Sertifisert Tester

Foundation Level Extension Pensum
Agile Tester

Norsk versjon 2015.N1
Basert på Engelsk versjon 2014

International Software Testing Qualifications Board

Norwegian Testing Board

Foundation Level Extension oversettelse: Norwegian Testing Board (www.istqb.no)

Endringshistorikk

Version	Date	Remarks
2015.N1	3.6.2015	Første offentlige versjon på norsk

Innhold

Takk

Originalen til dette dokumentet er på engelsk og ble produsert av en internasjonal arbeidsgruppe. Den engelske originalen er tilgjengelig på www.istqb.org, og alle forfattere er nevnt der.

Oversettelsen ble gjort av medlemmer i Norwegian Testing Board i 2015. Disse er: Thomas Borchsenius, Egil Gullbekkhei, Skule Johansen, Tor Kjetil Moseng, Hans Schaefer og Camilla Tveter.

Generell oversettelse fra engelske til norske begrep		
	Engelsk	**Norsk**
1	business representatives	representanter fra forretningsområdet
2	feature	Funksjon eller egenskap
3	whole-team approach	Tilnærming som involverer hele teamet
4	test charter	test charter
5	personals	persontyper
6	release	Leveranse, av og til release
7	build server	Byggeserver
8	backlog	kø
9	team	team
10	agile	smidig
11	syllabus	pensum, pensumliste
12	business	forretning
13	burndown charts	burndown-grafer

0. Introduksjon

0.1 Hensikten med dette dokumentet

- Dette pensumet er grunnlaget for sertifisering iht. ISTQBs regelverk på Foundation-nivå for Agile Tester. ISTQB® og NTB gjør dette pensumet tilgjengelig for følgende formål:
  - Til eksamensorganisasjoner for at disse kan lage eksamensspørsmål på norsk i overensstemmelse med læremålene.
  - Til bedrifter som tilbyr kurs for å veilede og i oppbygging av kursmateriell og læringsmetoder
  - Til kandidater som vil forberede seg til eksamen (som del av et kurs eller som privatister).
  - Til profesjonelle som utvikler, vedlikeholder og tester programvare for å bidra til å utvikle yrkespraksisen og som basis for bøker og artikler.

- Forutsatt forhåndsgodkjenning kan dette pensumet også benyttes til andre formål.

0.2 Oversikt

- Dokumentet “Foundation Level Agile Tester Overview” [ISTQB_FA_OVIEW] inneholder følgende informasjon:
  - Målsettingene for syllabus
  - Oppsummering av syllabus
  - Relasjoner mellom de forskjellige syllabi (pensumlister)
  - Beskrivelse av nivåinndeling for læremål (K-nivåer)
  - Vedlegg

0.3 Eksaminerbare læremål

- Læremålene støtter målsettingene med denne syllabus. De brukes for å generere eksamen. Prinsipielt er alle delene av denne syllabus eksaminerbare på K1 nivå. Dette betyr at kandidaten vil gjenkjenne, eller huske et begrep eller konsept. De spesielle læremålene på nivå K1, K2, og K3 er listet innledningsvis for hvert kapittel.
- Minimumstiden for akkrediterte kurs per kapittel er oppgitt i overskrften til hvert kapittel.

1. Smidig Programutvikling - 150 min.

Nøkkelord

Agile Manifest, smidig programvareutvikling, inkrementell utviklingsmodell, iterativ utviklingsmodell, programvarens livssyklus, testautomatisering, testbasis, testdrevet utvikling, testorakel, brukerhistorie (user story)

Læremål

1.1 Grunnleggende om smidig programvareutvikling

- FA-1.1.1 (K1) Husk de grunnleggende konsepter om smidig programvareutvikling basert på det Agile Manifest
- FA-1.1.2 (K2) Beskriv fordelene med involvering av hele teamet / selvorganiserende team
- FA-1.1.3 (K2) Beskriv fordelene med tidlig og hyppig tilbakemelding

1.2 Aspekter ved smidige tilnærminger

- FA-1.2.1 (K1) Husk smidige tilnærminger til programvareutvikling
- FA-1.2.2 (K3) Skriv testbare brukerhistorier i samarbeid med utviklere og representanter fra forretningen
- FA-1.2.3 (K2) Beskriv hvordan erfaringsmøter eller retrospektiver kan brukes som en mekanisme for prosessforbedring i smidige prosjekter
- FA-1.2.4 (K2) Beskriv bruken og hensikten med kontinuerlig integrasjon
- FA-1.2.5 (K1) Husk forskjellene mellom release- og iterasjonsplanlegging, og hvordan test tilfører verdi i hver av disse aktivitetene

1.1 Grunnleggende om smidig programvareutvikling

En tester i et smidig prosjekt vil arbeide annerledes enn i et tradisjonell prosjekt. Testere må forstå verdiene og prinsippene som underbygger smidige prosjekter De må også forstå hvordan de er en integrert del av et team sammen med utviklere og representanter fra forretningsområdet. Medlemmene i et smidig prosjekt kommuniserer med hverandre tidlig og ofte. Dette hjelper å fjerne feil tidlig og utvikle et kvalitetsprodukt.

1.1.1 Smidig programvareutvikling og det Agile Manifest

I 2001 ble en gruppe mennesker som representerer de mest brukte såkalt ”lette” utviklingsmetodikker for programvare enige om et felles sett av verdier og prinsipper som ble kjent som manifestet for smidig programvareutvikling eller det Agile Manifest [Agilemanifesto]. Det Agile Manifest inneholder fire uttalelser av verdier:

- Personer og samspill fremfor prosesser og verktøy
- Programvare som virker fremfor omfattende dokumentasjon
- Samarbeid med kunden fremfor kontraktsforhandlinger
- Å reagere på endringer fremfor å følge en plan

Det Agile Manifest hevder at selv om begrepene som står på høyre side har verdi, så har punktene til venstre enda høyere verdi.

Personer og samspill

Smidig utvikling er veldig personsentrert. Team av personer bygger programvare, og det er gjennom kontinuerlig kommunikasjon og samhandling, snarere enn avhengighet av verktøy og prosesser, at team kan arbeide mest mulig effektivt.

Programvare som virker

Fra et kundeperspektiv er fungerende programvare mye mer nyttig og verdifullt enn mye detaljert dokumentasjon. Det gir også muligheten til rask tilbakemelding til utviklingsteamet. I tillegg kan smidig utvikling gi betydelige fordeler mht. leveringstid, fordi fungerende programvare, riktignok med redusert funksjonalitet, er tilgjengelig mye tidligere i utviklingsprosessen. Smidig utvikling er derfor spesielt nyttig i raskt skiftende forretningsmiljøer hvor problemene og / eller løsninger er uklare eller hvor virksomheten ønsker å introdusere løsninger i nye problemområder.

Samarbeid med kunden

Kunder finner det ofte vanskelig å spesifisere systemet som de trenger. Samarbeid direkte med kunden øker sannsynligheten for å forstå nøyaktig hva kunden behøver. Å ha kontrakter med kunder kan være viktig, men regelmessig og tett samarbeid med dem er sannsynlig å bringe mer suksess til prosjektet.

Å reagere på endringer

Endring er uunngåelig i programvareprosjekter. Miljøet hvor virksomheten drives, lovgivning, konkurrentenes aktivitet, teknologiutviklingen, og andre faktorer kan ha store påvirkninger på prosjektet og dets mål. Disse faktorene må innpasses i utviklingsprosessen. Som sådan, å ha fleksibilitet i arbeidspraksis til omstilling er viktigere enn bare å strengt følge en plan.

Prinsippene
Kjernen i det Agile Manifest består av tolv prinsipper:

- Vår høyeste prioritet er å tilfredsstille kunden gjennom tidlige og kontinuerlige leveranser av programvare som har verdi.
- Ønsk endringer av krav velkommen, selv sent i utviklingen. Smidige prosesser bruker endringer til å skape konkurransefortrinn for kunden.
- Lever fungerende programvare hyppig, med et par ukers til et par måneders mellomrom. Jo oftere, desto bedre.
- Forretningssiden og utviklerne må arbeide sammen daglig gjennom hele prosjektet.
- Bygg prosjektet rundt motiverte personer. Gi dem miljøet og støtten de trenger, og stol på at de får jobben gjort.
- Den mest effektive måten å formidle informasjon inn til og innad i et utviklingsteam, er å snakke sammen ansikt til ansikt.
- Fungerende programvare er det primære målet på fremdrift.
- Smidige metoder fremmer bærekraftig programvareutvikling. Sponsorene, utviklerne og brukerne bør kunne opprettholde et jevnt tempo hele tiden.
- Kontinuerlig fokus på fremragende teknisk kvalitet og godt design fremmer smidighet.
- Enkelhet – kunsten å maksimere mengden arbeid som ikke blir gjort – er essensielt.
- De beste arkitekturer, krav og design vokser frem fra selvstyrte team.
- Med jevne mellomrom reflekterer teamet over hvordan det kan bli mer effektivt og så justere adferden sin deretter.

De ulike smidige metodene foreskriver hvordan disse verdiene og prinsippene skal føre til handling.

1.1.2 Samlet tilnærming i team

Tilnærming som bruker hele teamet betyr å involvere alle med kunnskap og ferdigheter som er nødvendige for å sikre prosjektets suksess. Teamet består av representanter fra kunden og andre interessenter som bestemmer produktegenskaper. Teamet bør være relativt lite; suksessfulle team har vært så få som tre medlemmer og så mange som ni. Ideelt sett deler hele teamet samme arbeidsplass, da samlokalisering sterkt forenkler kommunikasjon og samhandling. Hel-team tilnærming støttes gjennom daglige stående møter (se avsnitt 2.2.1) som involverer alle medlemmer av teamet, der arbeidet som pågår blir kommunisert og eventuelle hindringer for å gå videre blir tatt opp. Tilnærming som bruker hele teamet fremmer mer effektiv dynamikk i teamet.

Bruken av en slik hel-team tilnærming til produktutvikling er en av de viktigste fordelene med smidig utvikling. Fordelene inkluderer:

- Forbedret kommunikasjon og samarbeid innad i teamet
- Muliggjør at de ulike ferdighetene innenfor teamet får innflytelse til fordel for prosjektet
- Gjør kvalitet til alles ansvar

Hele teamet er ansvarlig for kvalitet i smidige prosjekter. Essensen av hel-team tilnærming ligger i at testere, utviklere, og representanter fra forretningsområdet jobber sammen i alle trinn i utviklingsprosessen. Testere vil jobbe tett med både utviklere og representanter fra forretningsområdet for å sikre at ønsket kvalitetsnivå er oppnådd. Dette inkluderer støtte og samarbeid med representanter fra forretningsområdet for å hjelpe dem med å opprette egnede akseptansetester, arbeide med utviklere for å bli enige om teststrategi, og bestemmer seg for tilnærminger for testautomatisering. Testere kan dermed overføre og utvide kunnskap om testing til andre teammedlemmer og påvirke utviklingen av produktet.

Hele teamet er involvert i alle konsultasjoner eller møter der produktegenskaper er presentert, analysert, eller estimeres. Konseptet med å involvere testere, utviklere og representanter fra forretningsområdet i alle temadiskusjoner er kjent som kraften av tre («power of three») [Crispin08].

1.1.3 Tidlig og hyppig tilbakemelding

Smidige prosjekter har korte iterasjoner slik at prosjektgruppen kan motta tidlig og kontinuerlig tilbakemelding på produktkvalitet gjennom hele utviklingsprosessen. En måte å gi rask tilbakemelding er ved kontinuerlig integrasjon (se avsnitt 1.2.4).

Når sekvensiell tilnærming på utvikling benyttes, kan kunden ofte ikke se produktet før prosjektet nesten er fullført. På dette tidspunktet er det ofte for sent for utviklingsteamet å effektivt løse eventuelle problemer kunden måtte ha. Ved å få hyppige tilbakemeldinger fra kunden etterhvert som prosjektet skrider frem kan smidige team innlemme de fleste nye endringer i produktutviklingsprosessen. Tidlig og hyppig tilbakemelding hjelper teamet å holde fokus på funksjoner med høyest forretningsverdi, eller tilknyttet risiko, slik at disse blir levert til kunden først. Det hjelper også å administrere teamet bedre siden kapasiteten til teamet er synlig for alle. For eksempel hvor mye arbeid kan vi gjøre i en sprint eller iterasjon? Hva kan hjelpe oss å produsere raskere? Hva hindrer oss fra å gjøre det?

Fordelene med tidlig og hyppig tilbakemelding inkluderer:

- Unngå misforståelser om krav, som ellers ikke hadde blitt oppdaget før senere i utviklingen når de er dyrere å rette.
- Avklare kundens ønsker om funksjoner, noe som gjør dem tilgjengelige for kundens bruk tidlig. På denne måten vil produktet bedre reflektere hva kunden ønsker.
- Oppdage (via kontinuerlig integrasjon), isolere og løse kvalitetsproblemer tidlig.
- Gi informasjon til teamet med hensyn til produktivitet og evne til å levere.
- Fremme konsekvent fremdrift i prosjektet.

1.2 Aspekter ved smidige tilnærminger

Det finnes en rekke av smidige tilnærminger som organisasjoner bruker. Felles praksis på tvers av de fleste smidige organisasjoner inkluderer samarbeid om å utarbeide brukerhistorier, retrospektiver, kontinuerlig integrasjon, og planlegging for hver iterasjon, samt for leveransene. Dette punktet beskriver noen av de smidige tilnærmingene.

1.2.1 Smidige tilnærminger til programvareutvikling

Det er flere smidige tilnærminger, som hver gjennomfører de verdier og prinsipper i det Agile Manifest på forskjellige måter. I dette pensumet, blir tre representanter av smidige tilnærminger vurdert: Extreme Programming (XP), Scrum og Kanban.

Extreme Programming
Extreme Programming (XP), opprinnelig introdusert av Kent Beck [Beck04], er en smidig tilnærming til programvareutvikling beskrevet av visse verdier, prinsipper og utviklingspraksis.

XP omfatter fem verdier for å lede utvikling: kommunikasjon, enkelhet, tilbakemelding, mot og respekt.

XP beskriver i tillegg et sett av prinsipper som retningslinjer: menneskelighet, økonomi, gjensidig nytte, likheter, forbedring, mangfold, refleksjon, flyt, mulighet, redundans, feil, kvalitet, små steg, og akseptert ansvar.
XP beskriver tretten hoved-teknikker: sitte sammen, hel-team tilnærming, informativ arbeidsområde, arbeid med energi, par-programmering, historier, ukentlig syklus, kvartalsvis syklus, slakk, ti minutters bygging, kontinuerlig integrasjon, test først programmering, og inkrementell utforming.
Mange av de smidige tilnærmingene til utvikling i bruk i dag er påvirket av XP og dets verdier og prinsipper. For eksempel, mange smidige team som følger Scrum bruker ofte teknikker fra XP.

