1DV610 - Laboration 1 - Tokenizer

Den muntliga beskrivningen

https://www.youtube.com/watch?v=3C_Udc4YA6c&ab_channel=ComputerScienceLNU

Syfte

- Skriva en bit kod som ska återanvändas av dig själv och andra.
- Sätta sig in i en ny problemdomän, lösa ett problem samt reflektera över sin lösning i kodkvalitétssyfte.
- Reflektera över och arbeta med kapitel 2 och 3 i Clean Code.

En tokeniserare

I laborationen kommer ni skapa en tokeniserare (Eng. tokenizer). Tokenisering används ofta som ett första steg när strängar skall tolkas, exempelvis av en kompilator.

Det korrekta begreppet är “lexikalanalys” som tar en textsträng och ger tillbaka de lexikala delar sk. “tokens” som strängen är uppbyggd av. En typ av token kan beskrivas exempelvis med ett reguljärt uttryck. En samling token-typer och deras beskrivningar blir en “lexikal grammatik” som beskriver alla typer av tokens som är giltiga.

Vi kan till exempel ha en beskrivning av tokentypen **WORD (ord)** som en samling bokstäver (a-ö och A-Ö) som sitter ihop utan mellanslag. Vi kan ha då även ha en beskrivning av en **DOT (punkt)** som just tecknet för punkt “.”. Vi kan använda reguljära uttryck (regex) för att beskriva vad som hör till vilken tokentyp.

Exempel. Med hjälp utav en lexikal grammatik bestående av endast WORD och DOT kan vi bryta upp en sträng i dess token. Strängen “Jag är glad”, består av WORD(“Jag”), WORD(“är”), WORD(“glad”). Medan strängen “Det regnar.” består av WORD(“Det”), WORD(“regnar”) och DOT(“.”)

Exempel 1. WordAndDotGrammar

Se följande exempel för att göra det tydligt hur en tokenizer bör fungera. Här använder vi regex för att beskriva vilka strängar som matchar token-typen.
Givet att vi har en tokeniserare som delar upp strängar i två olika typer av tokens: WORD och DOT. Se följande lexikala grammatik “WordAndDotGrammar”, här beskriven med regex och exempel:

Token-typ	**Regex**	**Exempel på matchningar**
WORD	/^[\w|åääöÅÄÖ]+/	*“a”, “Åsa”, “Meningen”*
DOT	/^\./	*“.”*

Tabell. En enkel lexikal grammatik som beskriver två olika typer av tokens.

Om vi ger följande sträng till tokeniseraren: *“Meningen består av ord.”.* Så kommer tokeniseraren hitta fem token-matchningar i meningen. Här beskriver jag sekvensen av ord med hjälp utav ett antal högerpilar. Noll pilar ger första tokenet i meningen. En högerpil ger andra tokenet och så vidare. Senare i laborationen kommer varje pil representera ett anrop till en funktion, mer om det senare. Vi har även en special-token END som betyder att inga fler tokens finns att matcha ur strängen.

Sekvens	**Token-Typ**	**Värde**
[]	WORD	“Meningen”
[>] *	WORD	“består”
[>>]	WORD	“av”
[>>>]	WORD	“ord”
[>>>>]	DOT	“.”
[>>>>>]	END	“”

*Tabell. Lägg märke till att mellanslagen inte följer med. Varje matchning har en typ och ett värde. Sist har vi även typen END som visar att inga fler token finns. * Varje matchning har en sekvens som berättar hur många tokens vi behöver hoppa fram innan vi når ett visst token. För att komma till WORD(“består”) måste först WORD(“Meningen”) blivit matchad vi representerar det som [>].*

Inte bara en enda grammatik

En tokeniserare skall kunna klara av olika typer av lexikala grammatiker för att kunna återanvändas i olika fall, att parsa texter, matematiska uttryck eller kanske programmeringskod.

