Jan Boddum Larsen
HTX Næstved

SOFTWARE DOKUMENTATION

TEKNOLOGI B OG A PÅ HTX

Indhold

Dokumentation af software i Teknologi på HTX 2
Overblik 2
    Kravspecifikation 2
    Blokdiagram 3
    Use Case Diagram 3
    Pseudokode 4
Dokumentation af koden 5
    Flowdiagram 5
    Tilstandsdiagram 7
        Design af tilstandsdiagrammer 7
    Klassediagram 8
    Kommentarer i koden 11
Test 12
Anbefalinger til gode arbejdsmetoder 12
Digitale værktøjer 13
Konklusion 13

Dokumentation af software i Teknologi på HTX

I samfundet sker der en bevægelse mod mere digitale løsninger i teknologi. Det betyder, at software bliver en større og større del af teknologien. Det kan være som dele af teknologien, men også som selvstændige digitale teknologier.

Dette gør, at der et øget behov for at indføre det i ungdomsuddannelserne og i undervisningen. Det har bl.a. betydet, at software er kommet ind i faget teknologi på HTX. Det bevirker, at der er skabt et behov for at kunne udvikle software, og videre at kunne dokumentere og kvalitetssikre softwareprodukter. Det er dokumentation og test af softwareprodukter, der er i fokus i dette dokument.

Dokumentet er struktureret med hovedafsnit, der behandler overblik over softwareproduktet, konkret dokumentation af koden og principper for test. Der afsluttes med anbefalinger om arbejdsmetoder, digitale værktøjer og en konklusion.

Overblik

Der findes forskellige metoder og værktøjer til at danne overblik over et softwareprodukt; men man skal først tænke i krav og test.

Kravspecifikation

Specifikke krav til softwaren, hvor kravene efterfølgende kan måles/testes, er vigtige at få stillet op. Det kan også være nødvendigt i denne fase at åbne lidt op for softwaren for senere at kunne teste de krav, man har til softwaren. Så for hvert eneste krav skal man efterfølgende kunne teste, om kravet er opfyldt.

Et eksempel kunne være, at man ønskede at udvikle en App, hvor man kan få en alarm X minutter før en bus kører. Ideen er, at når man stiger af bussen, scanner man en QR-kode og App'en finder ud af, hvornår den sidste bus kører og giver en alarm 10 min, før den kører. Et krav kunne være som angivet i venstre kolonne, og testproceduren i højre kolonne. Se Figur 1.

Krav	**Test**
Det skal være muligt at angive den tid der går fra alarmen til den sidste bus kører.	#1 Indtast ”10 min”. Se at der kommer en alarm 10 min før sidste bus. #2 Indtast ”-5” og se at der kommer en fejlbesked: ”Ugyldig tid, den skal være mellem 1 min til 20 min” #3 Indtast ”30 min” og se der kommer en fejlbesked: ”Ugyldig tid den skal være mellem 1 min til 20 min” #4 Vælg en bus, hvor den sidste kører efter middag og gentag test #1. Osv

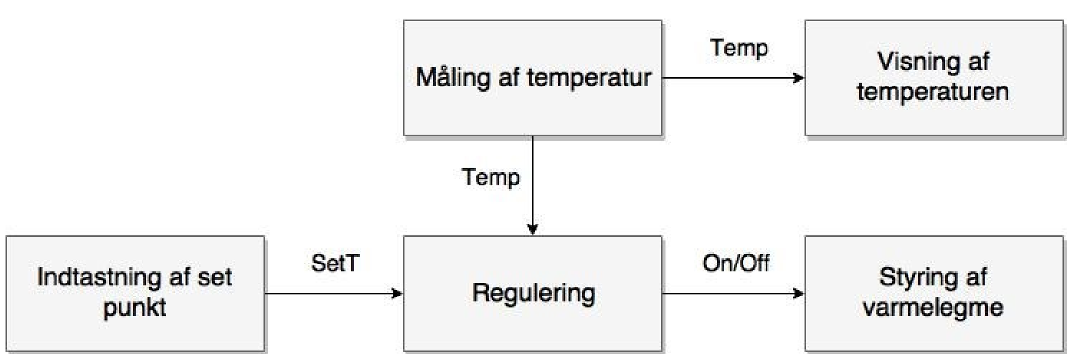
Figur 1 Krav og tilhørende tests

Blokdiagram

Benyt et blokdiagram til at give et overordnet overblik og til at beskrive de enkelte blokke og grænseflader for softwareproduktet. Da det skal skabe overblik, er det vigtigt, at der ikke er for mange blokke; så max omkring 5 til 6 blokke. Hvis man har behov for flere, så heller lave to eller flere blokdiagrammer og beskriv grænsefladerne mellem de enkelte blokdiagrammer.

