İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

ROCOCO İLE ROLE YÖNELİMLİ EŞ ZAMANLI PROGRAMLAMA

YÜKSEK LİSANS TEZİ

Cevat Serkan BALEK

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Yüksek Lisans Programı

HAZİRAN 2019

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

ROCOCO İLE ROLE YÖNELİMLİ EŞ ZAMANLI PROGRAMLAMA

YÜKSEK LİSANS TEZİ

Cevat Serkan BALEK
(504981180)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Yüksek Lisans Programı

Tez Danışmanı: Prof. Dr. Nadia ERDOĞAN

HAZİRAN 2019

İTÜ, Fen Bilimleri Enstitüsü'nün 504981180 numaralı Yüksek Lisans öğrencisi Cevat Serkan Balek, ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı "**ROCOCO İLE ROLE YÖNELİMLİ EŞ ZAMANLI PROGRAMLAMA**" başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı :	**Prof. Dr. Nadia Erdoğan** İstanbul Teknik Üniversitesi	.....
Jüri Üyeleri :	**Prof. Dr. Nadia Erdoğan** İstanbul Teknik Üniversitesi	.....
	Doç. Dr. Tolga Ovatman İstanbul Teknik Üniversitesi	.....
	Dr. Öğr. Üyesi Yunus Emre Selçuk Yıldız Teknik Üniversitesi	.....

Teslim Tarihi : 3 Mayıs 2019
Savunma Tarihi : 13 Haziran 2019

Sevgili Babam Kemal BALEK'in anısına,

"nüveyiz biz, ışık huzmeleriyiz…"

ÖNSÖZ

İster ufak bir hatası bile can kayıplarına yol açabilecek düzeyde hayat-kritik gömülü bir uygulama olsun, ister son kullanıcının zihnindekileri tam ve hatasız yapması beklenen misyon-kritik bir mobil uygulama olsun, yazılım ürünlerinin beklendiği gibi çalışması oldukça önemlidir.

Yazılım mühendisliğindeki en önemli konulardan biri, paydaşların isteklerinin net olarak anlaşılabilmesi ve aynı netlikte kodda ifade bulabilmesidir. Tutarsız veya eksik olarak ifade edilen bir gereksinim veya işlevselliğin, tasarım ve kodlama aşamalarında oluşan diğer hatalar gibi canlı bir hata olarak algılanması doğru bir yaklaşım olacaktır.

Programlama etkinliği, içinde hem teknik hem de sosyal unsurlar barındırır. Yazılım ürününü kullanan, analiz eden, geliştiren, test eden, yöneten ve pazara sunan; ürün mimarisini ilerleten ve ürüne destek veren, vb. herkes birer paydaştır. İster istemez, yazılım ürününün sadece belli bir yönü ile ilgilenebildiği için bütün resmin sadece bir parçasına odaklanabilen tüm paydaşların beklentilerini tutarlı olarak karşılayabilmek için yanal bir görüşe ihtiyaç vardır ki, uçtan uca izlenebilirlik sağlayan iyi bir yazılım yaşam döngüsü yönetimi sistemi bu ihtiyaca kısmen cevap verebilmektedir. Programlama uğraşı, tüm paydaşları ilgilendiren bütünün tutarlı ifade edilme sanatıdır.

Programlama kültürünün kendine has bir alt kültünü temsil eden salt kodcular ise, hasbelkader de olsa olaylara bütünsel bakabilme özelliğinini halen korumakta olan programcıların aksine bütünü görme özelliklerini neredeyse tamamen kaybetmek üzeredirler. Trygve Reenskaug, "programcıyı kodlamaktan alıkoyan herhangi bir yöntem iyi bir yöntemdir" derken belki de sadece "önce düşün sonra kodla" vurgusu yapmayıp bu yaraya da parmak basmak istemişti. Çok değil on yirmi yıl öncesine kadar revaçta olan analist programcılık, yerini aşırı uzmanlaşmaya bıraktı. "Full Stack Programmer" dediğimizde bile, gereksinimlerin belirlenmesi, analiz ve hatta tasarım aşamalarının pek kastedilmediği bir dönemde yazılım geliştiriyoruz artık.

Bu sorunun sosyal olduğu kadar oldukça teknik bir nedeni de var. Kodlama yaparken kullandığımız mevcut kavramlar ve araçlar, özellikle kullanılageldiği üzere nesneye yönelim, işlevselliğin biçemi olarak nesnelerin birlikte çalışabilirliğini kod içerisinde bütünleşmiş ve nesnelerin arasındaki iletişime odaklanmış olarak bize gösteremiyor.

Tez çalışmasında, programlamanın temelindeki bu önemli eksik çeşitli yönleri ve tezahürleriyle irdelenmiş, programcıların rol tabanlı eş zamanlı programlar geliştirebileceği ROCOCO adlı bir tümleşik geliştirme ortamı eklentisi geliştirilerek veriye olduğu kadar etkileşime de odaklanabilmenin yanısıra eş zamanlı davranışsal kodun kendine ait bir mahalde okunabilir olarak sunulabilmesinin mümkün olduğu bildiğimiz kadarı ile ilk kez gösterilmiştir. Umarız ki, bu tür gösterimler aracılığı ile programlamanın temelindeki bu önemli eksiği önce idrak sonra bertaraf etmek üzere anayol programlama dillerini tasarlayan takımlara etki etmek mümkün olacaktır.

Tony Hoare ne güzel söylemiş. "İki türlü tasarım vardır: biri hiçbir hata ihtiva etmeyeceği aşikar olacak kadar basittir, diğeri ise hiçbir hata aşikar olmayacak kadar karmaşık." Hedefimiz, karmaşık olan ama bir o kadar da mekanik altyapıları hatasız kurma işini ortama bırakarak, Hoare'nin sözünü ettiği basit ama zor tasarıma ulaşmak.

Zaman kısıtı içinde mümkün olan en iyi sonuca ulaşmak üzere beni daima yönlendiren değerli tez danışmanım Nadia Erdoğan'a, MVC ve DCI paradigmalarının vizyoneri Trygve Reenskaug ile birlikte DCI vizyonunun geliştirilmesine eş liderlik eden değerli Scrum Master eğitmenim James Coplien başta olmak üzere, birlikte yaptığımız sayısız tartışmalarla DCI yaklaşımını mevcut konumuna ilerleten object-composition grubunda fikirlerini sunan ve savunan tüm yazılım profesyonellerine, yaptığımız tartışmalarda sunduğu fikirlerle tezimin endüstri trendleri ile paralel yönlerini açığa çıkarmama yardımcı olan değerli arkadaşım ve danışman Ömer Demir'e, çalışmasını paylaşarak tezimin özgün katkılarına daha etkili bir şekilde ulaşmama katkı sağlayan Faraz Ahmadi Torshizi'ye, hep yanımda hissettiğim sevgili oğlum Ediz Balek'e, ve son olarak, en stresli anlarımda bile desteğini hiç esirgemeyen ve yaptığı özellikle İngilizce okumalar ve düzeltme önerileri ile fikrimi çok daha iyi sunmama destek olan sevgili eşim Eliz Balek'e sonsuz teşekkür ve minnet duygularımı ifade etmek isterim.

Haziran 2019 Cevat Balek

İÇİNDEKİLER

Sayfa

ÖNSÖZ ..... **vii**
İÇİNDEKİLER ..... **ix**
KISALTMALAR **11**
ŞEKİLLER **13**
ÖZET **15**
SUMMARY **17**
1. GİRİŞ **23**
1.1 Tezin Amacı 23
1.2 Problem 23
1.3 Gerçekten Nesneye Yönelememek 24
2. VERİ BAĞLAM ETKİLEŞİM VE EŞ ZAMANLI NESNEYE YÖNELİM **27**
2.1 Problemin Derinliği 28
2.2 Veri Bağlam Etkileşim Paradigması 28
2.3 Veri Bağlam Etkileşim Çalıştırma Modeli 31
2.4 Basit Eş Zamanlı Nesneye Yönelimli Programlama 33
3. ROCOCO İLE ROLE YÖNELİMLİ EŞ ZAMANLI PROGRAMLAMA **35**
3.1 Rol Tabanlı Programlama ile Eş Zamanlı Programlamanın Sentezi 36
3.2 ROCOCO ile Bağlam Sınıfı Tanımlama 37
3.3 ROCOCO ile Bağlam Sınıfının Yapılandırıldığı Kaynak Kod Satırı 40
3.3 ROCOCO ile Tezin Özgün Katkıları 41
4. ROCOCO GERÇEKLEME DETAYLARI **43**
4.1 Metaprogramlama 44
4.2 ROCOCO Bağlam Dönüşümü 46
4.3 ROCOCO Bağlam Yapılandırıcısı Dönüşümü 49
4.4 Gerçekleme ile Gösterilen Özgün Katkılar 50
5. TARTIŞMA VE SONUÇ **51**
5.1 Veri Nesnelerinden Kopamayan Davranış Kodu 52
5.2 Sonuç 53
KAYNAKLAR **55**
ÖZGEÇMİŞ **57**

KISALTMALAR

AOP : Aspect Oriented Programming
BDD : Behavior Driven Development
BPMN : Business Process Modeling Notation
DbC : Design by Contract
DCI : Data Context Interaction
(Veri Bağlam Etkileşim)
OIM : Object to -User- Interface Mapping
OOP : Object Oriented Programming
ORM : Object to Relational -Data- Mapping
REST : Representational State Transfer
ROCOCO : Role Oriented Concurrent Contexts
(Role Yönelimli Eş Zamanlı Programlama)
SCOOP : Simple Concurrent Object Oriented Programming
TDD : Test Driven Development
UML : Unified Modelling Language

ŞEKİL LİSTESİ

Sayfa

Şekil 2.1 : Veri Bağlam Etkileşim Vizyonu. **29**
Şekil 2.2 : Programcının Zihin Modelinde DCI Görünümü. **29**
Şekil 2.2 : Derleme Zamanındaki DCI Görünümü. **31**
Şekil 2.3 : DCI Çalıştırma Zamanı Görünümü **32**
Şekil 3.1 : JSCOOP ile Filozofların Akşam Yemeği. **35**
Şekil 4.1 : Dönüştürülen JSCOOP Kodu. **45**

ROCOCO İLE ROLE YÖNELİMLİ EŞ ZAMANLI PROGRAMLAMA

ÖZET

Rol tabanlı programlama ve eş zamanlı programlama konularında ayrı ayrı pek çok çalışma yapılmış ve pek çok araç geliştirilmiş olmasına rağmen, nesnelerin bir bağlam içerisinde birlikte çalışılabilirliğini özellikle eş zamanlı programlama açısından da ele alan çalışmaların sayısı azdır. Nesneye yönelim, rol tabanlı programlama ve nesneye yönelimli eş zamanlılık konularında literatür araştırması yapılarak, nesneye yönelimin en temel eksikliği, temelde sadece kalıtımı ve dolayısıyla sınıflar arasındaki hiyerarşik ilişkiyi baz alması nedeniyle işlevselliğin biçemini doğru yakalayacak bir kod yapısının oluşmasına mahal vermemesi olarak tespit edilmiştir. Kodda davranışı kolayca okuyup inceleyebilecek bağlama ait bir yerin mevcut olamaması nedeniyle, bu eksikliğin analiz ve tasarım diyagramları, test senaryoları ve test betikleri gibi kod dışındaki mecralar ile bertaraf edilmek zorunda kalındığı pek çok çalışmada dile getirilmiştir. Sistemin çalıştırma anında ne yaptığına dair programcının bilişsel hakimiyeti ve sorumluluk hissiyatı günden güne azalmaktadır. Hem nesnelerin birlikte çalışabilirliğini koda aynen yansıtabilmek hem de eş zamanlı programlamaya makul bir çözüm oluşturmak üzere, Data Context Interaction (DCI), Design by Contract (DbC) ve Simple Concurrent Object Oriented Programming (SCOOP) teknikleri, rol tabanlı ve eş zamanlı nesneye programlama yapmaya olanak veren ROCOCO tümleşik geliştirme ortamı eklentisinde birleştirilmiştir. Böyle bir eklentinin mevcut olmadığı günümüz tümleşik gelitirme ortamlarından bir adım ötede, örneğin bir kullanım senaryosu analizi ile oluşturulan UML eserlerinde kendini belli eden kullanım senaryoları, ön koşullar, garantiler, aktörler (roller) -ve dolayısıyla iletişim veya birlikte çalışma diyagramları- gibi daha önceden kodun içinde doğrudan yer bulamayan ve dolayısıyla kod ile kontrol edilemediği için programcının dolaylı olarak sorumlu hissetmediği kavramları da doğrudan kod içinde kullanarak çok daha sağlam ve okunabilir bir etkileşim kodu geliştirilebilecektir. Programcının hem davranışsal hem de eş zamanlılık anlamında ne yaptığı konusunda ek bilgilerin formal olarak tanıtılabildiği tümleşik geliştirme ortamı ise, programcıya daha reaktif uyarılarda bulunarak yapısal hataların oluşmasına mahal vermeyecek ve kodun eş zamanlı olarak beklendiği gibi çalışması için gereken kaynak kod dönüşümünü yapabilecektir. Bu çalışma ile, DCI çalıştırma modelindeki tek ipliklilik kısıtı da ilk kez kaldırılmıştır.

