"UML sınıf diyagramları, hem gereksinim analizi hem de nesne yönelimli yazılım sistemlerinin tasarımı için en önemli araçlardan biridir. Bu diyagramlar, sınıfları, özelliklerini ve işlemlerini, ayrıca sınıflar arasındaki çeşitli ilişki türlerini gösterir. 2. Bölümde, sınıf diyagramlarının temellerinden bazılarını, özellikler, işlemler ve genellemeler dahil olmak üzere tanıttık. 3. Bölümde, Nesne İstemci-Sunucu Çerçevesini temsil etmek için bu elemanlardan bazılarını bir araya getirdiğinizi gördünüz. Bu bölümde, ek örnekler kullanarak sınıf diyagramlarını detaylı bir şekilde inceleyeceğiz."

Bu bölümde aşağıdakiler hakkında bilgi edineceksiniz:

■ UML sınıf diyagramlarının en temel özelliklerini nasıl doğru bir şekilde kullanacağınız: sınıflar, ilişkiler, genellemeler ve arayüzler.

■ Nesne Kısıtlama Dili (OCL) temelleri.

■ Sınıf diyagramları ile modelleme yaparken karşılaşacağınız tipik problemler.

■ Sınıf diyagramlarını sistemli bir şekilde geliştirmek için adım adım süreç.

■ Sınıf diyagramlarını Java'da uygulamanın temel teknikleri.

UML (Unified Modeling Language), nesne yönelimli yazılım modellemesi için standart bir grafik dilidir. 1990'ların ortalarında James Rumbaugh, Grady Booch ve Ivar Jacobson'un işbirliği ile geliştirilmiştir. Her biri 1990'ların başlarında kendi gösterimlerini geliştirmiş olan bu üç geliştirici, UML'deki 'U' harfi 'birleştirilmiş' anlamına gelir, çünkü geliştiriciler daha önce geliştirdikleri dillerin en iyi özelliklerini birleştirdiler. UML standardının müstevlisi Object Management Group (OMG)'dir. OMG, 2004 yılında UML'nin 2.0 versiyonunu onaylamıştır.

UML, bir dizi diyagram türünü içerir, bunlar arasında:

■ Sınıf diyagramları, sınıfları ve aralarındaki ilişkileri tanımlar. Bu bölümün konusudur.

■ Etkileşim diyagramları, nesnelerin birbiriyle nasıl etkileşimde bulunduğu açısından sistemlerin davranışını gösterir. 8. Bölümde, etkileşim diyagramlarının iki türünü: sıra diyagramları ve iletişim diyagramlarını tartışacağız.

■ Durum diyagramları ve aktivite diyagramları, sistemlerin nasıl davrandığını gösterir. Bunları da 8. Bölümde sunacağız.

■ Bileşen ve dağıtım diyagramları, sistemlerin çeşitli bileşenlerinin mantıksal ve fiziksel olarak nasıl düzenlendiğini gösterir. Bunları 9. Bölümde ele alacağız.

Ancak UML, sadece diyagram çizme notasyonlarından oluşan bir setten çok daha fazlasıdır; aşağıdaki ilginç ek özelliklere sahiptir:

■ İle oluşturduğunuz diyagramlar, birleştirilmiş bir model oluşturmak amacıyla birbirine bağlanmış olarak tasarlanmıştır;

■ Detaylı semantiklere sahiptir, notasyonlarının birçok yönünün matematiksel anlamını tanımlar.

■ Uzantı mekanizmaları vardır, bu mekanizmalar yazılım tasarımcılarının UML'nin çekirdeğinin bir parçası olmayan kavramları temsil etmelerine olanak tanır. Bu mekanizmaların bazı örneklerini göstereceğiz.

■ Diyagramların elemanları hakkında çeşitli gerçekleri resmi olarak belirtmenize olanak tanıyan Nesne Kısıtlama Dili (OCL) adında ilişkilendirilmiş bir metin tabanlı dil vardır. Bu önemli konuyu bazı örnekler aracılığıyla tanıtacağız.

UML'nin amacı, yazılım geliştirmeye yardımcı olmaktır. Bir metodoloji değildir, çünkü adım adım ne yapılması gerektiğini açıklamaz.

Neden standart bir modelleme dili kullanmalısınız?

Bazı geliştiriciler, diyagramları veya modelleme dillerinin diğer özelliklerini kullanmadan küçük yazılım sistemleri geliştirmede başarılı olmuştur. Ancak sistemleri büyüdükçe, bu tür geliştiriciler 'büyük resmi' görmekte giderek daha fazla zorluk çekiyor ve kötü tasarımlar oluşturma ve işlerini çok daha uzun zaman alma eğiliminde oluyorlar.

Bu nedenle çoğu sistem diyagramlar kullanılarak belgelenir. Bunlar, yalnızca koda veya metinsel açıklamalara bakarak anlaşılması zor olan yapı ve işlevsellik görünümleri sağlar. Başka bir deyişle diyagramlar soyutlama sağlar.

Bir model yalnızca bir dizi diyagramın ötesine geçer. Bir model, sistem hakkında birbiriyle ilişkili bir dizi bilgiyi yakalar: bir diyagram, bu bilginin yalnızca bir görünümüdür. Birkaç diyagram, aynı bilgiyi, farklı gösterimlerle veya farklı ayrıntı düzeylerinde, biraz farklı şekillerde sunabilir. Diyagramdan bir öğeyi silebilir ve onu modelde tutabilirim; modelden bir öğeyi silersem, tüm diyagramlardan kaybolması gerekir.

