1.2 Olika paradigm inom programmering

Vad är ett paradigm? T.ex. att koda på ett sätt som kan återanvändas även i andra program, är ett paradigm eller åtminstone en del av ett paradigm. Att inte göra så tillhör ett annat paradigm. Fri marknadsekonomi är ett paradigm inom ekonomi, statligt styrd ekonomi ett annat. Man kan säga, ett paradigm är en samling av regler, rekommendationer, normer, konventioner, mönster, standards, metoder och teorier inom ett ämne, som delas och följs av de flesta inom ämnet under en viss tidsperiod. Ett paradigm ger en orientering som styr handlingen och föreligger därför före erfarenheten (a priori), likt en fördom. Efter erfarenheten jämförs och bedöms handlingen med paradigmet (a posteriori), likt en lärdom. Överensstämmer resultatet inte med paradigmet, kan paradigmet åtminstone delvis ifrågasättas. Ofta leder ämnets progression efter längre tidsperioder till byten av paradigm, s.k. *paradigmskiften*, förutsatt att ett nytt paradigm har ställts upp som bättre uppfyller de önskade kraven. I programmeringens historia är vi vittnen för många sådana paradigmskiften, se *Paradigmskifte* (sid 16).

Maskinorienterad programmering

Även kallad maskin*nära* programmierung, vilket innebär att man skriver instruktioner som enkelt och snabbt, ja nästan direkt kan utföras av datorns processor (CPU). Maskinorienterade programmeringsspråk ligger allra närmast hårdvaran. Ursprungligen kan sådana maskinorienterade instruktioner endast utföras på en konkret maskin, eftersom de är definierade just för den aktuella hårdvaran. Ett typiskt exempel för ett sådant språk är *Assembler* som fortfarande är läsbar källkod som omvandlas till maskinkodens ettor och nollor av ett speciellt program som heter *assemblator*. Själva översättningsprocessen kallas för *assemblering*. Fördelen med maskinnära språk är den enkla och därmed snabba åtkomsten till hårdvaran, vilket kan vara avgörande i vissa sammanhang, t.ex. för spelkonsoler. Nackdelen är den svårt läsbara och icke-portabla koden.

Deklarativ programmering

Innebär att man anger *vad* som ska göras, inte *hur* det ska gå till. Man nöjer sig med att säga vad man vill ha. Tillvägagångssättet tas hand om av programmeringsspråket. Ett typiskt exempel för ett sådant språk är *SQL* som står för *Structured Query Language* och är standardspråket för kommunikation med databaser. Med en SQL-sats ställer man en fråga till en databas. Man får som svar den datamängd som är efterfrågad i SQL-satsen. Hur SQL letar efter och hittar denna datamängd i den väldigt komplexa databasen, behöver programmeraren inte bry sig om. Man *deklarerar* endast sitt önskemål, precis som man beställer en maträtt på en retaurang. Deklarativ programmierung har många underkategorier.

Funktionell programmering

En typ av deklarativ programmierung är *funktionell programmering*. I detta paradigm består ett program av en samling matematiska funktioner som definieras och exekveras direkt med minsta möjliga tidsåtgång (runtime). Man undviker kod som

anses vara onödig overhead och fokuserar på effektivitet och funktionalitet hos de mest små moduler utan att man behöver ange i vilken ordning de ska exekveras. Ett typiskt funktionellt språk – dessutom det äldsta – är *Lisp*. I Visual Studio finns även ett funktionelltt språk som heter *F#*. Historiskt har funktionell programmering sitt ursprung i ett matematiskt forskningsprojekt på 30-talet som resulterade i den s.k. *Lambdakalkylen*. I C# har man tagit över dessa tankar genom att integrera *Lambdauttryck* i språket, vilket behandlades i Programmering 2.

Logikprogrammering

En annan typ av deklarativ programmierung är *logikprogrammering* som baseras på matematisk logik. Ett logikprogram består inte av ett antal instruktioner utan av en mängd *axiomer* som kan anses vara en samling definitioner och regler som alla program måste följa. All kod som skrivs kommer att exekveras endast enligt dessa axiomer. Man ställer en fråga och får svaret som en logisk slutsats ur axiomsystemet. Logikprogrammering har sitt ursprung i 70-talets forskningsaktiviteter kring *artificiell intelligens*. Det mest kända logikprogrammet är *Prolog*.

Händelsestyrd programmering

Detta paradigm är typiskt för grafiska applikationer *(GUI)*. Programkörningen är inte längre till 100% förbestämd av utvecklarens kod utan kan även styras – åtminstone delvis – av användaren under programkörningen genom musklickningar och tangenttryckningar, s.k. *händelser*. Även andra typer av händelser är tänkbara som påverkar både programförloppet och avslutningen i en mycket större utsträckning än det är fallet med rena textbaserade program. Exekveringen startar ofta i ett fönster med grafiska komponenter, som visas när programmet körs. Efter en händelse återgår kontrollen till operativsystemet, vilket dock inte betyder att körningen är avslutad, utan att programmet är redo att ta emot nästa händelse osv. *Händelsestyrd programmering* används bl.a. i Windowsprogrammering och är implementerad t.ex. i *C# Windows Forms Applications* med sin stora verktygslåda av förprogrammerade grafiska komponenter som *Controls* osv.

Spaghettiprogrammering

Självklart finns det inte ett uttalat paradigm som heter så. Det är snarare ett smeknamn som man ur ett historiskt och kritiskt perspektiv gett denna typ av programmeringsvana som man använt i de äldre språken i brist på bättre lösningar.