Nedenfor i Figur 2 er der angivet et blokdiagram over en temperaturregulering, f.eks. til en simpel el-radiator. Der er pile mellem de forskellige blokke og lidt tekst til, hvad der sker i de enkelte blokke og hvilke data, der flyder fra den ene blok til den anden.

Her ville man også kunne stille krav og test til de enkelte blokke i stil med bus-eksemplet i Figur 1, så man har veldefinerede grænseflader.



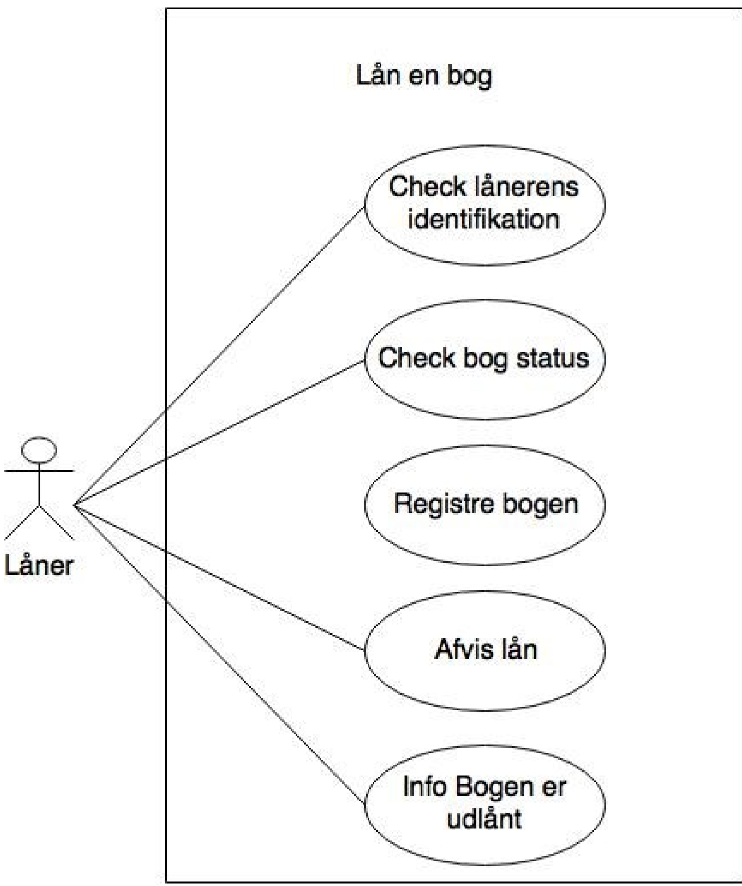
Figur 2 Blokdiagram over en temperaturregulering

Use Case Diagram

Et tredje værktøj til at give et overordnet blik på softwaren ville være use case diagram. Det er noget der bruges til at fortælle, hvordan man tænker, at de forskellige brugere skal benytte softwaren.

I Unified Modeling Language (UML) er der en fin standard, som man kan udvide mere end i eksemplet nedenfor i Figur 3. Eksemplet er et softwaresystem til udlån af bøger på biblioteket. Der er kun medtaget låneren, for at holde det simpelt.

Det ses, at der er nogle processer låneren er i berøring med, og så er der nogle processer, der kommunikerer internt. Dette kan være med til at beskrive hvordan systemet og brugeren kommunikerer sammen.



Figur 3 Use Case Diagram over hvordan en låner benytter software ifm. med lån af en bog

Pseudokode

Pseudokoden er en måde at beskrive det, man ønsker at programmere, i almindelige ord. Hvis vi igen tager udgangspunkt i eksemplet med temperaturreguleringen oppe fra blokdiagrammet, jfr. Figur 2, kan vi benytte pseudokode til at få et overblik over softwaren på forskellige detaljeringsniveauer, se Figur 4.

