Sensorveiledning Software Engineering v2019

Symbol	***Betegnelse***	***Generell, ikke fagspesifikk beskrivelse av vurderingskriterier***
A	Fremragende	Fremragende prestasjon som klart utmerker seg. Kandidaten viser svært god vurderingsevne og stor grad av selvstendighet.
B	Meget god	Meget god prestasjon. Kandidaten viser meget god vurderingsevne og selvstendighet.
C	God	Jevnt god prestasjon som er tilfredsstillende på de fleste områder. Kandidaten viser god vurderingsevne og selvstendighet på de viktigste områdene.
D	Nokså god	En akseptabel prestasjon med noen vesentlige mangler. Kandidaten viser en viss grad av vurderingsevne og selvstendighet.
E	Tilstrekkelig	Prestasjonen tilfredsstiller minimumskravene, men heller ikke mer. Kandidaten viser liten vurderingsevne og selvstendighet.
F	Ikke bestått	Prestasjon som ikke tilfredsstiller de faglige minimumskravene. Kandidaten viser både manglende vurderingsevne og selvstendighet.

Det vil i mange av oppgavene være mulig å begrunne andre løsninger enn de foreslåtte, da det bare er hovedtrekkene som beskrives for hver oppgave.

Emnene er såpass omfattende at det ikke vil være forventet at kandidaten går i dyptgående detaljer om hvert av punktene i hvert av svarene, men at kandidaten viser detaljert kunnskap der det er nødvendig for å forklare forskjeller mellom ulike metoder og konsepter.

Sensuren skal vurdere i hvilken grad kandidaten *i helhet* (og ikke utelukkende som en summering av “poeng” for hver oppgave, men omfanget angitt kan brukes som et utgangspunkt for vekting) har kunnskap om emnene til å vite når og hvordan de bør benyttes og hvordan de bygger opp under trygg og stabil programvareutvikling.

Oppgave 1

a) Planbasert: “alt” planlegges for hvert steg før det går videre til neste steg, flyten er stort sett i en retning gjennom de ulike stegene. Kan være byråkratisk, dokumentasjonstungt, rigid. Overordnet plan.

   Smidig utvikling: Utsetter avgjørelsene lengst mulig, tilpasses endringer i behov underveis i prosjektet, tett samarbeid med kunde. Ikke alle delene av systemet er like viktig. Verdi først, bruke verktøyene som gir mening for prosjektet. Ikke planløst.

b) 1) Stand-up: Kort, daglig møte med teamet. Skal ikke ta mer enn 10-15 min, identifisere ev. flaskehalser med inneværende arbeid, folk man venter på, etc. Kun status. Dybdediskusjoner tas i egne møter.

   2) Backlog: Samling av gjennomførbare tiltak for å “ferdigstille” produktet; en levende liste - ikke fullstendig ved prosjektoppstart og ikke tom før produktet tas i bruk. Ofte brukt sammen med en Kanban-bord, danner grunnlaget for burndown chart sammen med story points/velocity.

   3) Minimum Viable Product: Minste versjon av produktet vårt som skaper verdi, slik at vi kan validere og få satt produktet i produksjon så tidlig som mulig. Kvalitet og verdi først framfor “komplett” produktt.

   4) Three Amigos: Produkteier, Tester og Utvikler. Gjennomgår materiale fra samtaler med kunder og konkretiserer brukerhistorier for ytterligere formalisering og kvalitetssikring. Kilden til de levende kravene som hele tiden fylles i backloggen og kompletterer den levende kravspesifikasjonen og dokumentasjonen til systemet.

Oppgave 2

a) Nøkkelord: stabilitet, forutsigbarhet, en branch påvirker ikke hovedbranchen, åpent for eksperimentering, gir oss fortsatt alle features i et versjonskontrollsystem, kan vedlikeholde egne grener/versjonstre av applikasjonen. Lar oss samarbeide om egne branches på lik linje som hovedbranchen, uten at det påvirker versjonen som ligger ute. Påvirker ikke kode utenfor vår egen branch. Feature branching og Pull requests er også diskutert, noe som branching gir oss muligheten for å gjennomføre.

b) C: Vi kan se historikken for hvordan vi bygger “maskinen” vår, gå tilbake til tidligere versjoner av samme container, finne ut hvorfor noe brakk med miljøet som applikasjonen vår kjører i. Hvorfor endringer ble gjort. Kunne rulle tilbake til fungerende miljø.

O: Vi kan versjonskontrollere “datasenteret” vårt, se endringer i tildeling av ressurser, ha historikk på hvorfor endringer ble gjort, hvem og når endringene ble gjort, kunne rulle tilbake til oppsettet vi vet fungerte om noe brekker.

Dette er informasjon som ellers bare forsvinner (noen endret noe på disk, etc.) eller må logges manuelt i et tilhørende system.

Oppgave 3

a) Pakkesystemene lar oss si “Jeg vil ha dette biblioteket, i denne versjonen, tilgjengelig i prosjektet mitt”, og så ordnes resten automagisk. Vi slipper manuell håndtering av bibliotek, hvor ting skal legges inn, manuelt oppsett av hvilke bibliotek / kataloger som skal være tilgjengelig i prosjektet. Avhengighetene blir håndtert på en standardisert måte, slik at det er enkelere å sette seg inn i, eller starte, et nytt prosjekt. Gir oss mal for å bygge containere - hva som trengs for å bygge applikasjonen fra scratch uten å manuelt måtte lete opp avhengigheter og versjoner.

b) Oppgaven er å låse pakker til den versjonen som ble installert første gang pakken ble lastet ned, slik at man får akkurat samme miljø med samme avhengigheter neste gang pakkene installeres.

