NTNU
Kunnskap for en bedre verden

Institutt for Datateknologi og Informatikk

TDT4140 - Programvareutvikling

Leveranse 9: Refleksjonsrapport

Kodebase:
gitlab.stud.idi.ntnu.no/tdt4140-2022/landsby-2/gruppe_22/groupup

Forfattere:

Fornavn	Etternavn	Studentmail
Line	Rosland	linehro@stud.ntnu.no
Peter Johan	Flått-Bjørnstad	peterjf@stud.ntnu.no
Joachim	Fredheim	joachibf@stud.ntnu.no
Camilla	Kopperud	camillkk@stud.ntnu.no
Thorbjørn	Lundin	tslundin@stud.ntnu.no
Jørgen	Sandhaug	jorgeksa@stud.ntnu.no
Martin	Skatvedt	martskat@stud.ntnu.no

4997 ord inkl. figurer.

Mai, 2022

Introduksjon

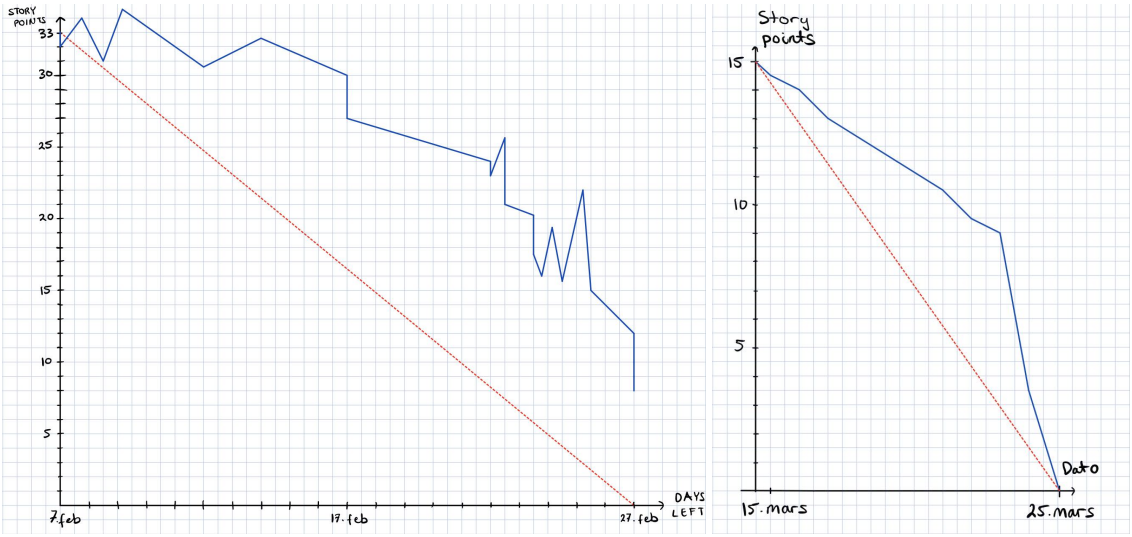
Vi har gjennomført et programvareutviklingsprosjekt i faget TDT4140 Programvareutvikling. Gjennom prosjektet har vi erfart hvordan det er å jobbe i et smidig utviklingsteam, og hvordan smidige praksiser påvirker utviklingsprosess og produkt. Etter grundig analyse av produkt, kodebase, produktkø, iterasjonskøer, brenndiagram og referater fra retrospektivmøter, har vi kommet frem til tre lærdommer knyttet til kravarbeid, utviklingsprosess og produktkvalitet. Denne rapporten drøfter disse lærdommene og trekker inn relevant analyse og refleksjon. Det vi ønsker å undersøke er hvordan vi kan bedre vår bruk av smidige utviklingspraksiser i fremtidige utviklingsarbeid.

1 Kravarbeid

1.1 Smidig kravarbeid

I løpet av utviklingsprosessen ble vi ofte nødt til å omprioritere og endre krav for å tilpasse oss kundens ønsker. I analysen av iterasjonskøene våre oppdaget vi at iterasjonskø 2, slik den var planlagt i forstudiet, i liten grad samsvarte med iterasjonskøen som ble fremarbeidet i iterasjonsplanlegging 2. Kun én av de opprinnelig planlagte brukerhistoriene ble inkludert, sammen med fire som opprinnelig tilhørte første iterasjon, samt to nye som ble utarbeidet grunnet endrende ønsker fra produkteier. En av de nye ønskene var funksjonalitet med høy prioritet i form av SuperGroupUp (se Vedlegg 1). Siden vi jobbet smidig og gjorde kravarbeid i iterasjoner, var vi godt rustet for å håndtere disse situasjonene. Dersom vi hadde brukt fossefallsmetoden i møte med kravarbeidet, der krav blir spesifisert i starten av prosjektet (Sommerville, 2016, s. 31), ville det vært vanskelig å rette oss etter endrende behov og uforutsette hendelser. Dette kunne i verste fall ført til et produkt som ikke var nyttig for kunden innen det var ferdig. Dette gjelder også for programvareutviklingsorganisasjoner, der krav "pleier å utvikle seg raskt og bli foreldet selv før ferdigstilling av prosjektet" (Cao & Ramesh, 2008, s. 60, vår oversettelse).

1.2 Brukerhistorier



(a) (b)

Figur 1: Brenndiagrammer for (a) iterasjon 1 og (b) iterasjon 2.

Ved sammelikning av brenndiagrammene så vi at utviklingen var jevnere i den andre enn den første iterasjonen (se Figur 1). Dette stemmer med opplevelsen vår av at vi jobbet mer parallelt i andre iterasjon enn i første, fordi vi ikke måtte vente på at andre brukerhistorier skulle bli fullført. Denne ventingen i første iterasjon var en konsekvens av at vi innførte brukerhistorier med avhengigheter. I retrospektiv 1 diskuterte vi at utviklingsoppgavene våre var for store og ofte avhengige av hverandre. Under planlegging av andre iterasjon fokuserte vi derfor på å lage uavhengige brukerhistorier og utviklingsoppgaver for å bedre teamets effektivitet. Denne utviklingen av brukerhistorier er i tråd med INVEST-kriteriene[1].