1. Metode (overordnet)	2. Metode (mere detaljeret)
Uendeligt loop (Mål temperatur Tænd/sluk varmelegemet Vis temperatur)	Uendeligt loop (Mål temperatur Er temperaturen under set-temperaturen? Tænd varmelegemet Ellers Sluk varmelegemet Vis temperatur)

Figur 4 Pseudokode over temperaturregulering af en el-radiator, overordnet og mere detaljeret

Pseudokode kan både benyttes som dokumentation, men også som et værktøj under udviklingen. På det mere detaljerede niveau kan pseudokode i nogle tilfælde nemt omsættes til egentlig programkode.

Ovenstående fire metoder kan anvendes til at give et overblik over den software, man har udviklet og ønsker at dokumentere. Det vil samtidig være naturligt at lade metoderne indgå i selve

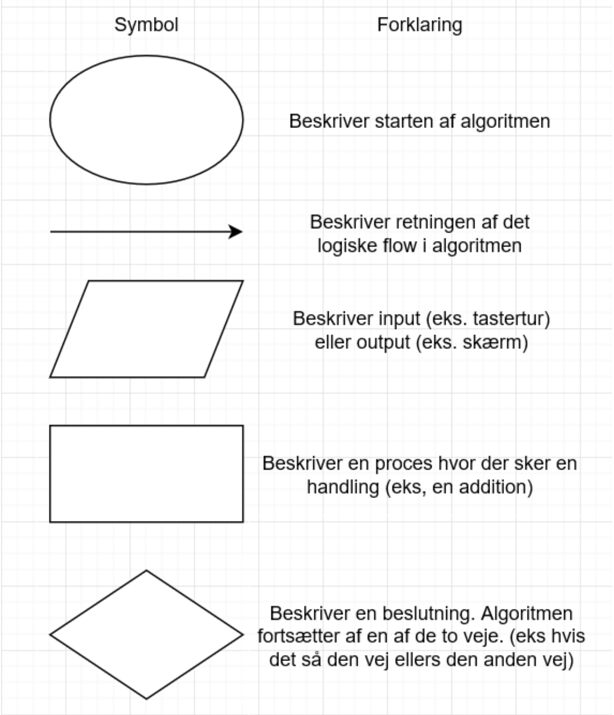
udviklingsprocessen. Det vil være en god idé, at have nogle af disse dokumentationskrav og -metoder klar, inden eleverne kommer i softwareværkstedet.

Dokumentation af koden

Ligesom under hovedafsnittet med metoder, der skaber overblik, er der i dette hovedafsnit med fokus på den mere konkrete dokumentation af koden også flere metoder, der kan benyttes. Der kan selvfølgelig benyttes flere af metoderne i det samme softwareprojekt. Bl.a. vil kommentarer i koden næsten altid være relevant.

Flowdiagram

Flowdiagram eller flowchart er en gammel metode til at dokumentere et program eller en del af et program. Det virker rigtig godt til at se forgreninger, input, output og start/stop. Så det beskriver flowet i programmet. Der er en række symboler, som er beskrevet nedenfor i Figur 5:



Figur 5 Almindeligt anvendte symboler i flowchart

Flowchart kan omsættes direkte til kode, og er derfor meget anvendeligt.

Tegneregler:

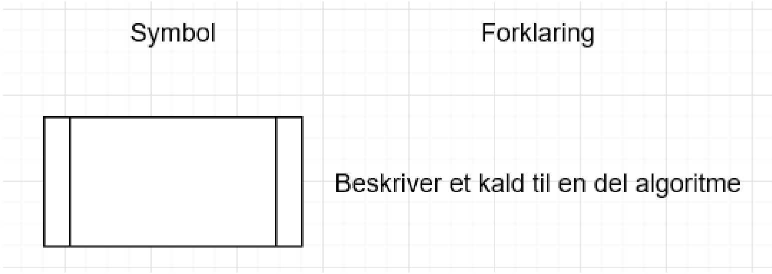
- Streger tegnes som rette linjer med pil.
- Kun en indgang til hvert symbol
- Kun beslutningssymbolet - diamanten - har flere udgange.

Nedenfor i Figur 6 er angivet et simpelt eksempel, hvor man ønsker at beregne summen af heltal fra 1 til og med N. Dette gør programmet ved successivt at lægge det næste tal i rækken til den foreløbige sum, indtil alle tallene er lagt sammen. På den måde beregnes f.eks. summen af tal fra 1 til og med 10 således til 55.

