HUR TILLÄMPA OBJEKTORIENTERING RÄTT

JUHA METSÄKALLAS

Naturkandidatavhandling i datavetenskap

Handledare Marina Waldén

Åbo Akademi

3.4.2011

Referat

I denna kandidatavhandling beskriver jag hur objektorientering blev till och vad objektorientering egentligen innebär. Jag presenterar utvecklingen av abstraktioner i programmeringsspråk, vad dessa abstraktioner betyder och vad de har med objektorienteringen att göra.

Även om sätt hur abstraktioner och begrepp förverkligas i olika programmeringsspråk skiljer sig från språk till språk – språken har ofta även olika termer för samma begrepp – ska ett språk ha vissa egenskaper för att kunna anses vara objektorienterat. Dessa egenskaper är inkapsling av data, abstrakta datatyper, ärvning och dynamisk bindning. Jag undersöker dessa egenskaper och visar hur de har förverkligats i några programmeringsspråk. Jag tar upp litet även sådana fenomen i språken som egentligen inte har med objektorienteringen att göra men som upplevs som tillhöra den.

Objektorienteringen är dock ingen patentlösning i alla situationer. Jag tar upp både situationer när den kan tillämpas och när inte.

Avslutningsvis diskuterar jag kort vad som har kommit efter objekt-orienteringen.

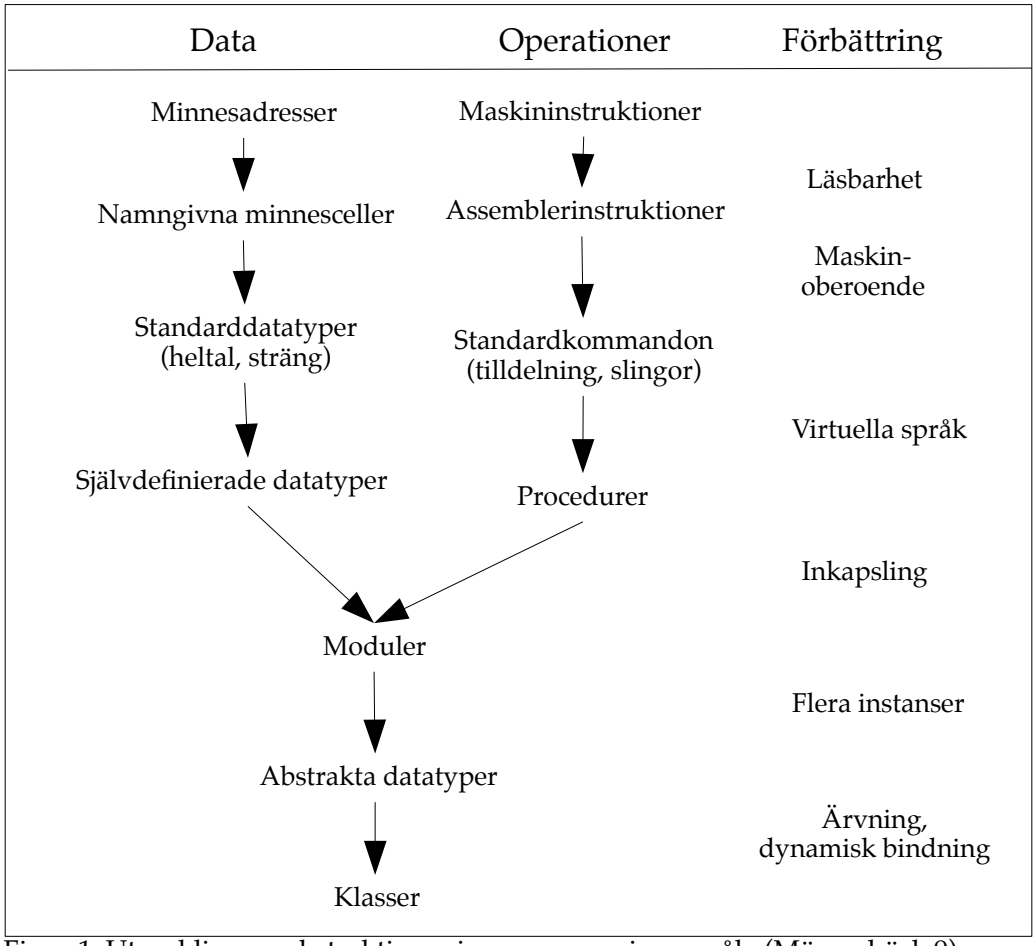
Resumo

Objektoj en...

