

Kompendie

TDT4242

May 16, 2012

Contents

Traceability	12
Challenges	12
Krav utvikling	12
Two types of requirements statements	12
Formulering av systemkrav	13
Domeneegenskaper	13
Målorientering	13
Typer mål	13
Målkategorisering	14
Goal refinement (Avoid, maintain, helt ned til agenter)	15
Kvalitetsmålprioritet til krav (Quality metrics)	15
Hvor får man mål fra?	15
Oppsummering	16
I dette kurset	16
Guidet naturlig språk (GNL) og Boilerplates (BP)	16
Finne krav:	16
Mennesker og Maskiner	17
GNL	17
Boilerplates	17
Oppsummering	18

Sporbarhet av krav	19
Utfordringer	20
Metamodeller	20
Tilnærminger til sporbarhet	22
Manuell sporingskobling	22
Scenario-drevet sporbarhet	23
Utviklingsfotspor	23
Ulemper ved bruk av scenario-drevet sporbarhet	24
Sporing via tagging	25
Konklusjon	26
Nyttige lenker	26
Kravsspesifikasjon og testing	26
Testabilitet	26
Utfordringer	28
Design by Objective	28
Krav til testbarhet	29
Noen tips til utforming av krav	30
Implementasjonen	30
Oppsummering	31
Requirements Traceability	31
Challenges	32
Meta-model	32
Tracability	32
Manuell	32
Scenario-driven	32
Tagging	32
Oppsummering	33

Test vs. inspection	33
Fit's List	34
Generell konsiderasjon	35
Dokumenter	35
Kode	35
Inspeksjonsprosesser	35
Walkthrough	35
Uformell inspeksjon	36
Formell inspeksjon	37
Fordeler	39
Ulemper	39
Testprosesser	39
Enhetstester	39
Fordeler	40
Ulemper	40
Funksjonell verifikasjonstesting – Systemtesting	40
Fordeler	40
Ulemper	40
Systemverifikasjonstesting – Akseptansetesting	40
Fordeler	41
Ulemper	41
Testing og inspeksjon – en kort dataanalyse	41
Testing og inspeksjon - noen termer	41
Defektkategorier	41
Triggere	42
Testing og inspeksjon i praksis	43
Inspeksjonsdata	44
Testdata	44
Funn	44
Inspeksjon som en sosial prosess	44
Datakilder	45

Testing og kost/nytte	47
Harde kostnader og myke fordeler	48
Skaping av verdi	48
Myke fordeler	48
Informasjon	48
Informasjonens verdi	49
Regret	49
Teststrategi	49
Teststrategiens formål	49
Testfokus	50
Teststrategiens innhold	50
Programvareintegritetsnivå	51
IEEE 1012 – General Software	51
ISO 26262 – Automotive Software	51
IEC 61508 – Safety Critical Software	53
Testobjektiver og -prioritering	53
Valg av testdata	53
Tilfeldig testing	53
Domenepartisjonerings testing	54
Risikobasert testing	54
Brukerprofiltesting	54
Bachs generiske risikoliste	54
Test- og systemnivåer	55
White box Black box Gray box	55
White box testing	55
McCabes syklomatisk kompleksitet	55
Beslutningstabell	56
Løkker	56
Feilmeldinger	57
Videre lesning	57

Black box testing	57
Testing av sanntidssystemer	58
Videre lesning	58
Gray box testing	58
Videre lesning	58
Grey box testing	58
Tilstandsbasert testing	58
Binders tilstander	59
Teststrategier for tilstand	59
Roundtrip path-tre	59
Teststrategi for sniksti	61
Sensor	61
Mutation testing	61
Type 1	61
Type 2 – fuzzing	62
Mutanttestingstrategier	62
Test-prioritering	63
Interessenter (Stakeholders)	63
Risikoidentifikasjon	64
Risikovurdering	64
Hvordan gjøre en risikovurdering	65
Kvalitativ vurdering	65
CORAS-modellen	67
Worry List	68
Risikobasert testing	68
Eksempel på bruk av ATM	69

Kravshåndtering	69
Kriterier for god kravshåndtering	70
Viewpoints, perspektiv og views	70
Ulike typer viewpoints	71
VORD-metoden	71
Kravhåndtering - Viewpoint	74
Fordeler med viewpoint-orientert tilnærming i kravhåndtering . .	74
Non functional requirement	75
Cross cutting (tverrgående) krav:	75
COTS (Commercial off the shelf)	76
Krav for outsourcing	76
Konklusjon	77
COTS-testing	77
Component meta-data	77
Retro-component	78
Bruk av retrospectors	78
Built-In Test(BIT)	78
Testing av komponenter	79
Valg av tester å utføre	79
BIT-arkitektur(Built in test-arkitektur)	79
Ulemper ved bruk av BIT	80
Self TEsting COTS Components (STECC)	80
Assessing COTS	80
Black box-test-reduksjon ved bruk av input-output-analyse	81
Outsourcing, subcontracting and COTS	81
Ansvar	81
Testing og tillit	81
Testing og outsourcing	82
Outsourcing-krav	82
Testing av COTS	83

Testing av intern robusthet	83
Testing av ekstern robusthet	84
Sekvensiell testing	84
Oppsummering	85
Enkle Bayesiske metoder	85
Ansvar	85
Testing og selvtillit	85
Skape tillit til produktet:	86
Skape tillit til prosessen:	86
Testing og outsourcing	86
Krav ved outsourcing.	86
Tillit i komponenten	87
Teste COTS	87
Robusthet	87
Domenetesting	89
Predikater	89
Stibetingelse (path condition)	89
Sti domener	90
Domenefeil	90
ON og OFF points	90
Når skal domenetesting brukes	91
Random testing	91
Chens observasjon	91
Problemet med ART	91
MART prosedyren (fire steg)	92
Testdekning	92
Hva er testdekning?	92
Kategorier	92
Forskjellige typer testdekning	92

Begrenset anvendelighet	93
Definisjoner	93
Bruksområder	93
Akseptansetest	93
Fault seeding	94
Typer domenefeil	94
Closure	94
Shifted-boundary	94
Tilted-boundary	94
Grenseverditesting	95
ON	95
OFF	95
Eksempel	95
Smidige krav gjennom brukerhistorier og -scenarioer	95
Brukerhistorier	96
Hvordan skrive en brukerhistorie	96
User story mapping	97
Fra brukerhistorie til test-case	99
CUCUMBER	99
Eksempel	99
Utfordringer ved bruk av smidige krav	99
Oppsummering	99
Nyttige URLer	100
Advanced Use cases	100
Hvordan finne aktører	100
Hvordan finne use-cases	101
Use case index	102
Tekstlige use-cases	102
Mis-Use cases	103
Hvorfor misuse case?	103
Use-case maps	103

Testdrevet utvikling	105
Testing for greenfield-prosjekter	105
Detaljer eller det “store bildet”	106
Usikkert terreng eller det kjente	106
Highest value vs. the low-hanging fruits	106
Happy paths vs. error situations	106
Essensielle TDD-konsepter	106
Enhetstesting-patterns	108
Ta vare på eller forkaste?	109
Testing for forandring av legacy-kode	110
Prosess	110
TDD og akseptansetesting	110
Valg av user story	110
Skrive tester for historien	111
Automasjon av testene	111
Pro TDD	112
Regresjonstesting	112
Automatisk regresjonstesting	112
Bug fixing	112
Firewall for regresjonstesting	113
Enkle regler for fire wall i et objektorientert system	113
Dependency	113
Ikke-funksjonelle krav	114
Functionality	114
Suitability	114
Accuracy	114
Interoperability	115
Compliance	115
Security	115

Reliability	115
Maturity	115
Fault tolerance	116
Recoverability	116
Availability	116
Usability	116
Understandability	116
Learnability	116
Operability	116
Likeability	116
Efficiency	116
Time behaviour	117
Resource utilisation	117
Maintainability	117
Changeability	117
Stability	117
Testability	117
Portability	117
Adaptability	117
Installability	117
Co-existence	118
Conformance	118
Replaceability	118
Setting av krav	118
MbO - Management by Objectives	118
Scenario testing	119
Noen regler for å skrive gode scenarioer	119
Brukere	119
List opp mulige brukere	119
List opp systemhendelser	120

List opp spesielle hendelser	120
List opp nytte	120
Jobb sammen med bruker	120
Les om denne typen systemer	120
Lag en liksomforretning	120
Risiko	121
Type 1 scenariotesting	121
Type 2 scenariotesting	121
Formell inspeksjon	121
Planlegging	122
Kick-off	122
Individuell kontroll	122
Loggføring	122
Redigering og oppfølging	123
Fordeler og ulemper ved formell inspeksjon	123
Testing	123
Enhetstesting	124
Systemtesting	124
Akseptansetesting	125
Del 2	125
Testing og inspeksjon - En kort dataanalyse	125
Defekt-kategorier	125
Inspeksjonstriggere	125
Kompendie i TDT4242	126
Kontributører	127
For å bygge	127
Ferdige kapitler	127
Gjenstår å gjøre	127

Traceability

From test to requirement : Why do we need this test?

From requirement to test : Where is this requirement tested?

Når en skal endre/justere et krav an en også finne relevante tester.

Challenges

Funksjonelle (Hva et system må gjøre) vs. ikke-funksjonelle(hvor bra systemet utfører funksjonene) “Defined operational capabilities -> Satisfy business needs”

Se foil 2-1 for faktorer som gjør prosjekter vanskelige og grunnen til at prosjekter blir kansellert.

Krav utvikling

Requirements Elicitation (Hvordan man kommer frem til krav) : Prosessen der man finner kravene til et system gjennom kommunikasjon med kundene, system-brukerne og andre stakeholders(interessenter). *Teknikker*: Metodisk ekstraksjon av konkrete krav fra høynivå mål. kvalitetsmålprioritet til krav. (Quality Metrics)

Eksempler på ikke-tilfredstillende krav: Ariane 5: Et dyrt rakettoppskytingsystem. Her antok man visse parametre fra forløperen Ariane 4, som viste seg å ikke stemme. Dette førte til selvdestruksjon og 500mill euro i dass. Airbus: Krav: Revers kan bare brukes når flyet har landet. Oversettelse: Revers kan bare bruke mens hjulene roterer. Implementasjon: Revers kan bare brukes mens hjulene roterer fort nok. Situasjon: Regnstorm og vannplaning, noe som resulterer i et krasj fordi reversen ikke kan benyttes (siden hjulene ikke roterer). Viktig å være nøye i modelleringen.

Formålet med smidig utvikling : utsette så mange avgjørelser så mye som mulig for å øke graden av frihet til ens forståelse er god nok. Øke forståelse ved å ta med folk som kan noe om lignende problemer.

Skille mellom kunde og faktisk bruker. Kjøper for å løse et (*konkret?*) problem.

Two types of requirements statements

Deskriptive statements : Observasjon av den virkelige verden, uavhengig av systemets oppførsel. Eks: Er togdørene åpne er de ikke lukket.

Preskriptive statements : Beskriver ønskelige egenskaper om systemet, som kan stemme avhengig av hvordan systemet oppfører seg. : Slik tilstanden oppfattes av systemet. : Enforced **solely** by the software-to-be. Formulert av fenomen

som deles mellom programvare og miljø. : Programvare forstår dette via sensorer, handler med aktuatorer. Eks : Togdører skal alltid være lukket mens toget beveger seg.

Formulering av systemkrav

Eksempel: Alle togdører skal være lukket mens toget beveger seg. I tillegg til denne preskriptive uttalelsen trenger vi hjelp fra andre komponenter: Togkontrolleren som er ansvarlig for trygg dørkontroll. Passasjerene må ikke åpne dører på en utrygg måte. At dørene fungerer skikkelig (agentene).

Domeneegenskaper

En domeneegenskap: Er en deskriptiv uttalelse om et problem. Uttalelsen burde holde uansett hvordan systemet oppfører seg, og er ofte i relasjon med fysiske lover. Eks: Et tog flytter seg kun hvis farten er !0 (ikke 0).

Målorientering

: Et mål er et mål systemet skal oppnå. Kan være alt fra høynivå strategier til lavnivå tekniske bekymringer. Systemet består av både programvaren og miljøet, interaksjon mellom agenter (enheter, mennesker, programvare). : Mål kan være på forskjellig nivå: : Høynivå: Et mål krever koordinering mellom mange agenter. Eks: Systemets transportkapasitet skal økes med 50% : Krav: Et mål som en enkelt agent har ansvaret for i *systemet*. : Forventning: Et mål som en enkelt agent har ansvaret for i *miljøet til systemet*.

Typer mål

- Soft goals
 - Eks: Passengers should be better informed about flights.
 - Improve
 - Maximise/minimise
 - Increase/reduce
- Behavioural goal
 - System behaviour
 - Agent behaviour
 - Achieve/Cease
 - Maintain/Avoid

02/img/2.png

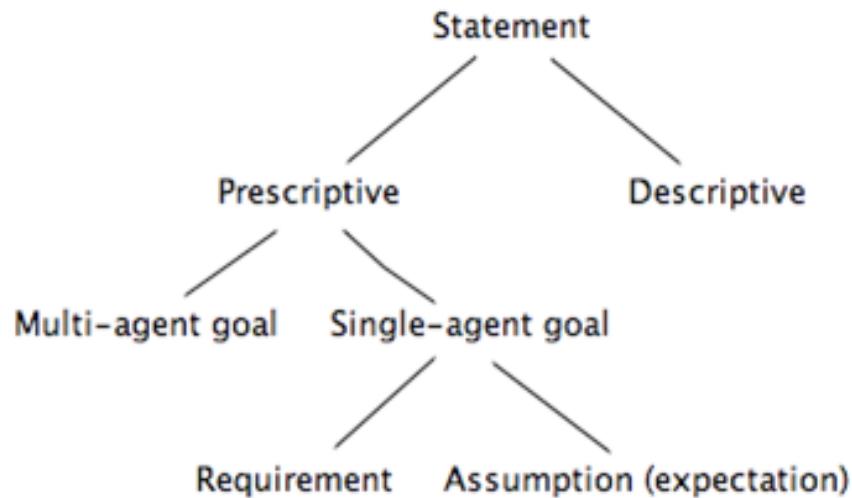


Figure 1: Goal statement typologi

Se foil 2-1 for eksempel på goal types, behavioural goal specialization (Maintain, Achieve, Avoid etc) og goal categorization.

Målkategorisering

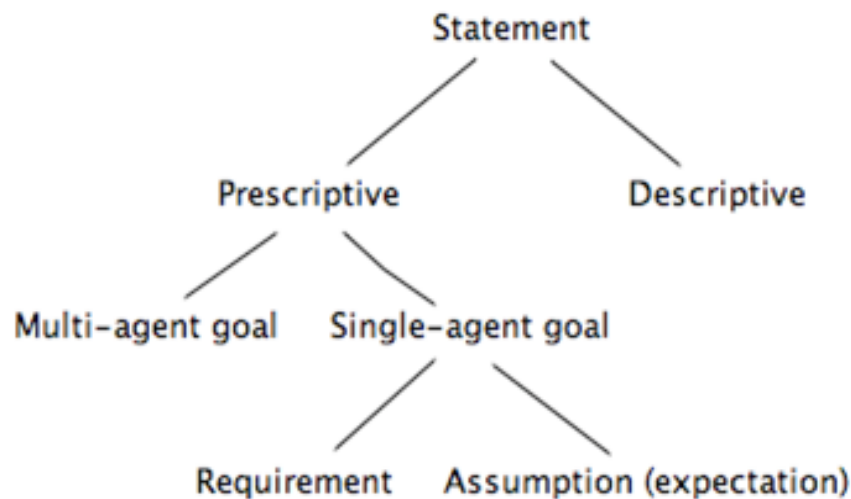
Se foil for “diagram”.

- Funksjonelle mål: Eksempel: Kontrolleren på et tog skal oppdatere togets akselerasjon til den kommanderte med en gang bekreftelse på akselerasjonskommandoen er mottatt. Funksjonelle mål vil tilfredstille agent forespørsler (Satisfaction). Funksjonelle mål vil holde agenter oppdatert på viktige systemtilstander (Information). Funksjonelle mål vil gi riktig respons til en hendelse (Stimulus-response).
- Ikke-funksjonelle mål: Togets fysiske hastighet og kommanderte hastighet skal ikke avvike mer enn X km/t. Ikke-funksjonelle mål uttrykker en kvalitet eller begrensning på en tjeneste. *Accuracy goal*: Ikke-funksjonelle mål krever at tilstanden til variabler programvaren kontrollerer reflekterer de fysiske tilstandene miljøagenten kontrollerer. Altså det at fysisk hastighet og kommandert hastighet ikke skal avvike mer enn X.

- Myke mål (soft goals): Forskjellig fra ikke-funksjonelle mål. Soft goals har ikke noe klart kriterie som bestemmer hvordan de oppfylles. F.eks “En minibank skal være mer brukervennlig”.

Goal refinement (Avoid, maintain, helt ned til agenter)

Se foil 2-1 for refinement tree, eksempler på refinement ++ Mekanisme for å strukturere komplekse spesifikasjoner på ulike nivåer. Et hovedmål kan deles opp i delmål som igjen kan deles opp i krav (på programvaren) og forventninger (på miljøet). Krav på programvaren assosieres kun med én agent. Mål og delmål kan være multiagent. 02/img/2.png



Kvalitetsmålprioritet til krav (Quality metrics)

- Kvalitative mål-krav : Tilnærming til kravspesifikasjon hvor det blir mindre sannsynlig å produsere sporinger som er ukonsistente, støyende, ufullstendige osv. Se 2-1 for fullstendig oversikt over kvalitetsmålprioritet og diagrammer (Ambiguity, inconsistency, forward referencing, opacity, noise, completeness) og hvordan disse regnes ut (formler).

Hvor får man mål fra?

Vi får mål fra: Preliminære analyser av det nåværende systemet. - Systematisk ved å søke på nøkkelord i dokumentasjonen vi har tilgjengelig, f.eks “objective”,

“purpose” osv. - Iterativ redusering av høynivå mål, ved å spørre hvordan og hvorfor? Gir et goal refinement tree. - Tilnærming : KAOS - Måldrevet kravinnhenting.

Oppsummering

- Mål kan defineres på ulike abstraksjonsnivå.
- To måltyper: Behavioral eller soft goals
- Flere kategorier av mål: Funksjonelle - ikke funksjonelle. Goal refinement gir en naturlig mekanisme for å strukturere komplekse spesifikasjoner på ulike nivåer (goal refinement tree).

I dette kurset

Funksjonelle mål og Quality of service

Guidet naturlig språk (GNL) og Boilerplates (BP)

Tre nivåer av krav:

- Uformell - Naturlig språk, fri tekst, ingen regler.
- Semiformell
 - Guided natural language (GNL): fri tekst men tillatte termer er definert i et vokabular.
 - Boilerplates (BP): Strukturert tekst og en ontologi. Vokabular + relasjoner mellom termer.
- Formell - tilstandsdiagrammer etc.

Finne krav:

Finn krav utifra naturlig språk -> Overfør disse kravene til en semi-formell kravmodell -> Avgrens modellen ytterligere for å finne detaljerte krav -> Skap en tidlig designmodell basert på kravene funnet. Felles for alle stegene er å benytte ordbok med felles vokabular; Valider og sjekk kravene opp mot konsistens og fullstendighet.

Mennesker og Maskiner

Grunnet høy kompleksitet i requirements engineering, må vi automatisere så mye som mulig. Mennesker og maskiner har ulike styrker og svakheter. Vi ønsker derfor å analysere krav på en måte som tillatter begge sider å bygge på sine sterke sider.

Maskiner: God på kvantitative data, raske og presise. God på å gjøre konsistente repetisjoner. Ikke god på å behandle variasjoner i skriftlig materiale og mønstergjenkjenning *Mennesker:* God på å behandle variasjoner i skriftlig materiale. Også god på feilkorrigerings.

