit** üyesinde saklanır. **RoleHierarchy.findMethod** metodu önce aranan ve saklanan metodun imzalarını kontrol eder ve uyuşma varsa arama yapmadan doğrudan saklanan metodu geri döndürür, aksi halde aramayı yaparak yeni bulunan metodu saklar.

Rol kazanımı, rol terki, rol aktarımı, bir rolün askıya alınması veya askıdan geri alınması işlemleri rol hiyerarşisinde değişikliklere neden olacaktır. Bu gibi durumlarda rol modeli, iyileştirme mekanizması için kullanılan değişkenlerinin içeriğini **void RoleHierarchy.invalidateLastHit()** metodunu çağırarak

boşaltır. Saklanan rolün az önce terk edilen veya başka bir sahibe aktarılan rolü göstermesi gibi olasılıklar göz önüne alındığında, boşaltma işleminin gereği açığa çıkacaktır.

8.2.7 Rollerin Askıya Alınması ve Askıdan İndirilmesi

Bölüm 8.1.2.1'de anlatıldığı gibi, rollerin askıya alınması ve askıdan indirilmesi işlemleri rollerin kendi üzerinden veya hiyerarşi kökü üzerinden yürütülebilir. Rollerin askıya alınması ve askıdan indirilmesi işlemlerinde **`Actor`** ve **`Role`** sınıflarının görevi, işlemi Bölüm 8.1.2.3'te anlatılan kısıtlama yöneticisinin onayına sunmaktır. İşlem onaylanırsa hiyerarşi yöneticisinin aşağıda anlatılacak ilgili komutlarından biri çağrılır. Hiyerarşi yöneticisi bazı durumlarda rol hiyerarşisinde Bölüm 8.2.4.1'de anlatılan özyineli aramayı başlatmaya gereksinim duyar. İşlem yapılacak rolü rol hiyerarşisinde simgeleyen düğüme doğrudan erişmenin mümkün olduğu durumlarda özyineli aramaya gerek kalmaz. Rol nesnesi üzerinden yürütülen askıya alma ve askıdan indirme işlemlerinde özyineli arama gereksizdir, ancak hiyerarşi kökü üzerinden yapılan işlemlerde önce rolün hiyerarşide aranması gereklidir. Bölüm 8.1.2.1'de askıdan indirme işleminin doğrudan rol nesnesi üzerinden çalıştırılamayacağı durum açıklanmıştır.

Bir rolün o anda askıda olup olmadığı, hiyerarşi ağacında o rolü simgeleyen düğümün **`private boolean RoleListItem.suspended`** üyesinde tutulur. Bu bilgiye **`boolean RoleListItem.isSuspended()`** metodu ile ulaşılabilir. Askıda olan bir role JAWIRO rol modelinin komutları ile erişilememesi için **`Role`** sınıfının **`public`** metotları önce **`isSuspended`** metodunun geri döndürdüğü değeri sınar. Eğer sınama başarısız ise komutun geri kalanını yürütülmez.

Bir rolün doğrudan kendisi üzerinden askıya alınmasını sağlayan **`Role.suspend()`** metodu, **`void RoleHierarchy.suspend(Role r)`** metodunu çağırır. **`RoleHierarchy.suspend`** metodu öncelikle askıya alınacak rolü simgeleyen düğümün **`void RoleListItem.setSuspended( boolean suspended )`** metodunu **`true`** parametresi ile çağırarak ilgili rolü askıya aldırır. Rollerin hiyerarşik yapısı nedeniyle, askıya alınan rolün alt rolleri de askıya alınmalıdır. Bu nedenle hiyerarşi yöneticisi son olarak **`void RoleHierarchy.suspendRest( RoleList rl )`** metodunu çağırarak askıya alınacak rolün alt rollerini simgeleyen

düğümlerin de **setSuspended** metotlarını çalıştırır. Bu sırada ağaç özyineleme ile dolaşılır. Bir rol hiyerarşi kökü üzerinden askıya alınacaksa, **Actor** örneği öncelikle Bölüm 8.1.1.3'te anlatılan **as(Role r)** rol geçişi komutu ile askıya alınacak rolü bulur ve ardından yukarıda anlatıldığı şekilde, o rolün doğrudan askıya alma işlemlerini başlatır. Hiyerarşi kökününe rol nesnesinin kendisini gösteren bir işaretçi verildiği için rolün normal veya nitelikli olması fark etmeyecektir, çünkü hiyerarşi yöneticisi özyineli arama sırasında nesne eşitliğine göre karar vermektedir.

Bir rolün doğrudan kendisi üzerinden askıdan indirilmesi, yukarıda anlatılan rolün kendisi üzerinden askıya alınması işlemine çok benzer. **Role.resume()** metodu, **void RoleHierarchy.resume(Role r)** metodunu çağırır. **RoleHierarchy.resume** metodu ise bu kez önce **RoleListItem.setSuspended** metodunu **false** parametresi ile çağırır ve ardından **void RoleHierarchy.resumeRest(RoleList rl)** metodu ile alt rolleri de askıdan indirir.

Bir rolün dolaylı olarak hiyerarşi kökü üzerinden askıdan indirilmesi ile doğrudan kendisi üzerinden askıdan indirilmesi işlemleri ise birbirinden biraz daha farklıdır. Askıda olan rollere JAWIRO rol modelinin komutları ile erişilemeyeceği için, hiyerarşi kökü askıdan indirilecek rolü basit bir rol geçişi komutu ile bulamaz. Üstelik **Actor** sınıfının biri normal biri nitelikli roller için olmak üzere, Bölüm 8.1.2.1'de anlatıldığı gibi iki farklı **ResumeRole** metodu bulunmaktadır. Rol nesnesine doğrudan bir işaretçi olmadığı için özyineli aramada rol tip adının yanı sıra, nitelikli roller için niteleyicinin de kullanılması gerekecektir. **Actor.resumeRole(String)** metodu **boolean RoleHierarchy.resumeRole(String className)** metodunu, **Actor.resumeRole(String,String)** metodu ise **boolean RoleHierarchy.resumeRole(String className, String identifier)** metodunu çağırır. **RoleHierarchy.resumeRole** metotları Bölüm 8.2.4.1'de anlatılan **RoleHierarchy.searchRole** metotlarından uygun olanını çağırarak askıya alınacak rol nesnesini elde eder. Bu noktadan sonra işler rolün doğrudan askıdan indirilmesine benzer: **RoleHierarchy.resumeRole** metodu önce işlemi Bölüm 8.1.2.3'te anlatılan kısıtlama yöneticisinin onayına sunar. Onay gelirse askıya alınacak rolü simgeleyen düğüm için **RoleListItem.setSuspended(false)** çağrısı ve bunun alt düğümleri için **RoleHierarchy.resumeRest** çağrısı ile gerekli askıdan indirme işlemleri tamamlanır.

8.2.8 Danışma ve Vekalet Mekanizmaları

Danışma ve vekalet mekanizmalarının kullanımı Bölüm 8.1.2.7'de anlatılmıştı. JAWIRO'da varsayılan mekanizmanın danışma olmasının nedeni, danışma mekanizmasının hiçbir ek işleme gerek kalmadan Java tarafından desteklenmesidir. Java dilinin **`this`** işaretçisi rol geçişi işleminin ardından her zaman mesajın son alıcısını gösterecektir. Dolayısıyla vekalet mekanizmasına hiç gerek duyulmayacağı durumlarda kullanıcının ek bir işlem yapmasına gerek kalmayacaktır.

Kullanılan mekanizmaya göre rol geçişi işleminden sonra çalıştırılacak metodun ilk veya son alıcısının saklandığı **`Actor`** ve **`Role`** sınıflarının **`self`** üyesine doğru değerin atanması, rol geçiş komutlarının son adımı olarak çalıştırılır. Geçiş yapılacak rolün normal rol, nitelikli rol veya hiyerarşi kökü olmasına göre Bölüm 8.1.1.1'de anlatılan uygun bir **`as`** metodu Bölüm 8.2.4.1'de anlatıldığı üzere **`RoleHierarchy`** sınıfının **`findRole`** veya **`findAggregateRole`** metodunu çalıştırır. Geçiş yapılacak rolün bulunması halinde, geri dönen sonucun **`self`** üyesine uygun bir değer atamak üzere **`void setSelf(Object newSelf)`** metodu çalıştırılır. Eğer o anda vekalet mekanizması etkinse **`newSelf`** parametresi olarak **`this`** işaretçisi verilir, çünkü **`as`** komutunu çalıştıran nesne mesajın ilk alıcısı olacaktır. Eğer o anda danışma mekanizması etkinse, sonucun **`self`** üyesine sonucun kendisi atanır.

8.2.9 Kalıcılık

Bölüm 8.1.2.8'de JAWIRO'nun kalıcılık yeteneği detaylı olarak incelenmişti. Bu bölümde ağırlıklı olarak, hiyerarşi yöneticisinin hiyerarşinin kalıcı ortama kaydedilmesi ve kalıcı ortamdan geri yüklenmesi sırasında üstlendiği görevler üzerinde durulacaktır.

Kalıcılık özelliğinin gerçeklenmesinde Java'nın Bölüm 7.2'de anlatılan serileştirme yeteneği kullanılmıştır. Serileştirilmek istenen her nesne, **`java.io.Serializable`** arayüzünü gerçeklemelidir. Bölüm 8.2.5'te anlatılan ad ile üye erişimi yeteneği için gerekli olan **`Field`** ve **`Method`** türünden üyeler ise bu zorunluluğu karşılayamaz. Bu nedenle söz konusu üyeler **`transient`** olarak tanımlanmıştır. Aksi halde rol hiyerarşisinin kalıcı ortama saklanması sırasında oluşacak aykırı durumlar işlemi engeller.

Bir rol hiyerarşisinin iç yapısından o hiyerarşinin yöneticisi sorumlu olduğundan, hierarşinin kökü dışındaki katılımcıları tek tek kalıcı ortama kaydetmek ve hiyerarşi için bilgi dosyası oluşturmak görevini kalıcılık yöneticisi değil hiyerarşi yöneticisi üstlenmiştir. Kalıcılık yöneticisi hiyerarşi kökünü kalıcı ortama kaydettikten sonra **`boolean Actor.saveHierarchy`** metodunu çağırır. Bu metot kalıcılık yöneticisinin doğrudan hiyerarşi yöneticisine erişmesini engellemek için konulmuş olup, tek görevi **`boolean RoleHierarchy.saveRest`** metodunu çağırmaktır. Hiyerarşi yöneticisi **`saveRest`** metodu içerisinde derinlemesine arama yaparak, rol nesnelerini serileştirerek kalıcı ortama saklama ve hiyerarşinin bilgi dosyasını oluşturma işini bir arada yapar. Tek bir rol nesnesini serileştirmek ise **`int RoleHierarchy.saveRole (Role r, String fileName)`** metodunun görevidir. Nesneleri birbirinden ayırt etmek ve kalıcı ortamda saklanacakları dosyaların adlarını elde etmek için **`Object. hashCode()`** metodu kullanılır. Kalıcılık tablosunda saklamak ve bilgi dosyasının adını oluşturmak üzere, kalıcılık yöneticisi hiyerarşi yöneticisinin **`hashCode()`** metodunu çalıştırmak zorundadır. Bu iş **`String Actor.hierarchyCode( )`** metodu üzerinden yapılır.

8.3 Başarım Ölçümleri

Bu bölüm tez çalışması kapsamında gerçeklenen JAWIRO rol modelinin başarımının incelenmesine ayrılmıştır. Başarım incelemesinin amacı, JAWRO ile rollerin kullanımının bir yazılım ürününün pratik olarak kullanımını engelleyecek herhangi bir çalışma anı ek yükü getirmediğinin kanıtlanmasıdır. Diğer rol modelleri ve çeşitli donanım ortamları ile ayrıntılı olarak yapılan karşılaştırmaların amacı ise herhangi bir donanımın diğerinden daha iyi olduğunu göstermek olmayıp, JAWIRO dahil olmak üzere literatürdeki herhangi bir rol modelinin ‘en iyi çalışma anı başarımına sahip’ olarak nitelendirilemeyeceğini göstermektir.

8.3.1 Genel Bilgiler

Yapılan başarım ölçümlerinin temelinde; derinliği ile derecesi verilen dengeli ve tüm yaprakları dolu bir rol hiyerarşisi ağacı oluşturmak, ardından da hiyerarşinin üyeleri üzerinde rollerin desteklenen özelliklerini çalıştırmak yatmaktadır. Oluşturulacak ağacın küçük derece ve derinlik değerlerine karşılık hiyerarşide çok sayıda rol

nesnesi bulunacağı açıktır. Örneğin derecesi 3 ve derinliği 5 olan küçük bir hiyerarşide dahi 120 rol nesnesi barınabilir. Büyük hiyerarşilerin gerektireceği binlerce rol için ayrı rol sınıfları hazırlamak pratik olmadığından, başarım ölçümü için nitelikli roller kullanılmıştır. Normal rollerin aksine, aynı nitelikli rol tipinin örnekleri farklı niteleyicilere sahip olmaları şartıyla aynı rol hiyerarşisinde yer alabilir. Bir döngü içinde artan sayacın değerini niteleyici olarak kullanarak aynı türden istenilen sayıda nitelikli rol örneği kolayca oluşturulmuştur.

Çalışma zamanları, herbiri rollerin farklı bir özelliğinin kullanımına ayrılmış metodların başında ve sonunda öğrenilen sistem zamanlarının birbirinden çıkartmak yolu ile ölçülmüştür. Zamanı elde etmek için ise **`System.currentTimeMillis()`** metodu kullanılmıştır. Ölçüm kodları peşpeşe birden fazla kez çalıştırılmış ve bunların aritmetik ortalaması alınmıştır.

JAWIRO'nun özgün katkılarından biri olan rol varlığı kontrolü ve rol geçişindeki iyileştirmeden Bölüm 8.2.6'da söz edilmişti. Ölçüm kodu bu iyileştirmeden yararlanamayacak ve yararlanabilecek şekilde iki farklı biçimde çalıştırılmıştır. Farklı çalışma biçimlerinin betimlenmesi için, nitelikli rolleri (sınıf adı, niteleyici) şeklinde gösterelim. İyileştirmesiz çalışma şeklinde (C,1)'den sırayla (C,2), (C,3), ... rollerine geçilirken iyileştirmeli çalışma şeklinden sırayla (C,2), (C,3), ... rollerinden (C,1) rolüne geçilmektedir. Her iki çalışma şeklinde de olası rol geçişlerinin tümü yapılmıştır, ancak bu geçişlerin çalıştırma sırası anlatıldığı üzere farklıdır. Rollerin diğer özelliklerine ait işlemler de benzer şekilde koşturulmuştur. Özetle, JAWIRO'nun iyileştirme yeteneklerinin açılıp kapatılması söz konusu değildir. Önemli olan, JAWIRO'nun iyileştirme yeteneğine uygun olan veya olmayan kodun çalıştırılmasıdır.

Başarım ölçümleri biri Intel, öteki AMD işlemciye sahip iki farklı sistem kullanılarak yapılmıştır. Intel sistemde 2.8GHz Pentium 4 işlemci, i865 yongalı anakart, 512MB RAM bellek ve Java 2 SE uyarlama 5.0.4 bulunmaktadır. AMD sistemde ise Athlon XP 2500+ işlemci, nVidia nForce2 yongalı anakart, 512MB RAM bellek ve Java 2 SE uyarlama 5.0.4 bulunmaktadır.

8.3.2 Rollerin Temel Özelliklerinin JAWIRO ile Başarımı

Rollerin temel özelliklerinin başarımını ve iyileştirme mekanizmasının etkilerini irdelemek için, AMD sistemde yapılan sabit hiyerarşi derinliğindeki ölçümleri içeren Tablo 8.4 incelenebilir. Milisaniye cinsinden verilen rakamlar, tek bir işlem için geçen ortalama süreleri göstermektedir.

JAWIRO'da bir rol hiyerarşisini kurmak için önce hiyerarşinin katılımcıları oluşturulmalı, ardından rol ekleme komutları ile hiyerarşi oluşturulmalıdır. Tablo 8.4'teki 3. adım olan hiyerarşi kurma adımı, ilk iki adımın sürelerinin toplamı ile elde edilmiştir.

Tablo 8.4: JAWIRO ile rollerin temel özelliklerinin AMD sistemdeki başarımı

Hiyerarşi derinliği: 6, Hiyerarşi derecesi: 3, Deneme sayısı: 10	Çalışma Süresi (ms.)		
	İyileştirmesiz	İyileştirmeli	%fark
Hiyerarşi üyesini oluşturmak	0,034	0,013	61,60
Rol ekleme komutu	0,000	0,000	-
Hiyerarşi kurmak	0,034	0,013	61,60
Rol varlığı kontrolü (Rol araması)	0,022	0,0002	99,27
Rol geçişi	0,022	0,0003	98,61
Rol çalıştırması	0,007	0,004	39,70
Rol geçişli çalıştırma	0,053	0,005	89,90
Rol aramalı ve geçişli çalıştırma	0,084	0,009	89,79

Başarım ölçümünün adımları, kurulum zamanı ve çalışma zamanı olarak iki gruba ayrılabilir. İlk iki adıma, ya da bunların toplamı olan üçüncü adıma kurulum zamanı denilmiştir. Rol hiyerarşisi bir kez kurulduktan sonra üzerinde rollerin temel özelliklerini kullanan işlemlerin yapıldığı Tablo 8.4'ün dördüncü adımı ve sonrasına ise çalışma zamanı adı verilmiştir. Rol modelinin başarımı daha çok çalışma zamanı için önem taşır, çünkü herhangi bir uygulamada ilk kurulum işlemleri sadece bir kez yapılıp uygulama sonlandırılıncaya dek sürekli çalışma zamanı işlemleri yapılacaktır.