Brenndiagrammet fra første iterasjon viser at teamet ikke fikk fullført alle de planlagte brukerhistoriene. Dette skyldes trolig uforutsette hendelser som sykdom, mye opplæring og uventet mye tid på

[1]INVEST: I - uavhengig, N - kan forhandles, V - verdifull for kunden eller bruker, E - kan estimeres, S - liten og T - testbar. (Cohn, 2004, s. 17)

parprogrammering. Dessuten hadde vi ingen tidligere erfaringer å basere estimeringen på ("*yesterdays weather*"), som gjorde det vanskelig å ta høyde for slike hendelser. Avhengige brukerhistorier i iterasjon 1, som for eksempel registrering og innlogging av bruker, førte også til problemer med estimering (Cohn, 2004, s. 17). Siden vi i andre iterasjon unngikk slike avhengigheter, klarte vi å estimere mer presist. Dette ble også medvirket av at vi nå hadde lært av tidligere erfaringer. En bedre estimering i andre iterasjon var også noe vi merket oss som positivt i retrospektiv 2. I andre iterasjon møtte vi iterasjonsmålet vårt, og klarte derfor å oppfylle forventningene produkteier hadde belaget seg på. Dette viser viktigheten av god estimering.

1.3 Tilbakemelding fra brukere

I løpet av iterasjon 2 gjennomførte vi en variant av UAT (User Acceptance Test) med produkteier, der produkteier fikk prøve å navigere applikasjonen og gjøre spesifikke oppgaver mens vi observerte. Grunnen til at vi gjorde dette var både for at produkteier skulle få innsikt i funksjonaliteten til applikasjonen, og for at vi skulle få konstruktive tilbakemeldinger. Blant annet fikk vi høre at navigasjonsbaren ikke var intuitiv. Disse observasjonene og tilbakemeldingene gav oss innsikt, men siden vi allerede hadde utført kravarbeid for iterasjon 2, fikk vi ikke utnyttet dette så godt som ønsket. Testen fikk oss likevel til å innse at forretningsorientert testing, der man får tilbakemeldinger både fra kunde og brukere, er en viktig del av godt kravarbeid. Dette støttes av El Emam og Madhavji (1995, s. 75), som sier at brukermedvirkning er "en av de viktigste faktorene som bidrar til suksessen i kravarbeidsfasen" (vår oversettelse). Derfor tenker vi at i tillegg til å gjennomføre UAT jevnlig med kunden, burde vi gjennomført brukbarhetstester, et verktøy som havnet utenfor vårt fokusområde. Dette er en måte å involvere sluttbrukere, der de får teste applikasjonen og gi tilbakemeldinger. Under en slik prosess kan man få nyttig innsikt rundt mangler og feil, brukeropplevelse og brukervennlighet. Dette kan så brukes videre til utforming av nye krav for produktet. Dersom denne prosessen gjøres iterativt, får man stadig tilbakemeldinger fra brukere som kan lede utviklingen i riktig retning. Selv om vi sparte mye tid på ikke å gjennomføre iterative brukbarhetstester, ville det ha hjulpet oss med å utforme krav som hadde ført til et bedre sluttprodukt. Uten den innsikten man får fra brukerne, er det vanskelig å si om produktet som ble utviklet er relevant og har verdi for sluttbrukeren.

1.4 Kommunikasjon

Under analysen av produkt, iterasjonskø og produktkø oppdaget vi at produktet ikke stemte overens med alle punktene i iterasjonsmålene. Dette er fordi brukerhistoriene selv ikke alltid stemte med iterasjonsmålene. Eksempelvis tar en av brukerhistoriene for seg funksjonalitet som er irrelevant for iterasjonsmålet i iterasjon 1. I tillegg opplevde vi flere ganger under prosjektet, spesielt i iterasjon 1, at ulike teammedlemmer hadde ulike tanker rundt hva en brukerhistorie skulle innebære. Teamet hadde altså ikke gode nok felles mentale modeller av kravene, og dette kan grunne i en svak kommunikasjon om kravene med produkteier.

Når man skal utvikle et produkt, er det helt nødvendig med god kommunikasjon mellom utviklere og kunde, slik at utviklerne vet hva de skal utvikle. Dette støttes av en studie av Cao og Ramesh (2008) som sier at "studiedeltakerne identifiserte den intensive kommunikasjonen mellom utviklerne og kunder som den viktigste kravarbeid-praksisen" (vår oversettelse). Under deler av vårt kravarbeid var produkteier utilgjengelig, noe som førte til at teamet vårt måtte utarbeide krav mer selvstendig. Utviklerne tok dermed på seg en stor del av rollen til produkteier, og på noen måter også rollen som kunde (Kniberg, 2015, s. 17). Dette skapte en uvanlig dynamikk rundt kravarbeidet der vi stadig følte oss usikre på hvilke krav vi skulle sette til produktet. Under utvikling førte det også til usikkerhet rundt om funksjonaliteten vi implementerte faktisk ville tilfredsstille kunden. Siden kravarbeidet i stor grad var styrt av utviklerne, fikk vi en skjevfordeling i kommunikasjonen, der utviklernes behov gjerne ble satt over kundens. Dette førte til at tekniske detaljer og utviklernes ønsker fikk større betydning under kravarbeidet. Dette skaper ubalanse i kommunikasjonen som ifølge Cohn (2004, s. 3) kan være ødeleggende for prosjektet; "Hvis en av sidene dominerer kommunikasjonen, taper prosjektet" (vår oversettelse). Det vil være mer gunstig om både kunde og utviklere tar del og får påvirke kravarbeidet, slik at man kan ivareta kundens behov og ønsker,

samtidig som at utviklingen er gjennomførbar for utviklerne og tekniske problemer tas hensyn til.

