Eksamen INF112 vår 2019

Oppgave 1: (10% av totalkarakter)
a: Forklar hva Minimum Viable Product (MVP) er og hva hensikten med MVP er. Forklar hvordan oppgaver bør prioriteres i et prosjekt for å lykkes. Forklar også hva teamet oppnår ved den foreslåtte måten å prioritere på.

Minimum Viable Product (MVP) er den minimale implementasjonen (produktet) man kan komme seg unna med som kan settes i produksjon og gi verdi. Verdi kan være flere ting: tilbakemelding på bruk, og omsetting av produkt/inntjening av verdi. MVP skal ha med de delene av prosjektet der det er høyest risiko (både på det tekniske planet men også funksjonalitetsmessig) og det er viktig å få tilbakemelding fra brukere.
Hensikten med MVP er å komme til et punkt der det går an å få tilbakemelding fra reelle brukere så fort som mulig, siden kursen i prosjektet justeres når det kommer reell bruk av systemet. Hvis det er helt nye features: ta med de som er helt unike så de blir utprøvd på brukere. Ta med den funksjonaliteten brukeren ikke kan klare seg uten. Systemet må være komplett nok til at brukeren kan bruke det (selv om bruker kanskje er klar over at dette er en testversjon). MVP betyr ikke nødvendigvis salgbart produkt, kun reell bruk. MVP sikrer at hele stacken i prosjektet blir utprøvd, fra persistering, kommunikasjon, forretningslogikk og brukergrensesnitt. Hensikten er å hele tiden lage funksjonalitet som er nyttig for brukeren, og for tidligst mulig å få tilbakemelding på hvordan nylaget funksjonalitet oppleves. MVP tester også at bygg og deploy fungerer, og at produksjonsmiljøet fungerer.

De oppgavene som har høyest risiko skal prioriteres først. Risiko kan være begrunnet i flere ting, feks: teknisk vanskelig å få til, virksomhetskritisk funksjonalitet. Ved å prioritere teknisk kompliserte oppgaver tidlig i kombinasjon med tidlig produksjonssetting, får vanskelige tekniske oppgaver tid til å modnes slik at riktig løsning kan velges før det er for sent. Tidlig erfaring med hvordan bruken er vil også hjelpe med å avgjøre om riktig løsning er valgt. Virksomhetskritiske oppgaver skal også velges først, siden det er denne funksjonaliteten som er mest verdt for brukere eller for produkteier. Ellers bør oppgaver innenholde hele stacken, slik at funksjonaliteten kan utvikles i sin helhet med en gang (så kommer reell bruk og reell tilbakemelding slik at justeringer kan gjøres).
Oppgaver som ikke gir reell verdi for kunden skal ikke prioriteres (kråkesølv).

Poeng: 10 totalt
2 for høyest risiko teknisk
2 for høyest risiko brukermessig
2 for oppgaver som krever full stack implementasjon
2 for tilbakemelding fra brukere
2 annet

Oppgave 2: (15% av totalkarakter)
a: Forklar hva som kjennetegner en god brukerhistorie og nevn 3-5 problemer vi ofte ser med brukerhistorier.

En god brukerhistorie er konkret. Den er skrevet slik at alle har samme forståelse av hva som skal lages. Hvis det er en bestemt type bruker som skal gjennomføre handlingen, skal rollen nevnes. Brukerhistorien bør inneholde HVEM som utfører oppgaven, HVILKEN funksjonalitet som skal utføres, og HVILKET behov som blir oppfylt. En god brukerhistorie skal også ha tilknyttet seg akseptansekriterier, slik at alle kan forstå når de er ferdige med å lage funksjonaliteten. Da blir det klart definert hva som skal til for å oppfylle brukerhistoriern.
Akseptansekriteriene kan gjerne skrives slik at det kan testes direkte fra kriteriene. Brukerhistorier skal ikke inneholde implementasjonsdetaljer, men kun behov (hva er problemet vi trenger å løse, ikke hvordan problemet løses).

