Ämnesdidaktiska principer för introduktionskurs i programmering

Ett förslag från Henrik Sandklef och Rikard Fröberg

Innehåll

Om detta dokument

Dessa principer är under utveckling och skall betraktas som ett levande dokument som kommer stegvis förfinas under iterationer av kursutveckling där nya erfarenheter utgör indata till principerna. Principer kan läggas till, strykas eller förfinas om ny erfarenhet påverkar principernas giltighet. Principerna skrevs ned som en dokumentation av de överväganden och tankar som växte fram under framtagandet av en kurs.

Målgrupp

Målgruppen för detta dokument är lärare och andra som är intresserade av processen att lära ut IT och programmering. Vi är öppna för kritik och ser gärna att våra tankar och principer kritiseras, kompletteras och kommenteras för att dokumentet skall kunna förbättras över tiden.

Bakgrund

När vi åtog oss uppdraget att reformera introduktionsveckorna till programmeringsdelen till kursen *TIG015 Informationsteknologi och informationssystem*[1], började vi med att slå fast en mängd principer vi skulle ha att hålla oss till. Principerna växte fram under diskussioner om kursens innehåll, diskussioner med tidigare studenter, reflektioner över tidigare tentamensresultat, slutsatser dragna ur insamlad data från två års undervisning av kursen och en dialog med handledare som själva gått kursen tidigare. Vi vill därför passa på att nämna att dessa principer inte kunnat vaskas fram utan givande diskussioner med studenter och studenthandledare.

Syftet med att dokumentera dessa principer var ett försök att konkretisera en idé om hur programmering kan läras ut med målet att alla studenter skall uppnå lärmålen och därmed klara av en tentamen. Det ges ingen enskild tentamen på introduktionsdelen i programmeringsdelen av kursen men denna idé skulle i förlängningen kunna appliceras på hela programmeringsdelen av kursen. Vi satte därför upp mål för vad studenterna skulle kunna efter introduktionsveckorna för att kunna tillgodogöra sig resten av programmeringsdelen i kursen på bästa sätt.

Programmeringsdelen i kursen använder Java som programmeringsspråk så vårt fokus var att lära ut programmeringsgrunder med syfte att förbereda studenterna på en kurs med Java som programmeringsspråk med allt vad det innebär.

Våra principer är dock agnostiska, hoppas vi, för vilket ämne som de skall appliceras på, vilket vi upplever gör att de går att applicera på hela kursen och andra kurser i Systemvetenskapliga programmet i deras helhet. Däremot är en del konkreta exempel bundna av den specifika kursdelen och programmeringsspråket Java.

---

[1] https://utbildning.gu.se/kurser/kurs_information?courseId=TIG015 Första kursen i Systemvetenskapliga programmet på Göteborgs Universitet

Värdegrund för didaktiken

Som ett steg i att ta fram principer för didaktiken har vi lutat oss mot en värdegrund. Denna genomsyrar principerna och vi känner att den är viktig att formulera då alla som vill försöka applicera dessa principer bör dela denna värdegrund (eller åtminstone vara medveten om den).

Värdegrunden kan sammanfattas i punktform:

- Var generös - dela med dig av ditt material i tid, publicera material och exempel
- Var tydlig - din roll är att förenkla studenternas inlärning
- Ha respekt för studenterna och deras svårigheter - ett ämne som *du* behärskar kan det vara svårt att inse att *studenterna* upplever som mycket svårt
- Använd tillgänglig IT som ett stöd för studenterna och för dig[2]
- Förutsätt ingen kunskap hos studenterna som inte är ett formellt och i förväg kommunicerat förkunskapskrav
- Motivera och inspirera - Varför är momentet viktigt? Varför skall studenten lyssna på din föreläsning (om just detta)?

Själva principerna följer längre fram i detta dokument och anges numrerade för enklare referens till enskilda principer.

Särskilt om val av kurslitteratur

En sidoeffekt av våra nedan föreslagna principer är att man också kan använda dem för att välja kurslitteratur till kursen. Den som tycker våra principer verkar användbara, och vill prova att använda sig av dessa i undervisningen, gör klokast i att finna litteratur som också uppfyller dessa principer.

Som kriterier för val av kurslitteratur passar särskilt följande principer:
§ 2. Undvik cirkeldefinitioner
§ 3. Undvik uppskjuten definition (så långt det är möjligt)
§ 5. Använd stegvis introduktion transparent och öppet
§ 6. Hänvisa tillbaka till tidigare stegvisa introduktioner
§ 8. Ge vägledande övningar
§ 9. Undvik för svåra övningar
§ 10. Undvik motsägelser (egna och mellan kolleger)
§ 11. Identifiera och hantera motsägelser
§ 12. Använd konsekvent terminologi och förklaringsmodell
§ 13. Använd konsekventa symboler och namn
§ 14. Visa inkonsekvens endast när syftet är att något kan varieras
§ 18 Fokusera en föreläsning(ett kapitel) på ett koncept åt gången

[2] Var dock noga med att välja IT-stöd som verkligen hjälper dig med undervisningen - det finns IT-stöd som kan göra saken värre. Lägg tid på att utvärdera och analysera de IT-stöd som används i undervisningen.

Vi planerar att författa även ett dokument om val av kurslitteratur, där vi avser fördjupa en diskussion om hur våra principer skulle kunna användas för att utvärdera kurslitteratur utifrån vår grundläggande syn på programmeringsdidaktik.

Tankar om didaktik

Med didaktik menar vi den definition som ges av Arfwedson i "Didaktik för lärare":

> *[Vetenskapen] om alla faktorer som påverkar undervisning och dess innehåll, och sätter fokus på lärande och hur lärande organiseras i till exempel klassrummet, online eller på museum.*[3]

Vi gör inte anspråk på att ha ett vetenskapligt förhållningssätt till didaktik eller pedagogik, utan har blott valt att kalla principerna för "didaktiska" i brist på bättre term. Våra avvägningar som ligger till grund för principer och didaktiska tankar kommer från erfarenhet och observationer från vår egen bakgrund som lärare (och i viss mån även som studenter, vilket dock ligger långt tillbaka i tiden).