PROGRAMMING IN ROLE ORIENTED CONCURRENT CONTEXTS WITH ROCOCO

SUMMARY

Despite the huge number of studies carried out and tools developed for role oriented programming and concurrent programming, there are only a few studies addressing the interoperability of concurrent objects in concurrent contexts. Based on a comprehensive research on object orientation and role oriented programming, the most fundamental deficiency of object orientation has been identified as the absence of a locus in code to capture the form of function because object orientation as we know it fundamentally and even solely rely on inheritance and hierarchical relationship between classes. In many studies, it is shown that this deficiency of lacking a place in code that can easily be read to examine the behavior is eliminated by means of non-code channels such as analysis and design diagrams, test scenarios and test scripts, etc.

The programmer's cognitive mastery and sense of responsibility about what the system will do at run time decreases day by day. In this study, Data Context Interaction (DCI), Design by Contract (DbC) and Simple Concurrent Object Oriented Programming (SCOOP) techniques are combined in ROCOCO (Role Oriented Concurrent Contexts) eclipse plugin extention to reflect the interoperability of concurrent objects in the code.

One step ahead of today's integrated development environments where no such plug-in exists, a much more robust and readable interaction code can be developed by being able to code concepts that have not previously been directly located in the code and therefore could not be controlled by the code thus prevented the programmer from feeling responsible for what she really can’t directly read in the code. An integrated development environment where additional information can be formally introduced about what the programmer is doing in both behavioral and concurrent terms is able to convert the source code to run concurrently as programmer depicted and will not let structural errors show up by giving more reactive warnings to the programmer. One of the greatest contribution of this thesis is the relaxation of the single-thread constraint in DCI execution model by synthesizing DCI with SCOOP and DbC ideas. ROCOCO reduces the role orientation which is done in DCI way to object orientation by transforming the code just after compilation time, and SCOOP handles concurrency for this object oriented code without losing the interactivity of IDE and programmer.

Remembering the visions of scientists who laid the foundation for object orientation half a century ago is very important in terms of understanding where the object orientation is actually aimed at. Alan Kay, one of the visionaries of object orientation, states the necessity of designing the computer as an extension of human mind, and adds that a good information system, a good education system and even a good programming environment should more or less understand what the end user, student, and even the programmer as the current user of the system is trying to accomplish through the connection of the computer and human mind. He spoke of the importance of creating a sense of immediacy by not letting too much delay in giving information, showing the result of what was done or warning about what is about to be done wrong. Trygve Reenskaug, the inventor of the Model View Controller (MVC) method, which is as old as object orientation, manifested MVC as a link between the human mind and the computer and stated that a good user interface allows direct manipulation of objects stored in computer's memory by the end user.

Advances in techology and ever increasing compuational power opens up new frontiers for programming world too. We can now start to speak about programs generating other programs, programs generating test cases by constraining the case space and even tools to prove that the code will completley work as expected or not by exploiting the formal specifications which were provided by the programmer. The modern integrated development environments (IDE) should be able to intract more throughly with programmer by immediately recognizing the impacts of her current amendments in the source code and warn her if anything may go wrong. This dialogue like interaction between IDE and programmer is important for maintaining the focus of the programmer on the code without waiting to observe what the software will do at run time. Thesis provides ROCOCO (Role Oriented Concurrent Contexts) as a plug-in to provide an environment for the programmer to code with full support on both role oriented and concurrent programming from the IDE which understands the intent of her by exploiting annotations and warns her at complite time about the structural errors her recent amendments on code will cause without the need to test the code at run time.

James Coplien and Trygve Reenskaug spoke about how the pendulum in object orientation has shifted since the mid-1990s, and how a mechanical software architecture that focuses more on software quality metrics, coupling and cohesion than on the pairing with human mind moved us away from the essence of object orientation.

Maria Siniaalto and Pekka Abrahamsson's work on Test Driven Development (TDD) not only supports this rhetoric on the over-importance of coupling and cohesion, but also gives a good idea of how big the unreadable behavioral code can actually be in the source code. For good testing of a source code, the TDD code being written is twice or three times the size of the original source code. This has two consequences:

1. The programmer should be responsible for the behavioral code which can only be exposed now as test scripts. Unaware of this responsibility, the programmer left the work to the tester with peace of mind. Worse, no one is aware of this situation.

2. At least two or three times the code maintenance required for the original program to run correctly must be performed for the test code to ensure that the test code is working and testing correctly.

When the code does not reveal exactly to how the sofware will behave at run time, the behavior is expressed with various analysis and design artifacts such as Unified Modeling Language (UML) collaboration and UML activity diagrams. Test scripts were coded with the xUnit family (JUnit, NUnit, etc.) to make sure that the software behaves as desired. The point we want to emphasize with these examples is that the deep need for all these techniques points to a very basic problem, a fundamental deficiency in object orientation; the absence of a place in code to describe behavior.

ROCOCO exploits the fact that the communication between the context and the data objects which play some role in that context is one way in DCI way role orientation. Data objects are not aware about the contexts they may play a role within, but contexts know at construction time all it needs to know about each data object as a role player. In DCI, formation of contexts is atomic, i.e. all data objects are bound to the context as role players at the time when the context is constructed. Therefore, we can take the source code line where the context instance is constructed as the reference for ROCOCO transformation to reduce the role oriented code to object orientated code. As programmer continues to use the IDE, each context construction code is visited by Java Development Toolkit (JDT) observers of the ROCOCO plugin. The specialized context class will be suffixed with _asCalledFrom_SourceCodeFileName_Line# and role classes in that context class will be rename refactored with data object classes which will play those roles at that specific source code line by the ROCOCO plug-in.

On each such line, different types of contexts which will employ different types of role player data objects are possible. Since the communication between contexts and data objects as role players is one way in DCI, we can use each such line as an anchor and let the ROCOCO plug-in generate the necessary context classes and generate the modified client code which is constructing that specific context on that specific source code line. ROCOCO makes a necessary assumptiong as a constraint that each context has only one interaction code which is called *maestro()*. Since there is only one interaction code, ROCOCO can immediately generate the code to call *maestro()* method just after the it visits the reference source code line for context construction.

Suppose that the data objects are created by the programmer as follows:

```
@separate Hesap kaynak = new Hesap(IBAN_kaynak);
@separate Hesap hedef = new Hesap(IBAN_hedef);
@separate Para para = new Para(50);
```

Also suppose that the context in which these data objects will be role players are:

```
@RolePlayers(
        mappings = {
                "Source=kaynak",
                "Destination=hedef",
                "Banknote=para"
                },
        adapters = {
                "Destination.credit = _Destination.deposit(b);"
                }
        )
@separate MoneyTransfer moneyTransfer = new MoneyTransfer();
```

A programmer who encounters the above code will recognize that role orientation is being used at this point, even if she has no knowledge of the subject. Programmer clearly understands the intent of the code by looking at the @RolePlayers annotation. Programmer will also notice that, although the role classes are Source, Destination and Banknote, the data objects which will play these roles will be of type Hesap and Para. ROCOCO visits necessary nodes, which are in this case variable declaration nodes, in JAVA Abstract Syntax Tree to find declared types of data objects. For example, Source role is mapped onto kaynak object which is decpared as type of Hesap. Also, the SCOOP @separate annotation specifies that the context itself can live concurrently on a different processor.

With ROCOCO, after specifying the role oriented and concurrent intent of her code, programmer feels safe that IDE will warn her whether there is a structural problem or not in her code which may lead to a problem or crush at run-time.

Following code is an example of the only interaction code in the context, *maestro()*.

```
public class ROCOCO_MoneyTransfer {
        …
        @await(condition="!s.isBusy()&&!d.isBusy()&&!b.isBusy()
        &&s.acquire()&&d.acquire()&&b.acquire()")
        public void maestro (
                @separate Hesap s,
                @separate Hesap d,
                @separate Para b {
                        b.setBanknote += s.commission(b);
                        s.debit(b);
                        b.setBanknote -= s.commission(b);
                        d.credit(b);
                        s.release();d.release();b.release();
                }
        }
        …
}
```

Here, another SCOOP @await annotation is used to specify the await condition.

The main contribution of the thesis is to demonstrate that it is possible to build an interactive development environment which will let the programmer specify not only the concurrent or role oriented code alone but specify both role oriented and concurrent code together, and to relax the single-threaded contstaint of DCI by this synthesis.

We hope that this thesis work, which is based on annotations only, will find a response in industry and we will have more realiable interactive development environments.

1. GİRİŞ

Giriş bölümünde, tez çalışması ile prototip bir çözüm sağlanması hedeflenen problem, yani -bildiğimiz ve evrildiği hali ile- nesneye yönelimde davranışsal kodun kendine ait bir yer edinememesi problemi incelenip irdelenerek, önerilen çözüm ile birlikte getirilen özgün katkılar özetlenecek ve tezin kavramsal çerçevesi çizilecektir.

1.1 Tezin Amacı

Amaç, mevcut nesneye yönelimdeki en temel eksiklik olduğu tartışılan 'davranışsal kodun kendine ait bir mahal bulamaması' problemi ile birlikte role yönelimli eş zamanlılık problemine de özgün bir çözüm getiren bir sentez sunmak ve bu sentez yaklaşımın gerçeklenebilirliğini ve bu sentezin DCI (Data Context Interaction) tek ipliklilik kısıtını da kaldırıldığını ROCOCO (Role Oriented Concurrent Contexts) tümleşik geliştirme ortamı eklentisi gerçeklemesi ile göstermektir.

1.2 Problem

Nesneye yönelim (object orientation) kavramının vizyonerlerinden Alan Kay der ki:

> Bilgisayar terimleriyle Smalltalk, bilgisayarın kendisi üzerine bir özyinelemedir. 'Bilgisayar öğelerini', her biri bütünden daha az güçlü olan programlama dillerinin normal teçhizatları olan veri yapılarına, prosedürlere ve fonksiyonlara bölmek yerine, her Smalltalk nesnesi, bilgisayarın tüm olanakları üzerine bir özyinelemedir. Bu nedenle, çok hızlı bir ağ tarafından birbirine bağlanmış binlerce ve binlerece bilgisayar kullanmaya oldukça benzeyen bir semantiği vardır (Kay, 1993).

Mevcutta uyguladığımız haliyle nesneye yönelim, bu orijinal vizyondan oldukça uzaktadır. Bildiğimiz nesneye yönelimin ana problemi, özünde nesne değil sınıf odaklı olmasıdır. Sınıfların örnekleri olan nesneleri davranışsal olarak da genişletmenin tek temel yolu, bir baz sınıftan miras yoluyla alt sınıflar türetmektir. Daha çok kategorizasyonu çağrıştıran hiyerarşi ve miras kavramları nesnenin veri perspektifi için oldukça anlamlı iken, davranış perspektifiyle gerçekten büyük bir uyumsuzluk içindedir. Mevcut nesneye yönelim hem verileri hem de davranışları farklılaştırmak için temel olarak yalnızca miras mekanizmasına sahip olduğu için, davranışsal kodun çeşitli alt sınıflara dağılmasının önüne geçecek bir mekanizma mevcut değildir.

1.3 Gerçekten Nesneye Yönelememek

Yarım asır önce nesneye yönelimin temelini atan bilim insanlarının vizyonlarını hatırlamak, nesneye yönelim ile aslında ne hedeflendiğini anlamak açısından oldukça önemlidir. Örneğin, yine Alan Kay, bilgisayarın insan zihninin bir uzantısı olarak tasarlanması gerekliliğinden, bilgisayarın ve insan zihninin bağlaşımı aracılığı ile iyi bir bilgi sisteminin, iyi bir eğitim sisteminin ve hatta iyi bir programlama ortamının, kendisiyle iletişime geçen son kullanıcının, öğrencinin ve hatta programcının ne yapmaya çalıştığını az çok kavrayabilmesinden, bilgi verme, yapılanların sonucunu gösterme veya yanlış yapılmak üzere olunanlar konusunda uyarma konusunda çok geciken tepkiler vermeyerek anındalık hissiyatı yaratmasının öneminden bahsetmiştir (Kay, 1972). En az nesneye yönelim kadar eski olan Model View Controller (MVC) yönteminin mucidi Trygve Reenskaug da MVC'yi insan zihni ile bilgisayarın bir bağlaşımı olarak tezahür etmiş, iyi bir kullanıcı arayüzünün son kullanıcının bilgisayarda saklanan nesneleri doğrudan değiştirebilmesine (direct object manipulation) olanak sağlaması gerekliliğine işaret etmiştir (Reenskaug, 1995). Richard Pawson, günümüzde popüler olan Object to Relational -Data- Mapping (ORM) benzeri olan Object to -User- Interface Mapping (OIM) işlemini Naked Objects adını verdiği yöntemle mümkün kılarak, İrlanda hükümetinin çok sık değişen mevzuatına bağlı olarak sürekli değiştirilmesi gereken mevzuatı temsil eden nesne kodlarından kullanıcı arayüzünü otomatik olarak elde ederek nesnelerin doğrudan değiştirilebilmesi vizyonunu en bariz biçimiyle hayata geçirmiştir (Pawson, 2004). Hydra, Siren gibi güçlü alternatiflerine rağmen, Naked Objects ile tam uyumlu Restful Objects Specification, REST ile sadece verinin değil tüm nesne modelinin bir kaynak olarak sunulabileceğinin ilk gösterimlerinden birisiydi (Haywood, 2013).