Scrum
Scrum er et smidig rammeverk som inneholder følgende vedtekter og praksis [Schwaber01]:

- Sprint: Scrum deler et prosjekt i iterasjoner (kalt sprint) av fast lengde (vanligvis 2-4 uker).
- Produktinkrement: Hver sprint skal resultere i et produkt som kan leveres (kalt et inkrement).
- Produktkøen (product backlog): Produktets eier (product owner) forvalter en prioritert liste over ideer for produktet (kalt produktkøen). Produktkøen endrer seg fra sprint til sprint.
- Sprintkøen (sprint backlog): Ved starten av hver sprint velger Scrum-teamet et sett med de høyest prioriterte kravene fra produktkøen. Siden Scrum teamet og ikke produkteieren velger de elementene som skal realiseres i løpet av sprinten, er utvalget referert til som å være etter ”pull” prinsippet heller enn etter ”push” prinsippet.
- Definisjon av ”Ferdig”: Å sørge for at det er et produkt som er mulig å levere ved slutten av hver sprint, drøfter og definerer Scrum teamet relevante kriterier for sprintens ferdigstillelse. Diskusjonen utdyper teamets forståelse av elementene i backlog og produktkrav.
- Tidsbegrensning: Bare de oppgaver, krav, eller funksjoner som teamet forventer å fullføre innenfor sprinten er en del av sprintkøen. Hvis utviklingsteamet ikke kan fullføre en oppgave innenfor en sprint, blir de tilhørende produktegenskaper fjernet fra sprinten og oppgaven flyttes tilbake til produktkøen. Tidsbegrensning gjelder ikke bare for oppgaver, men også i andre situasjoner (for eksempel å håndheve møtenes start- og sluttider).
- Åpenhet: Utviklingsteamet rapporterer og oppdaterer sprintens status på daglig basis i et møte kalt daglig scrummøte. Dette gjør innholdet og fremdriften for den gjeldendende sprinten, inkludert testresultater, synlige for teamet, ledelse og alle interessentene. For eksempel kan utviklingsteamet vise sprintens status på en tavle.

Scrum definerer tre roller:

- Scrum Master: sikrer at praksis og regler til Scrum blir implementert og fulgt, og løser eventuelle avvik, ressursproblemer, eller andre hindringer som kan hindre teamet fra å følge rutiner og regler. Denne personen er ikke teamets leder, men en coach.
- Produkteier: representerer kunden og genererer, vedlikeholder og prioriterer produktkøen. Denne personen er ikke teamets leder.
- Utviklingsteam: skal utvikle og teste produktet. Teamet er selvorganisert: Det er ingen teamleder, så teamet gjør selv beslutninger. Teamet er også tverrfaglig (se avsnitt 2.3.2 og avsnitt 3.1.4).

Scrum (i motsetning til XP) dikterer ikke bestemte teknikker for programvareutvikling (f.eks test først programmering). I tillegg gir Scrum ingen veiledning om hvordan testing skal gjøres.

Kanban

Kanban [Anderson13] er en ledelsestilnærming som noen ganger brukes i smidige prosjekter. Det generelle målet er å visualisere og optimalisere flyten av arbeid innenfor verdikjeden. Kanban benytter tre instrumenter [Linz14]:

- Kanbantavle: Verdikjeden som skal forvaltes er visualisert på en Kanbantavle. Hver kolonne viser en stasjon, som er et sett med aktiviteter, for eksempel utvikling eller testing. Elementene som skal produseres eller oppgavene som skal behandles er symbolisert av lapper eller kort som flyttes fra venstre til høyre over hele linja gjennom stasjonene.
- Begrenset pågående arbeid: Mengden av parallelle aktive oppgaver er strengt begrenset. Dette styres av det maksimale antall kort som er tillatt for en stasjon og / eller globalt for kanbantavla. Når en stasjon har ledig kapasitet, trekker medarbeideren en (lapp) fra den foregående stasjonen.
- Ledetid (lead time): Kanban brukes for å optimalisere den kontinuerlige strøm av oppgaver ved å minimere (gjennomsnittlig) ledetid for verdistrømmen, fra arbeid starter med en aktivitet til aktiviteten er ferdig eller levert.

Kanban har noen likheter med Scrum. I begge rammeverk visualiseres de aktive oppgavene (f.eks på en offentlig whiteboard), noe som gir åpenhet av innhold og fremdrift av oppgaver. Oppgaver som ennå ikke er tidsfestet venter i en backlog og flyttes inn på Kanbantavla så snart det er ny plass (produksjonskapasitet) tilgjengelig.

Iterasjoner eller sprinter er valgfrie i Kanban. Kanban prosessen tillater å levere enhet for enhet, snarere enn som en del av en større release. Tidsbegrensning som et synkroniseringsmekanisme er derfor valgfritt, i motsetning til i Scrum, som synkroniserer alle oppgaver innenfor en sprint.

1.2.2 Felles utarbeidelse av brukerhistorier

Dårlige spesifikasjoner er ofte en viktig årsak til at prosjekter mislykkes. Problemer med spesifikasjon kommer av brukernes manglende innsikt i deres behov, fravær av en global visjon for systemet, redundante eller motstridende funksjoner og andre misforståelser. I smidig utvikling er brukerhistorier skrevet for å fange opp krav fra perspektiver til utviklere, testere og representanter fra forretningsområdet. I sekvensiell utvikling oppnås dette felles syn på en funksjon gjennom formelle inspeksjoner etter at kravene er skrevet; i smidig utvikling oppnås dette felles syn gjennom hyppige uformelle vurderinger mens kravene blir skrevet.

Brukerhistoriene må omfatte både funksjonelle og ikke-funksjonelle egenskaper. Hver historie inkluderer akseptansekriterier for disse egenskapene. Disse kriteriene bør defineres i samarbeid mellom representanter fra forretningsområdet, utviklere og testere. De gir utviklere og testere et utvidet syn av funksjonen som representanter fra forretningsområdet kommer til å validere. En smidig gruppe vurderer en oppgave som ferdig når et sett med akseptansekriterier er oppfylt.

Testerens unike perspektiv vil typisk forbedre brukerhistorien ved å identifisere manglende detaljer eller ikke-funksjonelle krav. En tester kan bidra ved å gi representanter fra forretningsområdet åpne spørsmål om brukerhistorien, foreslå måter å teste brukerhistorien på, og bekrefte akseptansekriterier.

For felles utarbeidelse av brukerhistorien kan man bruke teknikker som brainstorming og tankekart. Testeren kan bruke INVEST teknikk [INVEST]:

- Independent - Uavhengig
- Negotiable - Forhandlingsbar

- Valuable - Verdifull
- Estimable - Estimerbar
- Small - Liten
- Testable - Testbar

Ifølge 3C konseptet [Jeffries00], er en brukerhistorie satt sammen av tre elementer:

- **C**ard (Kort): Kortet er det fysiske mediet som beskriver en brukerhistorie. Det identifiserer kravet, hvor kritisk det er, forventet varighet på utvikling og test, og akseptansekriteriene for historien. Beskrivelsen må være nøyaktig, fordi det vil bli brukt i produktkøen.
- **C**onversation (Samtale): Samtalen forklarer hvordan programvaren skal brukes. Samtalen kan dokumenteres eller være muntlig. Testere, som har et annet synspunkt enn utviklere og representanter fra forretningsområdet [ISTQB_FL_SYL], tar med verdifulle innspill til utveksling av tanker, meninger og erfaringer. Samtalen begynner i løpet av planleggingen for releasen og fortsetter når historien er tidsfestet.
- **C**onfirmation (Bekreftelse): Akseptansekriteriene, diskutert i samtalen, blir brukt til å bekrefte at historien er ferdig. Disse akseptansekriterier kan omfatte flere brukerhistorier. Både positive og negative tester bør brukes til å dekke kriteriene. Under bekreftelsen spiller ulike aktører rollen som tester. Disse kan omfatte utviklere og spesialister fokusert på ytelse, sikkerhet, samarbeidsevne, og andre kvalitetsegenskaper. For å bekrefte en historie som ferdig, bør de definerte akseptansekriteriene bli testet og være oppfylt.

Smidige team dokumenterer brukerhistorier på forskjellig vis. Uavhengig av tilnærmingen til hvordan brukerhistorien dokumenteres, bør dokumentasjonen være kortfattet, tilstrekkelig og omfatte bare det som er nødvendig.

1.2.3 Retrospektiver

Et retrospektiv er et møte avholdt i slutten av hver iterasjon for å diskutere hva som var vellykket, hva som kan forbedres, og hvordan inkludere forbedringene og opprettholde suksess i fremtidige iterasjoner. Retrospektiver dekker temaer som prosess, mennesker, organisasjoner, relasjoner og verktøy. Jevnlig gjennomførte retrospektiver, når aktivitetene følges opp, er kritisk til selvorganisering og kontinuerlig forbedring av utvikling og testing.

Retrospektiver kan resultere i at testrelaterte forbedringstiltak blir besluttet. Disse er fokusert på testeffektivitet, testproduktivitet, kvalitet på testtilfeller, og teamets tilfredshet. De kan også ta opp testbarhet av systemet, brukerhistoriene, funksjoner eller systemgrensesnitt. Årsaksanalyse av feil kan drive forbedringer av testing og utvikling. Generelt bør teamene gjennomføre bare noen få forbedringer per iterasjon. Dette tillater kontinuerlig forbedring ved et jevnt tempo.
Tidsplanlegging og organisering av retrospektivene avhenger av den bestemte smidige metoden som følges. Representanter fra forretningsområdet og teamet deltar i hvert retrospektiv som deltakere, mens en møteleder tilrettelegger, organiserer og driver møtet. I noen tilfeller kan teamene invitere andre deltakere til møtet.

Testere bør spille en viktig rolle i retrospektiver. Testere er en del av teamet og vil ta med deres unike perspektiv [ISTQB_FL_SYL], punkt 1.5. Testing skjer i hver iterasjon og bidrar vitalt til suksess. Alle teammedlemmer, testere og ikke-testere, kan gi innspill på både testing og andre aktiviteter.

Retrospektiver må skje i et profesjonelt miljø preget av gjensidig tillit. Egenskapene til en vellykket retrospektiv er de samme som for enhver annen review som er omtalt i Foundation Level pensumet [ISTQB_FL_SYL], punkt 3.2.

1.2.4 Kontinuerlig Integrasjon

For å kunne levere et inkrement av et produkt kreves det pålitelig, fungerende og integrert programvare ved slutten av hver iterasjon. Kontinuerlig integrasjon adresserer denne utfordringen ved å sette sammen (eng. merge) alle kodeendringer og ved å integrere komponenter periodisk, eller minst en gang per dag. Konfigurasjonsstyring, kompilering, bygging, distribusjon og testing er pakket inn i en enkelt automatisert og repeterbar prosess. Siden utviklere integrerer, bygger og tester kontinuerlig vil feil avdekkes raskere.

Etter utviklerens koding, testing, evt. debugging og innsjekking av koden til felles kodebase, vil en prosess for kontinuerlig integrasjon bestå av følgende automatiserte aktiviteter:

- Statisk kodeanalyse: utfører statisk kodeanalyse og rapporterer resultatet
- Kompilering: kompilerer og lenker koden, genererer utførbare filer
- Enhetstest: utfører enhetstestene, kontrollerer kodedekning og rapporterer testresultatet
- Installasjon: installerer programvaren i et testmiljø
- Integrasjonstest: utfører integrasjonstester og rapporterer testresultatet
- Rapport (dashboard): publiserer status på alle disse aktivitetene til et felles synlig sted eller ved å sende e-post med status til teamet.

En automatisert bygge- og testprosess utføres på daglig basis og oppdager integrasjonsfeil tidlig. Kontinuerlig integrasjon tillater smidige testere å kjøre de automatiserte testene jevnlig; i noen tilfeller som en del av den kontinuerlige integrasjonsprosessen, og å gi rask tilbakemelding om kvaliteten av koden til teamet. Testresultatene er synlige for alle teammedlemmene, spesielt når automatiserte rapporter er integrert i prosessen. Automatisert regresjonstesting kan utføres kontinuerlig i iterasjonen. Gode automatiserte regresjonstester dekker så mye funksjonalitet som mulig, inkludert brukerhistorier («user stories») levert i tidligere iterasjoner. God dekning med de automatiserte regresjonstestene hjelper til med å støtte bygging (og testing) av store integrerte systemer. Når regresjonstesting er automatisert kan smidige testere konsentrere den manuelle testingen på nye funksjoner, endringer og på å verifisere rettede feil.

I tillegg til automatiserte tester kan organisasjoner som benytter kontinuerlig integrasjon også bygge verktøy for kontinuerlig kvalitetskontroll. I tillegg til å kjøre enhets- og integrasjonstester kan slike verktøy kjøre statiske og dynamiske tester, måle og profilere ytelse, ekstrahere og formatere dokumentasjon ut fra kildekoden og støtte manuell kvalitetssikring. Denne kontinuerlige bruken av kvalitetskontroll har som mål å forbedre produktkvaliteten samt å redusere tiden det tar å levere programvaren ved å erstatte den tradisjonelle praksis ved å utføre kvalitetskontroll etter at all utvikling er utført.

Byggeverktøy kan lenkes til automatiske installasjonsverktøy som kan hente det korrekte bygget fra kontinuerlig integrasjons- eller byggeserver og installere programvaren i ett eller flere utviklings-, test-, ”staging-” eller til og med i produksjonsmiljø. Dette reduserer feil og forsinkelser forbundet med spesialiserte personer eller utviklere som ellers brukes til å installere programvaren i disse miljøene.

Kontinuerlig integrasjon kan gi følgende nytte:

- Muliggjør tidligere oppdagelse og lettere analyse av rotårsak av problemer med

integrering og endringer med konflikter mellom seg
- Gir utviklingsteamet jevnlig tilbakemelding om koden fungerer
- Holder programvareversjonen som testes maksimalt en dag fra versjonen som blir utviklet
- Reduserer regresjonsrisiko ved opprydding og omskriving av kode på grunn av rask re-testing og regresjonstesting av kodebasen etter små endringer
- Gir trygghet om at hver dags utviklingsarbeid er basert på et solid fundament
- Viser fremdriften mot den fullstendige produktleveransen og oppmuntrer dermed utviklere og testere.
- Eliminerer risikoen med tidsplanen ved en «big-bang» integrasjon.
- Tilbyr alltid tilgang på utførbar programvare gjennom hele iterasjonen for testing, demonstrasjon eller opplæringsformål
- Reduserer manuelle aktiviteter som gjentas
- Tilbyr rask tilbakemelding på valg som er gjort for å forbedre kvalitet og tester

Men, kontinuerlig integrasjon er ikke uten risikoer og utfordringer:

- Verktøy for kontinuerlig integrasjon må bli introdusert og vedlikeholdt
- Prosessen for kontinuerlig integrasjon må defineres og etableres
- Testautomatisering krever ekstra ressurser og kan være vanskelig å etablere
- Grundig testdekning er avgjørende for å nå gevinster med automatisert testing
- Et team kan stole for mye på enhetstester og utføre for lite system- og akseptansetesting

Kontinuerlig integrasjon krever bruk av verktøy, inkludert verktøy for testing, verktøy for å automatisere byggeprosessen og verktøy for versjonskontroll.

1.2.5 Leveranse- og iterasjonsplanlegging

Som nevnt i Foundation-pensumet [ISTQB_FL_SYL], er planlegging en kontinuerlig aktivitet, og dette er tilfelle i smidige livssykluser også. For smidige livssykluser benyttes to typer planlegging, leveranse- og iterasjons planlegging.