Våra tankar om didaktik för IT och programmering kan delas in i två delar:
(a) Grundläggande mål för utbildningsverksamhet
(b) En mängd principer för didaktiken för utbildning inom IT och programmering. (b) beskrivs i detalj i de följande avsnitten i detta dokument.

(a) Grundläggande mål för ubildningsverksamhet

Vi anser att utbildningsverksamhetens mål är att överföra kunskap från tidigare erfarenhet och vedertagen kunskap inom IT och programmering till studenterna. Det övergripande målet är att studenterna, givet en rimlig arbetsinsats, tillgodogör sig de kunskapsmål som kursplanen anger och att tentamen används för att mäta såväl *hur väl studenterna lever upp till den rimliga arbetsinsats som uppfyllandet av målen kräver*, som *hur väl utbildarna lyckats leverera den kunskapsöverföring som utlovas enligt kunskapsmålen i utbildningsplanen*[4].

(b) Principer

Principerna genomsyras av vår syn på förutsättningar för att uppnå målen i (a). På en övergripande nivå kan man förhålla sig till undervisningen som ett slags kontrakt mellan lärarna och studenterna. Lärarna antar *en generös inställning* som går ut på att studenterna skall ges alla verktyg, allt material och all information som behövs för att de ska kunna uppnå lärmålen. Studenterna antar ett ansvar som förpliktar dem att ta till sig den information som erbjuds eller aktivt efterfråga information som eventuellt saknas samt lägga ned erforderlig tid och ansträngning på att öva sina färdigheter i ämnet.

Ett exempel på *generositet* är att i god tid som lärare lägga upp övningar, föreläsningar, scheman, exempel, läshänvisningar och så vidare så att dessa finns tillgängliga för studenterna. Se avsnittet om **värdegrund** ovan.

---

[3] http://libris.kb.se/bib/8399845?vw=full Arfwedson, Didaktik för lärare

[4] Märk väl att detta förutsätter existensen av en väl avgränsad och väl definierad kursplan med tydliga, motiverade kunskapsmål i relation till utbildningens övergripande mål. Saknas en kursplan med kursinnehåll och mål på tillräckligt detaljerad nivå, tyder detta på att en sådan plan bör tas fram.

Vad beträffar studenternas ansvar kan ges som exempel en förväntan att de går på föreläsningarna, studerar material som läggs upp, gör övningar och ställer frågor när de kör fast eller inte förstår.

Den dynamik som vi hoppas uppnå går ut på att det måste finnas balans mellan lärarens generositet och studenternas ansvarstagande. Det hjälper inte att som lärare vara generös med att lägga upp material om ingen tar till sig materialet. Omvänt gäller att det inte hjälper att vara en ansvarstagande och aktiv student, om det saknas material och övningar eller om man inte får svar på sina frågor.

Naturligtvis bör *lärarens insatser* vägas i förhållande till kursens omfattning i tid och högskolepoäng, varför detta dokument i stor grad fokuserar på de principer man bör beakta under förberedande såväl som genomförande av en kurs.

Kurser bör ses som en samling levande dokument och andra artefakter som kontinuerligt utvärderas och förbättras (gärna i enlighet med de principer som föreslås i detta dokument). Det är emellertid viktigt att varje lärare inser att en kurs tarvar insatser från läraren vad gäller förebyggande kvalitetsarbete och formaliserat säkerställande av kvaliteten. Det är rimligt att läraren lägger tid i sin anställning på detta arbete och att läraren kan ta hjälp av kollegor och experter i arbetet med att säkerställa att materialet är korrekt och lämpligt. Studenter utgör i viss mån en expertgrupp, vad gäller expertis i att gå kurser, och bör inkluderas i kvalitetsarbetet.

En viktig källa till återkoppling av eventuella brister i kursen är reaktioner och synpunkter som framkommer i dialog med studenterna under och efter kursen (i t ex kursvärdering). Naturligtvis är direkt återkoppling under kursens gång att föredra eftersom läraren då har möjlighet att rätta till eventuella fel direkt, vilket gagnar samtliga studenter som då går kursen. Återkoppling via kursvärdering är värdefull men gagnar dessvärre först kommande årskullar studenter.

Tentamen är också i viss mån en värdemätare av kursens kvalitet. Detta gäller dock endast om tentamen är noggrant framtagen för att endast mäta de kunskaper som lärts ut och (helst) sådana färdigheter som studenterna givits möjlighet att öva på under handledda övningspass. Ett dåligt “tentaresultat” där tentamen väl återspeglar innehållet i kursen kan bara betyda två saker. Antingen har studenten inte studerat tillräckligt eller så har innehållet inte lärts ut tillräckligt bra. Det är därför väldigt viktigt att säkerställa tentamens kvalitet utifrån relevans och tydlighet. En uselt producerad tentamen är ingen värdemätare på vare sig studenternas kunskaper eller kursens kvalitet.

Mål och förhoppningar

Vi ser att (a) och (b) ovan tillsammas ger följande situation:
(1) Målen i (a) är de mest eftersträvansvärda målen och vad vi förstår de mest realiserbara målen.

(2) Principerna (b) i detta dokument erbjuder störst möjlighet att uppnå målen i (a).

Själva principerna för hur undervisning och material kan utformas beskrivs i följande avsnitt.

Principerna (b)

§ 1. Arbeta bakifrån från lärmålet

Identifiera vad slutmålet är, det vill säga vad för kunskap om programmering ska studenterna ha efter avslutad kurs. Formulera ett exempel på vad de ska kunna producera och beskriva.

Arbeta från detta slutexempel (som kan vara en inlämningsuppgift eller en tentamen) och analysera vilka begrepp och färdigheter som ingår i slutmålet.

Titta sedan på vart och ett av begreppen och färdigheterna och bryt ned dem i beståndsdelar: Vad för begrepp måste man behärska för att kunna beskriva detta begrepp?

Arbeta er ned på detta sätt tills ni når någon form av botten eller baskunskap. Notera särskilt beroenden av typen

> "För att beskriva A måste vi använda och behärska begreppen a_0, a_1, ...a_n."

Det innebär naturligtvis att kursen måste gå igenom a_0 osv innan man går in på A.

För att denna princip skall kunna uppfyllas måste kursplan vara skriven på ett sådant sätt att det tydligt framgår vad för lärandemål kursen har.