Tablo 8.4'teki sonuçlardan JAWIRO'nun rollerin çalıştırılmasına getirdiği ek yükün hem kurulum zamanı hem de çalışma zamanında göz ardı edilebilecek kadar küçük

olduğu görülmektedir. Özellikle rollerin doğrudan çalıştırmasının getireceği el yükün milisaniyenin yüzde birinden az olduğu ortaya çıkmıştır.

JAWIRO'nun özgün katkılarından biri olan rol araması ve geçişindeki iyileştirmenin başarıma katkısının önemli ölçüde olduğu da Tablo 8.4'te görülmektedir. Tek başına rol araması ve rol geçişi işlemlerindeki yüz kata varan iyileşme dışında, diğer işlemlerin başarımı da iki ile dokuz kat arasında iyileşmiştir. Tablo 8.4'te iyileştirmesiz ve iyileştirmeli çalışma süreleri arasındaki yüzde farklar da görülebilir.

Aynı başarım ölçüm kodunun bu kez de Intel sistemde çalıştırılması ile elde edilen sonuçlar ise Tablo 8.5'te görülebilir.

Tablo 8.5: JAWIRO ile rollerin temel özelliklerinin Intel sistemdeki başarımı

Hiyerarşi derinliği: 6, Hiyerarşi derecesi: 3, Deneme sayısı: 10	Çalışma Süresi (ms.)		
	İyileştirmesiz	İyileştirmeli	%fark
Hiyerarşi üyesini oluşturmak	0,026	0,0041	83,87
Rol ekleme komutu	0,013	0,000	100,00
Hiyerarşi kurmak	0,038	0,004	89,30
Rol varlığı kontrolü (Rol araması)	0,018	0,0003	98,41
Rol geçişi	0,018	0,0003	98,54
Rol çalıştırması	0,006	0,003	42,07
Rol geçişli çalıştırma	0,043	0,004	91,05
Rol aramalı ve geçişli çalıştırma	0,068	0,006	90,83

Tablo 8.4 ve Tablo 8.5'teki süreler karşılaştırılırsa, adımların çalışma sürelerinin sıralaması açısından AMD ve Intel sistemlerdeki başarımın paralel olduğu görülecektir. Örneğin her iki sistemde de en uzun süren işlem son adım, en kısa süren işlem ise rol çalıştırması adımıdır. Ancak Intel sistemdeki sürelerin daha düşük olduğu da görülmektedir. AMD ve Intel sistemlerin karşılaştırıldığı Tablo 8.6 incelendiğinde ise, iyileştirmesiz düzende JAWIRO'nun başarımının Intel sistemde genelde yaklaşık %20 oranında iyileştiği anlaşılacaktır. İyileştirmeli düzendeki sürelerde de Intel sistem daha başarılı olma eğilimini korumaktadır.

Tablo 8.6: AMD ve Intel sistemdeki başarımların karşılaştırılması (temel özellikler)

Hiyerarşi derinliği: 6, Hiyerarşi derecesi: 3, Deneme sayısı: 10	İyileştirmesiz			İyileştirmeli		
	AMD	Intel	%fark	AMD	Intel	%fark
Hiyerarşi üyesini oluşturmak	0,034	0,026	25,6	0,013	0,0041	68,75
Rol ekleme komutu	0,000	0,013	-	0,000	0,000	-
Hiyerarşi kurmak	0,034	0,038	-12,1	0,013	0,004	68,75
Rol varlığı kontrolü	0,022	0,018	19,05	0,0002	0,0003	-75,35
Rol geçişi	0,022	0,018	19,25	0,00031	0,00026	15,48
Rol çalıştırması	0,007	0,006	16,15	0,004	0,003	19,44
Rol geçişli çalıştırma	0,053	0,043	18,29	0,005	0,004	27,59
Rol aramalı ve geçişli çalıştırma	0,084	0,068	19,09	0,009	0,006	27,32

Tablo 8.7: Intel sistem üzerinde rollerin temel özelliklerinin başarımına hiyerarşi derinliğinin etkisi

JAWIRO ile komut başına ortalama işlem süreleri (ms.) (Hiyerarşi derecesi: 3)									
	Derinlikler:	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Adımlar	Hiyerarşi üyesini oluşturmak	0,010	0,006	0,000	0,006	0,016	0,010	0,006	0,007
	Rol ekleme komutu	0,013	0,009	0,013	0,013	0,009	0,018	0,088	0,452
	Hiyerarşi kurmak	0,009	0,008	0,013	0,019	0,024	0,028	0,094	0,459
	Rol araması	0,000	0,002	0,012	0,018	0,055	0,311	1,159	3,680
	Rol geçişi	0,000	0,000	0,000	0,018	0,054	0,309	1,175	3,654
	Rol çalıştırması	0,000	0,000	0,002	0,001	0,001	0,002	0,010	0,000
	Rol geçişli çalıştırma	0,000	0,002	0,011	0,018	0,056	0,329	1,211	3,752
	Rol aramalı ve geçişli çalıştırma	0,000	0,002	0,000	0,019	0,055	0,316	1,201	3,817
	Rollerin sayıları:	**12**	**39**	**120**	**363**	**1.092**	**3279**	**9840**	**29523**

Rollerin temel özelliklerinin JAWIRO ile kullanımının hiyerarşi derinliği ile bağıntısını araştırmak için, ölçüm kodu farklı hiyerarşi derinliklerinde çalıştırılmıştır.

İyileştirmeli çalışma düzeninin etkileri incelenmiş olduğundan, bu kez sadece iyileştirmesiz çalışma düzeni kullanılmıştır. AMD ve Intel sistemler arasındaki başarım farkı da incelendiğinden sadece Intel sistemin sonuçları verilecektir. Elde edilen sonuçlar Tablo 8.7'de görülebilir.

Daha önceki ölçümler sabit derinlikteki hiyerarşide yapılmıştır ve olası tüm işlemler çalıştırılmıştır. Örneğin 6 derinliğinde 3 dereceli tam dolu bir ağaçta kök hariç n = 363 rol nesnesi olduğundan, 363 rol varlığı kontrolü ve n^2 = 131769 rol geçişi işlemi yapılabilir. Derinliğin artması ile rol geçişi gibi komutlar için olası işlem sayısı üstel olarak artacak, bu da ölçüm kodunun çalışma süresinin aşırı uzamasına neden olacaktır. Dolayısıyla derinlik etkisinin incelendiği ölçümlerde kodun çalışma süresini makul sınırlar içerisinde tutmak için, olası tüm işlemler kümesinin rastgele seçilen bir alt kümesindeki işlemler çalıştırılmıştır.

Tablo 8.8: AMD sistem üzerinde rollerin temel özelliklerinin başarımına hiyerarşi derinliğinin etkisi

JAWIRO ile komut başına ortalama işlem süreleri (ms.) (Hiyerarşi derecesi: 3)									
	Derinlikler:	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Adımlar	Hiyerarşi üyesini oluşturmak	0,000	0,012	0,000	0,014	0,013	0,006	0,007	0,007
	Rol ekleme komutu	0,000	0,013	0,026	0,010	0,010	0,036	0,143	0,807
	Hiyerarşi kurmak	0,000	0,025	0,026	0,024	0,023	0,043	0,149	0,814
	Rol araması	0,000	0,003	0,005	0,022	0,072	0,363	1,361	4,174
	Rol geçişi	0,000	0,003	0,006	0,022	0,073	0,363	1,377	4,053
	Rol çalıştırması	0,000	0,000	0,002	0,000	0,002	0,000	0,000	0,000
	Rol geçişli çalıştırma	0,011	0,005	0,005	0,023	0,072	0,370	1,348	4,240
	Rol aramalı ve geçişli çalıştırma	0,000	0,003	0,009	0,024	0,073	0,363	1,383	4,223
	Rollerin sayıları:	**12**	**39**	**120**	**363**	**1.092**	**3279**	**9840**	**29523**

Tablo 8.7'deki verilerden hiyerarşi üyelerini oluşturmak ve rol çalıştırması işlemlerinin hiyerarşi derinliğinden etkilenmediği görülmektedir. Bu işlemlerde rol modelinin üstüne fazla iş düşmediğinden oluşan sonuç normaldir. Diğer işlemler bir

şekilde hiyerarşinin tümünün aranmasını gerektireceğinden hiyerarşideki eleman sayısı ile doğru orantılı olarak artmaktadır. Tablo 8.7'de kullanılan hiyerarşinin derecesi 3 olduğundan, hiyerarşi derinliğinin her bir artımında hiyerarşideki rol nesnesi sayısı üç kat artmaktadır. JAWIRO ile yapılan işlemlerin süresi de yaklaşık aynı oranda artmaktadır. Oluşan ek yükün belirli bir eşiğin üstüne çıktığı an kırılma noktası olarak tanımlanabilir. Eşik bir milisaniye olarak belirlenirse JAWIRO'nun kırılma noktası 9000 rol nesnesi olacaktır. Tablo 8.8'de verilen ve AMD sistem ile elde edilen rakamlar da Tablo 8.7'deki sonuçlarla benzeşmektedir.

8.3.3 Rollerin İleri Düzey Özelliklerinin JAWIRO ile Başarımı

Bu bölümde JAWIRO'nun rollerin ileri düzey özelliklerini gerçeklemedeki başarımı incelenmektedir. Önceki bölümdeki gibi, ilk ölçümler sabit derinlikte yapılacaktır.

Intel sistemde ile sabit hiyerarşi derinliğinde yapılan ölçümlerin sonuçları Tablo 8.9'da verilmiştir. Milisaniye cinsinden verilen rakamlar, tek bir işlem için geçen ortalama süreleri göstermektedir.

Tablo 8.9 incelendiğinde, rollerin ileri düzey yeteneklerinin de JAWIRO ile önemli bir ek yük olmadan gerçeklendiği görülebilir. En uzun süren işlemler dahi yarım milisaniyeyi fazla geçmemektedir. Bölüm 8.1.2.4 ve Bölüm 8.2.5'te incelenen ad ile üye erişimi yeteneğinin tüm rol hiyerarşisinde derinlemesine arama gerektirdiği göz önüne alınırsa, Tablo 8.9'daki üye ve metodlara ad ile erişim işlemlerinin en uzun süren işlemler olmasının nedeni anlaşılacaktır.

Rollerin ileri düzey özelliklerinden iyileştirmeli çalışmaya uygun olanları, Bölüm 8.2.6'da anlatıldığı üzere ad ile üye erişimi işlemleridir. Tablo 8.9'daki veriler incelendiğinde, iyileştirmeli çalışmada ad ile üye erişiminin başarımının çok büyük oranda iyileştiği görülecektir.

Tablo 8.9: JAWIRO ile rollerin ileri düzey özelliklerinin Intel sistemdeki başarımı

Hiyerarşi derinliği: 6, Hiyerarşi derecesi: 3, Deneme sayısı: 10	Çalışma Süresi (ms.)		
İşlem	İyileştirmesiz	İyileştirmeli	%fark
Yerel üye değişkene ad ile erişim	0,030	0,021	29,36
Üye değişkene ad ile erişim	0,554	0,002	99,70
Yerel üye metodu çalıştırmak	0,207	0,212	-2,40
Üye metoda ad ile erişim	0,554	0,006	98,90
Vekalet kapalı iken danışma ile metod çalıştırma	0,016	0,016	5,03
Vekalet açık iken danışma ile metod çalıştırma	0,016	0,019	-15,93
Metodu vekalet ile çalıştırma	0,025	0,016	34,73
Rolü askıya alma	0,0001	0,0000	67,86
Rolü askıdan geri alma	0,000	0,000	-
Rolü başka bir sahibe aktarma	0,628	0,613	2,37

Tablo 8.9 incelendiğinde, metot çalıştırmanın başarımının o anda kullanılan mekanizmanın vekalet veya danışma olmasından fazla etkilenmediği ortaya çıkacaktır. Bölüm 8.1.2.7 ve Bölüm 8.2.8'de incelendiği gibi, rol geçişinin ardından vekalet mekanizmasında JAWIRO mesajın ilk alıcısını **`self`** adlı üyede saklar, programcı ise bu üyeye tip dönüşümü yaparak erişir. Danışma mekanizmasında ise bu işlemlere gerek yoktur. Tablo 8.9'daki ölçüm sonuçları bu işlemlerin fazla yük getirmediğini ortaya çıkartmıştır. Danışma ve vekalet mekanizmaları ile ölçüm metodu çalıştırılırken JAWIRO'nun ad ile üye erişimi özelliği kullanılmadığı için, rol modelinin ad ile üye erişimindeki iyileştirme özelliği devreye girmemiştir. Dolayısıyla metodun vekalet ile çalıştırıldığı 7. adımdaki %34,7'lik iyileşme sadece Java'nın HotSpot adlı [44] çalışma anı derleyicisinden kaynaklanmaktadır.

Aynı başarım ölçüm kodunun bu kez de AMD sistemde çalıştırılması ile elde edilen sonuçlar ise Tablo 8.10'da görülebilir. Rollerin temel özelliklerinin aksine, JAWIRO ile rollerin ileri düzey özellikleri kullanılırken iyileştirmesiz çalışmada AMD sistem

daha iyi başarım sergilemiştir. Özellikle ad ile üye erişimi işlemlerinde %73'e varan bir başarım iyileşmesi görülmektedir. Bir başka deyişle, ad ile üye erişimi işlemlerinde AMD sistem Intel sisteme göre 3,74 kata varan oranlarda daha çabuktur.

Tablo 8.10: JAWIRO ile rollerin ileri düzey özelliklerinin AMD sistemdeki başarımı

Derinlik: 6, Derecesi: 3, Deneme: 10		Çalışma Süresi (ms.)		
Adım	İşlem	İyileştirmesiz	İyileştirmeli	%fark
1	Yerel üye değişkene ad ile erişim	0,013	0,000	100,00
2	Üye değişkene ad ile erişim	0,148	0,001	99,44
3	Yerel üye metodu çalıştırmak	0,155	0,177	-14,06
4	Üye metoda ad ile erişim	0,152	0,009	94,28
5	Vekalet kapalı iken danışma ile metod çalıştırma	0,025	0,023	7,27
6	Vekalet açık iken danışma ile metod çalıştırma	0,023	0,022	3,77
7	Metodu vekalet ile çalıştırma	0,023	0,023	0,00
8	Rolü askıya alma	0,0000	0,0000	-
9	Rolü askıdan geri alma	0,000	0,000	-
10	Rolü başka bir sahibe aktarma	0,822	0,824	-0,23

AMD ve Intel sistemlerin karşılaştırıldığı Tablo 8.11 incelendiğinde; iyileştirmesiz düzende AMD sistemin genelde daha iyi başarım sergilemesine rağmen, Intel sistemin iyileştirmeli düzende çoğu zaman AMD sistemi geçtiği görülecektir. AMD sistemi ad ile üye erişimi işlemlerinde iyileştirmesiz çalışma ile gösterdiği başarıyı, iyileştirmeli çalışmada koruyamamıştır.

Rollerin ileri düzey özelliklerinin çalışma sürelerine değişen hiyerarşi derinliğinin etkileri ise Tablo 8.12'de görülebilir. Bölüm 8.3.2'de olduğu gibi, derinliğin artması ile üstel olarak artacak olası işlem sayısının çalışma süresini kabul edilebilir sınırların üstüne çıkarmasını önlemek için, rollerin ileri düzey özellikleri de olası tüm işlemler kümesinin rastgele seçilen bir alt kümesindeki işlemler kullanılarak çalıştırılmıştır.

Tablo 8.11: AMD ve Intel sistemdeki başarımların karşılaştırılması (ileri özellikler)

Adım	İyileştirmesiz			İyileştirmeli		
	Intel	AMD	%fark	Intel	AMD	%fark
1	0,030	0,013	29,29	0,021	0,000	100,00
2	0,554	0,148	99,70	0,002	0,001	50,00
3	0,207	0,155	-2,21	0,212	0,177	16,51
4	0,554	0,152	98,90	0,006	0,009	-50,00
5	0,016	0,025	-1,58	0,016	0,023	-43,75
6	0,016	0,023	-17,77	0,019	0,022	-15,79
7	0,025	0,023	37,63	0,016	0,023	-43,75
8	0,0001	0,0000	65,85	0,000	0,000	-
9	0,000	0,000	-	0,000	0,000	-
10	0,628	0,822	2,31	0,613	0,824	-34,42

Tablo 8.12: Intel sistem üzerinde rollerin ileri düzey özelliklerinin başarımına hiyerarşi derinliğinin etkisi

JAWIRO ile komut başına ortalama işlem süreleri (ms.) (Hiyerarşi derecesi: 3)							
	Derinlikler:	3	4	5	6	7	8
Adımlar	Yerel üye değişkene ad ile erişim	0,000	0,001	0,000	0,001	0,004	0,008
	Üye değişkene ad ile erişim	0,024	0,059	0,187	0,544	1,704	5,638
	Yerel üye metodu çalıştırmak	0,005	0,002	0,007	0,017	0,046	0,159
	Üye metoda ad ile erişim	0,027	0,084	0,189	0,552	1,703	5,998
	Vekalet kapalı iken danışma ile metod çalıştırma	0,002	0,008	0,012	0,024	0,052	0,158
	Vekalet açık iken danışma ile metod çalıştırma	0,002	0,005	0,008	0,023	0,048	0,156
	Metodu vekalet ile çalıştırma	0,002	0,002	0,008	0,018	0,048	0,155
	Rolü askıya alma	0,000	0,000	0,000	0,004	0,001	0,000
	Rolü askıdan geri alma	0,000	0,000	0,000	0,000	0,002	0,001
	Rolü başka bir sahibe aktarma	0,002	0,002	0,000	0,000	0,000	0,001
	Rollerin sayıları:	**12**	**39**	**120**	**363**	**1.092**	**3279**

Tablo 8.12 incelendiğinde; rollerin askıya alınması ve askıdan indirilmesi işlemleri ile rolün bir başka sahibe aktarımı işlemlerinin hiyerarşi derinliğinden bağımsız olarak çok kısa sürelerde gerçekleştiği görülebilir. Rollerin hiyerarşik yapısının ağaç şeklinde gösterimi, bu işlemlerin çok az sayıda işaretçi değişimi ile yapılabilmesini sağladığı için başarım da yüksek çıkmaktadır. Üye değişken veya metoda ad ile erişim komutlarının çalışma süreleri ise, hiyerarşi derinliğinin altıdan yediye çıkması ile önemli oranda uzamaktadır. Bu nedenle ad ile üye erişimi işlemleri için kırılma noktası, 7 hiyerarşi derinliği olarak ele alınabilir. Geri kalan işlemler 8 hiyerarşi derinliğinde bile 0,2 milisaniyenin altında tamamlanmaktadır.