   Det gjør at vi får stabilitet og identisk miljø, og at oppgraderinger til nye pakkeversjoner blir en eksplisitt handling av utvikleren. Full oversikt over avhengighetene som ble installert. En feilkilde mindre.

Oppgave 4

a) Enhetstester er helt enkle tester uten avhengigheter, som oftest tester av en enkelt metode. Mye brukt for bibliotekskode og isolert funksjonalitet. Integrasjonstester “integrerer” flere deler av systemet og tester med avhengigheter. Interoperabilitet mellom modulene som viser at “hele” sekvensen med kode fungerer fra et nivå og nedover. Systemtester (også nevnt GUI-tester) er tester som benytter grensesnittet “utenfra” applikasjonen for å validere oppførsel.

   Testpyramide fra smal topp Systemtester - Integrasjonstester - Enhetstester.

b) Red - Skriv testen slik at du bekrefter at den fremtidige metoden returnerer riktig resultat. Resultatet foreløpig er at testen feiler og markeres som rød

   Green - Skriv koden i metoden slik at metoden returnerer dataene som er korrekte i henhold til testen din. Testen skal nå passere og markeres som grønn.

Refactor Rydd opp i koden din slik at den blir vedlikeholdbar og strukturert på en fornuftig måte for framtiden.

For gjenbruk, økt forståelse, dokumentasjon og mindre teknisk gjeld. Minner oss på å rydde opp koden for hver grønne test. Ved å bli kjent med refactoring metoder og verktøy vil vi kunne “gro” en god arkitektur ut av prototypepreget kode

Oppgave 5

Kodeoppgave. Kandidaten bør vise at de har beskrevet fire tester med hensiktsmessig navngiving, bruk av assertFalse og assertTrue, og at de tester årene som ikke er skuddår (og ikke nevnt eksplisitt som eksempler). De fleste kandidatene vil nok benytte eksemplemen som er gitt i oppgaveteksten, hvis ikke må det bekreftes at testdata faktisk gir riktig oppførsel iht. testen.

Rene syntaksfeil skal overses (manglende {, }, etc.), så lenge det er tydelig hva kandidaten mener.

```
test_divides_by_four_is_leap_year() {
   assertTrue(isLeapYear(1996));
}

test_divides_by_four_and_hundred_is_not_leap_year() {
   assertFalse(isLeapYear(1900));
}

test_divides_by_four_and_four_hundred_is_leap_year() {
   assertTrue(isLeapYear(2000));
}

test_is_not_leap_year() {
   assertFalse(1993);
   assertFalse(1998);
}
```

Oppgave 6

Det forventes ikke at kandidaten skriver mer enn en eller to setninger om hvert punkt.

Single Responsibility Principle

En funksjon, klasse eller modul skal bare ha én årsak/kilde til endring

Dette gjør dem enklere å teste, vedlikeholde og utvide

En klasse skal bare ha én årsak til å endre seg
Flere ansvarsområder gjør det mer utfordrende, om ikke umulig å gjenbruke de avhengighetsløse delene
Klassene vi "burde hatt" finnes ofte i lange metoder med mange variabler
Når man har flere ansvarsområder - med samme årsak til endring - må abstraksjonsnivået i klassen være på samme nivå.

Open / Closed Principle

Koden skal være enkel å utvide uten å måtte endre eksisterende kode.

Det er om å gjøre å endre minst mulig kode når nye features introduseres
Mange algoritmer er gjenbrukbare, men deler av innholdet varierer
Prioriter å gjøre koden utvidbar og fleksibel fremfor gjenbrukbar
Dette bidrar til å gjøre det lettere å introdusere ny funksjonalitet
Ved å holde hver klasse liten og utvidbar (se SRP) blir det også lettere å endre eksisterende funksjonalitet
Åpen for utvidelse, men lukket for endring

Liskov Substitution Principle

Ingen skjulte effekter eller behov ut over det som er synlig på- og logisk for interface / abstrakt klasse

Når en klasse eller funksjon benytter en abstraksjon må all funksjonalitet på den konkrete implementasjonen oppføre seg som forventet
Konkrete implementasjoner skal kunne endres og byttes ut uten å påvirke kode som benytter abstraksjonene
Ved brudd på dette prinsippet mister abstraksjonene nytteverdien sin og koden kunne like gjerne vært 100% tett koblet

Interface Segregation Principle

La konsumenten forholde seg til bare de funksjonene som er nødvendig
Forenkler bruk av komponenter som inneholder mye

Konsumenter av abstraksjoner bør kun måtte forholde seg til funksjonalitet som er relevant.
Altså bør abstraksjoner i seg selv være sett av sammenhengende funksjonalitet.

Dependency Inversion Principle

Dette muliggjør løst koblet kode, konfigurasjon, testing og resten av SOLID prinsippene
Hold kontrollen på ustabile avhengigheter så nærme main() som mulig
Injiser avhengighetene i constructors for å invertere kontrollen

Kode blir testbar fordi vi kan bytte ut lagrings- og kommunikasjonsmekanismer med test doubles
Abstraksjonene skal ikke peke på noen konkrete ustabile avhengigheter.
Vår kjernekode (domain model) skal kun avhenge av våre abstraksjoner.
Det er en ekstra fordel å kun benytte dataoverføringsobjekter (DTO'er) i abstraksjonene for ytterligere å skille kjernekoden fra omverdenen