Bir model, yazılım mühendislerinin sistem hakkında içgörü sahibi olmasına yol açabilir; problemleri ve diğer özelliklerini keşfetmek için modeli analiz edebilirler (manuel olarak veya araçlar kullanarak). Modelden oluşturulan basit diyagramlar aynı zamanda müşteriler ve kullanıcılarla iletişim kurmaya da yardımcı olabilir. Ancak bu anlaşılması kolay görünümlerin oluşturulması modelleyicinin sorumluluğundadır.

İyi tanımlanmış bir standart modelleme dili olan UML'nin kullanılması ek avantajlar sağlar:

■ Standart bir notasyon olduğu için modele bakan herkes aynı şekilde yorumlayabilecektir.

■ UML modelleri oluşturmak ve bir sistemin tamamı veya bir kısmı için simülasyonu, animasyonu ve/veya kod üretimini mümkün kılmak için çok çeşitli araçlar mevcuttur.

Sınıf diyagramları bir yazılım sisteminde bulunan verileri açıklar. 2. Bölüm'de öğrendiğiniz gibi, bu diyagramlardaki sınıfların çoğu gerçek dünyadaki şeylere karşılık gelir. Örneğin bir havayolu rezervasyon sisteminde Uçuş, Yolcu ve Havaalanı gibi sınıflar bulunacaktır.

Sınıf diyagramlarında gösterilen ana semboller şunlardır:

■ Veri türlerini temsil eden sınıflar.

■ Sınıf örneklerinin diğer sınıf örneklerine nasıl referans verdiğini gösteren ilişkiler.

■ Örneklerde bulunan basit veriler olan nitelikler.

■ Örnekler tarafından gerçekleştirilen işlevleri temsil eden işlemler.

■ Sınıfları miras hiyerarşileri halinde düzenlemek için kullanılan genellemeler. Bu sembollerin nasıl doğru şekilde kullanılacağını açıklayarak başlayacağız. Daha sonraki bölümlerde, sınıf

diyagramlarının daha gelişmiş özelliklerinin yanı sıra sınıf diyagramlarının çizilmesine yönelik adım adım bir yaklaşımı tartışacağız.

Sınıflar

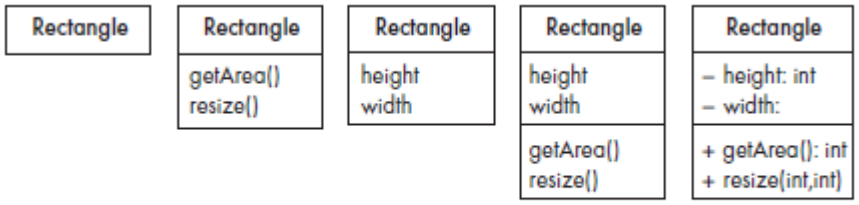
Bir sınıf, içinde sınıfın adının bulunduğu bir kutu olarak temsil edilir. isim her zaman tekil olmalı ve büyük harfle başlamalıdır.

Sınıf diyagramında bir sınıf çizdiğinizde sistemin o isimde bir sınıf içereceğini ve sistem çalıştığında o sınıfın örneklerinin oluşturulacağını söylüyorsunuz.

İsteğe bağlı olarak sınıf diyagramı her sınıfta yer alan öznitelikleri ve işlemleri de gösterebilir. Bu, bir sınıf kutusunun iki veya üç küçük kutuya bölünmesiyle yapılır: üstteki kutu sınıf adını içerir, sonraki kutu nitelikleri listeler ve alttaki kutu ise işlemleri listeler. Nitelikleri veya işlemleri belirtmek istemiyorsanız kutuyu çıkarmanız yeterlidir.

Şekil 5.1 bir sınıfın birkaç farklı ayrıntı düzeyinde nasıl çizilebileceğini göstermektedir. Ne kadar ayrıntı göstereceğiniz, gelişim aşamasına ve ne iletmek istediğinize bağlıdır. En soldaki örnekte, yalnızca sınıfın var olduğunu belirten yalnızca sınıf adı gösterilir. Aynı Rectangle sınıfının diğer dört gösteriminde ek ayrıntılar gösterilmiştir. Niteliklerin türü, özelliğin genel (+) veya özel (-) olup olmadığı dahil en fazla ayrıntı ve

İşlemlerin imzası en sağdaki örnekte gösterilmektedir. Tüm ayrıntılarıyla gösterildiğinde, bir işlemin imzası aşağıdaki gösterim kullanılarak belirtilir: OperationName(parameterName: parametreType,...): returnType.

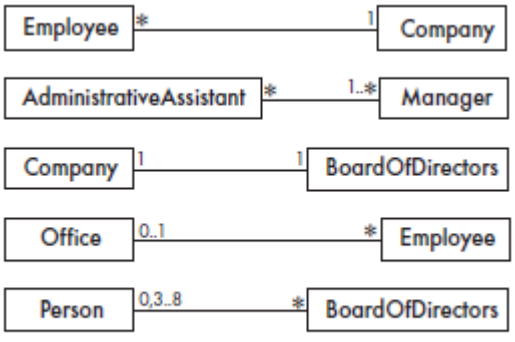


The **Rectangle** class at several different levels of detail

Associations and multiplicity

İki sınıfın örneklerinin birbirine nasıl referans vereceğini göstermek için bir ilişki kullanılır. İlişkilendirme sınıflar arasında bir çizgi olarak çizilir.

Çokluğu belirten semboller ilişkinin her iki ucunda gösterilir. Çokluk, ilişkinin bu ucundaki sınıfın kaç örneğinin, sınıfın diğer ucundaki bir örneğine bağlanabileceğini gösterir.