Hiyerarşi derinliğinin JAWIRO'nun rollerin ileri düzey özelliklerini çalıştırmadaki başarımını ölçen kod AMD sistemde çalıştırıldığında elde edilen sonuçlar ise Tablo 8.13'te verilmiştir ve Intel sistemde elde edilen sonuçlar ile paralellik göstermektedir.

Tablo 8.13: AMD sistem üzerinde rollerin ileri düzey özelliklerinin başarımına hiyerarşi derinliğinin etkisi

JAWIRO ile komut başına ortalama işlem süreleri (ms.) (Hiyerarşi derecesi: 3)							
	Derinlikler:	3	4	5	6	7	8
Adımlar	Yerel üye değişkene ad ile erişim	0,000	0,005	0,000	0,001	0,004	0,008
	Üye değişkene ad ile erişim	0,024	0,059	0,187	0,544	1,704	5,638
	Yerel üye metodu çalıştırmak	0,002	0,005	0,007	0,017	0,046	0,159
	Üye metoda ad ile erişim	0,027	0,084	0,189	0,552	1,703	5,998
	Vekalet kapalı iken danışma ile metod çalıştırma	0,002	0,008	0,012	0,024	0,052	0,158
	Vekalet açık iken danışma ile metod çalıştırma	0,002	0,005	0,008	0,023	0,048	0,156
	Metodu vekalet ile çalıştırma	0,002	0,002	0,008	0,018	0,048	0,155
	Rolü askıya alma	0,000	0,000	0,000	0,002	0,001	0,000
	Rolü askıdan geri alma	0,000	0,000	0,000	0,000	0,002	0,001
	Rolü başka bir sahibe aktarma	0,002	0,002	0,000	0,000	0,000	0,001
	Rollerin sayıları:	**12**	**39**	**120**	**363**	**1.092**	**3279**

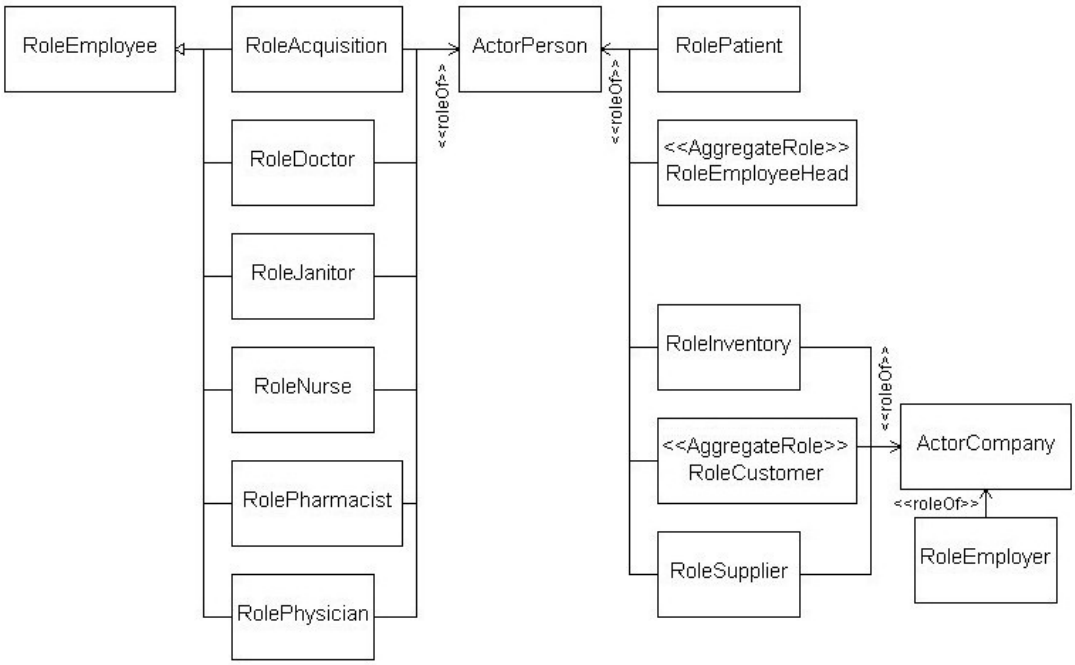
8.4 Örnek Uygulama

Bu bölümde, rollerin çeşitli özelliklerini JAWIRO ile kullanan bir uygulama programı incelenecektir. Bu bölümün odağı programın tanıtılması değil, dinamik bir sistemin JAWIRO kullanılarak rollerin yardımıyla nasıl modellenebileceğidir.

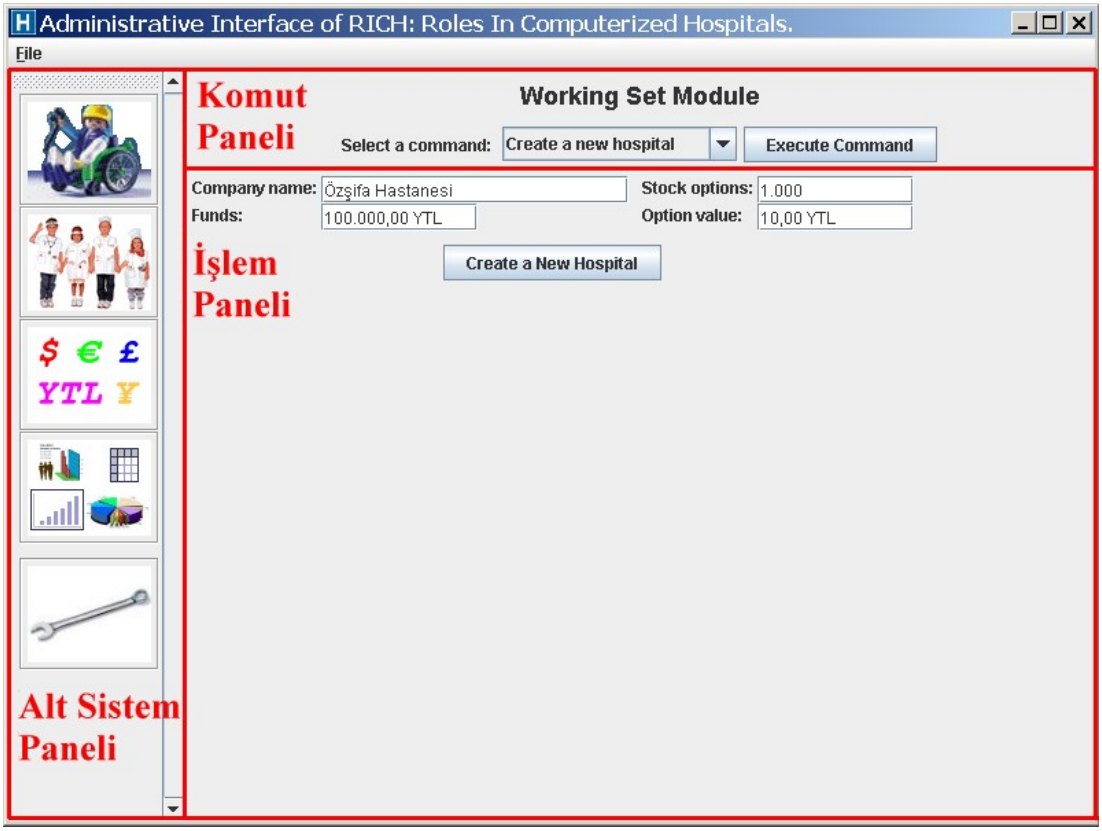
8.4.1 Modellenen Sistem

İçinde insan öğesi olan sistemler, çoğu zaman dinamik sistemler oluşturur. İnsan doğasının zamanla değişen ve evrimleşen dinamik yapısı, bu özelliğini içinde bulunduğu sisteme de yansıtır. Bir insan herhangi bir anda, çeşitli ortak alanlara ait olmaları nedeniyle birbirleriyle ilgili olan, çeşitli yükümlülükleri yerine getirir. Bu tür ortamların her biri, roller için örnek uygulama alanları oluşturur. Tez çalışması kapsamında örnek bir de uygulama geliştirmek için, böylesi bir alan olan hastane ortamı seçilmiştir. Örnek uygulama olarak basit bir hastane otomasyon sistemi hazırlanmıştır. Gerçeklenen sistemdeki sınıf ve nesne düzeyi kalıtım ilişkileri, Şekil 8.27'deki UML şemasında görülebilir. Uygulamaya RICH adı verilmiştir.

Şekil 8.27'de gösterildiği gibi; uygulamada etkin olan rol hiyerarşileri, kök olarak **`ActorPerson`** veya **`ActorCompany`** örneğine sahip olmalarına göre ikiye ayrılabilir. Bunlara sıra ile kişi hiyerarşisi ve şirket hiyerarşisi adı verilmiştir. Otomasyon sisteminin kurulduğu şirket ve uygulama içerisinde kullanıcı tarafından alım satım işlemleri yapmak amacıyla tanıtılmış diğer şirketlerin tümü, ayrı şirket hiyerarşileri ile modellenmektedir. Hastanedeki çalışanlar ve hastalar ise kişi hiyerarşileri ile modellenmektedir.



Şekil 8.27: Örnek uygulamadaki sınıf ve nesne düzeyi kalıtım ilişkileri



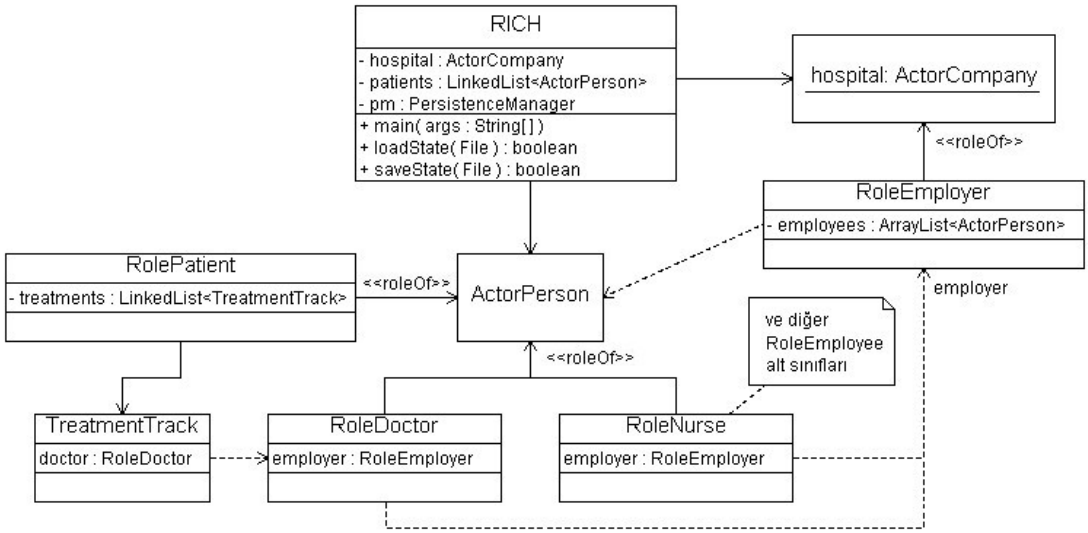
Şekil 8.28: Örnek uygulamanın kullanıcı arayüzü

8.4.2 Kullanıcı Arayüzü

Örnek uygulama amaca yönelik alt sistemlerden oluşmuştur. Uygulamanın kullanıcı arayüzü, Şekil 8.28'de işaretlendiği şekilde üç ana kısımdan oluşmaktadır. Sol sütun, üzerine tıklanınca her biri başka bir alt sisteme geçişi sağlayan beş düğme içermektedir. Bunlar yukarıdan aşağıya sırasıyla hasta, çalışan, satın alma, istatistik ve çalışma kümesi alt sistemleridir. Pencerenin sağ üstündeki şerit, kullanıcının geçiş yaptığı alt sistemle ilgili komutların listesini veren komut panelidir. Komut panelinin altında bulunan işlem paneli ise, seçilen komut ile ilgili öğeler içerir.

8.4.3 Gerçekleme Detayları

Örnek uygulamanın doktorlar ve hastalar arasındaki ilişkiler ile, şirketler ve kişiler arasındaki alım satım işlemleri olmak üzere iki odağı bulunmaktadır. İlk odak olan doktor ve hasta ilişkilerinin gerçeklemesi ile ilgili UML şeması Şekil 8.29'da görülebilir.



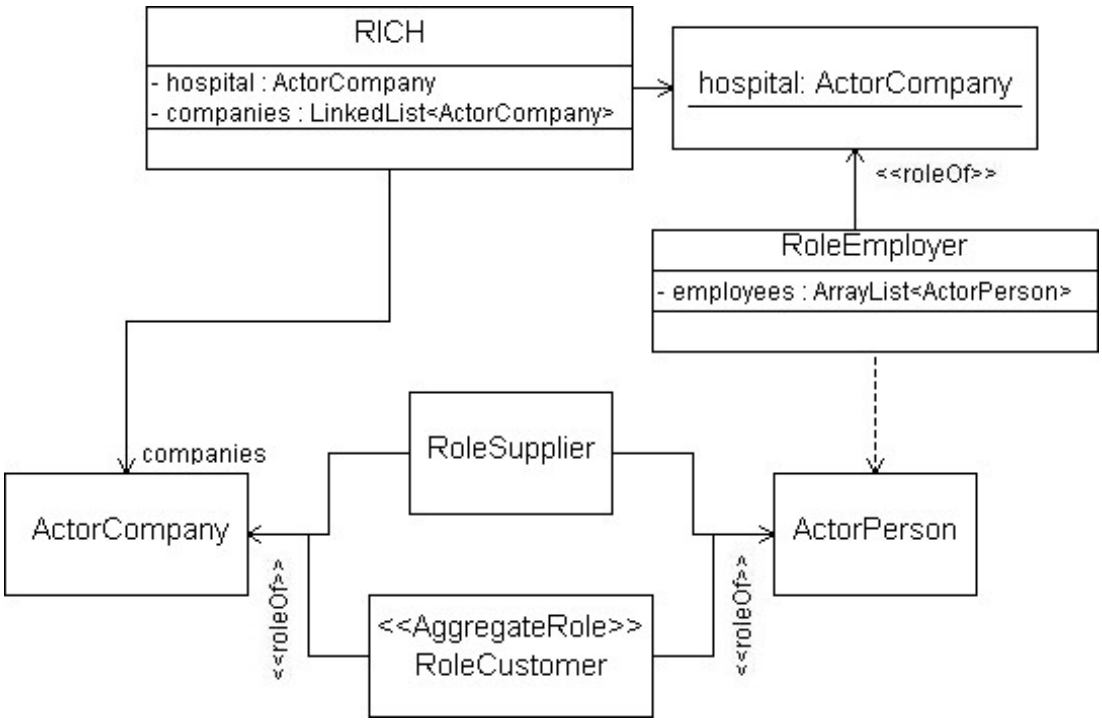
Şekil 8.29: Örnek uygulamada doktor ve hasta ilişkileri

Uygulamanın ana sınıfı **`RICH`** adını taşımaktadır. **`RICH`** sınıfı hasta rolüne sahip kişilerin hiyerarşilerini doğrudan saklamaktadır. Hastane çalışanları ise **`ActorCompany RICH.hospital`** nesnesinin kökü olduğu şirket hiyerarşisinin bir katılımcısı olan ve işveren rolünü modelleyen **`RoleEmployer`** örneğinin **`private ArrayList<ActorPerson> employees`** üyesinde saklamaktadır. Çalışanlar da işvereninden haberdardır.

Hastalar doktorlarından doğrudan haberdar değildir. Bir hastanın zaman içinde geçirdiği tedaviler **`private RolePatient.LinkedList<TreatmentTrack> treatments`** üyesinde saklanır. Hasta, bu üye üstünden hangi tedavisini hangi doktorun üstlendiğini bilebilir.

Uygulamanın o anki durumu istenildiği zaman **`public boolean RICH.saveState(File f)`** metodu ile kalıcı ortamda saklanabilir. Bu metot içerisinde uygulamanın kalıcılık yöneticisi olan **`private PersistenceManager RICH.pm`** üyesi kullanılarak tüm kişi ve şirket hiyerarşileri kalıcı ortama kaydedilir. Uygulamanın kalıcı ortamda saklanmış olan son durumu, uygulama yeniden çalıştırıldığında **`public boolean RICH.loadState(File f)`** metodu ile yine uygulamanın kalıcılık yöneticisi kullanılarak geri yüklenebilir.

Uygulamanın ikinci odak noktası olan şirketler ve kişiler arasındaki alım satım işlemlerinin gerçeklemesi ile ilgili UML şeması ise Şekil 8.30'da verilmiştir.



Şekil 8.30: Örnek uygulamada alım satım ilişkileri hakkında özet bilgi

Alım ve satım işlemleri, hem şirketler hem de çalışanlar tarafından yapılabilir. Hastane çalışanlarının **`hospital`** nesnesinin kökü olduğu şirket hiyerarşisinin işveren rolünde saklandığı belirtilmişti. Şirketler ise **`private LinkedList <ActorCompany> RICH.companies`** üyesinde saklanmaktadır.