Nedan skapar jag fyra olika tokeniserare som alla får olika lexikala grammatiker och olika strängar. “Klasserna” i exemplet är bara påhittade, er tokeniserare behöver inte heta “Tokenizer”. Er uppgift är att skriva en Tokenizer och testa den med ett par olika grammatiker.

```
wordAndDotGrammar = new G();
wordAndDotGrammar.add(new TT(“WORD”, “/^[\w|åäöÅÄÖ]+/”);
wordAndDotGrammar.add(new TT(“DOT”, “/^\./”);
Tokenizer textTokenizer = new Tokenizer(wordAndDotGrammar, “a red fish.”);
...
Tokenizer mathTokenizer = new Tokenizer(arithmeticGrammar, “3+4”);
Tokenizer javaTokenizer = new Tokenizer(javaGrammar, “public class …”);
Tokenizer jsTokenizer = new Tokenizer(jsGrammar, “class …”);
```

Vi skall därför visa ännu ett exempel där vi använder en annan grammatik än i exempel 1 för att göra det tydligare vad tokeniserarens uppgift är.

Exempel 2. ArithmeticGrammar

Vi skapar en ny lexikal grammatik “ArithmeticGrammar” som har följande lexikala element (token-typer), NUMBER, ADD och MUL. Här beskriven med regex och exempel på matchningar. Notera att denna inte läggs till WordAndDotGrammar utan bara har tre tokentyper.

Token-Typ	**Regex**	**Exempel på matchningar**
NUMBER	/^[0-9]+(\.([0-9])+)?/	“3.12”, “4” “45”
ADD	/^[+]/	“+”
MUL	/^[*]/	“*”

Om vi ger tokeniseraren som följer “ArithmeticGrammar” följande sträng *“3 + 2”* så kommer tokeniseraren hitta följande tre tokenmatchningar:

Sekvens	**Token-Typ**	**Värde**
	NUMBER	3
[>]	ADD	+
[>>]	NUMBER	2
[>>>]	END	

Exempel 3. MaximalMunchGrammar

Givet en annan lexikal grammatik “MaximalMunchGrammar” som **bara** har följande lexikala element.Notera att denna **inte bygger på** till ArithmeticGrammar utan bara har två tokentyper.

Token-Typ	**Regex**	**Exempel**
FLOAT	/^[0-9]+\.[0-9]+/	“5.1”
INTEGER	/^[0-9]+/	“5”, “56”

Om vi ger en tokeniserare som använder MaximalMunchGrammar strängen “3.14 5” kommer tokeniseraren när den stöter på “3.14 kunna matcha båda tokentyperna(FLOAT(3.14) och INTEGER(3).

Då praktiseras “maximal munch”-regeln och den tokentyp vars matchning får flest tecken matchade, dvs FLOAT som matchar 3.14 (fyra tecken).

Sekvens	**Token-Typ**	**Värde**
	FLOAT	“3.14” (4 tecken långt)
	~~INTEGER~~	~~3 (1 tecken)~~
[>]	INTEGER	5
[>>]	END	

OBS! “Maximal munch” är en regel som appliceras oavsett grammatik. Dvs om två token-typer matchar en del av strängen kommer den typ som matchar flest tecken vinna!

Exempel 4. Ett lexikalt fel

Om något inte matchas av någon tokentyp i grammatiken så är det en ogiltig sträng.

Om vi använder MaximalMunchGrammar för att skapa en Tokenizer med strängen “3.14 Hej! 4” så finns ingen Token-Typ som matchar “Hej!” därför kommer vi avbryta och ge ett felmeddelande enligt nedan. Det blir då ett lexikalfel.

Sekvens	**Token-Typ**	**Värde**
	FLOAT	“3.14”
[>]	Lexikalfel (Ex. Exception thrown)	No lexical element matches ”Hej! 3”

Tabell. Lägg märke till att undantaget inte kastas förrän Hej! kommer i strängen. Lägg också märke till att vi inte fortsätter skapa ytterligare tokens efter felet. Ingen token skapas till “ 4”.

Testa din förståelse

Givet tokeniseraren med ArithmeticGrammar. och strängen“10” fyll i nedan

Tokenmatchning	**Token-Typ**	**Värde**
Fyll i själv:	*Number*	*10*
Fyll i själv: [>]	*END*	*“”*

Strängen: “1.02”

Tokenmatchning	**Token-Typ**	**Värde**
Fyll i själv:		
Fyll i själv:		

Strängen: “ 4 *5.0f”