Innehållsförteckning

1 Inledning

Utvecklingen av programmeringsspråk kan enligt (Mössenböck 9) ses som en serie av abstraktioner som fört programmeringen allt längre bort från minnesceller och maskininstruktioner. Figur 1 nedan visar dessa abstraktioner och de förbättringar som uppnåtts.



Figur 1: Utveckling av abstraktioner i programmeringsspråk. (Mössenböck 9)

Jag kommer inte att behandla de abstraktioner som ägt rum före inkapsling utan jag utgår från den, fortsätter via abstrakta datatyper till ärvning och dynamisk bindning. Dessa utgör tillsammans det som kallas objekt-orientering. Jag presenterar dessa begrepp och belyser dem med enkla kod-exempel. Denna genomgång omfattar kapitel 2 i sin helhet.

Det finns dock flera sätt att se på objektorienteringen och i kapitel 3 tar jag upp alternativ till det vanligaste sättet, den klassbaserade objekt-orienteringen. De alternativ som jag visar baserar sig på prototyper och delegater. Alternativen behandlar jag endast till de delar de skiljer sig från den klassbaserade objektorienteringen.

I kapitel 4 går jag igenom några fall i mjukvaruutveckling där objekt-orienteringen kan utnyttjas med god effekt. Jag tar också upp några typiska misstag i tillämpning av objektorientering vilka är ofta resultat av ytlig syn på objektorientering och dess egentliga innebörd.

Kodexemplen i avhandlingen är skrivna på programmeringsspråken (i alfabetisk ordning) Java, Javascript, Modula-2, Oberon-2, Python och Realbasic. I exemplen antar jag att språket har fullt stöd för Unicode för identifierare. Läsare med kunskap om ett par högnivåspråk borde kunna förstå dem utan större svårigheter.

2 Vad objektorientering innebär

Fast objektorienteringen härstammar från 1960-talet blev den mera känd först på slutet av 1980-talet, i början av 1990-talet. Den infördes som ny programmeringsparadigm som sades lösa alla problem i mjukvaru-utvecklingen. Objektorienteringen kom dock inte ur intet utan de första stegen togs redan på 1960-talet. Den är mera en samling av olika abstraktioner och begrepp i olika programmeringsspråk som anses komplettera varandra än en viss särskild teknik.

2.1 Inkapsling och abstrakta datatyper

I utvecklingen av programmeringsspråk utgjorde de virtuella språken en behövlig förutsättning för objektorienteringen. Virtuella språk möjliggjorde självdefinierade datatyper och procedurer[1] (se figur 1). Genom att utnyttja denna abstraktion blev det möjligt att kapsla in data och operationer i logiska enheter, kallade moduler, så att de kunde utnyttjas endast på ett visst sätt. Grundtanken är att begränsa åtkomsten till den interna datastrukturen med den fördelen att den kan ändras utan att datatypens gränssnitt förändras, d.v.s. utåt syns inga ändringar.

Det bör noteras att modulerna var till början sådana att man kunde ha endast en variabel med en viss inkapslad datastruktur. Följande logiskt steg i utvecklingen blev då att kunna ha flera variabler med den samma interna datastrukturen. Begreppet abstrakta datatyper introducerades i s.k. modulära programmeringsspråk och det innebar att den inkapslade datastrukturen utgör en datatyp varav det kan finnas flera variabler (se figur 1). Dessa typ-inkapslande enheter kallades t.ex. *package* i Ada och *module* i Modula-2.

Ta en komplex datastruktur, t.ex. en prioritetskö, som exempel. Den förverkligas effektivast med hjälp av ett partiellt ordnat vänsterbalanserat träd, en *heap*, (för närmare beskrivning av strukturen och dess operation se t.ex. (“Heap (datastruktur) - Wikipedia”). I Modula-2 kan definitionen för en prioritetskö se ut såsom i figur 2 (för abstrakta datatyper i Modula-2 se t.ex. (Ford 172-173)) där `kö` utgör en abstrakt datatyp varav man bara avslöjat att en procedur `lägg` accepterar en parameter av den typen. Vidare notera att elementen i kön är av typ `elementtyp` som är också en abstrakt datatyp. Den importeras från en annan modul.

