Dokuz Eylül Üniversitesi-Mühendislik Fakültesi
Fen ve Mühendislik Dergisi
Cilt 19, Sayı 57, Eylül 2017

Dokuz Eylul University-Faculty of Engineering
Journal of Science and Engineering
Volume 19, Issue 57, September 2017

DOI: 10.21205/deufmd.2017195778

Bağlamsal Doğrulama için Bir Yazılım Tasarım Şablonu

Tuğkan TUĞLULAR*

İzmir Yüksek Teknoloji Enstitüsü, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, 35430, İzmir

(Alınış / Received: 31.01.2017, Kabul / Accepted: 09.06.2017,
Online Yayınlanma / Published Online: 20.09.2017)

Anahtar Kelimeler
Yazılım Tasarım Şablonu,
Bağlamsal Doğrulama,
Nesne Temelli Tasarım İlkeleri

Özet: Yazılım tasarım şablonları, tekrar eden yazılım tasarım problemleri için hazır çözümler sunar. Model-Görünüm-Denetçi (*İng. MVC*) gibi bileşik tasarım şablonları ise, var olan tasarım şablonlarının biraraya getirilmesi ile daha büyük ölçekli problemleri çözmek için geliştirilmektedir. Bu çalışmada bağlamsal doğrulama problemi için bir bileşik tasarım şablonu geliştirilmiştir. Bağlamsal doğrulama, bir işlem gerçekleştirilmeden önce o işlem için gerekli tüm nesnelerin gerekli koşulları sağladığının doğrulanması anlamına gelmektedir. Bileşik tasarım şablonu geliştirme yöntemi ile ortaya konan bağlamsal doğrulama tasarım şablonu; tek sorumluluk, açık-kapalı ve bağımlılığı tersine çevirme nesne temelli tasarım ilkelerine göre geliştirilmiştir. Geliştirilen bağlamsal doğrulama tasarım şablonu bünyesinde Ziyaretçi, Strateji, Dekoratör ve Bildirim tasarım şablonlarını barındırmaktadır. Ortaya konan bağlamsal doğrulama tasarım şablonu, kullanım şekli itibariyle iki örnek uygulama üzerinde tartışılmıştır.

A Software Design Pattern For Contextual Validation

Keywords
Software Design Pattern,
Contextual Validation,
Object oriented design principles

Abstract: Software design patterns provide readily available solutions for recurring software design problems. Composite design patterns, such as Model-View-Controller (MVC), provide solutions for large scale design problems by bringing existing design patterns together. In this work, a composite design pattern is developed for contextual validation problem. Contextual validation means validating all necessary conditions of all context objects defined by the context. In this work, development method for composite design patterns is followed by using single responsibility, open-closed, dependency inversion principles of object oriented design. The proposed contextual validation design pattern includes Visitor, Strategy, Decorator and Notification design patterns. Usage of the proposed contextual validation design pattern is discussed on two case studies.

*Sorumlu yazar: tugkantuglular@iyte.edu.tr

1. Giriş

Nesne temelli yazılım geliştirme sürecinde doğrulama; veri ve nesnelerin oluşturulmasında, bunların ister metot çağrısı şeklinde ister bir arayüz üzerinden iletilmesinde ve davranışlara ilişkin önkoşullarda önemli yer tutar. Doğrulaması yapılmamış bir veri, yazılımın hatalı çalışmasından çökmesine hatta ele geçirilmesine, örneğin komut enjeksiyonu, kadar geniş bir yelpazede soruna yol açabilir [1]. Nesne temelli yazılımlarda veriler nesne kabuğu içinde yer aldığı için bu noktadan itibaren veri ve nesne doğrulaması yerine sadece nesne doğrulamasından bahsedilecektir.

Nesnelerin doğrulamasında izlenecek yaklaşım veya yöntem konusunda programcıların sorularına internet üzerinde rastlamak mümkündür. İster bu sorulara verilen yanıtlarda, ister konu ile ilgili bağımsız bloglarda önerilen çözümler, var olan bir tasarım şablonunun, ki farklı farklı tasarım şablonları önerilmiştir, nasıl uygulanabileceğine yöneliktir. Bu alandaki bir diğer çözüm yazılım mimarilerinde kullanılan *javax.validation* kütüphanesi olup, çalışma mantığı kısıtlar üzerine kurulmuştur. Belirlenen kısıtların notlar (*İng. annotations*) olarak sınıf tanımları içine eklenmesi ile doğrulama kod parçacıkları otomatik üretilmektedir. Ayrıca, nesnelerin doğrulaması probleminin bir tasarım şablonu ile çözümüne ilişkin bir bilimsel yayın da bulunamamıştır. Bu eksikliği gidermek üzere, nesnelerin doğrulanmasına yönelik olarak yazılım tasarım şablonu geliştirme yöntemi izlenerek bir tasarım şablonu geliştirilmiştir.