James Coplien ve Trygve Reenskaug, nesneye yönelimde sarkacın 1990'ların ortalarından bu yana diğer yöne kaydığından, insan zihniyle bilgisayarın eşleşmesindensen çok bağlaşıma ve kohezyona önem veren mekanik bir yazılım mimarisi ile yazılım kalite metriklerinin nesneye yönelimin esasından uzaklaşmaya nasıl neden olduğundan bahsederler (Coplien ve Reenskaug, 2009). Maria Siniaalto ve Pekka Abrahamsson'un Test Driven Development (TDD) üzerine yaptığı çalışma, bağlaşım ve kohezyona aftedilen aşırı önem üzerine olan bu söylemi destekler nitelike olmakla kalmaz, yazılımın kaynak kodunda doğrudan okunamayan davranışsal kodun gerçekte ne kadar büyük olabileceğine dair iyi de bir fikir verir.

Bir kaynak kodun iyi test edilebilmesi için yazılmakta olan TDD kodu, orijinal kaynak kodun iki veya üç katı büyüklüğündedir (Siniaalto ve Abrahamsson, 2007).

Bunun iki anlamı vardır:

1. Ayrı test betikleri gerektiren davranışsal koddan asıl sorumlu olması gereken programcıdır. Bu sorumluluğun farkında olmayan programcı gönül rahatlığı ile işi testçiye bırakmıştır. Daha da kötüsü, kimse bu durumun farkında değildir.
2. Test kodunun da doğru çalıştığından ve doğru test ettiğinden emin olunması için programın doğru çalışması için yapılması gereken kod bakımının en az iki veya üç katı da test kodunun doğru çalışması için yapılmalıdır.

Dahası, çoğunlukla yazılımın gelişimi ile aynı hızda güncellenemeyen testler sürekli yeşil ışık yakıp, herşeyin yolunda olduğuna ve sistemin hatasız çalışmakta olduğuna dair hatalı ve oldukça tehlikeli bir çıkarım yapılabilmesine de sebebiyet verebilir.

Bir yazılımın çalıştırma anında nasıl davranış göstereceğinin birebir karşılığı kodda yer almayınca, Unified Modeling Language (UML) birlikte çalışma ve UML etkinlik diyagramlarındaki gibi çeşitli analiz ve tasarım eserleri ile davranışlar ifade edilmeye çalışılmış, yine yazılımların en azından test edilebildiği kadar istenilen davranışları gösterdiğinden emin olmak için xUnit (JUnit, NUnit, vb.) ailesi ile test betikleri kodlanmış ve buradan Test Driven Development (TDD) doğmuştur (Bjørnvig ve diğerleri, 2007). TDD kültürünün başlı başına yazılım mimarisini aşındırdığına dair çalışmalara kısaca işaret etmekle birlikte Behaviour Driven Development (BDD) ve Executable Documentation gibi teknik şartnamaleri çalıştırabilir dokümanlar olarak ifade etmeye yönelik çeşitli yöntemlere başvurulmasının altındaki neden barizdir.

Yazılımın hangi koşulda nasıl davranacağına sadece kaynak koda bakarak anlayamayacak durumdayız (Reenskaug, 2007). Davranışı değiştirmek veya baştan aşağı tanımlamak üzere başvurulan konfigüyasyon dosyaları, konvensiyonlar, cepheye yönelim (aspect orientation) ve BPMN (Business Processing Modeling Notation) gibi tekniklere duyulan ihtiyaç içinde bulunduğumuz tipik durumu özetler. Bu tekniklerin zararlı veya yararlı olup olmadıkları başka bir tartışmanın konusudur. Bu örneklerle vurgu yapmak istediğimiz konu, bu tekniklere duyulan derin ihtiyacın, kodlama pratiğinde davranışları dolambaçsız olarak betimlemek için kod içinde bir yer mevcut olmaması ile ilgili çok temel bir soruna, temel bir eksikliğe işaret etmeleridir.

2. VERİ BAĞLAM ETKİLEŞİM VE EŞ ZAMANLI NESNEYE YÖNELİM

Nesneler arasındaki etkileşim kodunun kendine ait bir yer bulamaması sorununu çözmek için arayüzler (interface) ve cepheye yönelim (aspect orientation) gibi kavramlar ile birlikte OT/J (Hermann, 2000) ve JAWIRO (Selçuk ve Erdoğan, 2011) gibi role yönelimli dil uzantıları başta olmak üzere pek çok çözüm geliştirilmiştir. Tüm bu çözümler, sorunun bir bölümünü çok iyi çözerler ve zengin özellikler sunarlar. Ancak, ileride detaylandıracağımız üzere, davranışın biçemini doğru yakalamak için bağlam ve rol gibi yeni olmasa da hak ettikleri rağbeti pek görmemiş kavramlara dayanmak gerektiği kadar, bir bağlam içindeki davranış için anlamı olmayan çok şekillilik gibi kavramları da budamak gereklidir. Rol tabanlı yaklaşımlar ve araçlar üzerinde bir literatür araştırması yapılmış ve davranışın biçemini doğru yakalamaya yönelik hem genişletmeyi hem de kısıtlamayı bir arada öneren ilk yaklaşımlardan biri olan Veri Bağlam Etkileşim (DCI) yaklaşımı role yönelimli eş zamanlılık probleminin çözümünün de ilk ayağı olarak baz alınmıştır (Reenskaug ve Coplien, 2009).

Yalın mimari ile mükemmel bir uyum içinde olan Veri Bağlam Etkileşim paradigması, karmaşıklaştırma eğiliminden kurtulabilmemizi ve okunabilirliği yüksek olan çok basit tasarımlara ulaşabilmemizi daha da mümkün kılacak derin bir potansiyele sahiptir. Nesneye yönelim, programcının ve son kullanıcının perspektiflerini kod içinde birleştirerek bir taşla iki kuş vurmayı, yani hem kullanılabilirliği hem de programın anlaşılabilirliğini aynı anda sağlamayı hedefliyordu. Hakim yazılım geliştirme dillerinin orijinal nesneye yönelim vizyonundan uzaklaşıp nesneden çok sınıfa yönelimli olmasının da getirdiği kısıtlamalar nedeni ile, nesneler yapıları iyi yansıtabilmelerine rağmen sistem etkileşimlerini yakalamada başarısız olmuşlardır.

Nesneye yönelimin esası, iletişim halindeki nesnelerin oluşturduğu ağların ortak bir hedefi başarmak üzere birlikte çalışmalarıdır. Nesneye yönelimli programlama sağduyusu, bunun nasıl çalıştığını betimleyen kodlar yazmamızı gerektirir. Ne yazık ki, farklı tercihte bulunarak kodu genellikle sınıflar halinde yazdık. Bir sınıf, örnekleri olan nesnelerin özellikleri hakkındaki her şeyi söyler; fakat ne bu örneklerin birlikte etkileşerek sistem davranışını nasıl gerçekleştirdiğini, ne de nesneler ve aralarındaki ilişkiler olarak sistem durumunun nasıl temsil edildiğini net olarak söylemezler. Sonuç, kodumuzun sistemin nasıl çalışacağı ile ilgili her şeyi açığa kavuşturamaz durumda olmasıdır ki, bu durum açıkça tatmin edici değildir (Reenskaug, 2009).

2.1 Problemin Derinliği

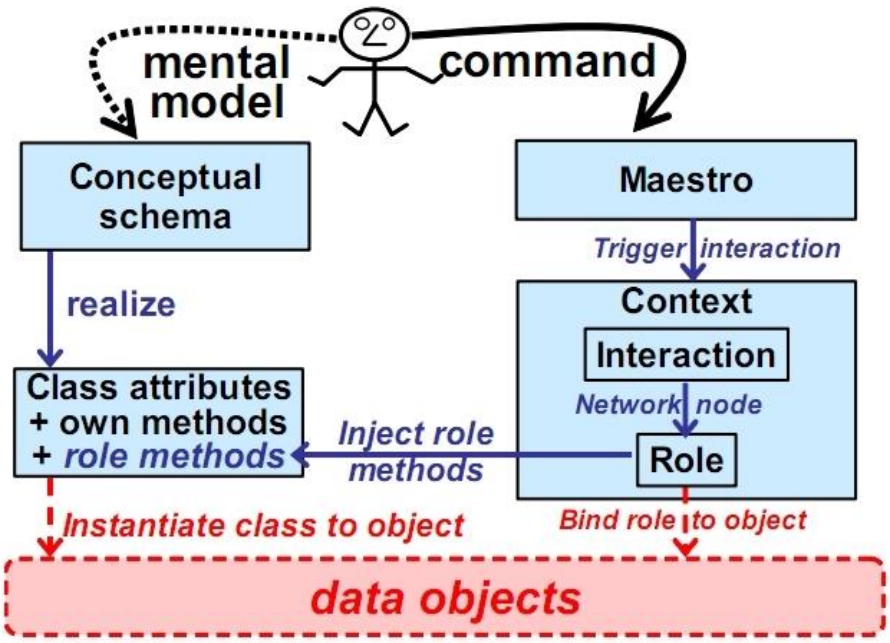
Nesnelerin birlikte çalışmasını gerektiren herhangi anlamlı bir işin tamamlanması için toplamda onlarca hatta yüzlerce satırın koordineli bir şekilde çalışması gerektiği halde, mevcut haliyle nesneye yönelimde her nesneye ait metodlardaki çalıştırılabilir komut satırı sayısının çoğu zaman iki elin parmaklarını geçmediğine ve bu yüzden onlarca hatta yüzlerece nesneye dağılmış kodun okunup kavranabilmesinin çok zor olduğuna dikkat çeken Valdecantos, DCI ile -bildiğimiz ve uyguladığımız haliyle- nesneye yönelimi kod okunabilirliği açısından deneysel olarak karşılaştırdığı çalışmasında, kod okumak için ayrılan sürenin her iki yaklaşımda farklı olabileceğine dair anlamlı bir istatiksel veri elde edememesine karşın, kod kavranabilirliği ve dikkatin odaklanması açısından DCI ile dikkate değer bir üstünlük sergilenmekte olduğu sonucuna varmıştır (Valdecantos, 2016).

Nesneye yönelim, insanlar ve zihin modelleri hakkında düşünebilmek ve koda yansıtabilmek prensibi ile ortaya çıktığı halde; nesneleri veri ve veri üzerindeki etkin kodlar olarak ele alıp, çok biçimliliği, bağlaşımı ve kohezyonu ön planda tutan diğer kamp üstün gelmiştir. Ancak, yapı formu dondurur. Formdan yapıya doğru ilerleme sürecinde alınan her karar, ister istemez esastan bir miktar uzaklaşılmasını gerektirir (Coplien, 2010). Derleyicilerin hakim ana görüşü destekleyecek şekilde sınıfa yönelimli ve dolayısıyla donuk olarak yapılaşmasının, nesneye yönelimin esasından gittikçe uzaklaşılmasını beraberinde getirdiğini rahatlıkla söyleyebiliriz.

2.2 Veri Bağlam Etkileşim Paradigması

Veri Bağlam Etkileşim paradigması, nesneye yönelimin esasını yakalayabilmek için, programı her biri belli bir sistem özelliğine odaklanan üç değişik perspektife ayırır.

Veri perspektifindeki kodlar, sistem durumunun gösterimini belirtir. Bağlam perspektifindeki kodlar, çalıştırma anında iletişim halindeki nesnelerin oluşturduğu ağları belirtir. Etkileşim perspektifindeki kodlar, ağı oluşturan nesnelerin sistem davranışını gerçekleştirmek üzere birlikte nasıl çalıştıklarını belirtir (Reenskaug ve Object Compostion Grubu, 2014).

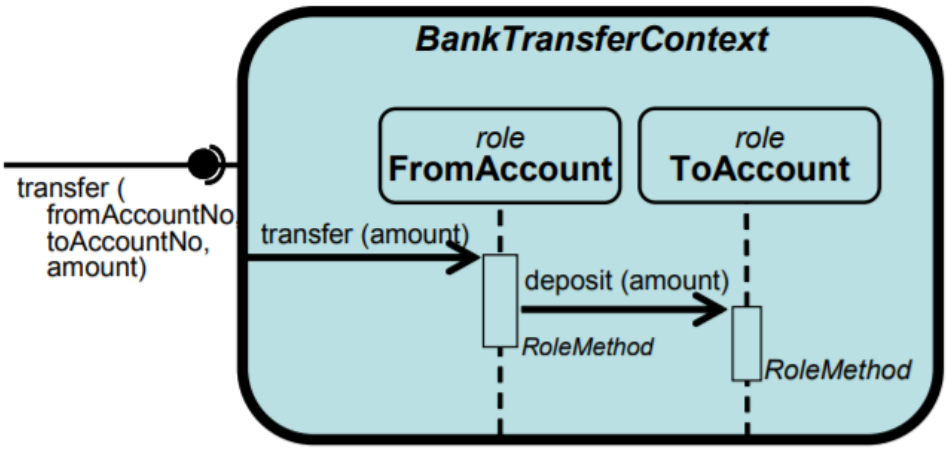


Şekil 2.1: Veri Bağlam Etkileşim Vizyonu (Reenskaug ve Coplien, 2009)

Veri-Bağlam-Etkileşim mimarisi; rol modelleme, iş birliği (collaboration) ve trait kavramlarına dayalıdır (Schärli ve diğerleri, 2003). UML, iş birliğini şöyle tanımlar:

"İş birliği, her biri özelleşmiş bir fonksiyonu yerine getiren unsurların (rol) birlikte çalışma yapısını tanımlar. Temel amacı, sistemin nasıl çalıştığını açıklamaktır; dolayısıyla gerçeğin sadece bu açıklamaya ilişkin olduğu farz edilen yönlerini dikkate alır. Katılımcı nesnelerin kimlikleri veya kesin sınıfları gibi detaylar gizli tutulur."