GNL og BP reduserer variasjon og gir maskinene muligheten til å gjøre det de kan best: Være raske, presise og konsistente. Ved å kombinere mennesker og maskiner og la disse gjøre det de kan best, får vi et bedre resultat. Det endelige målet er å tillate en maskin å assistere utviklere i analyseringen av krav når det kommer til konsistens, fullstendighet og sikkerhet.

GNL

Bruker fritekst med assistanse fra et vokabular. Gir oss krav på en uniform måte for å redusere misforståelser. Altså felles ord alle kan skjønne. Ingen formelle begrensninger, og krever minimal ekspertise. GNL prøver å være en mellomting mellom fritekst og mer formelle krav. Måler kvalitet gjennom korrekthet, konsistens, fullstendighet og redusert variasjon. GNL er en basis for semantisk prosessering for sjekk av krav. Bruker en ordbok som kan være en enkel taksonomi eller mer formell ontologi. *Ontologi* = Tesauros (Domenekonseppter som entiteter, termer og hendelser) + inferensregler (Relasjoner, attributter og aksiomer).

Fange kunnskap gjennom informasjon fra domeneeksperter. Implementer denne kunnskapen gjennom f.eks OWL, og verktøy som Protege. Verifiser at ontologien er korrekt.

Boilerplates

Tekst har fordelen med at den ikke er begrenset av noe, men man trenger en felles forståelse av konseptene som blir brukt for å uttrykke kravene og relasjonene mellom dem - altså hvordan dette presenteres. Boilerplates, eller såkalt “template based textual requirements specification” gir noen begrensninger for å redusere sjansen for ukonsistente uttrykk. *BP*: En basis for sjekking av krav. Enkle å forstå for interessenter i forhold til mer formelle representasjoner. BP kan brukes til både funksjonelle og ikke funksjonelle krav.

Classification of boilerplates	Attributes of boilerplates
Capability	<User>
Capacity (Maximise, Exceed)	<Capability>
Rapidity (Minimise, do not exceed)	<Quantity>
Mode (while, if, for...)	<Time unit>
Sustainability	<Event>
Timelines	<Operational condition>
Operational Constraints	<System function>
Exception	<Action>
	<Entity>
	<State>
	<Effect>

Figure 2: Boilerplates

RE Prosessen: En boilerplate består av faste termer og attributter. Den kan, men må ikke, ha en eller flere modi (modes). 1. Velg en boilerplate eller en sekvens med disse. Seleksjonen er basert på attributtene som må inkluderes og hvordan de er organisert - *faste termer*. 2. Hvis nødvendig, identifiser og inkluder *mode* boilerplates 3. Instansier alle attributter

Typisk boilerplate-eksempel: BP32 The <user> shall be able to <capability>. Attributtene kan være <user> = driver og <capability> = starte ACC systemet. Kravet blir da: The driver shall be able to start the ACC system. "Shall be able to" er en såkalt fast term, flere av disse i foilene. Modi kan være while, if, within, for osv. Se foil 2-3 for alle. Attributter er user, quantity, state, events osv. Foil for alle. Se foil 2-3 for BP eksempler både for funksjonelle og ikke-funksjonelle krav.

Oppsummering

Ved å bruke boilerplates og ontologier får vi: - En felles bruk av termer, og reduserer variasjonen av mulige presentasjoner. Krav som er like skal se like ut. - Redusert variasjon i form og innhold som forenkler bruken av automatiserte og semiautomatiserte verktøy for å sjekke kvalitet (completeness, consistency) og lage test cases.

Sporbarhet av krav

Sporbarhet av krav er av Gotel og Finkelstein definert som:

“[...] the ability to describe and follow the life of a requirement, *in both a forwards and backwards direction*, i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of the phases.”

Gotel og Finkelstein (1994)

En legger med andre ord vekt på at alle de krav som et prosjekt har skal være sporbare *hele veien* helt fra insepasjonen via utvikling og spesifikasjon til utplassering og bruk.

For å oppnå dette er det definert en rekke mål:

- Prosjekthåndtering
 - Status: “Når vil vi være ferdige?” og “Hva vil det koste?”
 - Kvalitet: “Hvor nære er vi å nå våre mål?”
- Håndtering av kvalitetssikring
 - Forbedre kvalitet: “Hva kan vi gjøre bedre?”
- Endringshåndtering
 - Versjonering, dokumentasjon av forandringer (Hvorfor? Hva? Når?)
 - Analyse av endringsinnvirkning
- Gjenbruk
 - Varianter og produktfamilier
 - Krav kan bli målrettet for gjenbruk
- Validering
 - Finne og fjerne konflikter mellom krav
 - Kravs grad av kompletthet
- Verifisering
 - Sikre at alle krav er oppnådd
- Systeminspeksjon
 - Identifisere alternativer og kompromisser
- Sertifisering og revisjon
 - Bevis for at standarder følges.

Utfordringer

Det foreligger en rekke utfordringer som en må ta hensyn til når det kommer til sporing av krav. Blant disse er:

Spor må identifiseres og registreres blant et stort antall hetrogene entiteter, instanser. Det kan være vanskelig å skape betydningsfulle relasjoner i en slik kompleks kontekst.

Spor er videre i konstant forandring og bevegelse, ettersom de vil kunne forandres ved forandringer i krav eller utviklingsartefakter.

Stor variasjon i verktøy, basert på matriser, hyperlenker, tagging, identifisere. Fortsatt må det meste av arbeidet utføres manuelt.

Sporinformasjonen er aldri komplett. Dette grunnet komplekse systemer av sporingsserververvelse og -vedlikehold.

Grunnet mangel på kvalitetsattributter er tillit en stor utfordringer. Det nytter eksempelvis ikke at en vet at 70% av alle sporingskoblinger er nøyaktige om en ikke samtidig vet hvilke koblinger som utgjør disse 70%.

Prosjektet har gjerne en rekke ulike interessenter. Disse har ulikt syn på prosjektet avhengig av sin rolle. Disse vil følgelig ha ulike syn på sporbarheten av krav.

Håndtering av kvalitetssikring handler om å maksimere et produkts *kvalitet*. Denne kvaliteten skal være dokumentert i kravsspesifikasjonen, og spørsmål som derfor spørres i denne sammenhengen er *“Hvor nære er vi kravsspesifikasjonen vår?”* og *“Hva kan vi gjøre bedre?”*.

Endringshåndtering handler om å spore effektene av hver enkelt endring. Dette gjøres ofte via firewall, etc.

Gjenbruk vil peke på de aspekter av en gjenbrukt komponent som behøves adapteres til de nye systemkravene. Til og med kravene i seg selv kan være ting som kan gjenbrukes.

Validering vil bruke sporbarheten til å peke på kravenes kvalitet når det kommer til kompletthet, tvedydighet, forward referencing, opasitet (tetthet). Vil også sikre at hvert av kravene dekkes av minst én del av produktet

Verifikasjon vil sjekke at de begrensninger en har overholdes.

I tillegg kommer det utfordringer relatert til sertifisering/audit, samt testing og vedlikehold.

Metamodeller

Modell : En abstraksjon av virkeligheten.

Metamodell : Modeller av modeller, en videre abstraksjon av virkeligheten som belyser egenskaper ved modellen.

Metamodeller for sporbarhet av krav benyttes ofte som basis for sporbarhetsmetodologier og -rammeverk. Dette for å kunne fastslå og definere hvilke typer artefakter som skal spores, samt definere hvilke typer relasjoner som kan etableres mellom disse artefaktene.

[STAKEHOLDER] -Manages-> [SOURCE] -Documents-> [OBJECT]

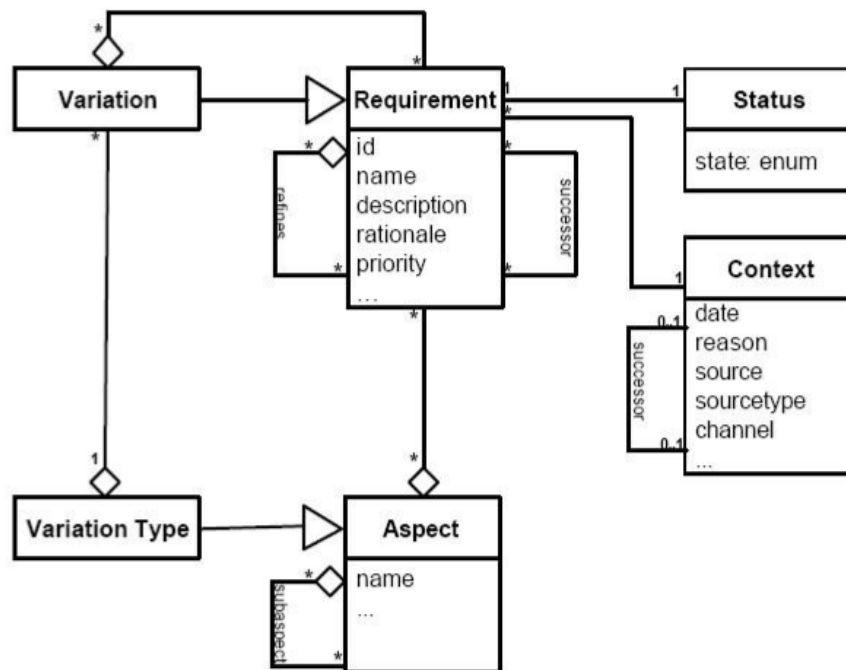
[STAKEHOLDER] -Has role in -> [OBJECT]

[OBJECT] -Traces to-> [OBJECT]

[STAKEHOLDER] -Has role in -> -Traces to-> [OBJECT]

Der skilles ofte mellom high-end- og low-end-sporbarhet.

03/img/3.png



European EMPRESS project: Meta model for requirements traceability

Figure 3: European EMPRESS project: Meta model for requirements traceability

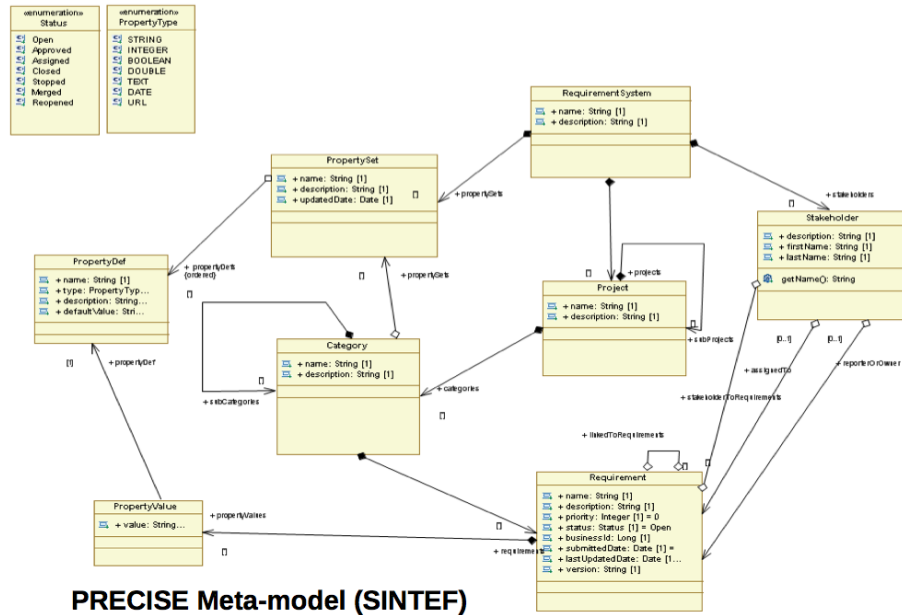


Figure 4: PRECISE Meta-model (SINTEF)

Tilnærminger til sporbarhet

Det er en kritisk oppgave å kunne etablere koblinger, både mellom ulike krav, og mellom krav og andre artefakter. Manuell kobling og vedlikehold av slike koblinger er både dyrt og utsatt for feil. En ønsker derfor å kunne helt eller delvis automatisere denne oppgaven.

Manuell sporingskobling

Manuell sporingskobling er den enkleste formen for sporbarhet. Her benytter en seg av *sporbarhetsmatriser*, enten ved bruk av hypertekst eller regnearkprogram som for eksempel Microsoft Excel for å skape kryssreferanseskjema. Der er i hovedsak to problemer med denne tilnærmingen: over tid vil det å vedlikeholde et stort antall koblinger bli vanskelig; og da koblingene er statiske (mangel på attributter) vil mangel på automasjon av oppgaver være begrenset.

Unik ID Krav	Kilde til krav	SW-krav-spek./ Funk. krav. dok.	Design- spek.	Program- modul	Test- case(r)	Vellykket test- verifikasjon	Modifikasjon av krav	Bemerkning
...

Scenario-drevet sporbarhet

Scenario-drevet sporbarhet er en *testbasert* tilnærming som benyttes for å avdekke relasjoner mellom krav, design og kodeartefakter. Alexander Egyed er anerkjent som forfatteren av denne tilnærmingen.

Trikset benyttet er å observere kjøretids-oppførselen til testscenarioer. Eksempler på verktøy som benytter seg av denne tilnærmingen er IBM Rational PureCoverage. Kjøretidsoppførselen til applikasjonen oversettes til en grafstruktur som benyttes til å indikere fellestrekk mellom entiteter assosiert med hendelsen.

Metoden benytter seg av det den kaller et *footprint* for å oppnå sporbarhet. Dette fotsporet inneholder informasjon om det settet klasser som ble eksekvert når et spesifisert scenario testes, og antallet metoder som ble eksekvert i hver av disse klassene.

Eksempel:

	<i>Test-scenario</i>	<i>Artefakt</i>	<i>Observerte Java-klasser</i>
1	Se liste over filmer	[s3]	[C, J, R, U]
2	Se kontekstuell informasjon om film	[s4, s6] [r2]	[C, E, J, N, R]
3	Velg/spill av film	[s8, s9] [r6]	[A, C, D, F, G, I, J, K, N, O, T, R, U]
4	Trykk stopp-knapp	[s9, s11] [r8]	[A, C, D, E, F, G, I, K, O, T, U]
5	Trykk spill-knapp	[s9, s11] [r9]	[A, C, D, F, G, I, K, N, O, T, R, U]
6	Bytt server	[s5, s7]	[C, R, J, S]
7

Det er imidlertid noen problemer tilknyttet til denne tilnærmingen. Det en kan komme over er at der finnes scenarioer som ikke dekker noen krav, og der kan finnes scenarioer som hører til flere krav. Slike hendelser må markeres i en separat tabell. En benytter seg her av symbolene “F” (fixed) og “P” (probable) for å markere et tilfelle, avhengig av hvor sikre vi er på at en gitt klasse tilhører et gitt scenario.

Utviklingsfotspor

Utviklingsfotspor (development footprint) er en løsning som gjør det mulig å skape sporbarhetsinformasjon, utviklet av (Inah Omoronyia)[<http://www.informatik.uni->

Table 2. Observeable Scenario Footprints

Class↓ Scenario→		A	B	C	D	E	F	G	H
CAboutDlg	0	10					10		
CILLDB	1			3	2	2	2	4	
CILLDBSet	2			4				4	
mailreader	3		3	4	7	1	1	5	1
parsing	4			6				6	
POP3	5		8	10	2	2	2	11	1
CPOP3Dlg	6		4	4	1	4	4	4	
CAPP	7					1	1		
CMainWin	8	3	5	5	3	6	7	5	
CSettingsDlg	9				15				

Figure 5: Eksempel

trier.de/~ley/db/indices/a-tree/o/Omoronyia:Inah.html] et al. Denne metoden krever at hver utvikler alltid identifiserer det krav/use-case som han/hun jobber med på et gitt tidspunkt. En utvikler kan kun jobbe med et enkelt use-case om gangen.

Resultatet av denne metoden vil være lik den som ved bruk av scenario-test-footpring-tabellen. Denne tabellen vil vise hvilke dokumenter, classer og lignende som har blitt aksessert gjennom arbeid med en gitt use-case.

Hovedproblemet med denne tilnærmingen er at det forekommer “falske” adganger. Dette kan skje eksempelvis ved at en utvikler ser på en del av koden som ikke tilhører det use-case som det på et gitt tidspunkt jobbes med. For å motvirke dette kan en supplere informasjonen som genereres med mer informasjon om en aksess’ natur, tidspunkt og person som utførte aksesseringen.

Typen aksesseringstyper gitt i denne metoden er:

- C - Create
- U - Update
- V - View

Ulemper ved bruk av scenario-drevet sporbarhet

Selv om scenario-drevet sporbarhet ofte er semi-automatiserte trenger de allikevel mye tid av systemingeniører som iterativt må identifisere et subsett

03/img/6.png

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
r0			F							F				F				F		F	F
r1	F		F	F		F	F		F	F	F			F	F			F		F	F
r2			F		F					F				F				F			
r3	F		F	F		F	F		F		F				F						F
r4																		F			
r5														F				F			
r6	F		F	F		F	F		F	F	F			F	F			F		F	F
r7	F		F	F		F	F		F		F			F	F			F		F	F
r8	F		F	F		F	F		F		F				F					F	F
r9	F		F	F		F	F		F		F			F	F			F		F	F
s3			P							P								P			P
s9	F		P	F		F	F		F		F				F						
s10			P																		P
s11			P											P				P		P	P
s12			P																	P	P

Figure 6: Eksempel på bruk av developer footprint

testscenarier og hvordan disse relaterer til kravartefakter. Videre er det ikke alltid at krav som per metodene gitt over ikke er relaterte, ikke er relaterte i en annen form (som for eksempel deling av data, implementasjons-pattern).

Jeg ser for meg det er her Singleton-patternet kan være problematisk (TK).

Sporing via tagging

Denne metoden er enkel både å forstå og å implementere. Ulempen er dog at den er sterkt avhengig av menneskelig innblanding. Prinsippet med tagging er at hver enkelt krav gis en *tagg*, enten manuelt eller av et verktøy. Hvert enkelt dokument, kodesnutt, etc. gis også hver sin tagg.

Ulike fremgangsmåter kan benyttes – tagger kan være enten enkeltnivå eller flernivå. Ved bruk av enkeltnivå (eks. R4) får en en enkel sporingsmatrise. Ved bruk av flernivå-tagger (eks. hvor R4.1 og R4.2 er sub-nivå av R4) vil en enkelt kunne gruppere logiske grupperinger av krav, og en kan dermed få mer detaljert sporingsinformasjon.

Sporbarhetens kvalitet vil bero på at vi alltid husker å (korrekt) tagge alle relevante dokumenter og artefakter. Der finnes verktøy som kan kontrollere at alle dokumenter i databasen er tagget, men korrektheten av disse taggene er derimot ikke gitt.

Konklusjon

Det å kunne spore krav er et svært viktig aspekt ved kravshåndtering.

Prosjektets ulike interessenter har ulike behov for sporbarhetsinformasjon.

Sporbarhet kan være komplekst for ikke-trivielle prosjekter.

Sporbarhet-meta-modeller gir en insikt på typen sporbarhetsinformasjon som kreves for et prosjekt.

Der eksisterer flere automatiserte tilnærminger for sporing av krav. Da de ulike automatiserte tilnærmingene har ulike styrker og svakheter vil den beste måten å benytte seg av disse ligge i kombinere disse riktig og dra nytte av synergieffekter dette gir.

Nyttige lenker

- [Requirements traceability \(wikipedia.org\)](https://en.wikipedia.org/wiki/Requirements_traceability)

Kravsspesifikasjon og testing

Testability : The capability of the software product to enable modified software to be validated. (ISO 9126)

Testbarhet tar for seg to hovedområder: hvor enkelt det er å teste en gitt implementasjon; og hvor *test-vennlig* et gitt krav er. Disse to problemene er ikke uavhengige av hverandre og må alltid sees på sammen.