§ 2. Undvik cirkeldefinitioner

Inom IT och programmering förekommer en mängd termer och begrepp där en term ofta måste förklaras i termer av andra IT-termer i olika lager. Det är viktigt att man undviker att introducera en term som definieras av andra nya termer. Ett exempel på detta skulle kunna vara:

> *En klass används som mall för att skapa objekt. Objekt är instanser av en klass.*

En sådan definition av termen "klass" bygger på en förståelse för termen "objekt", vilka i sin tur förklaras med termen "klass".

Här får man identifiera denna cirkulära definition och bryta ned den i tre termer:

- Klass
- Objekt
- Instans

Dessa termer bör introduceras var för sig och det är inte givet vilken ordning som är mest naturlig. Det viktiga är att så långt det är möjligt introducera, definiera och förklara ett nytt begrepp utan att förutsätta förförståelse för ännu ej introducerade begrepp.

Det finns tillfällen då man måste introducera två (eller fler) begrepp i samma kontext och då är det viktigt att försöka undvika inbördes beroenden mellan begreppen, det vill säga att varje begrepp kan förklaras inbördes så att en förklaring eller definition står på egna ben eller bygger på redan introducerade termer och begrepp.

§ 3. Undvik uppskjuten definition (så långt det är möjligt)

I situationer där ett koncept har många byggstenar är det lockande att fokusera på helheten och förespråka att vissa delar får sin förklaring längre fram så att studenterna får nöja sig med att kopiera vissa delar som de är utan att ännu lärt sig förstå varje del. En vanlig fras i litteratur eller undervisningssituationer är:

"Bry dig inte om X, Y och Z just nu, det kommer förklaras längre fram i boken/kursen"

Vi tror att detta är olyckligt eftersom det skapar en känsla av att något är mer komplext och svårbegripligt än vad det är, eftersom man explicit ber studenterna använda vissa delar av momentet utan att veta vad deras funktion eller roll är. Det finns också en risk att man aldrig kommer fram till det utlovade moment då dessa mystiska komponenter skall förklaras. En annan risk (eller möjlighet) är att nyfikna studenter inte låter sig besinnas och börjar läsa på i förväg för att få hela bilden klar för sig. Det skall alltid uppmuntras och ses som något positivt att någon vill läsa på i förväg men det kan skapa problem med lärarens planering eftersom det ofta dyker upp nya frågor och problem när studenterna så att säga läser på i förväg i ett avsnitt som är planerat att tas upp längre fram.

Ett tydligt exempel är ett program i programmeringsspråket Java som blott skall skriva ut "Hej" på skärmen, ett så kallat "Hello world"-program som brukar förekomma på första föreläsningen eller i första kapitlet i kurslitteraturen för programmeringsspråk.

I Java kan ett sådant program se ut som följer:

```
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello world!");
    }
}
```

Här är det viktigt som undervisare att ha klart för sig vad syftet med att studenterna ska skriva av detta program, kompilera det[5] och köra det är.

Oavsett vad syftet är så kan man konstatera att om detta är det första program en student ser så är i princip allt de skriver av utom texten "Hello world!" främmande och mer eller mindre obegripligt. Vill man fokusera på instruktionen för att skriva ut text på skärmen, det vill säga `println("Hello world!");` så kan man behöva säga att studenterna ska bortse från allt annat än så länge. Detta skapar en känsla för att programmering är oerhört komplicerat och byråkratiskt eftersom instruktionen består av 25 tecken av ett 134 tecken långt program eller ungefär 19%. Det är olyckligt om studenterna redan vid första kontakten får en bild av att 81% av ett så banalt program är obegripligt och för svårt att förklara på första lektionen.

[5] Kompilering är den process som färdigställer ett program i körbart format så att man kan exekvera programmet på datorn.

Detta är naturligtvis ett extremt exempel som är avhängigt val av programmeringsspråk vilket naturligtvis påverkar hur mycket kod som behövs för detta simpla program. Vår poäng här är att man får göra en avvägning av hur mycket begrepp och termer som behövs för att utföra en specifik uppgift så att studenterna exponeras för så lite obekanta termer som möjligt för att inte det som är fokus för föreläsningen (eller kapitlet) drunknar i oförklarade koncept som utlovas förklaring längre fram.

Det finns olika strategier för att hantera detta som man kan använda sig av för att slippa skjuta upp förklaringar av ingående termer till senare. Alltifrån att ta tjuren vid hornen och verkligen förklara varje ingående term direkt till att hitta en situation där man inte behöver alla termer.

Vårt syfte med denna princip är att undvika att framställa ett enskilt moment som onödigt komplicerat samt dölja störande moment som tar fokus för det man vill visa och diskutera. Dessutom finns även här en risk att man utlovar en förklaring “längre fram” som inte hinns med eller glöms bort samt, som sagt, risken att otåliga (eller ambitiösa och nyfikna) studenter inte kan ge sig till tåls och går händelserna i förväg genom att själva försöka reda ut det som skjutits upp till framtiden.

Vi vill verkligen inte avråda från självstudier och nyfikenhet men det problem vi identifierat är att planering av den ordning i vilken vi planerat ta upp olika moment onekligen störs om delar av studenterna tappar fokus och själva försöker läsa sig till koncept som planerats in först längre fram av skäl som motiveras av andra principer som till exempel att ordningen på introduktion av koncept är av stor pedagogisk betydelse.

§ 4. Undersök om verktyg finns för att dölja komplexitet

I de situationer som gör principerna §2 och §3 svåra att bemästra, till exempel det faktum att ett programmeringsspråk är omständligt och komplext i den bemärkelsen att det krävs mycket kod för att utföra en till synes banal uppgift, bör man överväga att introducera uppgiften med ett verktyg eller i en miljö där “onödig” komplexitet döljs eller inte behövs. Vi har laborerat med programmeringsmiljöer som reducerar programmet som ges som exempel i §3 till just raden

```
println("Hello world!");
```

Detta gör det möjligt att tala om instruktionen för att skriva ut text på skärmen, vad instruktionen heter och vilken form den har, att instruktionen avslutas med ett semikolon och att argumentet till instruktionen står inom parenteser och att text skrivs inom citationstecken. Vad vi slipper bortförklara är begrepp och termer såsom:

- public
- class
- static
- void
- main
- []
- {}

Detta utan att vi behöver skriva om ett enda tecken i det centrala i programmet, nämligen precis instruktionen:

```
println("Hello world!");
```

Detta är viktigt. Vi behöver alltså inte kompromissa med formen för det centrala i satsen när vi döljer "störande moment" eftersom den delen av programmet ser exakt likadant ut i programmeringsspråket Java. Vi kan då tryggt fokusera på instruktionen println, på parenteserna, citationstecknen och semikolonet, utan att den informationen skall vara irrelevant om vi längre fram kommer in i programmeringsspråket Java (där syntaxen för den centrala delen är identisk).