İş birliği, tam da role yönelimli programlamanın ısrar ile üzerinde durduğu konudur. Veri Bağlam Etkileşim çalıştırma modelinde, davranış programcının zihin modelinde aynen UML iletişim veya izomorfik iş birliği diyagramında yer aldığı gibi canlanır:



Şekil 2.2: Programcının Zihin Modelinde DCI Görünümü (Reenskaug, 2010)

Veri Bağlam Etkileşim paradigmasının temel katkısı, gerçekten de kullanıcının zihin modelinde yer aldığı şekliyle bir nesnenin, kod içinde kendisine has varlık bilgisini anlamlandıracak olan kendi metodlarına ek olarak; o andaki bağlamın belirlediği etkileşimde oynayacağı role ait rol metodlarının kendisine yerleştirilmesi aracılığıyla, gerçek anlamda nesneye yönelimin sağlanabilmesidir.

Gerçekten nesneye yönelimin bahsettiğimiz bu dinamik doğasından biraz taviz vermek pahasına da olsa, aracı (mediator) deseni kullanılarak da oldukça iş görür bir tasarıma ulaşılabilir. Veri Bağlam Etkileşim mimarisinin çözümlediği asıl konu, gerçekten nesneye yönelebilmek için gereken bağlam ve rol boyutlarını betimlerken davranış kodlarında çok şekilliliğe izin vermeyen kavramsal çerçeveyi de ihmal etmemesidir.

Böylece, kod, ilgili kullanım senaryosundan neredeyse kelimesi kelimesine türetilir. Böyle olması, kodu kullanıcının zihnindeki mantığın pek çok sınıfa rasgele biçimde yayıldığı -nesneye ve senaryoya değil- sınıfa yönelimli tasarımlardan çok daha okunaklı kılmaktadır. DCI sözlüğü (Reenskaug ve Object Composition Grubu, 2014), DCI çalıştırma modeli (Reenskaug, 2010) ve üzerinde araştırmalar yapılabilebilmesi için bir DCI referans gerçekleştirimi DCI yaklaşımını tanımlar (Coplien, 2015).

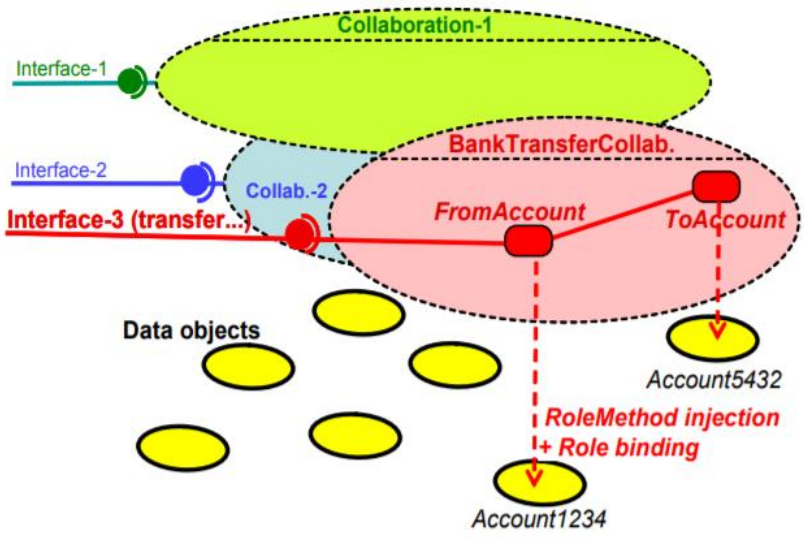
Yapılmakta olan yapısal hataları da programcıya anında gösterebilen gerçek bir tümleşik geliştirme ortamı sunarken DCI çalıştırma modelindeki önemli bir eksiği de bertaraf ederek, yani hem pratik hem bilişsel hem de teknik bir çözüm getirerek, programlamadaki temel eksikliğinin neden ve nasıl çözülmesi gerektiğine dair tezimizi ortaya koyuyoruz.

Konu uzmanları ile birlikte, sistemin ne olduğunun alan modelinin oluşturulması ve sürekli iyileştirilmesi ile, üzerinde tutarlı yapıların oluşturulabileceği soyut sınıflar olarak beliren sistem formu oluşur. Sistemin ne yaptığı, kullanıcı isterlerin belirlenmesi, kullanılabilirlik analizinin yapılması ve senaryoların oluşturulması ile netleşir ve alan modeli ile uzlaşarak şekillenir. Veri Bağlam Etkileşim paradigmasında, verilen bir rolü yerine getiren tüm nesneler aynı etkileşim mesajlarını aynı metodlarla işlerler. Bu metodlar rol metodlarıdır ve role ait özelliklerdir. Rol metodları veri sınıflarına yerleştirilirler (inject) ve veri sınıfları yerleştirilen bu metodları ezemez (override). Böylece, kodu okuyan, kodun içinde gizli hiçbir sürpriz olmadığına tam olarak güvenir. Çözümlenen temel eksiklik en saf anlatımıyla budur.

2.3 Veri Bağlam Etkileşim Çalıştırma Modeli

Diğer rol tabanlı dil uzantılarında olduğu gibi, DCI, algoritmayı veya prosedürü (etkileşimi), varlık veya sınıfı (veri) temsil eden koddan ayırır. Bunu yaparak, DCI, ayrışan yalın mimariyi etkinleştirir ve teşvik eder. Her ikisi de doğal olarak çok farklı hızlarda evrimleşen veriyi (sistemin ne olduğunu) ve işlevselliği (sistemin ne yaptığını, yani ne işe yaradığını) betimleyen kodlar aynı sepete konulmamalıdır.

DCI, polimorfizm kullanımına yalnızca en çok ihtiyaç duyulan ve en etkili olan yerde, varlıkları (verileri) tanımlarken izin vermesi açısından rol tabanlı diğer tekniklerden oldukça farklı bir konumdadır. Davranış kodlarında polimorfizm kesinlikle yasaktır, dolayısıyla davranış kodları doğrudan okunabilirdir. Plant UML aracnın liderlik ettiği ve gittikçe olgunlaşan metinden diyagrama dönüştürme teknikleri ile davranış kodlarını hem metin hem de görsel olarak aynı anda sunmak da mümkün olabilir.



Şekil 2.3: Derleme Zamanındaki DCI Görünümü (Reenskaug, 2010)

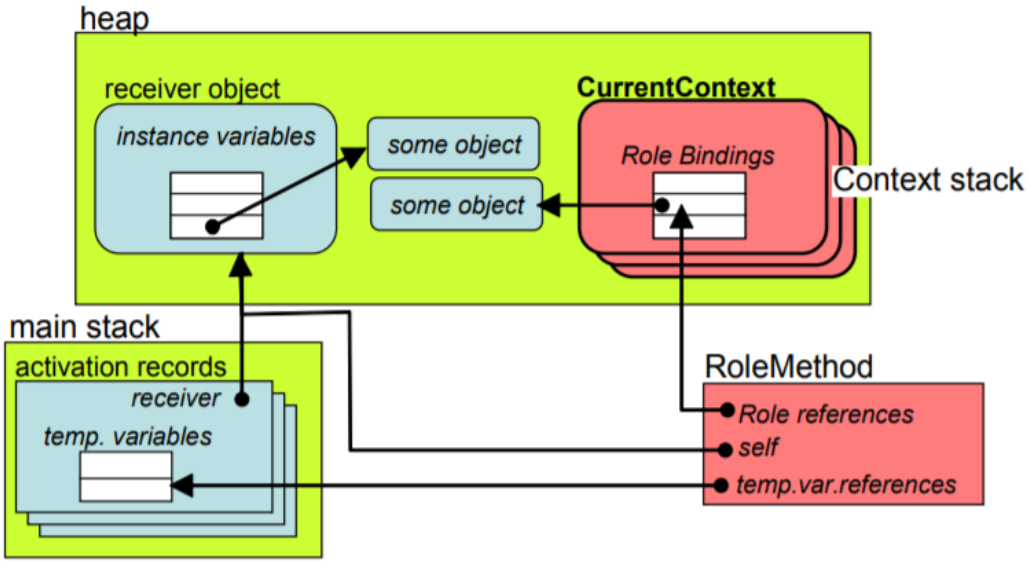
Derleme zamanındaki DCI görünümünün özeti, bağımsız olan veri nesneleri ve nesneler arasındaki farklı etkileşimler olarak iş birliklerini ifade eden bağlamlardır. Bağlamlar ile veri nesneleri arasındaki ilişki, bağlamdan nesneye doğru tek yönlüdür. Gevşek bağlanan veri ve bağlam, kodda kendilerine ait ayrı yerler bulabilmişlerdir. Örneğin, bir nesne uyumsuz olduğu için bir etkileşimde yer alamıyorsa, aynı işi yapan başka bir etkileşim kodunun içinde pekâlâ yer alabilir. Aynısı etkileşim için de geçerlidir. Bir etkileşim, aynı veriyi kendisine uyumlu başka bir nesne aracılığı ile de elde edebilir. Rol kontratı, her iki yakayı birbirlerine gevşek bağlayan bir arayüzdür.

Veri Bağlam Etkileşim (DCI) mimarisinde;

• veri, kökü alan sınıflarında mevcut olan alan nesnelerinde yaşar,

• bağlam, talep olduğunda alan nesnelerini senaryodaki pozisyonlarına getirir,

• etkileşim, son kullanıcının zihninde yer alan rolleri kullanarak yine son kullanıcının zihninde tezahür ettiği haliyle algoritmaları ifade eder.



Şekil 2.4: DCI Çalıştırma Zamanı Görünümü (Reenskaug, 2010)

Mevcut haliyle DCI çalıştırma modelinin önemli bir eksikliği vardır. DCI çalıştırma modelinin bağlamların eş zamanlı programlanmasına müsaade etmeyen tek-iplikllik (single-thread) kısıtını kaldırmak üzere, hem diğer mantıksal işlemcilerle eş zamanlı olarak hem de kendi mantıksal işlemsicinde sıralı olarak çalışan Simple Concurrent Object Oriented Programming (SCOOP) modeli baz alınarak koddan koda dönüşüme başvurulmuş, DCI tekniğinin eş zamanlı programlamada kullanımının önü açılmıştır. Böylece, SCOOP yaklaşınımı ilk kez öneren ve dilin bir özelliği olarak programcılara sunan köklü bir nesneye yönelimli programlama dili olan Eiffel dilinin kendisinde de çözümlenmemiş olan bu konuya, nesnelerin bağlam içinde aldıkları rollere yönelimli olarak eş zamanlı çalışabilirliklerinin sağlanmasına, bildiğimiz kadarıyla ilk kez çözüm getirmekteyiz. Adil bir eş zamanlı yürütme için yazılması gereken oldukça karmaşık teknik altyapı koduna gereksinimi kalmayacak olan programcının tek-iplikli ve çok-iplikli çalışma modelleri arasında bilişsel düzeyde de kopukluk yaşamayacak olmasının göz ardı edilmediğine özellikle vurgu yapmak istiyoruz.

2.4 Basit Eş Zamanlı Nesneye Yönelimli Programlama

Simple Concurrent Object Oriented Programming (SCOOP), tek-ipliklli ve çok-iplikli programlamayı aynı kod içerisinde bilişsel bütünlüğü bozmadan ve dikkat dağıtmadan etkileşimli olarak yapabilen programcının belirttiği spesifikasyonlara uyan eş zamanlı C kodunu üretme esasına dayalıdır (Brooke, 2007). Böylece, programcının eş zamanlılık problemlerini çözmesindense elindeki iş problemlerine odaklanıp algoritmaları çözmesi hedeflenmiştir. Programcı basit eş zamanlı programlama ihtiyaçlarını önnotlar (annotations) ile belirtir, eş zamanlıklık işini yapısal bir sorun olmayacağını garanti eden SCOOP halleder.

Design by Contract (DbC) (Meyer, 1992) ile birlikte çalışan SCOOP ilkelleri basittir;

- Bir nesne ya onu oluşturan işlemci içinde ya da başka bir işlemcide yaşar,
- Bir yordam ya onu çağıran işlemci içinde ya da başka bir işlemcide çalışır,
- DbC "koşullar gerçeklenene kadar yordamı bekletme" anlamında kullanılır.

