Relationer mellan objekt

Att utveckla en applikation

När man utvecklar en applikation börjar man självklart inte direkt att programmera. Först måste man analysera problemet och utveckla en design för lösningen. Här följer 4 huvudfaser vid applikationsutveckling.

1. Analysfas: Förstå problemet som ska lösas och kraven på programmet. Vad ska användaren göra med applikationen?
2. Designfas: Skapa en design av programmet. Vilka objekt/klasser kommer att behövas? Hur samverkar dessa objekt? Vilka operationer kommer att behövas i klasserna.
3. Implementationsfas: Realisera designen i något programmeringsspråk.
4. Testning: Uppfyller applikationen kraven från analysfasen?

Observera att det är först i det tredje steget, implementationen, som man börjar skriva kod.

Att hitta objekt och relationer mellan dessa

En viktig del av designfasen är att förstå vilka objekt, och alltså klasser, som behövs samt hur dessa objekt ska samverkar sinsemellan. I det senare fallet talar man om vilka relationer som finns mellan objekten.

Ett exempel, kortspel

Tänk dig att vi ska skapa en applikation där användaren ska kunna spela ett enkelt kortspel, t.ex. 21, mot datorn. Under analysfasen ska vi skaffa oss en klar beskrivning av vad som händer när spelet spelas. Detta kan vi göra genom att studera ett verkligt 21-spel och sedan överföra det till en situation där en användare spelar mot en dator.

Objekt

En del av beskrivningen kan se ut som nedan. De substantiv som kan tänkas beskriva blivande objekt i programmet har strukits under.

Dealern blandar kortleken och delar ut 2 kort till spelarens hand. Spelaren betraktar korten och beräknar deras totala poäng. Om kortens totala poäng är 21 har spelaren vunnit, om poängen är över 21 har spelaren förlorat. Om poängen är mindre än 21 anger spelaren om hon vill ha fler kort. . .

Här har vi följande kandidater till objekt/klasser: *kort, kortlek, hand, spelare, dealer och poäng.*

Naturligtvis måste man rensa en sådan lista från dubbletter (kandidater som beskriver samma sak), kandidater som inte är relevanta för problemets lösning och kandidater som snarare är datamedlemmar i något objekt än ett objekt i sig (poäng kan vara en datamedlem i klassen som representerar spelaren).

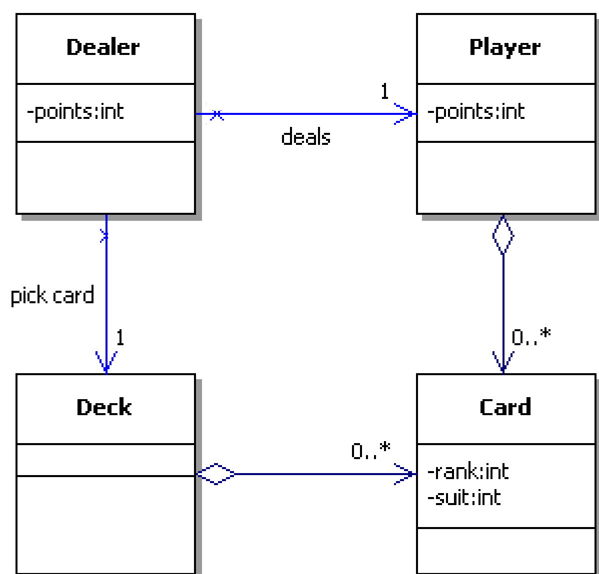
Relationer mellan objekt

Det är viktigt att vi kan beskriva vilka relationer som finns mellan objekten. Detta ger oss senare viktig information om operationer som måste definieras i klasserna och också om datamedlemmar som måste definieras.

I vårt exempel ser jag följande relationer:

- Dealern känner till kortleken (för att ta kort)
- Dealern känner till spelaren (delar ut kort)
- Kortleken har (består av) 0 till 52 kort
- Spelaren har ett antal kort

Vi beskriver vår lösning, och relationerna, så här långt med en bild.
[I de ”boxar” som beskriver klasser anges klassens namn högst upp, under detta datamedlemmar och längst ned metoder. ”-” står för private och ”+” står för public.]



Diagrammet följer syntaxen för klassdiagram i Unified Modeling Language, UML, ett generellt språk för modellering av alla typer av system.