Leveranseplanlegging ser framover mot leveringen av et produkt, ofte noen måneder før start på prosjektet. Leveranseplanlegging definerer og redefinerer produktkøen, og kan involvere oppdeling og detaljering av større brukerhistorier til en samling mindre historier, Leveranseplanlegging er basisen for en testtilnærming og testplan som dekker alle iterasjoner. Leveranseplanen er på et høyt nivå.
I leveranseplanleggingen etablerer og prioriterer representanter for forretningsområdet brukerhistoriene for leveringen. Dette skjer i samarbeid med teamet (se avsnitt 1.2.2). Basert på disse brukerhistoriene identifiseres prosjekt- og (produkt-)kvalitetsrisiko og det estimeres på et overordnet nivå (se avsnitt 3.2).

Testere er involvert i leveranseplanleggingen og er spesielt verdifulle i følgende aktiviteter:

- Definerer testbare brukerhistorier, inkludert akseptansekriterier
- Deltar i analyse av prosjekt- og kvalitetsrisiko
- Estimerer testarbeidet med brukerhistoriene
- Definerer nødvendige testnivå
- Planlegger testingen for leveransen

Etter leveranseplanleggingen starter planlegging for den første iterasjonen.
Iterasjonsplanlegging ser mot slutten av en enkelt iterasjon og tar hensyn til iterasjonens kø (”backlog”).

I iterasjonsplanlegging velger teamet fra de prioriterte brukerhistoriene i køen, utdyper brukerhistoriene, utfører en risikoanalyse for brukerhistoriene og estimerer arbeidet for hver brukerhistorie. Hvis en brukerhistorie er for vag og forsøk på å utdype det feiler så kan teamet forkaste den og ta neste brukerhistorie basert på prioritet.

Representanter fra forretningsområdet må svare på teamets spørsmål om hver brukerhistorie slik at teamet kan forstå hva de skal implementere og teste for hver historie.
Antall historier som velges baseres på det etablerte teamets kapasitet og estimert omfang av de valgte brukerhistoriene. Etter at innholdet i iterasjonen er klart blir brukerhistoriene brutt opp i oppgaver som utføres av passende teammedlemmer.

Testere er involvert i iterasjonsplanleggingen og gir spesielt verdi til de følgende aktivitetene:

- Deltar i den detaljerte risikoanalysen av brukerhistorier
- Bestemmer testbarheten av brukerhistoriene
- Utarbeider akseptansetest for brukerhistoriene
- Bryter opp brukerhistorier i oppgaver (spesielt testoppgaver)
- Estimerer arbeidsomfang for alle testoppgaver
- Identifiserer funksjonelle og ikke-funksjonelle aspekter ved systemet som skal testes
- Støtter og deltar i testautomatisering på flere nivå av testing

Leveranseplaner kan endres i prosjektets livsløp, inkludert endringer på individuelle brukerhistorier i produktkøen. Disse endringene kan bli utløst av interne og eksterne faktorer. Interne faktorer inkluderer evne til leveranser, hastighet og tekniske problemstillinger. Eksterne faktorer inkluderer oppdagelsen av nye markeder og muligheter, nye konkurrenter eller forretningstrusler som kan endre formål og/eller datoer for releasen. I tillegg kan iterasjonsplaner endres i løpet av en iterasjon. For eksempel kan en bestemt brukerhistorie ansett som enkel ved estimering være mer kompleks enn forventet.
Disse endringene kan være utfordrende for testere. Testere må forstå det store bildet for leveransen for å planlegge testingen, og de må ha tilstrekkelig testbasis og test orakel i hver iterasjon for å utvikle tester som diskutert i pensumet for Foundation Level [ISTQB_FL_SYL], kap 1.4. Informasjonen som trengs må være tilgjengelig for testeren tidlig, og likevel må endringer tas tak i i henhold til smidige prinsipper. Dette dilemmaet krever nøye beslutninger om teststrategier og testdokumentasjon. For mer om smidige testutfordringer, se [Black09], kap. 12.
Leveranse- og iterasjonsplanlegging skal både omfatte testplanlegging og planlegging av utviklingsaktiviteter. Spesielt testrelaterte emner er:

- Omfanget av testingen generelt, omfanget av testingen for de områdene som er valgt ut, testmålene og årsakene til avgjørelsene
- Teammedlemmene som skal utføre testaktivitetene
- Behov for testmiljø og testdata, når de trengs og om det vil bli tillegg eller endringer til testmiljø og/eller testdata før eller i løpet av prosjektet
- Tidsvalg, rekkefølge, avhengigheter og forutsetninger for funksjonelle og ikke-funksjonelle testaktiviteter (f.eks. hvor ofte kjøre regresjonstester, hvilke funksjoner som er avhengig av andre funksjoner eller testdata), også hvordan testaktivitetene relateres til og er avhengig av utviklingsaktiviteter
- Prosjekt- og kvalitetsrisiko som skal adresseres (se avsnitt 3.2.1)
- I tillegg bør teamets totale estimering inkludere betraktning av nødvendig tid og innsats for å ferdigstille testaktivitetene.

2. Grunnleggende prinsipper, praksis og prosesser innen smidig testing – 105 min.

Nøkkelord

Build verification test, konfigurasjonselement, konfigurasjonsstyring

Anmerkning: ”build verification test” er på norsk kalt for ”verifikasjonstest av bygget”.

Læremål

2.1 Forskjellen mellom tradisjonell testing og smidige metoder

FA-2.1.1 (K2) Beskriv forskjellene mellom testaktiviteter i smidige prosjekter og ikke-smidige prosjekter

FA-2.1.2 (K2) Beskriv hvordan utviklings- og testaktiviteter er integrert i smidige prosjekter

FA-2.1.3 (K2) Beskriv rollen for uavhengig testing i smidige prosjekter

2.2 Status for test i smidige prosjekter

FA-2.2.1 (K2) Beskriv verktøyene og teknikkene som brukes for å kommunisere teststatus i et smidig prosjekt, inkludert testframdrift og produktkvalitet

FA-2.2.2 (K2) Beskriv utviklingsprosessen for tester over flere iterasjoner og forklar hvorfor testautomatisering er viktig for å håndtere regresjonsrisiko i smidige prosjekter

2.3 Rollen og ferdigheter til en tester i et smidig team

FA-2.3.1 (K2) Forstå ferdighetene (kommunikasjon med andre personer, problemområde, og test) til en tester i et smidig team

FA-2.3.2 (K2) Forstå rollen til en tester i et smidig team

2.1 Forskjellene i testing mellom tradisjonelle og smidige metoder

Som beskrevet i Foundation Level syllabus [ISTQB_FL_SYL] og i [Black09], henger testaktiviteter sammen med utviklingsaktiviteter, og testing vil derfor variere ved bruk av ulike livssyklusmodeller. For å kunne arbeide effektivt, må testere forstå forskjellene mellom testing i tradisjonelle livssyklusmodeller (for eksempel sekvensielle som V-modellen eller iterative som RUP) og smidige livssykluser. De smidige modellene skiller seg i måten test- og utviklingsaktiviteter er integrert på, samt prosjektets arbeidsprodukt, navn, inngangs- og utgangskriterier for de ulike testnivåene, bruken av verktøy, og hvordan uavhengig testing kan benyttes effektivt.

Testere bør huske at ulike organisasjoner implementerer livssykluser ulikt. Fravik fra idealene til smidige livssykluser (se avsnitt 1.1) kan representere gode tilpasninger til praksis. Evnen til å tilpasse seg til prosjekters kontekst, inkludert utviklingspraksis som følges, er en nøkkelfaktor for suksess for testere.

2.1.1 Test- og utviklingsaktiviteter

En av hovedforskjellene mellom tradisjonelle livssykluser og smidige livssykluser er ideen om veldig korte iterasjoner, hvor hver iterasjon resulterer i fungerende software som gir verdi for interessentene. Ved begynnelsen av prosjektet er det en leveranseplanleggingsperiode. Denne følges av en sekvens av iterasjoner. Ved begynnelsen av hver iterasjon er det en periode med iterasjonsplanlegging. Når målene for iterasjonen er etablert, utvikles de valgte brukerhistoriene, utviklet software integreres med eksisterende software og testes. Disse iterasjonene er meget dynamiske, hvor utviklings-, integrasjons- og testaktiviteter kan skje parallelt og med overlapp. Testaktiviteter skjer gjennom hele iterasjonen - ikke som en sluttaktivitet.

Testere, utviklere og kunder har alle en rolle i testing, som ved tradisjonelle livssykluser. Utviklere utvikler og utfører enhetstester etterhvert som de utvikler funksjonalitet fra brukerhistoriene. Testere vil deretter teste disse. Kunder tester også brukerhistoriene under implementering. Kundene kan bruke nedskrevne testtilfeller, men de kan også eksperimentere med og bruke den utviklede funksjonaliteten for å gi raske tilbakemeldinger til utviklingsteamet.

I noen tilfeller vil det være behov for periodiske forbedrings- og stabiliseringsiterasjoner for å håndtere utsatte feil og annen form for teknisk gjeld. Men det er god praksis at ingen funksjoner blir betraktet som ferdige inntil de er integrert og testet med øvrig kode [Goucher09]. Enda en god praksis er å adressere feil som ikke er løst fra forrige iterasjon i starten på en ny iterasjon som en del av køen til denne iterasjonen (referert til som "fiks feil først"). Imidlertid så vil enkelte hevde at denne praksisen resulterer i en situasjon hvor den totale arbeidsmengden for aktuell iterasjon blir ukjent, og det vil bli vanskeligere å estimere når gjenværende funksjonalitet er ferdig. På slutten av iterasjonssekvensen kan det være flere aktiviteter for å klargjøre software for leveranse. I noen tilfeller vil leveranse skje på slutten av hver iterasjon.

Når risikobasert testing benyttes som en av teststrategiene, vil en høy-nivå risikoanalyse utføres under leveranseplanleggingen, ofte med testere som drivere under analysen. Imidlertid så vil de spesifikke kvalitetsrisikoene, assosiert med hver iterasjon, identifiseres og behandles i iterasjonsplanleggingen. Denne høy-nivå risikoanalysen kan påvirke sekvensen av utvikling, samt prioritet og dybde i testingen. Den påvirker også estimeringen av nødvendig testing for hver del av funksjonaliteten (se avsnitt 3.2).

I noen smidige metoder (for eksempel Extreme Programming), er parvis arbeid brukt. Parvis arbeid kan involvere testere som jobber sammen to og to for å teste noe. Parvis arbeid kan også

involvere en tester som jobber sammen med en utvikler for å utvikle og teste noe. Parvis arbeid kan være vanskelig når test-teamet er spredt over flere plasser, men prosesser og verktøy kan hjelpe til med gjennomføring av distribuert parvis arbeid. For mer informasjon om distribuert arbeid, se [ISTQB_ALTM_SYL], avsnitt 2.8.

Testere kan også opptre som test- og kvalitetstrenere (eng: coach) innen teamet ved å dele testkunnskaper og støtte kvalitetsbevarende arbeid innen teamet. Dette bidrar til en følelse av kollektivt eierskap for produktkvaliteten.

Testautomatisering på alle testnivåer skjer i mange smidige team, og dette kan bety at testere bruker tid på å lage, utføre, overvåke, og vedlikeholde de automatiserte testene og resultatene. På grunn av mye testautomatisering tenderer en høyere andel av den manuelle testingen i smidige prosjekter mot å bruke erfaringsbaserte og feilbaserte teknikker som angrep, utforskende testing, og feilgjetting (se [ISTQB_ALTA_SYL], avsnitt 3.3 og 3.4 og [ISTQB_FL_SYL], avsnitt 4.5). Mens utviklere fokuserer på å lage enhetstester, bør testere fokusere på å lage automatiserte integrasjons-, system- og systemintegrasjonstester. Dette leder til en tendens i smidige team mot å foretrekke testere med en god teknisk og testautomatiseringsbakgrunn.

Et sentralt smidig prinsipp er at forandring kan skje gjennom hele prosjektet. Derfor er lettvekts dokumentasjon om arbeidsproduktet favorisert i smidige prosjekter. Forandringer i eksisterende funksjonalitet har følger for testingen, spesielt for regresjonstesting. Bruken av automatisert testing er en mulighet for å håndtere mengden testinnsats ved endringer. Det er imidlertid viktig at endringsraten ikke overstiger prosjektteamets evne til å håndtere risikoene forbundet med endringene.

2.1.2 Arbeidsprodukter i prosjektet

Arbeidsproduktene i prosjektet som er av mest interesse for testere i smidige team faller typisk inn i tre kategorier:

1. Forretningsorienterte arbeidsprodukter som beskriver hva som er nødvendig (for eksempel kravspesifikasjoner) og hvordan en kan bruke produktet (for eksempel brukerdokumentasjon).
2. Arbeidsprodukter fra utvikling som beskriver hvordan systemet er bygget (for eksempel relasjonsdiagrammer for databaser), som faktisk implementerer systemet (for eksempel kode), eller som evaluerer individuelle deler av koden (for eksempel automatiserte enhetstester).
3. Arbeidsprodukter fra testing som beskriver hvordan systemet er testet (for eksempel teststrategier og -planer), som faktisk tester systemet (for eksempel manuelle og automatiserte tester), eller som presenterer testresultater (for eksempel arbeidstavler som diskutert i avsnitt 2.2.1).

I et typisk smidig prosjekt er det vanlig praksis å unngå å produsere mye dokumentasjon. Fokus er i stedet på software som virker, sammen med automatiserte tester som demonstrerer at kravene er oppfylte. Denne motivasjonen for reduksjon av dokumentasjon gjelder bare dokumentasjon som ikke leverer verdi til kunden. I et suksessfullt smidig prosjekt vil det være en balanse mellom å øke effektiviteten gjennom å redusere dokumentasjon og å gi nok dokumentasjon for å støtte kunde, testing, utvikling, og vedlikeholdsaktviteter. Teamet må gjøre en beslutning ved leveranseplanleggingen om hvilke arbeidsprodukter som er nødvendige, og på hvilket nivå dokumentasjonen av disse skal være.

Typiske kundeorienterte arbeidsprodukter fra smidige prosjekter inkluderer brukerhistorier og akseptansekriterier. Brukerhistorier er den smidige formen for kravspesifikasjoner, og må forklare hvordan systemet skal oppføre seg i forhold til en spesifikk funksjonalitet. En brukerhistorie bør

definere funksjonalitet som er liten nok til å kunne bli ferdigstilt i én enkel iterasjon. Større samlinger av relatert funksjonalitet, eller en samling av delfunksjonalitet som realiserer en mer kompleks funksjonalitet, kan refereres til som ”epics”. Slike ”epics” kan inkludere brukerhistorier for ulike utviklingsteam. For eksempel kan en brukerhistorie beskrive hva som er påkrevd på API-nivå (mellomvare), mens en annen brukerhistorie beskriver hva som er påkrevd på brukergrensesnitt-nivå (applikasjon). Disse samlingene kan bli utviklet over flere iterasjoner. Hver ”epic” og dens brukerhistorier må ha tilhørende akseptansekriterier.

Arbeidsprodukter fra utviklere på smidige prosjekter inneholder typisk kode. Utviklere i smidige team lager også ofte automatiserte enhetstester. Disse testene kan utvikles etter utviklingen av kode. I noen tilfeller lager utviklerne tester etter hvert, før hver del av koden er skrevet. Dette for å kunne verifisere at koden fungerer når den er skrevet. Mens denne fremgangsmåten er beskrevet som testdrevet utvikling, så er disse testene mer en form for utførbare lav-nivå designspesifikasjoner enn tester [Beck02].