Flowchart	
Start	Her starter programmet.
Læs N	Her indlæses et tal N, eksempelvis fra et tastatur.
m=1 sum=0	Variablene initialiseres, så vi ved hvad værdier de har i begyndelsen.
sum=sum+m	m (som er 1 første gang) lægges til variablen sum og gemmes i sum.
Er m=N (Nej → m=m+1)	Beslutning: Hvis m er nået op på det indtastede tal N, så gå ned til næste trin. Hvis den ikke er kommet op på N, læg 1 til m og gem den i m og gentag.
Skriv sum	Udskriv sum til et display.
Slut	Her slutter programmet.

Figur 6 Eksempel på flowchart, hvor summen af tal fra 1 til og med N beregnes.

Det ses, at det kunne blive nogle meget lange flowcharts, hvis hele algoritmen er meget lang. Så man kan evt. begrænse det til de passager i koden, man gerne vil fremvise, eller som har en kompleks struktur. Hvis algoritmen er for kompleks, kan man vælge at nedbryde den i forskellige delalgoritmer. Se symbolet for kald af delalgoritme nedenfor i Figur 7 .



Figur 7 Symbol for kald af delalgoritme

Det er også en god ide at benytte symbolet, når man gentager den samme ting flere steder i algoritmen.

Tilstandsdiagram

Tilstandsdiagrammer er gode til at beskrive forskellige tilstande, et program kan være i (venter det eksempel på input?). Det gode ved tilstandsdiagrammer er, at de giver en mulighed for at se tilstande i programmet, som kan være lovlige. Man kan sige, at det er en tabel over alle tilstande, og hvad programmet skal gøre for at flytte fra en tilstand til en ny tilstand.

Design af tilstandsdiagrammer

- Blokkene kaldes tilstande (states)
- Forbindelserne kaldes overgange (transitions)
- Der skal altid angives hvilket signal, der får enheden til at skifte tilstand
- Tilstandene svarer til udgangskombinationer
- Overgangene svarer til indgangskombinationer.

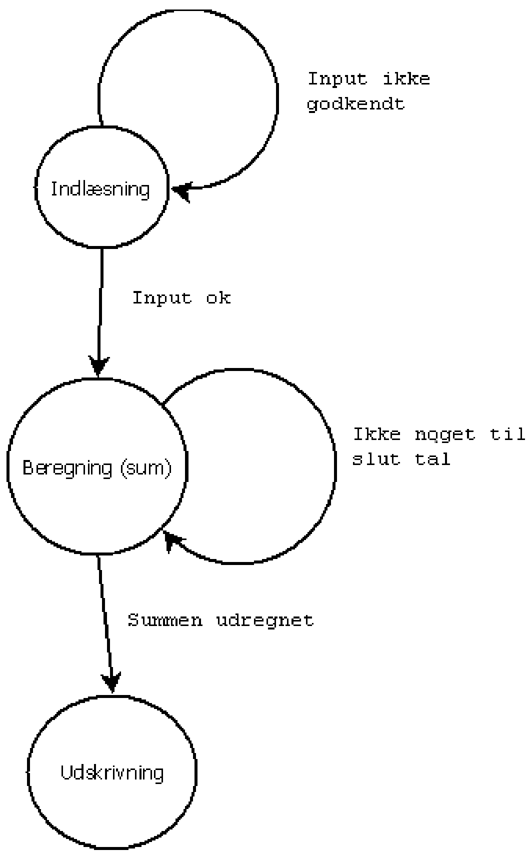
Det gode ved tilstandsdiagrammer er, at man sikrer sig, at programmet kun kan hoppe mellem de forskellige tilstande, og alle andre er ulovlige tilstande.

Tilstande tegnes normalt som cirkler (tilstande) med pile (hændelser) imellem, hvor der skal komme en hændelse, før man ændrer tilstanden. Det kan være en robot, der følger en streg og kører lige ud; der er en tilstand. En sensor aflæser, at robotten ikke er på stregen mere: Så er der sket en hændelse, og robotten ændrer tilstand til at skulle dreje.

Det betyder også, at tilstandsdiagrammer kan vise noget andet end flowcharts.

Eksemplet nedenfor er det samme som oppe under flowdiagrammet ovenfor, hvor der skal lægges nogle tal sammen.

Der er i alt 3 forskellige tilstande (indlæsning af tal, beregning, udskrivning af tal)



Figur 8 Tilstandsdiagram over beregning af sum af tal

Ligesom med flowcharts giver det mest mening at vælge noget ud fra ens kode og beskrive disse elementer med tilstandsdiagrammer. Det er bedst at tage de steder, der er lidt mere komplekse.