Testabilitet

For å kunne være testbart må et gitt krav være definert på en konkret måte. “*When the ACC system is turned on, the **Active** light on the dashboard shall be turned on.*” er et eksempel på et krav som er tilstrekkelig definert. Dette i motsetning til “*The system shall be easy to use*” som må endres for å gjøres testbart. En metode som brukes for å få et slikt krav mer definert og testbart er å bruke din indre 4-åring og spørre “*Hva mener du med det*” repetetivt til du får et krav som er mer kvantitativt testbart (“*Systemet skal kunne brukes effektiv etter tre dagers bruk*”).

Der er i hovedsak tre måter å kontrollere at en har oppnådd ens mål: *Test-eksekvering* (inkluderer black, white og grey box-testing); gjennom å *gjøre Eksperimenter* og *kodeInspeksjon*.

03/img/1.png

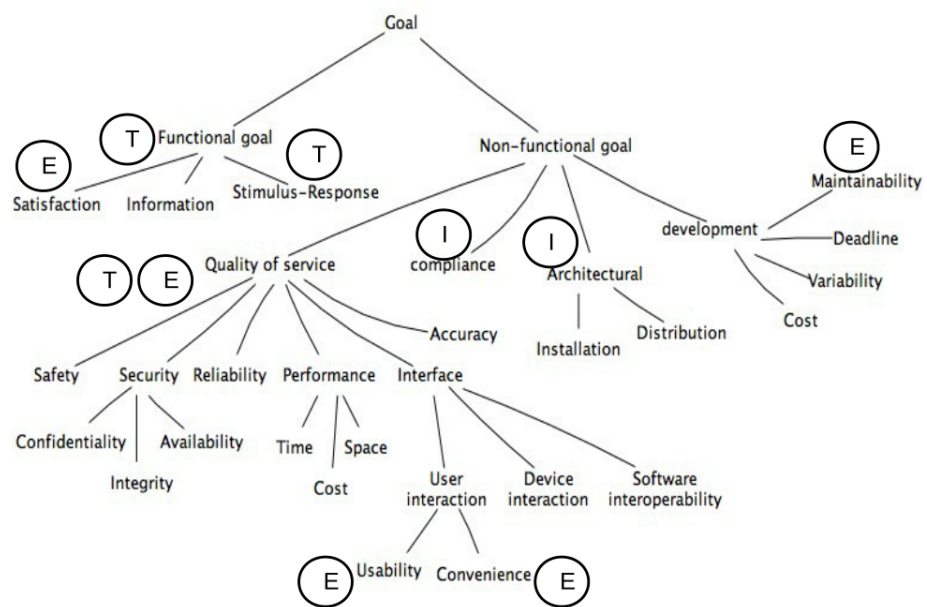


Figure 7: Konkrete krav fra høynivå-mål

Utfordringer

Problemer relatert til:

- Volumet til testene som skal utføres
 - responstid, lagringskapasitet
- Typen hendelse som skal testes
 - feilhåndtering, sikkerhetsmekanismer
- Systemets tilstand før det skal testes
 - uvanlig feiltilstand, en gitt transaksjonshistorikk

Design by Objective

- Tom Gilb

Der er imidlertid to problemer med denne metoden. For det første kan testene som metoden resulterer i være svært ekstensive, og dermed dyre. For det andre krever metoden tilgang til systemets sluttbrukere.

1. What do you mean by ? This will give us either (a) a testable requirement or (b) a set of testable and non-testable sub-requirements.
2. In case (a) we are finished. In case (b) we will repeat question 1 for each non-testable sub-requirements.

Der kan oppstå problemer knyttet til definering av systemets tilstander. Dersom et krav sier at et flys *reverse thrust* skal kunne reverseres *kun når flyet har landet*, er det et definisjonsspørsmål for hva som defineres som *landet*. Dette blant annet på grunn av de ulike sensorer som skal rapportere dette.

I flyeksemplet kan et svar på dette være at et fly har landet når en gitt vekt hviler på landingshjulene. Dette kan imidlertid være problematisk når flyet skal lande i motvind, og dermed genererer ekstra oppdrift, slik at flyet ikke vil kunne *lande* etter de definisjonen.

Først og fremst må kunden vite nøyaktig *hva* han vil og *hvorfor* han vil ha det. Det er ofte mye enklere å teste om kundens oppnår sine mål med applikasjonen enn det er å teste at et system oppfyller et gitt krav. *Hvorfor* er dog sjeldent en del av en kravsspesifikasjon.

Krav til testbarhet

En rekke andre krav ligger til grunn for en god kravsspesifikasjon. Et krav må være:

Korrekt Uten feil.

Komplett Må dekke alle situasjoner, “dersom X så...” og “dersom Y så...”. *Må også dette de tilfeller hvor X og Y ikke inntreffer.* Alt som ikke defineres er per definisjon utenfor kravsspesifikasjonen og skal ikke være tilstede.

Konsistent Et krav kan ikke være i konflikt med andre krav. Dette kan være en utfordring da vi da trenger en komplett oversikt over alle krav. Vi kan dog i de fleste tilfeller klare oss med å kontrollere alle krav som er relatert til samme hendelse, funksjon eller parameter.

Tydelig Konsepter av viktighet her er diagramnotasjon, beskrivelsesspråk og detaljnivå. Dette må tilpasses til den som er ment å lese dokumentene, det være kunde, utviklere eller testere.

Relevant Her vil en søke svaret på to spørsmål: “*Behøver vi virkelig dette kravet?*”; og “*Er kravet så strikt som det burde være?*”. For den som skal teste vil sistnevnte spørsmål være av viktighet, da et krav som er for strikt vil svare til mer arbeid for utvikleren, og et for slakt krav vil svare til mer arbeid for testeren.

Oppnåelig En må spørre seg selv om et krav i det hele tatt er mulig å oppfylle. Testerene kan være med i denne diskusjonen ved å hele tiden spørre om hvordan et gitt krav er ment å testes. Dette vil kunne tvinge alle involverte parter til å gjøre hvert krav bedre definert.

Dersom et krav er vanskelig å implementere vil det ofte og være vanskelig å teste.

Sporbart Hvert enkelt krav må kunne relateres til et eller flere: * program-varekomponenter * prosesssteg

Noen tips til utforming av krav

Unngå modifierende fraser som “etter nødvendighet” og “skal minimum gjøre X”. Dette kan tolkes svært ulikt av ulike oppdragstakere, og en kan i beste fall kun være sikret et absolutt minimum. Vær klar i formuleringen!

Unngå bruk av vage ord som “flagg”, “håndtér”, “spor”. Informasjonssystemer *mottar*, *lagrer*, *kalkulerer*, *rapporterer* og *sender* data, og vi bør helst bruke disse ordene for beskrive hva systemet skal gjøre.

Unngå bruk av udefinerte pronomen, som for eksempel “*det skal vises på skjermen*”. Dette krever at *det* defineres like før, og justeringer og endringer i rekkefølgen på krav kan ødelegge kravet fullstendig. En må og unngå bruk av “alle”, “få”, “andre”, etc. da disse er plassholdere for ikke-navngitte individer og er åpne for tolkning. Bruk *navnet skal vises på skjermen* hver gang.

Unngå bruk av passiv stemme. Ikke definér at “*Z skal kalkuleres*”, men heller “*Systemet skal kalkulere Z*”.

Angående negative krav, vil i prinsippet *alt* som ikke er definert i kravsspesifikasjonen være noe som systemet *ikke* skal gjøre. Bytt derfor ut alle tilfeller hvor spesifikasjonen sier noe om hva systemet ikke skal gjøre til aktive definisjoner på hva systemet *skal* gjøre – fra “*systemet skal ikke godta X*” til “*systemet skal forhindre Y*”.

Alle antakelser og sammenligninger må klart definerte. Det å anta status quo og peke på at systemet skal føre til “15% høyere throughput” eller “systemet skal adressere brukernes fremtidige behov” vil alltid peke mot framtiden og aldri kunne oppnåes.

Implementasjonen

Autonomitet Hvor mange andre systemer må være på plass for å teste et gitt krav? Der er en overhengende fare for at en må planlegge og implementere mer eller mindre komplekse “stubber” som tar for seg manglende systemer.

Observerbarhet Da ikke alle tester produserer utdata vi to spørsmål være viktig å besvare i slike tilfeller: Hvor enkelt er det å observere en testutførings progresjon?; og Hvor enkelt er det å observere et testresultat?

Gjentesteffektivitet (re-test efficiency) Hvor enkelt er det å utføre “test-kontrollér-forandre-gjentest”-sykelen? Dette inkluderer både testens observerbarhet og sporbarhet.

Test-gjenstartbarhet (test restartability) Hvor enkelt er det å stoppe testen midlertidig? Hvor enkelt er det å studere nåværende tilstand og utdata? Hvor enkelt er det å starte testen fra det punktet den ble stoppet? Hvor enkelt er det å starte testen fra start?

Oppsummering

For å sikre at et krav er testbart er det viktig at testere involveres helt fra starten av prosjekter, og at tester er en integrert del av kravet.

Legg merke til at selv om et krav er testbart betyr ikke dette at et krav er *enkelt* å teste.

Requirements Traceability

[the] ability to descibe and follow the life of a requirement, in borth a forwards and backwards direction [...] through periods of on-going refinements [...]. : Gotel and Finkelstein

Iterativ

- Project management
- QA manager
- Change management
- Reuse

Goals

- Validation
- Verification
- System Inspection
- Certification/Audits

High Level Design må være basert på krav. Veldig vanskelig og dyrt å endre underveis. Lurt å starte ut med et høy-nivå arkitekturelt design.

Challenges

Krav forandrer seg, ideer dukker opp

Different stakeholders usage viewpoint, different questions asked by different stakeholders.

- QA Management
- Change management
- Reuse
- Validation
- Verification
- Certification/Audis
- Testing, maintenance

Meta-model

Abstraksjon av en abstraksjon av et fenomen i den virkelige verden.

Tracability

Manuell

Scenario-driven

Semi-automatic

Kjører gjennom scenarioer, noterer hvilke metoder i hvilke klasser som brukes - *footprint*. Benytter seg og av sannsynlighet. Artikkel med eksempel på bruk av sannsynlighet.

Problem: Vanskelig å se på kun et enkelt krav om gangen. Mange krav er koblet sammen, deler hele eller deler av løsning.

Tagging

Hvert krav gis en tagg, enten manuelt eller automatisk. Markerer hver kodebit, dokument etc. med denne taggen.

Mye brukt i industrien.

Enkelt- eller flernivå-tagger. R1, R2, R3 (enkel) eller R4.1, R4.2 (flernivå).

Svakheter:

- Mange tagger, forandres noe må mye oppdateres.
- Kan ikke sjekke at alle tagger er korrekte.

Oppsummering

- Viktig aspekt i kravshåndtering
- Forskjellige interessenter, forskjellig infobehov
- ...

Test vs. inspection

Noen defekter er viktige ettersom de oppstår ofte. De fleste defekter er derimot mindre viktige, da de oppstår sjeldent. Problemet er å skille disse to tilfellene fra hverandre. Hvordan gjør en det mest effektivt og så nøyaktig som mulig?

Testing kan ikke alltid gjøres – til å begynne med har en ikke kode å teste med. Vi må derfor utføre ulike typer inspeksjoner. Dette er en av de største svakhetene med tradisjonell utvikling kontra smidig utvikling.

Fit's List

<i>Area of competence</i>	<i>Man</i>	<i>Machine</i>
Understanding	Good at handling variations in written material	Bad at handling variations in written material
Observe	General observations, multifunctional	Specialized, good at observing quantitative data, bad at pattern recognition
Reasoning	Inductive, slow, imprecise but good at error correction	Deductive, fast, precise but bad error correction
Memory	Innovative, several access mechanisms	Copying, formal access
Information handling	Single channel, less than 10 bits per second	Multi channel, several Megabits per second
Consistency	Unreliable, get tired, depends on learning	Consistent repetition of several actions
Power	Low level, maximum ca. 150 watt	High level over long periods of time
Speed	Slow – seconds	Fast

Et menneske er derfor bra når en behøver en evne til å håndtere varierte situasjoner, være innovativ og induktiv, samt når en vil gjenkjenne mønstre. Mennesker er på den andre siden mindre gode til å gjøre det samme om og om igjen på en konsistent måte. Dette gjør at menneskelig behandling av store data-mengder ikke er å anbefale. Til slikt benytter en seg av maskiner. Menneske og maskin utfyller hverandre ved å utnytte hverandres styrker, ved at maskinen hjelper mennesket å håndtere store mengder informasjon, og mennesket hjelper maskinen ved å gi den innovative inndata.

Generell konsiderasjon

Dokumenter

Design av arkitektur, system, sub-system og komponenter, samt pseudo-kode er eksempler på *dokumenter*, områder hvor tester enten ikke fungerer i det hele tatt, eller fungerer mindre bra. Her bør vi kun benytte oss av inspeksjoner.

Mennesker bruker sin erfaring og kunnskap til å identifisere mulige problemer, maskinen tilbyr støtte via å identifisere relevant informasjon.

Kode

For eksekverbar kode kan en benytte seg av inspeksjon, testing eller en kombinasjon av begge disse, avhengig av kodens størrelse og kompleksitet. Andre viktige faktorer som farger valget av inspeksjon og test er programmeringsspråket og algoritmer som brukes for realiseringen.

En generell regel er at om en står overfor en tilstrekkelig liten kodebase vil en kunne utføre inspeksjon av hele denne basen, for så å designe og utføre tester på bakgrunn av funnene her. I tilfelle kompleks kode, vil en og starte med inspeksjon av koden, men her vil en fokusere på algoritmer og logikk. Da en i slike tilfeller ikke kan teste *hele* kodebasen, vil en måtte utforme kompletthetskriterier for testene, som deretter designes og kjøres.

Kompletthetskriterier : SOMETHING

Inspeksjonsprosesser

For alle inspeksjonstyper - GIGO. Det kan og være en god idé å involvere kunderepresentanter.

Walkthrough

Intern. Bukes for tidlig beslutningstaking.

1. Skape en grov skisse av løsningen
 - Arkitektur, algoritme, etc.
2. Presentering
 - Forklare skissen til oppmøtte
3. Registrering av feedback og komme med forslag til forbedringer.

Fordeler Fordeler med walkthrough av koden inkluderer at den er både enkel og billig. Dette muliggjør at en kan samle ideer på et tidlig stadie i utviklingen.

Ulemper Ulemper er at da prosessen er uformell er der ingen forpliktelser hos deltakerne. Prosessen skaper og mange løse og irrelevante ideer.

Uformell inspeksjon

Intern.

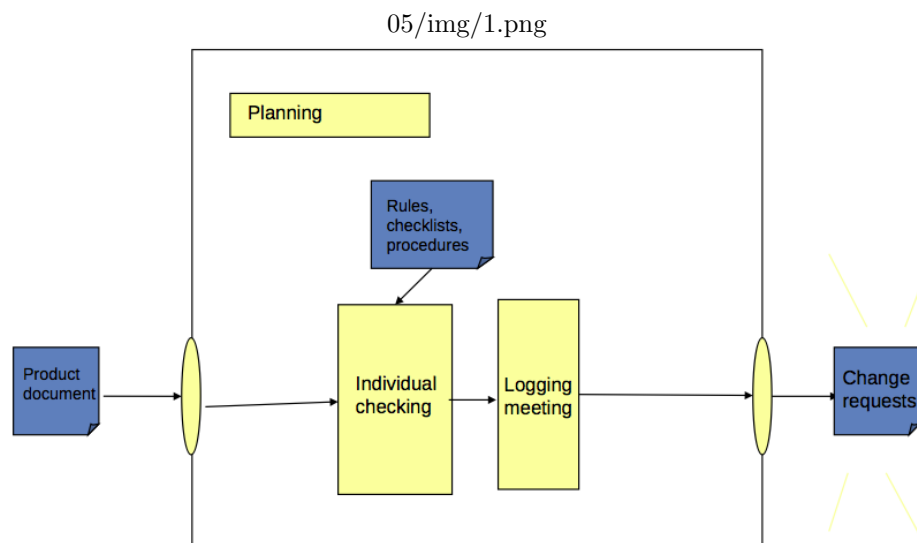


Figure 8: Uformell inspeksjon

Fordeler

- Enkel og billig å gjennomføre

- Kan utføres i alle av utviklingens stadier
- Har som regel en god kostnad/nytte-ratio
- Krever kun et minimumsnivå av planlegging

Ulemper Som ved bruk av [walkthrough](#) vil det ikke være en formell forpliktelse blant deltakerne. En plukker heller ikke opp potensielle prosessforbedringer.

Formell inspeksjon

Formell inspeksjon anbefales å være en del av den endelige akseptansetestingen av viktige dokumenter.

Som utredet av [T. Gilb](#) og [D. Graham](#)

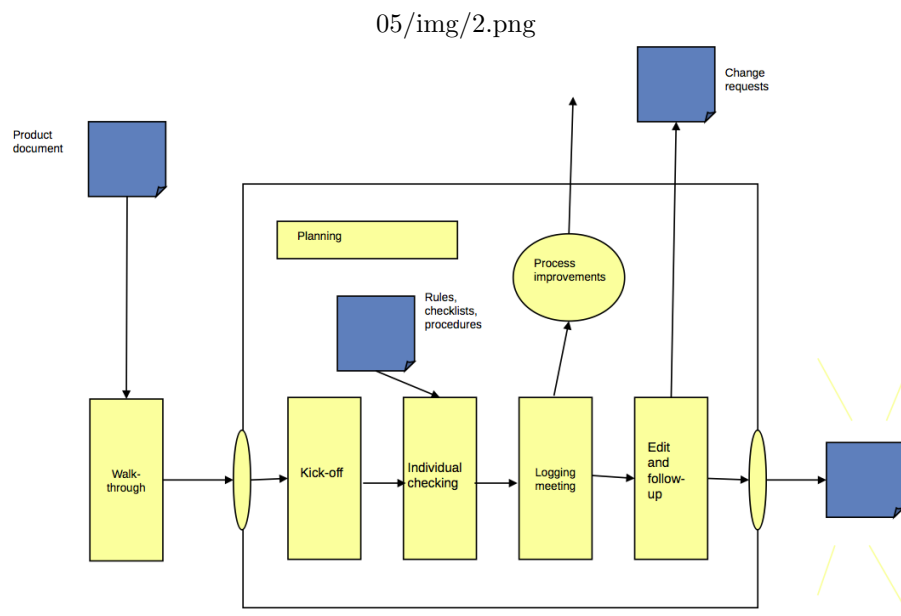


Figure 9: Formell inspeksjon

	<i>Range (%)</i>	<i>Typisk verdi (%)</i>
Planning	3-5	4
Kick-off	4-7	6
Individual checking	20-30	25
Logging meeting

Distribusjon av resusser (TODO: Resten av slide 24 har falt bort)

Initiering av inspeksjonsprosessen Inspeksjonsprosessen starter ved at forfatter sender en inspeksjonsforespørsel til QA-ansvarlig. Dersom dokumentet defineres å være klar for inspeksjon vil så QA-ansvarlig utnevne en inspeksjonsleder.

Planlegging Viktige momenter som planlegging skal besvare er *hvem som skal delta i inspeksjonen*. De som skal delta bør ha interesse, tid og den kunnskapen som behøves for å sette seg inn i inspeksjonen.

Kick-off

- Distribuering av nødvendige dokumenter
- Tildeling av inspeksjonsspesifikke roller og jobber
- Sette mål for resursser, deadlines, etc.

Individuell kontroll Den individuelle kontrollen er inspeksjonens hovedaktivitet. Her vil hver enkelt deltaker lese gjennom dokumentet for å se etter *potensielle feil* – det vil si inkonistens i forhold til krav eller total applikasjonsfølelse, eller *mangel på etterlevelse* i forhold til selskapet standarder eller godt håndtverk.

