UML-syntax

Lennart Andersson
Datavetenskap, LTH

20 januari 2013

1 Inledning

UML är en grafisk notation för utformning och beskrivning av objektorienterade system. Akronymen står för Unified Modeling Language. Notationen skapades av Booch, Rumbaugh och Jacobson i slutet av 1990-talet genom att de samordnade sina respektive notationer. UML har snabbt blivit en standard och det finns många böcker och modelleringsprogram som använder den.

Denna uppsats beskriver den triviala delen av UML-användning, syntaxen för notationen, dvs reglerna för hur UML-diagram får ritas. Vi ger också mening, semantik, åt notationen genom att relatera till Java-implementeringar. Vi kommer inte alls att diskutera relationen mellan verkligheten och UML-modellen och vad som är bra eller dålig design. Det finns ett tiotal olika sorters diagram i UML. Vi skall bara behandla några.

Den formella specifikationen av UML [2] omfattar över 1000 sidor. Specifikationen erbjuder många fler möjligheter och större friheter än vad som presenteras här. I vissa detaljer är specifikationen entydig medan den i andra lämnar stort utrymme för olika tolkningar. Vid bedömningen av lösningar till problem givna i kursen är det denna skrift som utgör norm.

2 Klassdiagram

Ett klassdiagram beskriver den statiska strukturen för modellen eller det program som representerar modellen, dvs vilka klasser som finns och hur dessa är relaterade. Detta är information som är lätt att extrahera ur programkoden och många modelleringsprogram kan ta källkod som indata och producera ett klassdiagram. De kan ofta också gå den andra vägen och producera källkod från ett klassdigram. Ett klassdiagrammet innehåller inte all information som finns i programmet. Implementering av operationer (metoder) ingår normalt inte klassdiagrammet. Omvänt kan klassdiagrammet innehålla information som inte alltid går att utläsa ur programmet. Det kan handla om modellerarens intentioner beträffande relationer mellan klasser.

Huvudkomponenterna i ett klassdiagram är klasser, gränssnitt (interface) och flera sorters relationer.

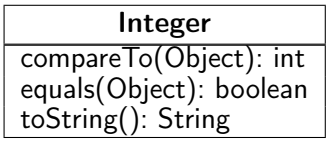
2.1 Stereotyper

Strängar omgivna av ”franska citationstecken”, «stereotype», kallas för stereotyper och används för att förtydliga användningen av en klass eller relation. En del stereotyper har givna betydelser i UML, som t ex «interface», «import» och «access», medan andra får introduceras fritt.

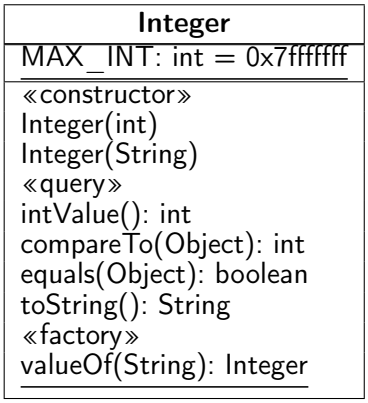
2.2 Klasser

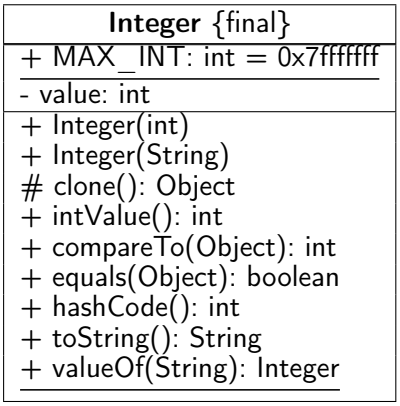
En klass representeras med en rektangel som innehåller en till tre avdelningar separerade av horisontella linjer. Den första avdelningen innehåller klassnamnet, den andra klassens attribut och en tredje dess metoder. Avdelningarna med attribut och metoder behöver inte innehålla alla attribut och metoder och får utelämnas helt.

Klassen Integer i `java.lang` kan beskrivas, med ökande detaljrikedom, på följande sätt.