Klassediagram

Det kan være ret svært at beskrive eksempelvis noget, der er lavet i programmet App-inventor - altså med objekter - i et flowdiagram. Men derimod giver det god mening at beskrive en metode i et objekt med et flowdiagram.

I mange softwareudviklingsværktøjer er der en grafisk overflade; de er ofte objektorienteret (det kendes fra App-inventor). Her vil klassediagrammer være meget anvendelige, så man kan beskrive de enkelte metoder og attributter. En klasse består af en metode der udfører en handling (det kan eksempelvis den handling der sker når man trykker på en knap). En attribut kan eksempelvis være teksten på knappen eller farven på knappen. Jeg har valgt at tage udgangspunkt i en UML-beskrivelse af klasser.

Så en klasse består af attributter og metoder. Attributter er normalt private, hvilket vil sige at de kun ændres gennem en metode. Dvs., at ændringen af navnet på knappen skal ændres gennem en metode.

Nedenfor er der vist et eksempel med dyr, hvor den øverste del er attributterne, her f.eks. navnet på dyret. Det neden under stregen i midten er metoder. Der er altid en metode til at skabe dyret, i dette tilfælde metoden ”Dyr()”. Herefter kan man benytte metoden ”setNavn()” til at give dyret et navn, eksempelvis en ko ”setNavn(’ko’)” . Det betyder i dette tilfælde, at attributten ”navn” bliver sat til ko.

Ideen er at lave en tegning som nedenfor på Figur 10 og beskrive, hvad de forskellige metoder gør. Eksempelvis vil man kunne beskrive metoden ”setNavn()” som på Figur 9 ved:

setNavn(nyNavn : String)

Beskrivelse	Sætter navnet på et dyr eller ændrer navnet.
Input nyNavn	 Tekststring med navnet på dyret
Output	Na
Sideeffekt	Na

Figur 9 Beskrivelse af metoden "setNavn()"

Output er hvis der kommer noget tilbage/retur fra metoden, som eksempelvis i metoden ”getNavn() : String”, som giver navnet på dyret tilbage.

Sideeffekt kan være, at den sender en besked via bluetooth eller åbner et nyt window.

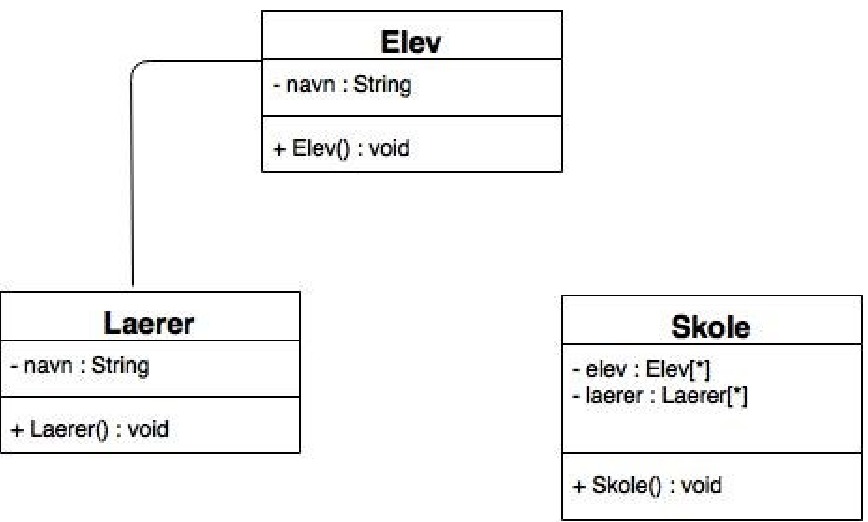
Dyr

- navn : String
- hojde: int
- vaegt : int
- farvoritMad : String
- hastighed : double

+ Dyr() : void
+ setNavn(nytNavn : String) : void
+ getNavn() : String
+ spise():void
+ maxAfstand(tid : int) : int

Figur 10 En klasse "Dyr" med tilhørende attributter (øverst) og metoder (nederst)

Nogle klasser er koblet til hinanden. Det kunne være elev og lærer; men det er ikke de samme ting, man vil gemme. Det kalder man associated (tilknyttede) klasser. De fælles ting kunne man have i en overordnet klasse ”skole”, som både indeholder elever og lærere. Se nedenfor.