Loggføring av møtet

- Loggføring av *saker* allerede funnet av inspeksjonens deltakere.
- Finne *nye* saker, basert på diskusjoner og ny informasjon som dukker opp under loggføringen av møtet.
- Identifisering av mulige forbedringer av inspeksjons- eller utviklingsprosessen.

Forbedring av produktet Etter endt inspeksjon vil dokumentets forfatter motta all loggføring av møtet. Alle oppføringer og saker som har kommet fram vil så kategoriseres som én av følgende:

- Feil i forfatters dokument
 - Gjør de nødvendige korrigeringer
- Feil i annens dokument
 - Informere dokumentets eier om mangelen
- Misforståelse innad i inspeksjonsteamet
 - Forbedre dokumentet for å hindre senere misforståelser

Kontroll av forandringer Det er inspeksjonsleders ansvar at samtlige saker i loggen avhendes på en tilfredsstillende måte.

Fordeler

- Kan brukes til å formelt akseptere dokumenter
- Inkluderer prosessforbedringer

Ulemper

- Dyr og tidskonsumerende
- Behøver ekstensiv planlegging for å være vellykket

Testprosesser

Enhetstester

Målet med enhetstester er å verifisere at koden fungerer som foreskrevet. Dette gjøres av utvikler én eller flere ganger i løpet av utviklingen. Enhetstesting er en uformell testmetode som utføres ved at en først implementerer (en del av) en komponent. En definerer så en eller flere tester som aktiviserer denne koden. Til sist vil en kontrollere resultatet mot de forventninger og nåværende forståelse av komponenten.

Fordeler

- Enkel måte for kontrollere at koden fungerer
- Kan benyttes sammen med programmering på en iterativ måte

Ulemper

- Vil kun teste en utviklers forståelse av spesifikasjonen.
- Kan behøve stubber eller drivere for å kunne teste.

Funksjonell verifikasjonstesting – Systemtesting

1. Basert på kravsspesifikasjoner, identifiser
 - Tester for hvert enkelt krav, inklusiv feilhåndtering
 - Startstadie, forventet resultat og sluttstadie
2. Identifisere avhengigheter mellom tester
3. Identifisere akseptansekriterier for testsuite
4. Kjøre tester og kontrollere resultatet opp mot
 - akseptansekriterier for hvert enkelt test
 - akseptansekriterier for hele testsuiten sett under ett

Fordeler

Tester systemets oppførsel opp mot kundens krav.

Ulemper

Systemtesting er en black box-test. Dersom man finner en feil vil en behøve ekstensiv debugging for å lokalisere kilden til feilen.

Systemverifikasjonstesting – Akseptansetesting

1. Kjør testene på nytt på kundens domene
2. Bruk systemet for å utføre et sett realistiske oppgaver med ekte data.
3. Forsøk på å bryte systemet, enten ved bruk av data som ikke er ment for bruk med store mengder data eller med ulovlig input.

Fordeler

- Skaper tillit til produktets evne til å være nyttig for kunde.
- Viser et systems evne til å operere i kundens miljø.

Ulemper

Kan tvinge systemet til å håndtere data det ikke var designet for, og dermed skape et ugunstig bilde av systemet.

Testing og inspeksjon – en kort dataanalyse

Testing og inspeksjon - noen termer

Defektkategorier

1. **Tilordning (Assignment)** Verdier blir tilordnet feil, eller ikke i det hele tatt.
2. **Sjekk (Checking)** Defekter som skyldes manglende eller feil validering av parametre eller data i conditional statements.
3. **Algoritme** Effektivitets- eller korrekthetsproblemer som kan fikses ved å bytte ut en algoritme eller datastruktur uten å endre design. Kan inneholde flere “Assignment”- og “Checking”-korreksjoner.
4. **Timing/serialisering** Nødvendig serialisering for en delt ressurs mangler, feil ressurs er serialisert, eller så er feil serialiseringsteknikk brukt.
5. **Grensesnitt** Kommunikasjonsproblemer mellom moduler, componenter etc.
6. **Funksjon** Krever designendring, da defekten påvirker enten brukergrensesnitt, grensesnitt for andre produkter eller grensesnitt mot maskinvaren.
7. **Build/Package/Merge** Problemer med build-prosessen, i biblioteker eller i versjonskontrollen.
8. **Dokumentasjon** Problemer med brukermanual, installasjonsmanual eller kodekommentarer. Må ikke forveksles med funksjons- eller grensesnittfeil som beskrives i dokumentasjonen.

Triggere

En trigger er en hendelse som gjør at man oppdager feil, gjør det mulig å evaluere effektiviteten av inspeksjoner og test scenarier. Det brukes forskjellige typer triggere for inspeksjon og for testing. I tillegg skilles det mellom triggere som brukes til white-box- og black-box-testing.

Inspeksjonstriggere En inspeksjonstrigger er en trigger som hjelper inspektører til å finne defekter i kode eller design dokumenter. Følgende triggere brukes for å finne defekter:

1. **Overenstemmelse med design** Inspektørene finner defekter ved å sammenligne designelementer eller deler av koden med kravspesifikasjonen.
2. **Forstå detaljer** Inspektørene finner feil ved å undersøke strukturen til og/eller hvordan en komponent opererer i detalj. Denne triggeren kan deles ytterligere opp:
 - a. **Operasjonell semantikk** Operasjonell semantikk beskriver hvilke sekvenser som blir utført i koden, altså meningen med programmet. Inspektøren har her den operasjonelle semantikken i bakhodet når defekten blir oppdaget.
 - b. **Bivirkninger** Når inspektøren undersøker dokumentert eller implementert kode, oppdages det flere defekter som en bivirkning.
 - c. **Samtidighet** For å kontrollere en delt ressurs trenger man å serialisere prosesser, og inspektøren kan oppdage defekter ved å inspisere denne prosessen.
3. **Bakoverkompatibilitet** Inspektøren finner inkompatibilitet mellom funksjonalitet beskrevet i dokumentasjonen eller koden, og tidligere versjoner av samme produkt.
4. **Kompatibilitet med andre tjenester** Inspektøren oppdager inkompatibilitet mellom funksjonalitet beskrevet i dokumentasjonen eller koden og andre systemer eller tjenester produktet må snakke med.
5. **Udokumentert oppførsel** Inspektøren bruker sin erfaring og kunnskap til å forutse udokumentert oppførsel av systemet, og finner defekter på denne måten.
6. **Konsistens/Fullstendighet** Defekten kommer til syne pga inkonsistent eller ufullstendig dokumentasjon eller kode.
7. **Språkavhengighet** Utvikleren finner en defekt under nærmere undersøkelse av språkspesifikke detaljer av implementasjonen av en komponent eller en funksjon.

Testtriggere En trigger omfatter hensikten bak det å lage en test case, og test case finner defekter når den kjøres. Valget av triggere avhenger av om det er white-box- eller black-box-testing.

White-box-triggere

1. **Enkel stidekning (Simple Path Coverage)** Test case skrives for å undersøke visse branches i koden, på bakgrunn av kunnskap om disse. Hver branch testes én gang.
2. **Kombinatorisk stidekning (Combinational Path Coverage)** Nesten det samme som enkel stidekning, forskjellen er at hver branch testes mer enn én gang. Hver branch testes med flere forskjellige forhold.
3. **Bivirkninger** Defekten viser seg ved uforutsett oppførsel, som ikke ble spesifikt testet.

Black-box-triggere

1. **Dekningstest** En test case av en enkel kodedel ved bruk av en enkel input finner defekten.
2. **Sekvensieringstest** Test casen som fant defekten kjørte sekvensielt, hvor to eller flere kodedeler (kan) kjøres uavhengig av hverandre.
3. **Interaksjonstest** Test casen som fant defekten startet en interaksjon mellom to eller flere kodedeler, hvor hver kodedel kan kjøres uavhengig av de andre. Interaksjonen mellom delene er mer komplisert enn å kjøre dem sekvensielt.
4. **Variasjonstest** Defekten blir oppdaget ved at man tester samme kode, men med forskjellige variasjoner av input.
5. **Bivirkninger** Defekten viser seg ved uforutsett oppførsel, som ikke ble spesifikt testet.

Testing og inspeksjon i praksis

Dette kapitlet presenterer funn ved testing og inspeksjon i praksis, basert på visse inspeksjons- og testdata.

Inspeksjonsdata

Vi ser på inspeksjonsdata fra tre forskjellige utviklingsaktiviteter:

- Høynivå design: arkitekturelt design
- Lavnivå design: design av subsystemer, komponenter, modler og data-modeller
- Implementasjon: realisering, skrive kode

Disse utgjør venstresiden av V-modellen.

Testdata

Vi ser også på testdata fra tre forskjellige utviklingsaktiviteter:

- Unit testing: Tester små enheter som metoder eller klasser
- Verifikasjonstesting av funksjoner: Funksjonell testing av en komponent, et system eller et subsystem
- Verifikasjonstesting av system: tester systemet under ett, inkludert maskinvare og brukere

Disse utgjør høyresiden av V-modellen.

Funn

Det viser seg at Paretos regel (om at 80% av effekten kommer fra 20% av årsakene) gjelder i det fleste tilfeller for både defektkategoriene og triggerne. Det vil si at omtrent 20% av defekttriggerne utløser 80% av defektene.

De vanligste defektene er relatert til enten dokumentasjon eller funksjon. Grensesnittdefekter er den eneste defektkategorien som kom på topp tre av de vanligste defektene både etter testing og etter inspeksjon. Samtidig er “bivirkninger” den eneste defekttriggeren som finnes blandt både testing- og inspeksjonsresultatene.

Inspeksjon som en sosial prosess

Man må ta høyde for at inspeksjoner også inneholder en menneskelig faktor, og ikke bare tekniske detaljer. Det er viktig å tenke på hvordan personer:

- Samhandler
- Samarbeider

Datakilder

Det har blitt gjennomført to eksperimenter relatert til de sosiale prosessene i inspeksjoner:

- UNSW - tre eksperimenter med 200 studenter, hvor fokuset var gevinst vs tap ved prosessen
- NTNU - to eksperimenter:
 - NTNU 1, 20 studenter fordelt på grupper hvor de bruker sjekklistor
 - NTNU 2, 40 studenter

Data fra UNSW-eksperimentet Programmene som ble inspisert bestod av:

- 150 linjer lang kode med 19 defekter
 - 350 linjer lang kode med 38 defekter
1. Hver student inspiserte koden individuelt og leverte inn hver sin rapport
 2. Studentene ble tilfeldig plassert på grupper med tre personer, til sammen 40 grupper
 3. Hver gruppe inspiserte koden sammen og leverte en rapport

For å forstå UNSW-eksperimentet, må man forstå to termer:

- Nominell gruppe - en gruppe personer som senere vil delta i en ekte gruppe, men som foreløpig jobber alene
- Ekte gruppe - en gruppe personer som kommuniserer direkte med hverandre, og samarbeider

Fra dette eksperimentet fant man at gruppen som enhet ikke fant 7 av defektene som enkeltpersonene fant. Samtidig fant de 5 defekter som enkeltpersonene ikke fant. Det vil si at prosessgevinst er 5 defekter, mens prosesstap er 7 defekter. Dette viser bare at det er store sjanser for at man ikke finner visse defekter ved å samarbeide i grupper, men samtidig er det store muligheter for å finne flere andre defekter.

Etter å ha analysert alle dataene fra dette eksperimentet, fant man at sannsynligheten for prosesstap (dvs. de defektene man taper/mister ved å bruke grupper) er 53%. Sannsynligheten for prosessgevinst (dvs. de ekstra defektene man finner ved å bruke grupper) var 30%. Man kan derfor konkludere med at det

å bruke grupper ikke har så stor nytteverdi, og at man like gjerne kan droppe denne delen.

Det var en 10% sannsynlighet for at studentene rapporterte om defekter i rapporten, til tross for at ingen fant disse defektene. De defektene de fant var det 80-95% sannsynlighet for at de nevnte i rapportene.

Grunnen til at defekter som blir funnet på egenhånd blir forkastet i gruppe rapporten, kan være at majoriteten i gruppa bestemmer. Hvis et gruppemedlem finner en feil som ingen andre finner, kan det hende det gruppemedlemmet gir etter for gruppepress.

- Ved prosessstap finner man mange nye defekter, men fjerner også mange.
- Ved prosessgevinst finner man mange nye defekter, men fjerner bare none få.
- Ved prosess-stabilitet finner mange defekter og fjerner omtrent like mange

Man kan dele gruppene opp i to deler:

- **Prosessgevinst** Alles bidrag teller, og man finner mange nye defekter
- **Prosesstap** Gruppepress, finner få nye defekter

Data fra NTNU1-eksperimentet Det var som sagt 20 studenter med i eksperimentet, som innsiserte et program på 130 kodelinjer. Programmet inneholdt 13 defekter.

1. Gruppene bestod av to, tre eller fem studenter
2. Halve gruppe brukte en skreddersydd sjekkliste
3. Hver gruppe innsiserte koden og leverte en rapport

Gruppestørrelsen og bruken av sjekklister var hovedfokuset under eksperimentet. Dessuten ble effekten av kombinasjonen av de to også undersøkt.

Resultatene viser at de største gruppene fant fire flere defekter enn de minste gruppene. Bruk av sjekkliste ga ikke utslag på hvor mange defekter gruppa fant. De minste gruppene som brukte sjekklister fant to flere defekter enn de største gruppene som brukte defekter. Standardavviket gjør at man kan utelukke alt annet enn gruppestørrelsen.

Data fra NTNU2-eksperimentet 40 studenter som innsiserte et program på 130 kodelinjer. Programmet inneholdt 12 defekter.

1. 20 PhD-studenter og 20 ingeniørstudenter på tredjeåret
2. Hver student innsiserte koden individuelt og leverte inn rapport

Defektene gikk under følgende kategorier:

- **Feil kode** - f.eks. feil parameter
- **Ekstra kode** - f.eks. ubrukte variabler
- **Manglende kode** - f.eks. ingen exception handling

De med minst erfaring og som nylig har kodet er bedre til å finne manglende kode, mens de med litt mer ingeniørutdanning var flinkere til å finne overflødig kode. Erfaring hadde ingen innvirkning på hvor flinke studentene var på å finne feil kode.

Testing og kost/nytte

For de fleste programvaresystemer vil antallet mulige input være store. På et tidspunkt vil en nå et punkt hvor en ny test vil koste så mye å implementere at nytten en får ut av den ikke er stor nok for at den er verd det. En bør med andre ord stoppe når *kostnaden er større enn forventet nytte* av neste test. Dette er en subjektiv prosess.

Som et minimum må en inkludere kostnader rundt det å utvikle og kjøre tester, samt kostnader tilkoblet til det å korrigere feil basert på resultatene. I tillegg er det mulig å inkludere kostnader rundt det å engasjere mennesker som kunne ha vært engasjert i andre, mer lønnsomme aktiviteter. Misfornøyde kunder kunder kan og representere en kostnad.

Når det kommer til nytte vil vi som minimum inkludere nytten det er å kunne avdekke flere feil før en slipper programvaren (det er mye dyrere å korrigere feil, jo lenger en venter). I tillegg kan en derivere nytte fra å ha et bedre rykte i markedet, å ha mer tilfredse kunder, samt å bruke eventuell ledig personellkapasiteten bedre.

Når kostnader og nytte har blitt identifisert kan en basere beslutninger på regnestykket *NYTTE - KOSTNADER*.

$LEVERAGE = (BENEFIT-COST) / COST$

Harde kostnader og myke fordeler

Et av de største problemene en har med å beregne kost/nytte er at kostnader ofte kommer med én gang, nytte får en resultatet av senere. Det er med andre ord lettere å identifisere og tallfeste kostnader enn nytte.

Det er dog ikke noens selskaps eneste mål å spare penger. Mål som er mer verdifulle er å øke profitt, markedsandeler, markedsrykte.

Skaping av verdi

Kreativitet. Det å flytte mennesker over fra det å teste over til å skape er å skape verdi.

Myke fordeler

For å gi myke fordeler verdi må man:

- identifisere viktige bedriftsmål og de faktorer som bidrar til disse målene
- mappe faktorene til hendelser relatert til testing
- spørre selskapet hva de er villige til å betale for å
 - øke en faktor (eks. markedsandel)
 - unngå økning av en faktor (eks. klager)

Disse spørsmål er må som regel ledelsen gi svar på. Vanskelig å gi konkrete svar, men man får som regel et intervall (10-30%).

Informasjon

Parametre av spesiell betydning:

$P(\text{feil})$ – sannsynligheten for å ta en feil beslutning $C(\text{feil})$ – kostnaden ved å ta en feil beslutning

Det vi må gjøre er å vurdere hvilken informasjon en kan få av at en testcase feiler eller utføres korrekt. Om kostnaden tilknyttet til å få denne informasjonen er for høy (informasjonen er av for liten verdi) vil det heller ikke være vits å kjøre testen.

$$\text{Risiko} = P(\text{feil}) * C(\text{feil})$$

Informasjonens verdi

Uten noen informasjon vil sannsynligheten til å ta en feil beslutning være 50%. Denne sannsynligheten vil en kunne senke ved å samle mer informasjon. I vårt tilfelle vil dette bety å kjøre flere tester. Dette koster selvfølgelig penger, noe som må tas med i betraktning.

Regret

Vurdert verdi av noe vi angrer på at vi ikke gjorde. I en kost/nytte-analyse er dette *muligheten vi aldri tok*.

If you do not grab this opportunity, how much would you be willing to pay to have it another time?

$$\text{Risk} = P(\text{wrong}) * \text{Cost}(\text{wrong})$$

$$\text{Total benefit} = \text{Regret} + \text{Benefit} \quad \text{Total cost} = \text{Risk} + \text{Cost}$$

$$\text{Leverage} = (\text{Total benefit} - \text{Total cost}) / \text{Total cost}$$

For å redusere problemets kompleksitet vil en kunne anta at kostadene til en feil beslutning er konstant og med mer investering i mer kunnskap vil sannsynligheten til feil ($P(\text{wrong})$) synke eksponensielt, samt at nytteverdi er tidsbegrenset.

Teststrategi

En teststrategis formål er å hjelpe system- og programvaretestere, samt “Test-and-Evaluation”-personale med å bestemme overordnet teststrategi når en skal utvikle eller modifisere programvareintensive system. En vil og assistere prosjektets interessenter – kunder og seniorledelse – med å godkjenne teststrategien. Testere og system- og programvareanalyse vil få hjelp til å fastslå målsetninger med testen, kvalifikasjonskrav, samt verifikasjon- og valideringskriterier.

Nøkkelkonsepter: Teststrategiens formål. Testfokus. Teststrategiens innhold. Programvareintegritetsnivå. Testmålsetninger og -prioriteringer.

Teststrategiens formål

En teststrategi har flere ulike formål. En ønsker å oppnå konsensus angående testmål og -målsetninger fra prosjektets interessenter. Dette inkluderer ledelse, utviklere, testere, kunder og sluttbrukere. En vil og kunne bedre håndtere forventninger til prosjektet helt fra starten av, og en vil bedre kunne sikre at

en til enhver tid er på riktig vei. Videre vil en og med en god teststrategi kunne identifisere typen tester som skal utføres på ethvert testnivå.

Når en så vil skrive denne strategien vil er det viktig at en er påpasselig med at uansett hva en gjør

En teststrategi vil i de aller fleste tilfeller oppstå av seg selv i løpet av et prosjekt. En kan derfor like gjerne bedre definere og dokumentere en skikkelig teststrategi for å oppnå best mulig resultat. En dokumentert strategi er den mest effektive måten å oppnå en tidlig enighet om mål og målsetninger.

I en teststrategi må en huske på å adressere menneskelige faktorer som eksempelvis brukervennlighet, og interoperabilitet mellom relevante systemer (bortsett fra i de tilfeller hvor systemet som skal implementeres er et enkeltstående system).