1.5 Akseptansetest

En måte å få tydeliggjort krav fra produkteier kan være at produkteier sammen med teamet forhåndsdefinerer akseptansetester til alle brukerhistorier før iterasjonens start. Dette er tester som verifiserer at en brukerhistorie har blitt implementert i tråd med den opprinnelige betydningen (Cohn, 2004, s. 15). Underveis i prosjektet opplevde teamet at vi var usikre på om produktet vi utviklet var av verdi for produkteier. Det å definere akseptansetester på forhånd stemmer godt overens med Kniberg (2015, s. 42) sitt forslag om å lage akseptansekriterier for å sikre at utviklingsteamet og produkteieren har samme idé om hva en brukerhistorie skal inneholde, og dermed at historien er verdifull for kunden (V fra INVEST). I tillegg ville dette tiltaket kunne ha sikret T (testbar) for alle brukerhistoriene. Gjennom prosjektet testet vi utviklingsoppgavene, men ikke brukerhistoriene som helhet, og dette svekket produktet fordi vi ikke kan si sikkert om brukerhistorien har blitt riktig implementert. Teamet markerte brukerhistorier som ferdige uten å ha testet hele historien med spesifiserte tester, og dette kan ha ført til mangler i essensielle funksjoner som ikke har blitt oppdaget. Ved å skrive akseptansetester kan man oppdage avhengigheter mellom brukerhistorier, fordi man kan se om en test er avhengig av at en annen brukerhistorie var fullført. Dermed kan man enklere lage uavhengige brukerhistorier i tråd med I i INVEST. Å forhåndsdefinere akseptansetester fører til et mer omfattende og tidkrevende kravarbeid, men vi ville likevel innført dette i fremtidig kravarbeid fordi det trolig vil styrke både produkt og utviklingsprosess betraktelig.

1.6 Lærdom

Lærdommen vår er at *det er verdt å sette av nok tid til å utføre et godt kravarbeidet.*

Vi har tydeliggjort at for at prosjektet skal kunne lykkes er det essensielt med godt kravarbeid, selv om dette er tidkrevende. Viktige ting å ta med seg er at kravarbeid skal gjøres iterativt, og man må bestrebe en balansert og hyppig kommunikasjon mellom utviklere og kunde. Dessuten er det viktig med kontinuerlige tilbakemeldinger fra sluttbrukere for å forbedre krav. I tillegg er det viktig å utvikle brukerhistorier uten store avhengigheter, samt ha tydelige akseptansetester for å sjekke om en brukerhistorie er tilfredsstilt. Dette arbeidet behøver i stor grad en aktiv produkteier. I videre utviklingsarbeid vil vi ha fokus på disse punktene for å bedre utviklingsprosessen, for å lettere utvikle et produkt som tilfredsstiller kundens behov samt er relevant for sluttbrukeren.

2 Utviklingsprosess

2.1 Scrum

Scrum-praksisen består av mange aspekter, og beskrives av Kniberg (2015, s. 21) som en verktøykasse. Dette innebærer at man ikke alltid tar i bruk alle verktøyene konsekvent, men finner frem til hva som fungerer best for teamet. Digital.ai sin årlige rapport fra 2021 viser til at 66% av teamene tar i bruk Scrum, men at kun 3% følger det konsekvent (VersionOne, 2021). Dette kan være et tegn på at verktøyet kan forbedres, eller mer sannsynlig, at hvert team aktivt tilpasser Scrum til sin arbeidsplass og teamets behov. Vi visste tidlig at vi ønsket finne Scrum-verktøyene som passet best for teamet, og det viste seg at gevinsten av å eksperimentere ikke bare kom etter vi hadde funnet en bra løsning, men også av det vi lærte underveis.

Ett av de største eksperimentene vi gjorde var rotering av scrumfasilitator. I første iterasjon byttet vi fasilitator mellom hvert daglige møte. Vi så verdien av at alle fikk prøve rollen, ettersom det skal gjøre rollen mindre fremmed (Stray mfl., 2020, s. 75), samt at alle fikk verdifull erfaring til arbeidslivet. Til tross for fordelene, opplevde vi under denne iterasjonen at de daglige møtene, og diskusjonsmøtene som ofte fulgte etter, hadde ingen fast struktur. Som et uerfarent scrumteam var det vanskelig å vurdere om årsaken var mangel på møte-erfaringer, eller om den roterende fasilitator-rollen ikke passet for teamet. Vi kunne fortsatt med roterende fasilitator og se om effektene forbedret seg over tid, men valgte under retrospektiv 1 å istedet gå over til fast scrumfastilitator. Et viktig punkt ved eksperimentering er at undersøker effekter av endringen (Rigby mfl., 2016, s. 9), og vi opplevde problemene ble fort forbedret med én fasilitator. Dette kom tydelig frem i andre retrospektiv, og med analysen gjort i ettertid innså vi at effekten kanskje var mer positiv enn forventet: Selv med strengere møteagenda og straffer var folk generelt mer fornøyde med hvordan scrumfasilitator håndterte avvik, noe som kanskje ikke hadde skjedd hvis det var kun én fasilitator fra starten av. Nå som alle hadde prøvd rollen, hadde man fått mer respekt for personen som måtte jobbe fast med den. Kniberg (2015, s. 74) mener det er viktig å eksperimentere med Scrum-metodene, og vi tenker nå at det ikke kun er for å finne frem til beste løsning, men at teamet får verdifulle erfaringer på veien.

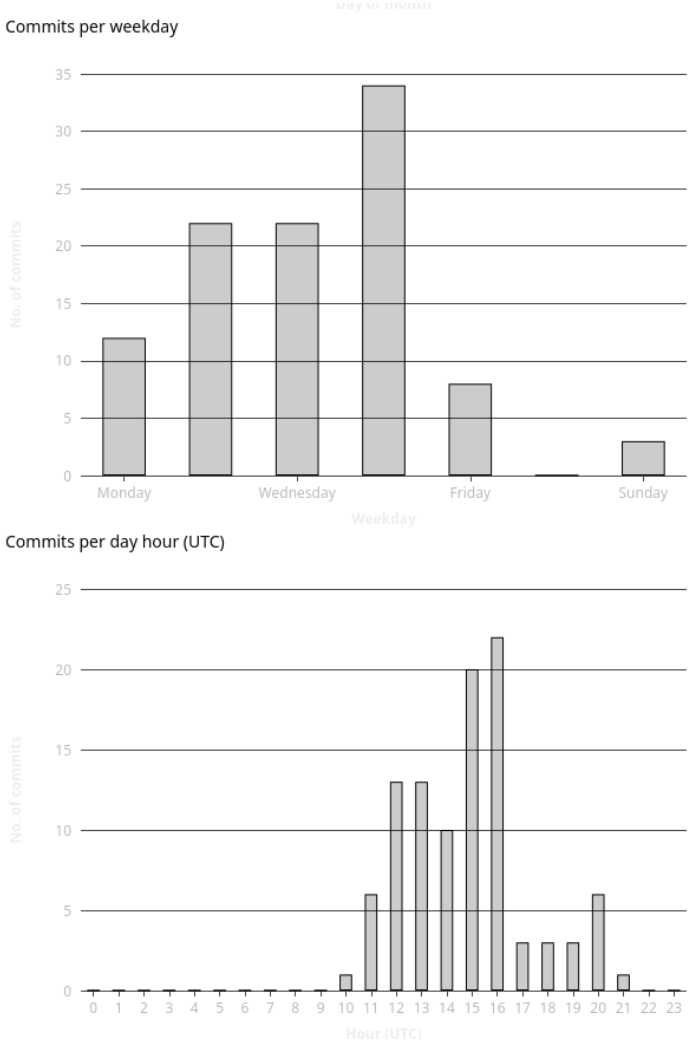
Et område som vi eksperimenterte mindre på, var bruken av grafikk under møtene. Hvert daglige møte inkluderte *de tre spørsmålene* (Kniberg, 2015, s. 74), og etter møtet ble det tatt opp eventuelle punkter knyttet til prosjektet. Møter som ikke inkluderte brenndiagram, kodeoppgaver på GitLab eller annen grafikk hadde oftest digresjoner, noe vi innså først i analysen etter prosjektet. Selv om Kniberg (2015, s. 74) nevner at et overbruk av grafikk kan være forstyrrende, tenker vi at det for oss eksisterer en nedre grense. Dette var tydelig et område som kunne forbedres ved å teste ulik grafikk.

