Tumregler för programmering

Lars Arvestad
Stockholms universitet

När man lär sig programmera börjar man med små övningar för att utveckla grunderna i programmering. För att lösa verkliga uppgifter, med praktisk relevans, behövs oftast betydligt större och mer svåröverskådliga program och dessa brukar kräva en god *programstruktur*, dvs en god logisk struktur där uppgiften bryts upp i mindre delar med hjälp av funktioner, klasser, och moduler. Som nybörjare kan det kännas svårt att komma igång med att skriva större program just för att det är svårt att hitta en struktur att börja med. Man vet att det behövs en uppdelning av programmet, men hur ska den göras?

Det är en konst att hitta rätt struktur på ett program. Erfarna programmerare tycker ofta att skillnaden på vad som bra och dålig struktur på ett program är uppenbar, men det är inte så lätt att lära ut den stilkänslan. Man kan också tycka att det går att se skillnad på vad som är bra och dåligt, men ändå tycka att det är svårt att aktivt göra de designval som krävs. Man utvecklar omdömet genom att öva, följa andras goda exempel, och att skriva egen kod som man förbättrar iterativt. Detta tar förstås tid.

Man kan notera att även duktiga programmera kan ha dålig ”stilkänsla” och skriva program som få andra kan läsa på ett enkelt sätt. Att en person har skrivit stora och välfungerande program är alltså inte en garanti för att programmen är välskrivna.

Ett förslag på en programstruktur behöver inte vara perfekt för att vara användbart. Även en suboptimal struktur kan vara en god start på en prototyp som kan ge insikter som ligger till grund för en bättre struktur. Och man lär av misstag! Moderna arbetsmodeller för mjukvaruutveckling går ut på att tidigt identifiera dåliga strukturbeslut och genast åtgärda dessa för att undvika att man betalar för misstagen senare i projektet. Omstrukturering av program kallas ibland ”refaktorering”, efter det engelska *refactoring*, och syftar till att förbättra koden steg för steg. Detta är så centralt i programutveckling att moderna utvecklingsverktyg har automatiserat stöd för vanliga refaktoreringsåtgärder.

Bra programstruktur är också viktigt för kodförståelse. Genom att undvika dåliga stilval gör du ditt program mer lättillgängligt för andra, samt för ditt framtida jag! Om du själv har svårt att förstå hur du tänkte tre månader tidigare, hur svårt är det då inte för andra att läsa samma program? Snygga och lättlästa program brukar dessutom vara lättare att felsöka.

När jag har frågat kollegor om var man läser om hur man strukturerar program så har flera hänvisat till boken *Code Complete* av Steve McConnell. Det är en utmärkt rekommendation, men antagligen inte för nya programmerare. För de som kan programmera finns det mycket material, men för nybörjaren tror jag det kan vara svårare att hitta lämplig läsning. Som stöd för att komma igång att strukturera program har jag samlat ihop några tumregler för programmering.

Dessa tumregler är definitivt inga lagar och det finns säkert många programmerare som inte skulle hålla med om dem, men jag tror de är en bra start för nybörjaren.

Även om tumreglerna är skrivna med Python i åtanke så är de ganska generella och bör gå att använda även för andra programmeringsspråk. Programmeringsspråk har dock olika kultur och förväntningar vad gäller hur man uttrycker sig och strukturerar sitt program. Dessa skillnader bottnar ofta i styrkor och svagheter hos språken och man bör följa de konventioner som finns för det språk du använder. Ett trevligt sätt att lära sig om hur man uttrycker sig i olika språk är att titta på videor från presentationer på konferenser och andra träffar som intressegrupper organiserar.

Den centrala tanken med tumreglerna är att *kod ska vara läsbar*. Genom att göra det lätt att läsa ditt program, vare sig det gäller hur man använder tomma rader, namnger variabler, eller väljer funktioner att implementera, blir det också lättare att förstå hur det är tänkt att beräkningarna ska ske.

1 Tumregler för att strukturera sitt program

- **En funktion bör få plats på en skärmsida**. Det gör det möjligt att snabbt granska hela funktionen utan att behöva belasta sitt minne med detaljer om vad en variabel är initialiserad till eller vilka parametrar funktionen tar. Med en omedelbar överblick av en funktion är det ofta lätt att avgöra hur en funktion fungerar utan att behöva förstå detaljerna.

  Om en funktion inte får plats på en skärmsida så bör man identifera block som kan brytas ut som oberoende enheter — det vill säga som funktioner.

- **Använd funktioner för att gruppera kodrader**. Programmerare kan skapa struktur i långa kodblock med hjälp av blankrader och kommentarer, men funktioner är ett starkare verktyg för att införa struktur. Syftet med ett block kod och vilka beroenden det har blir ofta tydligare om man använder flera korta funktioner istället för en lång funktion uppdelad i kodblock. Ytterligare en fördel är att det blir lättare att verifiera delar av koden med enhetstestning.