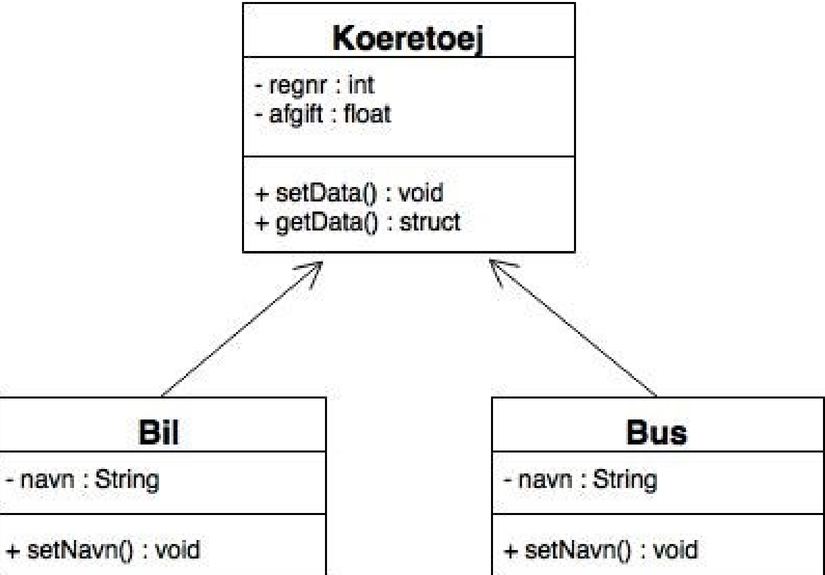


Figur 11 To tilknyttede klasser (”elev” og ”laerer”) til en overordnet klasse (”skole”)

Der en sammenhæng mellem ”laerer” og ”elev”, fordi en lærer altid har elever, ellers var han/hun ikke lærer på skolen. Dvs., man ville normalt have en skoleklasse, der knytter dem sammen, udover at de er på samme skole. Det er dog sjældent man laver denne forbindelse.

Arv er en måde til at en klasse kan arve ting fra en anden klasse:

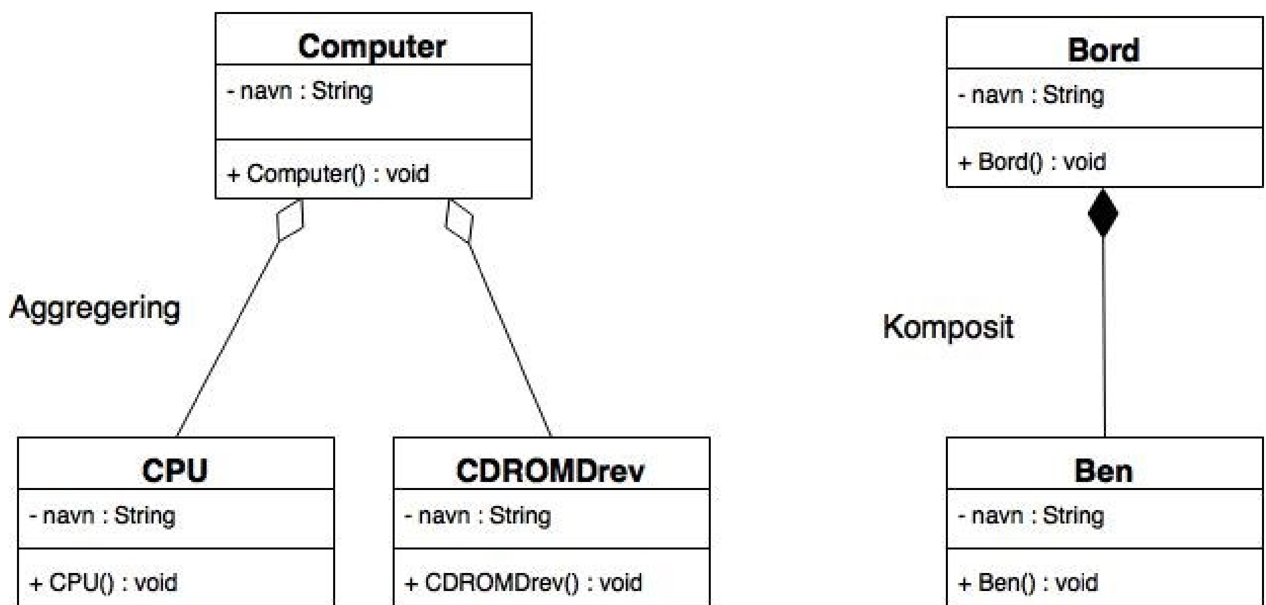
Så både bil og bus har et registreringsnr., så pilene indikerer at ”Bil” og ”Bus” arver fra ”Koeretoej”.