Typiske testprodukter i smidige prosjekter omfatter både automatiserte tester og dokumentasjon slik som testplaner, lister over kvalitetsrisikoer, manuelle tester, feilrapporter, og logger med testresultater. Dokumentene er nedskrevet på en så "lettvekts" måte som mulig, noe som faktisk også er ofte rett i tradisjonelle livssykluser. Testere vil også produsere testmål fra feilrapporter og testresultatlogger, hvor det igjen er fokus på en lettvekts fremgangsmåte.

I noen smidige implementasjoner, spesielt de som er regulert gjennom forskrifter, livs- eller sikkerhetskritiske, distribuerte eller veldig komplekse prosjekter og produkter, er det nødvendig med mer formalisering rundt arbeidsproduktene. For eksempel overfører noen team brukerhistorier og akseptansekriterier til mer formelle kravspesifikasjoner. Vertikale og horisontale sporbarhetsrapporter kan utarbeides for å tilfredsstille revisjon, forskrifter, og andre krav.

2.1.3 Testnivå

Testnivåer er testaktiviteter som er logisk knyttet sammen, typisk gjennom modenhet eller kompletthet av elementene under test.

I sekvensielle livssyklusmodeller blir ofte testnivåene definerte slik at sluttkriteriene for et nivå er del av startkriteriene for neste nivå. I noen iterative modeller gjelder ikke denne regelen. Testnivå overlapper. Kravspesifikasjon, designspesifikasjon, og utviklingsaktiviteter kan overlappe med testnivåene.

Overlapp vil skje i noen smidige livssykluser på grunn av at forandringer i krav, design, og kode kan skje på enhver tid i iterasjonen. Mens Scrum i teorien ikke tillater endringer i en brukerhistorie etter iterasjonsplanlegging, vil dette i praksis forekomme noen ganger. Under en iterasjon vil enhver brukerhistorie typisk gå sekvensielt gjennom følgende testaktiviteter:

- Enhetstesting, typisk utført av utvikleren
- Akseptansetesting av funksjoner, som noen ganger er delt i to aktiviteter:
  - Verifikasjonstesting av funksjoner, som ofte er automatisert, kan utføres av utviklere eller testere, og involverer testing mot brukerhistoriens akseptansekriterium
  - Valideringstesting av funksjoner, som vanligvis er manuell og kan inkludere utviklere, testere og kunderepresentanter som samarbeider om å bestemme hvorvidt det som er utviklet er klar for bruk, å synliggjøre progresjonen som er gjort, og å motta reelle tilbakemeldinger fra kunden.

I tillegg er det ofte en parallell prosess med regresjonstesting underveis i iterasjonen. Dette omfatter å gjenta de automatiserte enhetstestene og verifikasjonstestene fra nåværende og forrige iterasjon; vanligvis via et rammeverk for kontinuerlig integrering.

I noen smidige prosjekter kan det være et systemtestnivå som starter når den første brukerhistorien er klar for testing. Dette kan involvere å utføre funksjonell tester så vel som ikke-funksjonelle tester for ytelse, pålitelighet, brukbarhet og andre relevante testtyper.

Smidige team kan gjøre ulike former for akseptansetesting (begrepet er forklart i Foundation Level syllabus [ISTQB_FL_SYL]). Interne alfatester og eksterne betatester kan utføres enten ved avslutning av hver iterasjon, etter at iterasjonen er ferdig, eller etter en serie med iterasjoner. Akseptansetester med fokus på brukere, operasjon/drift, myndigheter, og kontrakt kan også utføres ved avslutning av hver iterasjon, etter avslutning av hver iterasjon, eller etter en serie av iterasjoner.

2.1.4 Testing og konfigurasjonsstyring

Smidige prosjekter involverer ofte mye bruk av automatiseringsverktøy for å utvikle, teste og styre software utvikling. Utviklere bruker verktøy for statisk analyse, enhetstester og måling av kodedekning. Utviklere sjekker kontinuerlig inn kode og enhetstester i et konfigurasjonsstyrings-system ved bruk av rammeverk for automatisert bygging og testing. Disse rammeverkene tillater kontinuerlig integrasjon av ny software i systemet, med repeterende kjøring av statiske analyser og enhetstester når ny software sjekkes inn [Kubaczkowski].

Disse automatiserte testene kan også inkludere funksjonelle tester på integrasjons- og systemnivå. Slike automatiserte funksjonelle tester kan lages ved å bruke funksjonelle testrammeverk, open-source verktøy for funksjonelt brukergrensesnitt, eller kommersielle verktøy. De kan bli integrert med de automatiserte testene som kjøres som del av et kontinuerlig integrasjonsrammeverk. I noen tilfeller er de funksjonelle testene adskilt fra enhetstestene og blir kjørt sjeldnere på grunn av tiden det tar å kjøre dem. For eksempel kan enhetstestene kjøres hver gang ny software er sjekket inn, mens de lengre funksjonelle testene bare kjøres daglig eller hver n'te dag.

Et mål med de automatiserte testene er å bekrefte at bygget fungerer og er installerbart. Hvis en av de automatiserte testene feiler bør teamet rette den underliggende feilen før neste innsjekking av kode. Dette krever en investering i sanntidsrapportering for god synlighet av testresultatene. Denne fremgangsmåten hjelper til med å redusere dyre og ineffektive "bygg - installer - feil - re-bygg - re-installer" sykluser som kan oppstå i mange tradisjonelle prosjekter, siden endringer som brekker bygget eller som forårsaker at software ikke kan installeres blir oppdaget fort.

Verktøy for automatisert testing og bygging hjelper med å håndtere regresjonsrisikoen med hyppige endringer som ofte skjer i smidige prosjekter. Å stole for mye på at regresjonsrisikoen håndteres av bare automatiserte enhetstester kan imidlertid være et problem, siden enhetstester vil ofte ha begrenset effektivitet når det gjelder å finne feil [Jones11]. Automatiserte tester på integrasjons- og systemnivå er også påkrevd.

2.1.5 Organisatoriske muligheter for uavhengig testing

Som diskutert i Foundation Level syllabus [ISTQB_FL_SYL] er uavhengige testere ofte bedre egnet til å finne feil. I noen smidige team lager utviklerne mange av testene som automatiserte tester. En eller flere testere kan være integrert i teamet og utføre mange av testoppgavene.

Imidlertid vil deres posisjon i teamet gi en risiko for tap av uavhengighet og objektivitet i evalueringene.

Andre smidige team opprettholder uavhengige, separate test-team, og tilordner testere etter behov under de siste dagene av hver iterasjon. Dette kan bevare uavhengighet, og disse testerne kan gi en objektiv og upåvirket evaluering av software som utvikles. Problemer med denne fremgangsmåten kan være tidspress, mangel på forståelse for den nye funksjonaliteten i produktet og samarbeidsproblemer med interessenter og utviklere.

En tredje mulighet er å ha et uavhengig og separat test-team hvor testere er tilordnet til smidige team over lengre tid fra begynnelsen av prosjektet. Dette gir de muligheten til å holde på uavhengigheten, samtidig som de får en god forståelse av produktet og et godt samarbeid med andre medlemmer i teamet. I tillegg så kan det uavhengige test-teamet ha spesialiserte testere utenfor det smidige teamet som jobber på langtids og/eller iterasjon-uavhengige aktiviteter som for eksempel å utvikle automatiserte testverktøy, utføre ikke-funksjonell testing, sette opp og vedlikeholde testmiljø og -data, og utføre testnivå som ikke passer inn i en iterasjon (eksempelvis systemintegrasjonstesting).

2.2 Teststatus i smidige prosjekter

Endringer skjer hyppig i smidige prosjekter. Disse endringene betyr at teststatus, testprogresjon og produktkvalitet utvikler seg konstant. Testere må finne måter å spre slik informasjon til resten av teamet for at teamet skal kunne ta beslutninger for en vellykket avslutning på hver iterasjon. I tillegg kan endringer påvirke eksisterende funksjonalitet fra tidligere iterasjoner. Derfor må manuelle og automatiserte tester oppdateres for effektivt å kunne håndtere regresjonsrisiko.

2.2.1 Kommunikasjon av teststatus, testprogresjon og produktkvalitet

Smidige team strever mot å ha software som virker på slutten av hver iterasjon. For å bestemme når teamet har fungerende software må de overvåke progresjonen for alle arbeidsproduktene i iterasjonen og leveransen. Testere i smidige team benytter ulike metoder for å finne testprogresjon og status, inkludert testautomatiseringsresultater, progresjon for testoppgaver og testhistorier på den smidige oppgavetavla, og burndown-grafer som viser teamets progresjon. Disse resultatene kan deretter kommuniseres til resten av teamet ved å bruke medium som kontrollpanel på wiki og kontrollpanel-lignende e-poster, så vel som muntlig under de daglige stående møtene. Smidige team kan bruke verktøy som automatisk genererer statusrapporter basert på testresultater og -progresjon, som igjen kan oppdatere wiki-lignende kontrollpanel og e-poster. Denne kommunikasjonsmetoden samler også målinger fra testprosessen, som kan benyttes for å forbedre selve prosessen. Å kommunisere teststatus på en slik automatisert måte vil også gi testere mer ledig tid til å lage og utføre flere testtilfeller.

Team kan bruke burndown-grafer for å overvåke progresjonen gjennom hele leveransen og innen hver iterasjon. En burndown-graf [Crispin08] representerer mengden av gjenværende arbeid mot tiden som er budsjettert til leveransen eller iterasjonen.

For å tilby en umiddelbar og detaljert visuell representasjon av hele teamets status, inkludert teststatus, kan teamet bruke smidige oppgavetavler. Brukerhistoriene, utviklingsoppgavene, testoppgavene, og andre oppgaver som er definert under iterasjonsplanleggingen (se avsnitt 1.2.5) er samlet på oppgavetavlen, ofte med ulike farger for å bestemme oppgavetypen. Under iterasjonen blir progresjonen styrt ved å flytte oppgavene på oppgavetavlen i kolonner som for eksempel å gjøre, under arbeid, verifikasjon, og ferdig. Smidige team kan bruke verktøy for å

vedlikeholde de ulike oppgavene og oppgavetavlen, noe som kan automatisere kontrollpaneler og statusoppdateringer.

Testoppgaver på oppgavetavla er knyttet til akseptansekriteriene som er definert for brukerhistoriene. Ettersom testautomatiseringsskript, manuelle tester, og utforskende tester for en testoppgave oppnår en godkjent status, blir oppgaven flyttet i ferdig-kolonnen på oppgavetavla. Hele teamet vurderer jevnlig oppgavetavlas status, ofte under de daglige stående møtene, for å forsikre at oppgavene på tavla har en akseptabel progresjon. Hvis noen oppgaver (inkludert testoppgaver) ikke flyttes eller flyttes for sakte, vil teamet vurdere å gjøre noe med de delene som kan blokkere progresjonen for disse oppgavene.

De daglige stående møtene omfatter alle medlemmene av det smidige teamet, inkludert testerne. På dette møtet vil alle kommunisere deres nåværende status. Agendaen for hvert medlem er [Agile Alliance Guide]:

- Hva har du utført siden det forrige møtet?
- Hva planlegger du å utføre til det neste møtet?
- Hva hindrer deg?
- Hver sak som kan blokkere testprogresjon blir kommunisert under de daglige stående møtene, slik at hele teamet er klar over sakene og kan løse disse.

For å forbedre den overordnede produktkvaliteten vil mange smidige team gjennomføre kundetilfreds-undersøkelser for å motta tilbakemeldinger på om produktet møter kundens forventninger. Teamet kan også bruke andre måleparametere for å forbedre produktkvaliteten, lik de parameterne som måles i tradisjonelle utviklingsmetodikker, som for eksempel forholdet mellom godkjent og feilet på testene, hyppighet av feiloppdagelse, bekreftelsestest- og regresjonstest-resultater, feiltetthet, antallet feil funnet og rettet, kravdekning, risikodekning, kodedekning, og kodeendring. Som med enhver livssyklus må parameterne som måles og rapporteres være relevante og støtte beslutninger.

2.2.2 Håndtering av regresjonsrisiko med manuelle og automatiserte testtilfeller

I et smidig prosjekt utvikler produktet seg etter hver iterasjon. Derfor vil også dekningsområdet for testing øke. Samtidig som kodeendringer i nåværende iterasjon testes, må testerne også verifisere at ingen feil har blitt introdusert i funksjoner som er utviklet og testet i tidligere iterasjoner. Risikoen med å introdusere feil i smidig utvikling er høy på grunn av omfattende kodeendringer (antall linjer kode lagt til, endret, eller slettet fra en versjon til en annen). Siden det å reagere på endringer er et nøkkel-prinsipp innen smidig metodikk, kan endringer også gjøres på tidligere leverte funksjoner for å møte kundens behov. For å holde oppe endringstakten uten å påføre store mengder teknisk gjeld er det kritisk at teamene investerer i testautomatisering på alle testnivå så tidlig som mulig. Det er også kritisk at alle testprodukter som automatiserte tester, manuelle testtilfeller, testdata, og andre testprodukter blir holdt oppdatert med hver iterasjon. Det er sterkt anbefalt at alle testprodukter blir vedlikeholdt i et konfigurasjonsstyringsverktøy. Dette for å sikre versjonskontroll og tilgjengelighet for alle team-medlemmene, og for å støtte påkrevde endringer fra endret funksjonalitet samtidig som historisk informasjon om testproduktene bevares.

På grunn av at fullstendig repetisjon av alle tester sjeldent er mulig, spesielt i tidsbegrensede smidige prosjekter, må testerne allokere tid i hver iterasjon for å vedlikeholde testmateriale. Dvs. se over manuelle og automatiserte testtilfeller fra tidligere og nåværende iterasjoner for å velge testtilfelle som er kandidater for et bibliotek av regresjonstester, og for å utelukke testtilfelle som ikke er relevante lengre. Tester som er skrevet i tidligere iterasjoner for å verifisere spesifikke

funksjoner kan ha liten verdi i senere iterasjoner på grunn av funksjonsendringer eller nye funksjoner som forandrer tidligere funksjoners oppførsel.

Når testere ser over testtilfeller bør de vurdere hvorvidt de passer for automatisering. Teamet må automatisere så mange tester som mulig fra tidligere og nåværende iterasjon. Automatiserte regresjonstester reduserer risikoen for regresjonsfeil med mindre innsats enn manuelle regresjonstester. Redusert innsats på regresjonstesting gir testerne mer frihet til å teste nye egenskaper og funksjoner på en bedre måte i nåværende iterasjon.

Det er kritisk at testere har evnen til å raskt identifisere og oppdatere testtilfeller fra tidligere iterasjoner og/eller leveranser som er påvirket av endringene fra den nåværende iterasjonen. Å definere hvordan teamet designer, skriver, og lagrer testtilfeller bør skje under leveranseplanleggingen. God praksis for testdesign og -implementering bør tas i bruk tidlig og brukes hele tiden. De kortere tidshorisontene for testing og den konstante endringen i hver iterasjon vil forverre effekten av dårlig testdesign- og testimplementasjonspraksis.