Vad vi vill ha sagt med denna princip är att det kan vara väl investerad tid att undersöka om det finns verktyg som hjälper till att avdramatisera ämnet utan att förlora essentiella beståndsdelar. Det kan också vara möjligt att sådana verktyg behöver utvecklas av läraren i det fall man inte finner något befintligt verktyg.

§ 5. Använd stegvis introduktion transparent och öppet

Många koncept inom IT och programmering har en stigande skala av komplexitet. Det faller sig ofta naturligt att introducera ett koncept först i sin allra simplaste form. Här är det viktigt att signalera till studenterna att detta är just den allra simplaste formen och att denna form stegvis kommer byggas på under kursen. Det skapar en förväntning av att något trivialt i förlängningen kommer utvecklas till något mer användbart och komplext, vilket kan vara en hjälp för att motivera att triviala exempel lärs ut och skall övas in. Det öppnar också för att repetera och hänvisa tillbaka i nästa steg när ett koncept byggs ut med fler delar och komponenter. Vanliga fraser som kan förekomma med denna strategi är:

> "Kommer ni ihåg hur X såg ut i sin simplaste form? Om vi behöver Y kan vi utöka X på följande sätt..."

Det underlättar för såväl läraren som studenten att se sammanhang och en röd tråd genom kursen och uppmuntra till naturliga punkter att gå tillbaka och repetera något. Ofta är det ju så inom IT och programmering att nya färdigheter bygger på tidigare färdigheter som förfinas och utökas. Med denna princip blir det lättare att såväl flagga för att en färdighet bara är början på något större, som att ge handfasta råd för vad som bör repeteras om man som student inte känner att man fullt kommer ihåg eller behärskar de förkunskaper som krävs för att momentet skall kunna förstås och ges mening.

Vi vill avråda från att dölja att ett moment blott visar en trivial och simpel form av ett koncept av flera skäl. Dels är det svårare att motivera för duktiga studenter varför något simpelt gås igenom. Dels är det svårare för studenterna att koppla tillbaka längre fram när konceptet utökas och fördjupas.

Det är också, anser författarna, befogat att här understryka vikten av en stringent och genomtänkt terminologi för koncept som introduceras. Genom att öppet redovisa att något är en grund eller byggsten som successivt kommer utökas och byggas ut längre fram, skapas en struktur för hur momentet och dess beståndsdelar skall benämnas. När momentet återkommer

och fördjupas är det med denna princip lättare att tidigt slå fast hur momentet benämns i varje återkommande del då det utökas, vilket vi tror leder till en mer sammanhållen benämning av begreppen och termerna som används.

Det skapar naturligtvis också en trygghet för studenterna att samma benämning återkommer när den återintroduceras och fördjupas.

§ 6. Hänvisa tillbaka till tidigare stegvisa introduktioner

Som nämns i §5, så är öppet redovisade stegvisa introduktioner av ett gradvis mer komplext och avancerat fenomen nyttigt för såväl studenter som läraren i det att det underlättar att tänka igenom såväl ordning som terminologi (benämningar av fenomenet) samt skapar förväntningar av att "det kommer mera". Denna konsekventa hållning skapar också möjligheter för läraren att konkret hänvisa till tidigare moment för repetition. En konsekvens av detta är att det blir tydligare för studenten vilka förkunskaper som krävs för ett moment samtidigt som det ges konkreta hänvisningar till vad för moment som behöver repeteras om förkunskaperna brister.

Genom att konsekvent hänvisa till tidigare steg ges en repetition så att säga på köpet när ytterligare steg läggs på ett koncept. Detta öppnar för att inleda en föreläsning (eller ett kapitel) med en kort repetition som skapar trygghet för studenten att detta moment inte är något helt främmande och nytt utan bygger på färdigheter och förståelse som studenten redan har (eller åtminstone förväntas ha) från tidigare moment. Känner studenten att den inte riktigt lever upp till de förväntade förkunskaperna ges åtminstone en kort repetition samt hänvisningar till vad som måste repeteras.

§ 7. Hitta och dokumentera balans mellan att dölja eller förklara

Om man använder principerna §§ 3,5 och 6 för att stegvis föra in nya nivåer av komplexitet så gäller det att försöka finna en nivå på hur långt tillbaka man måste gå i en genomgång.

Om man enligt §3 undviker att skjuta upp en förklaring genom att direkt förklara (eller repetera) alla ingående delar kan man i värsta fall traversera hela kursen baklänges till första lektionen. Det är naturligtvis att gå för långt och stjäl tid och fokus från målet med föreläsningen eller kapitlet. Det gäller här att tänka igenom och fatta beslut för ett slags rekursionsnivå[6] i hur långt man dekonstruerar begreppen i föreläsningen eller kapitlet.

Dokumentera dessa avvägningar och använd dessa beslut i nästa iteration av förbättring av kursen. Denna dokumentation är värdefull när man efter en längre tid reviderar en kurs och kanske inte längre minns hur vissa avvägningar motiverades. Det skapar också en möjlighet till diskussion med studenter och kolleger om dessa avvägningar.

[6] Det vill säga, i hur många steg man ska upprepa samma procedur i att repetera ingående delar i ett koncept

Det handlar alltså om att hitta avgränsningar för hur mycket repetition och förklaringar som skall ges. Ett exempel kan tas från en kurs i att baka en pizza. Man kanske måste förklara begrepp som deg, jäsning och gräddning. Men behöver man förklara mikrobiologin i jäsningsprocessen[7]?

§ 8. Ge vägledande övningar

Låt oss först säga att denna princip i synnerhet är applicerbar på introduktionskurser i programmering. Mer avancerade kurser kan naturligtvis ha mer utmanande övningar.