Fowler [2] programcıların nesneler için doğrulama metotlarını bağlam bağımsız olarak geliştirdiğini, oysa bir nesnenin bir bağlam için geçerli olurken aynı nesnenin bir başka bağlam için geçersiz olabileceğini, dolayısı ile nesneler için doğrulama metotlarının bağlam bağımlı olarak geliştirilmesi gerektiğini belirtmiştir. Örnek olarak, bir müşterinin bağlam bağımsız geçerli olması ile bir müşterinin otel rezervasyonu için (bağlam bağımlı) geçerli olması karşılaştırmasını yapmıştır. Bu çalışmada geliştirilen tasarım şablonu bağlam bağımlı olarak tasarlanmıştır.

Bağlamsal doğrulama problemi için bu çalışmada önerilen çözüm tasarım şablonu olarak sunulmuştur. Bunun nedenlerinden birisi tekrar tekrar kullanılabilir bir çözümün hedeflenmesidir. Bir diğeri de bağlamsal doğrulama alanına özgü terminolojinin oluşmasını ve kullanılmasını sağlamaktır.

Tasarım şablonlarını atomik ve bileşik olarak sınıflamak mümkündür [3]. Atomik tasarım şablonları tekrar eden problemlere yönelik ortak abstrakt bir çözümü temsil eder. Ancak tekrar eden problemlerin çerçevesi geniş ise bu durumda bir atomik tasarım şablonu yeterli olmayabilir. Onun yerine atomik tasarım şablonlarının biraraya getirilmesi ile ortaya konan bir bileşik tasarım şablonu çözüm olarak sunulabilir. Bunun en çok bilinen örneği Model-Görünüm-Denetçi (*İng. Model-View-Controller, MVC*) bileşik tasarım şablonudur. Model-Görünüm-Denetçi tasarım şablonu; Gözlemci (*İng. Observer*), Strateji (*İng. Strategy*) ve Kompozit (*İng. Composite*) atomik tasarım şablonlarından oluşmaktadır [4]. Bu çalışmada önerilen bağlamsal doğrulama tasarım şablonu da bir bileşik tasarım şablonudur.

Bir sonraki bölümde, izlenen yazılım tasarım şablonu geliştirme yöntemi tanıtılmış ve bağlamsal doğrulama tasarım şablonunun adım adım geliştirimi gösterilmiştir. Üçüncü bölümde bu çalışmada geliştirilen bağlamsal doğrulama yazılım tasarım şablonu ve kullanımı açıklanmıştır. Dördüncü bölümde,

geliştirilen bağlamsal doğrulama yazılım tasarım şablonu bir örnek üzerinde uygulanmış ve yapılan gözlemler değerlendirilmiştir. Son bölümde ise sonuçlar açıklanmış ve olası gelecek çalışmalar listelenmiştir.

2. Bileşik Yazılım Tasarım Şablonu Geliştirme Yöntemi

Bu çalışmada yazılım tasarım şablonu geliştirme yöntemi olarak önce konu araştırması ve incelemesi (*İng. analysis*) gerçekleştirilmiş, sonra bileşik tasarım şablonu geliştirme süreci [5] işletilmiştir. Geliştirme süreci şu adımlardan oluşmaktadır:

1. Konu için bileşik tasarım şablonu geliştirmede kullanılacak atomik ve bileşik tasarım şablonları belirlenir.
2. Belirlenen atomik ve bileşik tasarım şablonları için 2.a'dan 2.d'ye kadar olan adımlar gerçekleştirilir:
    a. Tasarım şablonları bağlam oluşturmaya göre sıralanır. Bir tasarım şablonunun bir diğeri için bağlam oluşturması beklenir. Böylece bir sonrakine bağlam oluşturan tasarım şablon dizisi elde edilir.
    b. Sıradaki tasarım şablonu alınır ve tasarım genişletilir.
    c. Ek tasarım şablonu gereksinimi ortaya çıkarsa atomik ve bileşik tasarım şablonları sıralamasına 2.a adımında belirtildiği şekilde yenileri eklenir.
    d. Sırada bekleyen tasarım şablonu varsa 2.b adımına dönülür.
3. Bileşik tasarım şablonuna gereken detaylar eklenir. Örneğin, gerektiği şekilde sınıf ve metot tanımları genişletilir.