Examples of possible multiplicities

İki sınıfın örneklerinin birbirine nasıl referans vereceğini göstermek için bir ilişki kullanılır. İlişkilendirme sınıflar arasında bir çizgi olarak çizilir.

Çokluğu belirten semboller ilişkinin her iki ucunda gösterilir. Çokluk, ilişkinin bu ucundaki sınıfın kaç örneğinin, ilişkinin diğer ucundaki sınıfın bir örneğine bağlanabileceğini gösterir. Şekil 5.2, çeşitliliğini gösteren bazı dernek örneklerini vermektedir.

1'in çokluğu, ilişkinin diğer ucundaki her nesneye tam olarak bir örneğin bağlı olması gerektiğini belirtir. Örneğin, Şekil 5.2'de her Çalışanla ilişkili yalnızca bir Şirket olabilir. Çok yaygın bir çokluk, normalde 'çok' olarak okunan ∗'dir ve sıfırdan büyük veya sıfıra eşit herhangi bir tam sayı anlamına gelir. Örneğin Şekil 5.2'de birçok çalışan bir şirketle ilişkilendirilebilir; bir olasılık, bir şirketin hiç çalışanı olmamasıdır. Teorik olarak bir üst sınır olmasa da, mevcut bellek miktarına ve işlem kapasitesine bağlı olan pratik bir üst sınır vardır.

İlişkilendirmenin diğer ucundaki bir nesneye sıfır veya bir nesne bağlı olabiliyorsa, bu durumda çokluğun 'isteğe bağlı' olduğu söylenir ve 0..1 gösterimi kullanılır. Örneğin Şekil 5.2, çalışan başına sıfır veya bir ofis olabileceğini göstermektedir. Yani bir çalışanın bir ofiste görevlendirilmesi isteğe bağlıdır (bazıları evde veya ofis gerektirmeyen bir işte çalışabilir).

Çokluğu, alt ve üst sınır arasında iki nokta olarak gösterilen bir aralık olacak şekilde de belirtebilirsiniz. Aralığa bazen aralık da denir. Örneğin, bir yelkenli teknenin 1 ile 3 arasında direğe sahip olabileceğini belirlerseniz, ilişkinin direk ucuna 1..3 yazarsınız. Yukarıda tartışılan 0..1 notasyonu aralığın özel bir durumudur. Bir aralığın üst sınırı yoksa yıldız işaretini kullanırsınız; dolayısıyla 0..∗ ve ∗ aynı anlama gelirken 1..∗ 'en az bir' anlamına gelir.

Çokluk belirli bir pozitif tamsayı olabilir; ayrıca virgüllerle ayrılmış birkaç çokluk değeri veya aralığı da belirtebilirsiniz. Örneğin, bazı yargı bölgelerindeki kanunun, bir yönetim kurulunun üç ila sekiz arasında üyeye sahip olması gerektiğini belirttiğini düşünün. Ayrıca, kurulda yeterli sayıda üye bulunmadığı takdirde kurul otomatik olarak feshedilir ve yeni seçimlerin yapılması gerekir; seçim sürecinde kurulda sıfır üye bulunur. Şekil 5.2'deki son örneğin çokluğu bu durumu yansıtmaktadır: Yönetim Kurulu'nda ya sıfır olabilir ya da 3 ila 8 kişi olabilir.

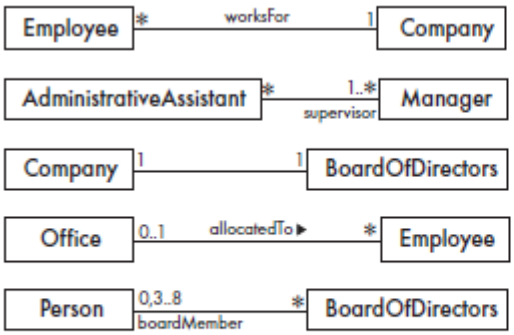
Aralıkları veya ikiden büyük kesin sayıları içeren spesifik çokluklar yaygın değildir ve yalnızca dikkatli bir şekilde düşünüldükten sonra belirtilmelidir. Örneğin, bir kişinin her zaman tam olarak iki ebeveyne sahip olması gerektiğini belirtmek isteyebilirsiniz. Ancak bunu yaparsanız, sistemin her zaman herkesin iki ebeveyninin kaydına sahip olmasını zorunlu kılarsınız. Böyle bir kurala sıkı sıkıya bağlı kalmak imkansız olacaktır çünkü herkes ebeveynlerinin kim olduğunu bilmez ve ayrıca sistemin ebeveynlerin ebeveynlerini sonsuza dek bilmek zorunda olması gerekir. Ebeveynler için daha makul bir çokluk 0..2 olabilir.

Bir ilişkilendirme ucunun çokluğunu belirtmezseniz, tanımsız olduğu söylenir. Bir sınıf diyagramının anlamının çoğu çokluklardan geldiğinden, çokluğu hiçbir zaman tanımsız bırakmamanızı önemle tavsiye ederiz. UML'nin önceki bazı sürümlerinde çokluğun boş bırakılması, bunun tanımsız yerine 'bir' olarak yorumlanması gerektiği anlamına geliyordu; bu kuralı kullanan bazı eski diyagramları görebilirsiniz.

Labeling associations

Her bir ilişki, ilişkinin doğasını açıklığa kavuşturmak için etiketlenebilir.

İki tür etiket vardır: ilişkilendirme adları ve rol adları. Şekil 5.3, Şekil 5.2'dekiyle aynı ilişkileri göstermektedir ancak etiketler eklenmiştir.