Notera att returtyper anges efter kolon. Det är tillåtet att utelämna parametertyper och returtyper under förutsättning att detta göres konsekvent inom ett diagram. I nästa diagram visas att statiska attribut och metoder markeras med understrykning och att attribut kan ges initialvärden. Man kan använda stereotyper som rubriker i metodlistor: «constructor», «query» och «update».



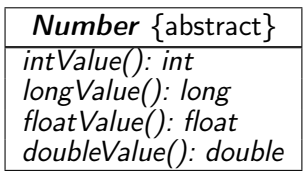


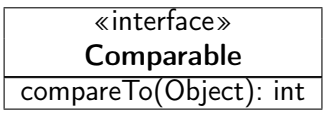
Det sista diagrammet visar synlighet för namnen med + för `public`, # för `protected` och - för `private`. Paketsynlighet anges med ~. Ytterligare egenskaper för klassen, attributen eller metoderna, som t ex `final`, kan noteras inom klamrar.

2.3 Abstrakta klasser och gränssnitt

Namn på abstrakta klasser och metoder markeras med kursiv stil eller när detta är otydligt med {abstract}.

Ett gränssnitt (interface) ritas också som en rektangel med stereotypen «interface» ovanför namnet på gränssnittet. Gränssnitt har ingen avdelning för attribut, men väl för metoder.





2.4 Relationer

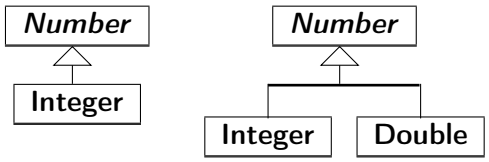
Det finns fem sorters relationer mellan klasser:

- En *generalisering* beskriver att en klass är en subklass till en annan.
- En *realisering* beskriver att en klass implementerar ett gränssnitt (interface).
- En *association* beskriver att en klass har ett attribut av en annan klass eller omvänt.
- En *inkapsling* beskriver en situation där en klass är deklarerad inuti en annan för att göra den osynlig för andra klasser.
- Ett *beroende* beskriver en relation mellan två klasser av annat slag än de ovanstående som medför att den beroende klassen kan behöva modifieras om den andra klassen förändras.

I ett klassdiagram visas relationer som linjer eller kurvor som förbinder klasser. Olika sorters associationer använder dekorationer som pilar, stereotyper och streckning. Att det finns en relation mellan två klasser syns i Java-koden genom att klassnamnet för den ena klassen används i den andra. Detta innebär att den senare klassen inte kan kompileras utan att den förstnämnda finns tillgänglig.

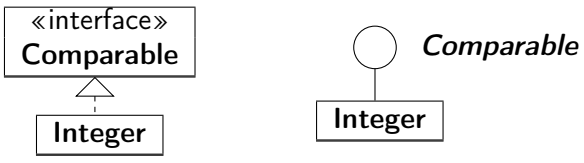
2.5 Generalisering

När en Java-klass utvidgar eller ärver en annan klass ritas detta alltid som en *generalisering* med en ofylld triangelpil mot superklassen. Om en klass har flera subklasser låter man dem ofta dela på pilen.



2.6 Realisering

En Java-klass som implementerar ett gränssnitt ritas med en streckad pil med en ofylld triangel mot gränssnittet. En förenklad variant ersätter gränssnittsrutan med en liten cirkel med namnet på gränssnittet vid sidan.



Martin [1] ritar ibland, i strid med UML, heldragna i stället för streckade pilar vid realisering.

2.7 Parametriserade klasser

I vissa objektorienterade språk kan man definiera klasser som har klassnamn som formella parametrar. I Java kallas detta för *generiska* klasser och i C++ för ”*template*”. Detta innebär att man kan definiera en List-klass med en parameter som talar om vilken typ elementen har. När man instansierar en sådan klass i programmet anger man den aktuella elementtypen som argument. I UML beskrivs denna konstruktion och instansiering med

I Java version 1.5 har Collection-klasserna i `jav.util` blivit generiska. Man kan t ex deklarera och skapa en lista som bara kan ha strängar som element:

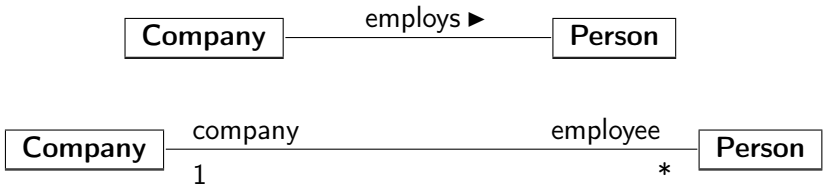
```
List<String> list = new ArrayList<String>();
```

2.8 Associationer

När en klass innehåller ett attribut av en annan klass finns det en tydlig relation mellan klasserna som kallas för *association*. I sådana sammanhang är det vanligt att inte visa attributet i klassrutan utan istället rita en linje mellan klasserna. När det i denna kurs föreskrivs att man skall konstruera ett klassdiagram med associationer betyder detta alla attribut som inte är primitiva typer eller har typen String skall ritas som associationer och inte visas i attributrutan.

Normalt sätter man ett namn på associationen ovanför linjen som klargör relationen. Man kan tillfoga en fylld triangel vid namnet som visar i vilken riktning namnet skall tolkas i förhållande till klasserna. En sådan triangel påverkar inte implementeringen.

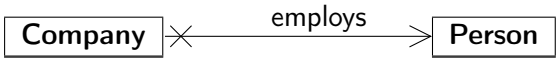
Vid associationens ändar kan man ange ”roller”, som ofta överensstämmer med attributnamn i programmet, och multipliciteter, som anger hur många instanser av den ena klassen som finns i en instans av den andra. Multipliciteten är ett heltal, som 1 eller 2, eller ett intervall, som 0..2. Man använder symbolen * för ett obegränsat antal, ensamt eller i intervall som 1..*. Det är tillåtet, men sällan meningsfullt, att kombinera flera multipliciteter med kommatecken. Rollnamn och multipliciteter som används i en klass skrives närmast den andra klassen och där är placeringen fri (över/under etc).



I programmet kan man finna

```
class Company {
    List employee = new ArrayList();
}
class Person {
    Company company;
}
```

Man kan tydligare visa vilka klasser i en association som har attribut av den andra klassen genom att sätta ut *navigeringspilar* och *navigeringskryss*. I figuren nedan har Company-klassen ett Person-attribut medan Person-klassen saknar Company-attribut.

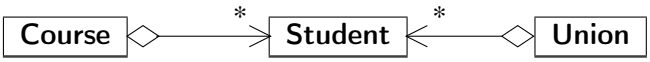


Om de finns attribut på ömse sidor ritar man pilar åt båda hållen.

Om man använder någon navigeringspil i ett diagram så tolkas alla associationer utan navigeringspilar som dubbelriktade. I ett diagram utan navigeringspilar har man inte specifierat hur man kan navigera.

2.9 Aggregering och sammansättning

Om man vill göra tydligt att ett objekt utgör en del av eller innehåller ett annat objekt dekorerar man associationen på ägarsidan med en ofylld romb. Detta kallas *aggregering*. Ett objekt kan vara aggregerat till flera andra objekt. I exemplet har en kurs, Course, referenser till 0 eller flera studenter och en studentkår, Union, referenser till studenter.



Ibland är associationen starkare så att ägarobjektet inte är meningsfullt utan sina delar och varje del har exakt en ägare. Då fyller man romben och kallar det för en *sammansättning* (composition).

Ibland är associationen starkare så att ägarobjektet inte är meningsfullt utan

För en sammansättning är det normalt ägarobjektet som skapar delobjekten och ansvarar för att de försvinner när ägarobjektet tas bort. I språk med automatisk minnesåtervinning (garbage collection), som Java, återvinnes orefererat minne av exekveringssystemet när behov uppstår.

2.10 Inkapsling

I Java kan man deklarera en klass inuti en annan. Anledningen kan vara att man vill ge objekt av den inre klassen direkt tillgång till alla attribut och metoder, även privata sådana, i den omgivande klassen.