Retrospektivmøtene hadde stort sett likt format, men selv uten eksperimentering opplevde vi at møtene ble bedre. I begge retrospektivene tok vi utgangspunkt i samme tidsskjema og aktiviteter fra Kniberg (2015, s. 85). Selv om Kniberg anbefaler sterkt å variere møtene for å holde dem spennende, valgte vi å gjennomføre det samme til vi ble gode på hva vi gjorde. I motsetning til daglige møter, var retrospektivmøtene mer omfattende, og gjennomført kun to ganger i løpet av prosjektet. Meyer (2018, s. 94) nevner at man bør bli godt kjent med de vanligste praksisene før man forsøker å bytte metoder, og vi opplevde at å gjøre de samme oppgavene ga oss muligheten til å forbedre oss før vi prøvde noe nytt.

2.2 Andre praksiser

2.2.1 Parprogrammering

Et viktig verktøy som ble brukt av gruppen under prosjektet var parprogrammering. Parprogrammering er en metode lånt fra ekstrem programmering og innebærer at to utviklere jobber sammen



Figur 2: Antall commits per ukedag (øverst) og antall commits per time i døgnet (nederst).

på samme datamaskin, der én tar rollen som navigatør[2] og én tar rollen som kjører[3] (Kniberg, 2015, s. 105). Et krav til prosjektet var at alle skulle parprogrammere med alle, og gruppen prioriterte derfor dette.

Vi identifiserte tidlig i prosjektet, under forarbeidet, at gruppen hadde en svært varierende erfaring med de valgte rammeverkene (se Vedlegg 1), og derfor bar parprogrammeringen i iterasjon 1 preg av å ha vært et pedagogisk verktøy, som kan minne om det som kalles *mentoring* (Meyer, 2018). Gruppen var tidlig klar over at valget om å aktivt bruke parprogrammering ville medføre en mulig alternativkostnad i form av mindre progresjon på prosjektet tidlig i forløpet, men tenkte at det kunne gi fordeler i form av jevnere fordeling av arbeid, bedre læringsutbytte, bedre teamkommunikasjon og større eierskap til prosjektet. Under sluttanalysen ble det tydelig at disse fordelene hadde inntruffet, og var spesielt synlig i retrospektivene, der mange av notatene nevnte de positive effektene av parprogrammering. I analysen observerte vi også at mest arbeid ble gjort på normale arbeidstider (se Figur 2), selv om flere i starten antydet at de foretrakk å jobbe selvstendig sent på kvelden. Vi tenker at parprogrammeringen kan ha oppfordret til å møtes på dagen, og jobbe sammen på like tidspunkter, likt som en arbeidsplass. Til slutt kom vi frem til at praksisen har delvis fungert som en uformell kodeinspeksjon og dermed øke kodekvalitet, noe Sommerville (2016, s. 71) også nevner som en fordel.

I iterasjon 2 ble parprogrammeringen mer slik den formelt sett skal være. Selv om vi aldri tok en

[2]Fra engelsk, "navigator"
[3]Fra engelsk, "driver"

konkret beslutning på det, utviklet parprogrammeringen seg til en naturlig inndeling i kjører og navigatør, der én i større grad lette etter dokumentasjon og påpekte eventuelle mangler mens den andre skrev kode (Sommerville, 2016, s. 66). Likevel ville det vært bedre om vi på dette tidspunktet hadde tatt et valg om å bruke denne inndelingen formelt, istedet for å blindt anta at alle brukte praksisen på like effektiv måte. Grunnen til at vi ikke gjorde dette var at vi på dette stadiet var fornøyde med hvordan parprogrammeringen hadde gått, noe vi kan være mer kritisk til i fremtiden. Likevel tenker vi at vi fant en form for parprogrammering som passet oss som team. Det er ikke alltid nødvendig å følge praksis slavisk, og man må forme den etter teamets behov (Kniberg & Skarin, 2010, s. 17).

2.2.2 Kanban

Videre brukte vi verktøy fra Kanban, spesifikt kanban-tavler. Kanban-tavler en er en av de tre hoveddelene i kanban, og står for visualisering av arbeidsflyten. (Kniberg & Skarin, 2010, s. 4). Kanban-tavlene går ut på å ha en tavle med kolonner, hvor hver kolonne representerer hvor i arbeidsflyten en oppgave er. Under hele prosjektet brukte vi GitLab sin innebygde kanban-tavle. Der bygde vi opp egne tavler med kolonnene: *open*, *in progress*, *needs review* og *review fixes*. Å blande verktøy fra Scrum og Kanban er et kjent fenomen, da svært få team går for rene praksiser og ender ofte opp med en *Scrum-inspirert* eller *kanban-inspirert* praksis (Kniberg & Skarin, 2010, s. 10).