1 I avhandlingen använder jag ordet procedur för all alla underrutiner d.v.s. både för funktioner och egentliga procedurer.

```
DEFINITION MODULE prioritetskö;
FROM element IMPORT elementtyp;
EXPORT QUALIFIED
   (* typ *) kö,
   (* procedur *) lägg;
   TYPE
      (* abstrakt datatyp *)
      kö;
   (* lägg till ett element in i en kö *)
   PROCEDURE lägg(VAR k: kö; VAR element: elementtyp);
END prioritetskö.
```

Figur 2: Definition av en prioritetskö som en abstrakt datatyp i Modula-2.

2.2 Klasser och metoder

Fastän abstrakta datatyper var ett steg som möjliggjorde byggande av stora datasystem – det är svårt att tänka sig stora program med bara en namnrymd; diskussion om namnrymder ligger utanför denna avhandling – skedde inkapslingen av procedurer i abstrakta datatyper dock endast indirekt: en procedur opererar nog med en datatyp men inte tillhör datatypen så att säga. Nästa abstraktion i utvecklingen av programmeringsspråk blev att tydliggöra detta samband.

2.2.1 Klass och typbundna procedurer

Klass är en abstraktion av en abstrakt datatyp med typbundna procedurer, metoder. Typbundenhet betyder att en metod inte kan existera utan ett

objekt. För att använda exemplet med en prioritetskö, är det fullt möjligt att anropa proceduren `lägg` i kodexempel 1 med en parameter av typ `kö` som inte existerar, d.v.s. är NIL (Null, None i andra språk). Med en typbunden procedur, en metod, är detta inte möjligt eftersom metoden inte finns om dess objekt inte finns.

I praktiken lagrar inte varje instans av en klass egen kopia av varje metod – metoden är ju fastställd (för undantag se kapitel 3) och kan inte ändras under exekveringen – utan instansen har en referens till klassen som har metodens kod. Denna referens behövs för polymorfism (se kapitel 2.5).

För att klargöra vilken klass en metod hör till behöver man i ett programmeringsspråk någon syntax för att indikera detta. I nästan alla språk visar man detta genom att deklarera (och förverkliga) metoder inom klassdefinitionen.

```
public class prioritetskö
{
   public void lägg(Element elem)
   {
      …
   } // lägg
} // class prioritetskö
```

Figur 3: Definition av en metod tillhörande en klass i Java.

Java-koden i figur 3 visar att metoden `lägg` tillhör en klass `prioritetskö` genom att metoden – både dess deklaration och implementering – är omsluten inne i blocket för klassdefinitionen, de yttre klammerparenteserna. Anrop av metoden sker genom s.k. punktnotation – `kö.lägg(elem)` där `kö` är en instans av klassen `prioritetskö` – så som i nästan alla objektorienterade språk.

2.2.2 *Klasser och moduler som skilda begrepp*

Man kan fråga sig huruvida det behövs både klasser och moduler: de båda kapslar in data och tillgängliggör det via metoder eller procedurer. Båda har sin plats. Moduler möjliggör procedurer som existerar utan instanser. Det klassiska exemplet är matematiska funktioner, t.ex. sin(x). Vad representerar det objekt som har en sinusfunktion som metod? Detta behov har lett till att språk som inte gör någon skillnad mellan klasser och moduler – t.ex. Java – har s.k. klassmetoder, även kallade statiska metoder för detta.

Det finns också fall där två eller flera klasser alltid används ihop: en instans `a` av en klass `A` har alltid en referens till en instans `b` av en klass `B`. Visserligen kan man gå den normala vägen d.v.s. `a` använder alltid åtkomstmetoder (engelska *mutator methods*) för att komma åt data i `b`, med av effektivitetsskäl vore det ofta bättre om `a` kom åt interna datastrukturer i `b` direkt (se kapitel 2.3). I språk som inte likställer moduler med klasser är detta inget problem: en Python fil kan ha flera klasser som enligt konventionen kommer åt varandras data direkt. I språk utan modulbegrepp måste man hitta på andra lösningar. T.ex. Java har löst detta med s.k. interna klasser medan C++ använder s.k. vänklasser (engelska *friend class*) för att tillåta åtkomsten till interna datastrukturer.