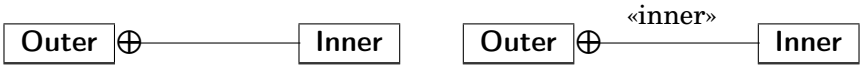
```
class Outer {
    private int n;
    class Inner {
    ...   n = 1;  ...
    }
}
```

I denna typ av inkapsling (inner class) är det bara ett Outer-objekt som kan skapa ett Inner-objekt och detta får alltså tillgång till de privata delarna av sin skapare.

En annan anledning att nästla klassser är att man vill göra en klass osynlig utanför den omgivande klassen, men att den i övrigt inte skall ha några särskilda privilegier beträffande instanser av den omgivande klassen. I detta fall skall den nästlade klassen vara `static` (static nested class).

```
class Outer {
    private static class Inner {
    ...
    }
}
```

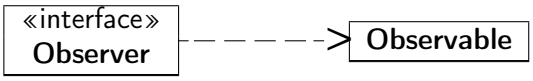
Ofta används nästlade klasser för att representera den yttre klassens interna tillstånd och bör därför vara privata. I UML används följande notation för en sådan privatisering. Om man vill göra det tydligt vilken form av inkapsling det handlar om kan man själv introducera en stereotyp.



2.11 Beroenden

Om en klass har en metod med någon parameter eller returvärde av en annan klass utan att ha några attribut av denna klass ritar man relationen med en streckad *beroendepil* med pilen mot den andra klassen. I följande exempel har gränssnittet Observer en metod.

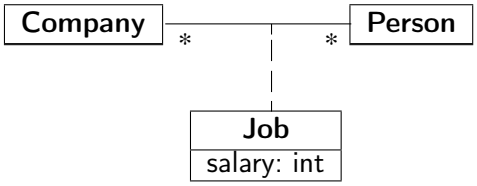
```
interface Observer {
    Object update(Observable observable, Object object);
}
```



2.12 Associationsklass

Det finns en hybrid mellan klass och association, en *associationsklass*, som enligt UML-specifikationen är en klass med associationsegenskaper eller omvänt. En associationsklass ritas som en streckad linje som ansluter mitt på en association och slutar med klassruta.

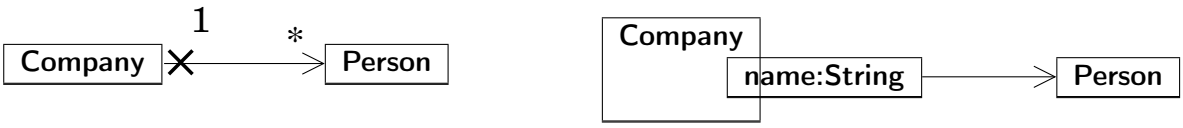
Intentionen är att till varje par av associerade objekt finns exakt ett objekt som kan innehålla attribut och metoder i vanlig ordning. Standardexemplet visar en association mellan Company och Person där varje anställningsrelation är kopplad till ett objekt som innehåller uppgift om t ex lön. En person kan vara anställd vid flera företag men ha högst en anställning hos ett företag.



Det finns ingen direkt motsvarighet till en associationsklass i Java.

2.13 Kvalificerad association

I den första figuren nedan kan ett Company-objekt referera till många Person-objekt. För att komma åt en person med givet namn måste man itererar över personerna tills man hittar rätt objekt. Om vi kan använda ett index eller en nyckel för att direkt hitta rätt objekt ritar vi detta som en *kvalificerad association* som i figuren till höger. Notera att multipliciteten på personklassen nu bör vara 0 eller 1; för ett givet namn bör det finns högst en person.



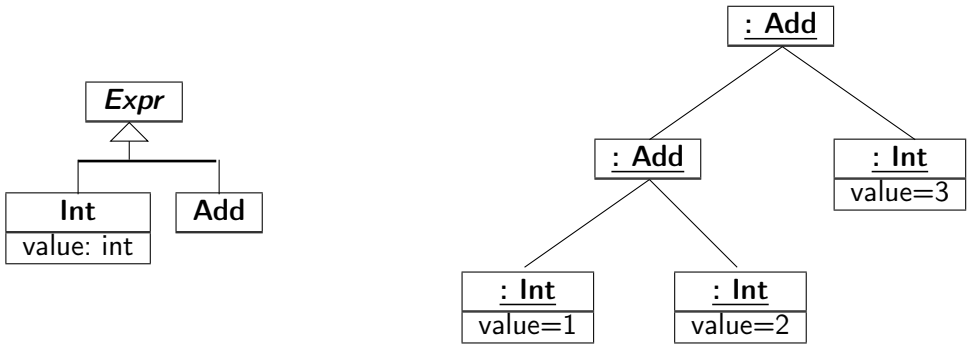
I implementeringen av Company använder vi kanske en hashtabell eller en databas med personnamnet som nyckel.

```
class Company {
    Map employees = new HashMap();
}
```