Så länge det inte fanns kontrollstrukturer använde man sig av s.k. *hoppsatser* för att åstadkomma loopar. Det reserverade ordet **`goto`** skickar programflödet till ett annat ställe i koden vilket man markerar med en *Label*, t.ex. med **`L`**. En label är ingen variabel utan en symbol som endast markerar ett ställe i koden. Den används i **`goto`**-satsen för att skicka programflödet till det markerade stället. Ironiskt nog finns det reserverade ordet **`goto`** fortfarande i C#. T.e.x. kan det se ut så här:

```
L: Console.Write("\n\tGissa ett tal mellan 1 och 20:\t");
   guessedNo = int.Parse(Console.ReadLine());
                              . . .
   if (guessedNo != 17) goto L;
```

Om det gissade talet *inte är 17* dvs om användaren *gissat fel*, ska programmet hoppa till **`L`** där användaren ges åter möjligheten att göra en ny gissning som sedan prövas, osv. Om däremot det gissade talet *är 17*, dvs om användaren *gissat rätt*, äger hoppet inte rum. Man har med en **`if`**-sats, som är en enkel *selektion*, i kombination med **`goto`** lyckats konstruera en loop.

Varför kallar vi det för spaghettiprogrammering när koden ovan fungerar? Anledningen är att hoppsatser leder i större program till förvirring. Föreställ dig att man har ett stort program, använder väldigt många **`goto`**-satser och utnyttjar fullt ut friheten att placera våra labels var som helst. Resultatet blir en kod som är svårt att kontrollera, uppdatera och underhålla. Programflödet liknar till sist en spaghettirätt. Sådana program uppfyller inte längre de krav om läslighet, förståelighet och ändringsbarhet. Det märks speciellt när en programmerare byter jobb och en efterträdare ska vidareutveckla programvaran. Ofta blir det helt omöjligt för efterträdaren att sätta sig in i koden. Redan på 60-talet ledde detta till en programvarukris och initierade utvecklingen av procedurala programmeringsspråk som Algol, Simula, Pascal, C, … där **`goto`**-satser kan och bör undvikas. Procedural programmering bannlyser användningen av **`goto`**-satser då en okontrollerad användning av hoppsatser i större program leder till spaghettiprogram som inte längre är lästliga, förståeliga och ändringsbara. Procedural programmering ersätter alla hoppsatser med kontrollstrukturer där det inte längre finns några labels då dessa är hårdkodade och placerade på fasta platser. Man borde ersätta **`goto`**-satsen med en kontrollstruktur av typ repetition, t.ex. en **`while`**-sats.

Procedural programmering

Motsatsen till deklarativ programmierung är *imperativ programmering*. Procedural programmering är den äldsta typen av imperativ programmering. Här anger man inte bara *vad* som ska göras, utan även – och framför allt – *hur* det ska gå till. Tillvägagångssättet är en väsentlig del av imperativa språk. Ett tillvägagångssätt som exakt och entydigt *beskriver* hur man löser ett problem, kallas för *algoritm.* Man kan *beskriva* en algoritm på många olika sätt, t.ex. på vanligt språk, med hjälp av grafik, med pseudokod, i form av ett flödesschema osv. Väljer man programkod för att beskriva algoritmen, har man ett datorprogram. Ofta måste även viss *data* (t.ex. indata) läggas till för att lösa problemet. Därför kan man säga:

Program = algoritm + data

Det var Niklaus Wirth, skaparen av programspråket *Pascal*, som på 60-talet ställde upp denna definition. Data är information i organiserad, strukturerad form. Men

vad exakt är en algoritm, och framför allt hur kan algoritmer *beskrivas*? Dessa frågor kommer vi att ägna oss åt i resten av det här kapitlet. Wirths definition återspeglar en algoritmorienterad syn på programmering som även kallas *procedural programmering*. Procedur är ett annat ord för algoritm. Modern till alla procedurala språk är *Algol*.

Objektorienterad programmering (OOP)

Om man i Wirths definition *Program = algoritm + data* lägger betoningen på data istället för på algoritmen och inte längre betraktar data som ett slags bihang till algoritmen utan som *objekt* kommer man till *objektorienterad programmering*. Den nya definition som kom upp på 80-talet och återspeglar den objektorienterade synen på programmering är:

Program = Modell av verkligheten

OOP syftar åt att efterlikna verkligheten. Man vill avbilda den reala världen – åtminstone den del som tillåter datorisering – och konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten genom att testa modellen. För att undvika filosofiska diskussioner kan vi anta att den reala världen består kort sagt av *objekt*. Världen kring oss är full med objekt: Människor, byggnader, bilar, tåg, flygplan, träd, möbler, böcker, butiker, skolor, bibliotek, kontor, anställda, kunder, varor, fakturor, order, bokningar, kurser osv. Objekten kan vara verkliga eller virtuella. Ett datorprogram försöker att beskriva dessa objekt. Beskrivningen kodas i *klasser*.

Ett objekt kan i regel utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas de för *metoder* – samma som i den procedurala programmeringen heter funktioner. En metod är en funktion som definieras i en klass. Namnbytet beror på att man i OOP måste definiera sina funktioner i klasser, därför att metoderna i regel ska vara bundna till objekt. Förenklat kan man säga: när ett objektorienterat program körs anropar metoder varandra och skickar därvid objekt till varandra. På så sätt simuleras verkligheten.

Paradigmskifte

Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.