Testfokus

Ens fokus i en teststrategi vil variere avhengig av de interessenter en på et gitt tidspunkt vurderer. Relevante interessenter kan være *brukere*, *analytikere*, *designere* og *programmerere*, og alle har ulike ønsker og behov til innhold og ordbruk i strategidokumentet. Det er derfor viktig å allerede helt i starten definerer den relevante interessenten for et gitt strategidokument. Dette før en definerer de tester som skal utføres.

Teststrategiens innhold

Nedenfor er en rekke *eksempler* til hva som kan være i en teststrategi. Avhengig av teststrategiens fokus vil valget av innhold følgelig variere. Bruk kun det som er nødvendig.

- prosjektplan, -risiko og -aktiviteter
- relevante forskrifter, avhengig av applikasjonens domene
- påkrevde prosesser og standarder
- støttende retningslinjer
- interessenter og deres målsetninger
- nødvendige ressurser
- testnivåer og -faser
- testmiljø
- fullføringskriterier for hver enkelt fase
- påkrevd dokumentasjon og ettersynsmetode for hvert enkelt dokument

Programvareintegritetsnivå

Der er flere ulike måter å definere programvareintegritetsnivå. Valget av integritetsnivå vil påvirke måten vi tester på.

IEEE 1012 – General Software

06/img/tabell1.png

4	Høy	Enkelte funksjoner påvirker kritisk systemytelse.
3	Stor	Enkelte funksjoner påvirker viktig systemytelse.
2	Moderat	Enkelte funksjoner påvirker systemytelsen, men omveier kan implementeres for å kompensere for dette.
1	Lav	Enkelte funksjoner påvirker systemytelsen, men skaper kun ubeleiligheter.

Figure 10:

06/img/tabell2.png

V&V activity	Development requirements level				Design level				Implementation level				Test level			
SW Integrity Level	4	3	2	1	4	3	2	1	4	3	2	1	4	3	2	1
Acceptance test execution													X	X	X	
Acceptance test plan	X	X	X													
Interface analysis	X	X	X		X	X	X		X	X	X					
Management and review support	X	X			X	X			X	X			X	X		
Management review of V&V	X	X	X		X	X	X	X	X	X	X	X				

Figure 11:

V&V Activities

ISO 26262 – Automotive Software

Her benyttes en ASIL-skala – A, B, C, D – som et resultat av å kombinere tre ulike faktorer. S (severity – hvor alvorlig en hendelse er), E (probability – sannsynlighet for hendelsen skal inntreffe) og C (controllability – hvor enkelt det er å kontrollere hendelsen dersom den inntreffer).

Hvordan finne ASIL-nivå

06/img/tabell3.png

Severity	Probability	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Figure 12:

IEC 61508 – Safety Critical Software

06/img/tabell4.png

Safety integrity level	High demand or continuous mode of operation
4	10^{-9} to 10^{-8}
3	10^{-8} to 10^{-7}
2	10^{-7} to 10^{-6}
1	10^{-6} to 10^{-5}

Figure 13:

$PFD_{avg} = F_t / F_{np}$. Tabellen over gir SIL-nivå, sammen med denne verdien.

Testobjektiver og -prioritering

Kun i helt spesielle tilfeller kan en teste absolutt alle inndata. En må derfor vite det overordnede målet med testingen, målsetningen med hvert enkelt testcase, og teknikker for design av testcaser for å oppnå våre mål på en systematisk måte.

Testobjektivene er vår kravsspesifikasjon for testing.

Valg av testdata

Valg av data å teste med er et av de viktigste valgene vi gjør når vi utformer en teststrategi. Fem populære metoder er pensum i faget.

Tilfeldig testing

I denne metoden benytter en seg av to handlinger.

1. Definér alle input-parametre (integer, real, string)
2. Benytt en tilfeldig test/nummer-generator for å produsere inndata til SUT (wtf is SUT?)

Hovedproblemet med denne metoden er mangelen på et orakel å teste resultatene opp mot. En må med andre ord kontrollere resultatene manuelt.

Tilfeldig testing benyttes som regel i forbindelse med robusthetstesting, hvor en vil finne ut om et gitt system kan håndtere (store mengder) ulike data uten å brette.

Domenepartisjonerings testing

Domene : Et sett input-verdier hvor programmet utfører den samme utregningen for hvert nummer i settet. Vi ønsker å definere domene slik at programmet utfører ulike utregninger på nærliggende domener.

Et program sies å ha en domenefeil dersom programmet velger feil domene, input-klassifikasjonen feiler.

Risikobasert testing

Ved bruk av risikobasert testing er det to aktiviteter som en vil gå gjennom.

1. Identifisere risiko eller kostnader ved å *ikke* levere en gitt funksjonalitet.
2. Bruke denne informasjonen til å prioritere tester.

Brukerprofiltesting

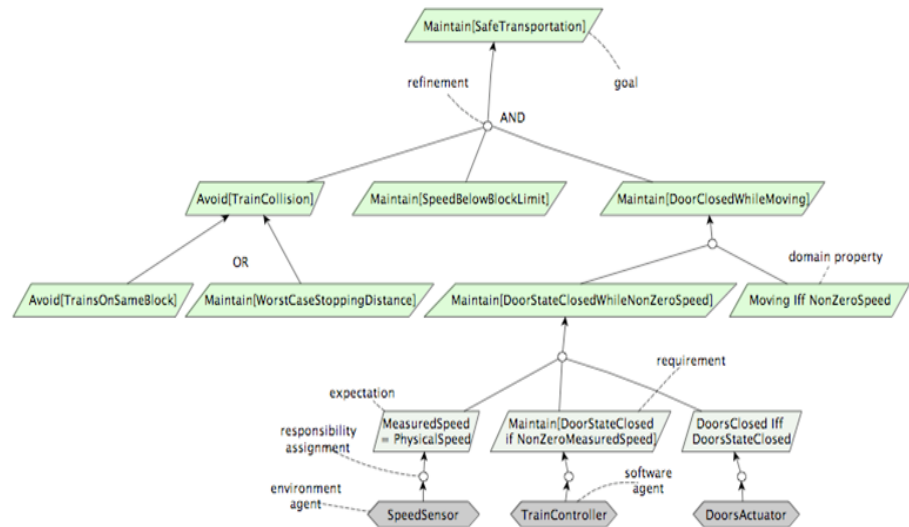
Hovedideen med brukerprofiltesting (user profile testing) er å generere tester som gjenspeiler brukerens faktiske måte å bruke systemet. Dersom brukerne i 80% av tilfeller kun leser ut en verdi fra databasen, endrer denne for så å lagre den tilbake må 80% av testene teste denne handlingen.

Denne testmetoden behøver dog god kunnskap til bruker og brukerdomenet.

Bachs generiske risikoliste

Se etter ting som er: * komplekse * nytt * endret * oppstrømsavhengig * nedstrømsavhengig * kritisk * presis * populær * strategisk * tredjepart (inkl. COTS) * distribuert * buggy * nylig svikt

Test- og systemnivåer



06/img/2.png

Kan teste på ulike nivåer fra nederst og opp :
 * Elektronisk nivå (DoorActuator sender riktig signal)
 * State nivå (Test at døren er lukket dersom state = DoorStateClosed)
 * Logisk nivå (Test at dørene er lukket så lenge man har $\neq 0$ speed, Maintain[DoorStateClosedWhileNonZeroSpeed])
 * Sikkerhetsnivå (Test at dørene er lukket så lenge toget flytter på seg, Maintain[DoorClosedWhileMoving])

Tester altså fra overordnede mål med multi-agenter, til requirements med single-agent.

White box Black box Gray box

White box testing

Her benytter en seg av innsikt internt i applikasjonen til å generere tester. Denne informasjonen benyttes så for å kunne oppnå testdekning i en eller annen for, i eksempelvis kode, stier og valgmuligheter. Debugging vil alltid være white box.

McCabes syklomatisk kompleksitet

Flytgraf

Kompleksiteten er definert som $v(G) = E - N + 2P$

$v(G)$ = syklomatisk kompleksitet
 E = antall kanter i grafen
 N = antall noder i grafen
 P = antall sammenkoblede komponenter

Den sykolomatiske kompleksiteten benyttes for å definere antall gjennomganger som er nødvendig for å kunne garantere (?) dekning for en test.

$v(G)$ er alltid minimum antall stier gjennom koden. Så lenge grafen er en DAG (directed asyclic graph) vil maksimum antall stier være $2^{|\{\text{predikater}\}|}$

$v(g) < \text{antall stier} < 2^{|\{\text{predikater}\}|}$

Et problem en kan stå ovenfor er løkker. En kan dog kunne se antallet stier, selv antallet strengt tatt er “uendelig”.

Beslutningstabell

Generell teknikk for å oppnå fullstendig testdekning, men kan i mange tilfeller føre til overtesting.

1. Lag en tabell med alle predikater
2. Sett inn alle kombinasjoner av True og False for hver predikat
3. Konstruér en test for alle kombinasjoner

Eksempel:

<i>P1</i>	<i>P2</i>
0 0 1 1	0 1
	0 1

Fungerer kun i de tilfeller hvor en står ovenfor binære beslutninger, og hvor skal teste mindre kodebolker. Legg merke at kode som er vanskelig å nå ved at predikater som skal til for at koden skal nåes er muligens ikke en nødvendig del av systemet.

Løkker

Vanlig fremgangsmåte:

- 0 ganger
- 1 gang
- 5 ganger
- 20 ganger

På denne måten kan en teste at koden utfører løkka mange nok ganger for å sannsynliggjøre at den fungerer som den skal.

06/img/1.png

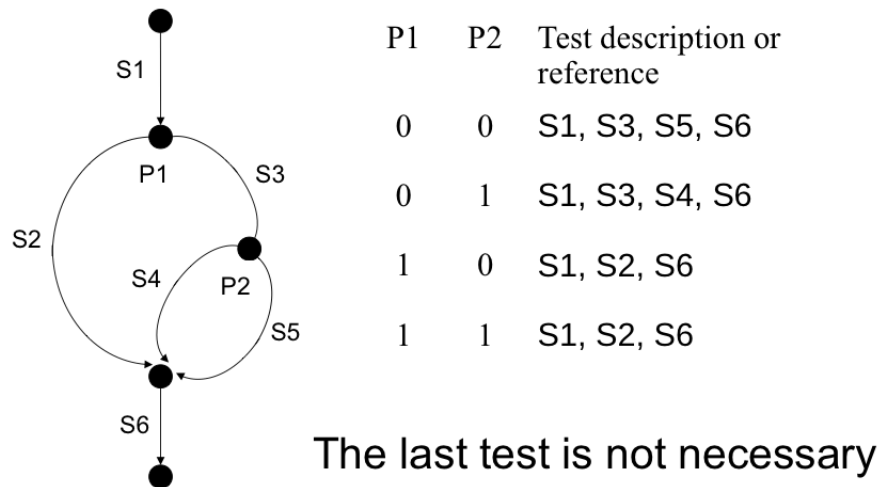


Figure 14: Beslutningstabell eksempel

Feilmeldinger

1. Identifisere alle feiltilstander.
2. Provosere frem identifiserte feiltilstander.
3. Kontrollere at alle feil håndteres på en tilfredsstillende måte.
 - Dette inkluderer at eventuelle feilmeldinger inneholder den informasjonen brukeren behøver for å utbedre problemet.

Videre lesning

[Wikipedia – White box testing](#)

Black box testing

Aka. funksjonell testing.

Utføres uten kunnskap til systemets indre. En vil mate et system med data for så å analysere resultatet.

1. Definere initiell komponenttilstand, input og forventet output for testen.

2. Sette komponenter i definert tilstand.
3. Mate med definert data.
4. Observere output og sammenligne med forventet resultat.

Hovedideen er at en ikke har adgang til til den koden som ligger til grunn for programmet.

Testing av sanntidssystemer

BSP, KSP, TKSP

Videre lesning

[Wikipedia – Black box testing](#)

Gray box testing

Gray box er en kobinasjon av både white box- og black box-testing, og gjøres med *begrenset* kunnskap om systemets indre.

Videre lesning

[Wikipedia – Gray box testing](#)

Grey box testing

Testing med begrenset kunnskap av de interne delene i systemet. Med tilgang til detaljerte designdokumenter ut over kravsspesifikasjoner. Tester genereres basert på informasjon som tilstandsbaserte modeller eller systemets arkitektur-diagrammer.

Tilstandsbasert testing

Deriveres fra forventet systemoppførsel, UML-diagrammer eller andre typer diagrammer. De fleste systemer vil ha et enormt antall tilstander.

Binders tilstander

Liste over vanlige tilstandsfeil. Kan brukes som input i tilstandsbasert testing eller statemachine/kodeinspeksjon

- Manglende eller ukorrekt tilstand
- Ekstra, manglende eller korrupt tilstand
- Sniksti (sneak path)
 - Melding godtatt når den ikke skal det
- “Ulovlig melding”-feil
- Trap-door
 - Systemet godtar udefinert melding

Vi kan velge mellom en eller flere av disse test-utvelgelseskriteriene: * Alle tilstander - testing passerer gjennom alle tilstandene * Alle hendelser - testing tvinger alle hendelser til å skje minst en gang * Alle handlinger - testing tvinger alle handlinger til å bli produsert minst en gang

Teststrategier for tilstand

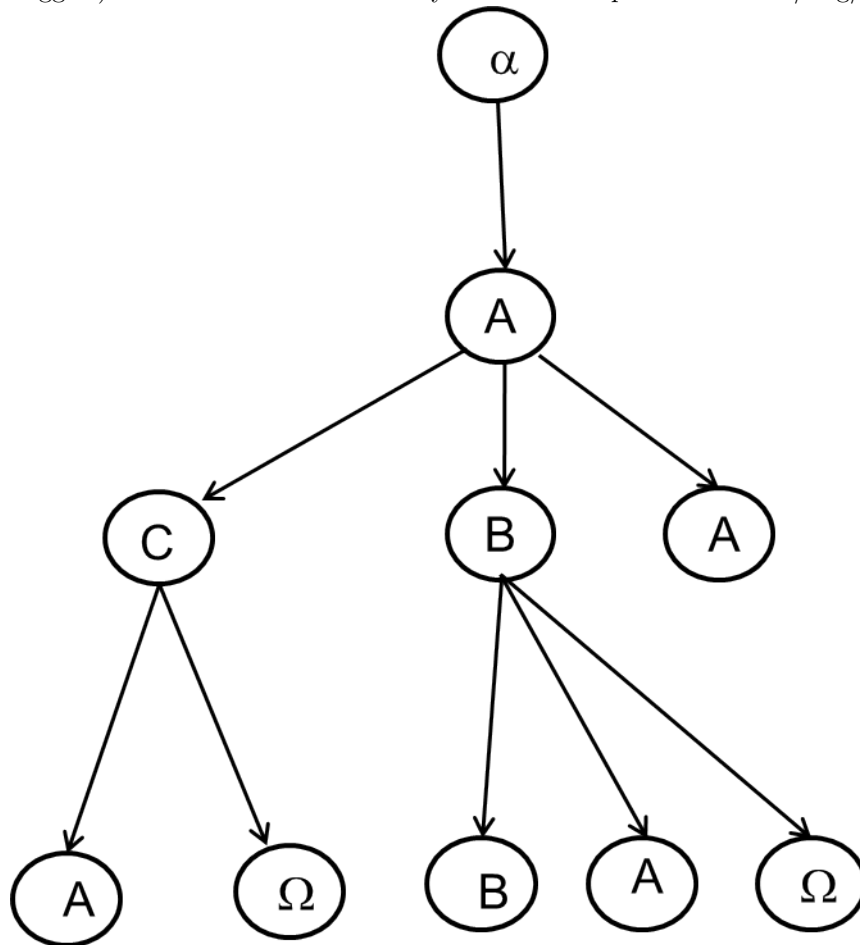
Alle round-trip-stier hvor alle sekvenser begynner og slutter i samme tilstand. Alle enkle stier fra første til siste state, er det loops bruk bare en iterasjon. Strategien hjelper oss til å finne alle invalide eller manglende stier, noen ekstra tilstander og alle hendelses- og handlingsfeil.

Roundtrip path-tre

Bygget fra et tilstandsdiagram og inkluderer alle round-trip stier(se over). Kan brukes til å sjekke likheten mot visse behavioral models, og kan finne snik-stier. En teststrategi basert på et round-trip path tre vil avsløre:

- Alle kontrollfeil i tilstanden
- Alle snikstier (melding godtatt når den ikke burde)
- Korrupte tilstander, uforutsigbar oppførsel.

Utfordringer: Må kunne observere og registrere entitetene (aktiviteter, triggers) for å kunne teste et system basert på disse. 06/img/3.png



Overganger/Transitions Hver overgang i et tilstandsdiagram har formen trigger-signatur[guard]/activity, hvor alle deler er valgfrie.

- Trigger signatur: Ofte en enkelt hendelse som utløser (triggers) en potensiell forandring i tilstand.
- Guard: En boolsk condition som må være sann for at overgangen skal skje.
- Aktivitet: En hendelse som blir gjort under overgangen.

06/img/4.png

ID	Start state	Event	Condition	Reaction	New state
1	α	constructor	-	-	A
2	A	a	p1	w	C
3	C	b	p2	u	A

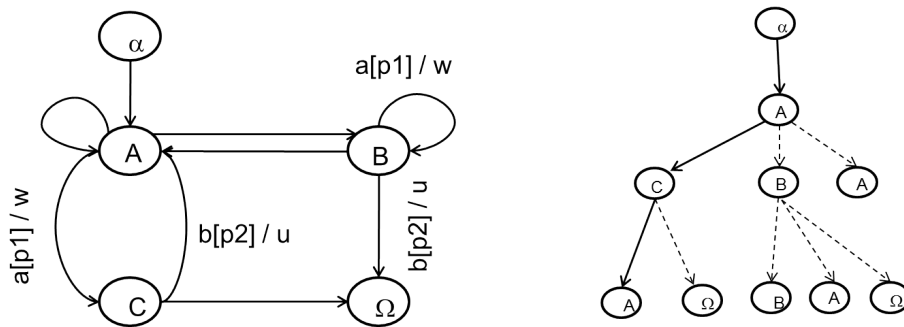


Figure 15: Testcase for $\alpha \rightarrow A \rightarrow C \rightarrow A$:

Teststrategi for sniksti

En sniksti (melding akseptert når den ikke burde) kan forekomme hvis:

- Det er en uspesifisert overgang/transition
- Overgangene forekommer selv om Guard-predikatet er falskt.