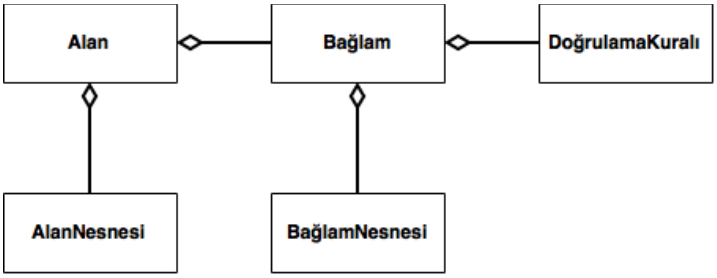
Bu adımlar gerçekleştirilirken Gamma ve arkadaşları tarafından tanımlanan nesne temelli tasarım önerileri izlenmiştir [6]: arayüzlere tasarım yapmak, yığışımı (*İng. aggregation*) kalıtıma (*İng. inheritance*) tercih etmek ve değişeni bulup kabuklamak (*İng. encapsulation*). Ayrıca, aşağıda listelenen nesne temelli tasarım ilkeleri de gözetilmiştir:

- Tek sorumluluk ilkesi (*İng. Single responsibility principle*): Bu ilkeye göre her sınıfın tek bir sorumluluğu olmalı ve bu sorumluluk tamamıyla o sınıf tarafından kabuklanmalıdır [7].
- Açık-kapalı ilkesi (*İng. Open-closed principle*): Bu ilkeye göre sınıflar genişlemeye açık olmalı ama değişikliğe kapalı olmalıdır [8].
- Bağımlılığı tersine çevirme ilkesi (*İng. Dependency inversion principle*): Bu ilkeye göre kullanan sınıf ile kullanılan sınıf arasındaki ilişki kavramsal düzeyde tanımlanmalıdır [7].

3. Bağlamsal Doğrulama Yazılım Tasarım Şablonu

3.1. Bağlamsal Doğrulama Tasarım Şablonu Geliştirimi

Bağlamsal doğrulama konusu üzerinde yapılan araştırmada, Evans'ın [9] "iş katmanının; alan (*İng. domain*) kavramlarını, işin durumu ile ilgili bilgileri, iş kurallarını ve iş mantığına ilişkin tüm işlemleri barındırdığı" ifadesi bulunmuştur. Evans'ın sözünü ettiği "iş mantığına ilişkin tüm işlemlere" bağlamsal doğrulama da dahildir. Dolayısı ile bağlamsal doğrulamanın iş mantığı katmanında yer almasına karar verilmiştir. Gösterim (*İng. presentation*) katmanında gerçekleştirilen doğrulamalar, örneğin form doğrulaması, bağlamsal doğrulama kapsamı dışındadır. Medic'e [10] göre, bağlamsal doğrulamaya ilişkin metotlar alan sınıfları içine konur ise, bu sınıflar büyük ve karmaşık olur. Bu yüzden bağlamsal doğrulama için ayrı sınıflar olmalıdır. Bu bilgiler doğrultusunda ve bağımlılığı tersine çevirme ilkesini ışığında Şekil 1'de yer alan sınıf çizgesi oluşturulmuştur.



Şekil 1. Bağlamsal doğrulama üst seviye sınıf çizgesi

Şekil 1'de bu çalışmaya esas oluşturan model gösterilmiştir. Bu modele göre bir alan kendisini oluşturan nesnelerden meydana gelir. Her alan kendi içinde birçok bağlam barındırır ve bir bağlamda davranış sergilemeden önce bağlamı oluşturan bağlam nesnelerinin, ki onlar alan nesnelerinin bir alt kümesidir, o bağlama ilişkin kurallar çerçevesinde doğrulanması gerekir. Bu doğrulamayı izleyen adımda, doğrulama başarılı ise ilgili davranış sergilenir.

Konu araştırması ve incelemesinden sonra bileşik tasarım şablonu geliştirme sürecine geçilmiştir. Bağlamsal doğrulama yazılım tasarım şablonu için ilk adımda konu ile ilgisi olan tasarım şablonları internet üzerinde yapılan araştırma ile çıkarılmış ve aşağıda alfabetik olarak sıralanmıştır:

- Bildirim (*İng. Notification*): Fowler [2] programcıların nesneler için geliştirdiği doğrulama metotlarının, doğrulama başarısız olduğunda boolean tipinde yanlış döndürecek veya başarısızlığı belirten bir istisna fırlatacak şekilde yazıldığını ifade etmiştir. Boolean tipinde yanlış döndürüldüğünde hatanın ne olduğunu anlamak olası olmamaktadır. İstisna mekanizması bu bilgiyi sağlar ancak ilk istisna yakalandığında program akışı kesilir ve kullanıcı var olabilecek diğer hatalardan haberdar olamaz [11]. Bu yaklaşımlar yerine Fowler [12], Bildirim tasarım şablonunu önermiştir. Bu şekilde, o anki doğrulama-ya ilişkin tüm hatalar türlerine göre saklanır ve ilgili sınıfa bildirilir.
- Dekoratör (*İng. Decorator*): Yapısal tasarım şablonlarından biri olan Dekoratör, bir örneğe (İng. *object instance*) aynı sınıfın diğer örneklerini etkilemeden bir davranış eklenmesini sağlar [6].
- Strateji (*İng. Strategy*): Davranışsal tasarım şablonlarından biri olan Strateji, bir algoritmanın davranışının çalışma zamanında seçilebilmesini sağlar [6]. Strateji doğrulama davranışı için genelde tercih edilir, çünkü doğrulama için seçilecek algoritmaya örneğin girdilere bakarak, çalışma zamanında karar verir.
- Ziyaretçi (*İng. Visitor*): Davranışsal tasarım şablonlarından biri olan Ziyaretçi, nesne yapısı içinde yer alan örnekler üzerinde gerçekleştirilecek yeni bir işlevi temsil eder. Ziyaretçi, bu işlevin örneklere ait sınıfları değiştirmeden tanımlanmasını sağlar [6].

İkinci adımın birinci alt adımında (2.a), ilk adımda çıkarılan tasarım şablonlarının bağlam oluşturacak şekilde sıralanması hedeflenmiştir. Bunun için çıkarılan tasarım şablonlarından şablon ikilileri oluşturulmuş ve hangi tasarım şablonunun diğeri için bağlam oluşturduğu belirlenmiştir. Aşağıda listelenen tasarım şablon ikililerinde solda olan bağlam oluşturan tasarım şablonudur:

- < Ziyaretçi , Strateji >: Ziyaretçi tasarım şablonu, strateji tasarım şablonu için bağlam oluşturur. Doğrulanacak bağlam içinde yer alan nesneler ziyaretçi tasarım şablonu kullanılarak ziyaret edilir ve bu nesnelerin her biri için ayrı ayrı belirlenmiş strateji işletilir. Böylece, stratejilerin farklı farklı bağlamlarda, ve

dolayısı ile onları oluşturan nesnelerde, yeniden kullanımı (*İng. reuse*) sağlanmış olur.

- < Strateji , Dekoratör >: Strateji tasarım şablonu, dekoratör tasarım şablonu için bağlam oluşturur. Bir stratejiyi oluşturan kuralların her biri dekoratör tasarım şablonu aracılığı ile stratejiye eklenir. Böylece, doğrulama kurallarının farklı farklı stratejiler için yeniden kullanımı sağlanmış olur.
- < Dekoratör , Bildirim >: Dekoratör tasarım şablonu, bildirim tasarım şablonu için bağlam oluşturur çünkü her doğrulama kuralı için bir bildirim olasılığı bulunmaktadır.

Sonuç olarak, bağlam oluşturma açısından tasarım şablonları aşağıdaki şekilde sıralanmıştır:

1. Ziyaretçi
2. Strateji
3. Dekoratör
4. Bildirim

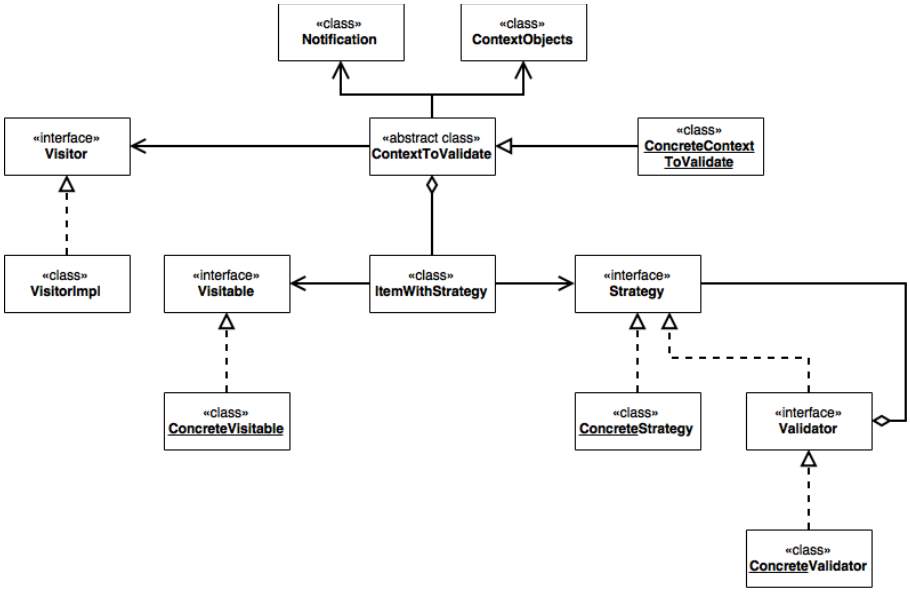
Geliştirim sırasında ek yeni bir tasarım şablonu gereksinimi ortaya çıkmamıştır. Dolayısı ile yukarıda belirtilen tasarım şablonları belirtilen sırada bağlamsal doğrulama tasarım şablonunu oluşturacak şekilde tasarıma eklenmiştir. Bileşik tasarım şablonu geliştirme sürecinin üçüncü adımında sözü edilen detayların eklenmesine, Bildirim tasarım şablonuna ait detayların eklenmesi örnek verilebilir. Başarısızlık, geleneksel olarak derleyicilerde "hata" veya "uyarı" olmak üzere iki değişik türde ifade edilmektedir. Buna "bilgi" türü de eklenebilir ve bu şekilde "hata" veya "uyarı" olmak üzere iki değişik türe sahip Bildirim tasarım şablonuna ait sınıf ve metot tanımları genişletilebilir.