8.4.4 Rollerin Kullanımı

Bu bölümde örnek uygulamanın, rollerin çeşitli özelliklerinden yararlanarak gerçeklenen işlemleri anlatılacaktır. Alt bölümler rollerin kullanılan özelliklerine göre oluşturulmuştur.

8.4.4.1 Rollerin Askıya Alınması ve Askıdan İndirilmesi

İlk olarak doktor ve hasta rolleri arasındaki ilişkiye göz atalım. Hastaneye gelen her hasta için yeni bir **`ActorPerson`** örneği oluşturulup buna yeni bir **`RolePatient`** rolü eklenir. Hastanede çalışmaya başlayan doktorlar için de yeni bir **`ActorPerson`** örneği oluşturulup buna yeni bir **`RoleDoctor`** rolü eklenir. Gerçek dünyada son derece olası bir senaryo olan bir doktorun hastalanması hali, örnek uygulamada rollerin askıya alma ve askıdan indirilmesi özellikleri kullanılarak gerçeklenmiştir.

Örnek uygulamanın çalışan modülündeki bir komutla, bir doktor için hastalık izini alma işlemi yapılabilir. Öncelikle uygulamanın işlem panelinde bu doktorun o anda tedavi ettiği hastaları listelenir. Kullanıcı bu hastalara başka birer doktor atadıktan sonra, hasta olan doktoru izine ayırma işlemini başlatacak düğmeyi tıklar. Bu anda tetiklenen eylem önce tüm hastaların başka doktorlara aktarılıp aktarılmadığını kontrol eder. Halihazırda işlem panelindeki GUI bileşenleri gerekli bilgilerle doldurulmuş olduğundan, bu kontrol geleneksel form doğrulama metodu ile yapılmıştır. Kontrol başarılı ise, öncelikle kişinin doktor rolü askıya alınır. Eğer doktor ilk kez hastalık izni almışsa kişiye yeni oluşturulan bir hasta rolü eklenir. Aksi halde, kişinin önceki hastalıklarını saklayan ve askıda bulunan hasta rolü askıdan geri alınır. Anlatılan işlemleri yapan kod Şekil 8.31'de verilmiştir. Bu andan itibaren kişinin doktor rolüne erişilemez, ancak hasta alt sisteminden bu kişiye sıradan bir hasta gibi erişilebilir.

```
ActorPerson person; RolePatient patient;
person = (ActorPerson)sickDoctor.getActor();
sickDoctor.suspend();
if( !sickDoctorPerson.resumeRole("demoApp.RolePatient") ) {
  /*doktor önceden hiç hasta olmamışsa askıdan indirilecek bir
  hasta rolü bulamayan resumeRole komutu false döndürür ...*/
  patient = new RolePatient( );
  person.addRole( patient );
  /*... ve bu yüzden kişiye ilk kez hasta rolü eklenir.*/ }
```

Şekil 8.31: Bir doktorun hastalanmasını modelleyen kod

Bir doktorun daha önce hasta olup olmadığı, Şekil 8.31'de verilmiş koddaki **`resumeRole`** komutunun geri döndürdüğü değer ile anlaşılır. Eğer kişi daha önce hastalanmamışsa, hasta rolü askıya hiç alınmamış demektir ve bu yüzden **`resumeRole`** komutu geriye **`false`** değeri döndürecektir.

Hastalanmış doktorun hastalığı tedavi edildiğinde, bir başka deyişle açık olan tüm tedavi işlemleri tamamlandığında, kişinin hasta rolü askıya alınıp doktor rolü askıdan geri alınır. Bu işlemi yapan kod parçası Şekil 8.32'de verilmiştir. İşlemin ardından kişinin doktor rolü eskisi gibi kullanılabilir.

```
/*Bu blok Kişi kaydı güncelleme kodunun bir parçasıdır.*/
if( person.canSwitch("demoApp.RolePatient") ) {
  /*Kişi hasta rolü oynuyorsa önce hasta bilgilerini
  güncelleyen komutları işle. Sonra hastanın süregelen
  tedavilerinin sayısını kontrol et ...*/
  if( openTrackCount == 0 ) {
    /*Süregelen tedavi kalmamışsa*/
    if( person.resumeRole("demoApp.RoleDoctor") ) {
      /*Eğer bu kişi önceden doktor olup askıdan
      indirilebilecek */
      person.suspendRole("demoApp.RolePatient");
    }
  }
}
```

Şekil 8.32: Hasta doktorun iyileşmesini modelleyen kod

8.4.4.2 Aykırı Rol İlişkilerinin Önlenmesi

Aykırı rol ilişkilerini önlemek üzere, **`RoleDoctor`** rolü içerecek şekilde oluşturulan her rol hiyerarşisine bir **`ConstraintDoctor`** örneği olan kısıtlama yöneticisi atanmıştır. Kısıtlama yöneticisinin amacı, doktor olarak oluşturulmuş çalışanların kayıtları güncellenirken, bu kişilerin başka bir çalışan tipine dönüştürülmelerini önlemektir. Doktorlar ve hastaları arasındaki sıkı ilişki, bu kısıtlamayı zorunlu kılmaktadır. Diğer çalışan tipleri için bu kısıtlama gereksizdir; bir çalışan yanlışlıkla hemşire yerine temizlikçi olarak işe başlatılmışsa bu hatanın düzeltilmesine izin verilmektedir. Söz konusu kısıtlamayı sağlayan kod Şekil 8.33'de görülebilir.

Hasta izini alacak doktorun tüm hastalarının başka doktorlara aktarılıp aktarılmadığı, form doğrulama kodu yerine bir kısıtlama yöneticisinin **`approveSuspend`** metodu içinde de kontrol edilebilirdi. Kısıtlama yöneticisi çözümünün form doğrulama çözümüne tercih edilmemesinin iki sebebi vardır: Öncelikle GUI kodu ile rol

hiyerarşilerinin üyelerinin kodu birbirinden ayrılmak istenmiştir. Bu sayede kullanıcı arayüzündeki değişikliklerin uygulama mantığını etkilemesi önlenmiştir. İkinci neden ise başarımdır: Kısıtlama yöneticisi çözümünde **`RICH`** sınıfının o anda koşturulan örneğinin yöneticiye atanması ve **`RICH.patients`** üyesinde ardışıl tarama yapılması gerekecektir. Bu da form doğrulama işleminden daha uzun sürecektir. Kontrolün kısıtlama yöneticisi yerine **`RolePatient.suspend`** metodu yeniden tanımlanması ile yapılması için de aynı değerlendirme geçerlidir.

```
public boolean approveResign( String parentClassName,
                                    String childClassName ) {
  if( childClassName.compareToIgnoreCase
                          ("demoApp.roleDoctor") == 0 )
      return false;
  return true;
}
```

Şekil 8.33: Doktor rolünün terkinin önlenmesi

Kısıtlama yöneticisi, rollerin bir başka sahibe aktarımı yeteneğinden de yararlanarak, örnek uygulamada bir başka amaçla daha kullanılmıştır. Bu amaç Bölüm 8.4.4.3'te anlatılacaktır.

8.4.4.3 Rol Aktarımı, Nitelikli Roller ve Kısıtlama Yöneticisi

Hastane çalışanları daha yüksek sorumluluklar almak üzere terfi ettirilebilir. **`RoleEmployeeHead`** rolü bu amaçla sisteme eklenmiştir. Doktorlar için bu rol baş hekimliğe benzetilebilir. Bir doktor hasta olduğunda baş hekim rolünü bir başkasına devretmelidir. Bu devir rol aktarımı ile yapılmaktadır. Her doktorun **`private RoleDoctor RoleDoctor.substitute`** üyesinde tutulan bir yedeği bulunur. Baş hekim rolü bu yedek doktora aktarılmaktadır. Aktarım işleminin otomatik olarak yapılması için **`ConstraintDoctor`** sınıfı ile gerçeklenen ve yalnızca doktor olarak sisteme girilen kişilerin rol hiyerarşilerine atanan bir kısıtlama yöneticisi kullanılmıştır. Bölüm 8.4.4.1'de anlatıldığı üzere hasta olan doktorun doktor rolü askıya alınacağı ve doktor iyileşince doktor rolünün askıdan indirileceği için, **`ConstraintDoctor`** sınıfının **`approveSuspend`** ve **`approveResume`** metotları baş hekim rolünün otomatik aktarımı kodu için en uygun yer olacaktır.

Sistemde belli bir anda sadece bir baş hekim olması durumunda aktarım işini kodlamak çok kolay olacaktı. Ancak modellenen sistemde birden fazla baş hekimin

olması, baş hekim rolünün devri işlemleri ile ilgili zorluklar çıkartacaktır. İlk zorluk, kendisi baş hekim olmamasına rağmen iki farklı baş hekimin birden yedeği olan bir doktordur. İki baş hekim birden hasta olduğu zaman, yedek doktor aynı tip rolün birden fazla örneğini oynamak zorunda kalacaktır. Bu nedenle **`RoleEmployeHead`** rolü nitelikli rol olarak gerçeklenmiştir. Hastalanan doktorun ve onun yedeğinin baş hekim olması durumu da birinciye benzer şekilde ikinci zorluğu oluşturur: Yedek doktorun hem kendi baş hekimlik rolünü hem de hastalanan doktordan aktarılan baş hekimlik rolünü birden yürütebilmesi için yine nitelikli rol kullanılmalıdır. Baş hekim rolünün devrinin otomatik olarak yapılmasını sağlayan kod Şekil 8.34'te görülebilir.

```
public boolean approveSuspend( String parentClassName,
                                    String childClassName ) {
  RoleEmployeeHead headRole;
  RoleDoctor substitute;
  if( childClassName.compareTo("demoApp.RoleDoctor") == 0 ) {
       substitute = doctor.getSubstitute();
       if( substitute == null )
            return false;
       while( doctor.canSwitch("demoApp.RoleEmployeeHead") ) {
            headRole = (RoleEmployeeHead)doctor.as
                              ("demoApp.RoleEmployeeHead");
            headRole.transfer( substitute.getActor() );
       }
  }
  return true;
}
```

Şekil 8.34: Kısıtlama yöneticisi ile baş hekim rollerinin otomatik olarak yedek doktora aktarılması

Önceden değinilen iki zorluk nedeniyle hastalık iznine ayrılacak doktorda iki baş hekim rolü olabileceği açıktır. Dikkat edildiğinde baş hekim rolünün devri işlemleri ile ilgili üçüncü bir zorluk daha görülecektir: Geçici olarak baş hekim rolünü üstlenen doktor da hastalanabilir. Bu durumda baş hekimlik rolü asıl baş hekimin yedeğinin yedeğine aktarılacaktır. Anlatılan üç zorluk bir araya geldiğinde, hastalık iznine ayrılacak doktorda üç veya daha fazla baş hekim rolü bulunması olaslığı ortaya çıkar. Bu nedenle Şekil 8.34'te bir **`while`** döngüsü içerisinde hastalanan doktordaki tüm baş hekim rollerinin aktarımı sağlanmıştır. Nitelikli role normal roller için olan komutla geçiş yapımaya çalışılırsa, o nitelikli rol tipinin hiyerarşide bulunan ilk örneği geri döndürülecektir. Bu sayede hastalanan doktorun tüm baş hekim rollerinin aktarımı mümkündür.

Hastalanan doktor iyileştiği zaman, doktorun önceden kaybettiği baş hekim rolünü otomatik olarak geri kazanmasını sağlayan kod Şekil 8.35'te görülebilir. Bir doktor hastalıktan kurtulunca diğer doktorlar, aslında hastalık izninden yeni dönen doktora ait olan baş hekim rolünü oynayıp oynamadıkları hakkında sorgulanır. Geri dönen doktora ait baş hekim rolü bulunursa rol eski sahibine geri aktarılır. Bu çalışma biçimi, baş hekim rolünün devri işlemleri ile ilgili değinilen tüm zorlukların üstesinden gelecektir.

```
public boolean approveResume( String parentClassName,
                                        String childClassName ) {
   RoleEmployeeHead headRole;
   RoleDoctor substituteDoctor;
   ActorPerson originalHead;
   if( childClassName.compareTo("demoApp.RoleDoctor") == 0 ) {
        substituteDoctor = doctor.getSubstitute();
        while( substituteDoctor != null &&
                     substituteDoctor != doctor ) {
            if( substituteDoctor.canSwitch("demoApp.RoleEmployeeHead"
                      ,person.getName()) ) {
                  headRole = (RoleEmployeeHead)substituteDoctor.as
                        ("demoApp.RoleEmployeeHead",person.getName());
                  originalHead = headRole.getOriginalHead();
                  headRole.transfer( originalHead );
                  return true;
            }
            substituteDoctor = substituteDoctor.getSubstitute();
        }
   }
   return true;
}
```

Şekil 8.35: Kısıtlama yöneticisi ile hastalıktan dönen doktora baş hekim rolünün geri aktarılması

Bu bölümde anlatılan işleri yapabilmek için, **`ConstraintDoctor`** sınıfında **`private RoleDoctor doctor`** ve **`private ActorPerson person`** üyeleri bulunmaktadır. Bu üyelere, Bölüm 8.1.2.3'te anlatıldığı şekilde, **`setActor`** metodu içinde değer atanır. Şekil 8.36'da bu işlem gösterilmiştir. Diğer bir yaklaşım ise, Bölüm 8.1.2.3'te anlatıldığı gibi, kişinin doktor rolüne askıda iken bile erişilmesine izin verilmesidir. Bu durumda **`doctor`** üyesine önceden değer atanması gerekmeyip, **`RoleDoctor`** sınıfının kurucu metodunda **`setReadableWhileSuspended (true)`** komutunu vermek gerekecekti.

Hiyerarşi kökünü gösteren bir işaretçi hazırda sağlanmışken bir de hiyerarşideki doktor rolünü gösteren diğer bir işaretçi olan **`doctor`** üyesinin saklanması, ilk

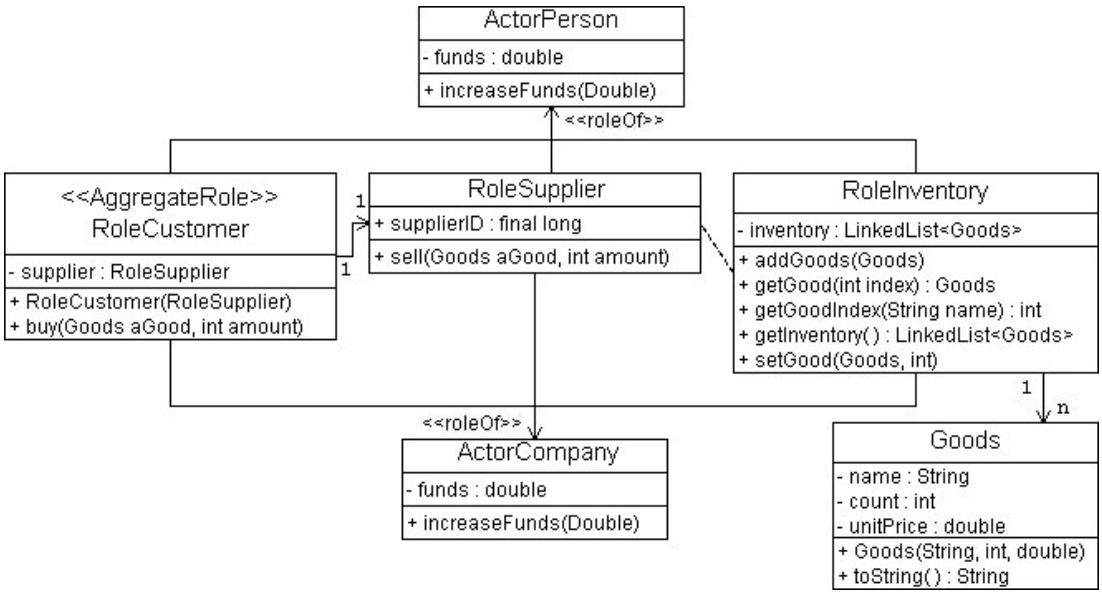
bakışta gereksiz görülebilir. Ancak **`ConstraintDoctor`**.**`approveResume`** metodu içerisinde olağan dışı işlem oluşmaması için **`doctor`** üyesi gereklidir. Bu metot içerisinde doktor rolünün **`getSusbtitute`** metodu kullanılarak hastalanan doktorun yedeğinin elde edilmesi gerekmektedir. Halbuki o anda henüz doktor rolü askıdan geri alınmamıştır, sadece işlem onaylanmaya çalışılmaktadır. Askıda bulunan bir role geçiş yapılması ise olağan dışı bir durum ve bu nedenle izin verilmeyen bir işlemdir. Dolayısıyla kişinin doktor rolüne, rol askıya alınmadan önce **`setActor`** metodu içerisinde bir işaretçi elde edilmelidir.

```
public void setActor( Actor anActor ) {
  person = (ActorPerson)anActor;
  if( anActor.canSwitch("demoApp.RoleDoctor") )
    doctor = (RoleDoctor)anActor.as("demoApp.RoleDoctor"); }
```

Şekil 8.36: Olağan dışı işlemlere yol açmamak için önceden rol geçişi yapılması

8.4.4.4 Nesne Düzeyinde Çoklu Kalıtım, Nitelikli Roller ve Ad İle Üye Erişimi

Modellenen sistemde şirketler ve kişiler birbirleri arasında hiçbir kısıtlama olmadan mal alıp satabilmektedirler. Başka bir deyişle satıcı ve müşteri rolleri, hem insanlar hem de şirketler tarafından oynanabilmektedir. Böylesi bir gereksinim, ancak nesne düzeyinde çoklu kalıtım ile karşılanabilir. Şirketler ve kişiler arasındaki alım satım işlemlerinin gerçeklemesi ile ilgili detaylı bilgi veren UML şeması Şekil 8.37'de görülebilir.