2.2.3 *Objektparameter*

En nära relaterad fråga i samband med metoder är hur man inom en metod vet vilket objekt som metoden har kapslats in i. Det är logiskt att överföra en referens till objektet som parameter i metodanropet. Alla objektorienterade programmeringsspråk gör så, men sättet hur överföring av denna objektparameter sker i praktiken skiljer sig från språk till språk. I vissa språk görs överföringen implicit, d.v.s. man ser inte ens parametern, medan

andra har olika slags konventioner angående var och hur man explicit kan överföra den. I det implicita fallet har parametern ett bestämt, speciellt namn, `me`, `self`, `this` eller `owner`. I det explicita fallet bör parametern oftast komma först och ha ett visst namn, t.ex. parametern `sender` alltid först i en metod i Object Pascal (känt även som Delphi Pascal), eller programmeraren ger explicit ett namn åt den såsom i Oberon-2. (Craig 129-130)

```
MODULE Prioritetskö;
TYPE
   Kö* = RECORD
            …
         END;
PROCEDURE (VAR kö: Kö)Lägg*(element: Element);
BEGIN
   …
END Lägg;

END Prioritetskö.
```

Figur 4: Namngivning av objektparametern i Oberon-2.

I figur 4 har programmeraren namngett objektparametern med `kö`[2] och det är detta namn som ska användas inom metoden. (Mera om programmeringsspråket Oberon-2 t.ex. i *Oberon-2 language report*, tillgänglig på webbsidan ("ETH - Oberon - Language Archives Oberon and Old Compilers")).

Observera att denna mottagarparameter spelar dubbel roll: den bestämmer vilken metod som ska anropas under exekveringen – mera om detta i kapitel 2.5 – och innehåller en referens till själva objektet så att man kommer åt dess data (Mössenböck 39-40).

2 Notera att Oberon-2 behandlar identifierare med olika skriftläge som olika. Konventionen är att inleda en datatyp med en stor bokstav medan variabler inleds med en liten bokstav.

2.3 Meddelanden

Det finns olika uppfattningar om vad som avses med meddelanden i objektorienterade programmeringsspråk. I en del objektorienterade språk används förmedling av meddelanden som synonym till metodanrop. I detta kapitel diskuterar jag meddelanden som överförda parametrar och bred- och flersändning av meddelanden. I kapitel 3.1 tar jag kort upp ännu ett sätt att se på meddelanden.

2.3.1 Meddelanden som överförda parametrar

Meddelanden kan användas som ett sätt att packa data till block och överföra dessa block som parametrar i metodanropen (Mössenböck 70-71) . Idén är att varje objekt har en metod, en meddelandehanterare, som mottar alla meddelanden och avgör hur objektet reagerar på dem.

```
TYPE
   Meddelande = RECORD END; (* bastyp för alla meddelanden *)
   Ritmeddelande = RECORD (Meddelande) END;
   Rotationsmeddelande = RECORD (Meddelande) rotation: REAL END;

   Figur = POINTER TO RECORD (* bastyp för alla figurer *)
              PROCEDURE (f: Figur) Hantera(VAR m: Meddelande);
           END;
```

```
TYPE
   Rektangel = POINTER TO RECORD (Figur) … END;
   PROCEDURE (r: Rektangel) Hantera(VAR m: Meddelande)
   BEGIN
      WITH
         m: Ritmeddelande DO… END
      | m: Rotationsmeddelande DO… END
      ELSE (* ignorera andra meddelanden *)
      END
   END Hantera;
```

Figur 5: Meddelande som parameter i Oberon-2. (Mössenböck 70-71)

I figur 5 utgör klassen `Meddelande` en överklass (mera om över- och underklasser i kapitel 2.4) till meddelande av typ `Ritmeddelande` och `Rotationsmeddelande`. Överklassen `Figur` definierar en (abstrakt) hanterare, `Hantera`, för alla meddelanden. Klassen `Rektangel` implementerar hanteraren (för namngivning av objektparametern i Oberon-2 se figur 4) och avgör hur instanserna av klassen reagerar på olika meddelanden[3].

Hanterarna kan i vissa fall förverkligas genom att överlagra eller omdefiniera metoder. Överlagring kan ändå inte helt ersätta en generell hanterare med en valsats i språk som inte tillåter en definition av en hanterare utan implementering. Att lämna utan implementering kan behövas i stora mjukvaruprojekt när det i ett senare skede av projektet visar sig att en ny typ av meddelande behövs för vissa objekt. Att skriva en tom hanterare för sådana objekt som inte behöver reagera på meddelandet kan vara för arbetsdrygt. Omdefiniering kan i sin tur resultera i att överklassen kommer att ha många tomma metoddefinitioner.