The associations of Figure 5.2, but with some association names and role names added

İlişkilendirme adı bir fiil veya fiil cümlesi olmalıdır ve çağrışımın ortasının yanına yerleştirilmelidir. Bir sınıf fiilin öznesi, diğer sınıf ise fiilin nesnesi olur. Örneğin, Çalışan ile Şirket arasındaki ilişkiye WorksFor adı verilir. Derneği bir yönüyle 'bir çalışan bir şirkette çalışıyor' şeklinde okuyabilirsiniz. İlişkilendirmeyi okuma yönü normalde açıktır ancak küçük bir ok (dolu üçgen) gösterilerek açıklığa kavuşturulabilir.

ilişkilendirme adının yanında (Şekil 5.3'teki dördüncü ilişkilendirmede olduğu gibi).

Bir ilişkilendirmeyi etiketlemenin başka bir yolu da rol adı kullanmaktır. Rol adları bir ilişkinin bir ucuna veya her iki ucuna da eklenebilir. Bir rol adı, ilişkilendirme bağlamında bağlı olduğu sınıf için alternatif bir ad görevi görür. Örneğin, Person ve BoardOfDirectors arasındaki ilişkide boardMember, kurul üyesi olan kişileri tanımlayan bir rol adıdır. Bu derneği şöyle okuyabilirsiniz: 'Yönetim kurulu ya

sıfır veya 3 ila 8 kişi kurul üyesi olabilir.

Hem ilişkilendirme adını hem de rol adlarını atlarsanız, bir ilişkinin adının varsayılan olarak yalnızca "ş" olduğunu düşünün. Bu çok bilgilendirici olmasa da bazı durumlarda yeterlidir çünkü ilişkinin anlamı sadece iki sınıfa bakarak netleşebilir. Örneğin Şirket ile Yönetim Kurulu arasındaki ilişkiye herhangi bir etiket eklememeyi seçtik; Dernek şunu okurdu: 'Bir şirketin bir yönetim kurulu vardır'.

Analizi gerçekleştirirken dikkat edilmesi gereken iyi bir kural şudur: ilişkilendirmeyi açık ve net hale getirmek için yeterli ad ekleyin. Normalde aynı ilişkiye hem rol adlarını hem de bir ilişki adını eklemek gerekli değildir. Örneğin, Şekil 5.3'teki ilk ilişkilendirmede Şirket'in yanına işveren rol adını ekleyebilirdik; ancak, WorksFor ilişkilendirme adı yeterlidir. Biz de yapabiliriz

ilişkilendirme adı yerine rol adını kullanmayı seçtiniz.

Analyzing and validating associations

İlişkilendirmeler oluştururken hata yapmak çok yaygındır; çokluğu yanlış anlamak özellikle kolaydır. Bu nedenle, anlamlı olup olmadığını doğrulamak için her çağrışımı her iki yönde de okuma alışkanlığı edinmelisiniz. En önemlisi, her zaman kendinize daha az kısıtlayıcı bir çokluğun da bazı durumlarda anlamlı olup olmayacağını sormalısınız. Daha az kısıtlayıcı derken, 'bir' veya başka bir belirli sayı yerine 'çok' veya 'isteğe bağlı' kullanmayı kastediyoruz.

Genel olarak sistemin esnekliğini artırmak için daha az kısıtlayıcı olma yönünde hata yapmalısınız. Örneğin, bir çalışanı denetleyen kişi sayısını 'bir' ile sınırlamak, bir kişinin birden fazla yöneticiye sahip olabileceği bir 'matris yönetimi' sisteminin getirilmesini zorlaştıracaktır. Öte yandan, gerekçelendirilmediğinde 'bir' yerine 'çok' ifadesini kullanmak sistemin karmaşıklığını artıracak ve verimliliğini azaltacaktır.

Aşağıdaki noktalar en yaygın üç kalıptan söz etmektedir:

çokluk, bunların her biri Şekil 5.2 ve 5.3'te gösterilmektedir.

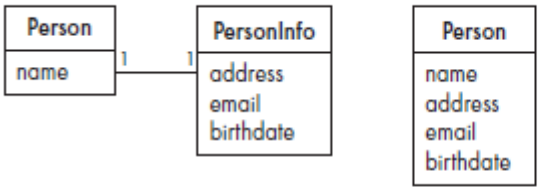
■ Bire çok. Bir şirketin birçok çalışanı vardır, ancak bir çalışan yalnızca bir şirkette çalışabilir. Birisinin birden fazla şirkette çalışarak ek iş yapabileceğini düşünerek bunun yanlış olduğunu iddia edebilirsiniz. Ancak şirket politikası, sistemimiz tarafından yönetilen şirketlerde ek iş yapılmasına açıkça izin vermeyebilir. Bu çokluk modeli doğru bir şekilde, bir şirketin 'paravan' şirket örneğinde olduğu gibi sıfır çalışana sahip olabileceğini göstermektedir. Son olarak, bir şirkette çalışmadığınız sürece çalışan olmanız mümkün olmadığından, Şirket tarafındaki çokluğun isteğe bağlı değil, tam olarak tek olduğu doğru bir şekilde gösterilmiştir.

■ Çoktan çoğa. Bir yönetici asistanı birçok yöneticinin yanında çalışabilir ve bir yöneticinin birden fazla yönetici asistanı olabilir. Elbette herhangi bir idari asistan ile yönetici arasında bire bir ilişki tipik bir durumdur, ancak genel olarak bir grup yönetici için çalışan asistanlar ve bir grup asistana sahip olacak kadar kıdemli yöneticiler vardır.