2.14 Objekt

För att illustrera hur en datastruktur ser ut i minnet vid ett visst tillfälle under exekveringen kan man rita ett *objektdiagram* med objektrutor analogt med klassrutor. Namnavdelningen skall innehålla ett eventuellt objektnamn, ett kolon och ett klassnamn och allt skall vara understruket. Aktuella värden på attribut kan uppvisas.

Enkla aritmetiska uttryck kan modelleras enligt klassdiagrammet till vänster. Den datastruktur som representerar uttrycket $(1+2)+3$ visas till höger:

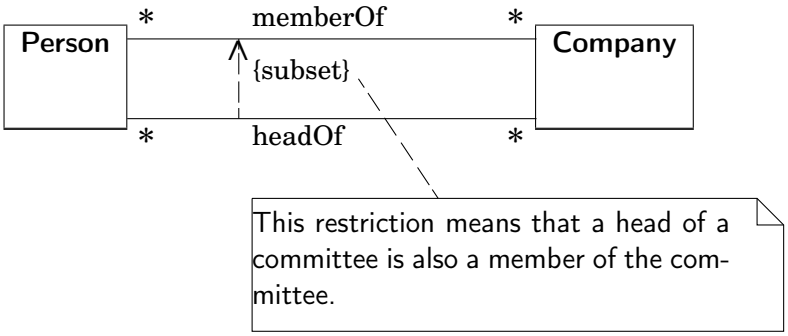


Linjerna mellan objekten kallas *länkar* (link) och får dekoreras med navigationspilar och aggregeringssymboler.

2.15 Anteckningar och restriktioner

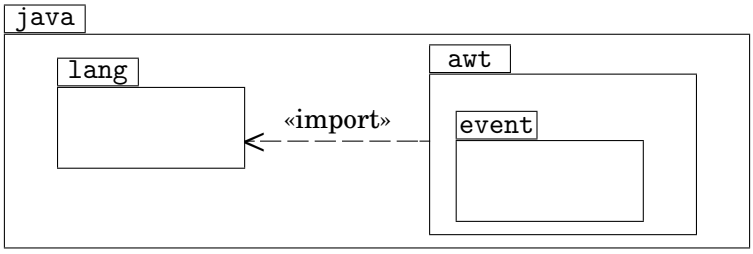
Anteckningar (notes) är rutor som kan innehålla all slags textinformation, som t ex programkod och kommentarer. Rutans övre högra hörn skall vara ”vikt”. Anteckningsrutor förbinds med de komponenter i ett diagram som de berör.

En *restriktion* är ett logiskt uttryck omgivet av klamrar. En restriktion kan placeras på en streckad linje med eller utan pil mellan två element i ett diagram eller inuti en anteckning med förbindelse till relevanta element.



2.16 Paket

Relationer mellan paket kan beskrivas. En paketruta ritas med en ”tab”-ruta ovanför till vänster. Namnet på paketet skrivs i tabrutan. Om paketrutan är tom får namnet skrivas där.



Vanliga beroendepilar används för paket. De märks med «access» eller «import», där det senare betyder att paketet importeras så att klassnamn kan användas utan kvalifikation. Ett paket som är ett ramverk kan förses med stereotypen «framework».