En nackdel med meddelanden som överförda parametrar är att det blir svårare att få klarhet i vilka meddelanden ett objekt reagerar på. När dessutom tolkning av meddelanden och läsning av parametrar i dem sker långsammare än med direkta metodanrop, lämpar sig meddelanden som

3 Satsen `WITH` motsvaras i flera språk av `switch/case` satsen.

överförda parametrar trots deras stor flexibilitet endast för bredsändning (se kapitel 2.3.2) och delegering (se kapitel 3.2) (Mössenböck 72-73) där ett metodbaserat angreppssätt i sin tur vore klumpigt.

2.3.2 Bred- och flersändning

Bred- och flersändning – på engelska *broadcast* respektive *multicast*, finska *yleis-* respektive *monilähetys* – betyder i samband med objektorienterad programmering att man sänder ett meddelande till alla respektive några objekt av en viss typ och dessa bestämmer vart för sig hur de reagerar på meddelandet. Även om bred- och flersändning associeras mestadels med händelser i användargränssnitt – t.ex. en musklick sänds till alla grafiska kontroller under pekaren – finns det i princip inget som förhindrar deras användning generellt.

Medan prototypbaserade objektorienterade programmeringsspråk har som regel inbyggda mekanismer för bred- och flersändning kan sådana förekomma även i klassbaserade språk.

```
Sub Display(p As ProcessResult)
   RaiseEvent Display p
End Sub

Event Display(p As ProcessResult)

Sub Display(p As ProcessResult)
   Dim c As New Clipboard
   c.Text = p.GetText
End Sub
```

Figur 6: Sändning av händelse i Realbasic. (Howe 77-78)

I figur 6 finns Realbasic-kod där en överklass definierar en metod, den första kodsnutten med namnet `Display`[4] som skapar, kastar en händelse av typen `Display` som sedan en underklass reagerar på med en meddelandehanterare `Display` (den tredje). ”Meddelandet” i exemplet är en instans av en klass `ProcessResult` med en metod `GetText`. Förutom fördelen att överklassen bättre kan styra programflöde med sändningarna tillåter den här sortens sändning också att sändaren inte behöver veta vilket objekt som reagerar på ett meddelande eller om ens något reagerar – se (Howe 79-80).

Vanligare är det dock att man måste i klassbaserade språk ofta nöja sig med att simulera bred- och flersändning. Denna simulering sker oftast genom att en behållarklass (engelska *container class*) erbjuder en speciell konstruktion, en iterator, med vars hjälp man kommer åt vart och ett objekt turvis i behållaren. (För mera information om iteratorer se t.ex. (Craig 44-49)).

2.4 Ärvning

Medan inkapsling och abstrakta datatyper förekommer i andra programmeringsspråk, ärvning är unikt för objektorienterade språk. Ärvning innebär att klasser bildar en hierarki och klasserna nedre i hierarkin kan utvidgas så att de bibehåller existerande data och/eller operationer och tillkommer nytt data och nya operationer eller omdefinierar de befintliga (Holm 19). Ärvning av data förekommer inte i klassbaserad objektorientering utan endast i prototypbaserad objektorientering (se kapitel 3.1). Ärvning av operationer, metoder, förekommer däremot i bägge angreppssätt.

Man kan tänka sig att genom att ärva implementering av metoder från flera klasser kan man lätt kombinera metoderna in i nya objekt. Om man tillåter att ärva från många klasser stöter man på s.k. diamantproblemet. Det betyder det att om klassen `A` definierar och kanske implementerar en metod

4 Det är bara en konvention i Realbasic att ge samma namn åt metoden och händelsen (de finns i separata namnrymder), något absolut krav finns inte.

`babbla` som två klasser, `B` och `C`, överlagrar och en fjärde klass `D` ärver både `B` och `C` utan att överlagra metoden, vilken version av metoden som ska köras när metoden anropas i `D`: `A`:s, `B`:s eller `C`:s? Språken som tillåter denna multipla ärvning måste har explicita regler hur bindning sker i sådana fall. Dessa regler blir ofta komplicerade och gör koden svårare att förstå. Därför tillåter man i flera språk bara enkel ärvning, d.v.s. en klass ärver implementering bara från en klass.

Det kan dock vara nyttigt att ett objekt kan behandlas som en instans av flera klasser. I stället att ärva implementeringen – se kapitel 2.2.1 – kan då en klass ärva gränssnittet från en annan klass. Ett annat alternativ till multipel ärvning är delegation som jag behandlar i kapitel 3.2.