Ayrıca bazı yöneticilerin hiç asistanının olmadığı da bir gerçektir. Bir asistanın geçici olarak sıfır yöneticiye sahip olmasının mümkün olup olmadığını düşündüğünüzde ilginç bir soru ortaya çıkıyor. Şekil 5.2 ve 5.3'te buna izin vermemeye ve sistemin her zaman en az bir yöneticinin her idari asistanı denetlemesini sağlamasını zorunlu kılmaya karar verdik.

■ Bire bir. Her şirket için tam olarak bir yönetim kurulu bulunmaktadır. Ayrıca yönetim kurulu tek bir şirketin yönetim kuruludur. Bir şirketin her zaman bir yönetim kurulu olması gerekir ve yönetim kurulu da her zaman bir şirkete ait olmalıdır. Yönetim kurulu üyelerinin tamamı istifa etse ne olur? Kurulun hala var olduğunu ancak geçici olarak sıfır üyeye sahip olduğunu söyleyebiliriz. En yaygın çokluk modeli bire-çoktur. Bir sonraki en yaygın olanı çoktan çoğadır. Bu iki kalıp birlikte çağrışımların büyük çoğunluğunu oluşturur. Daha sonra çoktan çoğa bir ilişkinin nasıl iki bire çok ilişkiye bölünebileceğini göreceğiz.

Bire bir ilişkiler daha az yaygındır. Böyle bir birliktelik gördüğünüzde, aslında uçlardan birinin veya her ikisinin de 'isteğe bağlı' mı yoksa 'çoklu' olarak mı değiştirilmesi gerektiğini kendinize sormalısınız. Bire bir ilişkilendirmenin anlamı, sınıflardan birinin örneğini oluşturduğunuzda, aynı anda diğerinin örneğini de yaratmanız gerektiğidir; ve birini sildiğinizde diğerini de silmeniz gerekir. Gerçek bir bire bir ilişki varsa, bunun daha sonra tartışılacak şekilde bir toplama olup olmadığını da düşünebilirsiniz. Yaygın bir hata, iki sınıfın gerçekte bir olması gereken iki sınıf arasında bire bir ilişki oluşturmaktır. Örneğin, Şekil 5.4'te Person ve PersonInfo, PersonInfo'nun niteliklerinin Person'a aktarılmasıyla tek bir Person sınıfı haline gelmelidir.



An inappropriate one-to-one association (left), and a corrected model showing a single class (right)

Figure 5.5 Associations related to booking passengers on a flight

Şekil 5.5, üç sınıfı içeren olası bir çokluk modelini göstermektedir.

Anlambilimine ayrıntılı olarak bakalım. Her Rezervasyon için her zaman tam olarak bir Yolcunun olması gerektiğini söyleyebiliriz, ancak her Yolcunun herhangi bir sayıda Rezervasyonu olabilir (yani farklı uçuşlarda ve tarihlerde). Benzer şekilde, her Rezervasyon için her zaman tam olarak bir Özel Uçuş olmalıdır, ancak her Özel Uçuşun herhangi bir sayıda Rezervasyonu olabilir (tabii ki uçağın kapasitesine kadar).

Şekil 5.5'teki sol ilişkilendirmeyi aşağıdaki iki yönde okuyup analiz edeceğiz:

‘Bir Yolcunun istediği sayıda Rezervasyonu olabilir’

Bu açıklama, bir yolcunun hiçbir şekilde rezervasyon yapamayacağını söylüyor. Bu makul görünüyor mu? Yine, buna cevap vermek için daha fazla gereksinim analizi yapılması gerekebilir. Olasılıklardan biri, herhangi bir rezervasyonu olmayan bir yolcu hakkındaki bilgileri saklayarak yer israf etmek istemeyebileceğimizdir; dolayısıyla, yalnızca bir rezervasyonu olan bir yolcuyu her zaman silmeye karar verebilir ve onu iptal edebiliriz. Eğer kararımız buysa, o zaman çokluk ∗'den 1..∗'e değiştirilebilir. Bu değişikliği yaparsak, yeni bir yolcu eklediğimizde şunu yapmamız gerekecek:

ilk rezervasyonlarını aynı anda ekleyin. Öte yandan, sisteme bazı yolcuları ekleyip daha sonra geri dönüp rezervasyon eklemek de uygun olabilir. Bu nedenle, bir Yolcunun Rezervasyon olmadan var olmasına izin verilmesinin kabul edilebilir olduğu sonucuna varacağız ve

çokluğu ∗'da bırakın.

Bir yolcunun birden fazla rezervasyonu olabilir mi? Evet, çünkü birisinin bir dizi uçuş ayarladığını hayal etmek kolaydır.

Association classes

Bazı durumlarda, ilişkili iki sınıfı ilgilendiren bir nitelik, sınıflardan herhangi birine yerleştirilemez. Örneğin, Şekil 5.6'da gösterilen, bir öğrencinin herhangi bir sayıda kurs bölümüne kayıt olabileceği ve bir kurs bölümünün herhangi bir sayıda öğrenciye sahip olabileceği ilişkiyi hayal edin. Öğrencinin notu hangi sınıfa yazılmalıdır?

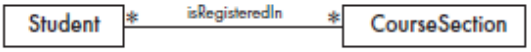


Figure 5.6 A many-to-many association. The 'grade' attribute can be put in neither class

Ne zaman çoktan çoğa bir ilişki gördüğünüzde şunu düşünmelisiniz:

bir ilişkilendirme sınıfının gerekli olup olmadığı.