Under prosjektarbeidet var vi splittet på bruken av kanban-tavler, noen foretrakk å bruke listen over *issues* som GitLab har innebygd, mens andre foretrakk kanban-tavlen. Dette gjorde at vi fikk bruke den måten vi foretrakk best for å visualisere arbeidsflyten, men kom på bekostning at vi måtte synkronisere begge, og la til ekstra kostnad i prosessen. For oss la dette til rette for at vi hele tiden hadde god oversikt over hvordan vi lå ann i iterasjonen. I tillegg kunne vi se hvor mange oppgaver som var igjen, og hvor mange oppgaver som var *WIP*.

Under iterasjon 2 av prosjektet minsket bruken av kanban-tavlen. En årsak til dette kan være at ikke alle fant kanban-tavlen nødvendig, og mistet fokuset på bruken av den under iterasjon 1. Men vi fortsatte å vedlikeholde tavlen, selv om den ble mindre brukt. En annen grunn kan være at listen over utviklingsoppgaver medførte nok visualisering, slik at vi ikke trengte kanban-tavlen i samme grad. Likevel tror vi ikke at dette hadde en stor innvirking på det endelige resultatet, da vi ikke brukte praksisene som ofte følger med kanban-tavler, som begrensing av antall WIP oppgaver.

2.3 Lærdom

Vi har lært at *det ikke eksisterer én smidig praksis som vil fungere for alle, og at hvert team må systematisk eksperimentere innenfor én eller flere praksiser for å oppnå en god utviklingsprosess.*

I noen situasjoner kan man fint introdusere ulike konsepter uten systematisk organisering, slik vi gjorde med parprogrammering. Andre ganger må man vurdere om tiden investert er verdt bekostningen på produktet, slik det var med roterende scrumfasilitator og Kanban-tavler. I fremtiden vil vi fortsette å prøve ulike smidige metoder, men på en mer kontrollert måte.

3 Programvarekvalitet og produktegenskaper

3.1 Teknologivalg og programvarearkitektur

Produktet vårt er en mobil-rettet webapplikasjon, bestående av et utvalg av dagens mest kjente teknologier (se Vedlegg 1). Vi valgte disse basert på hva vi forutså de tekniske kravene til produktet ville være, samt hvilke teknologier teamet hadde erfaring med, da prosjektet i utgangspunktet ikke hadde rom for at man skulle lære helt nye teknologier. Produkteiers originale produktbeskrivelse la sterkt trykk på et ønske om et pent design. Vi utarbeidet derfor et design i Figma, et populært verktøy for å lage brukergrensesnitt, som vi deretter implementerte med en dynamisk *frontend* stakk bestående av React og Chakra-UI, da medlemmer av teamet hadde jobbet med disse tidligere og erfarte dem som godt egnet for å lage et produkt med ønskede egenskaper. Disse valgene formet utviklingsprosessen vår, å bruke teknologier flere av medlemmene hadde kjennskap til bidro til å få igang utviklingsprosessen raskt, og gjorde det enklere for de mindre erfarne medlemmene av teamet å søke hjelp, for eksempel gjennom parprogrammering.

God arkitektur styrker programvarekvalitet. Dersom arkitektur prioriteres og teamet bruker tid på forarbeidet, kan koden bli mer robust og gjenbrukbar (Waterman, 2018, s. 100). I begynnelsen av iterasjon 1 brukte teamet mye tid på å bygge opp en stabil grunnmur. Det er vanskelig å forestille seg sluttproduktet i starten av en smidig utviklingsprosess, og vi opplevde at god planlegging gagnet oss senere. I tillegg studerte vi ulike utfall, som medførte at risikoen ble mindre og produktet blir mer tilpasningsdyktig og vedlikeholdbart (Waterman, 2018, s. 100). Produkteier forespurte for eksempel en ny funksjonalitet underveis, *SuperGroupUp* (se Vedlegg 1), og vi trengte kun noen få endringer for å oppgradere match-funksjonaliteten vår. Dessuten besto teamet av flere uerfarne utviklere, og vi opplevde at mye planlegging ga teamet større forståelse av produktet. Dette medførte også bedre kommunikasjon og samarbeid.

På den andre siden kan eksessiv planlegging medføre overflødig ingeniørarbeid (Abrahamsson mfl., 2010, s. 16), og man kan lage funksjonalitet som produkteier senere ikke trenger. I en smidig utviklingsprosess tilpasser man seg kunden fortløpende. Dersom man tar tekniske valg før utviklingsprosessen begynner, bryter man med smidige utviklingsprosesser hvor valg skal tas underveis i utviklingen. Plutselig ønsker kunden å gjøre store endringer, og da er det lurt å bygge opp strukturen slik at dette enkelt kan løses. Av og til er det også en stor fordel å komme først ut på markedet, og da kan det være lurt å redusere forarbeidet (Kniberg, 2015, s. 18). Brukerne av produktet ser ikke forarbeidet og arkitekturen, og det kan derfor være lurt å prioritere utvikling av funksjonalitet. Dette vil vi tenke på i senere utviklingsprosjekter.

Vi har nevnt at avhengige brukerhistorier og utviklingsoppgaver skapte problemer. Dette gjelder ikke bare for kravarbeid, men også for programvarekvalitet. Eksempelvis gjorde en person implementering av gruppeprofilside samtidig som en annen skrev koden som ville hente den tilhørende informasjonen fra databasen og sende den til klienten. I disse tilfellene opplevde vi økning i teknisk gjeld, noe som svekket programvarekvaliteten. Likevel var det en stor fordel at den todelte arkitekturen tillot oss å jobbe parallelt da vi fortere kunne levere et produkt med ønskede egenskaper. Det vil derfor være viktig å jobbe for at utviklingsoppgaver kan utføres i parallell, uten å være for avhengige av hverandre.

3.2 Testing gir robust kode

Gjennom prosjektet valgte vi å prioritere å utvikle funksjonaliteter til produktet fremfor å ha høy testdekning. I begge iterasjonene benyttet vi oss av enhetstesting, men brukte det i mindre grad enn planlagt da det krevde mye opplæring og var derfor veldig tidkrevende. Fordelen med å fokusere på utvikling av produktegenskaper er at produktet har mange funksjonaliteter og derfor kan framstå bedre enn andre produkter. Ulempen er at vi fikk mye teknisk gjeld og produktet ble upålitelig – produktkvaliteten ble svekket.