Ett sätt att stärka studenternas självförtroende är att ge dem övningar och uppgifter som är relativt avancerade men samtidigt ge dem så mycket hjälp på traven att de med mycket liten extern hjälp kan lösa dem. Det handlar om att analysera syftet med övningarna. Övningar som är så svåra att studenterna inte förstår vad de går ut på, för av förklarliga skäl med sig en rad problem. Studenterna får en känsla av att ämnet är för svårt eller att de inte lärt sig något.

Ett syfte med övningar kan vara att träna in nya färdigheter genom repetition som skapar en vana och trygghet. Det är bra för rena mekaniska färdigheter som exempelvis inom programmering att använda rätt syntax (som att inte glömma semikolon osv). Ett annat syfte kan vara att utmana studenterna att tänka i egna banor och finna egna lösningar.

Observera att inget av dessa syften uppnås om en uppgift är så svår (eller så olyckligt eller ofullständigt formulerad) att studenterna inte förstår vad de ska göra och inte kommer igång med uppgiften.

En strategi som vi prövat är att använda övningsuppgifterna som ett tillfälle att vägleda och repetera ett moment. Detta sker genom att frågan är formulerad på ett sätt som avslöjar de detaljer och delar i lösningen som behövs samtidigt som begrepp repeteras och förklaras i själva övningstexten. Det kan se ut så här:

> *“En metod i Java har ju en returtyp, ett namn och så eventuella parametrar som anges inom parenteser. I metodens kropp, som deklareras inom ett block som börjar med { och slutar med }, så sker själva arbetet som metoden skall utföra. Om metoden ska returnera ett värde, så måste ju metoden ha en sats med nyckelordet* `return` *där det värde som metoden tagit fram returneras. Värdet har som bekant samma typ som metodens returtyp.*
>
> *Skriv en metod med returtypen* `int`*, som tar två paramterar* `int a` *och* `int b`*. Metoden skall heta add och returnera värdet av a+b.”*

Jämför ovanstående med denna variant av exakt samma uppgift:

> “*Skriv en metod/ett program som tar två heltal som argument och returnerar summan av talen*”

[7] Vi menar detta som en uppriktig fråga; vi har ingen erfarenhet av undervisning i pizzalogi.

Den senare versionen är mycket enklare för läraren att skriva men förutsätter att studenterna själva klarar av att översätta underförstådda begrepp som *metod, argument*, *heltal*, *returnerar, summan* och hur dessa ska omsättas i Java-kod. Det vill säga ***uppgiften förutsätter just de färdigheter som syftet var att studenterna skulle öva*** på.

§ 9. Undvik för svåra övningar[8]

Det är vad vi anser viktigt att stärka studenternas självkänsla och tro på sin kapacitet att praktisera de kunskaper som lärts ut. Därför är det problematiskt att skapa situationer där studenterna kör fast och inte känner att de ens vet vad de ska börja när de ställs inför en uppgift. Programmering är av sin natur en färdighet i att lösa generella problem med hjälp av ett programmeringsspråk. En styrka hos datorer är dessas förmåga att utföra matematiska operationer mycket snabbt. Därför är det naturligt att många av de problem som löses med datorer mer eller mindre är av matematisk natur.

Emellertid så är en kurs i programmering mycket fokuserad på regler för hur kod skrivs i ett programmeringsspråk. Dessa regler för hur man uttrycker sig i programmeringsspråket, förklaras och beskrivs vidare medelst en stor uppsättning programmeringstermer. Det gör att studenter som aldrig tidigare programmerat ställs inför två begreppsvärldar. Dels en begreppsvärld för att beskriva typiska programmeringstekniker och koncept. Dels en begreppsvärld för hur dessa tekniker och koncept uttrycks i ett specifikt programmeringsspråk.

Först när man nått så långt i lärprocessen att man behärskar ett visst programmeringskoncept och hur detta koncept uttrycks i det språk som används i undervisningen, kan man använda konceptet för att lösa ett verkligt problem. För att få vana och känna trygghet i denna färdighet måste detta naturligtvis övas i stor omfattning.

Vad som händer om man ger en uppgift som matematiskt eller logiskt är för komplicerad för studenterna, är att fokus förskjuts från de programmeringskoncept som skall övas in. I stället läggs fokus på att förstå vad uppgiften går ut på och hur detta i allmänhet ska lösas, oaktat om man använder datorer eller papper och penna för att lösa problemet. Kör studenterna fast eller misslyckas att förstå problemet eller/och hur det kan lösas, skapas i värsta fall en känsla av att programmering är något mycket matematiskt tillika konceptuellt svårt, och att studenternas egen förmåga inte räcker till. Av dessa skäl anser vi att man noga bör överväga vilken nivå av svårighetsgrad de problem som skall lösas med programmering bör ligga på[9].

En annan aspekt som lett oss fram till denna princip är att det måste finnas en relation mellan vad man lär ut och vad man förväntar sig att studenterna ska klara av att utföra. Om lejonparten av kursen läggs vid rena programmeringskoncept och programmeringsspråkliga konstruktioner så bör uppgifterna reflektera detta. Om undervisningen inte omfattar matematik och

[8] Denna princip torde vara tillämplig på alla nivåer av programmeringskurser

[9] Notera att detta tarvar en hel del tänkande och diskussioner. Vi har noterat att många böcker och lärare ofta tar till problem som är enligt oss på tok för komplicerade, med förevändningen att “De är klassiska programmeringsproblem”.

problemlösning, så kan man inte förvänta sig att detta är något som studenterna skall klara av eller lära sig själva[10].

Stor tid och mycket reflektion bör därför under kursens konstruktion ägnas åt att slå fast förkunskapskrav vad gäller matematik och kompetens i problemlösning. Antar man studenter utan särskilda krav på matematik och problemlösningsförmåga så måste detta faktoriseras in i utformningen av kursen. I den mån matematik och problemlösning ändå förväntas ingå i lärmålen för kursen måste detta också läras ut och tränas under kursen på ett sätt så att förkunskapskraven matchar vad man förväntas ha med sig för att nå kursens lärmål.

Ovanstående är också viktigt för att studenterna skall ha rätt förväntningar av kursen utifrån vad de har med sig i form av kunskaper och färdigheter. Uppfyller man formellt kraven för att antas som student på kursen förväntar man sig inte att moment i kursen skall ställa krav på ytterligare (för)kunskaper.