(Mössenböck 63) framhäver att den största nyttan av ärvning inte kommer i form av återvunnen kod, att man inte behöver upprepa kod, utan i form av av återvunnet programmeringsgränssnitt där objekt i en underklass förstår samma metodanrop som objekten i överklassen. Ett klassiskt exempel på detta är en ström. En ström är en klass vars instanser skriver till eller läser från ett medium, en skärm, en fil eller en nätförbindelse. Genom att definiera strömmen som en abstrakt klass – en klass varav man inte kan skapa instanser – definierar man ett gränssnitt som alla instanser av underklasser förstår: öppna, skriva/läsa ett tecken o.s.v. Det är detta gemensamma gränssnitt som är viktigt, inte de interna datastrukturerna som är olika t.ex. en ström till en skärm har säkert en annan intern struktur än en annan ström till en fil.

2.5 Polymorfism och dynamisk bindning

Två begrepp som är relaterade till ärvning och sinsemellan är polymorfism, flerformighet, och dynamisk (metod)bindning. Man säger att en metod är polymorfisk om den förekommer i många versioner som skiljer sig

från varandra genom att acceptera parametrar av olika datatyper så att dessa typer bildar en arvhierarki. För att bestämma metoden och därmed det mottagande objektet måste språket ha entydiga regler för denna bestämning eftersom objekten kan utvidgas.

Den här bestämningsmekanismen kallas dynamisk bindning som i detta sammanhang betyder att det bestäms först under exekveringen vilket objekt som mottar ett anrop på metoden – eller ett meddelande till en hanterare (se kapitel 2.3.1) – inte vid kompileringstid så som i programmeringsspråk med statisk bindning (mera om denna bindning t.ex. i (Sebesta 579-581)).

Det bör dock observeras att det här slags dynamisk bindning inte heller är något nytt fenomen utan dynamisk metodbindning har funnits i form av procedurvariabler tidigare. I en del språk har användning av procedurvariabler varit klumpigt och svårt vilket lätt har lett till fel. (Mössenböck 5)

3 Alternativ till klassbaserad objektorientering

Objektorienteringen förstås ofta enbart såsom jag tog den upp i kapitel 2, d.v.s. man har moduler som innehåller klasser varav man skapa instanser som kapslar in data och procedurer. Det finns dock alternativa sätt att se på objektorienteringen. I detta kapitel tar jag fram prototypbaserad objektorientering och diskuterar litet om delegation.

3.1 Prototypbaserad objektorientering

I klassbaserad objektorientering sätter man stränga gränser för vad objekten står för genom att objekten tillhör en viss klass. Visserligen med

gränssnitten kan man få dem att *bete sig* såsom objekten i en annan klass men fundamentalt hör ett objekt (oftast) till en bestämd klass. Dessutom är denna ”klasstillhörighet” fastslagen en gång för allt under objektets livstid.

I prototypbaserad objektorienterad är angreppssätt helt annat, man har slopat klasser och låter varje objekt stå för sig självt. Det är endast objektens beteende som bestämmer dess egenskaper och likhet med andra objekt. Jämförelsen med verkligheten är här tydlig: t.ex. skillnaden mellan en byggnad och ett hus kan vara svårt att definiera. (Craig 58-63). Konsekvensen av denna koncentration på beteende är att abstrakta objekt (jfr. abstrakta klasser) har mindre betydelse i prototypbaserad objektorienterad programmering, abstrakta objekt är ju ofullständiga, det fattas något som medför att de inte kan bete sig så som ”normala” objekt.

Nya objekt skapas i prototypbaserade språk genom att kopiera befintliga objekt, men i stället för att bara ytligt kopiera namn och typer av data och referenser till metodimplementeringar såsom i klassbaserade språk denna kopiering – som ofta kallas kloning i detta sammanhang – omfattar själva data och koden för metoderna. En prototyp är således en samling av (godtyckligt) data och (godtyckliga) metoder. Hur mycket två objekt liknar varandra bestäms på basen av likheten i data och metoder (för en närmare beskrivning av denna bestämningsmekanism se t.ex. (Craig 61)).

Även om objekten inte ärver i den mening som det förstås i klassbaserad objektorientering innehåller objekten ofta en referens till det objekt de klonades från. De kan även innehålla referenser till andra objekt så att dessa kan utnyttjas när objektet behöver att ha något utfört. Denna mekanism kallas delegation. Dess kraft är att dessa referenser kan ändras under exekveringen – metoderna i klassbaserad objektorientering är ju statiska, de binds vid kompileringen. Delegation kallas därför emellanåt för dynamisk ärvning.