Vanlige problemer vi ser med brukerhistorier:

- for store (oppgavene blir for mange, og teamet klarer ikke holde oversikt over når man er ferdig i tillegg til at hver brukerhistorie tar veldig lang tid). Også kjent som isfjell
- oppgaver uten verdi (kråkesølv): tilsynelatende verdi, men egentlig gir ikke disse noe verdi for brukere av systemet.
- uklare: det er vanskelig å opparbeide seg en felles forståelse for hva som skal lages og når man er ferdig
- historier uten begrunnelse (småbarnsforelderen): mangler behov

Poeng: 6 totalt på a

3: hvem/hvilken funksjonalitet/hvilket behov, akseptansekriterier
3: en for hver brukerhistoriefeil (kan også være andre enn de jeg har oppgitt), men bør ha med enten for store eller for uklare for ikke å få trekk her, de er viktige

b: Både forelesere og studenter bruker tjenesten Mitt UiB. Lag 3 brukerhistorier med akseptansekriterier for studenter eller foreleseres bruk av Mitt UiB. Skriv kort om hvilke oppgaver som må løses for å realisere de ulike kravene.

1: Som student ønsker jeg å kunne se oversikt over alle forelesningene i fagene jeg tar slik at jeg kan vite når og hvor jeg skal møte til forelesning.

Akseptansekriterier:

- student kan se all informasjon om minst to fag
- student kan se når hver forelesning begynner og slutter
- student kan se hvor hver forelesning skal være

Hva må løses:

- brukergrensesnitt for å vise informasjon om fag
- modellere fag og student, samt persistens av dette
- forretningslogikk for å hente ut informasjon om lagrede studenter og fag
- forretningslogikk for å legge inn informasjon om studenter og fag (ikke påkrevd, kan fakes i test)

2: Som foreleser ønsker jeg å kunne sende beskjeder til alle studentene som tar faget mitt slik at studentene kan få viktig informasjon.

Akseptansekriterier:

- foreleser kan forfatte og sende en beskjed til alle studentene
- studenter (minst to) mottar sendt beskjed
- studenter som ikke tar faget får ikke beskjeden

Hva må løses:

- modellere foreleser og studenter
- brukergrensesnitt for å kunne lage en beskjed
- forretningslogikk for å sende melding til studentene, feks epost
- forretningslogikk for å vite hvem som skal ha beskjeden og ikke

3: Som student ønsker jeg å kunne se hvilke obligatoriske oppgaver som er godkjent slik at jeg vet om jeg får gå opp til eksamen.

Akseptansekritierier:

- student kan se status på minst 2 obligatoriske oppgaver
- student kan se om oppgavene er godkjent eller ikke
- student kan se at alle oppgaver er godkjente
- student kan se at ingen oppgaver er godkjente

Hva må løses:

- brukergrensesnitt for å vise status på oppgaver i et fag
- modellere fag, student og oppgaver i et fag
- forretningslogikk for å kunne hente oppgaver i et fag
- brukergrensesnitt som viser om oppgaven er godkjent eller ikke

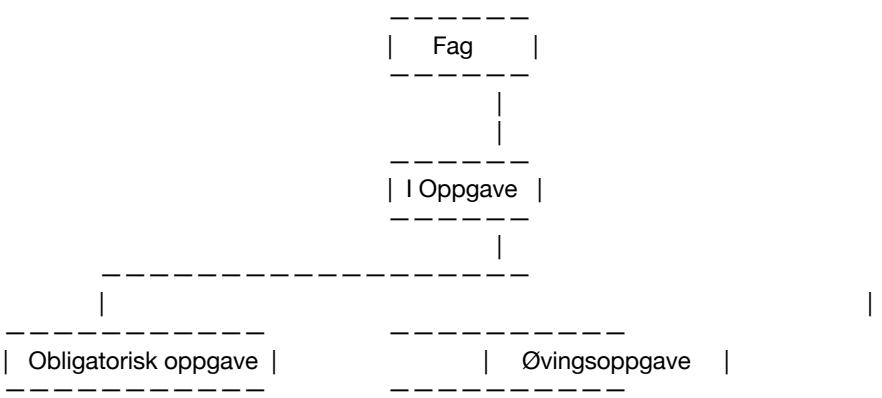
Poeng: 9

- 1 pr brukerhistorie (gitt god nok kvalitet), 1 for akseptansekriterier og 1 for at oppgavene som må løses er oppgitt (alt sammen x 3)

Oppgave 3: (5% av totalkarakter)
Hva er Open-closed principle og hvordan vil du designe koden din for å følge dette designprinsippet? Tegn (på papir) et enkelt klassediagram som viser et eksempel med klasser (bruk gjerne klasser fra prosjektet eller forrige oppgave).

Open-closed-principle handler om at koden din skal være åpen for utvidelser, men stengt for modifikasjon. Når systemet utvides med ny funksjonalitet, skal man ikke måtte endre mange steder i koden for å få lagt til endringen. Ideelt sett skal man kun trenge å legge til ny funksjonalitet og resten av koden fungerer som før.