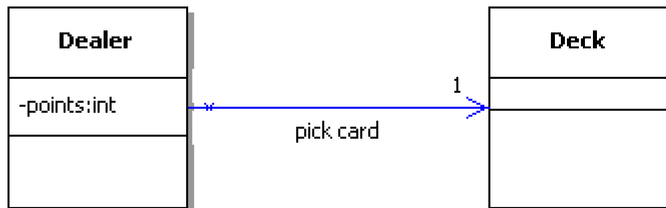
Det finns 3 huvudtyper av relationer mellan objekt i UML,

- Association: känner till
- Aggregat: har, är uppbyggd av
- Beroende (dependency)

Association

En association mellan objekt av typer beskriver att objekten *känner till* varandra. I vårt fall känner Dealer till Deck för att kunna ta kort ur denna.

En association ritas med ett streck mellan klassymbolerna för de aktuella objekten. Man kan ange riktning för associationen som en pil; pilens riktning är riktad mot den klass som tillhandahåller tjänsten, i detta fall Deck som tillhandahåller tjänsten att lämna ut ett kort.

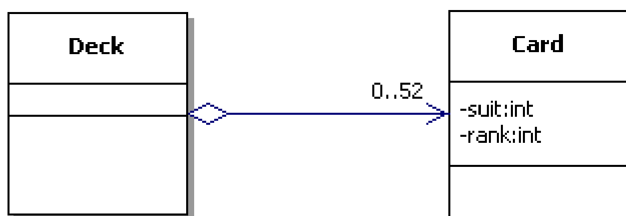


Siffran närmast 1 vid associationens anger kardinaliteten, eller multipliciteten, för associationen. Den ska läsas som att ett objekt av typen Dealer associerar till ett och endast ett objekt av typen Deck. Texten på associationssymbolen är en beskrivning av associationen.

Aggregat

Ett speciellt fall av association är aggregat. Aggregat innebär att en klass, den aggregerande, är *uppbyggd* av objekt av andra klasser. Aggregat beskrivs ofta med *har en* eller *består av*.
I vårt exempel består en kortlek av kort; objekt av typen Deck är aggregat av 0 till 52 objekt av typen Deck.

Ett aggregat representeras av ett heldraget streck mellan klasserna med en romb vid den aggregerande klassen.



Kardinaliteten är alltså 0..52 för Card-objekt i denna relation.

Delat aggregat versus komposition

Det kan vara bra att skilja på om delobjekten i ett aggregat exklusivt tillhör ett enda aggregat eller inte, komposition respektive delat aggregat.

Delat aggregat, shared aggregation

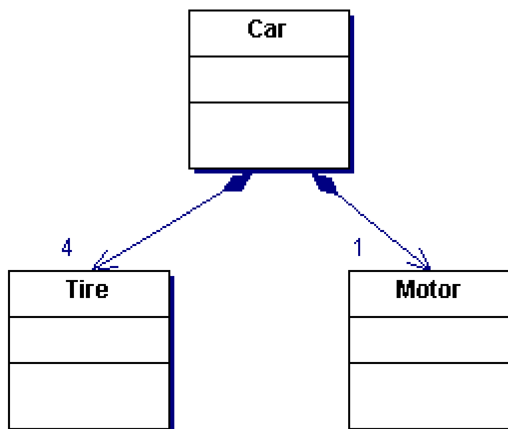
Vid delat aggregat är delobjekten som aggregatet består av utbytbara och/eller kan delas mellan, tillhöra, flera aggregat.
Relationen mellan Deck och Card ovan är ett exempel på delat aggregat. Ett kort i leken kan ju lämnas ut från denna för att i stället hamna hos en spelare.

Delat aggregat anges med en ofylld romb.

Komposition, composite aggragation

Vid komposition tillhör delobjekten exklusivt det aggregerande objektet. Delobjekten lämnas aldrig ut till andra objekt, och delobjektens livslängd bestäms av aggregatets livslängd, när aggregatet försvinner gör också delobjekten detta.

En bil består av en motor och fyra däck, som bara tillhör denna bil. När bilen försvinner, försvinner också motor och däck.



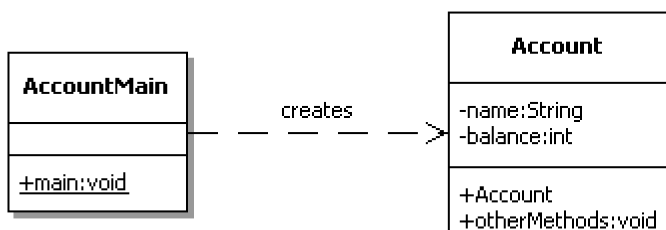
Som synes anges komposition med en fylld romb vid den aggregerande klassen.

Dependency, kortvarigt beroende

Dependency syftar i UML på att ett objekt är beroende av ett annat. Association och aggregat är exempel på beroenden, men man brukar använda begreppet dependency speciellt vid kortvariga, flyktiga beroenden.