- **Separera *beräkning* från *interaktion***. En funktion som gör en beräkning bör inte ha `print`- eller `input`-satser (eller motsvarande). Som exempel: om din ekvationslösare har en massa diagnostiska utskrifter så minskar kodens användbarhet i andra projekt där den skulle kunna användas. På samma sätt kan en funktion bli oanvändbar om den frågar användaren efter data istället för att ta informationen som parametrar. Se ett litet exempel i figur 1.

- **En funktion ska göra en sak och göra det bra.** Som exempel kan man fundera på hur man skriver ett litet program för att läsa in flera dataset, gör beräkningar (tex lite statistik) på vart och ett, och sedan skriva ut en sammanfattning av resultatet. En sådan uppgift går att skriva som ett sammanhållet program utan funktioner, men det blir enklare att förstå koden om man har:

  – en funktion för inläsning av en fil,

  - en funktion som gör beräkningen på ett dataset,
  - en funktion som loopar över indata-filerna och sammanfattar.

  Denna uppdelning har flera fördelar.

  1. Programstrukturen blir tydligare när man brutit upp koden i delar.
  2. Det är lätt att få en snabb överblick över en kort funktion, så koden blir mer lättläst.
  3. Det är lätt att verifiera att funktioner räknar rätt när deras funktionalitet är begränsad och lättförståelig.

- **Välj ett informativt namn på funktionen!** Dåliga namn är till exempel `start`, `compute`, och `f`, eftersom de inte säger något specifikt om vad funktionen gör. Namn som `remove_outliers`, `newton_raphson`, och `compute_integral` kommunicerar däremot ganska tydligt vad som ska göras, eller till och med *hur* det ska göras (`newton_raphson` är ett namn på en algoritm).

  Det kan vara lockande att använda en förkortning som variabelnamn, och det är bra om det är en etablerad förkortning. Det är mindre bra att hitta på egna förkortningar.

  Ett tecken på att man har ett dåligt variabelnamn är att det behövs en kort kommentar för att förklara det. Ett exempel:

  ```
  nc = 1.7        # normalisation constant
  ```

  Här är det bättre att kalla variabeln `normalisation_constant`.

  Det finns tillfällen när det inte är så viktigt med ett informativt namn. Det är till exempel en konvention att `main` är namnet på den funktion som är programmets startpunkt. Om man skriver en funktion som tar en annan funktion som argument kan det mycket väl vara försvarbart att kalla argumentet `f` (som förkortning av "function"), särskilt om dokumentationssträngen talar om vad `f` är.

  Kod förändras och då är det viktigt att komma ihåg att funktionsnamn ibland också bör ändras. Om en funktion heter `print_integral`, men inte gör några utskrifter (som den kanske gjorde i ett första utkast), så är det förvirrande. Kanske borde den heta `compute_integral`?

```
def birthdate():
    swedish_id = input()
    date_part = swedish_id.split('-')[0]
    return date_part
```

(a) Ett dåligt exempel.

```
def birthdate(swedish_id):
    date_part = swedish_id.split('-')[0]
    return date_part
```

(b) Ett bättre exempel.

Figur 1: Två exempel på en liten funktion för att extrahera födelsdatum från ett personnummer. (a) Ett dåligt exempel med minst två misstag: dels bryter funktionen mot tumregeln att separera användarinteraktion och beräkning, dels bryter man mot tumregeln att skicka data via funktionens parametrar. (b) En bättre lösning: man förutsätter att personnummret matas in någon annanstans. Denna funktion är mer generellt användbar.

- **En funktion bör bara bero av sina parametrar.** Man ska kunna förstå en funktion utan att titta på andra delar av ett program. Om man låter en funktion läsa globala variabler så blir funktionens beroenden mer otydliga, för läsaren tvingas titta igenom funktionens hela definition för att avgöra vad indata är. Det är ofta direkt olämpligt att manipulera globala variabler (dvs använda nyckelordet `global`) eftersom det kan bli svårt att hitta var en variabel ändras.

  Det finns tillfällen då det är försvarbart att använda globala variabler, men den brett accepterade tumregeln är att undvika dem.

- **Tänk på vad du returnerar!** Att ha en `return`-sats (eller flera!) gör din funktion tydlig och lättläst. Kom ihåg att man kan ha multipla returvärden!