Praktisk løses dette med et interface. Da vil alle andre klasser bruke interfacet og få inn en spesifikk instans som implementerer interfacet. Da er det bare å legge til en ny klasse med riktig funksjonalitet (feks øvingsoppgave) og en måte for brukeren å benytte den nye typen på, og ingen annen endring trengs. Fag vet ingenting som hvilke typer oppgaver som finnes og aksepterer alle klasser som følger kontrakten.



Fag har IOppgave, og så kan alle de ulike typene oppgave implementere interfacet.

i stedet for

Fag —> Obligatorisk oppgave
Fag —> Øvingsoppgave
hvor fag må utvides hver gang en ny type oppgave legges til.

Poeng: 5 totalt

- 2: åpen for utvidelse, lukket for modifikasjon (fortrinnsvis med litt forklaring)
- 2: interface med forklaring
- 1: meningsfullt klassediagram

Oppgave 4: (5% av totalkarakter)
Forklar kort hva personvernforordningen er og hva hensikten med forordningen er.

Personvernforordningen er et lovverk som skal regulere hvordan personopplysninger behandles, og privatpersoners rettigheter til egne personopplysninger i møte med alle som ønsker å bruke deres data. GDPR regulerer:

- hva data kan brukes til (uten å hente inn samtykke)
- at data ikke kan brukes til noe annet enn originalt formål,
- at data skal kunne slettes eller korrigeres,
- data skal kunne utleveres på et fornuftig format
- samtykke skal kunne tas tilbake (like lett som det ble gitt),
- samtykke skal bekreftes (personen skal vite at samtykke er gitt)
- data skal ikke lagres lenger enn nødvendig, lagres og behandles sikkert.
- Personvern skal være standardinnstillingen (opt-in heller enn opt-out).

Hensikten med forordningen er å sikre at personer skal ha kontroll på sine egne data, hva som finnes (samles inn) og hva informasjonen brukes til. Bedrifter skal ikke kunne selge videre personopplysninger uten at personen får vite og samtykker til det. Bedrifter skal tvinges til å opptre redelig og rettferdig, eksempelvis rundt samtykke. Det skal være like lett å trekke tilbake samtykke som at det er gitt, og det er krav til at personen skal vite at samtykke er gitt.

Poeng: 5 totalt

- 1: forklare at GDPR har med personvern å gjøre, plikter for de som ønsker å bruke personopplysninger
- 2: forklare hensikt
- 2: hva innebærer dette (feks mhp på samtykke, lagring overføring osv)

Oppgave 5: (10% av totalkarakter)
Forklar kort hva testdrevet utvikling er.
Forklar kort hva refaktorering er.
Forklar hvordan disse prinsippene påvirker utviklingsprosessen.

Test-drevet utvikling (TDD) handler om å drive utvikling ved hjelp av tester. Testen skrives først, før det finnes noe annen kode. Så skriver man den koden man trenger for å få testen til å kjøre, og for hver gang man får syntaksfeil (feks for en klasse ikke finnes) så kan den biten implementeres. Når koden så kompilerer vil testen feile, og forretningslogikken kan implementeres, slik at testen blir grønn. Dette tvinger utvikleren til å bruke sin egen kode i testen før den i det hele tatt eksistererer.
Et annet viktig prinsipp er at det ikke skal skrives mer kode enn nødvendig for å få testen til å bli grønn. Dette er for å sørge for at man ikke koder mer enn nødvendig for å få testen til å passere, slik at man ikke ender opp med store mengder kode uten testdekning.
En test skal være liten å kun dekke en spesifikk bit med funksjonalitet, så det ikke tar så lang tid å få testen til å kjøre.
TDD gjør at man helt fra begynnelsen må tenke på hva koden man skriver skal brukes til, og dette er viktig for å lage riktig kode. Men i tillegg blir koden lett å teste, og løsere koblet, slik at det er bedre separasjon (SRP) i koden, for ellers blir testene vanskelig å skrive. Dette fører til høyere kvalitet i koden.

Når testen(e) kjører, er det rom for å forbedre kvaliteten på koden, dette kalles refaktorering. Refaktorering er å endre struktur i koden uten å endre resultatet utad. Alle testene skal fortsatt kjøre.

Refaktorering gjøres for å heve kvaliteten på koden, øke lesbarheten og øke vedlikeholdbarheten på koden, slik at det blir lettere å jobbe med koden over tid.