Många prototypbaserade språk använder sig av meddelande i kommunikation mellan objekten i stället för metodanropen. Meddelanden innehåller identifierande information om vilket objekt meddelandet är avsett för eller sen de sänds som bred- eller flersändning.

Javascript är ett känt exempel på prototypbaserat objektorienterat programmeringsspråk.

Eftersom Python har inga åtkomstbegränsningar och stöder prototyp-baserad objektorientering med introspektion måste man vidta speciella åtgärder för att göra data och/eller metoder privata. Python stöder delegation.

Python stöder prototyper med introspektion – introspektion betyder att man kommer åt RTTI-blocken i objekten. (Lutz 426-427, 499-500)

3.2 Delegation

4 Hur tillämpa objektorienteringen

Objektorienterad programmering är mera än att ha data i form av objekt i fokus i stället för för algoritmer. Framgångsrik tillämpning av objekt-orientering förutsätter att man känner igen typiska situationer där objekt-orienteringen kan användas till god effekt samt att man aktar sig för några vanliga misstag. I detta kapitel visar jag sådana situationer och fall.

4.1 Typiska tillämpningar

Objekt lämpar sig bra för att modellera objekt i det reella värdet: köer, sensorer, reläer o.s.v. (Mössenböck 10-11)

I objektorienterad programmering gör man färre typkontroller och de som finns är mera flexibla än de i den procedurorienterade programmeringen.

(Mössenböck 75-94) räknar upp sex situationer där objektorienteringen lönar sig.

4.1.1 Abstrakta datatyper

Med klasser kan man strukturera sammanhängande data och operationer. Inkapsling hjälper att dölja implementeringsspecifika detaljer och därmed minskar komplexitet i ett mjukvarusystem. Inkapslingen sker dock med extra kod och minskad effektivitet. Det lönar sig att hitta balansen, t.ex. en människas lön går bäst att representera med ett reellt tal, inte med en klass.

4.1.2 Generisk programmering

En datastruktur kallas generisk om den kan jobba med olika typer av objekt. Vissa programmeringsspråk har en speciell datatyp för konstruktion av generiska datastrukturer, t.ex. `Variant` i Realbasic, medan andra har speciell syntax för generisk programmering.

```
public class AssociativtFält<N,V>
{
   private final N nyckel;
   private final V värde;
   public N geNyckel()
   {
      return nyckel;
   }//  geNyckel
   public V geVärde()
   {
      return värde;
   }//  geVärde
}//  class AssociativtFält
```

Figur 7: En implementering av ett associativt fält med generisk datatyp i Java.

Generisk programmering kan simuleras med arv - ofta med arv av gränssnitt – i språk utan explicit stöd för den. (Mössenböck 77, 81)

4.1.3 Heterogena datastrukturer

Heterogena datastrukturer är en av de nyttigaste tillämpningar av objekt-orienterad programmering som karakteriseras av att objekten har varianter som ska behandlas likadant och man vet inte på förhand antalet varianterna.

Det klassiska exemplet som man använder för att illustrera heterogena datastrukturer, är ett grafikprogram varmed man kan rita olika geometriska figurer: linjer, rektanglar, cirklar o.s.v. Utan objekt-orientering blir man tvungen att skriva kod där det i flera ställen testas vilken figur det är fråga om. Med heterogena datastrukturer kan man skriva kod som fungerar med figurer generellt.

4.1.4 *Delegation*

Som ovan konstaterats är delegation är mekanism där ett objekt i stället för att utföra ett jobb sig självt ber ett annat objekt att utföra det. Det klassiska exemplet på delegation är utskrift. När ett program ritar ut någonting på skärmen, sker detta på en rektangelformig area, ritområde (engelska *frame*). I stället för att ha olika rutiner för att rita samma utskrift på papper, är det bättre att abstrahera alla utskrifter till ett objekt som kan stå såväl för en rektangel på skärmen som för en skrivare. Detta objekt kallas en grafisk port. Skärmen och olika skrivare implementeras då som underklasser av porten och all utskrift sker via den.

```
class Port(object):
   def RitaLinje(self,…): pass

class Skärmport(Port):
   def RitaLije(self,…):
      …

class Ritområde(object):
   port = None
   def RitaLinje(self,…):
      self.port.RitaLinje(…)
```

Figur 8: Ett exempel på delegation: utskrift via port i Python.