Şekil 8.37: Örnek uygulamada alım satım ilişkileri hakkında detaylı bilgi

Modellenen sistemde aslında üç farklı nesne düzeyinde çoklu kalıtım ilişkisi söz konusudur. Bunlar;

1. Müşterileri modelleyen nitelikli rol sınıfı olan **`RoleCustomer`** örneklerinin hem **`ActorPerson`** hem de **`ActorCompany`** örnekleri tarafından oynanabilmesi,

2. Satıcıları modelleyen **`RoleSupplier`** örneklerinin hem **`ActorPerson`** hem de **`ActorCompany`** örnekleri tarafından oynanabilmesi,

3. Kişi veya şahısların sahip olduğu malları modelleyen **`RoleInventory`** örneklerinin hem **`ActorPerson`** hem de **`ActorCompany`** örnekleri tarafından oynanabilmesidir.

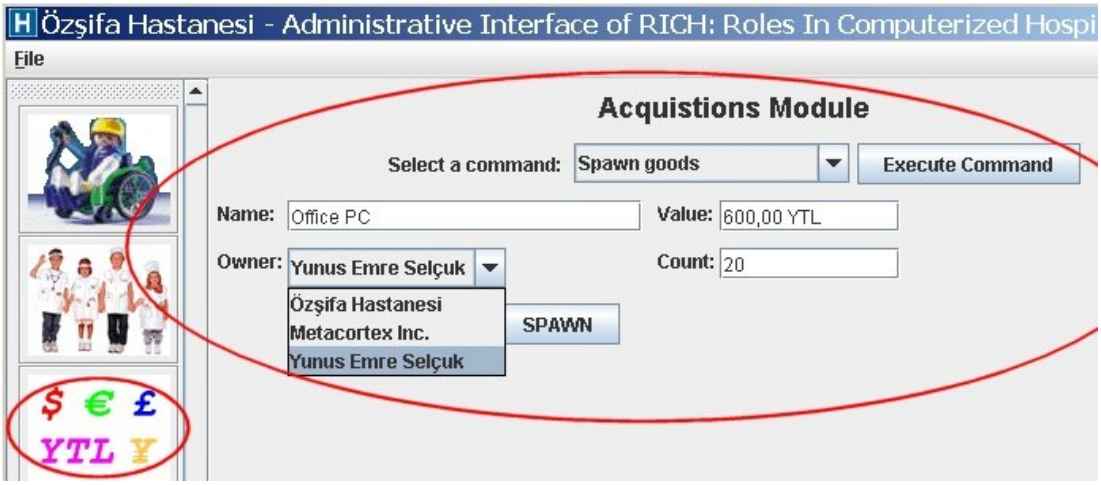
Sistemde alım-satımı yapılan malzemeler **`Goods`** sınıfı ile modellenmiştir. İnsan veya şirket olan bir varlığın sahip olduğu mallar ise **`RoleInventory`** rolünde bir bağlı liste veri yapısı olan ve envanteri modelleyen **`private LinkedList<Goods> inventory`** üyesinde tutulur. Uygulamada satıcı rolü verilen her varlığa mutlaka **`RoleInventory`** rolü de verilmektedir. Gerek RICH uygulamasının hedefi olan hastane olsun, gerekse kullanıcının sonradan tanımladığı şirketler olsun, tüm şirketler kendi hisse senetlerini simgeleyen en az bir tür mala sahiptir. Bu mal da envanter içerisinde saklanacağından her **`ActorCompany`** örneği mutlaka **`RoleInventory`** rolünü oynar. Sistemde her şirket için ayrı ayrı olmak üzere hisse senetlerini simgeleyenler dışında, önceden tanımlanmış hiç bir mal kalemi yoktur. Kullanıcı istediği ada, birim fiyata ve miktara sahip yeni mal kalemleri oluşturup bir varlığa atayabilir.

Bir alıcıya satış yapacak olan her varlık **`RoleSupplier`** rolünü oynamalıdır. Şirketler yukarıda anlatıldığı üzere zaten envanter defterine sahip olduklarından aynı zamanda satıcı rolüne de sahiplerdir. Bir insan ise ya bir satıcıdan bir malzeme aldığı zaman, ya da kullanıcı tarafından yeni bir mal kalemi oluşturulup kendisine atandığı zaman satıcı rolünü kazanmaktadır.

Bir varlık, birlikte çalıştığı her satıcı için başka bir **`RoleCustomer`** nitelikli rolü örneğine sahiptir. Aynı varlığın birden fazla müşteri rolünü birbirinden ayırt edecek niteleyici olarak, o müşteri rolünün birlikte çalıştığı satıcının **`supplierID`** üyesinden türetilen karakter katarı kullanılmıştır. Alım-satım işlemini, ilgili müşteri

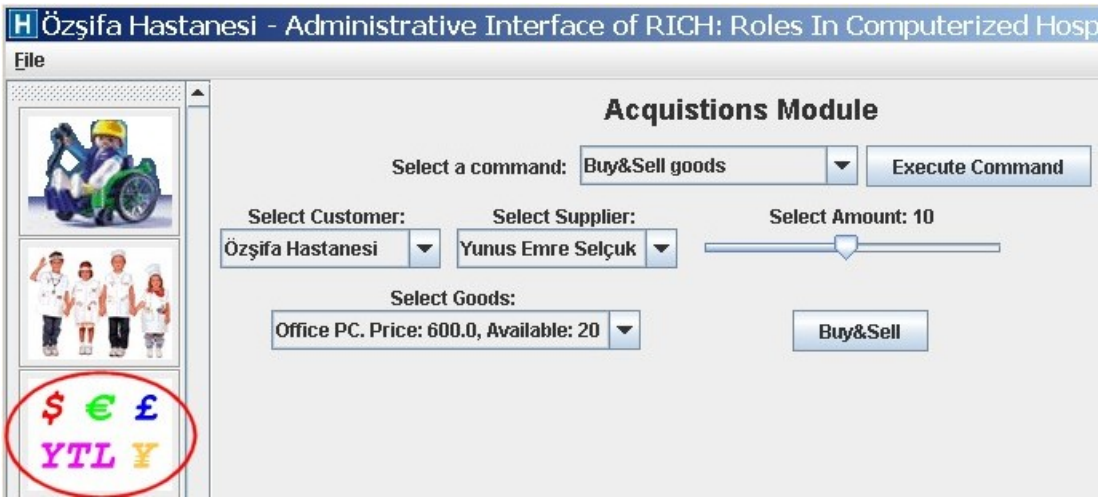
rolünün **`public boolean RoleCustomer.buy(Goods aGood,int amount)`** metodu başlatır. Fakat önce kullanıcı şu adımları grafik arayüz üzerinden yürütmelidir:

1. Bir şirketin hisse senedi dışında bir mal alım-satımı yapılacaksa bu mal, satın alma alt sisteminden "Mal oluştur/Spawn Goods" komutu seçilerek, herhangi bir şirkete veya kişiye atanmak üzere oluşturulmalıdır. Bu işlemin yapıldığı pencere Şekil 8.38'de görülebilir. Mal sahibi listesini dolduran öğeler hastaneyi simgeleyen **`Company`** örneği, kullanıcı tarafından oluşturulmuş şirketler ve hastanenin çalışanlarıdır. Mal önceden envanter sahibi olmayan bir kişiye atanırsa, o kişi otomatik olarak **`RoleInventory`** rolünü kazanır.



Şekil 8.38: Mal oluşturma ekranı

2. Satın alma alt sisteminden "Mal Alım-Satımı/Buy&Sell Goods" komutu seçilir. Bu işlemin yapıldığı pencere Şekil 8.39'da görülebilir. Burada kullanıcı yapmak istediği işlemi anlatan seçimleri yapar. Satıcı listesinden bir satıcı seçildiğinde, altındaki mallar listesinde otomatik olarak o satıcının sahip olduğu mallar gösterilir. Kullanıcı müşteriyi ve kaç adet mal satılacağını da seçtikten sonra sağ alttaki düğmeye basarak asıl alım-satım işlemini başlatır.



Şekil 8.39: Mal alım-satım ekranı

Asıl mal alım-satım işlemini başlatmadan önce uygulamanın yaptığı işlemler şunlardır:

1. Satıcı listesinden seçilen varlık zaten satıcı rolüne sahip değilse, o varlığa satıcı rolü kazandırılır. Aksi halde satış yapacak varlığın satıcı rolüne geçilir.

2. Alıcı listesinden seçilen varlık, ilk adımda elde edilen satıcı rolü ile daha önceden işlem yapmamışsa, alıcı varlığa seçilen satıcının adını niteleyici olarak kullanan yeni bir nitelikli müşteri rolü eklenir. Aksi halde ilgili müşteri rolüne geçilir.

3. Müşteri zaten bir envanter defterine sahip değilse, ya da başka bir deyişle **`RoleInventory`** rolünü oynamıyorsa, müşteriye yeni bir **`RoleInventory`** rolü eklenir.

4. Müşterinin **`buy`** metodu çalıştırılır. Bu ana dek anlatılan işlemleri yapan kod Şekil 8.40'ta görülebilir.

Bölüm 8.1.2.6'da anlatıldığı üzere, nesne düzeyi çoklu kalıtım kullanıldığında ortaya çıkabilecek tipleme belirsizliğini çözmek için JAWIRO'nun ad ile üye erişimi yeteneği kullanılabilir. Bu örnekte ilk bakışta tipleme belirsizliği yok gibidir: Alım-satım işleminin odağı olan malın aktarımı işlemleri şirket veya insan olabilecek farklı tip köklerde değil, bu iki farklı tipin de oynayabileceği başka bir ortak rolde gerçeklenmiştir. Fakat işlemin bir de parasal boyutu vardır: Alım-satım işleminin sonucunda alıcının bütçesinde eksilme, satıcının bütçesinde de artış yapılmalıdır ve bütçe de kök düzeyinde, **`ActorPerson`** ile **`ActorCompany`** örneklerinde

tutulmaktadır. Ad ile üye erişimi özelliğini kullanma gereksinimi bu noktada ortaya çıkmaktadır.

```
boolean result; Actor anActor; RoleInventory inventoryOwner;
RoleSupplier supplier; RoleCustomer customer;
//Adım 1
anActor = (Actor)suppliers.getSelectedItem();
if( !anActor.canSwitch("demoApp.RoleSupplier") ) {
  supplier = new RoleSupplier( );
  anActor.addRole( supplier );
}
else {
  supplier = (RoleSupplier)anActor.as("demoApp.RoleSupplier");
}
//Adım 2
anActor = (Actor)customers.getSelectedItem();
if( !anActor.canSwitch("demoApp.RoleCustomer",
                String.valueOf(supplier.supplierID) ) ) {
  customer = new RoleCustomer( supplier );
  anActor.addRole( customer );
}
else {
  customer = (RoleCustomer)anActor.as("demoApp.RoleCustomer");
}
//Adım 3
if( !anActor.canSwitch("demoApp.RoleInventory" ) ) {
  inventoryOwner = new RoleInventory();
  anActor.addRole( inventoryOwner );
}
//Adım 4
result = customer.buy( (Goods)goods.getSelectedItem(),
                amount.getValue() );
```

Şekil 8.40: Alım-satım işlemi öncesi uygulamanın yaptığı işlemler

Alım-satım işlemi için kullanıcının ve programın yaptığı ön hazırlıklar yukarıda anlatıldığı şekilde tamamlandıktan sonra, **`RoleCustomer.buy`** metodu çalıştırılır. İçeriği Şekil 8.41'de gösterilen bu metodun yürüttüğü işlemler ise şunlardır:

1. Müşteri rolü ile ilişkili satıcının **`sell`** metodu çalıştırılır.

2. Hiyerarşi kökünün **`RoleInventory`** rolüne geçiş yapılarak alınan mal kaleminin envanterde olup olmadığına bakılır. Eğer mal envanterde yoksa o mal için yeni bir kalem alınan adet ile oluşturulur ve yeni kalem envantere eklenir. Eğer mal envanterde önceden bulunuyorsa o sadece kalemin adedi arttırılır.

3. Hiyerarşi kökünün **`increaseFunds`** metodu yapılan harcama miktarı bütçeden düşürülecek şekilde ve JAWIRO'nun ad ile üye metod erişimi özelliği kullanılarak çalıştırılır.

```
public boolean buy( Goods aGood, int amount ) {
  int index;
  Goods myGood;
  boolean result;
  RoleInventory inventoryOwner;
  result = supplier.sell( aGood, amount );
    if( result ) {
      inventoryOwner = (RoleInventory)getActor().
                          as("demoApp.RoleInventory");
      index = inventoryOwner.getGoodIndex(aGood.getName());
      if( index == -1 ) {
        myGood = new Goods( aGood.getName(), amount,
                          aGood.getUnitPrice() );
        inventoryOwner.addGoods( myGood );
      }
      else {
        myGood = inventoryOwner.getGood( index );
        myGood.setCount( myGood.getCount() + amount );
        inventoryOwner.setGood( myGood, index );
      }
      getActor().executeLocalMethod( "increaseFunds", -
                          myGood.getUnitPrice()*amount );
    }
  return result;
}
```

Şekil 8.41: Müşteri rolünün satın alma metodu

```
public boolean sell( Goods aGood, int amount ) {
  int index;
  Goods myGood;
  RoleInventory inventoryOwner;
  inventoryOwner = (RoleInventory)getActor().
            as("demoApp.RoleInventory");
  index = inventoryOwner.getGoodIndex(aGood.getName());
  if( index == -1 )
    return false;
  myGood = new Goods( aGood.getName(),
            aGood.getCount() - amount, aGood.getUnitPrice() );
  inventoryOwner.setGood( myGood, index );
  getActor().executeLocalMethod( "increaseFunds",
            +myGood.getUnitPrice()*amount );
  return true;
}
```

Şekil 8.42: Satıcı rolünün satış metodu

Şekil 8.42'de verilen ve müşteri rolünün çalıştırdığı **`RoleSupplier.sell`** metodu da benzer işlemleri satıcı tarafında aşağıdaki gibi yürütür:

1. Hiyerarşi kökünün **`RoleInventory`** rolüne geçiş yapılarak satılan malın kalemi bulunur ve o kalemden satış adedi kadar düşülür.

2. Hiyerarşi kökünün **`increaseFunds`** metodu yapılan satış miktarı bütçeye eklenecek şekilde ve JAWIRO'nun ad ile üye metod erişimi özelliği kullanılarak çalıştırılır.

8.4.4.5 Kalıcılık

Gerçeklenen örnek uygulama, üzerinde çalışılan hastanenin tüm durum bilgisini kalıcı ortamda saklayabilir ve kalıcı ortamdan geri yükleyebilir. Bu amaçla JAWIRO'nun kalıcılık yeteneği ve Java dilinin **`java.util.properties`** paketi birlikte kullanılmıştır.

Üzerinde çalışılan hastanenin durum bilgisini oluşturan öğeler iki gruba ayrılabilir. Birinci grup öğelere varlık öğeleri adı verilmiştir ve her varlık öğesi uygulamada oluşturulan rol hiyerarşilerinden birini simgeler. İkinci grup üyelere ise basit öğeler adı verilmiştir ve varlık öğelerinin yüklenebilmesi için gerekli bilgileri içerir. Rol hiyerarşilerinin saklama ve yükleme işlemlerinde Bölüm 8.1.2.8 ve Bölüm 8.2.9'da anlatılan kalıcılık yöneticisi, basit öğelerin saklama ve yükleme işlemlerinde ise Bölüm 7.5'te anlatılan **`java.util.properties`** paketi kullanılmıştır. Saklanılan basit öğeler şunlardır:

- Kalıcılık yöneticisinin yolu ve adı,
- Üzerinde çalışılan hastanenin rol hiyerarşisinin kökünü simgeleyen **`RICH.hospital`** üyesinin kalıcılık yöneticisine kayıt ettirilirken kullanılan anahtar,
- Tanımlı şirketlerin sayısı,
- Tanımlı şirketleri simgeleyen rol hiyerarşilerinin köklerini kalıcılık yöneticisine kayıt ettirilirken kullanılan anahtarlar,
- Tanımlı hastaların sayısı,
- Tanımlı hastaları simgeleyen rol hiyerarşilerinin köklerini kalıcılık yöneticisine kayıt ettirilirken kullanılan anahtarlar.

Örnek uygulamanın durumunu kalıcı ortama kaydetmekten **`public boolean RICH. saveState(File f)`** metodu sorumludur. Bu metot basit öğe bilgilerini yukarıda anlatılan sırayla hesaplayıp **`Properties RICH. stateSaveLoadSettings`** üyesine yazar. Her bir basit öğe hesaplandıktan sonra o basit öğe ile ilgili varlık öğeleri **`private PersistenceManager RICH.pm`** üyesi olan kalıcılık yöneticisi üzerinden kalıcı ortama kaydedilir. Örneğin tanımlı şirketlerin sayısı alındıktan sonra her şirketin rol hiyerarşisi döngü içerisinde tek tek kalıcı ortama kaydedilir. Üzerinde çalışılan hastanedeki personelin, hastanenin işveren rolünde tutulduğu Bölüm 8.4.3'te anlatılmıştı. Bu yüzden hastane personeli, **`RICH.hospital`** köklü rol hiyerarşisi kalıcı ortama kaydedilirken kendiliğinden kalıcı ortamda saklanmış olur. Son olarak **`saveState`** metodu basit öğeleri kalıcı ortama kaydeder. Dosyanın adı ve yerleşkesi, kullanıcının uygulamada "File – Save" komutunu vermesi veya çalışma kümesi alt sisteminden "Save this hospital" komutunu vermesi ile ortaya çıkan dosya kayıt etme diyalog penceresi ile belirlenir. Uygulama bu dosyaya **`.rich`** uzantısını verir.