Videre kan man argumentere for at økt risiko på et produkt ikke nødvendigvis er et stort problem. Hadde vi utviklet en pacemaker hadde konsekvensene vært fatale, men vi har utviklet en sosial

platform, hvor enkelte problemer ikke nødvendigvis vil påvirke brukerens opplevelse. På en annen side er det alltid viktig å ha et pålitelig produkt for å øke etterspørselen av produktet på markedet.

På retrospektivmøte etter iterasjon 2 diskuterte vi fraværet av testing, og et aksjonspunkt til iterasjon 3 ble å teste all essensiell funksjonalitet. Dette gjorde vi selv om det kan være vanskelig å se behovet for testing fra et forretningsperspektiv. Vi erfarte at etterhvert som produktstørrelsen vokste, vokste også behovet for å redusere teknisk gjeld og behovet for å lettere kunne vedlikeholde produktet.

Med et forretningsperspektiv, i motsetning til et teknisk perspektiv, kan det være vanskelig å se verdien av testing. Vi har tidligere nevnt at vi gjennomførte en UAT med produkteier i løpet av iterasjon 2. Selv om vi ikke fikk utnyttet denne prosessen maksimalt, gjorde det oss oppmerksomme på potensialet ved forretningsorientert testing. En type forretningsorientert testing som direkte påvirker produktkvalitet er exploratory testing (ET). ET er en form for testing der testeren stadig designer og utfører tester og analyserer resultatene(Crispin & Gregory, 2009, s. 102). Ved slutten av hver iterasjon forsøkte vi å gjennomføre ET, der noen utviklere prøvde å teste alle aspekter ved applikasjonen, i et forsøk på å oppdage problemer eller bugs. Dette gjorde at vi oppdaget flere problemer som deretter kunne fikses. Selvom vi ikke fikk fullt utbytte av forretningsorientert testing i vår utvikling, tenker vi at det er et verktøy som bør benyttes ettersom det fremmer programvarekvalitet.

Kniberg diskuterer test-drevet utvikling (TDD) som et alternativ til manuell testing. TDD kan være vanskelig, men en god teknikk å benytte, særlig til utvikling av ny kode. Oppsettet blir mer komplekst, men produktkvaliteten øker. (Kniberg, 2015, s. 105–108) Ved å benytte test-drevet utvikling kan det blir lettere å se fremgang underveis fra et forretningsperspektiv, og testing flettes kanskje bedre inn i den smidige utviklingsprosessen. Med dette tatt i betraktning, kan det være interessant å utforske test-drevet utvikling. Som nevnt tidligere, er det viktig å hele tiden utforske nye metoder for å kunne perfeksjonere den smidige utviklingsprosessen.

3.3 Lærdom

Lærdommen vår er at *arkitektur, gode teknologivalg og tester er viktig for å styrke programvarekvaliteten og muliggjør ønskede egenskaper, men kan hindre den smidige utviklingsprosessen.*

Høy programvarekvalitet og antallet egenskaper til produktet påvirkes av tidsbruken på valg av arkitektur og bruken av testing. Det finnes ingen fasit på hva man bør legge vekt på, og i fremtidige utviklingsprosjekt vil vi ha fokus på å tilpasse utviklingsprosessen til vår fordel, slik at det blir høyest mulig programvarekvaliteten og at produktet får ønskede egenskaper. Produktet og markedet påvirker også hvor viktig programvarekvaliteten er.

Noe planlegging og testing er viktig for å lettere koordinere teamet og redusere risiko, men for å opprettholde en smidig utviklingsprosess er det lurt å utsette valg og ikke gjøre unødvendig arbeid. Videre kan det være interessant å forsøke å praktisere TDD for å utforske nye metoder. Det er viktig å kunne utvikle prosessen og tilpasse seg til teamets beste.

4 Konklusjon

Gjennom rapporten har vi undersøkt hvordan vi kan bedre vår bruk av smidige praksiser i fremtidige utviklingsarbeider. I analysen av prosjektet vårt så vi hvordan en svakhet i kravarbeid førte til en mindre effektiv utviklingsprosess og et potensielt mangelfullt produkt. Vi oppdaget at å eksperimentere rundt hvordan vi brukte Scrum, Kanban og andre smidige praksiser påvirket prosjektet, og hvordan produktkvaliteten vår ble påvirket av våre valg av teknologi, arkitektur og testing. På grunnlag av analysen, egne refleksjoner og relevant fagstoff kom vi frem til følgende tre lærdommer:

- Det er verdt å sette av nok tid til å utføre et godt kravarbeidet.
- Det eksisterer ikke én smidig praksis som vil fungere for alle, og hvert team må systematisk eksperimentere innenfor én eller flere praksiser for å oppnå en god utviklingsprosess.
- Arkitektur, gode teknologivalg og tester er viktig for å styrke programvarekvaliteten og muliggjør ønskede egenskaper, men kan hindre den smidige utviklingsprosessen.

I fremtidige utviklingsprosjekter vil vi ha fokus på disse tre lærdommene fordi vi mener at de vil føre til en bedre utviklingsprosess og et bedre produkt som oppfyller kundens ønske. Ved å sette av nok tid til å gjennomføre kravarbeid vil vi kunne få et riktigere bilde av kundens ønske, for eksempel ved å forhåndsdefinere akseptansetester, samt å produsere et produkt som er nyttig for brukeren, som kan oppnås med brukertester. Et slikt kravarbeid vil forbedre arbeidsprosessen til utviklerene ved at det vil være enklere å danne uavhengige brukerhistorier slik at man kan jobbe parallelt og enklere kunne estimere riktig tidsbruk.
Systematisk eksperimentering med smidige praksiser i fremtidige utviklingsprosjekter vil føre til at man kan finne hvilke verktøy som passer best for akkurat det teamet man er medlem av da, slik at teamet både får en bedre utviklingsprosess og nyttige erfaringer på veien. Praksiser man kan eksperimentere med er bruk av ekstremprogrammerings-praksiser som parprogrammering, scrumpraksiser som scrumfasilitator og formen på møter som retrospektiv, samt bruken av grafikk som kanban-tavler.
Ved i fremtidige prosjekter å bruke tid på å ta gjennomtenkte valg av teknologier og arkitektur styrker man muligheten til høy programvarekvalitet, samt at det vil bli enklere å implementere produktegenskaper kunden ønsker. Likevel vil et stort tidsbruk på dette i starten av et prosjekt kunne hindre en smidig utviklingsprosess fordi man kan få vanskeligheter med å tilpasse seg endrede krav underveis i utviklingen. Vi vil derfor i fremtiden ha fokus på å ikke bruke mer tid enn nødvendig på valg av teknologier og arkitektur i starten av prosjektet da dette kan kunne føre til unødvendig ingeniørarbeid. Vi vil vurdere formen for, og graden av, testing i fremtidige prosjekter med hensyn på markedet og om nye funksjonaliteter eller programvarekvalitet er viktigst.