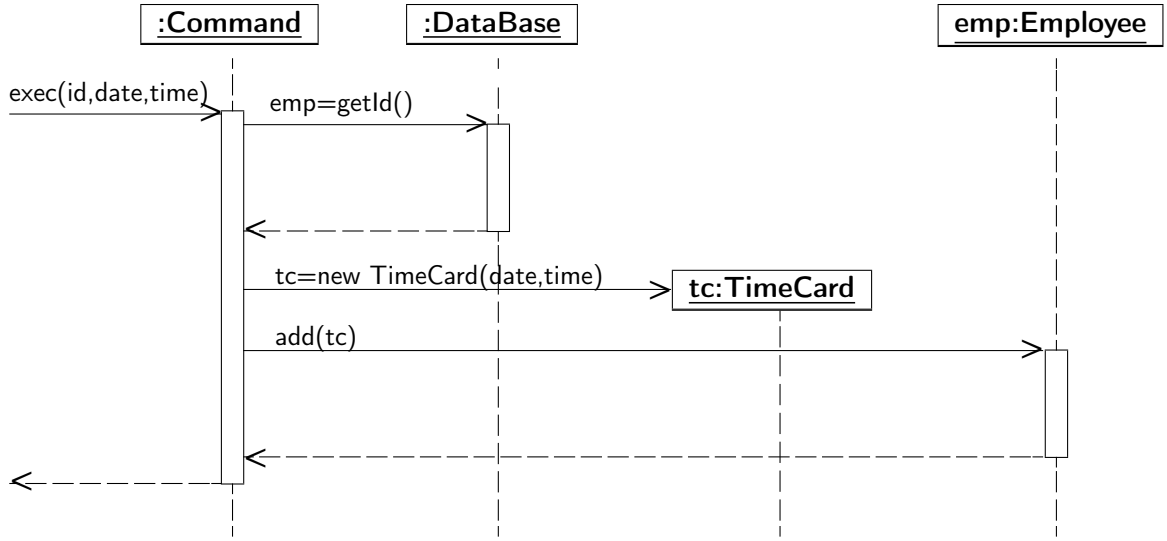
Det är tillåtet att rita klassrutor inuti en paketruta och förse dem med synlighetsattribut.

I stället för att rita ett paket inuti ett annat kan använda inneslutningssymbolen (⊗—).

Om man använder ett klassnamn från ett annat paket kvalificeras det på ”C++”-maner, `java::awt::Frame`.

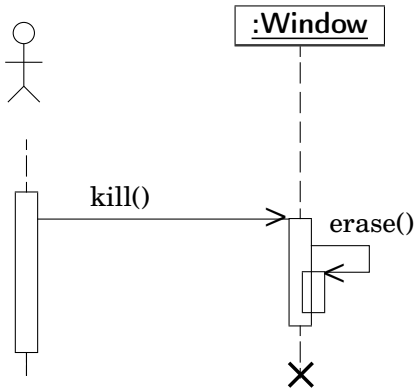
3 Sekvensdiagram

I ett sekvensdiagram visar man en dynamisk bild av en del av exekveringen av ett program. Det finns en tidsaxel som normalt tänks riktad nedåt. De aktuella objekten ordnas horisontellt. Varje objekt har en streckad vertikal livslinje som visar när det existerar. Ett objekt som skapas visas på den plats efter tidsaxeln där det skapas. Ett objekt aktiveras när någon metod i objektet anropas. Man ritar en pil med fylld spets från det aktiverande till det aktiverade objektet. Livslinjen övergår då i en långsmal rektangel som slutar vid metodaktiveringens upphörande. Återhoppet från metodaktiveringen kan makeras med en streckad pil med öppen pilspets. Återhoppspilar får utelämnas. Vid aktiveringspilarna får man visa metodanrop och hur returvärden tas om hand.



Vi ser att ett Timecard-objekt skapas under exekveringen av exec-metoden. De små pilar (○→) som Martin använder för parametrar och returvärden saknas i UML.

Ett metodanrop kan initieras av en människa eller ett program eller hårdvara utanför det system man modellerar. I sekvensdiagrammet ritas en sådan *aktör* som en streckgubbe. Om en metod aktiverar en annan metod i samma objekt ritas aktiveringsrektangeln delvis ovanpå den första aktiveringen. Om ett objekt upphör att existera avslutas livslinjen med ett stort kryss.

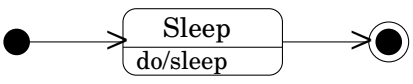


4 Tillståndsdiagram

4.1 Speciella tillstånd

`Starttillståndet` för ett system anges med en liten fylld cirkel. Ofta finns det en omedelbar övergång från starttillståndet till ett annat tillstånd.