Geliştirim sonucunda ortaya konan bağlamsal doğrulama tasarım şablonu izleyen bölümde açıklanmıştır.

3.2. Bağlamsal Doğrulama Tasarım Şablonu Tanıtımı

Bağlamsal doğrulama tasarım şablonu; doğrulanması gereken her bağlam için, o bağlamda bulunan her nesne için, o nesnenin ziyaret edilerek kendisi için doğrulayıcıların (*İng. validator*) biraraya getirilmesi ile oluşturulmuş stratejinin işletilmesi ve tüm strateji işletim sonuçlarının bildirim olarak döndürülmesini sağlayan tasarım şablonu olarak tanımlanmıştır.

Geliştirilen bağlamsal doğrulama tasarım şablonu sınıf çizgesi Şekil 2'de sunulmuştur. Her sınıfın türü; arayüz (*İng. interface*), soyut sınıf (*İng. abstract class*) ve somut sınıf (*İng. class*) olmak üzere sınıf isimlerinin üzerinde belirtilmiştir. Sınıf isimlerinde altçizgi bulunmayan tüm sınıflar, alan ve bağlam ile ilişkisi bulunmayan tekrar kullanılabilir sınıflardır. Bu sınıflar *tr.edu.iyte.ContextualValidation* paketi (*İng. package*) altında toplanmıştır (bakınız Şekil 3) ve bu paket her projeye herhangi bir değişiklik gerektirmeden eklenebilir. Sınıf adında altçizgi bulunan sınıflar ise bağlamsal doğrulama için alana ve bağlama özgü olarak hazırlanması gereken sınıflardır. Bunların hazırlanma ve kullanım yöntemi izleyen bölümde anlatılacaktır.

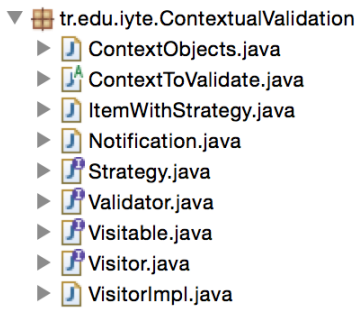


Şekil 2. Bağlamsal doğrulama tasarım şablonu sınıf çizgesi

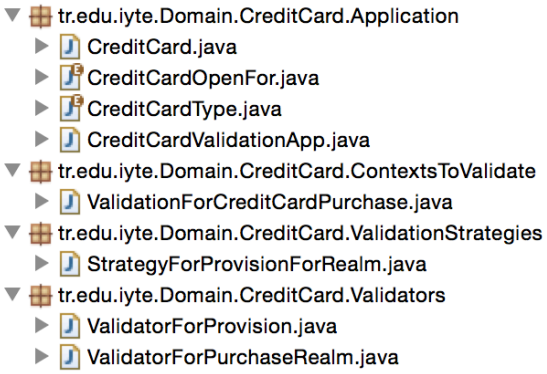
Bağımlılığı tersine çevirme ilkesi kapsamında bağlamsal doğrulama tasarım şablonu, arayüz sınıfları ve bir soyut sınıf üzerine oturtulmuştur. Açık-kapalı ilkesi doğrultusunda stratejiler doğrulayıcıların biraraya getirilmesi ile, doğrulanacak bağlamlar ise <nesne,strateji> ikililerinin biraraya getirilmesi ile oluşturulmuştur. Bağlamsal doğrulama tasarım şablonunda yer alan sınıfların tamamı tek sorumluluk ilkesi kapsamında tasarlanmıştır.