Pensumlitteratur

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc.

Crispin, L. & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams* (1. utg.). Addison-Wesley Professional
OCLC: ocn243543532.

Kniberg, H. (2015). *Scrum and XP from the Trenches - 2nd Edition*. InfoQ. www.infoq.com/minibooks/scrum-xp-from-the-trenches-2

Kniberg, H. & Skarin, M. (2010). *Kanban and Scrum: making the most of both*. C4Media.

Meyer, B. (2018). Making sense of agile methods. *IEEE Software*, *35*(2), 91–94.

Rigby, D. K., Sutherland, J. & Takeuchi, H. (2016). Embracing Agile.

Sommerville, I. (2016). *Software engineering* (10. ed., global ed). Pearson.

Stray, V., Moe, N. B. & Sjoberg, D. I. K. (2020). Daily Stand-Up Meetings: Start Breaking the Rules. *IEEE Software*, *37*(3), 70–77.

Waterman, M. (2018). Agility, risk, and uncertainty, part 1: Designing an agile architecture. *IEEE Software*, *35*(2), 99–101.

Eksterne kilder

Abrahamsson, P., Babar, M. A. & Kruchten, P. (2010). Agility and Architecture: Can They Coexist?, Volume (27), Issue (2).

Cao, L. & Ramesh, B. (2008). Agile requirements engineering practices: An emprical study. *IEEE software*, *25*(1), 60–67. https://doi.org/10.1109/MS.2008.1

El Emam, K. & Madhavji, N. (1995). A field study of requirements engineering practices in information systems development. *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, 68–80. https://doi.org/10.1109/ISRE.1995.512547

VersionOne, C. (2021). 15th annual state of agile report. *collab. net*. https://info.digital.ai/rs/981-LQX-968/images/RE-SA-15th-Annual-State-Of-Agile-Report.pdf

Vedlegg

1 Teknisk beskrivelse

1.1 Problembeskrivelse

Produkteier ønsker en plattform for vennegjenger med felles interesser å finne, samt planlegge møter med andre vennegjenger.

1.2 Tekniske krav:

- Opprettelse, sletting, redigering og autentisering av brukere
- Opprettelse, sletting og redigering av grupper
  - Innebærer også å bestemme gruppens interesser, som er måten man finner passende grupper.
- En søkemotor for å finne andre grupper
  - Søk skal kunne gjøres på interesse, sted, møtedato og alder
- Funksjon for å håndtere GroupUp-forespørsler
  - Endring i spesifikasjon fra produkteier underveis: I tillegg til vanlig GroupUp-forespørsel, skal man håndtere SuperGroupUp-forespørsel som krever et GruperUp Gold-medlemsskap.
- Sette møtedato mellom to grupper som har GroupUpet
- Administrering av grupper/brukere

IKKE tekniske krav:

- Chatfunksjon
  - Tidkrevende å implementere på et nivå som kan konkurrere med andre plattformer. Bør implementeres først når man ser at produktet har verdi for brukere. Nåværende løsning er å legge ved kontaktinformasjon.
- Betalingsløsning
  - Plattformen skal demonstrere at brukere har ulik tilgang basert på en betalingsløsning, men selve betalingsløsningen skal implementeres først når man ser at produktet har verdi for brukere.
- Grensesnitt for større enheter
  - Grensesnittet skal i utgangspunktet designes for smarttelefoner, med muligheten for å utvide til større enheter på et senere tidspunkt.

1.3 Ordliste

Rammeverk: En samling verktøy / kodebibliotek for å produsere mer kompliserte systemer

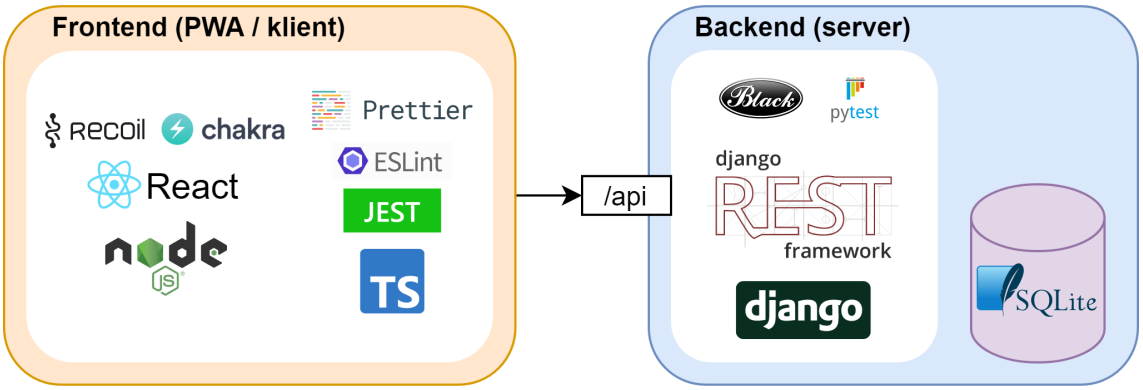
Progressive Web App (PWA): En nettside som kan lastes ned og kjøres som en mobilapplikasjon. Tilgjengelig på flere nettlesere i dag (f.eks. *Starbucks* og *Twitter*). Løsningen lar deg bruke web-rammeverk for å designe grensesnitt for en mobilapplikasjon.

Serialisering: Oversetting av data til ulike format, f.eks. database-innhold til nettverks-data (*JSON*)

REST-API: Standarder for å produsere et forutsigbart og pålitelig logisk grensesnitt over nettet.