Särskilt viktigt är detta vid examinering av studenternas kunskaper och färdigheter vid slutet av kursen. Det bör finnas ett ett-till-ett-förhållande mellan vad som examineras och vad som lärts ut och tränats på under kursen, med reservation för att vissa förkunskaper krävs utöver vad som lärs ut under kursen.

Genom att gå igenom denna process vid såväl skapandet av kursen som skapandet av examinationen av kursen, så kan man också transparent och öppet motivera övningar och examensfrågor på ett sätt som föregriper kritik och en känsla av att nivån på övningar och prov var för svår i förhållande till de förväntningar som studenten skaffat sig utifrån kursplan och antagningskrav. Denna ansträngning och analys ger värdefull indata till hur förkunskapskrav skall sättas, vilka färdigheter som skall läras ut och övas samt hur övningsuppgifter samt prov skall utformas[11].

§ 10. Undvik motsägelser (egna och mellan kollegor)

Förberedelser för föreläsningar och instuderingstexter kräver en kollegial diskussion och samsyn. I de fall man beslutar sig för att förenkla eller delförklara något avancerat är det viktigt att detta sker på ett strukturerat och genomtänkt vis. Dokumentera och sprid sådana beslut i lärarkollegiet för att undvika att skapa motsägelser. Var noga med att för studenterna understryka när en förenkling används så att studenterna dels har en möjlighet till fördjupning, dels har en möjlighet att kräva att en utlovad fördjupning också kommer längre fram.

En förenkling bör formuleras och utformas på ett med verkligheten överensstämmande sätt så att det inte tummas för mycket på sanningen. Det är med andra ord viktigt att skilja på en förenkling och en liknelse som inte riktigt håller eller stämmer överens med fakta. Risken är

[10] Här är kursplanerna, och i synnerhet kunskapsmålen i dessa, till stor hjälp för vägledning. Som vanligt gäller att dåligt genomtänkta eller/och formulerade kursplaner dock inte är till något stod för lärarna.
[11] Notera här att vi förespråkar något så radikalt som en återkopplingsloop från kursutveckling till kursplansförfattande.

annars att man genom en förenklad bild av en komplex verklighet längre fram tvingas ge en förklaring som går stick i stäv med förenklingen. Detta gäller i lika stor grad för en enskild lärare som mellan lärarkollegor.

Om man tummar på sanningen i för stor utsträckning i syfte att förenkla ett koncept är risken stor att studenterna förr eller senare genomskådar detta och förlorar förtroende för ämnet såväl som för utbildningen. Därför är det viktigt att man som arbetslag är öppen med hur man presenterar förenklingar och koncept så att man inte ger två sinsemellan motsägelsefulla beskrivningar, definitioner eller förklaringsmodeller. Fördelen med denna lärardiskussion är att man också kan hjälpas åt med att finna en rimlig nivå på förenklingar samtidigt som man får förståelse för hur studenterna fått något förklarat för sig av kollegorna[12].

Detta gäller också vid val av kurslitteratur. Finner man problem med förklaringar i kurslitteraturen måste detta hanteras på något vis. Antingen genom att man anpassar sin egen förklaring till att inte i för stor omfattning stå i kontrast till förklaringar som ges i litteraturen. Eller att man kommunicerar till studenterna att vissa kapitel eller avsnitt i boken inte är kompletta och att man då i stället erbjuder eget eller annat kompletterande material.

Är det för många problem med litteraturens förhållningssätt till den egna pedagogiken eller rent av sanningen bör man naturligtvis kraftigt överväga att byta litteratur till kursen.

§ 11. Identifiera och hantera motsägelser

Ibland kan situationer trots princip § 10 ändå uppstå på grund av slarv, externa källor eller rena fel i övningar, litteratur eller andra källor. Det är då viktigt att dokumentera detta och föra in det i den iterativa processen med ständig förbättring av kursen. Dessutom bör man ta ett steg tillbaka och lyfta detta med studenterna. Det kan öppna för intressanta diskussioner och reflektioner tillsammans med studenterna. Man tar ett kliv tillbaka och diskuterar ämnet utifrån tillsammans med studenterna och frågar sig “Hur kunde det bli så här?” “Vad tror ni att man avsåg?” och så vidare.

Om man upptäcker en motsägelse mellan en kollega och den egna förklaringsmodellen så bör detta lyftas kollegialt. Det är inte viktigt att slå fast vilken förklaring som är bäst eller mest korrekt. Fokus bör däremot ligga på att hantera det upptäckta och enas om en åtgärd. Samma sak gäller om man upptäcker ett problem mellan föreläsningar och kurslitteratur. Lyft detta problem så fort som möjligt mellan kollegor och besluta om detta måste åtgärdas. Är det en allvarlig diskrepans mellan vad litteraturen säger och vad lärarna lär ut så måste detta förr eller senare hanteras.

[12] Det är inte heller angenämt att hamna i en situation där man tvingas välja mellan att säga emot en kollega eller acceptera en halvsanning. Genom att ha en prestigelös kollegial diskussion kan man lära av varandra samtidigt som man förebygger motsägelser.

§ 12. Använd konsekvent terminologi och förklaringsmodell

Ett sätt att uppfylla principerna §§ 10,11 är att vid konstruktionen av en kurs eller en iteration av kursförbättring skapa en term- och konceptkatalog och dokumentera denna. Ofta finns det två eller fler termer för samma sak inom programmeringsnomenklaturen. Dokumentera detta och bestäm vilken term som skall användas som huvudterm. Förankra detta mellan kollegor och undersök vilka termer kurslitteraturen begagnar sig av. Kommer man fram till att flera termer för samma sak är befogat att lära ut och använda omväxlande måste detta också dokumenteras och framförallt kommuniceras till studenterna.

Ju mer stringent term-, koncept- och förklaringskatalog man lyckas ta fram, desto färre problem med motsägelser och vagheter utsätts studenterna för.

§ 13. Använd konsekventa symboler och namn

Studenter som är noviser inom programmering söker efter mönster i exempel och förklaringar. Det är därför en stor fördel om litteratur och kolleger använder samma stil och symboler när exempel ges till studenterna. Det gäller såväl exempel i texter och litteratur som i föreläsningar, exempel på svarta tavlan och i presentationer, live-programmering och i övningsuppgifter och prov.