3.3. Bağlamsal Doğrulama Tasarım Şablonu Tanıtımı

Bağlamsal doğrulama tasarım şablonu kullanım yöntemi küçük bir örnek üzerinden anlatılacaktır. Bu örnekte, söz konusu alanın sadece kredi kartı, onun yardımcı sınıfları ve uygulama (*İng. application*) sınıfından oluştuğunu ve Şekil 4'de görülen *tr.edu.iyte.Domain.CreditCard.Application* paketinin bu alanı temsil ettiğini düşünelim. Kredi kartı sınıfı; kart üzerinde yer alan bilgiler ile kartın nerelerde kullanıma açık olduğu (*CreditCardOpenFor.java*), bakiye, limit bilgilerini tutmakta ve karttan çekme ile ödeme işlevlerini tanımlamaktadır. Ayrıca, uluslararası alışverişe açık mı, internet alışverişine açık mı, şu miktar için provizyonu var mı gibi sorulara da boolean tipinde doğru veya yanlış yanıtı dönmektedir.



Şekil 3. tr.edu.iyte.ContextualValidation paket içeriği



Şekil 4. tr.edu.iyte. Domain.CreditCard.* paketleri

```
public void accept(Visitor visitor, Strategy<Visitable> strategy,
                   ContextObjects contextObjects, Notification notification) {

    visitor.visit(this, strategy, contextObjects, notification);

}
```

Şekil 5. Tüm alan nesneleri için standart accept metodu gerçekleştirimi

Bağlamsal doğrulama tasarım şablonu, alan nesnelerinin sınıf tanımına standart olarak sadece *implements Visitable* eklenmesini ve *Visitable* arayüzü tarafından zorlanan *accept* metodunu Şekil 5'de görüldüğü biçimde sınıf içinde standart olarak gerçekleştirmesini beklemektedir. Bunun dışında alan nesnelerine bir ekleme yapılmasına gerek yoktur. Bu yaklaşım nesne temelli tasarım ilkelerinden tek sorumluluk ilkesi ile de örtüşmektedir. Alan nesnesine yeni bir sorumluluk verilmemekte sadece ziyaretçi tasarım şablonuna uyması sağlanmaktadır.

Alan nesneleri hazırlandıktan sonra doğrulanacak bağlamların bir pakette, doğrulama stratejilerinin bir pakette ve doğrulayıcıların bir pakette toplanması önerilmektedir. Şekil 4'de buna uygun bir paket yapısı kullanılmıştır. Bu sayede doğrulanacak bağlamların, doğrulama stratejilerinin ve doğrulayıcıların izlenmesi ve kullanımı kolaylaşacaktır.

Kredi kartı örneğinde tek bir alan nesnesi olduğu için doğrulanacak bir bağlama bir doğrulama stratejisi karşılık gelmektedir. Ancak Bölüm 4'de verilen örnek uygulamada doğrulanacak bağlam içinde birden fazla doğrulama stratejisi olabileceği görülecektir. Kredi kartı örneğinde iki doğrulayıcı bulunmakta ve bu örneğin her iki doğrulama stratejisinde de bu iki doğrulayıcı birlikte yer almaktadır. Eğer bu doğrulayıcılar statik olsa idi, toplam dört doğrulayıcı olması gerekirdi. Oysa doğrulayıcıların gerek duyduğu bilgileri (nesne formunda) ve nesneleri bağlam nesne listesi şeklinde elde etmesi sağlanmış ve bu sayede daha esnek bir yapı oluşturulmuştur.

```
public void validate(Visitable obj,
                     ContextObjects contextObjects,
                     Notification passedNotification) {

    // bildirim nesnesinin ilklenmesi
    notification = passedNotification;

    // somut strateji temeli
    if (obj == null) {
        notification.addError("NULL OBJECT");
    }

    // ek doğrulayıcılar
    validator = new ValidatorForPurchaseRealm(validator);

    validator = new ValidatorForProvision(validator);

    validator.validate(obj, contextObjects, notification);
}
```

Şekil 6. Örnek bir validate metodu gerçekleştirimi

Bu yaklaşımın uygulanmasına örnek olarak *StrategyForProvisionForRealm* sınıfı içinde *Strategy* arayüzü tarafından zorlanan *validate* metodu Şekil 6'da verilmiştir. Bu metot içinde, öncelikle bildirim nesnesi ilklenmekte, sonra somut strateji temeli olarak ziyaret edilmesi istenen nesnenin var olup olmadığı denetlenmekte ve ardından stratejiyi tamamlayacak ek doğrulayıcılar dekoratör tasarım şablonu yaklaşımı ile eklenmektedir.