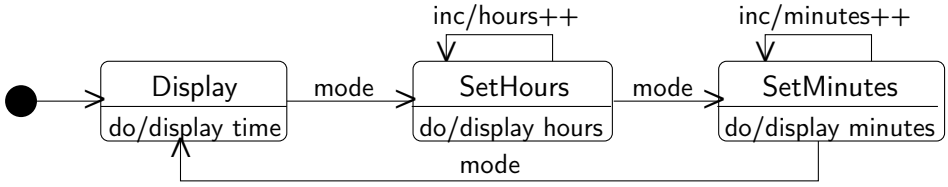
Ett system kan också ha ett sluttillstånd som ritas som en liten cirkel som innehåller en mindre fylld cirkel. Det får inte finnas några övergångar från ett sluttillstånd.



4.2 Tillståndsövergångar

Ibland uppvisar ett objekt olika beteende beroende på i vilket tillstånd det befinner sig. Följande exempel beskriver en digital klocka som har två knappar för att ställa in tiden. Med den ena knappen, `mode`, väljer man om klockan skall visa tiden, ge möjlighet att ändra timmar eller att ändra minuter och med den andra, `inc`, ökar man timvärdet eller minutvärdet. Klockan befinner sig alltså i ett av tre tillstånd, **Display**, **SetHours** och **SetMinutes**.

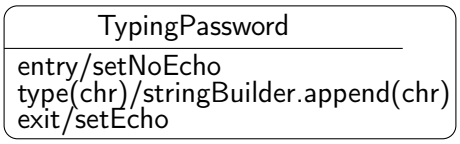
I ett *tillståndsdiagram* ritas *tillstånden* som rektanglar med rundade hörn och *övergångar* mellan dem som pilar.



Ett tillståndsdiagram beskriver en *tillståndsmaskin*.

4.3 Enkla tillstånd

Ett enkelt tillstånd har i regel ett namn som anges i den övre delen av rektangeln. Namnet bör indikera vad som försiggår när objektet befinner sig i tillståndet men vilka åtgärder som vidtages anges i den nedre delen av rektangeln med hjälp av den notation som beskrivs i nästa avsnitt.



4.4 Övergångssträngar

En *övergångssträng* har formen

event(*parameters*)[*guard*]/*action*

Här är *event* namnet på en händelse som kan ha parametrar, *guard* är ett logiskt villkor och *action* är åtgärd som skall utföras om händelsen inträffar och villkoret är sant. Om det finns flera parametrar separeras de med kommatecken.

En händelse är normalt en signal från omvärlden som skall påverka systemet, som t ex en knapptryckning eller en sensor som larmar. Händelsenamnet kan ha parametrar för att man bekvämt skall kunna hantera likartade händelser. Om man trycker på siffran *n* på en telefon kan detta representeras med händelsen *dial(n)*. Villkoret kan innehålla parametrarna och attribut i programmet. Åtgärden anges med ett beskrivande namn, ett anrop av en metod eller av programsatser.

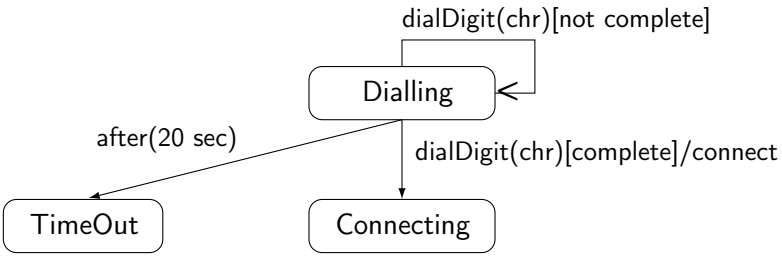
Både villkoret och åtgärden får utelämnas tillsammans med sina avgränsare. Utan villkor utförs åtgärden alltid om händelsen inträffar. Utan åtgärd sker bara en tillståndsövergång. Man kan också utelämna händelsen varvid villkoret testas hela tiden och åtgärden utföres när det är sant.