Figur 12 To klasser "Bil" og "Bus", der arver fra "Koeretoej"

Det efterfølgende er lidt mere avanceret og er måske ikke nødvendigvis et krav til dokumentation ved objektorienteret programmering. Det er udelukkende taget med p.gr.a helheden.

To andre typer forbindelser mellem klasser består af en åben diamant (aggregering) og en lukket diamant (komposit), se Figur 13. Komposit er lidt stærkere end aggregering, da et bord kun er et bord, hvis det har ben. Computeren kan godt leve uden CD ROM drev.



Figur 13 To typer af forbindelser mellem klasser: Aggregering og Komposit - henholdsvis åben og lukket diamantsymbol

Kommentarer i koden

Det er vigtigt at skrive kommentarer i koden, da det hjælper til at forstå, hvorledes den er opbygget. Alle dele af koden som f.eks. procedurer, funktioner, metoder og objekter bør have en samlet kommentar i toppen af kodeafsnittet. Kommentaren skal beskrive, hvad koden i afsnittet laver, inputs, outputs og sideeffekter. Kommentarerne kunne følge rådene vist i venstre kolonne, et konkret eksempel er givet i højre kolonne, se Figur 14:

Generelle råd til kommentarer i toppen	Eksempel med funktion til beregning af areal af cirkel

Beskrivelse:
”Hvad gør funktionen m.m.”
Input Variabler:
”Beskrivelse af alle input variablerne”
Output Variabler:
”Beskrivelse af alle output variablene”
Sideeffekter:
”Ændringer af globale variable, sender beskeder til andre enheder m.m”
	// Beskrivelse:
	// Beregner arealet af en cirkel.
	// Input:
	// float radius - Radius på cirklen.
	// Output:
	// float areal - Areal af cirklen.
	// Sideeffekter:
	// Ingen

Figur 14 En samlet kommentar i toppen af kodeafsnittet: Generelle råd og konkret eksempel

I selve koden er det også vigtigt at skrive kommentarer. Nedenfor i Figur 15 er angivet to eksempler på kommentarer i koden til en funktion, der beregner arealet af en cirkel. Begge eksempler har de samme kommentarer i toppen, næsten lige med tekst fra Figur 14.

<table>
<tr><td colspan="2">Eksempel: Funktion til beregning af areal af cirkel - kommentarer i toppen af koden og i selve koden</td></tr>
<tr><td colspan="2" align="center">Kommentarer i toppen af funktionen, fælles</td></tr>
<tr><td colspan="2">// Beskrivelse:
// Beregner arealet af en cirkel.
// Input:
//   float radius - Radius på cirklen.
// Output:
//   returnerer float areal - Areal af cirklen.
// Sideeffekter:
//   Ingen</td></tr>
<tr><td>Overfladiske kommentarer</td><td>Detaljerede kommentarer</td></tr>
<tr><td>float BeregnCirkelAreal(float radius)
{
  float pi=3.14;
  float areal;
  areal = pi*radius*radius;
  return areal;
}</td><td>float BeregnCirkelAreal(float radius)
{
  float pi=3.14; // definerer pi, tilnærmet da …
  float areal;
  areal = pi*radius*radius; // beregner arealet
  return areal;   // returnerer arealet af cirklen
}</td></tr>
</table>

Figur 15 Kommentarer i koden - overfladiske og detaljerede

Det er ikke således, at den ene kommentering er bedre end den anden. Det kan give god mening kun at skrive få kommentarer, hvis funktionen er forholdsvis simpel, som i eksemplet ovenfor i Figur 15.

Test

Der skal laves en testplan, som tester for alle kravene i kravspecifikationerne, samt evt. afhængigheder. Skriv ned om resultaterne af testen er succesfulde. Hvis der er test cases, som ikke lykkes, skriver man en kommentar om, hvad man har iagttaget. Dette gælder også ved begrænset succes.

Afhængigheder kunne være, hvis koden er i en bestemt tilstand. Hvis det er tilfældet, kunne man prøve at teste for ugyldige tilstande.