De allra flesta programmeringsspråk har någon form av rekommendationer och kodstandarder. Identifiera eller välj ut en och försök att förankra och följa den mellan kolleger och kurslitteratur. Syftet är som sagt att underlätta för studenterna att finna mönster och nycklar för kod som de utsätts för. Det minskar också otydlighet och motsägelser om man har ett stringent sätt att presentera kod på.

§ 14. Visa inkonsekvens endast när syftet är att något kan varieras

Det finns fall där inkonsekvens har en pedagogisk poäng. Det är när syftet med ett exempel är att visa att det finns mer än ett sätt att uttrycka sig på. Försök att undvika att lära ut detta som en bieffekt av att alla exempel ser olika ut och använder olika stil. Sträva hellre efter att ha en genomtänkt och konsekvent stil och utformning av era program- och kodexempel tills att själva syftet med en genomgång är att visa på flera sätt att uttrycka en viss konstruktion.

§ 15. Egna modeller bör motiveras (och inte examineras)

Det är inte fel att skapa egna modeller för att förklara en dator eller programmering. Man kan tänka sig att man inleder med ett påhittat programmeringsspråk eller ett påhittat symbolspråk för att beskriva koncept inom programmeringen. Detta bör då dokumenteras för kommande kursförbättringsiterationer. Dessutom är det viktigt att kommunicera till studenterna att den egna modellen är skapad av er och inte är en del av lärmålen per se.

Det är också föga meningsfullt att examinera kunskaper i en egenutvecklad modell som blott har ett pedagogiskt syfte. Det är viktigt för studenterna att förstå vad som är ämnets faktiska beståndsdelar och vad som är rena pedagogiska verktyg och förklaringsmodeller. Risken är

annars att studenterna får en känsla för att ett symbolspråk eller liknande är en faktisk del av programmering.

I den mån man ändå tycker att en modell eller ett pseudospråk är en viktig färdighet som studenterna faktiskt skall behärska i kommande kurser och i programmeringssituationer efter avslutade studier, så bör naturligtvis sådana modeller eller pseudospråk definieras, specificeras och dokumenteras så att de går att studera och använda som instuderingsmaterial. I dessa fall måste man naturligtvis också ge ut övningar i att använda modellerna och samtidigt erbjuda handledning för att lösa övningarna.

Vi anser dock att sådana egna eller delvis egna modeller snarast skall tjäna som ett pedagogiskt verktyg under en introduktionskurs i programmering. Risken är annars att studenterna upplever att de skall lära sig mer än ett programmeringsspråk vilket kan upplevas som mer än nog.

§ 16. Parföreläsningar är en styrka

En metod som vi provat och använt oss av är att uppträda i par som lärare under såväl traditionella föreläsningar som i videoföreläsningar. Vår övertygelse är att detta är en styrka som, trots att den faktiska ekonomiska kostnaden är dubbel för själva undervisningstimmarna, betalar sig i längden.

Vad vi har identifierat är att två lärare skapar en dynamik där 1 + 1 är större än 2. På ett plan så skapar det dynamik i det att slarvfel av den ena läraren snabbt kan upptäckas och korrigeras av den andre. På ett annat plan så skapas situationer där den ena läraren kan ta på sig rollen att “ifrågasätta” eller komplettera det den andra säger, på ett lekfullt sätt. Om man involverar studenterna i diskussionen lärarna sinsemellan uppstår många oväntade reflektioner där studenterna bjuds in i samtalet.

Vad gäller videoföreläsningar så fungerar denna princip väl även här. Det kan vara tröttsamt att höra en röst genom en hel föreläsning. Två röster skapar en dynamik som mer påminner om en radiosituation där ett ämne snarare diskuteras än avhandlas. Här kan den ene läraren vara den som ställer frågor och “inte förstår”.

§ 17. Envägsföreläsningar är filmer

Denna princip handlar helt enkelt om att vissa föreläsningar har en tämligen statisk karaktär där fakta staplas på varandra. Det kan inom programmering handla om att lista en rad besläktade koncept och termer. Eller om ett längre exempel som gås igenom från början till slut. Om föreläsningen inte är av en natur som bjuder in studenterna i en dialog eller diskussion, så passar föreläsningen bättre som videofilm.

Vi har arbetat med att publicera föreläsningar som videofilmer som vi kräver att studenterna skall ha sett innan den faktiska föreläsningen. När föreläsningen hålls så kan vi förutsätta att eleverna sett filmen eller filmerna som går igenom ämnet för dagens föreläsning och kan under

föreläsningstimmarna fokusera på frågor, reflektioner och fördjupningar av det som tagits upp i filmerna.

Vår tanke med detta är att det är slöseri med tid att kräva att studenterna samlas under två timmar för att lyssna på något som de lika gärna kan tillgodogöra sig genom andra medier såsom video. På detta vis kan vi utnyttja föreläsningstillfället till aktiviteter som faktiskt kräver att alla studenter är samlade såsom frågestunder och reflekterande dialog om programmering i allmänhet och i synnerhet det avsnitt som togs upp i föreläsningsfilmerna inför dagen.

Svagheten med denna princip är naturligtvis att den bygger på att studenterna verkligen sett filmerna inför föreläsningen för att diskussionen och frågestunden skall vara givande. Å andra sidan så finns filmerna kvar och kan användas för repetition, vilket inte går att säga om en fysisk föreläsning som inte spelats in.

§ 18 Fokusera ett avsnitt på ett koncept åt gången

Det är viktigt att studenterna får klart för sig vilket koncept ett moment eller ett exempel handlar om. Därför bör det vara transparent och tydligt vad som är huvudfokus för en föreläsning eller en föreläsningsdel. Undvik att tala om flera saker samtidigt för att minimera förvirring hos studenterna. Det gör det också lättare att identifiera vad som är fel med en föreläsning eller ett exempel. Om studenterna har problem med koncept X, så ska det vara lätt att hitta exempel och föreläsningsmaterial som tog upp och fokuserade på X, vilket blir mycket svårare om man blandar flera koncept i samma föreläsning och exempel.