Bruken av testautomatisering på alle nivå gjør at smidige team kan gi raske tilbakemeldinger på produktkvalitet. Velskrevne automatiserte tester gir et levende dokument for systemfunksjonalitet [Crispin08]. Ved å sjekke de automatiserte testene og deres tilhørende testresultater inn i konfigurasjonsstyringssystemet, sammenfallende med versjoneringen av produktets bygging, kan smidige team vurdere den testede funksjonaliteten og testresultatene for et hvert bygg på et hvilket som helst tidspunkt.

Automatiserte enhetstester kjører før kildekoden er sjekket inn i konfigurasjonsstyringssystemet for å sikre at kodeendringer ikke ødelegger software-bygget. For å redusere byggestopp, noe som kan forsinke progresjonen for hele teamet, bør ikke kode sjekkes inn uten at alle de automatiserte enhetstestene kjører fint. Automatiserte enhetstestresultater gir umiddelbar tilbakemelding på kode- og byggekvalitet, men ikke på produktkvalitet.

Automatiserte akseptansetester kjører regelmessig som en del av den kontinuerlige integrasjonen av hele systemet. Disse testene kjøres mot et komplett systembygg minst en gang hver dag, men blir generelt ikke kjørt ved hver kodeinnsjekk siden de tar lengre tid å kjøre enn automatiserte enhetstester og kan sinke kodeinnsjekkingen. Testresultatene fra automatiserte akseptansetester gir tilbakemelding på produktkvalitet i forhold til regresjonstest av forrige bygging, men de gir ikke status på den overordnede produktkvaliteten.

Automatiserte tester kan kjøres kontinuerlig mot systemet. Et initialt subsett av automatiserte tester for å dekke kritisk systemfunksjonalitet og integrasjonspunkter bør lages umiddelbart etter et nytt bygg er installert i testmiljøet. Disse testene blir generelt kjent som verifikasjonstest av bygget. Resultater fra verifikasjonstester av bygget gir rask tilbakemelding på utviklet software etter installasjon slik at teamet ikke kaster bort tid på å teste et ustabilt bygg.

Automatiserte tester i regresjonstestbiblioteket blir generelt kjørt som del av det daglige hovedbygget i det kontinuerlige integrasjonsmiljøet, og igjen når et nytt bygg blir installert i testmiljøet. Så snart en automatisert regresjonstest feiler, stopper teamet og undersøker årsakene for den feilede testen. Testen kan ha feilet på grunn av gyldige funksjonelle endringer i nåværende iterasjon, noe som kan gi behov for å oppdatere testtilfelle og/eller brukerhistorie for å reflektere det nye akseptansekriteriet. Alternativt kan testen måtte tas ut hvis en annen test er laget for å dekke endringene. Men hvis testen feilet på grunn av en feil er det god skikk å rette feilen før teamet fortsetter med nye funksjoner.

I tillegg til automatisering av testtilfeller kan følgende testoppgaver også automatiseres:

- Generering av testdata
- Laste testdata inn i system

- Installere bygg i testmiljø
- Tilbakestille testmiljø (for eksempel database eller web-sider) til referanseinstallasjon
- Sammenligne dataresultater

Automatisering av disse oppgavene reduserer arbeidsmengden og gjør at teamet kan bruke tid til å utvikle og teste nye funksjoner.

2.3 Testerens rolle og ferdigheter i et smidig team

I et smidig team må testerne samarbeide tett med alle de andre medlemmene og med andre interessenter i prosjektet. Dette har flere følger for ferdighetene en tester må ha, og aktivitetene de utfører innen et smidig team.

2.3.1 Ferdigheter for testere i smidige team

Testere i smidige team bør ha ferdighetene som nevnes i pensum/syllabus for Foundation Level [ISTQB_FL_SYL]. I tillegg til disse bør en tester i et smidig team være kompetent i testautomatisering, testdrevet utvikling, akseptansetestdrevet utvikling, strukturbasert testing, svart-boks testing, og erfaringsbasert testing.

Ettersom smidige metodikker avhenger av samarbeid, kommunikasjon og interaksjon mellom team-medlemmene, i tillegg til eksterne representanter, bør testere i et smidig team ha gode mellommenneskelige (sosiale) evner. Testere i smidige team bør:

- Være positiv og løsningsorientert med team-medlemmer og eksterne representanter
- Vise kritisk, kvalitetsorientert og skeptisk tenkning om produktet
- Aktivt skaffe informasjon fra eksterne representanter (i stedet for å kun forholde seg til skrevne spesifikasjoner)
- Nøyaktig evaluere og rapportere testresultater, testprogresjon og produktkvalitet
- Arbeide effektivt for å definere testbare brukerhistorier, spesielt akseptansekriterier, med representanter fra kunde og andre eksterne
- Samarbeide innen teamet, arbeide i par med utviklere og andre team-medlemmer
- Reagere raskt på endringer, inkludert å endre, lage eller forbedre testtilfeller
- Planlegge og organisere eget arbeid

Det er svært viktig at testere, også testere i agile prosjekter, hele tiden videreutvikler sine evner og ferdigheter, også de sosiale.

2.3.2 Rollen til en tester i et smidig team

Rollen til en tester i et smidig team inkluderer aktiviteter som genererer og gir tilbakemeldinger ikke bare på teststatus, testprogresjon og produktkvalitet, men også om prosesskvalitet. I tillegg til aktivitetene beskrevet andre steder i pensum/syllabus, omfatter disse aktivitetene:

- Forstå, implementere og oppdatere teststrategien
- Måle og rapportere testdekning over alle aktuelle dekningsdimensjoner
- Sikre god bruk av testverktøyene
- Konfigurere, bruke og styre testmiljø og testdata
- Rapportere feil og arbeide med teamet for å rette disse
- Hjelpe og gi råd til andre team-medlemmer i relevante testaspekter
- Sikre at de aktuelle testoppgavene er definerte og tidsbestemt under leveranse- og iterasjonsplanlegging

- Aktivt samarbeide med utviklere og forretningsrepresentanter for å klargjøre krav - spesielt i forhold til testbarhet, konsistens og kompletthet
- Delta proaktivt i teamets retrospektiver ved å foreslå og utføre forbedringer

Innen et smidig team er hvert team-medlem ansvarlig for produktkvalitet og spiller en rolle i å utføre testrelaterte oppgaver.

Smidige organisasjoner kan komme over noen testrelaterte organisasjonsrisikoer:

- Testere arbeider så tett med utviklere slik at de mister den riktige testinnstillingen
- Testere blir tolerante for eller tier om ineffektivitet eller arbeidspraksis av lav kvalitet i teamet
- Testere kan ikke holde følge med alle endringene i en tidsbegrenset iterasjon
- For å håndtere disse risikoene kan organisasjoner vurdere ulike muligheter for å bevare uavhengigheten som diskutert i avsnitt 2.1.5.

3. Smidige testmetoder, teknikker og verktøy – 480 min.

Nøkkelord

Akseptansekriterier, utforskende testing, ytelsestesting, produktrisiko, kvalitetsrisiko, regresjonstesting, test tilnærming, test charter, testestimering, testutførelse, teststrategi, testdrevet utvikling, rammeverk for enhetstesting
Anmerkning: Ingen norsk begrep er valgt for test charter.

Læremål

3.1 Smidige metoder

- FA-3.1.1 (K1) Gjenkjenne konsepter for testdrevet utvikling, akseptansetestdrevet utvikling, adferdsdrevet utvikling
- FA-3.1.2 (K1) Gjenkjenne konseptet for testpyramiden
- FA-3.1.3 (K2) Kunne oppsummere testkvadrantene og deres relasjoner mot testnivåer og testtyper
- FA-3.1.4 (K3) Være i stand til å kunne praktisere testrollen i Scrum-teamet, i et smidig prosjekt

3.2 Vurdere (produkt-)kvalitetsrisiko og estimering av testinnsats

- FA-3.2.1 (K3) Vurdere kvalitetsrisiko i et smidig prosjekt
- FA-3.2.2 (K3) Estimere testinnsats basert på (produkt-)kvalitetsrisiko og innhold i iterasjoner

3.3 Teknikker i smidige prosjekter

- FA-3.3.1 (K3) Tolke relevant informasjon for å støtte testaktiviteter
- FA-3.3.2 (K2) Forklare risikobærere og interessenter hvordan testbare akseptansekriterier skal defineres
- FA-3.3.3 (K3), Være i stand til å kunne skrive testtilfeller for akseptansetestdrevet utvikling for gitte brukerhistorier
- FA-3.3.4 (K3) Være i stand til å skrive testtilfeller ved å benytte Black-Box testdesign teknikker både for ikke-funksjonell og funksjonell oppførsel for en gitt brukerhistorie
- FA-3.3.5 (K3) Uføre utforskende testing i et smidig prosjekt

3.4 Verktøy i smidige prosjekter

- FA-3.4.1 (K1) Gjenkjenne de ulike verktøy som er tilgjengelige for testere i forhold til bruk og aktiviteter i et smidig prosjekt

3.1 Smidige testmetoder

Det er enkelte framgangsmåter for test som kan følges i ethvert utviklingsprosjekt (smidig eller ikke) for å kunne produsere kvalitetsprodukter. Dette inkluderer å skrive tester på forhånd for å angi rett oppførsel, fokusere på tidlig forebygging, identifikasjon og fjerning av defekter, samt å sørge for at riktige testtyper benyttes til rett tid og som en del av riktig testnivå. De som utøver smidige metoder må ta sikte på å introdusere denne type praksis tidlig. Testere i smidige prosjekter har en sentral rolle i å veilede bruken av denne type testpraksis gjennom hele livssyklusen.

3.1.1 Testdrevet utvikling, akseptansetestdrevet utvikling og adferdsdrevet utvikling

Testdrevet utvikling, akseptansetestdrevet utvikling og adferdsdrevet utvikling er tre komplementære teknikker som benyttes av smidige team for å utføre testing på tvers av de ulike testnivåene.

Hver teknikk er et eksempel på grunnleggende prinsipper ved testing, (fordeler ved tidlig testing og kvalitetssikring) da testene defineres før selve koden skrives.

Testdrevet utvikling
Testdrevet utvikling (TDU) brukes til å utvikle kode ved hjelp av automatiserte testtilfeller. Prosessen for testdrevet utvikling er som følger:

- Definer en test som skal fange opp utviklerens konsepter rundt ønsket funksjon av en liten del av koden.
- Kjør testen. Det er forventet at testen skal feile siden koden ikke eksisterer.
- Skriv koden og utfør deretter testen igjen fram til testen er vellykket.
- Refaktorer koden etter at testen er vellykket. Utfør deretter testen på nytt for å sørge for at den fortsatt er vellykket.
- Gjenta denne prosessen for neste del av koden ved å kjøre den forrige testen og den nylig tillagte testen.

Testene som skrives er primært på enhetsnivå og er kodefokusert, men de kan også skrives på integrasjons- eller systemnivå. Den testdrevne utviklingen fikk sin popularitet gjennom "Ekstrem programmering (XP)" [Beck02], men brukes også i andre smidige metoder og også i sekvensielle livssykluser. Testdrevet utvikling hjelper utviklere til å fokusere på tydelig definerte, forventede resultat. Testene automatiseres og brukes i en kontinuerlig integrasjon.

Akseptansetestdrevet utvikling
Akseptansetestdrevet utvikling [Adzic09] kjennetegnes ved at det ved opprettelse av brukerhistorier defineres akseptansekriterier og tester (se avsnitt 1.2.2). Akseptansetestdrevet utvikling er en tilnærming som bygger på samarbeid som gir alle interessenter en mulighet til å forstå hvordan programvarekomponenten skal fungere og hva utviklere, testere og forretningsrepresentanter trenger for å sikre denne funksjonen. Prosess for akseptansetestdrevet utvikling er forklart i avsnitt 3.3.2.

I akseptansetestdrevet utvikling gjenbrukes testene når det skal gjøres regresjonstesting. Det er spesifikke verktøy som støtter produksjon og utføring av slike tester, ofte innenfor en kontinuerlig integrasjonsprosess. Disse verktøyene kan koble seg til data- og tjenestenivåer i applikasjonen, som tillater at tester kan kjøres på system- eller akseptansenivå. Akseptansetestdrevet utvikling gir mulighet for rask feilretting og validering av framtidig adferd. Det er med på å avgjøre om akseptansekriteriet er oppfylt for den enkelte funksjon.

Adferdsdrevet utvikling
Adferdsdrevet utvikling [Chelimsky10] tillater at utvikleren fokuserer på testing av kode basert på programvarens forventede adferd. Da testene er basert på forventet adferd er testene oftere lettere å forstå for andre teammedlemmer og interessenter. Spesifikke adferdsdrevne rammeverk for utvikling kan brukes i arbeidet med å definere akseptansekriterier basert på følgende gitt/når/deretter format:

- Gitt ("given") noen innledende kontekster,
- Når ("when") en hendelse inntreffer,
- Deretter ("then") sikre noen resultater.

Ut i fra disse kravene genererer rammeverket til den adferdsdrevne utviklingen kode. Den utviklede kode kan brukes av utviklere for å utvikle testtilfeller. Adferdsdrevet utvikling hjelper utviklere med å samarbeide med interessenter, inkludert testere, for å definere nøyaktige enhetstester fokusert på forretningsbehovet.

3.1.2 Testpyramiden

Et programvaresystem kan bli testet på forskjellige nivåer. Typiske testnivåer er (nedenfra og opp i pyramiden): enhet, integrasjon, system, og akseptanse (se [ISTQB_FL_SYL], avsnitt 2.2).
Testpyramiden legger vekt på gjennomføring av et stort antall tester på de lavere nivåene (bunnen av pyramiden). Når utviklingen beveger seg oppover til de øvre nivåer i pyramiden vil antallet tester avta.

Vanligvis er enhets- og integrasjonstester automatisert og laget ved hjelp av API-baserte verktøy. På system- og akseptansenivå er de automatiserte testene laget ved hjelp av GUI-baserte verktøy. Konseptet med testpyramiden er basert på prinsippet om tidlig kvalitetssikring og tidlig test (dette for å eliminere defekter så tidlig som mulig i livssyklusen).

3.1.3 Testkvadranter, testnivåer og testtyper

Definisjon på testkvadranter er: "å tilpasse testnivåer med passende testtyper i den smidige metodikken" (Brian Marick [Crispin08]). Testkvadrantmodellen og varianter av denne bidrar til å sikre at alle viktige testtyper og testnivåer er inkludert i utviklingsprosessen. Denne modellen gir også en måte å differensiere og beskrive testtyper for alle interessenter, inkludert utviklere, testere og for representanter fra ulike forretningsområder.

I testkvadrantene kan testene være forretnings- (bruker) eller teknologirettet (utvikler). Noen tester støtter arbeidet utført av smidige team og bekrefter programvarens adferd. Andre tester kan bekrefte eller kritisere produktet. Testene kan være manuelle, helautomatiske, en kombinasjon av manuell og automatisk eller manuell, med støtte av verktøy. De fire kvadrantene er:

- Kvadrant K1 er enhetsnivå, teknologirettet og støtter utviklere. Denne kvadranten inneholder enhetstester. Disse testene bør være automatisert og inkludert i den kontinuerlige integrasjonsprosessen.
- Kvadrant K2 er systemnivå, rettet mot forretningsområde og bekrefter produktets virkemåte. Denne kvadranten inneholder funksjonstester, eksempelvis, brukerhistorier, prototyper for å sjekke brukeropplevelse og simuleringer. Disse testene sjekker akseptansekriteriene og kan være manuelle eller automatiske. Testene er ofte laget i løpet av arbeidet med utvikling av brukerhistorier. Utvikling av tester kan føre til bedre kvalitet på brukerhistoriene. De er nyttige når det opprettes automatiserte regresjonstester.