Şekil 5.7'nin sol yarısındaki diyagram aşağıdaki şekle dönüştürülebilir:

sağ yarıda yalnızca bire çok ilişkileri kullanan diyagram. Bir birlik sınıfıyla çoktan çoğa ilişkiyle bağlanan herhangi bir sınıf çifti bu şekilde dönüştürülebilir. 'Çok' çokluğun dönüşümden önceki ve sonraki konumlarına özellikle dikkat edin.

Şekil 5.7'nin her iki yarısı da aynı şekilde uygulanacaktır. Ancak bazen soldaki versiyon daha nettir çünkü Öğrenci ile CourseSection arasındaki ilişkinin önemini vurgulamaktadır. Diğer zamanlarda, özel bir gösterim gerekmediği için doğru versiyonun okunması daha kolay olabilir.

Şekil 5.5'in, Şekil 5.7'nin sağ yarısındaki modeli izlediği için bir ilişkilendirme sınıfını kullanacak şekilde de dönüştürülebileceğini unutmayın.

Bir ilişkilendirme sınıfının iki çoktan bire ilişkilendirmeye sahip olması, bir örnek oluşturmak için zaten iki ilişkili sınıfın örneklerine sahip olmanız gerektiği anlamına gelir.

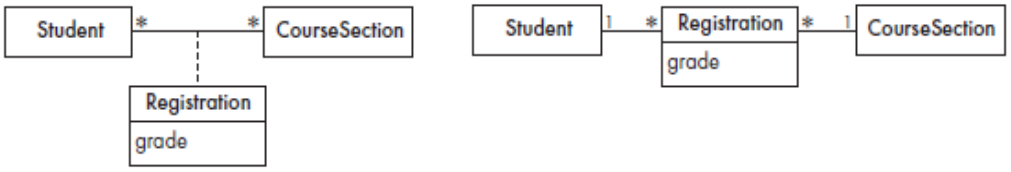


Figure 5.7 A many-to-many association with an association class, and an equivalent diagram using two one-to-many associations

Reflexive associations

Bir ilişkinin bir sınıfı kendisine bağlaması mümkündür. Bunun iki örneğini Şekil 5.8'de bulabilirsiniz. Bir ders, öncelikle başka önkoşul derslerin alınmasını gerektirebilir. İki ders neredeyse aynı materyali kapsıyorsa, bunlardan birini almak öğrencinin diğerini almasını engelleyebilir ve bunun tersi de geçerlidir; bu tür derslerin birbirini dışladığı söylenir. İlk ilişki asimetriktir çünkü her iki uçtaki sınıfların rolleri açıkça farklıdır. İkincisi ise simetriktir. Anlamı açıklığa kavuşturmak için, bir ilişki adı yerine rol adlarını kullanarak asimetrik dönüşlü ilişkilendirmeyi etiketlemelisiniz.

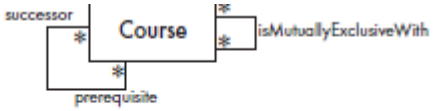


Figure 5.8 Two examples of reflexive associations

Links as instances of associations

Bir nesnenin bir sınıfın örneği olduğunu söylediğimiz gibi, bir bağlantının da bir ilişkinin örneği olduğunu söylüyoruz. Her bağlantı iki nesneyi (ilişkide yer alan iki sınıfın her birinin bir örneği)

birbirine bağlar. Örneğin, Şekil 5.5'te her Rezervasyon için Yolcu-Rezervasyon ilişkisinin bir bağlantısı olacaktır.

Directionality in associations

İlişkilendirmeler ve bağlantılar varsayılan olarak çift yönlüdür. Yani, eğer bir Sürücü nesnesi bir Araba nesnesine bağlıysa, o zaman Araba da dolaylı olarak o Sürücüye bağlıdır. Eğer arabayı tanıyorsanız, sürücüsünü bulabilirsiniz; ya da sürücüyü tanıyorsanız, arabayı öğrenebilirsiniz.

Bir uca bir ok ekleyerek bir ilişkinin bağlantılarının gezinilebilirliğini sınırlamak mümkündür. Örneğin, Şekil 5.10 bir takvim uygulamasında bulunabilecek iki sınıfı göstermektedir. Bu uygulamanın kullanıcısı herhangi bir sayıda yazılı notu herhangi bir günle ilişkilendirebilir. Day sınıfının bir örneğinin, kendisiyle ilişkili Note örnekleri hakkında bilgi sahibi olması gerekir; ancak bir Nota sahipseniz ait olduğu Günü belirlemeye gerek duyulması beklenmez.

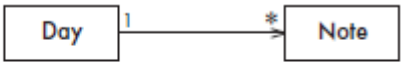


Figure 5.10 A unidirectional association

İlişkilendirmeleri tek yönlü yapmak verimliliği artırabilir ve karmaşıklığı azaltabilir, ancak aynı zamanda sistemin esnekliğini de sınırlayabilir.

Generalization

Bunların üst sınıfa işaret eden küçük bir üçgen kullanılarak temsil edildiğini hatırlayacaksınız. Isa kuralına ve diğer bazı kurallara da uymaları gerekir. Burada genellemeler oluştururken dikkate alınması gereken bazı konuları daha sunacağız.

Avoiding unnecessary generalizations

Yeni başlayanların yaptığı yaygın bir hata, genellemeyi abartmaktır. Şekil 5.11'de bir müzik mağazası tarafından satılabilecek farklı türdeki ürünlerin sınıflandırması gösterilmektedir. Ancak her sınıfın varlığını haklı çıkarabilmek için o sınıfta farklı şekilde yapılacak bazı işlemlerin olması gerekir. Şekil 5.11 örneğinde, çoğu sınıf için farklı yöntemlerin yazılması gerektiğini hayal etmek zor olacaktır. Örneğin, JazzRecording, ClassicalRecording ve BluesRecording, nasıl satıldıkları veya müşterilerin onlar hakkında ne tür bilgiler edinebileceği açısından farklılık göstermez.