SCOOP, programcıdan *@separate* önnotu (annotation) aracılığı ile bir nesnenin veya o nesneye ait yordamın aynı işlemci de mi farklı işlemci de mi çalışacağı ile birlikte eş zamanlı çalışmak üzere işaretlenen nesneye ait yordamların bekleme koşullarını belirtmesini ister. Tüm senkronizasyon işini SCOOP halleder. SCOOP, her temsili işlemcinin kendi işlerini sırayla yapıyor olması esasına dayanır.

Tek-iplikli çalışmak üzere tasarlanan bir nesne için, eğer o nesnenin kendisi, başka işlemcilerde çalışabilme tercihini belirtmiş olan nesneler oluşturmuyor, böyle nesneleri parametre olarak almıyor veya atama işlemlerinde eş zamanlı nesneleri kullanmıyor ise kaynak kod dönüşümüne tabi olmaz. Kaynak kod dönüşümü esas itibarı ile, bir fonksiyonun geri dönmesini gereklilikten-dolayı-bekleme (wait-by-necessity) veya bir metodun (etkileşimin) çalışmasını ön koşulun gerşekleşmesine kadar bekleme (await) gibi konuları adil olarak çözümleyen global bir zamanlayıcı ile birlikte çalışarak bekleme yapması söz konusu olan atama ve çağrı kodunun önüne semafor koyma ve iş bitince semaforu bırakma işini yapar. Eğer başka işlemcideki bir nesnenin değer dönmeyen bir metodu çağrılıyorsa, eş zamanlı çalışılıyor demektir.

3. ROCOCO İLE ROLE YÖNELİMLİ EŞ ZAMANLI PROGRAMLAMA

Tez çalışmasında, JSCOOP kullanılarak JAVA dilinde DCI ve SCOOP yöntemleri birleştirmiş ve ROCOCO (Role Oriented Concurrent Conxexts) olarak adlandırdığımız melez bir yaklaşım ile role yönelimli eş zamanlı programlama gerçekleştirilmiştir. Adil bir eş zamanlılık sağlayacak karmaşık kodun otomatik olarak oluşturulması ile detayları bilmesi gerekmeyen programcı için iki önnot önemlidir.

@separate: alan veya parametre, başka bir temsilci işlemcide yaşayan bir nesnedir.

@await: yordam, parametreleri kullanılarak yazılan koşul gerçekleştiğinde başlatılır.

```
public class DiningPhilosophers {
    public static void main(String [] args) {
        @separate (dir="Fork[@separate] forks")
        Fork[] forks = new Fork[5];
        for (int i = 0; i < 5; i++) {
            @separate Fork fork = new Fork();
            forks[i] = fork;
        }
        @separate(dir="Philosopher[@separate] philosophers")
        Philosopher[] philosophers = new Philosopher[5];
        for (int i = 0; i < 5; i++) {
            @separate Philosopher phil = new Philosopher(i);
            phil.leftFork = forks[i];
            phil.rightFork = forks[(i+1)%5];
            philosophers[i] = phil;
            startPhilosopher(phil);
        }
    }
    private static void startPhilosopher(@separate Philosopher p) {
        p.live();
    }
}
```

Şekil 3.1: JSCOOP ile Filozofların Akşam Yemeği (Torshizi, 2009)

Mevcut yapıyı günümüz eclipse ortamında taşımak ve basit pürüzleri ortadan kaldırmak dışında JSCOOP kodları üzerinde bir çalışma yapılmamış, ileride detaylandıracağımız üzere ROCOCO ile atomik bağlam kurma ve çalıştırma kısıtlarına uyulduğu için derleme anında nesneye yönelime indirgenebilen role yönelimli bir kodun bildiğimiz haliyle nesneye yönelime indirgenebilmesinin ve en nihayetinde JSCOOP aracılığı ile eş zamanlı çalışan bir koda dönüştürülebilmesinin mümkün olduğunun gösterilmesine odaklanılmıştır.

3.1 Rol Tabanlı Programlama ve Eş Zamanlı Programlamanın Sentezi

DCI sözlüğünde yer alan aşağıdaki tanımlara sadık kalınarak eş zamanlı çalışacak program yazma kuralları ve gerekli derleyici desteği ile SCOOP ve DCI sentezi yapılmış ve DCI çalıştırma modelindeki tek-iplikli çalışma kısıtı ortadan kaldırılmıştır.

- **Bağlam** (@Context), bir etkileşimi gerçekleştirmek üzere birlikte çalışan tüm rolleri ilgili rol veri nesneleri ile eşleştiren bir nesne ağını belirtir. Her biri ayrı bir iç sınıf olan rol sınıflarından, rolleri oynamak üzere atanan veri nesnelerini eşleştiren alanlardan ve etkileşim kodunu içeren metodun kendisinden oluşur.

- **Ön Koşul** (@pre), etkileşime başlamak için (MAESTRO bloğu) karşılanması gereken, rol nesnesi kontratları ve rol metodları cinsinden yazılan bir koşuldur.

    - **Rol** (@Role), bir bağlam örneği içinde etkileşen nesneler ağındaki bir nesnenin takma adıdır. Nasıl davranacağı ilgili rol iç sınıfında kodlanmıştır.

    - **Rol Nesne Kontratı veya Veri Metodu** (@RoleObjectContract veya @DataMethod), yalnızca belirli mesajlara cevap verebilen, yani belirli metodlara sahip olan nesnelerin ilgili rolü oynayabileceğinin güvencesidir.

    - **Veri Metodu Uyumlayıcısı (**@DataMethodAdapter), veri nesnesindeki metodu rol için gereken veri metoduna uyumlayan method uyumlayıcısıdır. Tezin özgün katkılarından biri olan veri metodu uyumlayıcıları sayesinde, rol oyuncularının rol nesne kontrantlarını birebir aynı imzaya sahip bir metod ile gerçekleştirme gerekliliğinin önüne geçilmiştir. Rol oyuncuları, içinde rol oynayabileceği bağlamlar hakkında gerçekten en ufak bir bilgiye sahip değildir. Bir veri nesenesinin bir role uygun olup olmadığının tespit edilmesi işi tamamen bağlama ve bağlam örneğinin oluşturulduğu anda tanımlanan veri metodu uyumlayıcılarına bırakılmıştır. Uyumlayıcılar, bağlam için gereksiz olan uyumsuz parametrelerden birini düşürmek veya birim dönüşüm hesaplamaları gibi basit işlemleri de gerektiğinde yapabilir.

    - **Rol Metodu** (@RoleMethod), bağlam içinde kodlanmış tüm rol oyuncu nesneleri arasında paylaşılan ve yalnızca bağlam içinde yaşayan durumsuz ve polimorfik olmayan metodlardır, rol iç sınıflarının metodlarıdırlar.

- **Rol Eşleşmesi** (@RoleMap), roller ve rol oyuncuları olan veri nesneleri arasında geçerli olan eşlemelerdir. Tez çalışmasında bağlam sınıfının alanları olarak tasarlanmışlardır.

- **Rol Oyuncusu** (@RolePlayer), iletişim kuran nesneler ağındaki bir rolün yerini alan veri nesneleridir.

- **Son Koşul** (@post), etkileşimin geçerliliğini kanıtlamak için rol nesnesi kontratları ve rol yöntemleri açısından yazılan bir koşuldur.

Her ne kadar bağlam ve veri nesneleri gevşek bağlı olsa da bağlam nesnelerinin aksine veri nesnelerinin polimorfizmi de destekleyen nesneye yönelimden taviz vermeden kodlanabilmelerinden ve her ne kadar çalıştırma anındaki sınıfları belli olmasa da veri nesnelerinin baz sınıflarının veya cast edildikleri sınıfların belli olmasından hareketle, bağlam oluşturulduğu anda baz alınacak sınıfa ait alanlar ve metodlar bellidir. Bu sayede, uyumsuz rol oyuncularını derleme anında hata olarak tespit etmenin mümkün olması ile bu durum programcıya belirtilerek çalıştırma anında hata alınması önlenir.

Tez çalışmasında gerçekleştirilen tümleşik geliştirme ortamı eklentisinin böyle bir statik kontrolün yapılmasının sağlanabildiğini göstermesi de özgün katkılardan biridir. Böylece hem eş zamanlılığın, hem kullanım senaryolarının, hem de bağlamların programcının belirttiği gibi çalışacağı tasarım anında kontrol edileben,

- eş zamanlı da olabilen etkileşimler (kullanım senaryosu gerçekleştirimleri)

- hiçbir bağlama arayüz (interface) ile göbekten bağlı olmayan veri nesneleri

- ve her biri kendi yığınında sıralı çalışan temsili işlemciler aracılığı ile hem iç içe hem de eş zamanlı çalışabilen bağlamlar

mümkün kılınmıştır.

3.2 ROCOCO ile Bağlam Sınıfı Tanımlama

ROCOCO'da bağlam, *@Context* önnotu ile işaretlenen bir sınıf olarak tasarlanmıştır.

```
@Context
public class MoneyTransfer { … }
```

Bağlamın en temel görevi, bağlam içindeki rollerin oyuncuları olarak etkileşimde bulunacak veri nesnelerini bir araya getirmektir. Bağlam örneğinin oluşturulduğu anda gerçekleştirilecek bu işlemi daha sonra detaylandıracağımızı ifade etmekle yetinerek, eşleşen rollerin bağlam sınıfının alanları olarak tasarlandığını belirterek devam edelim.

```
@Context
public class MoneyTransfer {
        @separate @RoleMap public Source source;
        @separate @RoleMap public Destination destination;
        @separate @RoleMap public Banknote banknote;
        …
}
```

Programcı, *@separate* önnotu ile bu bağlamda rol alacak veri nesnelerinin farklı temsili işlemcilerde eş zamanlı olarak çalışabileceklerini ROCOCO'ya bildirmiştir. *@RoleMap* önnotu ilgili alanın bir rol eşleşmesini ifade ettiğini belirtir. Örneğin, *source* alanı, bağlam örneği oluşturulduğunda kaynak hesap rolünü oynayacak bir veri nesnesine referans olacaktır. Bağlamlar durumsuzdur (stateless), dolayısıyla bağlama ait *@RoleMap* önnotu ile işaretlenmeyen herhangi bir alan mevcutsa uyarı verilebilir. *@RoleMap* önnotunun hiç kullanılmaması tercih edilirse, bağlamları durumsuz (statetless) kullanma kısıtını da uygulayarak tüm bağlam alanlarının bir rol eşleşmesini ifade ettiği varsayımında bulunulabilir. Daha az anahtar kelimeye ihtiyaç duyulması açısından ROCOCO'da *@RoleMap* önnotunun kullanılmaması tercih edilmiştir. ROCOCO, bağlam örneği oluşturulduğu anda karşılanmayan veya fazla olan bir rol oyuncusu tanımı ile karşılaşırsa, bunu bir derleme hatası olarak programcıya bildirir. Bağlam atomik tek bir çağrı ile oluşturulur, bir kez kuruldu mu rol oyucuları değişmez.

```
@Context
public class MoneyTransfer {
        @separate @RoleMap public Source source;
        …
        @Role
        private class Source {
                @DataMethod
                public void debit(Banknote banknote) {}
                @RoleMethod
                public double commission(Banknote banknote) {
                        return banknote.getAmount() * 0.08;
                }
        }
        …
}
```

ROCOCO'da roller iç sınıflar olarak tasarlanmışlardır, *@Role* önnotu ile işaretlenirler. *@Role* önnotu ile işaretlenmeyen iç sınıflar da yardımcı (helper) sınıflar olarak bağlam sınıfı içinde yer alabilirler. Böylelikle, veri metodu uyumlayıcılarının kullanacağı pek çok basit dönüştürme kodu (örneğin, birim dönüştürücüsü) bağlam içinde kodlanabilir. *@DataMethod* önnotu ile belirtilen metod veri nesnesinde aranacaktır, gövdesi boştur. İlgili veri metodu uyumlayıcısı tanımlanmamış ise, birebir imza ile eşleşme aranır. *@RoleMethod* önnotu ile belirtilen metod bu bağlamda rol oyuncusu adına çalıştırılır. Yerleştirmesiz (injectionless) DCI kullanılır, böylece veri nesnesi rol metodu ile ilgili hiçbir doğrudan veya dolaylı bilgiye sahip olmaz. DCI, bağlamlar ile veri nesneleri arasındaki ilişkiyi tek yönlü tasarlandığından, yerleştirmesizlik istenen bir şeydir. Aynı zamanda ROCOCO ile role yönelimin nesneye yönelime indirgenmesi için gereklidir. Veri nesnelerinin rol metodları ile ilgili hiçbir bilgiye sahip olmaları gerekmediğinden, *@RoleMethod* anahtar kelimesinin de kullanılmasına gerek yoktur, *@DataMethod* önnotu ile işaretlenmeyen tüm methodlar ayrı birer rol metodu olarak kabul edilmiştir. Programcının olan biteni algılayabilmesi için *@RoleMethod* önnotu kullanımı önerilir.

```
@Context
public class MoneyTransfer {
        …
        @await(condition="!s.isBusy()&&!d.isBusy()&&!b.isBusy()
        &&s.acquire()&&d.acquire()&&b.acquire()")
        public void maestro (
                @separate Source s,
                @separate Destination d,
                @separate Banknote b {
                        b.setBanknote += s.commission(b);
                        s.debit(b);
                        b.setBanknote -= s.commission(b);
                        d.credit(b);
                        s.release();d.release();b.release();
                }
        }
        …
}
```

Yukarıda etkileşim kodu gösterilmiştir. Method *@await* önnotu ile işaretlendiğinde ROCOCO belirtilen koşul sağlanana kadar metodu bekletir. Dikkat edilirse, mantıksal bir kilit olan acquire metodları da bekleme koşuluna bağlanmıştır. Bunun nedeni, yarış durumu oluşmaması için mantıksal kilitlemenin de adil eş zamanlılıktan sorumlu olan global zamanlayıcı tarafından çağrılan "bekleme koşullarını kontrol et" metodunda yapılması gerekliliğidir. Kilitleme metod gövdesinde yapılırsa yarış durumu oluşur.