Naturligtvis är det svårt att undvika att endast ta upp ett koncept isolerat. Denna princip handlar om att göra det tydligt för studenten vad som är fokus för exemplet eller föreläsningen. Inom programmering så är det svårt att isolera ett koncept eftersom det finns så många beroenden mellan koncepten. Lyft fram vad som är huvudfokus innan och under föreläsningen och markera gärna typografiskt det som är fokus.

Undvik att beskriva huvudfokus för föreläsningen i termer av eller med hjälp av andra koncept som också är relativt obekanta för studenterna.

Separera praktik från teori

När du går igenom t ex en while-loop[13], gör då detta utifrån ett praktiskt perspektiv. Detta för att påvisa applikationer för en while-loop och syntaxen för densamma. Vanligtvis är studenterna är inte vid introduktion av ett koncept redo att reflektera över det. Ett koncept måste förstås rent praktiskt innan reflektion kan göras om exempelvis komplexitet eller effektivitet.

Anpassa reflektioner efter behov och kunskap

Hur mycket reflektion som behövs vid ett givet tillfälle är svårt att avgöra, men vi bör sträva efter att inte reflektera för djupt till en början. Man kan lägga lite mer djupa reflektioner som ett extramoment eller tydligt visa för studenterna att denna reflektion inte kommer att examineras.

[13] En konstruktion inom imperativ programmering för upprepning av instruktioner.

§ 19 Relevanta och motiverande exempel och övningar

Undvik krystade hitta på-exempel. Motivera till fortsatt studerande, kittla nyfikenheten.

§ 20 Balans mellan generella koncept och specifika tillämpningar

Här ger vi helt enkelt två konkreta exempel på vår bedömning av vad som är generellt och vad som är specifikt:

I/O: generell - principer för in- och utmatning är generella

Scanner/Format/printf: specifika - dessa är blott exempel på verktyg för I/O

Vi tror att det är mycket värt att ägna lite tankekraft åt hur balansen mellan det generellt användbara och det specifikt tillämpade.

§ 21 Låt inte ett språks egenskaper färga material om ett annat språk

Om du undervisar i Java, vilket är ett objektorienterat språk, är det viktigt att allt material utgår ifrån objektorienterade problem. Objektorienterade språk är bra på att lösa en viss typ av problem, andra programmeringsmetoder är bra på andra saker.

Speciellt viktigt är detta om man byter språk från t ex ett procedurellt språk till ett objektorienterat språk i en kurs. Försök då undvika att “ta med” ett sätt att tänka och exempel från det tidigare språket till det senare språket, annat än för att kontrastera.

§ 22 Efterfråga feedback och hantera den på ett adekvat sätt

Efter varje kurs görs en utvärdering. Detta är ett bra redskap om utvärderingen görs på ett bra sätt och hanteras därefter. Men det är en ganska långsam process då resultatet av utvärderingen inte blir nytta för studenten.

Om man, som komplement till kursutvärdering, ber studenterna om feedback under kursens gång kan man snabbt anpassa undervisningen efter studenternas behov. Självklart måste en sådan anpassning ske inom rimliga gränser.

Vi vill trycka på denna princip då den snabbare förbättrar en kurs, ger studenterna bättre inflytande över sina studier, vilket i sig verkar motiverande, samt att det minskar risken att få negativ kritik i den avslutande officiella kursutvärderingen. Det kan här vara användbart att använda sig av konstruktioner i likhet med “klassrepresentanter” och gärna någon extern part såsom program- eller utbildningsansvarig, för att skänka legitimitet till utvärderingen samt för att få extern feedback på kursen medan kursen är igång.

Stegrande övningar, bra och väldokumenterade svar

Det finns tillfällen då man upptäcker eller inser att en övning var alldeles för specifik eller trivial för att vara användbar för studenterna i deras lärande och förståelse för programmering som verksamhet. Utnyttja dessa upptäckter för att förfina och förbättra övningarna, är vår rekommendation.

Till exempel så kan man behålla den simplistiska övningen och lägga på lager av förfining och utveckling. Genom att börja med något enkelt (och specifikt), så finns möjligheten att fånga studenternas uppmärksamhet och behålla den. Lägg på förfiningar och ökande svårighetsgrad och låt studenten bygga ut och förfina lösningen och få den giltig i fler (mer generella situationer).

Ett av våra exempel bygger på en enkel övning som går ut på att programmatiskt finna antalet förekomster av en viss bokstav i en given textsträng. Detta är ju knappast något som spontant lockar till programmering (särskilt inte om textsträngen är ett normallångt ord där man okulärt direkt kan se och räkna antalet förekomster av bokstaven utan att ens starta datorn).

Denna övning utgör däremot en perfekt grundplåt för att väcka en diskussion och reflektion kring vad programmering är och vilka problem som låter sig lösas medelst programmering.

Varför inte bygga vidare på en sådan övning och stegvis göra den mer programmeringsmässig?

En tänkt progression kan kvara:

- Skriv ett program som kan räkna antalet “o” i “Liverpool”
- Skriv ett program som anropar en funktion som räknar antalet “o” i “Liverpool”
- Skriv om funktionen så att den som anropar den anger vilken bokstav (vilket tecken) i “Liverpool” som skall räknas
- Skriv om funktionen så att den som anropar anger såväl tecken som det “ord” som skall undersökas
- Skapa ett programbibliotek där funktionen ingår, så att funktionen kan användas i flera framtida program

Avsluta uppgiften med en reflektionsuppgift: *Vad var problemet med det ursprungliga programmet? Hur brukar program se ut? Vilken funktion fyller funktioner? Vad är tecken på att en funktion kan göras mer generell?*

Kodförståelse

En kollega sade (ungefär) “Studenterna borde tränas mer i att läsa och förstå kod innan vi kräver att de ska kunna skriva egen kod. Som barn lär man ju sig att läsa innan man lär sig att skriva och det finns skäl för det”. Kanske borde vi inleda varje övningsområde med ren kodförståelse. Ett par inledande övningar med kod från området som är fokus för övningarna, där uppgifterna går ut på att läsa och förstå kod som handlar om eller använder det nya konceptet. Detta är något som vi enbart delvis experimenterat med och inte har tillräckliga data för att dra några slutsatser om. Vi välkomnar naturligtvis läsarnas egna reflektioner och erfarenheter.