Örnek uygulamanın durumunu kalıcı ortamdan geri yüklemekle **`public boolean RICH. loadState(File f)`** metodu sorumludur. Uygulamada "File – Load" komutu veya çalışma kümesi alt sisteminden "Open an existing hospital" komutu verildiği zaman ortaya çıkan dosya açma diyalog penceresi ile **`saveState`** metodunun oluşturduğu basit öğeleri içeren dosya seçilir. Bunun ardından çalışan **`loadState`** metodu saklanan basit öğeleri yukarıda anlatılan sırada yükler. Her bir basit öğenin ardından, ilgili rol hiyerarşisi kalıcılık yöneticisi üzerinden kalıcı ortamdan geri yüklenir.

8.4.4.6 Rollerin Temel Özellikleri

Örnek uygulamada rollerin kullanımı da göstermiştir ki, rollerin temel özellikleri gerçekten dinamik sistemlerin modellenmesinde en yoğun olarak kullanılan çekirdek yeteneklerden oluşmaktadır. Rollerin ileri düzey özelliklerinin her biri, uygulamanın önceki bölümlerde anlatılan yüksek düzey işlevlerinden birine veya bir kaçına yönelik olarak kullanılmıştır. Rollerin temel özellikleri ise uygulamanın her noktasında kaçınılmaz olarak karşımıza çıkmaktadır.

9. JAWIRO'NUN GÜNCEL ROL MODELLERİ İLE KARŞILAŞTIRMASI

Bu bölüm, tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO'nun diğer güncel rol modelleri ile karşılaştırılmasına ayrılmıştır. Yapılan karşılaştırmalar özellikler ve başarım açısından iki ayrı grupta incelenebilir.

9.1 Gerçeklenen Özellikler Açısından Karşılaştırmalar

Dinamik sistemlerin modellenmesi için yapılan önerilerden Bölüm 3'te incelenen rol kavramı dışındaki çalışmalar, önemli zorluklar ve eksikler içermektedir. Söz konusu eksikler ilgili çalışmaların değerlendirme bölümlerinde incelenmiştir. Bölüm 3'te değinilen çalışmalardan bazılarının çözmeyi hedeflediği problemler doğrudan dinamik sistemlerin modellenmesi olmayıp, birincil amaçları farklı problemleri çözmektir. Üstelik bu çalışmalar kendi alanlarında çoğu kez başarılı ilerlemeler göstermişlerdir. Bununla birlikte, dinamik sistemlerin modellenmesi sorunu gerçek dünya uygulamalarının büyük çoğunluğunda ortaya çıktığından, söz konusu çalışmalar tez çalışmasının asıl ilgi alanı olan rol yaklaşımı ile az veya çok ilgi taşımaktadır. Sonuç olarak Bölüm 3'te incelenen çalışmalar dinamik sistemlerin modellenmesi açısından rol yaklaşımları kadar iyi özelliklere sahip olmadığından, bu çalışmalar özellik açısından karşılaştırmaya alınmamışlardır.

Tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO'nun, literatürdeki diğer rol modelleri ile gerçeklenen özellikler açısından karşılaştırılması yerinde olacaktır. Bölüm 6'da incelenen söz konusu çalışmalardan Bölüm 6.2'de incelenen Smalltalk ile gerçeklenen rol modeli [2], Bölüm 6.7'de incelenen Schrefl ve Thalhammer'ın Java ile yaptıkları gerçeklemenin [43] geçmiş temellerini oluşturarak güncelliğini yitirdiğinden, özellik açısından karşılaştırmaya alınmamıştır. Bölüm 6.1'de incelenen Fibonacci [24] de güncel bir çalışma olmadığı için değerlendirmeye alınmamıştır.

Özellik açısından yapılan karşılaştırmalara tasarım kalıplarının eklenmesi, başarılı tasarım kalıplarının güncelliğini yitirmeyerek yazılım uygulamalarına önemli katkılar

sağlamaları nedeniyle yerinde olacaktır. Bu amaçla Bölüm 4'te incelenen tasarım kalıpları arasından rol nesneleri kalıbı seçilmiştir. Yapılan seçimin nedeni, Bölüm 4.4'te de değinildiği gibi, rol nesneleri kalıbının rollerin daha çok olduğu veya daha sık yeni rollerin ortaya çıktığı sistemler için diğer kalıplardan daha uygun olmasıdır.

Seçilen çalışmaların özellik açısından karşılaştırılması Tablo 9.1 ve Tablo 9.2'de görülebilir. Jawiro, Schrefl'in rol modeli [43] ve rol nesneleri tasarım kalıbının [21] ayrı bir grup olarak gösterilmelerinin nedeni, bu çalışmaların Bölüm 9.2'de çalışma zamanı başarımı açısından da karşılaştırılmasıdır.

Tablo 9.1: Roller ile ilgili güncel çalışmaların ilk grubunun, rollerin desteklenen özellikleri açısından karşılaştırması

	Rollerin Özellikleri	**Jawiro**	**Schrefl [43]**	**Kalıp [21]**
Temel Özellikler	Gerçekleme dili	Java	Java	Java
	Başka yazılıma bağımlı olmama	+	+	+
	Rol hiyerarşileri	+	+	-
	Çalışma anı rol varlığı kontrolü	+	+	+
	Nitelikli roller	+	+	+
	Çakışma engeli	+	+	+
	Sınıf düzeyi ve nesne düzeyi kalıtımın birlikte kullanımı	+	+	+
İleri Düzey Özellikler	Aykırı rol ilişkilerinin önlenmesi	+	-	-
	Kalıcılık	+	-	-
	Rol aktarımı	+	-	-
	Nesne düzeyinde çoklu kalıtım	+	-	-
	Ad ile üye erişimi	+	-	-
	Baskın roller	+	-	-
	Rollerin askıya alınması ve askıdan indirilmesi	+	-	-
	Danışma ve vekalet mekanizmalarının birlikte kullanımı	+	-	-
	Rol araması ve geçişinde iyileştirme	+	-	-

İlk grup çalışmalar incelendiğinde, rol nesneleri tasarım kalıbının rollerin hiyerarşik düzenini desteklememesi dışında, rollerin temel özelliklerinin tümünün desteklendiği

görülmektedir. Rollerin hiyerarşik düzeninin getirdiği rollerin de rol sahibi olabilmeleri yeteneğinin yararları Bölüm 5.2.1.2'de incelenmişti. Rollerin ileri düzey özelliklerinin tümünün ise sadece JAWIRO tarafından desteklendiği ve bunlardan ilk ikisi dışındakilerin tez çalışmasının özgün katkıları olduğu da Tablo 9.1 ve Tablo 9.2 birlikte incelendiğinde görülecektir.

Tablo 9.2: Roller ile ilgili güncel çalışmaların ikinci grubunun, rollerin desteklenen özellikleri açısından karşılaştırması

	Rollerin Özellikleri	**Dec-Java[32]**	**GRM [33]**	**INADA [26]**	**Chame-leon[29]**
Temel Özellikler	Gerçekleme dili	Java	Java	C++	Java
	Başka yazılıma bağımlı olmama	+	-	-	-
	Rol hiyerarşileri	+	-	-	+
	Çalışma anı rol varlığı kontrolü	-	-	+	+
	Nitelikli roller	+	-	-	-
	Çakışma engeli	+	-	+	+
	Sınıf düzeyi ve nesne düzeyi kalıtımın birlikte kullanımı	-	+	+	+
İleri Düzey Özellikler	Aykırı rol ilişkilerinin önlenmesi	-	+	-	-
	Kalıcılık	-	-	+	-
	Rol aktarımı	-	-	-	-
	Nesne düzeyinde çoklu kalıtım	-	-	-	-
	Ad ile üye erişimi	-	-	-	-
	Baskın roller	-	-	-	-
	Rollerin askıya alınması ve askıdan indirilmesi	-	-	-	-
	Danışma ve vekalet mekanizmalarının birlikte kullanımı	-	-	-	-
	Rol araması ve geçişinde iyileştirme	-	-	-	-

İkinci grup çalışmalar incelendiği zaman, daha rollerin temel özelliklerinin gerçeklenmesinde önemli eksiklikler göze çarpmaktadır. Rollerin ileri düzey özellikleri arasında ise sadece aykırı rol ilişkilerinin önlenmesi Lee ve Bae'nin [33], kalıcılık ise Aritsugi ve Makinouchi'nin [26] çalışmalarının özgün katkılarıdır ve bu özellikler de JAWIRO tarafından desteklenmektedir.

Özetle, sadece rollerin temel özelliklerine gereksinim duyan dinamik sistemlerin modellenmesinde ilk grupta incelenen çalışmalar olan rol ilişkileri tasarım kalıbı [21], Schrefl ve Thalhammer'ın rol modeli [43] veya tez çalışması kapsamında gerçeklenen rol modeli olan JAWIRO'nun rahatça kullanılabileceği söylenilebilir. Ancak rollerin ileri düzey özelliklerinden herhangi birine gereksinim duyulduğu zaman, JAWIRO literatürdeki diğer çalışmalar arasından sıyrılarak öne çıktığı görülmektedir.

9.2 Çalışma Zamanı Başarımı Açısından Karşılaştırmalar

Çalışma zamanı başarımı da araçların değerini belirlemekte en az desteklenen özellikler kadar önemli bir ölçüttür: Kabul edilebilir bir süre içerisinde tamamlanmayan işlemler, uygulamalar için hiç bir yarar sağlamaz. JAWIRO'nun rollerin temel ve ileri düzey özelliklerinin kullanımında kabul edilebilir bir başarım gösterdiği Bölüm 8.3'te gösterilmişti. Sadece yetenek açısından bakıldığı zaman ve yalnızca rollerin temel özelliklerine gereksinim duyulduğunda; JAWIRO, Schrefl ve Thalhammer'ın rol modeli [43] ve rol ilişkileri tasarım kalıbının [21] kullanılabileceği ise Bölüm 9.1'de anlatılmıştı. Bölüm 9.1'de incelenen ilk grubu oluşturan bu üç çalışma, Bölüm 9.2'de rollerin temel özelliklerinin kullanımının çalışma zamanı başarımına etkileri açısından birbirleri ile karşılaştırılacaktır. Tasarım kalıpları herkes tarafından gerçeklenebilir. Diğer iki çalışmadan Schrefl'ın rol modeline http://www.dke.uni-linz.ac.at/research/projects/roles/index.html, JAWIRO rol modeline ise http://www.yunusemreselcuk.com/jawiro/index.html adresinden erişilebilir.

JAWIRO'nun çalışma zamanı başarımının ölçümü için Bölüm 8.3.2'de kullanılan kod, bu bölümde incelenen diğer çalışmalara Bölüm 8.3.1 ve Bölüm 8.3.2'de anlatılan detaylar değiştirilmeden, aynı şekilde uyarlanmıştır. Bu koşullar altında, sabit hiyerarşi derinliğinde ve Intel sistem kullanılarak yapılan karşılaştırmalar Tablo 9.3'te görülebilir. Sonuçlar önce iyileştirmesiz çalışma düzeninde incelendiğinde, rol modellerinin hiyerarşi kurulumu sırasında gösterdiği başarımın rol ilişkileri tasarım kalıbına göre önemli ölçüde kötü olduğu görülecektir. Ancak Bölüm 8.3.2'de de anlatıldığı gibi, kurulum işlemleri yalnızca uygulamanın başında ve bir kez yapılacağı için, asıl ağırlığı diğer işlemlerin başarımı oluşturacaktır. Rol modelleri rollerin diğer temel özelliklerinin çalıştırılmasında rol ilişkileri tasarım kalıbı ile

denk veya tasarım kalıbından daha iyi başarım göstermektedir. Rol varlığı kontrolünde en iyi başarımı Schrefl'in rol modeli göstermiştir. Bu üstünlük, rollerin **`java.util.Hasthable`** veri yapısında saklanmasından ötürü hızla kontrol edilebilmesinden kaynaklanmaktadır. Schrefl'in rol modeli bu avantajını rol geçişi işleminde koruyamamaktadır; JAWIRO rol geçişinde Schrefl'in rol modelinden yaklaşık %40 daha iyi başarım sergilemektedir. Rollerin doğrudan veya geçişli çalıştırılmasında üç çalışma da yaklaşık olarak birbirine denk başarım göstermiştir. Rol geçişli ve aramalı çalıştırmada ise JAWIRO diğer çalışmalardan yaklaşık %30 daha iyi başarım sergilemektedir. Bölüm 8.2.6'da anlatılan iyileştirme özelliği bu üstünlüğü doğurmaktadır.

Tablo 9.3: Intel sistem üzerinde yapılan başarım karşılaştırması

Hiyerarşi derinliği: 6, Hiyerarşi derecesi: 3, Deneme sayısı: 10	Temel işlemler için komut başına işlem süresi (ms.)					
	İyileştirmesiz Çalışma			İyileştirmeli Çalışma		
	JAWIRO	Schrefl	Kalıp	JAWIRO	Schrefl	Kalıp
Hiyerarşi kurmak	0,038	0,085	0,004	0,04	0,094	0,003
Rol varlığı kontrolü	0,018	0,010	0,041	0,0003	0,010	0,041
Rol geçişi	0,018	0,030	0,041	0,0003	0,030	0,041
Rol çalıştırması	0,006	0,005	0,004	0,003	0,003	0,003
Rol geçişli çalıştırma	0,043	0,050	0,056	0,004	0,034	0,044
Rol aramalı ve geçişli çalıştırma	0,068	0,097	0,110	0,006	0,067	0,088

Tablo 9.3'teki sonuçlar iyileştirmeli çalışma düzeni çerçevesinde karşılaştırıldığında, JAWIRO dışındaki çalışmalarda çok önemli bir iyileşme görülmemektedir. Bunun nedeni JAWIRO'nun Bölüm 8.2.6'da anlatılan iyileştirme özelliğinin diğer çalışmalarda bulunmamasıdır. Yine de rol geçişli çalıştırma ile rol aramalı ve geçişli çalıştırma işlemlerinde; Schrefl'in rol modelinde %30, rol ilişkileri tasarım kalıbında ise %20 başarım artışı görülmektedir. Bu başarım artışı sadece Java'nın HotSpot adlı [44] çalışma anı derleyicisinden kaynaklanmaktadır. Halbuki JAWIRO rol çalıştırması hariç tüm işlemlerde %89 - %98,5 arasında başarım artışı göstermiştir.

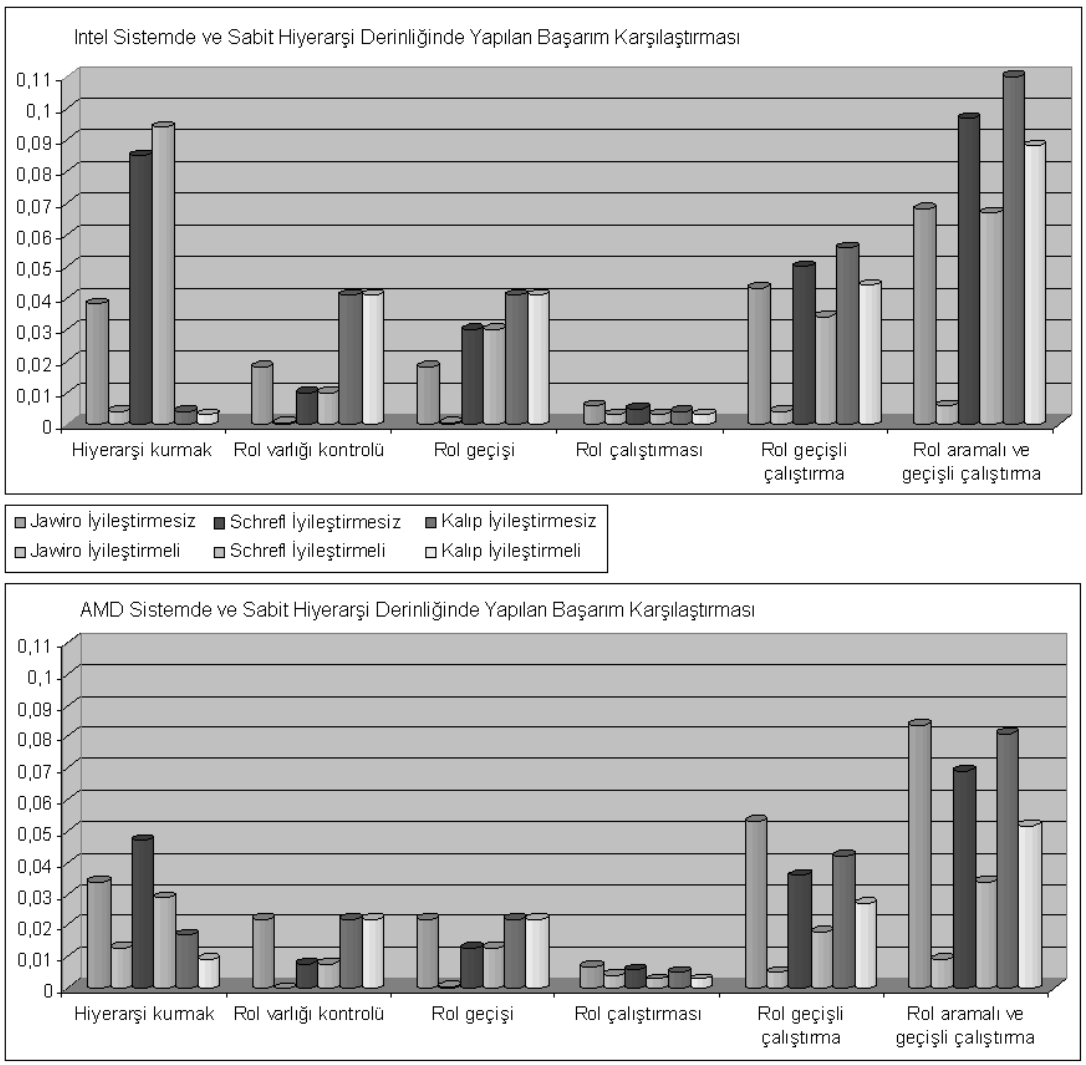
Rol çalıştırması işlemi ise doğrudan metot çalıştırmadan oluştuğu için bu işlemde herhangi bir başarım artışı gözlenmesi zaten beklenilemez.