Role yönelimli eş zamanlılığı desteklemek üzere ROCOCO'da getirilen kısıt gereği, bağlam örneğinin yapılandırıldığı anda tüm rollerin bağlanması ve etkileşimin başlatılması gereklidir. Aksi taktirde, tümleşik geliştirme ortamının (IDE), bağlamın doğru kurulup kurulmadığını inceleyebileceği bir referans noktasından söz edemeyiz. Her bağlam için bir ve yalnız bir adet olabilecek etkileşim kodu, DCI sözlüğüne ithafen ROCOCO'da *maestro()* olarak isimlendirilmiştir; *maestro()* methodunun bağlamda zaten mevcut rol eşleşmelerini parametre olarak almasının nedeni, JSCOOP'un hangi rolün hangi temsili işlemcilde çalıştığının izini formal parametreler ile takip etmesidir. Bekleme koşulu da yine bu formal parametreler üzerinden erişilen metodlar ile yazılır.

3.3 ROCOCO ile Bağlam Sınıfı Örneğinin Yapılandırıldığı Kaynak Kod Satırı

ROCOCO, yukarıdaki gibi tanımlanan bağlam sınıfı örneklerinin atomik olarak yapılandırıldığı kaynak kodu satırında belirtilen rol eşleştirrmelerini referans alarak, orijinal bağlam sınıfı kodunu kopyalar ve kopyalanan bu bağlam kodu üzerinde rolleri oynayacak veri nesnelerinin metodlarını çağırmak üzere gerekli değiştirmeleri yapar. Bağlam örneğinin oluşturulduğu satırdaki bağlam sınıfı ismini de türetilmiş bağlam sınıfının ismi ile değiştirerek, o satıra özel bağlam örneğinin oluşturulmasını sağlar. Tüm bu işlemler derleme esnasında programcıya destek olmak amacıyla jazılmış Java Development Toolkit (JDT) ve dolayısıyla JAVA Soyut Sözdizim Ağacı (JAVA AST) kullanılarak yapılır. Çözümün detayları ve gerekli sözdizimi aşağıdaki gibidir.

Rolü oynayacak veri nesnelerinin aşağıdaki gibi oluşturulduğunu düşünelim:

```
@separate Hesap kaynak = new Hesap(IBAN_kaynak);
@separate Hesap hedef = new Hesap(IBAN_hedef);
@separate Para para = new Para(50);
```

Bu veri nesnelerinin rol oyuncusu olacağı bağlamı aşağıdaki şekilde yapılandıralım:

```
@RolePlayers(
        mappings = {
                "Source=kaynak",
                "Destination=hedef",
                "Banknote=para"
                },
        adapters = {
                "Destination.credit = _Destination.deposit(b);"
                }
        )
@separate MoneyTransfer moneyTransfer = new MoneyTransfer();
```

Yukarıdaki kod ile karşılaşan bir programcı, konu hakkında hiçbir bilgisi olmasa dahi, bu noktada role yönelimin kullanılmakta olduğunun farkına varacaktır. Ne yapılmak istendiği *@RolePlayers* önnotuna bakarak açıkça anlaşılmaktadır. Ayrıca, *@separate* önnotu ile bağlamın kendisinin de farklı bir temsilci işlemcide yaşayabileceği belirtilir.

Sağduyu ile anlaşılacağı üzere, *mappings* rol eşleşmelerini ve *adapters* veri metodu adaptörlerini ifade eder. Yani, yeni oluşturulan *moneyTransfer* bağlamında *kaynak* veri nesnesi *Source* rolünü, *hedef* veri nesnesi *Hedef* rolünü, *para* nesnesi ise *Banknote* rolünü oynayacaklardır. *Destination* rolünde yer alan *credit* veri metodu için ise bir veri metodu adaptörü kullanılmış ve bu rolü oynauacak olan *Hesap* sınıfında aynı işi yapan *deposit* metodu çağrılmıştır. *@RolePlayer* önnotu, önnot işleme mekanızması kullanılarak işlenir ve herhangi bir hata derleme esneasında programcıya gösterilirir.

Yukarıdaki kodda, farklı ipliklerde çalışabileceği öngörülen hedef ve kaynak hesap sınıfları ile kurulan eş zamanlı bir bağlam ile para transferi işlemi betimlenmiştir. Para transferi işleminin kendisi de başka bir iplikte çalıştırılarak diğer komuta geçilecektir.

Programcının yukarıdaki gibi sade olarak okuyabildiği kod ve önnotlara dayanarak role yönelimli eş zamanlılığı sağlayan ROCOCO, ileride detaylandıracağımız üzere bu işi metaprogramlama olarak nitelendirecebileğimiz JAVA önnot işleme ile başarır.

3.4 ROCOCO ile Tezin Özgün Katkıları

Tezin ortaya koyduğu sentez yapının esas katkısı, DCI çalıştırma modelinde çok-ipliklilliğin de desteklenmiş olmasıdır. Böylece, DCI yaklaşımının önündeki engel ortadan kaldırılmıştır. Eiffel dilinden C diline dönüşümün yıllarca süren araştırmalar ile ilerleyen performansına benzer şekilde, rol tabanlı eş zamanlama ortamını mümkün kılan kod dönüşümü de üzerinde araştırmalar yapılmaya devam ettikçe çok daha iyi performansla çalışacaktır. Böylece, programcı ile programıcının ne yapmaya çalıştığını formal olarak kavrayan tümleşik geliştirme ortamının anında etkileşimi ile programcının iyi yapamayacağı veya rutin olduğu için hata çıkarabilecek olduğu işleri tümleşik geliştirme ortamına yaptırarak, programcının üst seviye bir dilde yazılım geliştirirken aynı zamanda da en iyi performansa veya en iyi performansa çok yakın bir optimum performansa sahip bir çözüm oluşturması mümkün kılınacaktır.

DCI semantiğinin JSCOOP ve DbC ile birleştirilerek en sık karşılaşılan eş zamanlılık problemlerinine role yönelimli olarak da çözüm sağlanmasının yanısıra, hem analistler hem de programcılar tarafından iyi bilinen kullanım senaryosu ve kullanım senaryosu gerçekleştirme tekniğinin UML diyagramlarındaki yansımasının birebir aynısının diyagramdan çok daha verimli bir geliştirme ortamı olduğu su götürmez bir gerçek olan tip güvenli bir koda metin bazlı olarak yansıtılabilmesi de tezin diğer katkısıdır.

Bu, Functional UML (fUML) ve Action Semantics ile UML dünyasına getirilmek istenen, eylemlerin oldukça net olarak belirtilebildiği diyagram bazlı çalıştırılabilir davranış modelleme çözümlerinin metin bazlı bir çözüm olarak koda ve tümleştirilmiş geliştirme ortamına taşınması anlamına gelir ki, kendine ait bir yer bulan davranışsal kodu okumanın çeşitli dillerde yazılan test betiklerini okumaya, çalıştırmaya ve hatta düzenlemeye alışkın olan analistler için de çok daha kolay bir etkinlik haline gelmesi ile yazılım kalitesinde gözle görülür artış olacaktır.

ROCOCO (Role Oriented Concurrent Contexts) geliştirme ortamı eklentisi, rol ile rol oyunucularının metod uyumlamarını da (data method adaptations) yapabilir ve olası uyumsuzlukların tümleşik geliştirme ortamı üzerinde programlama etkinliği esnasında programcıya anında gösterilebilir. Role yönelimli kod yazımı anında tümleşik geliştirme ortamı (IDE) üzerinde yapılan kontroller aracılığı ile programcıya sağlam kod yazmaya yönelik geribeslemeler verebilmenin yanısıra, cepheye yönelim veya arayüz ile elde edilemeyecek bir dolaylı bağlanma serbestliği de sağlanmış olur.

Şöyle ki, nesne arayüzün gerektirdiği bir metodu bire bir aynı imza ile gerçekleştirmek zorunda değildir. Aksine, bağlamın oluşturulduğu anda analist veya programcı o bağlamdaki hangi rol metodunun nesnedeki hangi veri metodu ile nasıl uyumlanabileceğini betimleyebilir. Bir adım ilerisinde, uyumlama için gereken birim dönüştürücü, vb. gibi sık kullanılan yardımcı (helper) fonksiyonlar yine veri nesnesinin değil de bağlamın kodunda gerçeklenmesi tercih edilerek, nesnelerin bağlamlar hiç yokmuş gibi geliştirilmeye devam edilmesi daha da aşikar kılınabilir.

Çözümlenen asıl konu DCI çalıştırma modelindeki tek ipliklik çalışma kısıtını ortadan kaldırmak olsa da, eş zamanlı çalışmayı bir kenara bırakacak olsak dahi, önerilen önnotlar ile birlikte rollerin iç sınıflar olarak kodlanması ve bağlamın otomatik oluşturulması sayesinde oldukça okunabilir bir role yönelimli koda JAVA diline hiçbir anahtar kelime eklemeden ulaşılabileceğinin gösterilmesi de tezin ayrı bir katkısıdır.

4. ROCOCO GERÇEKLEME DETAYLARI

ROCOCO prototip geliştirme ortamının gerçekleştirilmesinde açık kaynak kodlu çözümlerin gelişmesine sunduğu katkının yanısıra zengin bir araç kütüphanesi ile birlikte canlı bir topluluğa sahip olan eclipse baz alınmıştır. On yılı aşkın bir süredir, eclipse geliştirme ortamlarının kendisinin de geliştirilmesine katkı sağlamak üzere "eclipse classic" ile başlayıp zaman içinde "eclipse standard" ve en nihayetinde "eclipse IDE for eclipse committers" adını alan ve tümleşik geliştirme ortamı (IDE) eklentisi geliştirmeye mümkün kılan Java Development Kit (JDT) içeren bir eclipse IDE sürümü dağıtılmaktadır. JDT kullanılarak geliştirilmiş olan JSCOOP, programcı tabiri ile çapakları da temizlenerek günümüz eclipse teknolojisine taşınıp "eclipse IDE 2019-03 for eclipse commiters" ortamında tekrar çalışır hale getirilmiştir. SCOOP yaklaşımının temellerini eclipse ortamında JAVA dilinde gerçekleştirmiş olan JSCOOP, bir deneme ortamı sunması ve üzerine geliştirme yapılmasına olanak sağlaması dolayısıyla uygun bulunmuştur. Zaman kısıtı nedeniyle, belirtilen önemli JSCOOP eksikliklerinin de bertaraf edilmesi ile tam bir gerçekleme hedeflemek yerine tezin asıl özgün katkısı olan DCI ile SCOOP sentezinin mümkün olduğunun gösterilmesine ağırlık verilmiştir.

ROCOCO ile, programcının belirttiği spesifikasyonlara bağlılığı tasarım aşamasında tümleşik geliştirme ortamı tarafından sağlanan ve koşulların çalıştırma zamanında ihlal edilebilmesi durumu ile karşılaşabilmesi ihtimali durumunda programcının uyarıldığı etkileşimli bir eş zamanlı ve rol tabanlı tümleşik geliştirme ortamı eklentisi protoripi geliştirilerek, okunabilirlikte ve yazılım kalitesinde önemli bir katkı sağlayacak olan role yönelim ve eş zamanlılık kavramlarının yazılım geliştirme ortamlarımıza birlikte entegre edilmesinin mümkün olduğu gösterilmiştir.

Çözüm mimarisi, aşağıdaki kavramların sentezi ile oluşturulmuştur:

- DCI (Veri Bağlam Etkileşim)
- SCOOP (Basit Eş Zamanlı Nesneye Yönelimli Programlama)
- Metaprogramlama (önnot -annotation- işleme ile kod üreten kod)

4.1 Metaprogramlama

DCI için gerekli olan roller ile rol oyunucularının bağlanması ve ilgili veri metodlarının dolaylı olarak çağırabilmeleri ile birlikte SCOOP için gerekli olan tüm eş zamanlılık kod dönüşümünü sağlamak üzere metaprogramlamaya başvurulmuştur.

Diğer rol tabanlı yaklaşımlardan farklı olarak, bağlam ile rol oyuncusu veri nesnelerinin tek boyutlu iletişimi ile bağlamın veri nesnesini tam olarak bilmesi ve veri nesnesinin bağlam hakkında hiçbir şey bilmemesi kısıtının önemine vurgu yaparak iç içe bağlamların yanısıra eş zamanlı bağlamları da mümkün kıldığı tez çalışmamızda gösterilen DCI kısıtlarının önemli bu teknik soruna da çözüm olduğu gözlenmiştir. Bağlam (etkileşim) kodundan veri nesnesi koduna doğru tek yönlü iletişim mevcuttur.