Sensor

Se foiler forelesning 6 for mer eksempler

Mutation testing

Type 1

1. Skriv en kodesnutt
2. Skriv et sett tester
3. Test og korrigér til test-suite kjører uten feil
4. Forandre tilfeldig deler av koden (kodemutering)

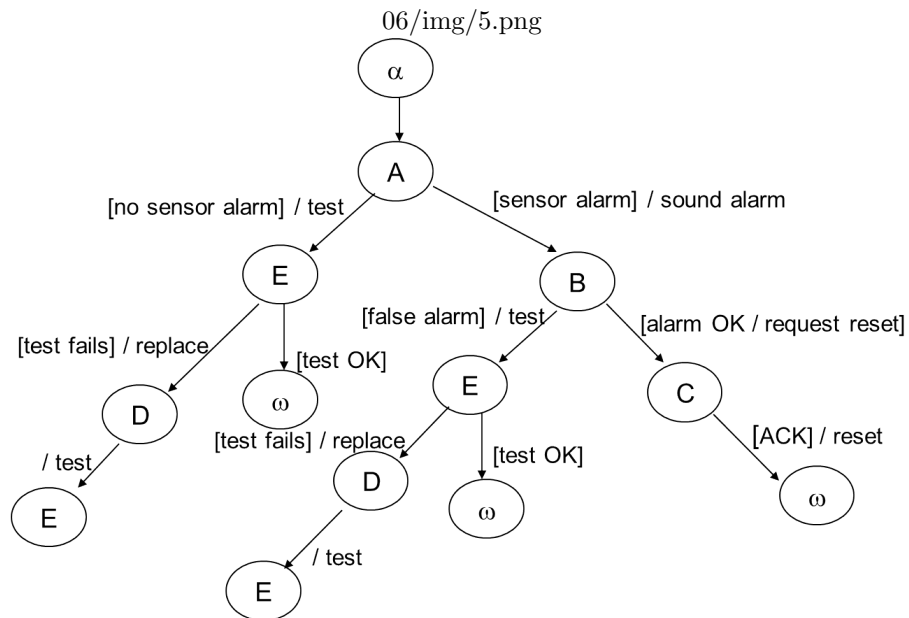


Figure 16: Sensor round-trip path tre

5. Kjør test-suite igjen

Dersom testsuiten kjører uten feil er det noe galt med test-suiten, og den må utvides. Dersom den feiler, gå tilbake til steg 4 og skap en ny mutant. Testeprosessen stopper når alle av X sine nye mutanter er identifisert av den nyeste test-suiten.

Type 2 – fuzzing

Har mye til felles med tilfeldig testing. Forskjellen er at vi her: starter med input som fungerer, endre (mutér) den på en tilfeldig måte. Viktig å tenke på at små forandringer i input kan gi store forskjeller i output. Se foil forelesning 6.3 for eksempel på mutasjonstesting.

Mutanttestingstrategier

Antallet mutanter er stort. For å få et tålelig stort test sett må vi ha visse strategier (referer til Papdakis og Malevris hvis man bruker de). Mutant-teststrategier er enten av 1. eller 2. orden.

* 1. Orden: Utfør en tilfeldig seleksjon av en del av de genererte mutantene, f.eks 10% eller 20% * 2. Orden: Kombinør to mutanter til å få en komponent å teste, her vil strategien avhenge av hvordan vi velger mutantene.

Se foiler for eksempler:

Kommentarer til 1. orden: Måler testeffektivitet gjennom : Test effektivitet (antall testcases / antall identifiserte feil) og kostnadseffektivitet (antall testcases + antall ekvivalente mutanter / antall identifiserte feil) Å velge 10% av alle genererte mutanter er best med tanke på kostnadseffektivitet og testeffektivitet. Såkalt Strong mutation - bruk alle genererte mutanter- er verst.

Kommentarer til 2. orden: SU_F2Last (Same Unit and First combined with Last) scorer høyest på testeffektivitet. Random mix scorer høyest på kostnadseffektivitet. Ingen 2.ordens strategi er mer effektiv enn Rand(10%) strategien, hvor $Fd = \text{Antall test cases} / 1.34$

Test-prioritering

Selv om der er mange måter å prioritere tester på vil dette kurset fokusere mest på *risikobasert prioritering*. Nøkkelord er: *risikovurdering* (assessment) og *prioriteringsmekanismer*.

Risikobasert testing er ikke et nytt fenomen, risikovurdering har blitt brukt lenge av selskaper som utvikler programvare. Dette har dog skjedd på en ustrukturert måte, uten nødvendig dokumentasjon. Vi behøver systematiske metoder for å best kunne håndtere denne risikoanalysen og utføre risikobasert testing.

Risiko er et *forhold mellom et system og dets miljø*. Derfor vil en risiko, dens hyppighet og viktighet variere med interessentene. Sannsynligheten for at den vil oppstå en systemkarakteristikk, men konsekvenser av den vil allikevel variere. Første steg er derfor å identifisere og analysere interessentene.

Interessenter (Stakeholders)

De to hovedgruppene interessenter er *kunder* og *selskapet*. Kunder vil kunne miste penger, enten direkte eller indirekte (på grunn av feilen vil en kunne miste forretninger). Selskapet vil og kunne miste penger via tap av troverdighet. Det er derfor svært viktig at alle interessenter er involverte i risikovurderinger. De vil ha ulik ekspertise og erfaring, som kan bidra til et bedre, bredere data-grunnlag. Videre, grunnet den ulike erfaringen til interessentene må metodene som benyttes for vurderingen være både *enkle i bruk* (helst skal en ikke behøve opplæring i metodene) og *enkle å forstå* (mennesker har lite tillit til ting de ikke forstår).

Risikoidentifikasjon

Det en starter med når en skal identifisere relevante risiko er systemets definerte *use cases*. Hver av deltakerne må derfor på forhånd gjøre seg kjente med disse diagrammene. Det er og lurt med en oppvarmingsøvelse for å utslette misforståelser og å bli enige om en felles prosess. En slik øvelse kan være å gå gjennom resultatene fra tidligere risikoidentifikasjonsprosesser.

I løpet av risikoidentifikasjonsprosess søker en svar på følgende spørsmål for hver enkelt funksjon:

- Hvordan kan denne funksjonen feile?
- Hva er sannsynligheten for at det eksisterer defekter i denne delen av koden?
- Hvilke konsekvenser har disse feilene for interessentene?

Videre vil en kartlegge alvorligheten av hver enkelt feil-modus, og dokumentere resultater fra prosessen i en konsekvenstabell.

<i>Function</i>	<i>Failure mode descripton</i>	<i>Code involved</i>	<i>User</i>	<i>Cust.</i>	<i>Dev.</i>
Hva brukeren ønsker å oppnå.	Hvordan funksjonen kan feile.	Systemelementer involvert. Hvor sannsynlig er det at feilen vil forårsake feilmodus.			

Risikovurdering

For å være nyttig må en risikovurdering være basert på relevant erfaring, være ankret i den virkelige verden, og være et resultat av en dokumentert og *agreed-upon* prosess. En risikovurdering er en subjektiv prosess, men det er svært viktig å ha et solid datagrunnlag for de tall og prognoser som prosessen resulterer i.

Spørsmål som besvares gjennom en slik prosess er gjerne spørsmål som:

- Kan dette virkelig skje? Har det samme eller noe lignende skjedd før?
- Kan vi forklare en plausibel årsak? Kan vi identifisere en konsekvenskjede for hendelsen?
- Hvor ille kan det bli?
- Hvor ofte har dette skjedd før?

Hvordan gjøre en risikovurdering

Overordnet er det to måter å gjøre risikovurderinger: *kvantitativt* og *kvalitativt*. Kvalitative vurderinger gjøres enten via en *sannsynlighet-/konsekvensmatrise* og via *GALE* (Global At Least Equivalent) -metoden. Kvantitativ vurdering gjøres med *CORAS*-modellen.

Kvalitativ vurdering

Når en utfører kvalitativ vurdering benytter en seg av skalaer. Disse må en selv bestemme seg for på forhånd – en skala fra 1 til 10 er like “riktig” som lav-middels-høy-type kategorier.

Kategorier Når en benytter seg av kategorier er det svært viktig å gi en kort beskrivelse av hva hver enkelt kategori impliserer. Dette gjøres ved å relatere det til noe kjent, det vil si for eksempel prosjektets størrelse, selskapets turnover og/eller selskapets profitt. Et eksempel på dette er for konsekvenser si at kategorien **høy** vil bli gitt konsekvenser som på en alvorlig måte setter prosjektets lønnsomhet i fare. I tilfellet for sannsynlighet kan en si at **lav** betyr at hendelsen *kan* inntreffe, men bare i ekstreme tilfeller.

Produktet av **konsekvens** * **sannsynlighet** brukes for å videre bestemme viktigheten av en risiko. En generell regel er at en kun vil se på de risikoer over en gitt sperregrense (M eller L).

GALE-modellen GALE-modellen er en metode som brukes for beslutningstaking hvor en vil bestemme seg for om hvorvidt en forandring skal introduseres eller ei. Modellen er noe mer komplisert enn den versjonen som benyttes i dette kurset, hvor kun risikovurderingsskjemaet benyttes. Dette skjemaet fokuserer på avvik fra nåværende gjennomsnitt.

GALE-indeksen regnes ut på bruk av vurdering av en hendelses *frekvens*, *sannsynligheten* for at en oppstått hendelse skal forårsake problemer og hendelsens *alvorlighet*. Risikoindeksen er definert som $I = Fe + Pe + S$.

<i>Frekvensklasse</i>		<i>Hendelser per prosjekt</i>	<i>Fe</i>
Veldig Ofte	200	Hvert prosjekt	6
Ofte	100	Et prosjekt i blant	5
Sannsynlig	40	Hvert 10. prosjekt	4
Av og til	10	Hvert 100. prosjekt	3
Sjeldent	1	Noen få ganger i selskapets levetid	2
Usannsynlig	0,2	En eller to ganger i selskapets levetid	1
Utrolig	0,01	En gang i selskapets levetid	0

Frekvens-score for en hendelse

<i>Klassifisering</i>	<i>Interpretasjon</i>	<i>Pe</i>
Sannsynlig	Det er sannsynlig at hendelsen, dersom den oppstår, vil føre til et problem.	3
Av og til	Hendelsen, dersom den oppstår, vil av og til føre til et problem.	2
Sjeldent	Det er en liten sjanse for at hendelsen, dersom den oppstår, vil føre til et problem.	1
Usannsynlig	Det er usannsynlig at hendelsen, dersom den oppstår, vil føre til et problem.	0

Sannsynlighetsscore for en hendelse

<i>Alvorlighetsklasse</i>	<i>Interpretasjon</i>	<i>S</i>
Alvorlig	Andelen forekommede problemer har alvorlige konsekvenser i mye større grad enn gjennomsnittlig.	2
Gjennomsnittlig	Andelen forekommede problemer har alvorlige konsekvenser i en gjennomsnittlig grad.	1
Små	Andelen forekommede problemer har alvorlige konsekvenser i mye mindre grad enn gjennomsnittlig.	0

Alvorlighetsscore for en hendelse

CORAS-modellen

CORAS er utviklet som et rammeverk for det å vurdere sikkerhetsrisiko. Det som er av viktighet her er hvordan CORAS relaterer seg til kvalitative riskokategorier som omhandler selskapets turn-over.

Da det er vanskelig å finne realistiske verdier for alle risiko og at det ikke alltid er innlysende hvordan en sammenligner kvalitative og kvantitative risiko, er verdien av kvantitative risiko og muligheter begrenset, selv om de tilbyr håndfaste verdier. Ved bruk av CORAS-tabeller er det svært viktig å huske at utviklere, kunder og brukere vil ha ulike verdier på de ulike postene, da en har ulike utgangspunkt for det å verdsette disse.

08/img/coras.png

Consequence values

Category	Insignificant	Minor	Moderate	Major	Catastrophic
Measured related to income	0.0 – 0.1%	0.1 – 1.0%	1 – 5%	5 – 10%	10 – 100%
Measured loss due to impact on business	No impact on business. Minor delays	Lost profits	Reduce the resources of one or more departments Loss of a couple of customers	Close down departments or business sectors	Out of business

Figure 17:

Det er mulig å forstå frekvenser på to måter: målt i antall uønskede hendelser per år; og andel feilende etterspøringer/leveringer.

CORAS frekvens-tabell

<i>Kategori</i>	<i>Sjeldent</i>	<i>Usannsynlig</i>	<i>Mulig</i>	<i>Sannsynlig</i>	<i>Svært trolig</i>
Antall uønskede hendelser per år	1/100	1/100 - 1/50	1/50 - 1	1 -12	> 12
Antall uønskede hendelser per etterspørring	1/1000	1/500	1/50	1/25	1/1
Interpretasjon av antall etterspørringer	Uønskede hendelser skjer aldri.	Hvert tusende gang systemet brukes.	Hver femte gang systemet brukes.	Hver tiende gang systemet brukes.	Hver gang systemet brukes.

Eksempel Under er et eksempel på bruk.

Tekst	Tall
<i>Årlig inntekt</i>	100.000.000
<i>Feilkonsekvens</i>	0.001 - 0.01 (mindre)
<i>Feilfrekvens</i>	1 per år - 2 per 100 år (mulig)

maksimal risiko = 100.000.000 * 0.01 * 1 = NOK 1.000.000

minimal risiko = 100.000.000 * 0.001 * 1/50 = NOK 2.000

Worry List

Testing er en “sampling”-prosess, noe som vil si at dersom vi finner mange defekter i en komponent bør vi konkludere med at komponenten har mange defekter. Denne konklusjonen vil ikke nødvendigvis være den samme hvis vi baserer konklusjonen på antallet defekter funnet i en inspeksjon.

Risikobasert testing

Risikobasert testing involverer følgende to steg:

1. Identifisere hvordan produktet interagerer med sitt miljø. Dette gjøres for å forstå konsekvenser av feil.
2. Identifisere og rangere risiko etter sannsynlighet og konsekvens.

08/img/1.png

		Consequences		
		High (3)	Medium (2)	Low (1)
Number of errors registered	High 10 -			
	Medium 3 – 9			
	Low 0 - 2			

Figure 18: Testcase for alfa -> A -> C -> A:

- Dersom det er en white box- eller grey box-test bør en også identifisere mulige kausale hendelseskjeder med den hensikt å forstå feilmekanismene.
3. Definere tester som kan brukes for å forsikre at sannsynligheter for defekter i koden er lav.
 - Black box-testing – tilfeldig testing eller domenetesting
 - White box- eller grey box-testing – forsikre at alle identifiserte kausale hendelsesforløp benyttes og testes skikkelig.
 4. Kjøre testene. En starter med de høyeste risikoer og jobber seg nedover.

Eksempel på bruk av ATM

Kravshåndtering

Egenskaper i en effektiv Requirements Engineering (RE) prosess 1. Minimere forekomst av feil i krav 2. Dempe virkningen av endring i krav 3. Kritisk for suksess i ethvert utviklingsprosjekt

Målet med RE-prosessen er å sikre at krav for et system kan tildeles en spesi-
fikk programvarekomponent som da har ansvar for å tilfredsstille kravet. Når

08/img/ATM-ex.png

Subsystem			Consequences		
Function	Failure mode descripton	Code involved	User	Cust.	Dev.
Ta ut penger	Bruker tar ut mer enn tillatt.	W-1, M-1	L	M	H
	ATM registrerer feil beløp.	Acc-1, M-1	H	H	H
	Feil konto aksesseres	Acc-2	L	H	H
Sette inn penger	Feil beløp registreres	M-1, V-1	H	H	H
	Feil konto	Acc-2	H	H	H
Forespørsel	Feil konto	Acc-2, V-1	M	L	M
	Feil beløp returneres	V-1, M-1	M	M	M

Funksjon	Komponenter	S	Fe	Pe	I
Sette inn – feil beløp registreres	M-1, V-1	2	4	3	9
Forespørsel – feil konto	Acc-2, V-1	1	3	2	6

Figure 19:

en slik tildeling er mulig: * Den resulterende programvaren er ordentlig/riktig modulisert * Modulene har tydelige grensesnitt * Alle krav er tydelig separerte

Kriterier for god kravshåndtering

Håndtere view-points for systemet som skapes. Håndtere ikke-funksjonelle krav og soft goals. Håndtere identifikasjon og håndtering av krysskutting og ikke-krysskyttende krav Håndtere innvirkningen av bruk av COTS, outsourcing og subcontracting.

Viewpoints, perspektiv og views

- Viewpoint defineres som et standpunkt brukt av et individ når man undersøker et interesseområde- i vårt tilfelle, kombinasjonen av agent, og viewet som agenten innehar.
- Et perspektiv er definert som et sett fakta, observert og modellert i henhold til bestemte aspekter av virkeligheten.
- Et view defineres som en integrasjon av disse perspektivene.
- Et viewpoint-språk er brukt til å representere viewpoints.

Ulike typer viewpoints

Datakilder eller -sluk : Viewpoints som produserer eller konsumerer data. Analyse involverer sjekking av at data produseres og konsumeres, samt at antagelser om kilden og sluket er gyldige. *Representasjonsrammeverk*: Viewpoints som representerer visse deler av en systemmodell (f.eks state-machine representasjon), passer særlig bra for sanntidssystemer. *Mottakere av tjenester*: Viewpoints som er eksterne til systemet og som mottar tjenester fra det. Passer best til interaktive systemer.

VORD-metoden

1. Identifikasjon av viewpoint - Identifisere viewpoints som mottar systemtjenester, og identifiser tjenestene som blir gitt til hvert viewpoint
2. Strukturering av viewpoint - Gruppere relaterte viewpoints inn i hierarki. Fellestjenester gis på høyt nivå i hierarkiet
3. Dokumentering av viewpoint - Utbedre beskrivelsen av de identifiserte viewpoints og tjenester.
4. Mapping av viewpoint - Gjøre om analysen til et objektorientert design.

Viewpoint template

Reference: The view point name

Attributes: Attributes providing viewpoint information

Events: A reference to a set of event scenarios describing how the system reacts to viewpoint events

Services: A reference to a set of service descriptions

Sub-VPs: The names of sub-viewpoints

08/img/2.png

08/img/3.png

Service template

Reference: The service name

Rationale: Reason why the service is provided.

Specification: Reference to a list of service specifications. These may be expressed in different notations.

Viewpoints: A List of viewpoint names receiving the service

Non-functional requirements: Reference to a set of non-functional⁷² requirements which constrain the service.

Provider: Reference to a list of system objects which provide the service.

Et eksempel med kunde (customer) og skal ta ut penger (cash withdrawal)

Reference: Customer

Attributes: Account number; PIN; Start transaction

Events: Select service;
Cancel transaction;
End transaction

Services: Cash withdrawal
Balance inquiry

Sub-Viewpoints:
Account holder
Foreign customer

08/img/4.png

08/img/5.png

Reference: Cash withdrawal

Rationale: To improve customer service and reduce paperwork

Specification: Users choose this service by pressing the cash withdrawal button. They then enter the amount required. This is confirmed and, if funds allow, the balance is delivered.

Viewpoints: Customer

Non-functional requirements: Deliver cash within 1 minute of amount being confirmed

Provider: _____

Kravhåndtering - Viewpoint

Fordeler med viewpoint-orientert tilnærming i kravhåndtering

- Assisterer med å forstå og kontrollere kompleksiteten ved å separere interessene til de ulike aktørene
- Eksplisitt gjenkjenne mangfoldet det er av kilder til krav
- Gir en mekanisme for å organisere og strukturere dette mangfoldet
- Får en slags fullstendighet
- Gir mulighet for kravkildene eller stakeholders til å identifisere og sjekke deres bidrag til kravene.

Non functional requirement

Et mål er oppnådd når alle dets delmål er oppnådd. NFR Framework er sentrert rundt såkalte soft goals, som ikke har noe klart kriterie for oppnåelse. Slike soft goals er oppnådd når det er nok positive og lite negative bevis for målet og vice-versa når det er mye negativt og lite positivt. Soft goals analyseres i relasjon med hverandre.

Non functional requirement Framework (NFRF): Analyse for ikke-funksjonelle krav:

1. Starter med soft goals som representerer ikke-funksjonelle mål. Disse er interessantene blitt enige om, f.eks usability, flexibility etc.
2. Hvert soft goal refineres så ved å bruke dekomposisjon.

Dekomposisjon kan baseres på : Generell ekspertise rundt f.eks security, flexibility. Domene-spesifikk kunnskap. Prosjekt-spesifikk kunnskap. Se foilene for eksempel på NFR-analyse/-dekomposisjon.

Fordeler ved NFR rammeverket: NFR oppnås ved å samle kunnskap om domenet for et system skal bygges. NFRF fokuserer på å klargjøre meningen til de ikke-funksjonelle kravene. NFRF gir alternativer for oppfyllelse av soft goals.