  Det finns dock flera möjliga misstag som man ska undvika vad gäller returvärden:

  - Returnera inte numeriska värden som strängar även om du vet att de ska hamna i en sträng senare. Det är mer framtidssäkert att returnera numeriska värden som just numeriska värden.
  - Returnera inte ”magiska värden” för att markera fel eller problem. Använd hellre särfall som är det etablerade sättet att hantera fel. Om du skriver kod för att iterativt lösa en ekvation så är `return 1000000` en dålig markör för “konvergerade inte”. Det är då bättre att skriva `raise Exception("Did not converge")`.

    Rent generellt är det problematiskt med värden som ska ha en särskild betydelse som användaren av en funktion måste ha koll på. Det finns ett undantag: `None` är ett acceptabelt magiskt värde som med fördel kan användas för att markera brist på resultat.
  - Hitta inte på egna sätt att paketera flera värden som ett returvärde. Det är, till exempel, inte bra att lagra numeriska värden i en sträng för att senare extrahera dem. Dels blir det alldeles för lätt problem med konvertering fram och tillbaka, dels är det långsamt att göra konverteringar av det slagt. Det finns ett etablerat och praktiskt sätt att returnera flera värden och det är med *multipla returvärden*.

- **Varje funktion bör ha en dokumentationssträng**.

- **Variabler är bättre än litteraler**. Det är svårt att läsa en litteral som “10” i kod och veta vad det talet representerar. Om man däremot har raden `n_iterations = 10` så hjälper identifieraren dig att ge mening till värdet 10. Genom att referera till `n_iterations` blir ditt program mer lättläst än om du refererar till litteralen själv. En annan viktig fördel är att ditt program blir lättare att ändra: istället för att leta upp alla (relevanta) användningar av 10 räcker det med att ändra en tilldelning.

2 Indikationer på problem

Även om man har bra koll på hur man borde skriva kod är det lätt att tappa bort sig och, kanske av slentrian, börja bryta mot god praxis. Då är det bra att uppsikt efter enkla indikatorer på att man gör fel. Här är några sådana.

- **Djup indentering** indikerar en onödigt komplicerad implementation med nästlade loopar och villkorssatser. Koden blir då svår att läsa för att det är många kombinationer av villkor att hålla reda på och man bör försöka ”platta till” koden. Se ett exempel i figur 2.

  En vanlig orsak till djup indentering är att man har flera olika logiska fall — skapa då funktioner för de olika fallen. På så vis blir fallanalysen tydligare och de olika funktionsanropen indikerar vad som sker i fallen.

- **Saknar funktioner parametrar?** Funktioners syfte är vanligen att ta några data och beräkna något. Beräkningen behöver inte vara matematisk utan kan vara enkel behandling av data. Det är mindre vanligt att funktioner helt saknar parametrar, men det finns förstås legitima exempel, till exempel för utskrift av en meny eller initialisering av datastrukturer. Hos nybörjare är det dock inte ovanligt att man finner det bekvämt att lägga data i en eller flera globala variabler som funktioner sedan arbetar mot. En sån konstruktion gör det dock svårt att se hur funktioner beror av varandra; man måste läsa funktionerna noggrant för att identifiera programflödet.

- **Har du svårt att skapa enhetstester?**

  Det kan vara ett tecken på att det är dags för omstrukturering. Du kanske inte har tillräckligt enkla funktioner, har skrivit dem utan returvärden, eller har för starka beroenden av globala variabler.

```
def get_valid_records(data_records):
    good_records = []
    for item in data_records:
        if item.x >= 0:
            if item.y > 0:
                if item.val > 0:
                    if item.location:
                        good_records.append(item)
        elif item.y < 0:
            if item.val > 0:
                if item.location:
                    good_records.append(item)
    return good_records
```

(a) Ett exempel på "djup indentering". Kan du enkelt läsa av vad som gör att en datapost betraktas som "bra" av koden?

```
def get_valid_records(data_records):
    good_records = []
    for item in data_records:
        if record_quality(good_records, item):
            good_records.append(item)
    return good_records

def record_quality(good_records, item):
    '''Return True for a "good" item, False otherwise.'''
    if not item.location:
        return False

    if item.x >= 0 and item.y >= 0 and item.val > 0:
        return True
    elif item.x < 0 and item.y < 0 and item.val < 0:
        return True
    else:
        return False
```

(b) Denna kod är ekvivalent med (a), men strukturerad på annat sätt. Iteration och hantering av bra poster separeras från bedömningen av posterna. I `record_quality` är villkoren inte nästlade, utan samlade så att lättare ser de villkor som gäller för bra poster. Dessutom tar funktionen ett tidigt beslut om `item.location` är tom.

Figur 2: Två sätt att skriva kod för att iterera igenom poster (`data_records`) och samla ihop de som uppfyller vissa villkor. Vilken variant är lättast att läsa, (a) eller (b)?