Bağlam ile rol oyuncusu nesnelerinin tek yönlü iletişimi kısıtını içermesi dolayısı ile DCI için gerekli olan rol bağlamalarının derleme anında gerçekleştirilebilmesi sayesinde çalıştırma anında yorumlanan yorumlamalı (interpretive) bir çözüm yerine derleme zamanında kod oluşturulması tercih edilmiş ve bu sayede yansıtmaya (reflection) olan ihtiyaç da bertaraf edilerek sadece veri metod uyumlayıcılarının çağrılması esnasında zorunlu bir dolaylılık nedeniyle cereyan eden çok cüzi bir kayıp ile, ki veri metodu uyumlayıcısı her halikarda çağrılacağı için bu bir kayıp sayılmaz, en iyi DCI dönüşüm performansı elde edilmiştir. Bağlam örneğinin oluşturulması ve veri metodu uyumlaması gibi tüm meseleler derleme aşamasında koddan koda dönüşüm ile halledilmişlerdir, çalıştırma anında hiçbir performans kaybı yaşanmaz. Eş zamanlılığı sağlamak için oluşturulan kodun SCOOP olmadan adil bir eş zamanlı bir programlama yapmaya oranla çok daha sağlam ve optimum olacağı ilgili çalışmada gösterilmiştir (Brooke ve diğerleri, 2007).

Algoritma karmaşıklığı açısından DCI ile SCOOP sentezi olan ROCOCO çözümümüzün tasarım gereği her zaman optimum bir performans sağlayacağını düşünmekteyiz. Elbette, ileride yapılacak çalışmalar ile gerçek performans ölçümleri yapılabilir ve özellikle eş zamanlılık dönüşüm performansı iyileştirilebilir.

Tezin asıl amacı, programcının davranışsal kodu içine alan bir bilişsel bütünlük içinde tümleşik geliştirme ortamının da desteğini alarak role yönelimli eş zamanlı programlama yapabilmesinin mümkün olduğunu göstermektir. Bir prototip olsa da, ROCOCO ile bunun yapılabileceği gösterilmiştir.

Örnek *maestro*() metodunun eş zamanlı çalışabilmesi için gerekli olan JSCOOP kod dönüşümünü, JSCOOP soyut sözdizim ağacındaki (AST) düğümleri ziyaret eden JDT gözlemleyicileri (observers) ile gerçekleştirir.

```
public void maestro(JSCOOP_Transfer_asCalledFrom_ParaTransferi_29 interaction,
JSCOOP_Hesap kaynak, JSCOOP_Hesap hedef, JSCOOP_Para para ) {
        jscoop_new_locks = new HashSet<JSCOOP_Processor>();
        jscoop_new_locks.addAll(this.getProcessor().getCurrentLocks());
        jscoop_new_locks.add(this.getProcessor());
        ((JSCOOP_Runnable)
interaction).getProcessor().pushLocks(jscoop_new_locks);
        JSCOOP_Call jscoop_single_call;
        JSCOOP_Call jscoop_wait_calls[];
        jscoop_wait_calls = new JSCOOP_Call[1];
        jscoop_requesting_locks = new LinkedList<JSCOOP_Processor>();
        jscoop_requesting_locks.add(((JSCOOP_Runnable) kaynak).getProcessor());
        jscoop_requesting_locks.add(((JSCOOP_Runnable) hedef).getProcessor());
        jscoop_requesting_locks.add(((JSCOOP_Runnable) para).getProcessor());
        jscoop_lock_request = new JSCOOP_LockRequest(((JSCOOP_Runnable)
interaction), jscoop_requesting_locks,
((JSCOOP_Runnable)interaction).getProcessor().getLockSemaphore());
        jscoop_args = new Object[3];
        jscoop_arg_types = new Class[3];
        jscoop_args[0] = kaynak;
        jscoop_args[1] = hedef;
        jscoop_args[2] = para;
        jscoop_wait_calls[0] = new JSCOOP_Call("maestro", jscoop_arg_types,
jscoop_args, jscoop_lock_request, (JSCOOP_Runnable) interaction);
        jscoop_wait_calls[0].setLockPassingFinished(true);
        ((JSCOOP_Runnable)interaction).getProcessor().addRemoteCall(jscoop_wait_
calls[0]);
        }
```

Şekil 4.1: Dönüştürülen JSCOOP Kodu

Basit eş zamanlı nesneye yönelim çözümü olan JSCOOP dönüşümüne girdi olabilmesi amacı ile role yönelimli ve eş zamanlı olarak betimlenen bir kodun, nesneye yönelime atomik olarak indirgenmesinden ROCOCO sorumludur. Dönüşüm şırasıyla şöyledir:

Role Yönelimli Eş Zamanlı Betimlenen Kod → ROCOCO →JSCOOP → JAVA Kodu

Kod dönüşümünün yapılmamasını gerektirecek yapısal bir hata olduğu konusunda programcıyı uyarmak için de tümleşik geliştirme ortamı içinde çalışan JDT kullanılır. Oluşabilecek tüm yapısal hatalar derleme aşamasında yakalanabildiği için, programcı çalıştırma anını beklemeden programının doğru çalışacağı hakkında bilgi sahibi olur. Bunun tek istisnası, programcının kendi yazacağı bekleme koşulları aracılığı ile yarış durumu oluşturabilecek olmasıdır. ROCOCO, diğer hataları formal olarak yakalar.

4.2 ROCOCO Bağlam Dönüşümü

Öncelikle, referans olarak kullanılacak kaynak kodu satırından bahsedelim. Bağlam yapılandırıcısını çağırmak üzere daha önce verdiğimiz kod örneğini yineleyelim:

```
@separate Hesap kaynak = new Hesap(IBAN_kaynak);
@separate Hesap hedef = new Hesap(IBAN_hedef);
@separate Para para = new Para(50);
…
@RolePlayers(
        mappings = {
                "Source=kaynak",
                "Destination=hedef",
                "Banknote=para"
                },
        adapters = {
                "Destination.credit = _Destination.deposit(b);"
                }
        )
@separate MoneyTransfer moneyTransfer = new MoneyTransfer();
```

Yeni bağlamın client.java kaynak dosyasının 19. satırında oluşturulduğunu düşünelim. Bağlam örnekleri oluşturmak üzere bağlam yapılandırıcısının çağrıldığı her satır için role yönelimi nesneye yönelime indirgemek üzere ROCOCO iki temel iş yapar.

1. İstemcinin dönüştürülmüş halini saklamak için *ROCOCO_Client.java* isminde yeni bir dosya yaratır. JAVA soyut sözdizim ağacı aracılığı ile, sınıf örneği (ClassInstanceCreation) oluşturulan tüm düğümler ziyaret edilererek, bir örneği yapılandırılmak üzere olan yeni sınıfın *@Context* önnotu ile işaretlenip işaretlenmediğine bakılır ve bir bağlam oluşturulmakta olunup olunmadığına karar verilir. İstemcinin dönüştürülmesi için, istemci kodunun içinden en az bir tane bağlam yapılandırıcısı çağrılmış olması yeterlidir.

2. Bağlamı özelleştirmek üzere bağlamın dönüştürülmüş halini saklamak için *MoneyTransfer_asCalledFrom_client_19.java* isminde yeni bir dosya yaratır. Bağlamın oluşturulduğu her farklı kaynak kod satırında, roller ile eşlenecek veri nesnelerinin tipleri farklı olabileceği için her farklı kaynak kod satırı için farklı bir isim verilen bağlam sınıfları oluşturulur.

```
@Context
public class MoneyTransfer {
        @separate @RoleMap public Source source;
        @separate @RoleMap public Destination destination;
        @separate @RoleMap public Banknote banknote;
        …
        }
```

ROCOCO_MoneyTransfer.java dosyasında rol sınıfları yerine veri sınıfları kullanılır. İlk dönüşüm, rol eşleşmelerini ifade eden bağlam sınıfına ait alanlarda *Source* ve *Destination* yerine *Hesap*, *Banknote* yerine *Para* kullanımıdır:

```
public class ROCOCO_MoneyTransfer {
        @separate public Hesap source;
        @separate public Hesap destination;
        @separate public Para banknote;
        …
        }
```

Bu örnekte, *Hesap* veri tipi hem *Source* hem de *Destination* rollerinde kullanılmıştır.

İkinci dönüşüm, *maestro()* ile ifade edilen etkileşim kodu parametre tiplerinde benzer şekilde *Source* ve *Destination* yerine *Hesap*, *Banknote* yerine *Para* kullanımıdır. Bu dönüşüm sayesinde, metod imzası veri metodları ile birebir uyan veri nesnesi metodlarını çağırmak üzere hiçbir ek maliyet gerekmeksizin etkileşim kodu aynen kullanılabilir. Veri metodu uyumlayıcısı var ise ne yapılacağı ileride anlatılacaktır.

```
public class ROCOCO_MoneyTransfer {
        …
        @await(condition="!s.isBusy()&&!d.isBusy()&&!b.isBusy()
        &&s.acquire()&&d.acquire()&&b.acquire()")
        public void maestro (
                @separate Hesap s,
                @separate Hesap d,
                @separate Para b {
                        b.setBanknote += s.commission(b);
                        s.debit(b);
                        b.setBanknote -= s.commission(b);
                        d.credit(b);
                        s.release();d.release();b.release();
                }
        }
        …
}
```

Üçüncü dönüşüm, rol metodlarının ilgili rol ile eşleşen veri nesnesini parametre alacak şekilde rol iç sınıfının bir metodu olmak yerine bağlam sınıfı içinde konumlanmasıdır:

```
public class ROCOCO_MoneyTransfer {
        …
        public double commission (@separate Hesap _Source,
        Para banknote) {
                return banknote.getAmount() * 0.08;
        }
        …
}
```

Bu dönüşümü yapabilmek için etkileşim metodu ve rol sınıfı içindeki tüm metodlar içinden yapılan tüm *commission* çağrıları JAVA soyut sözdizim ağacı ile ziyaret edilerek, ilk parametre olarak rol oyunucusunu alacak şekilde değiştirilir. Örneğin,

```
b.setBanknote += s.commission(b);  çağrısı
```

```
b.setBanknote += commission(s, b);  olarak değiştirilir.
```

Dördüncü dönüşüm, benzer şekilde veri metodu uyumlayıcısının oluşturulması,

```
public class ROCOCO_MoneyTransfer {
        …
        public void credit (@separate Hesap _Destination,
        Para b) {
                _Destination.deposit(b);
        }
        …
}
```

ve çağrıldığı her noktada ilk parametreyi alacak şekilde değiştirilmesidir. Örneğin,

```
d.debit(b);  çağrısı
```

```
debit(d, b);  olarak değiştirilir.
```

Böylece, *MoneyTransfer* bağlam sınıfı baz alınarak *ROCOCO_MoneyTransfer* özelleştirilmiş bağlam sınıfını oluşturan ROCOCO dönüşümü tamamlanmış olur.

4.3 ROCOCO Bağlam Yapılandırıcısı Dönüşümü

Bağlam yapılandırıcısı için yapılacak dönüşüm, bağlamın oluşturulduğu satıdaki *MoneyTransfer* tip ismini *MoneyTransfer_asCalledFrom_client_19* olarak değiştirmek, rol oyuncularını parametre olarak alacak olan *maestro()* sarmal metodunu oluşturmak ve çağırmaktan ibarettir. Dönüşüm öncesindeki aşağıdaki kod,

```
@RolePlayers(
        mappings = {
                "Source=kaynak",
                "Destination=hedef",
                "Banknote=para"
                },
        adapters = {
                "Destination.credit = _Destination.deposit(b);"
                }
        )
@separate MoneyTransfer moneyTransfer = new MoneyTransfer();
```

dönüşüm sonrasında aşağıdaki şekli alır:

```
@separate MoneyTransfer_asCalledFrom_client_19 moneyTransfer =
new MoneyTransfer_asCalledFrom_client_19();
maestro(moneyTransfer, kaynak, hedef, para);

…

void maestro(@separate MoneyTransfer_asCalledFrom_client_19
moneyTransfer, @separate Hesap kaynak, @separate Hesap hedef,
@separate Para para) { moneyTransfer.maestro(kaynak, hedef, para); }
```

Böylece, dönüştürülmemiş orijinal kodda programcının tek bir atomik satırda belirttiği etkileşim kodunu çağıracak olan dönüştürülmüş kod oluşturulmuş olur. Bu kod, hem derleyicinin kontrolünden hem de JSCOOP eş zamanlılık yapısal kontrollerinden geçirildiği için oluşan tüm yapısal derleme hataları bağlamın yapılandırıldığı, yani programcının bildiği tek kaynak satırında derleme uyarıları olarak gösterilmektedir.

Bu sayede, programcı tümleşik geliştirme ortamı ile etkileşiminde herhangi bir farklılık veya gecikme yaşamayarak bilişsel olarak ne olup bittiğinden kopmayacaktır.