4.1.5 *Anpassningsbara byggnadsdelar*

Anpassningsbara byggnadsdelar är sådana datastrukturer som kan anpassas för att erbjuda litet annan funktionalitet. Det klassiska exemplet är typografiska stilar i text. Låt oss ha en klass `Text` som erbjuder all funktionalitet för att hantera text men som saknar typografiska stilar, d.v.s.

texten har ett fixt typsnitt, den saknar fet eller kursiv stil m.m. I stället för att skriva en klass för text med typografiska stilar `StiladText` från början, kan man utvidga klassen `Text`.

Denna utvidgning kan ske via ärvning eller s.k. klassutvidgning som kan uppfattas som ”lätt ärvning”.

```
Function Innehåller(Extends sträng as String,
                        delsträng as String) as Boolean
   return sträng.InStr(delsträng) > 0
End Function
```

Figur 9: Utvidgning av en datatyp Realbasic utan ärvning. ()

(“Extends - REAL Software Documentation”)

4.1.6 Halvfärdiga programbibliotek

Halvfärdiga programbibliotek används för standardisera och därmed förbättra och försnabba mjukvaruutveckling. De utnyttjar ofta generisk programmering eller delegation.

Det förstnämnda möjliggör att halvfärdiga datatyper kan lagras med sina metoder i olika programbibliotek för att utnyttjas i flera mjukvaruprojekt (Mössenböck 63).

4.2 Vanliga misstag

4.2.1 Triviala klasser

Det lönar sig inte kapsla in data som är inte invecklat, där inkapsling bringar ingen nytta, t.ex. det finns vanligtvis ingen vits att representera lön som instanser av en klass när ett reellt tal räcker. Lyckligtvis är nästan uteslutande alla programmeringsspråk som används i större skala för objekt-orienterad programmering är hybridspråk d.v.s. sådana som stöder såväl objektorienterad som icke-objektorienterad programmering.

4.2.2 Aggregation och komposition

Man förväxlar relationer mellan objekt. Det klassiska exemplet gäller punkter i ett ritprogram. Man utvidgar en klass ”punkt” till en klass ”linje”. Då *är* linje begreppsmässigt ett specialfall av punkt, fast det som man kanske syftade var att linje *har* två ändpunkter.

4.2.3 Felaktig ärvning

Att förväxla över- och underklass: en kvadrat är ett specialfall av en fyrhörning och inte tvärtom.

4.2.4 Identiska varianter

Två klasser har endast olika värden på ett och samma data, t.ex. röda och gula stugor blir skilda objektklasser i stället för en klass ”stuga” med attribut ”färg” med varierande värden.

4.2.5 Felaktig association med en klass

Metoderna associeras med fel klass: avlägsnande av ett element ur listan är en operation på listan, ej på element.

5 Närliggande fenomen

I det fjärde kapitlet kommer jag ta upp fenomen i objektorienterade programmeringsspråk som strängt taget inte är nödvändiga ur objekt-orienteringens synvinkel men som ofta associeras med objektorienteringen.

6 Sammanfattning

Utvidgningsbara abstrakta datatyper med dynamisk bindning kallas klasser som är centrala i objekt-orienterad programmering. M.a.o. objekt-orienterad programmering är programmering med abstrakta datatyper genom utnyttjande av ärvning och dynamisk bindning. (Mössenböck 6)
Utvecklingen av programmeringsspråk har förstås inte stannat i objektorientering och nya idéer har presenterats. Jag avslutar avhandlingen med en kort diskussion om dessa nya rön.

Litteratur

Craig, I. *Object-oriented Programming Languages : Interpretation*. London: Springer, 2007. Print.

"ETH - Oberon - Language Archives Oberon and Old Compilers." Web. 23 Feb 2011.

"Extends - REAL Software Documentation." Web. 2 Apr 2011.

Ford, Gary. *Modula-2 : A Software Development Approach*. New York: Wiley, 1985. Print.

"Heap (datastruktur) - Wikipedia." Web. 22 Feb 2011.

Lutz, Mark. *Learning Python*. 4th ed. Beijing ; Sebastopol: O'Reilly, 2009. Print.

Mössenböck, Hanspeter. *Object-oriented Programming in Oberon-2 Objektorientierte Progammierung in Oberon-2*. Berlin: Springer, 1993. Print.