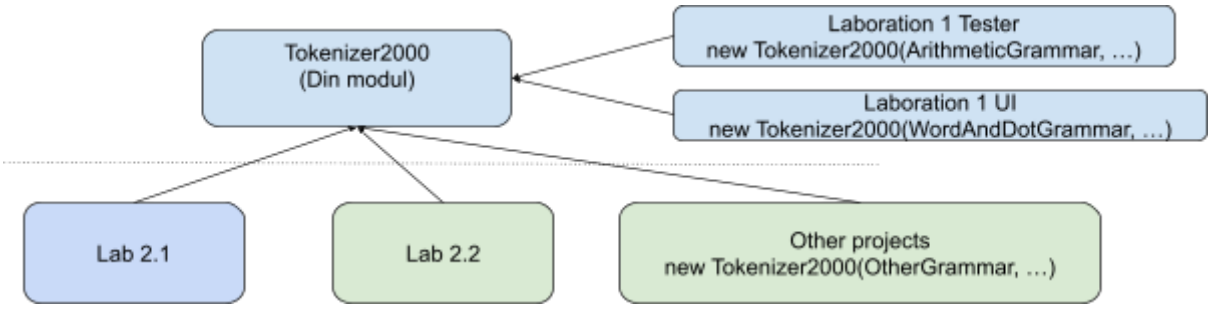
Tokenmatchning	**Token-Typ**	**Värde**
Fyll i själv:		
Fyll i själv:		
Fyll i själv:		
Fyll i själv:		

En beskrivning av koden du skall skapa

Du skall skriva en “modul” som tillhandahåller en tokeniserare. En modul kan vara en komponent, en klass, ett bibliotek, ett api eller liknande. Exempel på moduler är Console i Node.js som används för att lösa problemet med att skriva till konsolen. Modulen tillhandahåller ett “interface” bestående av klasser och metoder för att lösa ett problem.

Din modul ska tillhandahålla ett interface för att tokenisera strängar. Det skall vara enkelt för en annan programmerare (läs student) att använda sig av din modul utan din hjälp. Programmeraren skall själv kunna skriva sin egen grammatik.

Du behöver själv använda dig av olika grammatiker för att kunna testa din tokeniserare. Skilj därför på din modul som löser tokenisering och de implementationer som använder modulen för att testa den. Du skall kunna återanvända dina klasser till att tokenisera olika grammatiker utan att ändra koden på din tokeniserare. Exempelvis kan vi ha ett projekt som använder tokeniseraren med nummer och ett annat med strängar men båda projekten använder samma klass för tokeniserare.



Figur. Din modul skall kunna användas från flera olika projekt och av olika parter. De gröna projekten skrivs av andra programmerare än dig själv. De olika projekten tillhandahåller själva olika grammatiker men använder din modul “Tokenizer2000”.

En modul(en samling kod som skall återanvändas) kan skapas på olika sätt beroende på språk och vad som är praktiskt. Ni väljer här hur återanvändandet skall ske. Börja enkelt med filer i samma projekt.

Exempel 1. Daniel lägger java-källkoden till Tokenizer2000 samt andra klasser som behövs för att använda tokeniseraren i en egen katalog och har sedan separata källkods kataloger för de olika projekt som använder Tokenizer2000. I de olika projekten inkluderas Tokenizer2000 genom relativa sökvägar. På sikt avses skapa ett .jar-bibliotek.

Exempel 2. Emma skapar en node-modul till sin tokeniserare. De projekt som skall använda hennes tokeniserare får anropa require.