Cross cutting (tverrgående) krav:

Et delmål, et konkret krav kan være involvert i oppfyllingen til flere enn et høynivå-mål. Mange ikke-funksjonelle krav faller i denne kategorien. F.eks : Ytelse er en faktor av systemarkitekturen. Vi kan ikke utvikle en ytelsesmodul som er uavhengig av de andre delene til systemet. Det er slike krav som kalles tverrgående krav. Eksempler på slike er : Sikkerhet, mobilitet, tilgjengelighet og sanntidsbegrensninger.

Aspekt-orientert requirement engineering Handler om å identifisere tverrgående krav tidlig i krav og arkitekturdesign fasen, istedet for i implementasjonsfasen. 4 steg : Identifisere, fange (capture), komponere (compose) og analysere. Se foiler for dyptgående eksempel av et banksystem med tverrgående krav. Her prøver man å skrive om krav for å fjerne spredte konsepter.

COTS (Commercial off the shelf)

Når størrelsen og kompleksiteten til systemer vokser, ser man på COTS som en mulig løsning. Her blir krav begrenset av tilgjengeligheten til passende COTS-komponenter. Viktig aspekt er å evaluere mulig COTS programvare tidlig i livssyklusen til prosjektet.

Impact av COTS : For bedriftsapplikasjoner kan man bruke et stort COTS-produkt til å levere et eller flere krav (f.eks MS Office eller Oracle) For innebygde sanntids- eller sikkerhetskritiske domener, er COTS forventet å være små, og kreve store mengder av "lim-kode" til å integrere COTS komponentene med resten av systemet.

Problemer med COTS:

- Begrenset tilgang til produktets interne design
- Beskrivelsen av kommersielle pakker er noen ganger ufullstendig og forvirrende
- Kunder har en begrenset sjanse til å verifisere hvorvidt et ønsket krav er møtt.
- De fleste av seleksjonsavgjørelsene er basert på subjektivitet, slik som partnerskap og markedsføring.

Fordeler med COTS: vi får et produkt som har blitt testet mange ganger av ekte brukere med konsekvent forbedring av programvarekvaliteten. Se foil 8.2 (Requirements) for eksempel på COTS-krav.

Krav for outsourcing

Outsourcing er en management strategi, hvor en organisasjon outsourcer store, ikke-kjernefunksjoner til tjenestetilbydere og tredjeparter. Outsourcing blir stadig mer vanlig. Skiller mellom:

- Onshore : Outsource et prosjekt innad i eget land
- Offshore : Outsource til land utenfor europa (f.eks India)
- Nearshore : Nearshore vil for Norge og Skandinavia være baltiske land (ikke lange avstander)

Faser i outsourcing: *Seleksjon :* Velge underleverandør. *Monitoring :* Signere kontrakte og følge underleverandørens arbeid til produktet er levert. *Fullføring :* Akseptanse og installering av produktet, og også vedlikehold av produktet i dets levetid.

Fordeler og ulemper *Fordeler:* Kostnadsbesparende. Bedre tjenestelevering og kvalitet. Holde tritt med teknologisk innovasjon. *Ulemper:* Firmaer mister kontroll over business-processes og inhouse ekspertise.

Konklusjon

Flere måter å identifisere og behandle krav på, hver med sine iboende kompleksiteter og avhengigheter. Viewpoints prøver å eksplisitt modellere interessene til de ulike aktørene. NFR-rammeverk fokuserer på å modellere soft goals og klargjøre deres mening. Tidlige aspekter fokuserer på å identifisere tverrgående forhold i krav i en tidlig fase av prosjektet. Finnes ytterligere kravbehandlingshensyn når man bruker COTS eller outsourcer.

COTS-testing

Ofte brukte tilnærminger til testing av COTS:

- Component meta-data
- Retro-components (retrospector)
- Built-in test (BIT)
- STECC strategy
- COTS

Component meta-data

Meta-data : Data relatert til komponenter, *men er ikke kode*.

Typer meta-data er for eksempel:

- Tilstandsdiagrammer
- Quality of Service-informasjon
- Pseudokode og algoritmer
- Test-logger
 - Hva har blitt testet?
- Bruks-mønstre
 - Hvordan har komponenten blitt brukt til nå?

Meta-data kan ta fryktelig mye plass, og er derfor en integrert del av komponenten, dog *lagret separat*. Meta-data lastes ned/inn ved nødvendighet. Integrert i komponenten, men separat fra funksjonalitet.

Hva kan en så bruke disse meta-dataene til i denne sammenhengen?

- *Round trip path test* basert på tilstandsdiagrammer
- En kan skape *funksjonelle tester* basert på algoritmer eller pseudo-kode.
- Test-logger kan skape et grunnlag for *relevansevurdering* av tester.

Retro-component

Retrospector : Et verkøty som registrerer testing og utføringshistorien til en komponent. Denne informasjonen lagres som [meta-data](#).

Retro-component : Programvare med en retrospector.

Bruk av retrospectors

Gjennom å samle informasjon om bruk har vi et bredere grunnlag for å kunne fjerne feil, samt teste programvare. For brukere av COTS-komponenter vil retrospectors gi verdifull informasjon om *hvordan komponentene ble testet* og *hvordan komponenter har blitt brukt av andre*. Sistnevnte vil si oss noe om hvorvidt komponenten blir brukt på nye måter, noe som fører med seg en større risiko.

Built-In Test(BIT)

Nøkkelord:

- Run-Time Testability(RTT)

Her trenger en to sett med tester:

- I komponenten vil en teste at (test-) miljøet oppfører seg som foreskrevet.
- I komponentens klienter vil en teste at komponenten implementerer de semantikker som klienten har blitt utviklet for å forvente.

Testing av komponenter

Når en vil teste komponenter gjør en det ved å utføre følgende to steg:

1. Bring komponenten til testens starttilstand.
2. Kjør testen.
3. Kontrollere at
 - resultater er som forventet, og
 - slutt-tilstand er som forventet.

En må liste opp *starttilstand*, *forutsetning* hvor det er hensiktsmessig, *hendelse* og *slutttilstand*. Under er et eksempel for en enkel girkasse.

.	<i>Starttilstand</i>	<i>Forutsetning</i>	<i>Hendelse</i>	<i>Slutttilstand</i>
1	Neutral	[momentum < ReverseMomentum]	<code>toReverse()</code>	Reverse
2	Reverse	-	<code>toNeutral()</code>	Neutral
3	Neutral	[momentum < Gear1Momentum]	<code>toGear1()</code>	Gear1
4	Gear1	-	<code>toNeutral()</code>	Neutral
5

Valg av tester å utføre

Ved valg av tester er det særlig to punkter en må vurdere: *testenes kvalitet* og *testenes størrelse*. Jo mer utfyllende test dess “bedre” er den, dessverre betyr dette også at jo mer utfyllende jo større er testen. Når det gjelder testens størrelse så ønsker en at testen skal være rask å utføre, noe som legger sterkt press på å holde størrelsen nede. Løsningen på dette er å ha flere forskjellige sett med tester som utføres på ulike tidspunkt.

BIT-arkitektur(Built in test-arkitektur)

BIT-arkitekturen består av følgende arkitektur:

- Komponenten
 - Med et eller flere grensesnitt og implementasjoner av funksjonalitet

- BIT-RTT
 - Gir støtte for testingen
- Eksterne tester
 - Kjører de nødvendige testene
- Håndterer (handler)
 - Tar seg av feil, exceptions, fail-safe-oppførsel og lignende
- Konstruktør (constructor)
 - Initialiserer komponenter, slik som eksempelvis håndtererene og eksterne testere.

Ulemper ved bruk av BIT

BIT er av en *statisk natur*, en kjører én eller flere *forhåndsdefinerte* tester flere ganger, og når en test er definert som vellykket vil komponenten være friskmeldt, og testen være over. Videre vil en generelt ikke kunne være helt sikker på at de tester som utføres er de som er krevd av komponenters faktiske klienter og brukere. Dessuten har komponent-tilbyder *antakelser* om brukerens krav, noe som kan være både feil og/eller unøyaktig.

Self TEsting COTS Components (STECC)

STECC har mye til felles med [BIT](#), hovedforskjellene er at BIT er statisk og kjører en eller flere forhåndsdefinerte tester. STECC er dynamisk og genererer nye tester basert på beskrivelser. STECC vil også kunne interagere med testeren.

En testgenerator vil generere tester, og være det leddet som kommuniserer med server for utveksling av metadata. Testgenerator vil kunne kommunisere til (?) testeren, samt sende spørringer [et sted].

Assessing COTS

Når en vurderer kandidatkomponenter for testing, må utviklere spørre seg selv følgende tre spørsmål for hvert definerte scenario:

1. Oppfyller komponenten de behov *utvikler* har?
2. Er komponentens *kvalitet* god nok?
3. Hvilken innvirkning vil komponenten ha på *systemet*?

Naturlig å betrakte svarene på disse spørsmålene i henhold til ulike scenarioer. Trengs komponenten C for systemet S? Er C av høy nok kvalitet? Har C en positiv innvirkning på S?

Black box-test-reduksjon ved bruk av input-output-analyse

Tilfeldig testing er aldri komplett. For å utføre komplett funksjonell testing kan en redusere antall test-caser ved hjelp av input-output-analyse. Relasjoner mellom input og output kan en identifisere ved bruk av *statisk analyse* eller ved *kjøre analyse* (execution analysis) av programmet.

Det en ønsker å avdekke er hvilke inndata som påvirker hvilke utdata, for så å kunne finne den minimale mengden forskjellig testdata for å holde størrelsen nede (og hastighet oppe). Etter analysen kan en utføre black box-testing basert på de data som har blitt funnet. Se foil 9-1 for eksempel på en slik reduksjon.

Outsourcing, subcontracting and COTS

Ansvar

Det er utviklers hele og fulle ansvar for at det produktet som leveres er av en gitt kvalitet. Det er videre kun mulig å få godtgjørelse fra et selskap om en kan bevise kontraktbrudd fra leverandør sin side. Det er derfor viktig å kunne bygge opp tillit ved å teste tilstrekkelig. Dette er selvfølgelig svært viktig ved bruk av COTS.

Testing og tillit

En test har ulike roller i de ulike fasene av utviklingen. I selve utviklingen og produksjonen av kode er det viktig for en test å kunne avdekke feil og mangler ved denne koden. I akseptansefasen, er det viktig for en test å kunne bygge tillit til programvarens komponenter. Ved bruk av COTS er det spesielt viktig å bygge opp denne tilliten.

Tillit og troverdighet er noe som må *defineres* eksplisitt. System og miljø må óg defineres. Tillit er relatert til *produkt* (eks. en COTS-komponent), *prosess* (*hvordan* komponenten ble utviklet og testet) og *mennesker* (*hvem* som utviklet og testet komponenten). Dette er et eksempel på et produkttillits-mønster.

Det samme gjelder tillit til prosessen. For å bygge denne tilliten er det viktig å sikre at prosessen sporbar, og prosessen og benyttes korrekt, metoden som benyttes svarer til problemet som skal løses, samt at utviklerteamet er kompetent. Dette er et eksempel på et prosessstillits-mønster.

Testing og outsourcing

Dersom utvikling outsources må testing være en integrert del av utviklingsprosessen. Testing er dermed kontraktrelatert. Dersom en skal benytte tillitsmønstre må en derfor inkludere krav til komponent (hva), personells kompetanse (hvem) og prosessen (hvordan).

Outsourcing-krav

En kontrakt som brukes ved outsourcing bør inkludere:

- Krav til personell. En behøver å se CVer til hver av utviklerne.
- Krav til utviklingsprosessen, inkludert testing. Dette kan komme av egen revisjon av prosessen, eller av standardiserte sertifikater som f. eks. ISO 9001.

Der er videre viktig å kunne se og inspisere en rekke viktige artefakter:

- Prosjektplanen
 - Når skal hva gjøres?
- Teststrategi
 - Når skal testing av våre krav til komponenten testes?
- Testplan
 - Hvordan vil tester kjøres?
- Testlogg
 - Hva vil testene resultere i?

Tilliten vi vil ha til komponenter vil være avhengig av hvor tilfredse vi er med svarene på disse spørsmålene. Tillit kan imidlertid bero på vår tidligere erfaring med selskapet. Generelt vil kontraktens rigiditet på dette feltet være avhengig av denne tidligere erfaringen.

Testing av COTS

Testing av COTS skjer gjerne ved bruk av black box-testing eller domain partition testing. Anekdotiske bevis eksisterer som sier at en drar mest nytte av å fokusere på tester for *intern* og *ekstern robusthet*. Hva en behøver for å teste disse typene robusthet, samt viktigheten av de varierer med typen komponent.

Intern robusthet : Komponentens evne til å håndtere feil i komponenter eller miljø. For å teste dette behøver en wrappere, feilinjeksjon, m.m. Intern robusthet vil være viktigst i de komponenter som kun er synlige inne i systemets grenser.

Ekster robusthet : Komponentens evne til å håndtere feil i inndata. Her behøver en kun komponenten “as is”. Ekstern robusthet er viktig hvor komponenten er en del av brukergrensesnittet.

Testing av intern robusthet

Intern robusthet dreier seg om en komponents evne til å håndtere alle feilaktige situasjoner som f. eks. minnefeil og feilende funksjonskall. En ønsker at koden i slike tilfeller vil gå til et definert, trygt, stadie etter å ha gitt en feilmelding. Informasjonstapet etter et slikt tilfelle skal og være minimalt.

Wrapper Wrapper : En implementasjon som definerer den funksjonaliteten vi ønsker tilgang til. Dette kan, men behøver ikke, være et objekt. : “Wrapper”-klassen tilbyr et objektgrensesnitt samt metoder som håndterer implementasjonen. Istedenfor å kalle på en metode i implementasjonen direkte, kaller klienten metoden via wrapperen som videre aksesserer implementasjonen.

Bruk av en wrapper er nødvendig for å oppnå en rekke ønskede effekter.

- Kontroll av komponentens input, selv om komponenten er satt inn i et realistisk system.
- Kunne samle og rapportere alle inn- og utdata fra komponenten.
- Vi kan manipulere exception handling, samt påvirke kun den ene komponenten alene.

Fault-injeksjon En *fault* er en abnomal kondisjon eller defekt som i tur kan forårsake en *failure*. *Fault*-injeksjon involverer det å forsettelig legge inn feil i programvaren for å fremprovosere en respons. *Fault*-injeksjon involverer to steg.

Først vil en identifisere det settet *faults* som kan oppstå i en applikasjon, modul, klasse eller metode. Det er med andre ord ingen poeng i injisere nettverksfeil i

programvare som ikke bruker nettverkskommunikasjon. En injiserer så slike feil i programvaren for å evaluere hvordan disse håndteres. En ønsker å kartlegge om applikasjonen detekterer feil, hvorvidt feilen isoleres og om applikasjonen overlever?

// TODO: Fyll ut litt mer her

Testing av ekstern robusthet

Feilhåndtering må testes for å kontrollere at:

- Feil i inndata bryr ikke applikasjonen, men gir en forståelig feilmelding. Denne feilmeldingen er lett forståelig for de intenderte sluttbrukerne, informasjonsnivået må tilpasses interessenter.
- Applikasjonen skal kunne fortsette etter feilen med minimum tap av informasjon.
- Applikasjonen skal gå til en *definert* tilstand ved feil.

Feilmeldinger En bruker skal ha all den informasjonen som behøves for å kunne korrigere feil i inndata, samt fortsette arbeidet fra nåværende tilstand. Dette krever innsikt i brukeren og brukerens behov. Dette testes ved å få en bruker til å arbeide med en realistisk oppgave, for så å skape en feil.

Ved å observere brukeren og brukerens reaksjon vil en kunne se hvorvidt der er noen assistanse i feilmeldingen.

Sekvensiell testing

For å kunne utføre sekvensiell testing trenger man en rekke variabler.

- Target failure rate p_1
- Uakseptabel failure rate p_2 , hvor $p_2 > p_1$
- Akseptabel sannsynlighet for å gjøre en type I eller type II beslutningsfeil
 - og disse to verdiene brukes for å kalkulere a og b hvor
 - $a = \ln(\text{beta} / (1 - \alpha))$
 - $b = \ln((\text{beta} - 1) / \alpha)$

WHOA DUDE! Dette ble plutselig veldig matematisk!

Got it. Magic.

Oppsummering

Ved testing av programvare ($p < 10^{-3}$) vil den sekvensielle testmetoden behøve et stort antall tester og bør derfor kun brukes for å teste robusthet basert på tilfeldig genererte inndata. Ved inspeksjon av dokumenter ($p < 10^{-1}$) vil metoden gi verdifull informasjon selv når en inspiserer et rimelig antall dokumenter.

Enkle Bayesiske metoder

Bayesisk statistikk benyttes for å kombinere de tre faktorene: *testresultater*; *kontraktuelle forpliktelser*; og *tidligere erfaring med leverandør*.

Bayes' teorem : $P(B|A) = \frac{P(A|B) P(B)}{P(A)}$

For å estimere B vil vi bruke sannsynligheten av våre observasjoner som $P(A)$, og bruke $P(B)$ som modell på vår tidligere kunnskap.

\ TODO: senere. Mye senere.

Ansvar

Viktig å tenke på:

- Firmaet som bringer produktet til markedet har fullt ansvar for produktets kvalitet
- Er bare mulig å søke om oppreisning etter en outsourcing, dersom vi kan bevise at de ikke fullførte sin kontrakt.

Testing og selvtillit

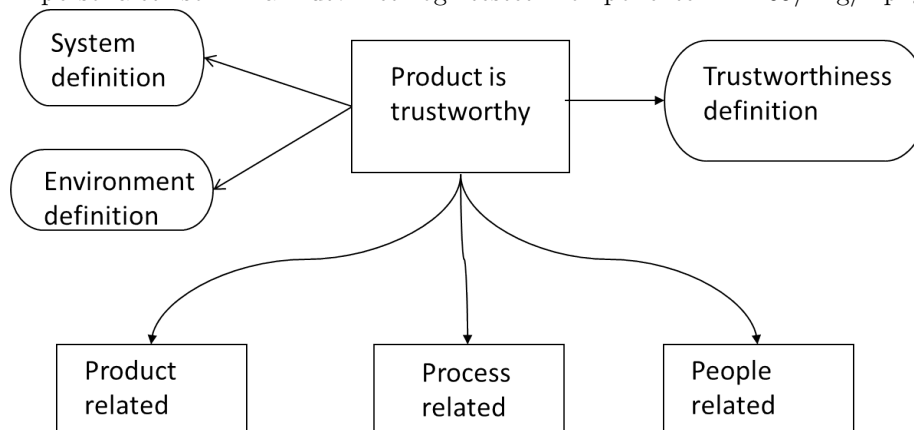
Rollen testing har under:

- utvikling - Finne og fjerne defekter
- Akseptanse - Bygge selvtillit i komponenten

Når vi tester COTS eller komponenter hvor utviklingen har blitt outsourced/subcontracted, ønsker vi å bygge selvtillit(confidence).

Skape tillit til produktet:

Man kan bygge tillit basert på : Produktet i seg selv, f.eks en COTS komponent. Prosessen - hvordan den ble utviklet og testet. Folket - personalet som har utviklet og testet komponenten. 09/img/2.png



Skape tillit til prosessen:

Tillit i prosessen stammer fra tre kilder: Hvem gjør det? - "Teamet er kompetent" . Hvordan blir det gjort? - "Metoden adresserer problemet". Vi kan sjekke at prosessen blir brukt korrekt - "Prosesen er sporbar".

Testing og outsourcing

Når vi outsourcer utvikling, må testingen være en integrert del av utviklingsprosessen. Testing blir da altså et kontraktsspørsmål. Fra trustworthiness mønsteret ser vi at vi må inkludere krav for: Komponentene - Hva . Kompetansen til personalet - Hvem . Prosessen - Hvordan.

Krav ved outsourcing.