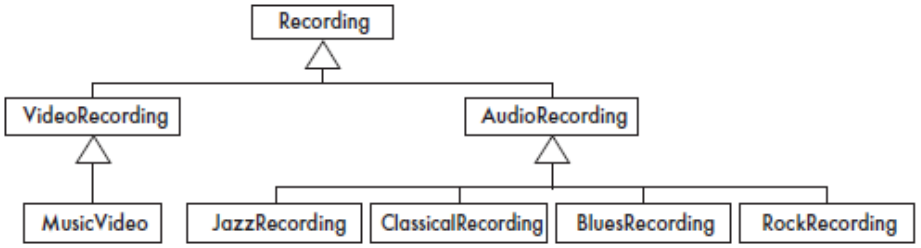


Figure 5.11 A hierarchy of classes in which there would not be any differences in operations. This should be avoided

Şekil 5.11'deki bilgiyi modellemenin daha iyi bir yolu, Şekil 5.12(a)'daki gibi bir sınıf diyagramı oluşturmaktır. Şekil 5.11'deki sınıfların çoğu artık RecordingCategory'nin örnekleri haline gelir ve Şekil 5.12 (b)'de gösterildiği gibi hiyerarşinin kendisi de bir örnekler hiyerarşisi haline gelir. Aslında, Şekil 5.12 (b) bir nesne diyagramı örneğidir; bunları birazdan daha detaylı tartışacağız.

Handling multiple generalization sets

Bir genelleme kümesi, ortak bir üst sınıfa sahip etiketli bir genellemeler grubudur; etiket, üst sınıfı iki veya daha fazla alt sınıfa ayırmak için kullanılan kriterleri açıklar. Tüm genellemeleri tek bir açık üçgen kullanarak bir kümede birleştirmek en açıktır. Etiketi açık üçgenin yanına yerleştirirsiniz.

Zooloji programında kullanılan genelleme kümelerinin iki örneği Şekil 5.13'te gösterilmektedir. Hayvanlar, yaşam alanlarına göre suda yaşayan ve karada yaşayan hayvanlara veya yiyecek türlerine göre etoburlar ve otçullar olarak ikiye ayrılabilir.

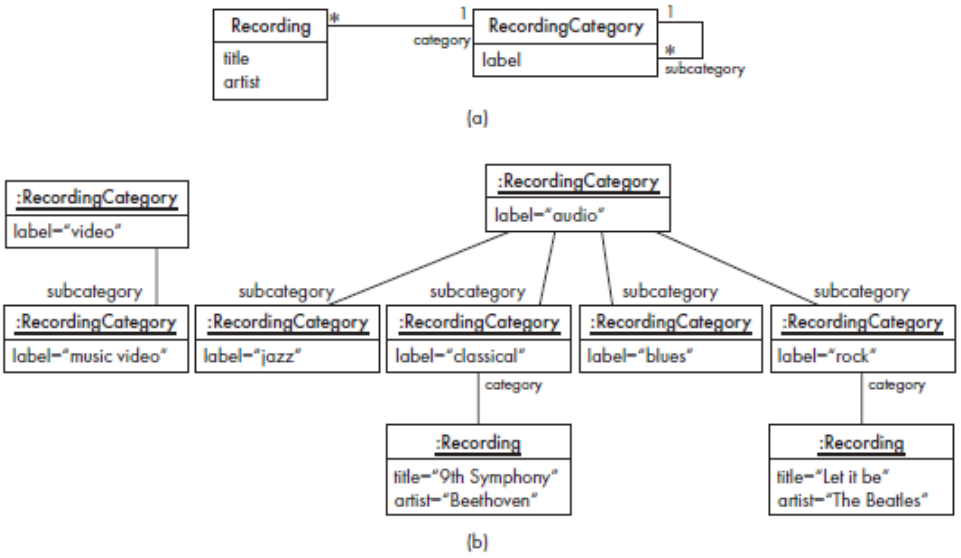


Figure 5.12 (a) Modeling a taxonomy of products using a single class, and (b) an object diagram generated from this

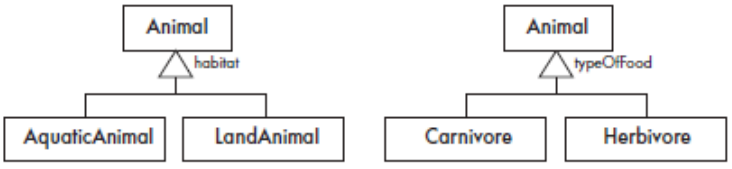


Figure 5.13 Two generalization sets

Bir genelleme kümesinin etiketi tipik olarak şu özelliklere sahip bir nitelik olacaktır:

Her alt sınıfta farklı değer. Yukarıda tartışıldığı gibi, onların varlığını haklı çıkarmak için alt sınıfların özelliklerinde başka farklılıkların da olması gerekir; bu farklılıklar nitelikler, işlemler veya ilişkiler olabilir. Örneğin, bir Etoburun bir av ilişkisi, avlanma Stratejisini açıklayan bir özelliği ve bu verileri işlemeye yönelik operasyonları olabilir.