- Kvadrant K3 er system- eller brukerakseptansenivå og er rettet mot forretningsområder. Kvadranten inneholder tester som kritiserer produktet ved å benytte realistiske situasjoner og data. Denne kvadranten inkluderer utforskende testing, situasjoner, prosessflyt, brukervennlighetstesting, brukerakseptansetesting, alfatesting og betatesting. Testene er ofte manuelle og brukerorienterte.
- Kvadrant K4 er system- eller operasjonelt akseptansenivå og er teknologirettet. Kvadranten inneholder tester som kritiserer produktet. Denne kvadranten inneholder ytelsestester, lasttester, stresstester og skalerbarhetstester, sikkerhetstester, test for vedlikehold, minnehåndtering, kompatibilitet og interoperabilitet, datamigrering, infrastruktur og gjenoppretningstester. Testene er ofte automatiserte.

Under enhver iterasjon kan tester fra en, flere eller alle kvadranter være nødvendig. Testkvadrantene gjelder snarere for dynamisk testing enn for statisk testing.

3.1.4 Testerens rolle

Gjennom hele denne læreplanen er det generelle referanser til smidige metoder og teknikker, og rollen til en tester i de forskjellige smidige prosjektene. I dette avsnittet ses det spesielt på rollen til en tester i et prosjekt som følger Scrum livssyklus [Aalst 13].

Teamarbeid
Et av de fundamentale prinsippene i smidig utvikling er teamarbeid. I smidig metode samarbeider hele teamet som består av utviklere, testere og forretningsrepresentanter. Følgende er organisatorisk og adferdsmessig beste praksis i Scrum team:

- Kryssfunksjonell: Hvert teammedlem har med seg et sett med ferdigheter inn i teamet. Teamet samarbeider om teststrategi, testplanlegging, testspesifikasjoner, testutførelse, testevaluering og rapportering av testresultat.
- Selvorganiserende: Temaet kan bestå av bare utviklere, men som nevnt i avsnitt 2.1.5 vil det være ideelt med en eller flere testere.
- Samlokalisert: Testere sitter sammen med utviklere og produkteier.
- Samarbeid: Testere samarbeider med teamets medlemmer, andre grupper, interessentene, produkteier og Scrum Master.
- Fullmakt: Tekniske beslutninger som gjelder design og testing blir avgjort av teamets medlemmer (utviklere, testere og Scrum Master) i samarbeid med produkteier og andre team om nødvendig.
- Forpliktet: Testeren er forpliktet til å stille spørsmål og vurdere produktets adferd og egenskaper i forhold til forventningene og behovene til kunder og brukere.
- Transparent: Progresjon av utvikling og test er synlig på den smidige oppgavetavlen. Se avsnitt 2.2.1
- Troverdig: Testeren må sikre testingens troverdighet, implementering og gjennomføring. Dersom troverdighet ikke er til stede vil interessentene heller ikke stole på testresultatene. Troverdighet oppnås ofte ved å gi informasjon til interessentene om testprosessen.
- Åpen for tilbakemeldinger: Tilbakemelding er en viktig del av å lykkes i ethvert prosjekt, spesielt i smidige prosjekter. Tilbakeblikk tillater teamene å lære av suksesser og feil.
- Elastiske: I likhet med alle andre aktiviteter i smidige prosjekter må også testaktiviteter være i stand til å reagere på endringer.

Punktene ovenfor maksimerer sannsynligheten for vellykket testing i Scrum prosjekter.

Sprint null

Sprint null er den første iterasjonen i prosjektet hvor det foregår mange forberedelsesaktiviteter (se avsnitt 1.2.5). Testeren samarbeider med teamet på følgende aktiviteter i løpet av denne iterasjonen:

- Identifisere omfanget av prosjektet (ved hjelp av «backlog» / kø)
- Lage en første systemarkitektur og prototyper på et høyt nivå
- Planlegge, anskaffe og installere nødvendige verktøy (for eksempel for testledelse, feilhåndtering, testautomatisering og kontinuerlig integrasjon)
- Utarbeide en foreløpig teststrategi for alle testnivåer, adressering, testomfang, teknisk risiko, testtyper (se avsnitt 3.1.3), og mål for testdekning
- Utføre en første analyse av kvalitetsrisiko (se avsnitt 3.2.1)
- Definere måledata som skal benyttes for å måle testprosessen, fremdriften av testingen i prosjektet og produktkvalitet
- Angi definisjonen av "ferdig"
- Utforme oppgavetavla (se avsnitt 2.2.1)
- Definer når test skal fortsette og når test skal avsluttes før leveranse av systemet til kunden

Sprint null setter mål for hva testing skal oppnå og hvordan test skal utføres for å oppfylle målene som settes for de enkelte sprintene.

Integrasjon
I smidige prosjekter er målet at kunden kontinuerlig skal oppleve en verdi (helst i hver sprint). For å oppnå dette bør integrasjonsstrategien inneholde både design og test. For å muliggjøre en kontinuerlig teststrategi for levert funksjonalitet og egenskaper er det viktig å identifisere alle avhengigheter mellom underliggende funksjonalitet og spesifikasjoner.

Planlegging av test
Siden test er fullt integrert i det smidige teamet bør planlegging av test starte parallelt med planleggingen av leveransen og oppdateres for hver sprint. Planlegging av test for leveranser og for hver sprint bør inkludere de punktene som ble tatt opp i avsnitt 1.2.5.

Planleggingen av sprintene resulterer i oppgaver som synliggjøres ved hjelp av en oppgavetavle. Hver oppgave bør ha en estimert tid før gjennomføring på 1-2 dager. Alle testaktiviteter bør synliggjøres slik at en kontinuerlig test kan utføres.

Smidige testmetoder
Flere testmetoder er nyttige for testere i smidige team. Noen av disse er:

- Parvis arbeide: To teammedlemmer (eksempelvis en tester og en utvikler, to testere eller en tester og en produkteier) sitter sammen og gjennomfører testene eller andre oppgaver som er lagt inn i sprinten.
- "Voksende" testdesign: Tester og diagrammer bygges gradvis ved hjelp av brukerhistorier og annen grunnlag for testen. Ofte startes det med enkle tester som deretter videreutvikles til mer kompliserte tester.
- Tankekart: Bruk av tankekart er et nyttig hjelpemiddel under test [Crispin08]. Et eksempel er at testere kan bruke tankekart i arbeidet med å identifisere hvilke tester som skal gjøres, synliggjøre teststrategi og beskrive testdata.

Disse metodene, i tillegg til andre metoder, er beskrevet i pensum og i kapittel 4 i Foundation Level pensum [ISTQB_FL_SYL].

3.2 Evaluering av kvalitetsrisiko og estimering av testinnsats

Et typisk mål for testing i alle prosjekter, smidige eller tradisjonelle, er å redusere risiko med produktkvaliteten slik at det har et akseptabelt nivå i forkant av leveranse. For å identifisere kvalitetsrisiko (eller produktrisiko) kan testere i smidige prosjekter benytte samme teknikker som i tradisjonelle prosjekter. Testerne vurderer tilknyttede risikonivåer og beregner den innsatsen som kreves for å redusere risikoen godt nok. Risikoen skal så dempes ved hjelp av testdesign, implementering og utførelse. Det må påpekes at det, siden smidige prosjekter har korte iterasjoner og høy endringshastighet, vil være nødvendig med noen tilpasninger av teknikkene.

3.2.1 Evaluering av kvalitetsrisiko i smidige prosjekter

En av flere utfordringer i testing er rett valg, tildeling og prioritering av testbetingelser. Dette innebærer å avgjøre rett mengde innsats som må tildeles for å dekke hver funksjon med tester, sette de resulterende testene i en rekkefølge slik at effektiviteten blir optimalisert og at effektiviteten til testarbeidet som skal utføres blir optimal. Identifisering av risiko, analyse av risiko og risikoreduserende strategier kan benyttes av testere i smidige team for å få hjelp til å avgjøre antall testtilfeller som skal kjøres. Siden det er et samspill mellom mange begrensninger og variabler kan det være nødvendig med kompromiss.

Risiko kan føre til negativt, uønsket resultat eller hendelse. Risikonivå identifiseres ved å vurdere sannsynligheten av at risikoen oppstår og hva innvirkningen av risikoen vil være. Når hovedeffekten til det potensielle problemet rammer produktkvaliteten, blir potensielle problemer betegnet som kvalitetsrisiko eller produktrisiko. Når hovedeffekten til det potensielle problemet påvirker prosjektets suksess blir det potensielle problemet betegnet som prosjektrisiko eller planleggingsrisiko [Black07], [vanVeenendaal12].

Kvalitetsrisikoanalyser i smidige prosjekter foregår på to områder:

- Planlegging i forhold til leveranse: forretningsrepresentanter som kjenner til funksjonene i leveransen gir oversikt over risikoer på høyt nivå. Hele teamet, inkludert tester(e), kan hjelpe til med risikoidentifisering og vurdering.
- Iterasjonsplanlegging: teamet identifiserer og vurderer kvalitetsrisikoen

Eksempler på kvalitetsrisiko for et system inkluderer:

- Feilkalkulering i rapporter (funksjonell risiko relatert til nøyaktighet)
- Treg respons på brukeraksjoner (en ikke-funksjonell risiko relatert til effektivitet og responstid).
- Vanskeligheter med å finne fram i skjermer og felt (en ikke-funksjonell risiko relatert til brukervennlighet og forståelighet)

Som tidligere nevnt begynner en iterasjon med iterasjonsplanlegging. Planleggingen kulminerer i beregnede oppgaver på oppgavetavlen. Disse oppgavene kan delvis prioriteres i forhold til nivået på kvalitetsrisiko. Oppgaver som har høy risiko bør startes tidligere og inneholde høy testinnsats. Oppgaver med lavere risiko kan startes senere og inneholde en lavere testinnsats.

Følgende steg beskriver et eksempel på hvordan prosessen til kvalitetsrisikoanalysen i et smidig prosjekt kan utføres i løpet av iterasjonsplanleggingen:

1. Samle medlemmene i det smidige teamet, inkludert tester(e).
2. Sette opp alle elementene i køen for gjeldene iterasjon (f.eks. på en oppgavetavle).
3. Identifisere kvalitetsrisiko med hvert enkelt element, hvor alle relevante kvalitetsegenskaper blir tatt med i betraktningen.

4. Vurdere alle de identifiserte risikoene hvor to aktiviteter inkluderes: kategorisering av risiko og beslutning av risikonivå basert på innvirkningen og sannsynligheten for feil.
5. Bestemme i hvilken grad testing skal gjøres proporsjonalt med risikonivå.
6. Velg rette testteknikk(er) for å vurdere hver enkelt risiko, basert på risiko, risikonivå og de relevante kvalitetsegenskapene.

Testeren vil deretter utforme, implementere og utføre tester for å redusere risiko. Dette inkluderer funksjoner, oppførsel, kvalitetsegenskaper og egenskaper som påvirker kundens, brukerens og interessentenes grad av tilfredshet.

I løpet av prosjektet bør teamet holde seg oppdatert på tilleggsinformasjon som kan endre grad av risiko og/eller risikonivå som hører til kjente kvalitetsrisiko. Det bør gjøres periodiske justeringer på kvalitetsrisikoanalysen, som igjen resulterer i justeringer av test. Justeringer inkluderer å identifisere ny risiko, revurdere nåværende risikonivå og evaluere effektiviteten av de risikoreduserende aktivitetene.
Kvalitetsrisiko kan også reduseres før test utføres. Et eksempel er at dersom det dukker opp problemer med brukerhistorier i løpet av risikoidentifiseringen så kan teamet gå nøye gjennom brukerhistorien som en avhjelpende strategi.

3.2.2 Estimering av testinnsats basert på innhold og risiko

Ved planlegging av leveransen av produktet estimerer det smidige teamet arbeidet (innsatsen) som må gjennomføres før produktet kan leveres. Estimatet skal også inkludere et estimat av testinnsats. En velbrukt metode for estimering i smidige prosjekter er "planning poker" som er en konsensusbasert teknikk. Produkteieren eller kunden leser brukerhistorien for de som gjør estimatet. Hver og en av de som skal gjøre estimatet har en kortstokk med kortverdier som likner Fibonacci-rekken (f.eks., 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …) eller en annen progresjon for valg (for eksempel størrelser på klær S, M, L, XL osv.). Verdiene representerer antall “Story-poeng”, antall dager, eller andre enheter som skal estimeres. Fibonacci-rekken anbefales da tallene i sekvensen reflekterer at usikkerhet vokser med størrelsen på brukerhistorien. Et resultat med høyt estimat betyr vanligvis at brukerhistorien kan være misforstått eller det betyr at brukerhistorien bør deles opp i mindre brukerhistorier.

De som skal gjøre estimatet skal drøfte funksjonaliteten og stille spørsmål angående produktet dersom de trenger mer informasjon. Utvikling, testmengde, brukerhistoriens kompleksitet og omfang av test virker inn på estimeringen. Før "poker planning" starter er det anbefalt å avklare risikonivået til elementene i køen i tillegg til prioritet som produkteier har oppgitt. Når funksjonaliteten er diskutert fullt ut skal hver av de som utfører estimeringen velge et kort som representerer hans/hennes estimat. Deretter viser alle kortene sine samtidig. Dersom alle har samme verdi på kortet er det denne verdien som blir estimatet. Dersom det er ulike verdier på kortene diskuteres dette før en ny runde gjennomføres. Det fortsettes med "planning poker" helt til enighet oppnås eller til eventuelt innførte regler oppfylles (eksempelvis bruk av median eller høyeste verdi). Reglene innføres slik at antall runder begrenses. Diskusjonen sikrer et troverdig estimat for å ferdigstille oppgavene som ligger i produktkøen og for å synliggjøre hvilke oppgaver som skal gjøres.

3.3 Teknikker i smidige prosjekter

Mange av testteknikkene og testnivåene som blir brukt i tradisjonelle prosjekter kan også brukes i smidige prosjekter. Men det er spesielle hensyn og varianter av testteknikker, terminologi og dokumentasjon som bør vurderes for smidige prosjekter.

3.3.1 Akseptansekriterier, tilstrekkelig testdekning og annen informasjon relevant for test

I smidige prosjekter skisseres kravene som brukerhistorier, ved hjelp av en prioritert backlog (kø), i den innledende fasen. De innledende kravene er korte og følger et forhåndsdefinert format (se avsnitt 1.2.2). Ikke-funksjonelle krav, slik som brukervennlighet og ytelse, er også viktig og kan spesifiseres som unike brukerhistorier eller koblet til andre funksjonelle brukerhistorier. Ikke-funksjonelle krav kan følge et forhåndsdefinert format eller standard, for eksempel [ISO25000], eller en bransjespesifikk standard.