1.4 Vår løsning

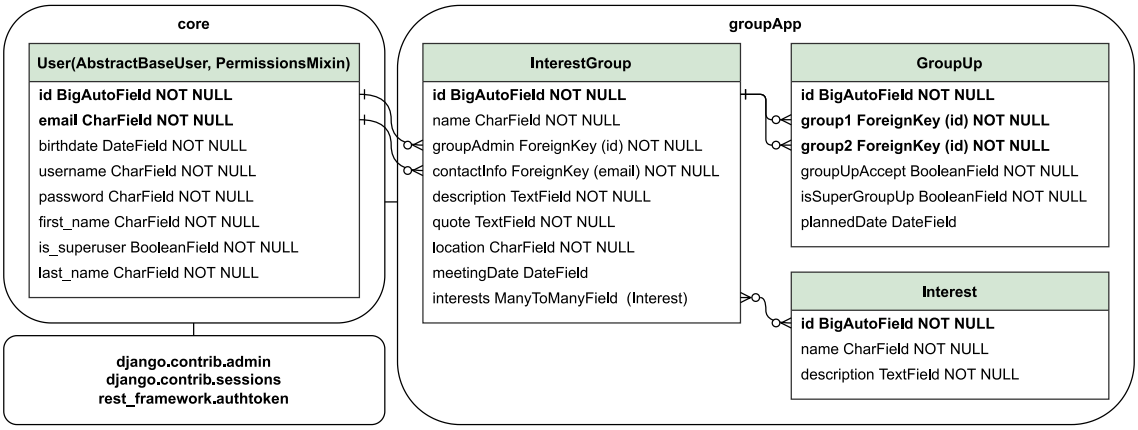
Vår løsning oppfyller de tekniske kravene med en plattform delt opp i to deler: En webapplikasjon tilgjengelig som nettside og som mobilapplikasjon, videre referert til som *PWA* eller klienten, og en backend med et *REST-API* for å håndtere kjernelogikk og lagring, videre referert til som serveren.



Figur 3: Teknologistakk for frontend og backend.

1.5 Datamodell

Datamodellen er implementert på serveren med *Django*, et skalerbart rammeverk for å knytte databaser, kjernelogikk og presentasjon over nettet. Følgende diagram viser hvordan de ulike entiteter er knyttet sammen i databasen:



Figur 4: Datamodell for kjernelogikk.

1.6 Logikk

For at mobilapplikasjon skal kommunisere sikkert med andre enheter, bør kjernelogikken håndteres av en ekstern server. Klienten kan opprette brukere, logge inn, sende GroupUp-forespørsler og mer ved å kommunisere med serveren gjennom et REST-API. Eksempler noen kall følger nedenfor.

/api/

POST groups/

```
{   "name" : "Ekstremfiske",
    "description" : "Hver søndag kveld, Nidelven.
    ↪  Vi sees",
    "quote" : "Fisking for de ekstreme",
    "interests" : "Fisking,Fisk,Svømming",
    "location" : "Trondheim",
    "meetingDate" : "2022-05-29" }
```

201 CREATED

```
{   "..." : "...",
    "id" : 1,
    "groupAdmin" : 1,
    "contactInfo" :
        "user1@email.com"
        ↪  }
```

GET groups/1/findGroupUp/?interesse=fisking,dansing — 200 OK

```
[
    {   "id" : 4,
        "name" : "Allværsfiskerne",
        "interesser" : ["fisking", "..."]
        "..." : "..." },
    {   "id" : 11,
        "name" : "Break-Up",
        "interesser" : ["dansing", "..."]
        "..." : "..." },
    { "...", "..." }
]
```

GET groupups/2 — 200 OK

```
{   "group1" : { "id" : 4,
        "name" : "Allværsfiskerne",
        "interesser" : ["fisking", "..."]
        "..." : "..." }
    "group2" : { "id" : 11,
        "name" : "Break-Up",
        "interesser" : ["dansing", "..."]
        "..." : "..." },
    "isSuperGroupUp" : false,
    "groupUpAccept" : true }
```

Disse er implementert på samme server med en utvidelse for Django kalt *Django REST framework (DRF)*. DRF er lagd for å knytte kjernelogikk til et REST-API, og gjør at vi kan bytte klienten (PWA) med f.eks. et Windows-program og fortsatt bruke det samme API-et. DRF tilbyr også autentisering og token-håndtering, slik at PWA-en kan bruke API-et sikkert uten å skrive passordet sitt for hver knapp man trykker. Til slutt har Django en konfigurerbar side for administrering av databasen, som betyr at rapporterte brukere kan undersøkes i administrator-panelet, og slettes ved behov.

1.7 Presentasjon

Som implisert over, er presentasjon ikke gjort av Django, men et heller av et eget rammeverk. Klienten er bygd på *React*, et bibliotek for å designe intelligente nettlesergrensesnitt. React bruker *React Router* for navigasjon, *Recoil* for tilstandshåndtering, *ChakraUI* for design, og *Node-fetch* for kommunikasjon med API-et.

1.8 Skalering

Rammeverkene er valgt med tanke på at plattformen skal tjene et økende antall brukere. Per nå lagres databasen på samme server som håndterer kjernelogikken (*SQLite*), men Django lar deg endre dette til en ekstern databaseserver (f.eks. *PostgreSQL*) når dette blir nødvendig. Styling-rammeverk som ChakraUI, tilstandshåntering som Recoil, og lint-verktøy sikrer at koden i hver fil er uniform. Til slutt er hele klienten skrevet i *TypeScript*, som sikrer at kompliserte objekter og funksjoner håndteres på en forutsigbar måte.

1.9 Testplan

Testing styrker produktkvaliteten, og i løpet av utviklingsprosessen skal det tas i bruk enhetstester, user acceptance tests (UAT) og utforskende tester (ET). *Jest* lar deg skrive tester for interaksjon med React-grensesnitt og teste om knapper og API-kall oppfører seg som forventet, og er derfor ideell for enhetstesting på klienten. For serveren har Django et innebygd testrammeverk, og kan brukes sammen med DRF for å se at kjernelogikk, autentisering og databasen reagerer riktig på API-kall. Slike logiske tester skal kjøres hver gang kode skal inn i kodebasen, oppnådd med *GitLab* sin innebygde CI. For grensesnittet skal UAT sjekke om produktegenskapene oppfyller kravene til produkteier, samt gjennomføre utforskende tester for å systematisk oppdage uventet oppførsel.