Fokus på testing gjør at koden lar seg teste lettere og er mer frikoblet fra hverandre (mindre rigid kode). Dette gjør det lettere å gjøre endringer i koden over tid. Man må dog passe på at kvaliteten på test-koden også er bra, slik at ikke testene hindrer nødvendige endringer i koden (rigid, bare på annet nivå). Refaktorering gjort jevnlig sikrer at kodebasen reflekterer verden slik den er nå og at den lar seg utvide og endre videre. Forretningsregler og verden (både i forhold til behov og teknisk) endrer seg, da må koden også følge med for å fortsette å være relevant.

Poeng: 10 totalt

- 4: forklare TDD og red/green/refactor
- 2: forklare refaktoreringsprinsippet
- 6: påvirkning, feks helst ha med at testing fører til at koden blir brukt med en gang (må tenke brukeropplevelse på koden), refaktorering øker lesbarhet og vedlikeholdbarhet, kvalitetspåvirkning, trygghet for å gjøre endringer

Oppgave 6: (5% av totalkarakter)
Forklar hva working directory, staging area og local repository er i git. Forklar de viktigste fordelene ved å bruke versjonskontroll i programvareutvikling.

Working directory er arbeidsområdet ditt lokalt på disk der du har sjekket ut en gitt commit fra et repository. Staging area er der git får oversikt over alle endringene som er gjort og som skal legges til i git. Basert på disse endringene beregner git en hash som er summen av endringene som gjøres. Når du gjør git add <filer> blir endringene i filene tatt med fra working directory til staging area. Når du så gjør git commit vil alle endringene som er lagt til i staging area legges til i ditt lokale repository, hvor alle endringene du har gjort også er lagt til.

Versjonskontroll gjør at du sporer endringene du har gjort. Du får oversikt over hvilke endringer som er gjort, når, og av hvem. Du får backup hvis du har arbeidet på ulike enheter, du er garantert at det er samme versjon av koden så lenge alle har samme commit. Lettere å dele kode. Lettere å eksperimentere fordi du kan kaste endringene og gå tilbake til "forrige savegave" (forrige punkt der du vet at koden virker). Reduserer risiko ved å gjøre endringer og øker tryggheten når man utvikler (gitt ryddig commit-historikk og regelmessige commits).

Poeng: 5 totalt:

- 2: forklaring av working directory, staging area og local repository
- 3: fordeler ved å bruke versjonskontroll, feks deling, sporing av endringer, økt trygghet, lavere risiko generelt

Oppgave 1	**Poengsum: 10 totalt**	**MVP-oppgave**
	2	Høyest risiko teknisk
	2	Høyest risiko brukermessig
	2	Oppgaver med fullstack-implementasjon
	2	Tilbakemelding fra brukere
	2	Annet
Oppgave 2a	**Poeng: 6 totalt**	**Brukerhistorier – hva er det**
	3	Hvem/funksjonalitet/behov, akseptansekriterier
	3	1 p pr typiske feil, bør virkelig ha med «for stor» og/eller «for uklar» for ikke å få trekk
Oppgave 2b	**Poeng: 9 totalt**	**Lag brukerhistorier m/ oppgaver og kriterier**
	Max 3 pr brukerhistorie * 3 historier	1 p brukerhistorie, 1 p akseptansekriterier, 1 p for hvilke oppgaver som må løses
Oppgave 3	**Poeng: 5 totalt**	**Open-closed principle**
	2	Åpen for utvidelse, lukket for modifikasjon (med litt forklaring
	2	Forklare interface som løsning
	1	Meningsfullt klassediagram

Oppgave 4	**Poeng: 5 totalt**	**Personvernforordningen**
	1	Forklare GDPR = personvern, plikter for de som ønsker å bruke personopplysninger
	2	Forklare hensikt
	2	Praktiske konsekvenser (samtykke, lagring, overføring, sletting osv)
Oppgave 5	**Poeng: 10 totalt**	**TDD og refaktorering**
	4	Forklare TDD, red/green/refactor
	2	Forklare hva refaktorering er
	4	Påvirkning, feks at testing gjør at koden blir brukt med en gang, løsere kobling, lesbarhet, vedlikeholdbarhet, trygghet for å gjøre endringer osv.
Oppgave 6	**Poeng: 5 totalt**	**Versjonskontroll (git)**
	2	Forklare working dir, staging area og local repository
	3	Fordeler v versjonskontroll, som deling, sporing av endringer, lavere risiko/økt trygghet osv.