Publikt interface

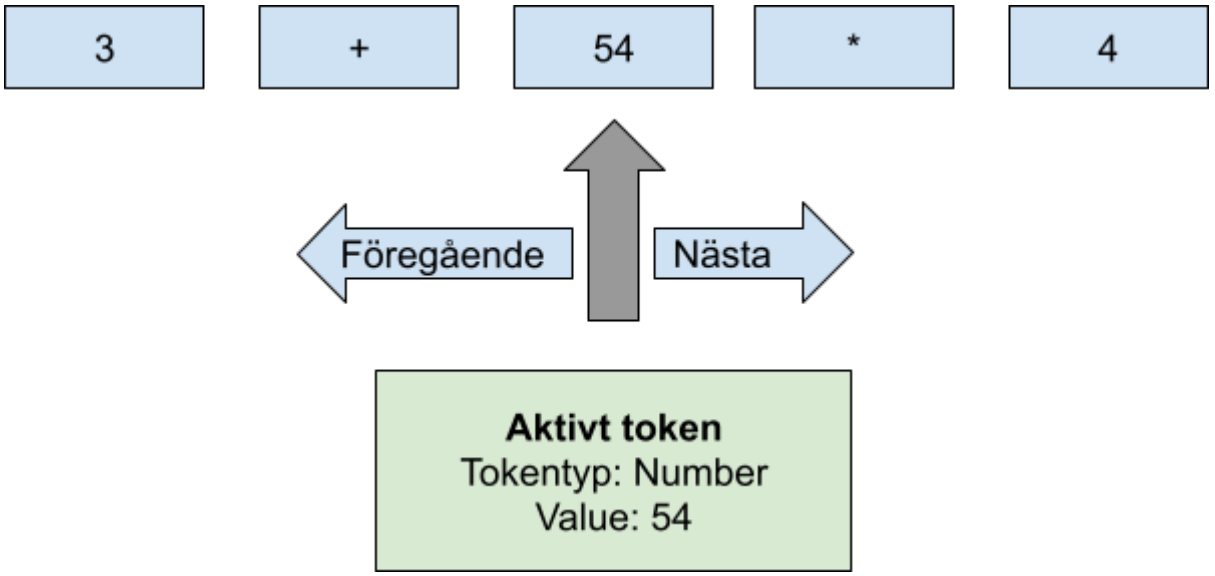
För att fungera i flera olika projekt bör din kod fungera på liknande sätt som andra studenters kod utan att för den skull kopiera det sättet.

OBS! Eftersom namngivning ingår i uppgiften så kommer jag vara lite flytande med vad metoderna kan/bör heta eller vilka klasser som skall finnas i det publika interfacet, detta skall ni själva bestämma och inte ta från någon annan.

Bakgrund: Ni kan förvänta er att tokeniseraren i framtiden blir en del av en parser i **laboration 2**. **En parser** kan skapa syntaktisk mening utav tokens. Till exempel kan en parser säga att *"3 + 4"* är en matematiskt korrekt uttryck eller att strängen *"Denna mening saknar punkt"* är ogiltig för det finns ingen punkt som avslutar meningen.

Aktivt token

När en tokeniserare skapas och får en sträng kommer första tokenet ur strängen vara aktivt. Det behöver då finnas en metod för att kunna få det aktiva tokenet samt att kunna stega aktivt token fram[>] och tillbaka[<]**.** Symbolerna > och < används för att vi skall kunna prata om olika anrop till metoder som stegar fram och tillbaka i strängen ett token i taget. Vi kan ex skriva [>>] för att beskriva att vi stegat fram två token.



*Figur. Figuren nedan illustrerar tokenisering från "3 + 54 * 4" och där 54 är det aktiva tokenet av typen nummer efter att ha stegat två steg framåt [>>].*

Exempel 5 ExclamationGrammar och att kunna stega bakåt!

Se följande exempel

Token-typ	**Regex**	**Exempel**
WORD	/^[\w]+/	"a", "Ewa"
EXCLAMATION	/^\!/	"!"

Sekvens	Aktivt token
>	WORD(“world”)
>>	EXCLAMATION(!)
><	WORD(“Hello”)

Tabell. Givet grammatiken ExclamationGrammar och meningen “Hello world!”. Kommer första aktiva token vara WORD(“Hello”) om vi anropar nästa[>] får vi istället WORD(“world”) om vi sedan anropar föregående [<] får vi WORD(“Hello”) igen, vi kan skriva sekvensen av anrop som [>] + [<] = [><]

Metoder hos er tokeniserare

Ni har stor frihet i utformningen av er modul men följande metoder är lämpliga. Ni kan lägga till fler metoder och dela upp koden i klasser. Det är även fritt att placera dessa metoder i olika klasser eller dela upp dem i flera steg.

- **Konstruktor**: Programmeraren kan skicka med en beskrivning av de **olika** typer av tokens(exempelvis en lista/array) som finns i grammatiken samt strängen som skall tokeniseras. Beskrivningen av varje tokentyp bör bestå av ett namn på tokentypen (ex. WORD, NUMBER etc) samt en beskrivning av hur matchningen går till (ex. ett reguljärt uttryck regexp”).
- En metod som ger längsta tokenmatchning på aktiv plats i strängen.
  En tokenmatchning skall innehålla **tokentyp** samt **värdet** (texten som matchats).
- En metod där vi kan be tokeniseraren om att hoppa fram aktivt token till **nästa** [>] token i strängen.
- En metod där vi kan be tokeniseraren att hoppa tillbaka aktivt token till **föregående** [<] tokenet i strängen.