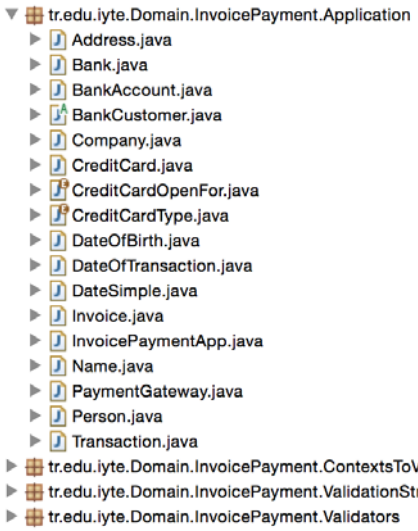
4. Örnek Uygulama

Örnek uygulama olarak bir fatura ödeme uygulaması seçilmiştir. Bu uygulama için isimleri Şekil 7'de görülen 17 sınıf kodlanmıştır.

Fatura ödeme uygulaması *InvoicePaymentApp* içindeki main metodunun çalıştırılması ile başlar. İlk olarak fatura ödemesi için gereken alan nesneleri oluşturulur. Ardından bir fatura düzenlenir. Bu örnekte fatura ödemesinden önce bağlamsal doğrulama gerektiği varsayılmış ve bu doğrulama noktasında önerilen bağlamsal doğrulama tasarım şablonu kullanılmıştır. Önerilen bağlamsal doğrulama tasarım şablonunun kullanım yöntemi örnek uygulama üzerinde açıklanarak anlatılacaktır. Doğrulama noktasında önce doğrulama bağlamı için gerekli nesneler oluşturulur. Bu nesneler şunlardır:

- bağlam nesneleri (*İng. context Objects*),
- söz konusu bağlamda doğrulanacak unsurlar için doğrulama stratejileri (*İng. itemsWithStrategies*),
- söz konusu bağlamda doğrulama sonucu olarak başlığı atanmış boş bildirim (*İng. notification*) nesnesi,

Sonra Şekil 8'de görülen kaynak kod bloğu yazılır:



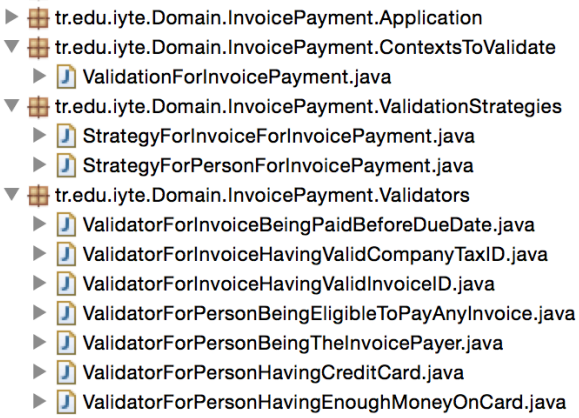
Şekil 7. tr.edu.iyte.Domain.InvoicePayment.* paketleri

```
ContextToValidate contextToValidate = new ValidationForInvoicePayment(contextObjects,
                                                                       itemsWithStrategies,
                                                                       notification);
Notification notificationResult = contextToValidate.validate();
if (!notificationResult.hasErrors()) {
```

Şekil 8. Bağlamsal doğrulama için standart kaynak kod bloğu

Şekil 8'de görüldüğü üzere bağlamsal doğrulamanın çağrıldığı kaynak kod bloğunda, bağımlılığı tersine çevirme ilkesi doğrultusunda fatura ödeme uygulama sınıfları ile kullanılan bağlamsal doğrulama sınıfları arasındaki ilişki kavramsal düzeyde tanımlanmıştır. *ValidationForInvoicePayment* sınıfı, *ContextToValidate* sınıfını genişleten somut sınıftır (bakınız Şekil 2). Doğrulama metodu olan *validate* metodundan dönen bildirim içinde hata yok ise fatura ödemesi için gerekli kodlar çalıştırılır, hata(lar) var ise bu(nlar) kullanıcıya bildirilir.

Fatura ödemesi için bağlamsal doğrulamayı oluşturan stratejiler ve doğrulayıcılar Şekil 9'da gösterilmiştir.