Om en klass har en metod där andra objekt används endast under exekveringen av metoden (lokala variabler i metoden snarare än datamedlemmar) är det att betrakta som dependency. T.ex. har de flesta klasser ett beroende till klassen String (via toString-metoden). En klass med en main-metod där objekt skapas lokalt i main kan sägas ha ett beroende till dessa objekt.

Dependency anges med en streckad pil som i exemplet nedan.

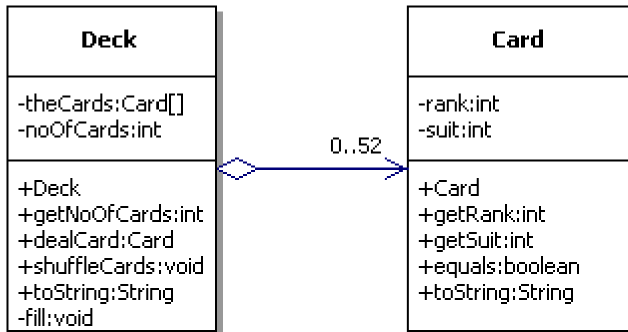


Operationer, metoder

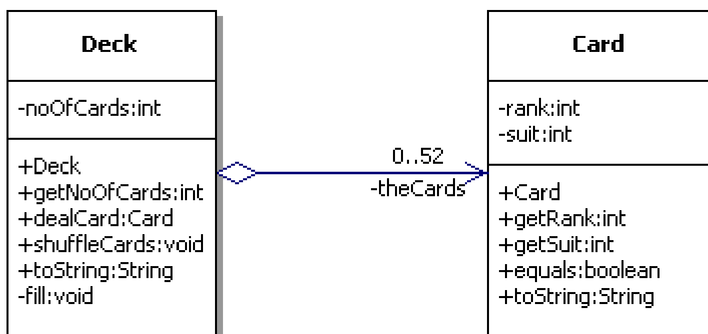
I ett senare skede av designfasen söker vi vilka operationer, metoder, som behövs. Ofta får vi information om dessa operationer från relationerna mellan objekt.

Ett exempel i fallet med kortspelet: ”*Dealern blandar kortleken och delar ut 2 kort …*”. Detta ger oss att vi behöver metoderna shuffle och getCard i klassen Deck.

Ett fullständigt klassdiagram för Deck kanske ser ut som nedan. Notera att aggregatet har realiserats med en datamedlem, theCards, av typen Card-array i klassen Deck.



Det är vanligt att man inte anger datamedlemmar som representerar en association, eller ett aggregat, inuti klassymbolen. Det framgår ju redan av relationssymbolen att de finns, som i figuren nedan.



Kodexempel i Java

Som du nog redan förstått realiseras association och aggregat med datamedlemmar som refererar det/de objekt som relationen pekar mot. I en aggregerande klass finns datamedlemmar som är referenser till de delobjekt som det aggregerande objektet har.

I exemplet ovan med kortleken, Deck, kan det se ut så här i källkoden.

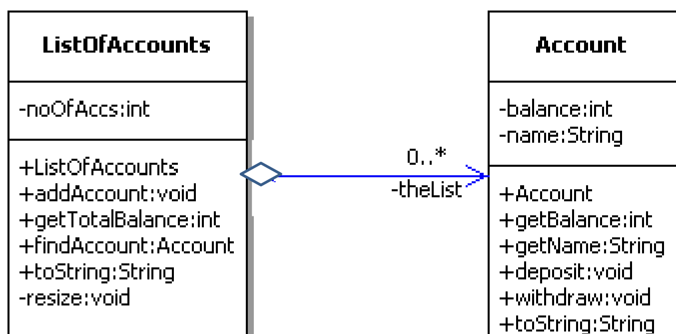
```
public class Deck {

        private Card[] theCards; // Aggregation
        private int noOfCards;

        /** Create a new deck with 52 cards.
         */
        public Deck() {
                theCards = new Card[52];
                // Create the 52 individual cards
                ...
        }

        ...
}
```

Exempel på association alternativt delat aggregat

Det är vanligt att vi behöver en klass som lagrar en samling av objekt av någon typ[1]. Vi har då en klass som har en array av objektreferenser, till de objekt som ska lagras, som datamedlem.
I just detta fall är det inte klart om detta ska beskrivas som en association, känner till, eller som ett aggregat, har/består av. Faktum är att i detta fall ser implementationen i Java likadan ut oavsett om vi betraktar relationen som en association eller ett aggregat.