Koden ska kasta undantag om något fel händer, exempelvis om tokeniseraren inte kan matcha en del av texten mot någon av tokenbeskrivningarna. Detta undantag kastas först när en del av strängen som inte matchas och vi ber om aktivt token.

Koden bör hantera slutet på tokenströmmen så att den som använder tokeniseraren kan förstå att det inte finns fler tokens att få. Detta genom att vi returnerar ett special-token END.

Koden skall ta hänsyn till Maximal munchregeln beskrivet i exempel ovan.

Verifiering och validering av din modul.

För att nå samförstånd om kodens funktion behöver den testas. Hur detta sker är mindre viktigt än att det sker. Som jag ser det finns det tre val (men fler varianter kan finnas).

- Ni testar er kod genom att skapa ett användargränssnitt (webb, console, ui) ni går manuellt genom varje testfall och matar in indata och observerar själva manuellt utdata och jämför med förväntat utdata (se tabell). Ni knyter då < och > till olika indata som användaren ger.
- Ni skapar en testapplikation som automatiskt kör varje test var för sig och kör den koden och observerar testernas utfall antingen med kod eller manuellt.
- Ni skapar automatiska enhetstester för varje testfall

Testfall

För godkänd och väl godkänd
Testfallen beskrivs med en grammatik, en sträng, samt en sekvens av anrop till nästa [>] och föregående [<] samt förväntat aktivt token. **Fyll själv i de tomma rutorna**

Namn	**Grammatik**	**Sträng**	**Sekvens**	**Förväntat**
TC1	WordAndDotGrammar	“a”	[]	WORD(“a”)
TC2	WordAndDotGrammar	“a aa”	[>]	WORD(“aa”)
TC3	WordAndDotGrammar	“a.b”	[>]	DOT(“.”)
TC4	WordAndDotGrammar	“a.b”	[>>]	
TC5	WordAndDotGrammar	“aa. b”		WORD(“b”)
TC6	WordAndDotGrammar	“a .b”	[>><]	DOT(“.”)
TC7	WordAndDotGrammar	“”	[]	END
TC8	WordAndDotGrammar	“ ”	[]	
TC9	WordAndDotGrammar	“a”		END
TC10	WordAndDotGrammar	“a”	[<]	
TC11	WordAndDotGrammar	“!”	[]	Exception
TC12	ArithmeticGrammar	“3”	[]	NUMBER(“3”)
TC13	ArithmeticGrammar	“3.14”	[]	NUMBER(“3.14”)
TC14	ArithmeticGrammar	“3 + 54 * 4”	[>>>]	MUL(“*”)

TC15	ArithmeticGrammar	“3+5 # 4”	[>>>]	
TC16	ArithmeticGrammar	“3.0+54.1 + 4.2”	[><>>>]	
Eget...				

För Väl godkänd
Utöka ArithmeticGrammar så att den kan hantera parenteser[(][)], division[/] och subtraktion[-]. Skapa själv testfall för detta. Låt de två olika parenteserna bli olika tokentyper (vänsterparantes och högerparantes)

Kodkvalitetskrav

Namngivning (kapitel 2)

Läs kapitel 2.
Skapa en tabell över fem namn på identifierare (Ex. klasser, metoder och variabler) som finns i ditt **publika interface** hos modulen. Utgå ifrån kapitel-2s titlar och ange de viktigaste “reglerna” som applicerats **eller skulle kunna appliceras** på just ditt namn. Försök variera regler mellan namnen så att inte alla har samma titlar applicerade.

Ange även en kort reflektion. Ni kanske upptäcker en brist hos er namngivning. Det är här tillåtet att antingen ändra er kod eller leva vidare med den, men bristen skall då finnas angiven. Jag ser hellre att ni hittar och reflekterar över era brister än att ni döljer dem.

Exempel.