4.4 Gerçekleme ile Gösterilen Özgün Katkılar

Çalışmanın özgün katkılarını şöyle özetleyebiliriz:

- JAVA diline hiçbir eklenti yapmadan, bağlam içindeki rollerin, roller içindeki rol kontratlarının, ve -eğer gerekir ve istenirse- veri içindeki metodların verinin oynayacağı role ait rol kontratları ile uyumlayıcılarının sadece JAVA önnotları (annotations) aracılığı ile tanımlanabileceğinin ve çalıştırılabileceğinin gösterilmesi ile birlikte,
- JAVA soyut sözdizim ağacında, bağlam örneğini oluşturan yapılandırıcının (constructor) çağrıldığı kaynak kod satırı ziyaret edilerek, bir nesnenin bir rolü oynayıp oynayamayacağının ilgili önnotlar kullanılarak tespit edilebilmesi,
- Bir uyumsuzluk durumunda eclipse IDE üzerinde ilgili satırda hatanın programcıya gösterilebilmesi,
- Bağlam örneğini oluşturan yapılandırıcının (constructor) çağrıldığı kaynak kod satırına özel olarak ilgili bağlam sınıfından türetilen bağlam sınıfının, rol kontratı ile aynı imzalı olması veya rol kontratı ile o andaki uyumlayıcı önnotlarına dayanarak uyumlanan veri metodunu çağırmak üzere rol kontrat metodunun içeriğinin derleme zamanında değiştirebilmesi aracılığı ile, eş zamanlı rol eklentisinin yakalayacağı hatalar dışında ek bir derleyici hatası üretilmeden hem rol tabanlı hem de eş zamanlı çalışma kontrollerini yapabilen kodun çalıştırma anında değil derleme anında oluşturulabilmesi dolayısıyla programcının bilişsel bütünlük içinde programlama ortamını kullanmaya devam etmesinin sağlanması,
- JAVA tip sistemine herhangi bir ek yapılmaması dolayısı ile, -yine önnotlar ile çözümlenen- veri metodu uyumlayıcılarının uyumlanan veri nesnesi metodlarını çağırması istisnası dışında zorunlu performans tavizi vermeyerek nesneye yönelimli sistemler için geliştirilen ve bekleme koşullarını da dikkate alan JSCOOP (SCOOP) ilkellerinin dışına taşmadan eş zamanlı rol desteğini de sunacak şekilde kod dönüşümünün mümkün olduğunun ve bu sayede DCI çalıştırma modelindeki tek ipliklilik şartının ortadan kaldırılabildiğinin ilk kez gösterilmesidir.

5. TARTIŞMA VE SONUÇ

Umarız ki, özellikle nesneye yönelimdeki programlamadaki temel aksakılığı yeterince iyi ifade ederek konunun önemine dikkat çekebilmişizdir. Tartışma bölümünde nesneye yönelimli tasarım ve programlamaya dair önemli fikirlerin hemen hemen hepsinde eksikliğinin tezahür ettiğini tespit ettiğimiz bu konuyu irdeleyeceğiz. Böylece, her ikisi de gerekli olduğu halde programcının kendisinin el ile oluşturması gereken karmaşık kodlar nedeniyle anayola giremeyen iki yaka olan rol tabanlı programlamayı ve eş zamanla programlamayı birleştirmenin önemini vurgulayacağız.

Önce rol tabanlı programlamadan başlayalım. Kodda kendilerine ait bir yeri anayol programcılıkta hiç bulamamalarına rağmen bağlam ve rol kavramları en azından nesneye yönelimli analiz ve tasarımda UML kullanımıyla yerini bulmuştur. Tartıştığımız üzere, UML diyagramlarını tanıtmak için yapılan etkinlik, iş birliği gibi tanımlamaların rol tabanlı programlamada kullandığımız temel yapıtaşları ile örtüşmesi oldukça çarpıcıdır. Kullanıcı senaryosu güdümlü programlamanın temel taşları olan senaryo çerçeveleri, aktörler, entity, boundary ve controller nesneleri, iş birliği ve etkileşim gibi unsurların neredeyse tamamı, role yönelimli programlamadaki bağlamlara, rollere, bağlam içideki etkileşim kodlarına tekabül ederler. Sağduyumuza bu kadar yakın olan kullanıcı senaryosu güdümlü programlamanın anayola girememiş olmasının temel sebebi, kodlar ile modelleri eşlenik tutacak güncellemelerin otomatik olamaması ve dolayısıyla diyagramların çalışan kodu yansıtamaması idi. Rol tabanlı programlama ile diyagramlardaki bilgiyi sadece metine dönüştürmekle kalmayıp aynı zamanda tip-güvenli bir programlama ortamına da dahil edebiliyoruz.

Bağlamdan veri nesnelerine tek yönlü veri akışının vurgunlandığı DCI ile UML yaklaşımlarının neredeyse tamamen örtüştüklerini, her ikisinin tanımlarına bakarak rahatlıkla söyleyebiliriz. Aralarındaki tek fark, DCI bir programlama paradigması olduğu için kullanım senaryosu gerçekleme adımlarının serbest değil, belirtilen sırada olmasıdır, ki eş zamanlı çalışmayı da işin içine dahil eden tez çalışmamızla birlikte DCI yaklaşımını esneterek UML yaklaşımına biraz daha yaklaştırdımızı düşünüyoruz.

Diğer yandan, çalıştırılabilir UML spesifikasyonları özellikle eylem semantiği (action sematics) ekleriyle eş zamanlı çalışmayı da dahil eden güçlü kavramlar olsa da, kimbilir sadece diyagram olarak yaşadıkları ve kod ile yeterince tümleştirilemedikleri

için anayol programlama akımına dahil olamamışlardır. Diyagram büyük resmi görmeyi sağlar, metin ise anlamayı ve detaylara odaklanmayı, yani hata yapmamayı.

5.1 Veri Nesnelerinden Kopamayan Davranış Kodu

SCOOP ve DbC gibi iki önemli kavramı etkileşimli yazılım mühendisliği (ISE) adını verdikleri ortamlarında yıllardır başarı ile sunan Eiffel dili bile, davranışların kullanım senaryoları olarak ifade edilmesinin nesneye yönelimli tasarımdaki yaratıcılığı öldüreceğini savunarak, bu konuyu çözüm ortamlarının dışında tutmuş ve saf nesneye yönelim diyebileceğimiz bir kulvara yönelmiştir. Naked Objects ile bir MVC ideali olan doğrudan nesne etkileşimini ilk kez gerçekleştiren Richard Pawson da tez çalışmasında kullanım senaryolarının nesneye yönelimin gerçek doğasının ortaya çıkmasındaki en büyük engel olduğunu yazmaktan çekinmez. Demektir ki, anayol olmayan en yenilikçi ve cesur çözümler bile önyargıdan nasiplerini almışlardır. Buradaki temel sorun, sistemin ne olduğu ile sistemin ne yaptığının birbirleri ile uzlaşı içinde gelişmesi gereken iki ayrı evrim hızına sahip olan iki ayrı kavram olduğunun ayırdına bir türlü varılamaması, sistemin ne yaptığına dair kodun sistemin ne olduğunu betimleyen kodun bir alt parçası olarak kod içinde paramparça biçimde yer almasıdır. Madalyonun bu iki yüzünü birbirine karıştırmadan ortaya koyan ilk yaklaşım DCI'dır. Davranış kodunda mirası yasakladığı gibi, veri kodunda mirası sonuna kadar destekler. Bağlam kodunda kendine ait bir mahal edinen davranışsal kodun okunabilirliği kadar, davranışları arayüz metodları ile desteklemek yükünden kurtulan veri nesnelerinin özgürlüğüne de önem verilir. Veri nesneleri bu özgürlükten mahrum kalsa ne olur?

Kütüphane kaynak kodlarındaki bakım ve güncelleme maliyetlerinin orantısız bir şekilde artması nedeniyle çığ gibi büyüyen arayüz gerçekleştirme kısıtlarını en aza indirmek amacı ile, JAVA dünyasında bir icat daha yapıldı: *default interface methods*.

Varsayılan arayüz metodu, eğer arayüzü destekleyecek olan bir nesnede ilgili imzaya sahip bir metod yoksa onun yerine çalıştırılacak olan metoddur. Bir zorunluluk neticesinde doğmuş olsa da, varsayılan arayüz metodunu role yönelimli programlamaya yönelik küçük ama önemli bir anayol dil eklentisi olarak görebiliriz. Ancak, yarattığı belirsizlik ile ne arayüzü destekleyecek olan nesne kodu yazarını ne de diyelim ki DCI amaçlı kullanılacak ise bağlam kodu yazarını rahat uyutmayacak, tüylerini diken diken edecek tehlikeli bir dil eklentisidir varsayılan arayüz metodu.

5.2 Sonuç

Programcının bilişsel bütünlüğünü bozmadan eş zamanlı role yönelimi tümleşik bir geliştirme ortamı olarak ROCOCO gerçeklenmesi aracılığı ile, derin olduğu tespit edilen temel programlama probleminin ve nesnelerin eş zamanlı çalışabilmelerinin, programcının hata yapmasına mahal vermeyen ve optimum bir çalıştırma performansı sağlayan aynı çatı altında nasıl çözümlenebileceği gösterilmiştir. Umarız ki, sadece önnotlara (annotations) dayalı olan ve bir prototip olan bu çalışma endüstride bir karşılık bulur ve çok daha güvenli tümleşik geliştirme ortamlarına yakında kavuşuruz.

KAYNAKLAR

G. Bjørnvig, J. O. Coplien, N. Harrison (2007). "A story about user stories and test-driven development", Better Software

P. J. Brooke, R. F. Paige, J. L. Jacob (2007). "A CSP model of Eiffel's SCOOP", Journal Of Object Technology, FormalAspects of Computing, 19(4):487–512

J. O. Coplien, T. Reenskaug (2009). "The DCI architecture: a new vision of object-oriented programming", Artima, https://www.artima.com/articles/dci_vision.html

J. O. Coplien (2010). "Restoring function and form into patterns", Gertrud & Cope

N. Schärli, S. Ducasse, O. Nierstrasz, A. Black (2003). "Traits: composable units of behavior", Proc. of ECOOP'03, LNCS, vol. 2743; Springer Verlag, page 248—274, [DOI] 10.1007/b11832

D. Haywood (2013). "The restful objects specification", http://www.restfulobjects.org/

S. Hermann (2000). "Object Teams for Java", https://wiki.eclipse.org/OTJ

A. C. Kay (1972). "A personal computer for children of all ages", Xerox PARC

A. C. Kay (1993). "The early history of Smalltalk", ACM

G. T. Leavens, Y. Cheon (2006). "Design by contract with JML", http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf

B. Meyer (1992). "Applying design by contract", IEEE Computer, 0018-9162/92/1000-0040S03.00

R. Pawson (2004). "Naked Objects", University of Dublin, Trinity College

T. Reenskaug (1995). "Working with objects: the OORAM software engineering method", Prentice Hall Software, ISBN-13: 978-0134529301, ISBN-10: 0134529308

T. Reenskaug (2007). "The case for readable code", Department of Informatics, University of Oslo

T. Reenskaug (2009). "Rethinking the foundations of object orientation and of programming", OreDev, https://vimeo.com/8235394

T. Reenskaug (2010). "DCI execution model", Department of Informatics, University of Oslo

T. Reenskaug, Object Composition Grubu (2014). "DCI glossary", http://fulloo.info

Y. E. Selçuk, N. Erdoğan (2011). “Role models—implementation-based approaches to using roles”, Software: Practice and Experience

M. Siniaalto, P. Abrahamsson (2007). “Does TestDriven Development Improve the Program Code? Alarming Results from a Comparative Case Study”, CEE-SET

F. Torshizi, J. S. Ostroff, R.F. Paigie, M. Checkik (2009). "The SCOOP Concunrreny Model in Java-like Languages ", Communicating Process Architectures

H. A. Valdecantos (2016). “An empirical study on code comprehension: DCI compared to OO”, B. Thomas Golisano College of Computing and Information Sciences Department of Software Engineering, Rochester, New York

ÖZGEÇMİŞ

Ad Soyad **: Cevat Balek**

Doğum Tarihi ve Yeri **: İstanbul 1972**

E-posta **:** cevatbalek@yahoo.com

ÖĞRENİM DURUMU:

- **Lisans** **:** 1998, Boğaziçi Üniversitesi, Mühendsilik Fakültesi, Bilgisayar Mühendisliği Bölümü

1993 yılından bu yana, endüstriyel otomasyondan ses tanıma sistemlerinve bankacılık uygulamalarına kadar geniş bir yelpazede yazılım çözümleri oluşturan bir çok AR-GE kuruluşunda çeşitli rollerde görev yaptı. Tüm çalışmalarında araştırma, tasarım, okunabilirlik, kalite, güvenirlilik, paydaşların iletişimi ve mühendislik etiği konularına titizlikle eğildi. On yılı aşkın bir süredir, görev aldığı önemli kuruluşlarda yalın ve çevik yaklaşımları bizzat uyguladı ve yaygınlaşmasına öncülük etti. İTÜ MİAM'da (Müzikte İleri Araştırmalar Merkezi) bir sene bestecilik üzerine çalışan Cevat Balek, halen İTÜ ARI Teknokent bünyesinde AR-GE çalışmalarına devam etmektedir.