Aynı üst sınıfı paylaşan birden fazla olası genelleme kümesinin olduğu Şekil 5.13 gibi durumlar, ilginç modelleme zorlukları ortaya çıkarmaktadır. Her iki genelleme kümesini de aynı modele dahil ederseniz, Java gibi uygulama ortamlarında bir sorun ortaya çıkar: Bir hayvan Suda Yaşayan Hayvan ise, aynı zamanda Etobur da olamaz. Bu nedenle Şekil 5.13, köpekbalıkları gibi suda yaşayan etoburların temsil edilmesini zorlaştıracaktır. Bu sorun bizi hayvanların mümkün olan tüm habitat ve yiyecek türü kombinasyonlarına sahip olmalarını sağlayacak bir yol aramaya yöneltiyor. Şekil 5.14'te gösterilen çözümlerden biri, daha yüksek seviyeli bir genelleme seti (burada habitat) oluşturmak ve ardından hiyerarşide daha düşük bir seviyede yinelenen etiketlere sahip genelleme setlerine sahip olmaktır. Bunun dezavantajı tüm özelliklerin olmasıdır.

ikinci genelleme kümesiyle ilişkili olanların da kopyalanması gerekir.

Örneğin, hem AquaticCarnivore hem de LandCarnivore için bir av birliği sağlamanız gerekir. Bu çözümle ilgili bir diğer sorun da sınıf sayısının çok fazla artabilmesidir. Omnivorları eklemek istiyorsanız hem AquaticOmnivore hem de LandOmnivore'u eklemeniz gerekir. Bu nedenle Şekil 5.14 ideal bir çözüm değildir.

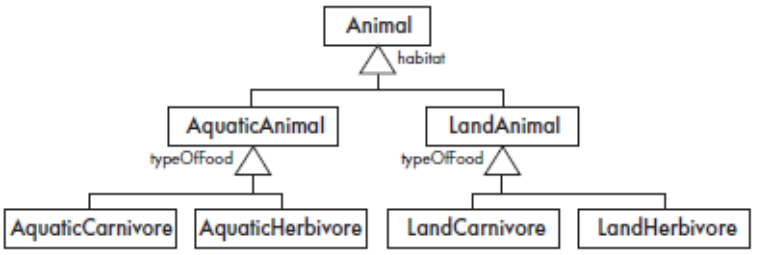


Figure 5.14 Allowing different combinations of features by duplicating a generalization set label at a lower level of the hierarchy. Duplication like this should be avoided

Çoklu kalıtım kullanan başka bir olası çözüm Şekil 5.15'te gösterilmektedir. Bu yaklaşım daha fazla sınıf ve genelleme kullanır ancak özelliklerin tekrarlanmasını önler. Ancak çoklu kalıtım genellikle çok fazla karmaşıklık katar. Bu örnek normalde bundan kaçınılması gerektiğinin bir nedenini göstermektedir; ikinci bir neden ise Java'da çoklu kalıtımın bulunmamasıdır.

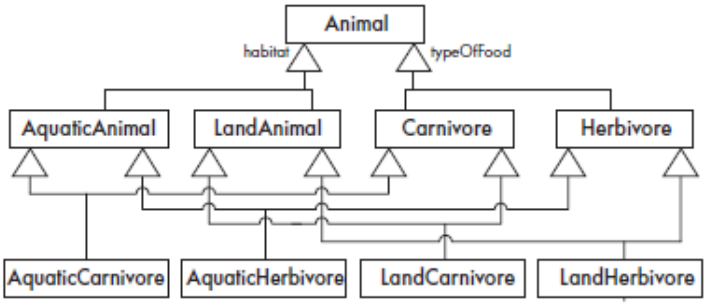


Figure 5.15 Allowing different combinations of features by using multiple inheritance. This is complex and should be avoided

Avoiding having objects change class

Genellemeler oluştururken ortaya çıkabilecek diğer bir sorun da nesnelerin sınıf değiştirmesine gerek kalmamasıdır. Genel sınıfta. Çoğu programlama dilinde sınıfı değiştirmek kesinlikle mümkün değildir;

bu nedenle orijinal nesneyi tamamen yok etmeniz ve ikinci sınıfın yeni bir örneğini yaratmanız gerekir. Bu karmaşıktır ve hataya açıktır çünkü tüm örnek değişkenleri kopyalamanız ve eski nesneye bağlanan tüm bağlantıların artık yeni nesneye bağlandığından emin olmanız gerekir.

Bir nesnenin sınıfını değiştirme ihtiyacı Şekil 5.16'da gösterilmektedir. Bir öğrencinin öğrenimi sırasında devam durumunun tam zamanlıdan yarı zamanlıya veya tam zamanlıdan yarı zamanlıya değişebileceği açıktır. Bu durumu sisteminizde bir PartTimeStudent'i yok ederek ve bir FullTimeStudent oluşturarak veya öğrencinin durumu her değiştiğinde tam tersini oluşturarak modellemek istemezsiniz. Bu nedenle Şekil 5.16 zayıf bir modeldir. Olası bir çözüm, katılım durumu'nu Öğrenci'nin bir niteliği haline getirmek ve iki alt sınıfı tamamen hariç tutmaktır. Sorun

bu, PartTimeStudent ve FullTimeStudent'te farklılık gösterecek herhangi bir işlem için polimorfizm avantajını kaybetmemizdir. Bir sonraki bölümde tartışılan oyuncu-rol modeli yine daha iyi bir çözüm sağlayabilir.

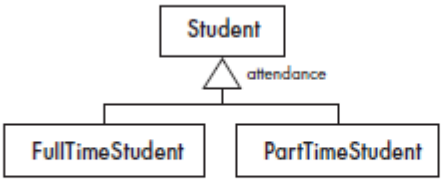


Figure 5.16 A situation in which objects will need to change class from time to time. Generalizations of this type should be avoided