Brukerhistorier utgjør et viktig grunnlag for utvikling av tester. Andre mulige testgrunnlag er:

- Erfaring fra tidligere prosjekter
- Eksisterende funksjoner, funksjonalitet og systemets kvalitetsegenskaper
- Kode, arkitektur og design
- Brukerprofiler (kontekst, systemkonfigurasjoner og brukeradferd)
- Informasjon om feil fra eksisterende og tidligere prosjekter
- En kategorisering av feil i en taksonomi for feil
- Gjeldende standarder (f.eks. [DO-178B] for programvare i flybransjen)
- Kvalitetsrisiko (se avsnitt 3.2.1)

Under hver iterasjon lager utviklerne kode for de funksjoner og egenskaper som er beskrevet i brukerhistoriene. Koden verifiseres og valideres ved hjelp av akseptansetest. For å være testbare bør akseptansekriteriene inneholde følgende [Wiegers13]:

- Funksjonell adferd: Brukerens handlinger, m.a.o. input under visse konfigurasjoner, viser seg i den utenfor observerbare adferden til systemet.
- Kjennetegn på kvalitet: Hvordan systemet utfører den angitte adferd. Egenskapene kan også bli referert til som kvalitetsegenskaper, eller ikke-funksjonelle krav. Vanlige kvalitetsegenskaper er ytelse, pålitelighet, brukervennlighet osv.
- Scenarier (brukerhistorier): En sekvens av handlinger mellom en ekstern aktør (ofte en bruker) og systemet for at et bestemt mål eller virksomhetsoppgave skal oppnås.
- Forretningsregler: Aktiviteter som kun kan utføres i systemet under visse vilkår som er definert av eksterne prosedyrer og begrensninger (f.eks. prosedyrer som brukes av et forsikringsselskap for å håndtere forsikringskrav).
- Eksterne grensesnitt: Beskrivelser av forbindelsene mellom systemet som skal utvikles og omverdenen. Eksterne grensesnitt kan deles inn i ulike typer (brukergrensesnitt, grensesnitt mot andre systemer etc.).
- Begrensninger: Design og implementering som begrenser utviklerens muligheter. Enheter med innebygd programvare må ofte respektere fysiske begrensninger som størrelse, vekt og grensesnitt.
- Datadefinisjoner: Kunden kan beskrive format, datatype, tillatte verdier, og standardverdier for et dataelement i sammensetningen av en kompleks virksomhetsstruktur (f.eks. postnummer i postadresse).

I tillegg til brukerhistorier og deres tilhørende akseptansekriterier er følgende informasjon relevant for testere:

- Hvordan systemet er ment å fungere og tenkt brukt
- Systemets grensesnitt som kan brukes og som gir tilgang til å teste systemet
- Om gjeldende verktøystøtte er tilstrekkelig

- Om testeren har nok kunnskap og ferdigheter til å utføre de nødvendige testene

Testere vil ofte oppdage behovet for ytterligere informasjon (for eksempel kodedekning) i iterasjonene. Testerne skal samarbeide med resten av medlemmene i det smidige teamet for å tilegne seg denne informasjonen. Relevant informasjon spiller en rolle når det skal avgjøres om en bestemt aktivitet kan regnes som ferdig. Definisjonen av ferdig er kritisk i smidige prosjekter og definisjonen kan være svært ulik (som beskrevet i andre avsnitt).

Testnivåer
Hvert testnivå har sin egen definisjon av "ferdig". Listen nedenfor gir eksempler som er relevant for de ulike testnivåene.

- **Enhetstesting**
  - 100 % beslutningsdekning der dette er mulig. Det gjøres nøye vurderinger av eventuelle kode som er umulig å utføre
  - Statisk analyse gjøres på all kode
  - Ingen alvorlige defekter forblir urettede (rangeres ut ifra prioritet og alvorlighet)
  - Ingen kjente uakseptable tekniske usikkerheter skal gjenstå i design og kode [Jones11]
  - All kode, enhetstester og resultater på enhetsnivå skal gjennomgås
  - Alle enhetstester skal automatiseres
  - Viktige egenskaper ved systemet er innenfor de avtalte rammer (eksempelvis ytelse)

- **Integrasjonstesting**
  - Alle funksjonelle krav er testet, med både positive og negative tester. Antall tester skal være basert på størrelse, kompleksitet og risiko
  - Alle grensesnitt er testet
  - Alle kvalitetsrisiko er testet som avtalt
  - Ingen gjenstående alvorlige defekter (prioritert etter viktighet og risiko)
  - Alle feil som er funnet ved hjelp av test er rapportert
  - Alle regresjonstester er automatisert (der dette er mulig) og er lagret i et felles repository

- **Systemtest**
  - Det er utført ende-til-ende test på brukerhistoriene og funksjonaliteten.
  - Alle ulike roller på brukere samt ulik brukerbakgrunn er dekket
  - De viktigste kvalitetsegenskapene ved systemet er dekket (eksempelvis ytelse, robusthet og pålitelighet)
  - Testingen er utført i et produksjonslikt miljø. Hardware og software for alle konfigureringer er testet så grundig det produksjonslike miljøet tillater.
  - Kvalitetsrisiko er testet som avtalt
  - Alle regresjonstester er automatisert (der dette er mulig) og er lagret i et felles repository
  - Alle feil er rapportert og om mulig rettet
  - Ingen gjenstående alvorlige feil (prioritert etter viktighet og risiko)

- **Brukerhistorier**

For oppnåelse av definisjon "ferdig" for brukerhistorier må følgende kriterier oppfylles:

- Brukerhistoriene i iterasjonen er komplette, teammedlemmene har forstått brukerhistoriene og det er utarbeidet detaljerte, testbare akseptansekriterier
- Alle elementene i brukerhistoriene er spesifisert og gjennomgått. Dette inkluderer også ferdigstillelse av brukerhistoriene for akseptansetest.
- Oppgaver som er nødvendige for implementering og test av de utvalgte brukerhistoriene er identifisert og estimert av teamet.

- **Funksjoner**

For oppnåelse av definisjon "ferdig" for funksjoner, som kan også omfatte komplekse (sammensatte) brukerhistorier, kan følgende kriterier være aktuelle:

- Alle brukerhistorier, med akseptansekriterier, er definert og godkjent av kunden
- Designet er komplett, ingen gjenstående uklarheter rundt det tekniske
- Koden er komplett, ingen uklarheter rundt det tekniske eller gjenstående forbedringer
- Enhetstester er utført og det definerte nivået for testdekning er oppnådd
- Integrasjonstester og systemtester er gjennomført i tråd med de definerte dekningskriterier
- Ingen gjenstående alvorlige feil
- Dokumentasjon av funksjonalitet er komplett (inkludert release notes, brukermanual og on-line hjelpefil)

- **Iterasjon**

For oppnåelse av definisjon "ferdig" for iterasjoner kan følgende kriterier være aktuelle:

- Alle funksjoner I iterasjonen er ferdigstilt og individuelt testet i tråd med kriteriene satt til den enkelte funksjon
- De feil som ikke er rettet og ikke er kritiske for iterasjonen registreres i produktkøen med prioritet
- Integrasjonene som skal gjøres i iterasjonen er ferdigstilt og testet
- Dokumentasjon er skrevet, gjennomgått og godkjent

På dette stadiet er programvaren potensielt sett klar til leveranse da iterasjonen er fullført. Men alle iterasjoner ender ikke med leveranse da det kan være resterende påfølgende iterasjoner som må ferdigstilles før en eventuell leveranse.

- **Leveranse**

For oppnåelse av definisjon "ferdig" for en leveranse, som kan inneha flere iterasjoner, kan følgende kriterier være aktuelle:

- Dekningsgrad: Alle relevante tester er gjort. Tilstrekkeligheten av testene bestemmes av hva som er ny eller endret funksjonalitet, kompleksitet og risiko for defekter.
- Kvalitet: Det er enighet om at antall feil (f.eks. hvor mange feil som blir funnet per dag eller per transaksjon), feilfrekvens (f.eks. antall feil funnet i forhold til antall brukerhistorier, mengde og/eller kvalitetsegenskaper), antall gjenværende feil i forhold til akseptable grenser, konsekvensene av uløste og øvrige feil (f.eks. alvorlighetsgrad og prioritet) er forstått og akseptert. Risiko knyttet til hver identifisert kvalitetsrisiko er forstått og akseptert.
- Tid: Dersom det forhåndsbestemte tidspunktet for levering er nådd, gjøres betraktninger relatert til forretningshensyn som er forbundet med leveranse (eller ikke) av systemet
- Kostnad: De estimerte livssykluskostnadene benyttes til beregning av avkastning på investeringen for det leverte systemet. Hoveddelen av kostnadene angår som

oftest vedlikehold da det vanligvis oppstår/oppdages feil etter produksjonssetting av systemet.

3.3.2 Bruk av testdrevet utvikling

Akseptansetestdrevet utvikling er en “test først” tilnærming. Testbeskrivelser er laget før implementering av brukerhistorier. Testbeskrivelsene kan være manuelle eller automatiserte og er laget av det smidige teamet (utvikler, tester og representant fra forretningen) [Adzic09]. Første steget er en workshop hvor spesifisering er i fokus. Her analyseres, diskuteres og skrives brukerhistorier i et samarbeid mellom utviklere, testere og representanter fra forretningen. Eventuelle mangler, uklarheter eller feil i brukerhistoriene løses i løpet av spesifiseringsworkshopen.

Etter spesifiseringsworkshopen utarbeides testene. Dette gjøres enten som et samarbeid hvor alle teammedlemmene deltar, eller kun av testerne. Testene skal deretter valideres av en uavhengig part (for eksempel kunden). Testene er eksempler som beskriver de ulike elementene i en brukerhistorie. Elementene i testene skal hjelpe teamet i arbeidet med å implementere brukerhistorien. Siden eksempler og tester går ut på det samme benyttes disse begrepene om hverandre. Arbeidet startes med enkle eksempler og åpne spørsmål.

De første testene som gjøres er vanligvis positive tester. Positive tester bekrefter korrekt funksjonalitet (når de kjører uten feil), hvor hele sekvensen av aktiviteter utføres. Etter at de positive testene er utført bør teamet utarbeide negative tester som også dekker det ikke-funksjonelle krav (bl.a. ytelse og brukervennlighet). Testene skal være forståelig for alle interessenter. De skal vise de nødvendige forutsetninger og de nødvendige inndata og resultat for testgjennomføring.

Eksemplene skal dekke alle elementene i brukerhistorien. Det skal ikke skal være nødvendig å legge til flere elementer underveis. Dette betyr at et eksempel kun skal beskrive hendelser som er inkludert og beskrevet i brukerhistorien. Det skal holde med ett eksempel for å beskrive et element i en brukerhistorie.

3.3.3 Funksjonell og ikke-funksjonell ”Black Box” testdesign

I smidig testing lager testerne testene parallelt med at utviklerne programmerer. Testerne lager testene med brukerhistoriene og brukerhistorienes akseptansekriterier som basis (noen tester, slik som utforskende tester og andre erfaringsbaserte tester lages senere underveis i testgjennomføringen, se avsnitt 3.3.4). Testerne kan benytte seg av vanlig «black box» testdesign, slik som ekvivalensklasser, grenseverdianalyser, beslutningstabeller og tilstandsbasert testing, i arbeidet med utarbeidelsen av testene. For eksempel kan grenseverdianalyse benyttes til å velge testverdier når en kunde begrenses i antall produkter han kan kjøpe.

I mange tilfeller kan ikke-funksjonelle krav dokumenteres som brukerhistorier. «Black box» testdesignteknikker (eksempelvis grenseverdianalyse) kan også brukes til å lage tester for ikke-funksjonelle kvalitetsegenskaper. Brukerhistorien kan inneholde ytelses- eller pålitelighetskrav. For eksempel kan en gitt utførelse ikke overskride en gitt tidsbegrensning eller det er et akseptert maksimalt antall ganger en operasjon kan svikte.

For mer informasjon om bruk av "black box" testdesignteknikker, se Foundation Level pensum [ISTQB_FL_SYL] og Advanced Level Test Analyst syllabus [ISTQB_ALTA_SYL].

3.3.4 Utforskende testing og smidig testing

Utforskende testing er en sentral del av smidige prosjekter da det er kort tid til å gjøre testanalysen og fordi brukerhistoriene har et begrenset detaljeringsnivå.

I utforskende testing gjøres testdesign og testgjennomføring på samme tid, veiledet av et forberedt test charter. Et test charter synliggjør testbetingelsene for en tidsbegrenset testsesjon. I utforskende testing leder resultatet av den siste testen utformingen av den neste testen som skal gjennomføres. Teknikker for «White box» og «Black Box» testing kan benyttes i arbeidet med å utforme tester som ved forhåndsdesignet test.

En test charter kan inneholde følgende informasjon:

- Aktør: Forventet bruker av systemet
- Formål: Temaet til charter, inkludert aktørens mål, dvs. testbetingelser
- Oppsett: Hva må være på plass for at testingen kan starte?
- Prioritet: Den relative viktigheten av charteret, basert på prioriteten definert for den tilhørende brukshistorien eller risikonivået
- Referanser: Spesifikasjoner (for eksempel brukerhistorie), risiko eller andre informasjonskilder
- Data: De data som er nødvendige for testgjennomføring
- Aktiviteter: En liste over ideer om hvilke funksjoner aktøren ønsker av systemet (for eksempel “Logg på systemet som en overordnet bruker”) og hva som da er interessant å teste (både positive og negative tester)
- Notater om testorakel: Hvordan skal produktet evalueres for å finne ut om resultatene er korrekte (for eksempel er hendelsene på skjermen i tråd med det som står i brukermanualen)
- Variasjoner: Alternative aksjoner og evalueringer for å utfylle ideene som er beskrevet under aktiviteter.

For å administrere eksplorativ testing kan metoden sesjonsbasert teststyring benyttes. En sesjon er definert som en uavbrutt periode med testing som kan ta mellom 60 og 120 minutter. Testsesjoner inkluderer følgende:

- Overblikksesjoner (for å lære hvordan det virker)
- Analysesesjoner (for å evaluere funksjonen eller egenskaper)

Dyp testdekning (grenseverdier, spesialtilfeller, scenarier, interaksjoner). Kvaliteten til testene avhenger av testerens evne til å stille relevante spørsmål om hva som skal testes. Eksempler omfatter:

- Hva er viktigst å finne ut om systemet?
- På hvilken måte kan det feile?
- Hva skjer hvis …?
- Hva burde skje når …?
- Er kundens behov, krav og forventninger oppfylt?
- Kan systemet installeres (og avinstalleres om nødvendig) for alle støttede oppgraderingsveier?

Under selve testutførelsen bruker testeren kreativitet, intuisjon, kunnskap og evner for å finne mulige problemer ved produktet. Testeren må ha god kunnskap og forståelse av programvaren under test, ha kunnskaper om forretnings- eller problemområdet, kunnskaper om hvordan programvaren blir brukt og hvordan å finne ut når systemet feiler.