Başarım karşılaştırması AMD sistemde yapıldığı zaman elde edilen sonuçlar ise Tablo 9.4'te görülebilir. İyileştirmesiz çalışma düzeninde ilk dikkati çeken nokta, hiyerarşi kurulumu dışında tüm işlemlerde Schrefl'in rol modelinin en iyi başarımı göstermesidir. Intel sistemden AMD sisteme geçişte JAWIRO'nun başarımı %16 ile %19 arasında değişen oranlarda kötüleşmiş, buna karşılık Schrefl'in rol modelinin ve rol ilişkileri tasarım kalıbının başarımı %21 ile %57 arasında değişen oranlarda iyileşmiştir. AMD sistemde elde edilen sonuçlar iyileştirmeli çalışma düzeni çerçevesinde karşılaştırıldığında ise, JAWIRO'nun iyileştirmeli çalışma yeteneğinin Intel sistemde olduğu gibi %89 - %99 arasında başarım artışı kazandırdığı görülecektir.

Tablo 9.4: AMD sistem üzerinde yapılan başarım karşılaştırması

Hiyerarşi derinliği: 6, Hiyerarşi derecesi: 3, Deneme sayısı: 10	Temel işlemler için komut başına işlem süresi (ms.)					
	İyileştirmesiz Çalışma			İyileştirmeli Çalışma		
	JAWIRO	Schrefl	Kalıp	JAWIRO	Schrefl	Kalıp
Hiyerarşi kurmak	0,034	0,047	0,017	0,013	0,029	0,009
Rol varlığı kontrolü	0,022	0,008	0,022	0,0002	0,008	0,022
Rol geçişi	0,022	0,013	0,022	0,0003	0,013	0,022
Rol çalıştırması	0,007	0,006	0,005	0,004	0,003	0,003
Rol geçişli çalıştırma	0,053	0,036	0,042	0,005	0,018	0,027
Rol aramalı ve geçişli çalıştırma	0,084	0,069	0,081	0,009	0,034	0,052

Rollerin temel özelliklerinin Tablo 9.3 ve Tablo9.4'te verilen başarım karşılaştırmalarının bir arada verilen görsel biçimi için ise Şekil 9.1 incelenebilir.



Şekil 9.1: Rollerin temel özelliklerinin sabit hiyerarşi derinliğinde çalıştırılması ile elde edilen başarım sonuçları

Rol hiyerarşisinin derinliğinin JAWIRO'nun rollerin temel özelliklerini çalıştırmadaki başarımı üzerindeki etkisi Bölüm 8.3.2'de incelenmişti. Oluşan ek yükün bir milisaniyenin üstüne çıktığı an kırılma noktası olarak tanımlandığında, JAWIRO için bu değerin 9000 rol nesnesi veya 9 hiyerarşi derinliği olacağı da bu incelemede anlatılmıştı. Aynı inceleme Schrefl'in rol modeli için yapıldığında elde edilen değerler Tablo 9.5'te, rol ilişkileri tasarım kalıbı için tekrarlandığında ise oluşan sonuçlar ise Tablo 9.6'da görülebilir. Bu bilgiler doğrultusunda Schrefl'in rol modelinin kırılma noktasının Intel sistem için 9, AMD sistem için 10 hiyerarşi derinliği olduğu söylenebilir. Rol ilişkileri tasarım kalıbı için ise kırılma noktası 9 hiyerarşi derinliği olarak ortaya çıkmıştır. Değişen hiyerarşi derinliklerinde yapılan ölçümler, hiyerarşi derinliği 6 olarak sabit tutulduğunda yapılan ölçümlerle

benzeşmektedir. Dolayısıyla Tablo 9.3 ve Tablo 9.4 kullanılarak yapılan değerlendirmeler, Tablo 9.5 ve Tablo 9.6 için de geçerliliğini korumaktadır. Değişen hiyerarşi derinliklerinde de Intel sistemde JAWIRO, AMD sistemde de Schrefl'in rol modeli; başarım üstünlüğünü korumaya devam etmektedir. Ancak hiyerarşi derinliği en üst değerler olan 9 ve 10 düzeylerine çıktığında, JAWIRO Intel sistemde gösterdiği başarım üstünlüğünü koruyamamakta ve Schrefl'in rol modelinin başarımı JAWIRO'ya denk olmaktadır.

Tablo 9.5: Schrefl'in rol modeline hiyerarşi derinliğinin etkisi

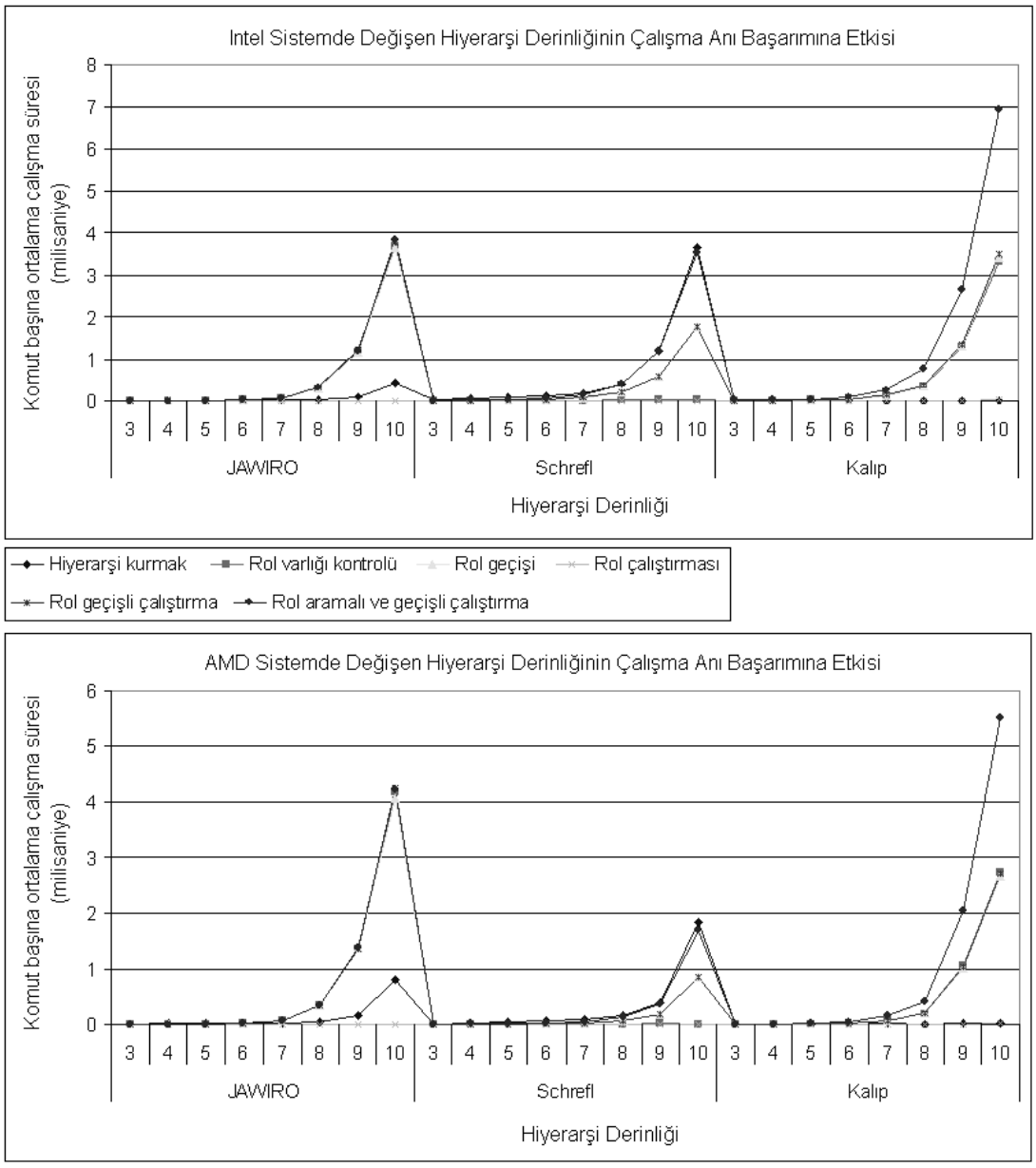
Schrefl'in rol modeli ile komut başına ortalama işlem süreleri (ms.) (Hiyerarşi derecesi: 3)									
	Derinlikler:	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Intel Sistem	Hiyerarşi kurmak	0,039	0,069	0,090	0,118	0,171	0,421	1,207	3,634
	Rol araması	0,011	0,011	0,008	0,010	0,009	0,019	0,019	0,033
	Rol geçişi	0,011	0,014	0,019	0,031	0,073	0,201	0,596	1,774
	Rol çalıştırması	0,000	0,000	0,002	0,000	0,000	0,002	0,000	0,000
	Rol geçişli çalıştırma	0,000	0,014	0,022	0,032	0,075	0,204	0,599	1,758
	Rol aramalı ve geçişli çalıştırma	0,012	0,023	0,033	0,064	0,150	0,410	1,195	3,532
AMD Sistem	Hiyerarşi kurmak	0,003	0,022	0,035	0,065	0,080	0,147	0,397	1,845
	Rol araması	0,008	0,009	0,006	0,008	0,009	0,008	0,013	0,007
	Rol geçişi	0,005	0,008	0,008	0,014	0,027	0,065	0,189	0,854
	Rol çalıştırması	0,000	0,002	0,000	0,000	0,000	0,001	0,003	0,000
	Rol geçişli çalıştırma	0,005	0,006	0,009	0,015	0,027	0,070	0,189	0,846
	Rol aramalı ve geçişli çalıştırma	0,011	0,016	0,020	0,028	0,054	0,135	0,388	1,702
	Rollerin sayıları:	**12**	**39**	**120**	**363**	**1.092**	**3279**	**9840**	**29523**

Tablo 9.6: Rol ilişkileri tasarım kalıbına hiyerarşi derinliğinin etkisi

Rol ilişkileri tasarım kalıbı ile komut başına ortalama işlem süreleri (ms.) (Hiyerarşi derecesi: 3)									
	Derinlikler:	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Intel Sistem	Hiyerarşi kurmak	0,000	0,000	0,000	0,004	0,007	0,010	0,013	0,012
	Rol araması	0,000	0,007	0,015	0,042	0,132	0,371	1,276	3,313
	Rol geçişi	0,000	0,011	0,019	0,042	0,129	0,368	1,299	3,409
	Rol çalıştırması	0,000	0,000	0,000	0,000	0,001	0,001	0,000	0,016
	Rol geçişli çalıştırma	0,003	0,013	0,016	0,044	0,137	0,387	1,331	3,500
	Rol aramalı ve geçişli çalıştırma	0,015	0,016	0,031	0,088	0,272	0,766	2,666	6,926
AMD Sistem	Hiyerarşi kurmak	0,000	0,000	0,000	0,004	0,014	0,011	0,013	0,015
	Rol araması	0,000	0,000	0,008	0,024	0,074	0,214	1,055	2,719
	Rol geçişi	0,003	0,006	0,010	0,023	0,077	0,212	1,006	2,661
	Rol çalıştırması	0,000	0,000	0,000	0,000	0,000	0,001	0,003	0,016
	Rol geçişli çalıştırma	0,003	0,009	0,016	0,024	0,078	0,213	1,038	2,701
	Rol aramalı ve geçişli çalıştırma	0,000	0,006	0,016	0,047	0,154	0,425	2,044	5,510
	Rollerin sayıları:	**12**	**39**	**120**	**363**	**1.092**	**3279**	**9840**	**29523**

Hiyerarşi derinliğinin JAWIRO, Schrefl'in rol modeli ve rol ilişkileri tasarım kalıbına yaptığı etkilerin bir arada görülebilmesi amacıyla Şekil 9.2 incelenebilir. Şekil 9.2'de, Tablo 8.12, Tablo 8.13, Tablo 9.5 ve Tablo 9.6'da verilen sonuçlar grafik şeklinde verilmiştir.

Özetle, birkaç bin rol nesnesi içeren sistemlerde incelenen tüm yaklaşımlar rollerin temel özelliklerini kullanmada fark edilebilir bir ek yük getirmemektedir. Çalışma anı başarımının önemli olduğu uygulamalarda, kullanılacak rol modeli yazılımı çalıştıracak sisteme göre seçilmelidir. Intel sistemlerde JAWIRO, AMD sistemlerde ise Schrefl'in rol modelinin diğerine göre %20-%30 oranları arasında daha iyi çalışma anı başarımı sergilediği gösterilmiştir. Rol nesnesi sayısı on binlere çıktığı sistemlerde ise Schrefl'in rol modeli başarım göstermektedir.



Şekil 9.2: Rollerin temel özelliklerinin sabit hiyerarşi derinliğinde çalıştırılması ile elde edilen başarım sonuçları

10. SONUÇLAR VE TARTIŞMA

Dinamik sistemlerin etkin bir biçimde modellenmesi için önerilen yaklaşımlardan biri olan rol modelleri, kullanışlılığı ile dikkat çeken bir araştırma alanı sunmaktadır [33]. Dinamik sistemleri modellerken nesne düzeyinde özelleştirme yapılması, sınıf düzeyinde özelleştirmeden daha uygundur [2] ve rol modellerinin öncelikli işlevi de nesne düzeyinde kalıtım yeteneği sunmaktır. Tez çalışmasının amacı ise Java diline rol desteğinin kazandırılmasıdır. Bu hedefe JAWIRO adlı rol modeli gerçeklenerek ulaşılmıştır.

Rol modelleri dışındaki çeşitli yaklaşımların yeterince etkili olamadıkları Bölüm 3'te incelenmişti. Yaygın kullanılan bir nesneye yönelik programlama dilinin rol desteği ile genişletilmesinin avantajları ise yeni bir dil veya dil eklentileri öğrenilmek zorunda kalınmaması ve rol tabanlı programlamanın NYP'yi doğal bir biçimde genişletmesidir [2]: NYP'nin sağladığı sınıf düzeyinde kalıtım ile RTP'nin sağladığı nesne düzeyinde kalıtım ilişkileri birbirini tamamlar. Mevcut rol modellerinin yaygın olarak destekledikleri rol ilişkileri, dinamik sistemlerin en temel gereksinimlerini karşılamaktadır. Bu ilişkiler tez çalışmasında rollerin temel özellikleri olarak adlandırılmışlardır. Bununla birlikte, literatürde mevcut olan rol modelleri roller ile gerçeklenebilecek her ilişki türünü desteklememektedir. Literatürdeki rol modellerine ögün birer katkı olarak INADA [26] kalıcılık özelliğini ve GRM [33] de aykırı rol ilişkilerini önleme özelliğini kazandırmışlardır. JAWIRO bunlara roller ile gerçeklenebilecek altı yeni özellik eklemiştir ve bunların hepsi birden tez çalışmasında rollerin ileri düzey özellikleri olarak adlandırılmışlardır. JAWIRO'nun özgün katkılarını oluşturan ve Bölüm 5.2.2 ile Bölüm 8.1.2'de tanıtılan bu özellikler şunlardır:

- Rol aktarımı,
- Nesne düzeyinde çoklu kalıtımın desteklenmesi,
- Ad ile üye erişimi,

- Baskın roller,
- Rollerin askıya alınması ve askıdan indirilmesi,
- Danışma ve vekalet mekanizmalarının birlikte kullanımı.
- Bölüm 8.2.6'da değinilen rol araması, rol geçişi ve ad ile üye erişimi özelliklerinin çalışma anı başarımını iyileştiren teknik de JAWIRO'nun diğer bir özgün katkısıdır.

JAWIRO rollerin kullanımı için sağladığı çok sayıda özelliği, çalışma anı başarımına hissedilebilir bir ek yük getirmeden kullanıma sunar. Bölüm 8.3'te gösterildiği üzere, sayıları binlerle ölçülen rol nesnelerine sahip bir hiyerarşide bile JAWIRO'nun getirdiği ek yük komut başına bir milisaniyeden azdır. JAWIRO'nun çalışma anı başarımının diğer çalışmalarla denk olduğu Bölüm 9.2'de gösterilmiştir.

Özetle, sadece rollerin temel özelliklerine gereksinim duyan bir sistemin modellenmesi için JAWIRO tek seçenek değildir: Bu gibi sistemlerde literatürde mevcut herhangi bir rol modeli veya uygun bir tasarım kalıbı kullanılabilir. Ancak rollerin ileri düzey özelliklerine gereksinim duyan sistemlerde, sağladığı özgün katkılar nedeniyle JAWIRO tercih edilebilir. Aksi takdirde gereksinim duyulan özellikleri gerçekleme yükü programcının üzerinde düşecektir. Burada JAWIRO'nun yazılım projelerine kalite ölçütlerine göre getirdiği kazanımlar unutulmamalıdır: Bölüm 5.4'te incelendiği gibi JAWIRO rol modelinin kullanılması ile yeniden kullanılabilirlik ve bakım kolaylığı artmaktadır. Üretilecek yazılıma aynı yararları tasarım kalıpları kullanarak kazandırmak mümkün olmakla birlikte zordur. Kalite ölçütlerine göre 'iyi' yazılım üretmek ve literatürdeki tasarım kalıplarına hakim olmak, ancak zaman içerisinde kazanılan deneyim ile mümkün olabilir. Ancak JAWIRO kullanıcılara bir dinamik sistemi modelleyen 'iyi' bir yazılım üretmek için doğal ve kolay bir yol sunar.