Namn och förklaring	Reflektion
Tokenizer2000 Klassnamn på huvudklassen i modulen	**Avoid Disinformation** 2000 betyder inget speciellt och tillför därför inget till namnet. **Don't Be Cute** Det kan verka sött men är vilseledande. De som inte är millennium-romantiker kan missa det roliga. Bara "Tokenizer" är ett tydligare namn.
boolean TokenMatch.isBetter(other) Metodnamn på metod som avgör om en tokenmatchning är bättre än en annan baserat på maximal munch.	**Method Names** Is hintar boolskt returvärde. Argumentet och metodnamnet är tänkt att läsas som "is this better (than) other". **Use Problem Domain names** Better är oklart i kontexten och borde bytas ut mot "Maximal munch" som problemdomänen använder MEN detta är inte säkert att programmeraren som skall läsa är insatt i detta därför bör vi istället använda "Solution Domain". **Use Solution Domain names** hasMoreMatchedCharacters är eventuellt tydligare och kräver inte att läsaren vet om Maximal munch.

Funktioner (kapitel 3)

Läs kapitel 3.
Skapa en tabell över dina fem längsta metoder. utgå ifrån kapitel-3s titlar och ange de viktigaste reglerna. Föreslå förändringar.

Metodnamn och länk eller kod	antal rader (ej ws)	Reflektion
boolean TokenMatch.isBetter(other)	1	**Do one thing** Metoden gör bara en sak (jämför längden på matchad text med längden på other).

		Function Argument Metoden har bara ett argument (**monadic**) Eftersom jag har skrivit i javascript vore det bra att typen på argumentet **other** framgår via exempelvis en metodkommentar. **Common Monadic Form** Vi ställer en fråga om argumentet, är this bättre än other och metoden gör endast då en “query” (**Comman Query Separation)** och ändrar inte värdet på objektet eller argumentet.

Reflektion
Skriv en kortare reflektion (halv sida 12pt) där du beskriver dina erfarenheter från din egen kodkvalitet. Använd begrepp från boken.

Inlämning och examination

Koden lämnas in genom formulär Daniel kommer tillhandahålla i Slack några dagar innan deadline (se schema). Formuläret kommer innehålla ert namn samt länk till git-repositorium samt några frågor.

Testrapport och kodkvalitetskrav lämnas in som ett markdowndokument i ert repositorium där även reflektionen skrivs. Jag planerar att skapa en mall för detta.

Regler

- Skriv **all** kod själv, skriv inte tillsammans med någon (sida vid sida-programmering).
- Kopiera inte kod från någon annan. Skriv inte av kod.
- Använd inte bibliotek eller färdiga metoder för att lösa huvudproblemet att skapa tokeniseraren. Om ni tänker if-satser och for-loopar så gör ni rätt. Ni får använda reguljära uttryck för att beskriva olika token-typer. Ni får använda metoder som måste användas.
- Koden skall kunna delas med klasskamrater under senare laborationer och workshop.
- Koden är skriven i ett programmeringsspråk som förekommit tidigare än den här kursen i utbildningen.
- Objektorienterad kod med klasser och metoder i klasser. Du kan ha kod utanför klasser men bara om den behövs för att starta upp koden (ex. nodejs

“server.listen(port…”). Inga metoder i dina klasser får vara statiska mer än om det behövs för att starta upp koden (Ex. java “public static void main(...”-

För Godkänd nivå “E”

- De flesta testfall skall fungera, något enstaka testfall får misslyckas eller känd bugg får finnas. Tokeniseraren måste dock fungera i stort. Det finns en testrapport som visar vad som fungerar och hur det är testat.
- Koden är förberedd för att återanvändas. Den kod som skall användas av andra kan lätt bli tillgänglig för en annan student.
- All kod finns med historik på git.
- Tabeller med Kodkvalitetskrav är ifyllda för det publika interfacet.
- En reflektion är skriven baserat på vad studenten har lärt dig
- Studenten uppfyller Reglerna.

För högre betyg

- Samtliga punkter från “För godkänd nivå ”
- Samtliga testfall är uppfyllda. Egna testfall finns för MaximalMunch och andra fall som kan identifieras exempelvis i utkanten av strängen.
- Utökad grammatik i Exempel 2 för paranteser subtraktion och division, utökade testfall för olika fall för detta.
- Testfallen är automatiserade
- Det är mycket tydligt utifrån git hur modulen skall återanvändas.
- En tydlig separation mellan modulen som skall återanvändas och de sätt som modulen har används för att testa den.
- Kodkvalitetskraven är varierade i vilka regler de använder. Studenten är tydlig i hur regler och kodnamn samt metoder hänger ihop.
- En välskriven reflektion är skriven som baseras på erfarenheter och lärdomar utifrån laborationen samt bokens kapitel 2 och 3.