I vårt exempel ska vi studera en klass, ListOfAccounts, som har till uppgift att lagra en samling av kontoobjekt, Account. Vi vill att det ska vara möjligt att lägga till kontoobjekt till vår lista och att söka efter ett visst objekt i listan.

Se UML-diagrammet nedan. Kardinaliteten (siffrorna på aggregatsymbolen) står för att en kontolista kan innehålla från inget till obegränsat antal konton (0..*).



I klassen kommer det att behövas en array med kontoreferenser som datamedlem (av typen Account[]). Vi låter den ha plats för 100 konton initialt. Det behövs också en datamedlem, noOfAccs,

[1] Det finns visserligen färdiga API-klasser, som ArrayList, men här är poängen att exemplifiera relationer och hur dessa realiseras i kod.

som anger hur många konton som för tillfället finns i listan.
Konstruktorn skapar arrayen och sätter noOfAccs till 0.

```
public class ListOfAccounts {

    private Account[] theList;
    private int noOfAccs;

    public ListOfAccounts() {
        theList = new Account[100];
        noOfAccs = 0;
    }

    ...
```

Vi lägger till nya objekt i listan med metoden addAccount. Det nya objektet placeras efter det sista i listan och antalet räknas upp ett steg. Innan detta utförs måste vi dock kontrollera om arrayen är full och om så är fallet utöka storleken (det senare görs i hjälpmetoden resize). Så här ser addAccount ut:

```
    public void addAccount(Account a) {
        if(noOfAccs >= theList.length) {
            this.resize();
        }
        theList[noOfAccs] = a;
        noOfAccs++;
    }
```

Vi behöver en metod som returnerar en referens till ett sökt konto. Kom ihåg att vi inte kan komma åt inkapslade konton och deras metoder om inte aggregerande klassen själv lämnar ut referenser till dessa. Metoden findAccount tar ett sökt namn (kontoinnehavare) som argument och returnerar en referens till kontot eller, om det inte hittas, null.

```
    public Account findAccount(String name) {
        Account a = null; // initialize
              String foundName;
        for(int i = 0; i < noOfAccs; i++) {
            foundName = theList[i].getName();
            if(foundName.equals(name)) { // Not ==!
                a = theList[i];
                break;
            }
        }
        return a;
    }
```

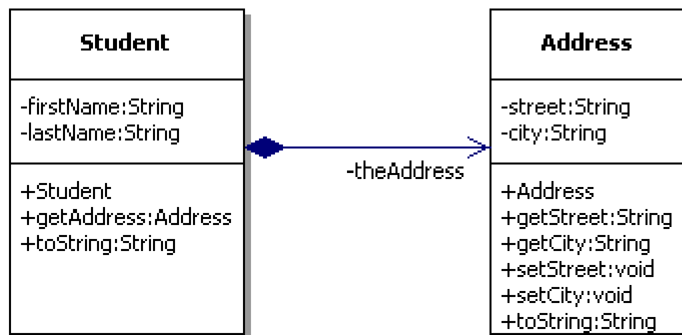
Notera att det via den *referens* som returneras, av metoden findAccount, blir möjligt att ändra i det Account-objekt som finns i ListOfAccounts privata array!

Det fullständiga kodexemplet finns på kurssidan.

Exempel på komposition, composite aggregation

Komposition är ett specialfall av aggregat där delobjekten inte kan delas med andra objekt och där delobjekten försvinner då det aggregerande objektet försvinner.

Betrakta följande situation. Information om studenter ska lagras i ett register. En student har en adress, denna adress modelleras av en separat klass. Studentklassen deklarerar alltså en datamedlem som är en referens till ett adressobjekt, se nedan.



Vi vill dock inte att det ska vara möjligt att via metoden getAddress, i klassen Student, ändra den privata adressen. Kom ihåg att detta blir möjligt om vi lämnar ut en referens till den privata datamedlemmen theAddress.

Lösningen blir att låta konstruktorn i klassen student själv skapa adressobjektet och sedan aldrig lämna ut en referens till detta. Metoden getAddress skapar en kopia, en klon, av adressobjektet och returnerar en referens till detta.
Detta är ett exempel på komposition, composite aggregation.

```
public class Student {

   private String firstName, lastName;
   private Address theAddress;

   public Student (String first, String last,
                   String street, String city) {
      firstName = first;
      lastName = last;
      theAddress = new Address(street, city); // Create the address object
   }

   public Address getAddress() {
          // Return a copy (i.e. a clone)
          return new Address(theAddress.getStreet(), theAddress.getCity());
   }

   ...

}
```