Tez çalışması iki ulusal ve beş uluslar arası konferansta yayınlanan bildirilerle tanıtılmıştır. Hazırlanan ilk bildiri tez çalışmasının ilk aşamalarında yapılan literatür taramasının sonuçlarını içermektedir [45]. JAWIRO rol modeli geliştirilmeye başlandığında bir ulusal [46] ve bir de uluslar arası bildiride [47] tanıtılarak rollerin temel özellikleri anlatılmıştır. Rollerin ileri düzey özellikleri de gerçeklenmelerinin

ardından bir ulusal [48] ve bir de uluslar arası bildiride [49] tanıtılmıştır. Rol modeli tamamlandıktan sonra rollerin temel ve ileri düzey özellikleri ile birlikte başarım ölçümlerini de içeren bir bildiri, özellikle rol modelleri hakkında bir alt konu alanı içeren AAAI 2005 fall konferansına kabul edilmiştir [50]. Son olarak rollerin ileri düzey özelliklerinin ÇES'de kullanılabileceğini gösteren bir bildiri daha hazırlanmıştır [13] ve bu bildirinin genişletilmiş hali LNCS'de yayınlanmıştır [51].

JAWIRO halen geliştirmeye açık noktalar içermektedir. Bunlardan ilki kalıcılık özelliğidir. JAWIRO mevcut haliyle ortogonal olmayan bir yapıya sahiptir. Ortogonal sistemler [52] nesnelerin tiplerinden bağımsız olarak işlenmesi, artımsal evrimleşme, tip doğruluğu sağlaması gibi avantajlar sağlar. Ayrıca kalıcılık dosyaları için kullanılan şifreleme güçlendirilebilir ve daha esnek hale getirilebilir: Bölüm 8.1.2.1'de anlatılan kısıtlama yöneticisinde olduğu gibi strateji tasarım kalıbı kullanılarak kullanıcıya kendi şifreleyicilerini kullanma olanağı sağlanabilir. Kalıcılık yöneticisi için diğer bir geliştirme alanı ise, hiyerarşilerin her kalıcı ortama saklanması işleminde rol nesnelerinin eski hallerinin silinmesinin bir zorunluluk değil seçimlik bir işlem olmasıdır. Bu sayede uygulamanın çeşitli aşamalarında, kalıcı ortamda yer alan önceki bilgiler kullanılarak bazı çözümlemeler yapılabilir. Örneğin bir hata durumunda, hatanın kaynağını bulmak için bu bilgilerden yararlanılabilir.

JAWIRO çeşitli eklemelerle bir rol tabanlı erişim kontrolü (RBAC) [53] sistemi olarak kullanılabilir. Bir RBAC modelinde roller bir organizasyonun işlevlerini simgeler ve yetkilendirmeler bireysel kullanıcılara değil rollere atanır. Kullanıcılar ise uygun rolleri oynamak üzere yetkilendirilir. Böylesi bir kullanım için JAWIRO daha gelişmiş bir kısıtlama yöneticisine sahip olmalıdır. Örneğin rol geçişi işlemleri de kısıtlanabilmelidir.

JAWIRO'nun kısıtlama yöneticisinin geliştirmeye açık diğer bir yönü de kısıtlamalar ile ilgili kurallar için bir söz dizimi hazırlanmasıdır. Bu söz dizimi ölümcül kilitlenmelere neden olacak kısıtlama kurallarının önceden belirlenmesi için de kullanılabilir.

Rol hiyerarşileri arasındaki ilişkiler de geliştirmeye açıktır. Örneğin kurulu bir rol hiyerarşisinin klonlanması ve birden fazla rol hiyerarşisinin rollerinin tek bir hiyerarşide uygun biçimde birleştirilmesi gibi yetenekler gerçeklenmeye değerdir.

Nitelikli roller JAWIRO'nun geliştirmeye açık diğer bir yönüdür. Nitelikli rollerin varlığı için yapılan kontrollerde joker karakter kullanımı söz konusu olabilir. Bu durumda kontrol komutu aranan koşula uyan roller arasında dolaşmayı sağlayacak bir **`java.util.Iterator`** işaretçisi geri döndürebilir.

Rollerin özellikleri etmenlerin bireysel davranışlarını ve eşgüdümlerini modellemek için kullanılabilir [13]. JAWIRO'nun bu şekilde bir ÇES gerçeklemesinde kullanılması da açık geliştirme alanlarından biridir.

Tez çalışması dağıtılmış çalışmayı destekleyecek yönde de geliştirilebilir. Dağıtılmış çalışma ile, rol hiyerarşisinin katılımcılarının ağ üzerindeki farklı düğümlerde bulunması anlatılmak istenmektedir.

Sınıf düzeyinde kalıtımla türetilmiş rollerin durumu da daha fazla incelenmeye değer ve geliştirmeye açıktır. A rol sınıfı ve bundan sınıf düzeyi kalıtımla türetilmiş B rol sınıfı olsun. Bir A örneği olan a nesnesi ile B örneği olan b nesnesinin aynı hiyerarşide bulunması durumu daha ayrıntılı olarak incelenebilir. Modellenen sistemin gereksinimlerine göre şu durumlardan biri gerçekleşebilir:

- Nesnelerin karşılıklı dışlamalı olması: Bir kısıtlama yöneticisi ile bu kural özel veya genel olarak yürütülebilir. Özel yürütme sadece "A rolü varsa B'yi, B rolü varsa A'yı ekleme" şeklinde tanımlanabilir. Genel yürütme ise "Eklenenecek her rolün sınıf düzeyi kalıtım zincirini incele. Eklenecek rolün üst sınıflarından biri hiyerarşide mevcutsa bu rolü ekleme" şeklinde tanımlanabilir. Anlatılan genel yürütme kuralının **`RoleHierarchy`** sınıfına kodlanması sayesinde JAWIRO'ya dahil edilmesi düşünülebilir.

- Nesnelerin aynı hiyerarşide aynı anda bulunabilmesi: Bu durum için ek bir işleme gerek yoktur. Ancak "en çok evrimleşmiş rol" kavramı hakkında cevaplanması gereken sorular ortaya çıkar. B sınıfı A'dan daha çok özelleşmiştir ama a rolü hiyerarşide b'den daha derinde bulunuyorsa hangisinin "en çok evrimleşmiş rol" olacağı tartışmaya açıktır.

- A sınıfına çağrıların b nesnesine yönlendirilmesi: b nesnesi her zaman a nesnesinin yerine kullanılabileceğine göre, a rolüne gelen çağrıların b rolüne otomatik olarak iletilmesi istenebilir. Ancak bu işlevin JAWIRO'ya kazandırılmasından önce üstteki iki madde üstünde görüş birliğine varılması ve gerçeklemenin ona göre yapılması gerekir. Mevcut durumda söz konusu çağrıların JAWIRO'nun ad ile üye metot erişimi yeteneği ile yapılması sorunu çözecektir.

- Nesnelerin birbirinin yerini alması: a yerine b, veya b yerine a örneğinin geçirilmesi istenebilir. Mevcut durumda programcı kendi vereceği veya kısıtlama yöneticisi içine yazacağı komutlarla bu işi yapabilir.

- Tip değişimi: a örneğini B tipine çevirmek istenebilir. Mevcut durumda programcı kendi vereceği veya kısıtlama yöneticisi içine yazacağı komutlarla bu işi yapabilir.

Tez çalışması kapsamında gerçeklenen rol modelini içeren Java paketine http://www.yunusemreselcuk.com/jawiro/index.html adresinden erişilebilir. Bu adreste JAWIRO'nun kullanıcı arabiriminin dokümantasyonu çevrimiçi olarak ve indirilebilir sıkıştırılmış dosya olarak bulunabilir. JAWIRO rol modelini kullanan ve Bölüm 8.4'te incelenen örnek uygulama da aynı adreste bulunmaktadır. Tüm bu kaynaklar tez kitabı ile birlikte verilen CD'de de bulunmaktadır.

KAYNAKLAR

[1] **Ungar, D. and Smith, R.B.,** 1987. SELF: The Power of Simplicty, *ACM Conference on Object Oriented Programming Systems, Languages and Applications,* Orlando, USA, 227-242.

[2] **Gottlob, G., Schrefl, M. and Röck, B.,** 1996. Extending Object-Oriented Systems with Roles, *ACM Trans. on Information Systems,* **14,** 268-296.

[3] **Stroustrup, B.,** 2000. The C++ Programming Language, Addison Wesley Professional, US.

[4] **Zendler, A.M.,** 1998. Foundation of the taxonomic object system, *Information and Software Technology,* **40,** 475-492.

[5] **Horstmann, C.S. and Cornell, G.,** 2005. Core Java 2, Volume I-Fundamentals, 7th ed, Prentice Hall, NJ.

[6] **Skarra, A.H. and Zdonik, S.B.,** 1986. Aspects: Extending Objects to Support Multiple, Independent Roles, *Proceedings of the ACM OOPSLA Conference*, New York, USA, 483-495.

[7] **Crawford, D., ed.,** 2001. Aspect-Oriented Programming Special Issue, *Communications of the ACM*, **44,** 29-125.

[8] **Hutchinson, B.A., et. al.,** 1986. Object Structure in the Emerald System, *Proceedings of the ACM OOPSLA Conference*, Portland, USA, 76-86.

[9] **Becht, M., et. al.,** 1999. ROPE: Role Oriented Programming Environment for Multiagent Systems, *4th IECIS Int'l. Conf. on Cooperative Information Systems,* Edinburgh, Scotland, 325-333.

[10] **Kendall, E.A.,** 2000. Role Modeling for Agent System Analysis, Design and Implementation, *IEEE Concurrency,* **8,** 34-41.

[11] **Cabri, G., Ferrari, L., Leonardi, L.,** 2005. Exploiting Runtime Bytecode Manipulation to Add Roles to Java Agents, *Science of Computer Programming,* **54,** 73-98.

[12] **Cabri, G., Leonardi, L., Zambonelli, F.,** 2003. BRAIN: A Framework for Flexible Role-Based Interactions in Multiagent Systems, *Proc. Conf. on Cooperative Information Systems*.

[13] **Selçuk, Y.E. and Erdoğan, N.,** 2005. A Role Model for Description of Agent Behavior and Coordination, *Proc. 6th Int'l Workshop on Engineering Societies in the Agents' World,* Kuşadası, Türkiye, 227-237.

[14] **Banerjee, J., et. al.,** 1987. Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *ACM Trans. on Office Information Systems,* **5,** 311-322.

[15] **Li, Q. and McLeod, D.,** 1988. Object Flavor Evolution in an Object-Oriented Database System, *Proceedings of the ACM Conference on Office Information Systems,* New York, USA, 265–275.

[16] **Wong, R.K.,** 1997. The Management of Changing Types in an Object-Oriented database, *Proceedings of the ACM RBAC Conference,* Fairfax, VA, USA, 109-120.

[17] **Yılmaz, G.,** 2001. Dağıtılmış Nesneye Dayalı Sistemler İçin Dağıtılmış Bileşik Nesne Modeli, *Doktora Tezi,* İ.T.Ü. Fen Bilimleri Enstitüsü, İstanbul.

[18] **Harrison, W. and Ossher, H.,** 1993. Subject-Oriented Programming - a Critique of Pure Objects, *Proceedings of the ACM OOPSLA Conference*, Washington DC, USA, 411-428.

[19] **Pace, J.A.D. and Campo, M.R.,** 2001. Analyzing the Role of Aspects in Software Design, *Commications of the ACM,* **44,** 66-73.

[20] **Kiczales, G., et. al.,** 2001. An Overview of AspectJ, *Proc. European Conf. on Object-Oriented Programming,* Budapest, 327-353.

[21] **Fowler, M.** Dealing with Roles, http://martinfowler.com/apsupp/roles.pdf, yayınlanmamış makale.

[22] **Gamma, E., et. al.,** 1994. Design Patterns Elements of Reusable Object Oriented Software, Addison-Wesley, Boston.

[23] **Kristensen, B.B.,** 1996. Conceptual Abstraction Theory and Practical Language Issues, *Theory and Practice of Object Systems*, **2**, 143-160.

[24] **Albano, E., et. al.,** 1993. An Object Data Model With Roles, *Proc. Int'l. Conf. Very Large Data Bases*, Saratoga, Calif., USA, 39–51.

[25] **Albano, E., et. al.,** 1985. Galileo: A Strongly Typed, Interactive Conceptual Language, *ACM Transactions on Database Systems,* **10,** 230-260.

[26] **Aritsugi, M. and Makinouchi, A.,** 2000. Multiple-Type Objects in an Enhanced C++ Persistent Programming Language, *Software-Practice and Experience,* **30,** 151-174.

[27] **Yu, G., Kaneko, K. and Makinouchi, A.,** 1996. Transaction Management for a Distributed Object Storage System WAKASHI: Design, Implementation and Performance, *International Conference on Data Engineering,* New Orleans, USA, 460-468.

[28] **Eliot, J. and Moss, B.,** 1992. Working with Persistent Objects: to Swizzle or not to Swizzle, *IEEE Transactions on Software Engineering,* **18,** 657-673.

[29] **Graversen, K.B.,** 2003. Implementation of a Role Language for Object-Specific Dynamic Separation of Concerns, *AOSD'03 Workshop "SPLAT"*, Boston, USA.

[30] http://www.csg.is.titech.ac.jp/openjava/

[31] http://java.sun.com/j2se/1.4.2/docs/guide/reflection/

[32] **Betini, L., Capecchi, S. and Venneri, B.,** 2003. Extending Java to Dynamic Object Behaviours, *Electronic Notes in Theoretical Computer Science,* **82**.

[33] **Lee, J-S. and Bae, D-H.,** 2002. An Enhanced Role Model For Alleviating the Role-Binding Anomaly, *Software-Practice and Experience,* **32,** 1317-1344.

[34] **Chiba, S.,** 2000. Load-Time Structural Reflection in Java, *Proc. European Conf. for Object-Oriented Programming,* Cannes, France. *LNCS,* **1850,** 313-336.

[35] **McConnell, S.,** 1993. Code Complete, Microsoft Press, Redmond, Washington, USA, 557-560.

[36] **Peckham, J. (ed.),** 2003. Practicing Software Engineering In The 21st Century, Idea Group Inc., PA, USA.

[37] **Cabri, G., Ferrari, L. and Leonardi, L.,** 2005. Applying Security Policies Through Agent Roles: A JAAS Based Approach, *Science of Computer Programming,* **59,** 127-146.

[38] **Barnard, J.,** 1997. Discussion and Survey of Reusability Metrics, *Report Reference: REUSABILITY–1 for the MOOD Project,* Department of Computer Science, University of Warwick, Coventry, UK.

[39] **Terry, C. and Dikel, D.,** 1996. Reuse Library Standards Aid Users in Setting Up Organisational Reuse Programs, *Embedded Systems Programming Product News.*

[40] **NASA,** 1995. Software Quality Metrics for Object Oriented System Environments, NASA Software Assurance Technology Center, 22-23.

[41] **Brito e Abreu, F. and Melo, W.,** 1996. Evaluating the Impact of Object-Oriented Design on Software Quality, *Proc. 3rd International Software Metrics Symposium,* Berlin, Germany, 90-99.

[42] **Ossher, H. and Tarr, P.,** 2001. Using Multidimensional Separation of Concerns to (Re)shape Evolving Software, *Communications of the ACM,* **44,** 43-50.

[43] **Schrefl, M. and Thalhammer, T.,** 2004. Using Roles in Java, *Software-Practice and Experience,* **34,** 449-464.

[44] http://java.sun.com/products/hotspot/

[45] **Selçuk, Y.E. and Erdoğan, N.,** 2003. How to Solve the Inefficiencies of Object Oriented Programming: A Survey Biased on Role-Based Programming, *7th World Multiconf. on Systemics, Cybernetics and Informatics,* Orlando, Florida, USA, 160-165.

[46] **Selçuk, Y.E. ve Erdoğan, N.,** 2004. JAWIRO: Java İçin Bir Rol Modeli, *TBD 21. Ulusal Bilişim Kurultayı,* Ankara, Türkiye, 17-29.

[47] **Selçuk, Y.E. and Erdoğan, N.,** 2004. JAWIRO: Enhancing Java with Roles, *19th Int'l. Symp. on Computer and Information Sciences,* Antalya, Turkey, *LNCS*, **3280,** 927-934.

[48] **Selçuk, Y.E. ve Erdoğan, N.,** 2004. Java Diline İleri Düzey Rol Desteği Kazandırılması: JAWIRO, *Havacılıkta İleri Teknolojiler ve Uygulamaları Sempozyumu,* İstanbul, Türkiye, Cilt II, 601-606.

[49] **Selçuk, Y.E. and Erdoğan, N.,** 2004. JAWIRO: An Extended Role Model for Java, *Int'l. Conf. on Computational Intelligence,* İstanbul, Turkey, 207-210.

[50] **Selçuk, Y.E. and Erdoğan, N.,** 2005. Using Roles with JAWIRO, *AAAI 2005 Fall Symposium Series, Roles: An Interdisciplinary Perspective subtopic,* Arlington, Virginia, USA.

[51] **Selçuk, Y.E. and Erdoğan, N.,** 2006. A Role Model for Description of Agent Behavior and Coordination, *Lecture Notes in Computer Science,* **3963,** 29-49.

[52] **Atkinson, M., and Morrison, R.,** 1995. Orthogonally Persistent Object Systems. *VLDB Journal,* **4,** 319-401.

[53] **Sandhu, R. (ed.),** 1995. *Proc. First ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, USA.

ÖZGEÇMİŞ

Yunus Emre Selçuk 1976 yılında Gelibolu'da doğmuştur. 1997 yılında İstanbul Üniversitesi Bilgisayar Bilimleri Mühendisliği bölümünden lisans derecesini almıştır. 2000 yılında İstanbul Teknik Üniversitesi, Bilgisayar Mühendisliği bölümünde yüksek lisans çalışmasını tamamlamıştır.