Et sett med heuristiske regler (tommelfingerregler) kan benyttes under test. En slik regel kan hjelpe testeren med å forstå hvordan testingen skal gjøres og hvordan evaluering av testresultatene skal gjøres [Hendrickson]. Følgende er eksempler på dette:

- Grenseverdier
- CRUD (Create, Read, Update, Delete)
- Konfigurasjonsvariasjoner
- Avbrytelser (for eksempel logg av, skru av eller omstart)

Det er viktig å dokumentere testprosessen så detaljert som mulig. Uten detaljert dokumentasjon vil det være vanskelig å gå tilbake og se hvordan et problem i systemet ble oppdaget. Listen nedenfor viser eksempler på informasjon som kan være nyttig å dokumentere:

- Testdekning: Hva slags inndata ble benyttet, hvor mye er testet og hvor mye gjenstår det å teste
- Notat angående evaluering: Observasjoner gjort under test, virker systemet og den delen som ble testet stabilt, hva slags feil som ble avdekket, hva er planlagt som neste steg ifølge de aktuelle observasjonene og andre lister av ideer for testingen
- Risiko og strategiliste: hvilke risiko har blitt dekket og hvilke gjenstår og hvilken grad av viktighet har de, blir den opprinnelige strategien fulgt, er det behov for endringer
- Observasjoner, hendelser, spørsmål, avvik: ethvert uventet oppførsel, alle spørsmål omkring hvor effektiv testingen var, enhver tvil om ideene eller testforsøk, testomgivelse, testdata, misforståelser rundt funksjonalitet, testskript eller systemet under test
- Aktuell oppførsel: Logg av systemets aktuelle oppførsel som bør tas vare på (for eksempel video, skjermkopier, outputfiler)

Informasjonen som er logget burde bli lagt inn eller oppsummert i et verktøy for statusoppfølging (for eksempel et teststyringsverktøy, oppgavestyringsverktøy eller oppgavetavle). Det bør lagres på en måte som gjør det lett å forstå aktuell status for all testingen som er utført.

3.4 Verktøy i smidige prosjekter

Verktøy som er beskrevet i Foundation pensum [ISTQB_FL_SYL] er relevante og brukes av testere i smidige team. Ikke alle verktøy brukes på samme måte og noen verktøy er mer relevante for smidige enn tradisjonelle prosjekter. Selv om verktøy for testledelse, kravstyring og feilstyring kan brukes av smidige team vil noen smidige team ta i bruk et verktøy som omfatter alt (for eksempel ALM (application lifecycle management) eller oppgavestyring). Slike verktøy gir funksjonalitet som er relevant til smidig utvikling (slik som oppgavetavler, forbruksgraf/“burndown charts” og brukerhistorier). Konfigurasjonsstyringsverktøy er viktige for testere i smidige team på grunn av det høye antallet automatiske tester på alle nivåer og nødvendigheten for å lagre og styre de tilhørende skripter o.l.

I tillegg til verktøyene som er beskrevet i Foundation pensum [ISTQB_FL_SYL] kan testere i smidige prosjekter også benytte verktøyene som er beskrevet i de påfølgende avsnittene. Disse verktøyene brukes av hele teamet for å sikre samarbeid og deling av informasjon, noe som er ett av hovedelementene i smidig praksis.

3.4.1 Verktøy for oppgavestyring og oppfølging

I noen tilfeller bruker smidige team fysiske tavler (whiteboard, korktavle) for å synliggjøre oppgaver, brukerhistorier, tester og andre ting som er inkludert i den aktuelle iterasjonen. Andre team bruker ALM eller oppgavestyringsverktøy i tillegg til elektroniske tavler. Slike verktøy hjelper med følgende formål:

- Å loggføre (bruker)historier og deres relevante utviklings- og testoppgaver, for å sikre at ingenting går tapt i en iterasjon
- Å samle estimater til teammedlemmer om deres oppgaver og automatisk beregne hvor mye det koster å implementere en historie, slik at planleggingsmøtene blir mer effektive
- Å knytte utviklings- og testoppgaver til samme historie for å gi et fullstendig bilde av hva teamet må gjøre for å implementere den
- Å sammenstille oppdateringer fra utviklere og testere om status på oppgaver etter hvert som de er ferdige. Dermed gir de automatisk et øyeblikksbilde av status for hver historie, iterasjonen og hele utgivelsen.
- Å gi en visuell representasjon (gjennom måledata, grafer og ”dashboards”) av nåværende status for hver brukerhistorie, iterasjon, og leveranse. Dette tillater alle interessenter, inklusive folk på geografisk distribuerte team, å raskt oppdatere seg om status
- Å integrere med konfigurasjonsstyringsverktøy. Dette kan tillate automatisk logging av check-in av kode og bygg mot oppgaver og eventuelt automatisk statusoppdatering for oppgaver

3.4.2 Verktøy for kommunikasjon av og deling av informasjon

I tillegg til e-post, dokumenter og verbal kommunikasjon bruker smidige team ofte tre ekstra typer verktøy for kommunikasjonsstøtte og deling av informasjon. Dette er wikis, direktemeldinger (instant messaging), og desktop-/skjermdeling.

Wikis tillater oppbygging av og deling av en online kunnskapsbase som inneholder forskjellige aspekter av prosjektet. Dette kan være:

- Diagrammer om produktets funksjoner, diskusjoner om funksjoner, diagrammer om prototyper, fotografier av resultat av diskusjoner på whiteboard og annen informasjon
- Verktøy for utvikling og/eller testing som andre deltakere i teamet har funnet brukbare
- Måledata, grafer og visninger av produktstatus. Dette er spesielt nyttig når en wiki er integrert med andre verktøy som eksempelvis byggserveren og oppgave-forvaltningssystemet. Dette fordi verktøyet kan oppdatere produktstatus automatisk
- Diskusjoner mellom teammedlemmene. Dette kan være som instant messaging og e-mail, men på en slik måte at informasjonen deles med alle teammedlemmene. Instant messaging, telefonkonferanse- og videokonferanseverktøy gir følgende nytteverdier:
  - De tillater direkte kommunikasjon i sanntid mellom teammedlemmene, spesielt ved distribuerte team
  - De involverer distribuerte team i stående møter
  - De reduserer telefonregningen fordi en bruker IP-telefoni slik at kostnadsproblemer med kommunikasjonen ved distribuerte team unngås.

Skjermdeling og verktøy som kan logge det som er på skjermen gir følgende nytteverdi:

- I distribuerte team er det mulig å gjøre produktdemonstrasjoner, kodereviews og parvis arbeid
- De gjør det mulig å lagre produktdemonstrasjoner på slutten av hver iterasjon, og disse kan legges på teamets wiki

Disse verktøyene bør benyttes for å komplettere og øke kommunikasjon, ikke for å erstatte ansikt-til-ansikt kommunikasjon i smidige team.

3.4.3 Verktøy for å bygge og distribuere programvare

Som diskutert tidligere er daglig bygging og idriftsetting en sentral praksis i smidige team. Dette krever bruk av verktøy for kontinuerlig integrasjon og for distribuering av bygget. Bruken, nytten og risiko med slike verktøy er beskrevet i avsnitt 1.2.4.

3.4.4 Verktøy for konfigurasjonsstyring

I smidige team blir konfigurasjonsstyringsverktøy ikke bare benyttet til å lagre kildekode og automatiserte tester, men også manuelle tester og andre arbeidsprodukter blir ofte lagret i same bibliotek som produktets kildekode. Dette gir sporbarhet mellom hvilke versjoner av programvaren som ble testet og med hvilken versjon av testene. Det tillater rask endring uten å miste informasjon om brukerhistorien. Versjonsstyringssystem finnes I både sentralisert og distribuert form. Teamets størrelse, struktur, plassering og krav til integrasjon med andre verktøy bestemmer hvilket versjonsstyringssystem som er det rette for det smidige prosjektet.

3.4.5 Verktøy for testdesign, implementering og utføring

Noen verktøy er brukbare for smidige testere på spesielle faser i testprosessen. De fleste verktøyene er ikke nye eller spesifikt laget for smidige prosjekter. De tilbyr funksjonalitet som er viktig i arbeidet med raske og hyppige endringer som tas og gjøres i smidige prosjekter.

- Verktøy for testdesign: Bruk av verktøy som tankekart er blitt populære for rask design og testdesign for nye produktområder.
- Verktøy for forvaltning av tester: Denne typen verktøy brukt i smidige prosjekter kan fungere som hele teamets ALM-verktøy eller testledelsesverktøy.
- Verktøy for å lage eller generere testdata: Verktøy som genererer data for å fylle databasen til en applikasjon er veldig nyttige når en behøver store mengder data og kombinasjoner av data i forbindelse med test. Disse verktøyene kan også hjelpe med å redefinere databasestrukturen når et produkt gjennomgår endringer under et smidig prosjekt, samt med å rydde opp i skriptene for å generere disse data. Dette hjelper med rask oppdatering av testdata når endringer gjøres. Noen verktøy for forberedelse av testdata benytter produksjonsdata som grunnlag og ved hjelp av scripter tas sensitive data bort eller de anonymiseres. Andre slike verktøy kan hjelpe med å validere store mengder av input- og outputdata.
- Verktøy for å laste opp testdata: Etter at data har blitt generert for testing må disse lastes inn i applikasjonen. Manuell input av data er ofte langsom og feilbefengt. Men datalastingsverktøy finnes og de kan gjøre prosessen pålitelig og effektiv. Mange datagenereringsverktøy inneholder en komponent som laster disse dataene opp. I andre tilfeller er det mulig å laste opp store datamengder i selve databaseverktøyet.
- Automatiske testutføringsverktøy: Det finnes noen automatiske testutføringsverktøy som er bedre tilpasset til smidig testing. Spesifikke verktøy finnes både som kommersielle og som åpen-kildekode verktøy for å støtte framgangsmåter som å teste først, adferdsdrevet utvikling, testdrevet utvikling, og akseptansetestdrevet utvikling. Disse verktøyene tillater testere og forretningsfolk (brukere) å vise forventet systemoppførsel i tabeller eller naturlig språk med forhåndsdefinerte stikkord.
- Eksplorative (utforskende) testverktøy: Verktøy som tar opp og logger aktiviteter som en tester har gjort med en applikasjon under en utforskende test er nyttige for testeren og utvikleren fordi de tar opp det som har hendt. Dette er hendig når en finner en feil nettopp fordi aksjonene før feilen oppstod er tatt opp og dermed kan brukes for å beskrive feilen til utviklerne. Å logge hva som er gjort i en slik sesjon kan vise seg å være nyttig hvis testen til slutt blir lagt til den automatiske regresjonstestsamlingen.

3.4.6 Verktøy for bruk av skytjenester og virtualisering

Virtualisering tillater at en enkel fysisk ressurs (en server) arbeider som mange adskilte, mindre ressurser. Når en bruker virtuelle maskiner eller maskiner i skyen, har teamene et større antall servere tilgjengelige for utvikling og test. Dette kan være nyttig for å unngå forsinkelser som henger sammen med ventingen på fysiske servere. Å få tatt i bruk en ny server eller å restaurere/gjennopprette en server går raskere dersom det er funksjonalitet for å ta øyeblikksbilder ("snapshots"). Denne funksjonen kan være innebygget i de fleste virtualiseringsverktøy. Noen av testledelsesverktøyene bruker virtualiseringsteknikker for å ta øyeblikksbilder av servere på det tidspunktet en feil blir oppdaget. Dette tillater testere å dele øyeblikksbildet med utviklerne som skal undersøke feilen.

4. Referanser

4.1 Standarder

- [DO-178B] RTCA/FAA DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.
- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE), 2005.

4.2 ISTQB dokumenter

- [ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus, Version 2012
- [ISTQB_ALTM_SYL] ISTQB Advanced Level Test Manager Syllabus, Version 2012
- [ISTQB_FA_OVIEW] ISTQB Foundation Level Agile Tester Overview, Version 1.0
- [ISTQB_FL_SYL] ISTQB Foundation Level Syllabus, Version 2011, på norsk tilgjengelig på www.istqb.no

4.3 Bøker

- [Aalst13] Leo van der Aalst and Cecile Davis, “TMap NEXT® in Scrum”, ICT-Books.com, 2013.
- [Adzic09] Gojko Adzic, “Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing”, Neuri Limited, 2009.
- [Anderson13] David Anderson, “Kanban: Successful Evolutionary Change for Your Technology Business”, Blue Hole Press, 2010.
- [Beck02] Kent Beck, “Test-driven Development: By Example”, Addison-Wesley Professional, 2002.
- [Beck04] Kent Beck and Cynthia Andres, “Extreme Programming Explained: Embrace Change, 2nd ed.”, Addison-Wesley Professional, 2004.
- [Black07] Rex Black, “Pragmatic Software Testing”, John Wiley and Sons, 2007.
- [Black09] Rex Black, “Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 3rd ed.”, John Wiley and Sons, 2009.
- [Chelimsky10] David Chelimsky et al, “The RSpec Book: Behavior Driven Development with Rspec, Cucumber, and Friends”, Pragmatic Bookshelf, 2010.
- [Cohn04] Mike Cohn, “User Stories Applied: For Agile Software Development”, Addison-Wesley Professional, 2004.
- [Crispin08] Lisa Crispin and Janet Gregory, “Agile Testing: A Practical Guide for Testers and Agile Teams”, Addison-Wesley Professional, 2008.
- [Goucher09] Adam Goucher and Tim Reilly, editors, “Beautiful Testing: Leading Professionals Reveal How They Improve Software”, O'Reilly Media, 2009.
- [Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, “Extreme Programming Installed”, Addison-Wesley Professional, 2000.
- [Jones11] Capers Jones and Olivier Bonsignour, “The Economics of Software Quality”, Addison-Wesley Professional, 2011.
- [Linz14] Tilo Linz, “Testing in Scrum: A Guide for Software Quality Assurance in the Agile World”, Rocky Nook, 2014.
- [Schwaber01] Ken Schwaber and Mike Beedle, “Agile Software Development with Scrum”, Prentice Hall, 2001.
- [vanVeenendaal12] Erik van Veenendaal, “The PRISMA Approach”, Uitgeverij Tutein Nolthenius, 2012.

- [Wiegers13] Karl Wiegers and Joy Beatty, “Software Requirements, 3rd ed.”, Microsoft Press, 2013.

4.4 Agil terminologi

Begreper som finnes i ISTQBs terminologiliste står listet i starten av hvert kapitel. For vanlige begreper i den agile (smidige) verden har vi brukt følgende Internet-ressurser for å finne frem til definisjoner:

http://guide.Agilealliance.org/
http://whatis.techtargetcom/glossary
http://www.scrumalliance.org/

Vi oppfordrer leserne å sjekke disse sidene hvis de finner ukjente begreper i dette dokumentet. Lenkene ovenfor eksisterte da dette dokumentet ble utgitt.

4.5 Andre referanser

De følgene referansene viser til informasjon som finnes på internett. Tilgjengeligheten på referansene ble sjekket da denne syllabus ble publisert, men vi kan ikke påta oss ansvar dersom lenkene ikke er tilgjengelige lenger.

- [Agile Alliance Guide], http://guide.Agilealliance.org/.
- [Agilemanifesto], http://www.agilemanifesto.org.
- [Hendrickson]: Elisabeth Hendrickson, “Acceptance Test-driven Development”, http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview.
- [INVEST] Bill Wake, “INVEST in Good Stories, and SMART Tasks”, http://xp123.com/articles/invest-in-good-stories-and-smart-tasks.
- [Kubaczkowski] Greg Kubaczkowski and Rex Black, “Mission Made Possible”, http://www.rbcs-us.com/images/documents/Mission-Made-Possible.pdf.