Man använder *after(time)* för att beteckna händelsen att man befunnit sig i ett tillstånd tiden *time* och *when(condition)* för händelsen att villkoret *condition* blir sant.

Det finns några fördefinierade interna händelser för varje tillstånd som bara får användas i den nedre halvan av en tillståndsrektangel. De är *entry* som inträffar när man kommer in i ett tillstånd, *exit* som inträffar när man lämnar ett tillstånd och *do* som används för att ange åtgärder som företas hela tiden man befinner sig i ett tillstånd.

4.5 Övergångar

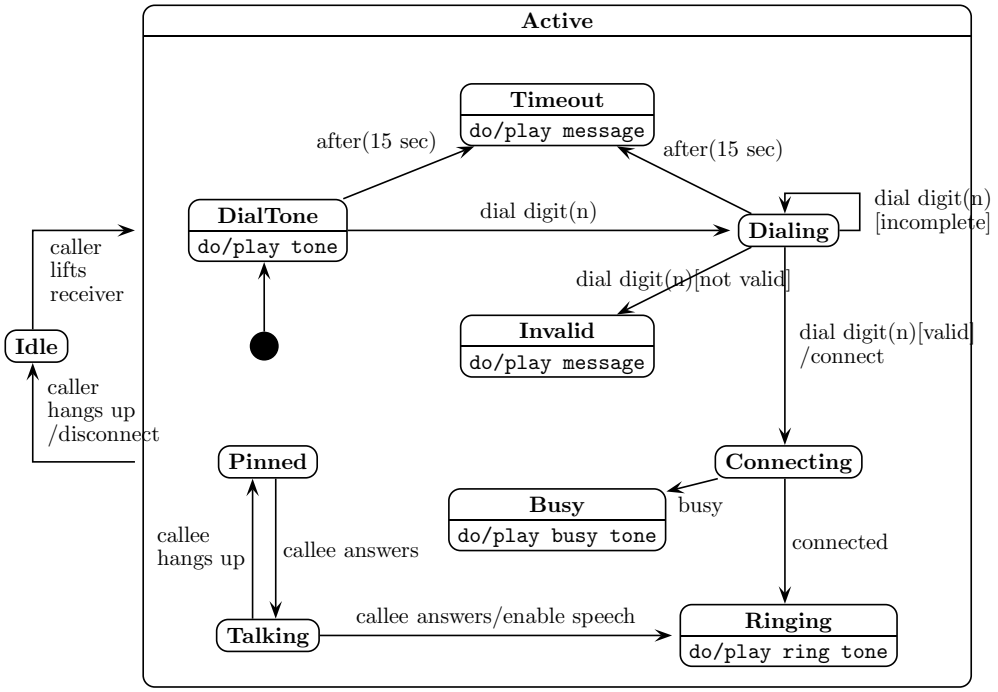
Externa händelser ger normalt upphov till tillståndsövergångar. En sådan övergång ritas med en pil mellan tillstånden tillsammans med en övergångssträng som anger när övergången skall ske och eventuell åtgärd som skall vidtas. Om det saknas en övergångssträng på en övergång innebär detta att övergången sker omedelbart.



Tillståndsdiagram skall vara deterministiska så att en händelse i ett givet tillstånd inte kan ge upphov till två olika övergångar. Olika händelser förutsätts ej inträffa ”samtidigt”.

4.6 Sammansatta tillstånd

Ett tillstånd kan innehålla ett tillståndsdiagram med starttillstånd, övergångar och sluttillstånd. Detta kan vara lämpligt för att strukturera ett stort diagram och visa strukturen på de sammansatta tillstånden separat. Det kan också förenkla diagrammet om många eller alla tillstånd har en övergång till ett tillstånd utanför det sammansatta. Detta utnyttjas i följande exempel. Om man befinner sig någonstans i det sammansatta tillståndet **Active** och uppringaren lägger på luren så hamnar man i tillståndet **Idle**.



5 Referenser

1. Martin, R.C., *Agile Software Development – Priciples, Patterns, and Practices*, Prentice-Hall, 2002.

2. Object Managment Group, *Unified Modeling Language (UML), version 1.5*, `http://www.uml.org/`.