Şekil 9. Fatura ödemesi için bağlamsal doğrulamayı oluşturan stratejiler ve doğrulayıcılar

Örnek uygulamada fatura ödeme bağlamı için doğrulanacak unsurlar fatura ve ödeyecek kişi olarak belirlenmiş ve her biri için ayrı bir strateji tanımlanmıştır. Bu sayede bağlamsal doğrulamada kullanılan kurallar bir hiyerarşiye yerleştirilmiş olur. Strateji ve doğrulayıcı sınıflarının adlandırılmasında kullanılan yöntem ile okunurluk arttırılmıştır. Fatura için ödeme tarihi geçmemiş fatura olması, fatura üzerindeki şirket vergi numarasının geçerli olması ve fatura seri numarasının geçerli olmasına yönelik doğrulayıcılar tanımlanmıştır. Faturayı ödeyecek kişi için ise yaş itibari ile fatura ödeyebilecek durumda olması, faturanın kesildiği kişi olması, kredi kartının kendisine ait olması ve kredi kartında ödeme için yeterli kullanılabilir limit bulunmasına yönelik doğrulayıcılar kullanılmıştır. Bağlamsal doğrulama tasarım şablonunu oluşturan sınıflar ise Ek A'da verilmiştir.

5. Sonuç

Bu çalışmada bağlamsal doğrulama problemi için bir yazılım tasarım şablonu ortaya konulmuştur. Var olan Ziyaretçi, Strateji, Dekoratör ve Bildirim tasarım şablonlarından yararlanılarak oluşturulan bir bileşik tasarım şablonu çözüm olarak sunulmuştur. Bileşik tasarım şablonlarının en çok bilinen örneği Model-Görünüm-Denetçi tasarım şablonudur. Böylece bileşik tasarım şablonlarına bir yenisi eklenmiştir.

Bileşik tasarım şablonu geliştirme yöntemi ile ortaya konan bağlamsal doğrulama tasarım şablonu; tek sorumluluk, açık-kapalı ve bağımlılığı tersine çevirme nesne temelli tasarım ilkelerine göre geliştirilmiştir. Önerilen bağlamsal doğrulama tasarım şablonunun kullanımına yönelik öneriler iki örnek üzerinde verilmiştir.

Gelecek çalışma olarak *javax.validation* doğrulama kütüphanesi ile entegrasyon planlanmaktadır. Ayrıca, önerilen bağlamsal doğrulama tasarım şablonu ile ortaya konan bilgi birikimi kullanılarak bağlamsal doğrulama için alana özgü modelleme dili geliştirilebilir ve bu sayede bağlamsal doğrulamaya ilişkin kaynak kodların kısmen de olsa otomatik olarak üretilmesi sağlanabilir. Son olarak, bağlamsal doğrulama yazılım şablonunun, Meyer ve Arnout [13] tarafından ziyaretçi tasarım şablonunun bir bileşene dönüştürülmesine benzer şekilde, bir bileşen haline getirilmesi söz konusu olabilir.

Kaynakça

[1] Tuglular, T., Belli, F., Linschulte, M. 2016. Input Contract Testing of Graphical User Interfaces, Int J Softw Eng Knowl Eng. World Scientific, Cilt. 26(2), s. 183–215. DOI: http://dx.doi.org/10.1142/S0218194016500091 .

[2] Fowler, M. 2005. Contextual Validation. http://martinfowler.com/bliki/ContextualValidation.html (Erişim Tarihi: 15.10.2016).

[3] Riehle, D. 1997. Composite Design Patterns. 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications New York, USA, s. 218–28.

[4] Hericko, M., Beloglavec, S. 2005. A composite design pattern identification technique, Informatica, Cilt. 29(4), s. 469–476. DOI: bilinmiyor.

[5] Shalloway, A., Trott, J.R. 2004. Design patterns explained: a new perspective on object-oriented design. 2nd edition, Addison-Wesley, Boston, USA, 480s.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994. Design patterns: Elements of reusable object-oriented software. Addison-Wesley Professional, Boston, USA, 395s.

[7] Martin, R.C. 2003. Agile software development: principles, patterns, and practices. Pearson, USA, 529s.

[8] Meyer, B. 1997. Object-oriented software construction. 2 edition, Prentice Hall, New York, USA, 1296s.

[9] Evans, E. 2004. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, Boston, USA, 560s.

[10] Medic, J. 2016. Context Validation in Domain-Driven Design. https://www.toptal.com/scala/context-validation-in-domain-driven-design (Erişim Tarihi: 15.10.2016).

[11] Fowler, M. 2014. Replacing Throwing Exceptions with Notification in Validations. http://martinfowler.com/articles/replaceThrowWithNotification.html (Erişim Tarihi: 15.10.2016).

[12] Fowler, M. 2004. Notification. http://martinfowler.com/eaaDev/Notification.html (Erişim Tarihi: 15.10.2016).

[13] Meyer, B., Arnout, K. 2006. Componentization: the Visitor example, IEEE Computer, Cilt. 39(7), s. 23–30. DOI: 10.1109/MC.2006.227.