Softwaren skal testes af folk. der ikke har været med til at udvikle softwaren, så funktionalitet, layout og robusthed bliver kvalitetssikret.

Anbefalinger til gode arbejdsmetoder

En del af arbejdet med dokumentation af softwareprodukter i teknologifaget bør ses i sammenhæng med arbejdsprocessen i forbindelse med systematisk produktudvikling og dets faser i faget i det hele taget, hvor

dokumentationen ud over kvalitetssikring også har læringsformål og skal ses i sammenhæng med øvrige krav til rapportering. Derudover er der mere specifikke gode råd og anbefalinger om arbejdsprocesser mv. ifm. softwareudvikling, som følgende:

- Start med at dokumentere det overordnede, så læseren får et overblik
- Dokumentér så meget som det er muligt, inden du går i gang med koden
- Brug nogle af de værktøjer, der er gennemgået her, til at skabe overblik
- Lav kommentarer til koden, så den kan forstås af en nybegynder
- Lav den tilhørende dokumentation, så den kan forstås af en nybegynder
- Lav kommentar i koden, imens du skriver koden
- Husk at opdatere dokumentationen undervejs, og når du er færdig med koden
- Overvej hvilken struktur, koden har, og vælg den dokumentationsform, der passer til
- Som hovedregel har man altid lavet for lidt dokumentation.

Digitale værktøjer

Der er specielt ét gratis værktøj, som er nemt at gå til, og som samtidig kan køre i browseren. Her kan man tegne de fleste diagrammer:

https://www.draw.io/

Konklusion

Behovet for både dokumentation og kvalitetssikring af softwareprodukter i teknologifaget på HTX er blevet større de senere år, pga. en generel bevægelse mod mere digitale løsninger i samfundet generelt, en større interesse blandt eleverne i at arbejde med dette og endelig en ændring i fagene og deres indhold.

I dette dokument har der været fokus på metoder og værktøjer til at kunne dokumentere arbejdet med udvikling af software i teknologifaget, både på det overordnede og overbliksskabende niveau og på det mere konkrete kodeniveau. Arbejdet med kravspecifikationer og tilhørende testplaner er behandlet, lige så vel som anbefalinger til at arbejde med dokumentation i forbindelse med produktudviklingen.

I teknologifaget løser vi problemer, men det er samtidig et fag, hvor kompetencer og læring er i højsædet. Det betyder mange gange konkret stillingtagen til, hvilke softwareelementer, der skal inddrages, og hvor vægten på dokumentationen skal være.

F.eks. kan brug af programkomponenter såsom programbiblioteker, kodesegmenter, procedurer og eksterne biblioteker hjælpe eleven til at udvikle korrekte programmer, der løser et problem, hvis vel og mærke, der tages hånd om dokumentation og kvalitetssikring af udviklingen af softwareløsningen.

Ved eksempelvis at benytte modeller, pseudokode, flowdiagrammer og/eller problemtræer i udviklingsprocessen, kan eleven blive bevidst om, at et programs funktionalitet bedst beskrives i et højniveau sprog (f.eks. ved at beskrive hvorledes brugeren interagerer med programmet) og ikke i et lavere niveau sprog, der er karakteriseret ved at forklare, hvordan de enkelte instruktioner i programmet fungerer.

I objektorienteret programmering beskrives klasser med metoder og attributter. Hvis man f.eks. har en knap, som sender en besked (metode), og knappen er grøn (attribut), vil man ikke nødvendigvis beskrive alle metoder og attributter. I dette eksempel er det måske ikke nødvendigt at beskrive attributten med farven; måske er kommunikation og tilknytninger mellem forskellige klasser vigtigere. Måske vil Tidslinjediagrammer være aktuelle at inddrage i en dokumentation (Disse er dog ikke gennemgået i dette skrift - tidslinjediagrammer er en model over kommunikation mellem sender og modtager, som foregår på forskellige tidspunkter eller i realtidssystemer).

Så niveauerne for bredde og dybde af dokumentationen af software, vil afhænge af den faktisk udviklede software, det problem, det skal løse og den sammenhæng, det i øvrigt indgår i.

Som minimum kan følgende i nogle tilfælde foreslås: Softwaren dokumenteres med flowchart og ledsagende beskrivelser. Desuden vedlægges en kommenteret kildekode som bilag. Det er vigtigt, at eleven i detaljer begrunder de valg, der ligger bag de softwaremæssige udformning.