Når man skal tegne en outsourcing kontrakt bør man inkludere:

- Krav til personell - riktige personer til jobben, se CV til hver person.
- Utviklingsprosessen - inkludert testing. Slik tillit kan komme fra sertifikater(f.eks ISO 9001) eller egne prosessrevisjoner.
- Prosjektplan - når skal de gjøre hva?
- Test strategi - Hvordan vil de teste komponentkravene våre?

- Testplan - Hvordan vil testene bli kjørt?
- Testlogg - hva var resultatet av testene?

Tillit i komponenten

Tilliten vi har til en komponent avhenger av hvor fornøyd vi er med svarene på spørsmålene over. Vi kan dog også bygge tillit på tidligere relasjoner med selskapet. Dess større denne tilliten basert på tidligere erfaringer er, jo mindre strenge behøver vi være i kontrakten.

Teste COTS

COTS testes ved å benytte black-box testing eller domene-partisjonstesting. Erfaringer tilsier at vi vil få mest utav innsatsen ved å fokusere på tester for : Intern robusthet, og ekstern robusthet.

Robusthet

- Intern Robusthet : Evnen til å behandle feil i komponenten eller dens miljø. Her trenger vi wrappers, feilinjeksjon etc. Her er komponenter som kun er synlige innenfor systemets grenser.
- Ekstern robusthet : Evnen til å behandle feil input. Her trenger vi kun komponenten “som den er”. Her er komponenter som er del av brukergrensesnittet.

Viktigheten av disse to typene vil variere , avhengig av typen komponentener vi tester.

Intern robusthet Intern robusthet er evnen til å :

- Overleve alle feilaktige situasjoner som; Minnefeil - både kode og data. Feiling av funksjonskall, inkludert kall til OS funksjoner.
- Gå til en definert , *trygg* tilstand etter å ha gitt feilmeldingen.
- Fortsette etter en slik feilaktig situasjonen med et minimumstap av informasjon.

Wrappers En wrapper fungerer som en slags mellomtjener som gir oss et grensesnitt hvor vi kan få tilgang til tjenester andre steder i systemet. Wrapperne har som hovedoppgave å kalle andre klasser og metoder, gi oss tilgang til disse gjennom et veldefinert grensesnitt uten å berøre andre deler. Dette gir lavere kompleksitet. En wrapper har to essensielle karakteristikk:

1. En implementasjon som definerer funksjonaliteten vi vil ha tilgang til. Dette kan være et objekt, men trenger ikke være det (Eksempel på ikke-objekt implementasjon: Vi trenger tilgang til funksjoner til en DLL).
2. “Wrapper” klassen gir oss tilgang til implementasjonen og metoder for å administrere implementasjonen. Klienten kaller en metode på wrapperen som igjen aksesserer implementasjonen som er nødvendig for å fullføre oppgaven.

Hvorfor wrappers? : Vi kontrollerer komponentens input, selv om komponenten er i et ekte system. Vi kan samle og rapportere input og output fra komponenten. Vi kan manipulere feilbehandlingen på denne ene komponenten uten å berøre resten av systemet.

Fault injection En fault er en defekt eller unormal tilstand som kan føre til svikt i systemet. Ved fault injection setter vi inn feil med vilje slik at vi kan se responsen systemet gir. Målet er å ikke gjenskape omstendighetene som fører til en slik feil. Fault injection brukes ofte med stress testing, og blir sett på som en viktig del av å utvikle robuste systemer. To steg i fault injection:

1. Identifiser sett med feil som kan forekomme i et system, modul, klasse, metode. Eksempel: Ikke noe vits å sette inn nettverksfeil dersom applikasjonen ikke bruker et nett.
2. Utøv disse feilene for å evaluere hvordan applikasjonen svarer. Typisk finner applikasjonen feilen, er feilen isolert og “overlever” applikasjonen?

Se foil 9-2 for utøvende eksempel på fault injection.

Ekstern robusthet Feilbehandling må bli testet for å vise at: Feil input gir en feilmelding. Feilmeldingen skal være forståelig for de tenkte brukerne. Systemet skal fortsette etter feilen med et minimalt informasjonstap. Ekstern robusthet er evnen til å :

- Overleve input av feilaktig data (no crash)
- Gi en enkel-å-forstå feilmelding som hjelper brukeren til å rette på input-feil.

- Dette kan kun testes ved å involvere brukerne og er dermed ikke like lett å teste som de andre punktene. Vi trenger å vite hvilken info brukeren trenger for å kunne korrigere feil i input og fortsette der han slapp i samme tilstand.
 - Den enkleste måten å teste feilmeldinger på er å be en bruker om å : Begynne på en arbeidsoppgave, sette inn en feil i input underveis i oppgaven. Vi kan da observere hvordan brukeren prøver å komme seg ut av situasjonen og hvor fornøyd han er med hjelpen han får fra systemet.
- Gå til en definert tilstand
 - Fortsette etter feilsituasjonen med et minimumstap av informasjon.

Sekvensiell testing Se foil for utøvende eksempel. P = target failure rate. hvis den er $p < 10^{-3}$ så trenger man et stort antall tester. Det bør derfor kun bli brukt for robusthettesting basert på automatisk generert random-input Hvis $p < 10^{-1}$, så vil sekvensiell testing gi brukbare resultater selv om man inspiserer et forholdsvis stort antall dokumenter.

Bayesian metoder Fremfor å bygge tillit kun på test resultater, kontrakter, eller tidligere erfaring kan man kombinere disse tre faktorene, og dette kalles bayesian statistikk.

Se foil 9-2 for dybtgående eksempel på bayes og sekvensiell testing med bayes.

Domenetesting

Mye feil, må derfor kontrolleres nøye.

Predikater

Antar at alle predikater er enkle. Dette innebærer at de bare inneholder en relasjonsoperator. Tillatte operatorer er $=$, $<>$, $>=$, $<=$, $<$ og $>$. Predikater som ikke er enfoldige kan deles i to eller flere enkle predikater.

Stibetingelse (path condition)

- Enhver sti gjennom et system defineres av en stibetingelse- forbindelsen av alle predikatene som man møter på denne stien.
- Et input traverserer en sti hvis og bare hvis stibetingelsen er oppfylt.

- Stibetingelsen definerer et stidomene som utgjør samlingen av alle input som medfører at stien blir utført.

Sti domener

- Et stidomene omgir seg med et grenseområde som består av en eller flere grenser der hver av de samsvarer med et predikat.
- En grense er * lukket hvis det er definert av et predikat som inneholder $=$, \leq eller \geq . Hører da til stidomenet. * åpen hvis det er definert av et predikat som inneholder $<$, $<$ eller $>$. Disse hører ikke til stidomenet.
- Lukket domene - det som ligger på linja hører til domenet.
- Åpent domene - det som er over eller under linja hører til domenet

Domenefeil

- Et domenefeil oppstår hvis et input traverserer feil sti gjennom programmet (f.eks. et spesifikt datainput forårsaker at programmet ekseverer en uønsket sti)
- Vi har ingen måte for å vite de riktige grensene på, og det finnes ingen riktig og unik versjon av programmet.
- Når domenefeil oppstår langs en sti, kan det tenkes at det blir fremkalt av en av de gitte grensene som er forskjellig fra den riktige grensen.

ON og OFF points

Test strategien er en strategi for å velge ON og OFF points som følger: * ON point for en * lukket grense som ligger PÅ grensen * åpen grense som ligger nær grensen og tilfredsstiller ulikhet relasjonen. * OFF point ligger nær grensen og på den åpne siden eller- alternativt- oppfyller ikke stibetingelsen assosiert med denne grensen.

Brukes som følger: * For å teste en lukket grense bruker vi: * To ON points for å identifisere grensen * En OFF point for å teste at den riktige grensa ikke ligger på den åpne siden av grensa * For å teste en åpen grense blir rollene på ON og OFF point byttet om. * I alle tilfeller er det viktig at OFF points er så nær ON points som mulig

Et nøyaktighetsproblem *Strategien som forklares krever at ON points ligger helt nøyaktig på grensen. For ethvert par av reelle tall, er det alltid et tredje reelt tall som ligger mellom dem. Dog, for en datamaskin er ikke dette tilfelle

på grunn av begrenset presisjon. Derfor eksisterer det grenser der ingen ON point kan bli representert i datamaskinen.

En forenklet strategi * Vi forkaster kravet om at grensen nøyaktig kan bli identifisert. Vi kan da også forkaste kravet om at ON point ligger nøyaktig på grensen. Dette fjerner nøyaktighetsproblemet. I tillegg kan vi redusere antall points til en per grense. Den eneste feilen som ikke vil bli oppdaget er hvis den ekte grensen passerer mellom et ON og et OFF point. Derfor må disse to ligge nær hverandre.

(For ikke-lineære grenser se forelesning 10.1, foil 20-23)

Når skal domenetesting brukes

Domene testing, slik som beskrevet her, krever at vi vet hvordan input blir delt opp i forskjellige input domener. Det er derfor bare mulig å bruke det på små kodesnutter.

Random testing

Beskrives som følger: 1. For hvert input parameter, generer en randomisert, men lovlig verdi. 2. Anvend hele input-settet til SUT (whaaaaat???)

Chens observasjon

- Input som ligger nær hverandre i inputdomenet har en tendens til å gå gjennom den samme stien. For å finne de fleste feil, bør man derfor spre test casene så mye som mulig. *Denne tilnærmingen kalles Adaptiv Random Testing (ART) og vi vil nå se på fire tilnærminger av denne:
 - Partisjon Adaptiv Random Testing
 - Basic Random Testing
 - Basic Adaptiv Random Testing
 - Mirror Adaptiv Random Testing (MART) (Litt vanskelig å skrive noe om disse fra forelesningsfoilene (masse bilder++), se forelesning 10.1.1 om Adaptive Random Testing)

Problemet med ART

Alle versjonene av ART krever et stort antall beregninger på grunn av fjerntliggende beregninger og sammeligninger. MART er derimot enklere og krever mindre beregninger.

MART prosedyren (fire steg)

1. Del in ut domenet i m atskilte (disjoint) subdomener, der en velges som kilde-subdomene. Resten er speil- subdomener.
2. Anvend D-ART prosessen for å generere den neste test casen fra kilde-subdomenet. Utfør dette test caset og avslutt hvis man finner en feil.
3. Anvend speil-funksjonen til test caset fra steg 2 for å generere et test case for hvert av de speilede subdomenene. Utfør test casene i sekvensiell rekkefølge og stopp når en mangel/ blir feil.
4. Gjenta steg 2 og 3 til man finner første feil eller til man når en stopp-betingelse.

(D-ART prosessen er foil 34 i forelesning 10.1.1 om Adaptive Random Testing)

Testdekning

Hva er testdekning?

La c være enten krav eller statements, da får vi at $C = (\text{enheter testet}) / (\text{antall enheter})$. Testdekning er altså andelen testede enheter blandt alle enheter.

Kategorier

Det er to hovedkategorier inn under testdekning:

- **Programbasert dekning:** Kategori relatert til testdekning på programkoden.
- **Spesifikasjonsbasert dekning:** Kategori relatert til testdekning på spesifikasjonene eller krav.

Forskjellige typer testdekning

Programbasert testdekning deles opp i tre grunnleggende kategorier:

- **Statement** Prosentandel av statements i koden som blir testet
- **Branch (gren)** Prosentandel av code branches som blir testet
- **Grunnleggende stidekning (basic path coverage)** Prosentandel av grunnleggende stier i koden som blir testet.

Begrenset anvendelighet

Når et testkriterium har begrenset anvendelighet, betyr det at kriteriumet kan bli tilfredstilt av et begrenset testsett.

Testdekningskriteriet i dette eksempelet har begrenset anvendelighet, ved at kriteriene relateres til bare gjennomførbar kode. Så når “all kode” nevnes senere i kapitlet, menes det bare gjennomførbar (feasible) kode.

Definisjoner

Uttrykksdekning (statement coverage) angir antall testede kodeuttrykk av alle kodeuttrykk i prosent.

Testdekning i code branches er antall testede stier i koden blandt alle mulige stier, i prosent.

Basis-stidekning går ut på at man finner et basissett av stier i koden, dvs. det minste settet av stier man trenger for å gå alle mulige veier gjennom koden.

Bruksområder

Det finnes mange måter å bruke dekningsverdiene på. Her vil vi se på to forskjellige bruksområder for testdekning:

- Som en akseptansetest
- For estimering av en eller flere kvalitetsfaktorer, f.eks. pålitelighet

Akseptansetest

På høyere nivå er følgende veldig enkle akseptansekriterier:

1. Kjør en test suite
2. Har man nådd akseptansekravet, f.eks. 95% branch-dekning?
 - Ja - stopp testing
 - Nei - skriv flere tester. Det kan være lønnsomt å ha et verktøy som viser hva som ikke er testet, slik at man lettere kan velge nye test cases

Hvis man måler testdekning som et akseptansekriterie, er det viktig å unngå redundans. Da blir det lettere å identifisere kode som ikke er testet, og dermed lettere å finne nye test cases.

Fault seeding

Konseptet går ut på at man:

1. Legger inn/“Sår” feil i koden
2. Kjører testsettet
3. En av to ting kan skje:
 - Alle innlagte feil blir oppdaget
 - En eller flere av de innlagte feilene blir ikke oppdaget

Når/hvis en eller flere av disse feilene ikke blir funnet av testen, viser det hvilke deler av koden som har til gode å bli testet. Denne informasjonen sørger for at man lettere kan definere nye test cases.

Det er et problem med fault seeding, og det er hvor og når man skal legge inn feil. Dette problemet har to løsninger:

- Lagre seed faults fra tidligere og bruke dem på nytt
- Bruk en erfaringsdatabase med vanlige feil og hvor de vanligvis opptrer, og lag en seed ut ifra denne databasen

Typer domenefeil

Closure

Oppstår dersom en grense er åpen, men intensjonen er at den skal være lukket og vice versa. Eksempler på dette er at den relasjonelle operatøren \leq er implementert som $<$.

Shifted-boundary

Den implementerte grensen er parallell med den intenterte grensa. Et eksempel på dette er at grensen $x+y > 4$ er implementert som $x+y > 5$.

Tilted-boundary

Feil verdier i den konstante koeffisienten i variabelen i predikatet. Et eksempel på dette er hvor grensen er $x+0,5*y > 5$, men skulle ha vært $x+y > 5$.

Grenseverditesting

Datapunkter på eller nær grenser er mer sensitive for domenefeil, det vil si, datapunktet faller i feil domene. Målet er å identifisere de datapunkter mest sensitive til domenefeil, slik at feil kan oppdages ved å eksaminere programvaren med disse input-verdiene. Basert på denne ideen vil vi definere to typer datapunk *ON* og *OFF*.

ON

Et punkt er *på* eller *nære ved* grensen om * et punkt kan bli valgt til å ligge akkurat på grensen, så velg det. Dette krever at `TODO *` en ulikhet fører til en tilnærmet løsning, velg et punkt veldig nære grensen.

OFF

Et punkt som ligger vekk utenfor grensen. Mens vi velger ett *OFF* punkt, må vi vurdere om grensen er åpen eller lukket mhp domenets interesse. * Åpen: et *OFF* punkt, men som er på innsiden av domenet med en liten avstand til grensen. * Lukket: et *OFF* punkt, men som er på utsiden av domenet med liten avstand til grensen.

Eksempel

Eksempel (Lukket): Anse domenet $D1$ med en grense $x + 7y \geq 6$. *Et OFF punkt ligger på utsiden av domenet. $(-1, 0.99)$ ligger på utsiden av $D1$.* Eksempel (Åpen): Anse domenet $D2$ som ligger på oversiden av $D1$ med en åpen grense $x + 7y < 6$. $(-1, 0.99)$ ligger på innsiden av $D2$.

Smidige krav gjennom brukerhistorier og -scenarioer

Hovedprinsippene bak smidige krav er: * Aktiv brukerinvolvering * Smidige team må ha mulighet til å gjøre valg * Krav dukker opp og utvikler seg sammen med programvaren som utvikles * Smidige krav er "såvidt tilstrekkelige" * Krav utvikles i små deler * Nok er nok – bruk 80/20-regelen * Samarbeid, samhandling og kommunikasjon mellom alle teammedlemmer er helt essensielt

Skriftlige krav er godt uttenkte, gjennomgåtte og redigerte. Varige. Enkelt å dele i grupper mennesker. Skriftlige krav tar imidlertid mye tid å produsere, kan bli mindre relevante over tid, og kan enkelt misforstås.

Muntlige krav gir en mulighet til øyeblikkelig tilbakemelding og avklaring, noe som kan gi en bedre felles forståelse. Informasjonsutvekslingen er fylt med informasjon. Når ny informasjon kommer er det mulighet for å bedre kunne justere krav. Kan utløse ideer om problemer og muligheter.

Brukerhistorier

User stories seek to combine the strengths of written and verbal communication, where possible supported by a picture. – Kent Beck

En brukerhistorie er en tekstlig beskrivelse av en hypotetisk bruker av systemet. Disse historiene beskriver og består av brukerens behov, beskrivelse av produktet, planleggingsenheter, tegn for samtale og mekanismer for samtaleutsettelse.

En brukerhistorie består av tre deler: en *beskrivelse*, en *samtale* og en *bekreftelse*. Beskrivelsen er en tekstlig beskrivelse av brukerhistorien for bruk til planlegging og som en “huskelapp”. Samtalen er en seksjon som skal fange opp mer informasjon om brukerhistoren og alle samtalers detaljer. Bekreftelsen er en seksjon dedikert til å formidle de tester som skal utføres for å bekrefte at brukerhistorie er komplett og fungerer som forventet.

Hvordan skrive en brukerhistorie

Det en trenger for å skrive en brukerhistorie er *hvem* (brukerens rolle), *hva* (et mål) og *hvorfor* (et rasjonale). Dette bidrar til å avklare hvorfor en funksjon er nyttig, kan påvirke en funksjons funksjon samt gi en gode ideer for andre nyttige funksjoner som kan bidra til å støtte brukerens mål.

As a [user role] I want to [goal] so I can [reason].

As a *registered user* I want to *log in* so I can *access subscriber-only content*.

1. Start med en tittel
2. Legg ved en konsis beskrivelse ved å bruke ovenstående templates
3. Legg ved andre relevante notater, spesifikasjoner og skisser
4. Skriv akseptansekriterier *før programvaren bygges*

En brukerhistorie er detaljert nok når den gjør teamet i stand til å starte arbeidet sitt, og etablerer flere detaljer og avklaringer ved utviklingstid.

Huskeregel: INVEST

- Independent - bør være så uavhengige av hverandre som mulig

- Negotiable - må være mulig å forhandle i løpet av planleggingsfasen
- Valuable - må være verdifulle for brukeren, skrives på brukerens språk, funksjoner ikke oppgaver
- Estimatable - må være mulig å estimere, må inneholde nok informasjon (men ikke for mye)
- Small - må være liten (men ikke for liten)
- Testable - må være skrevet slik at den er testbar

Brukerhistorier prioriteres i en *backlog* slik at de mest verdifulle oppføringene har høyest prioritet. Den samlede massen prioriterte brukerhistorier kalles en *produkt-backlog*.

User story mapping

En tilnærming for organisering og prioritering av brukerhistorier er *user story mapping*. Her vil en gjøre arbeidsflyten og verdikjeden synlig, vise relasjonen mellom større historier og deres barn, hjelpe til med å bekrefte backloggens fullstendighet, gi en nyttig prioriteringskontekst, planlegge utgivelser i komplette og verdifulle stykker funksjonalitet.

Her vil en arrangere aktiviteter og oppgavesentrerte historiekort romlig (spatially) for å kunne identifisere større historier. En legger ut disse kortene, fra venstre mot høyre, i den rekkefølgen en ville forklart de til en person som spør: Hva vil mennesker gjøre med systemet.

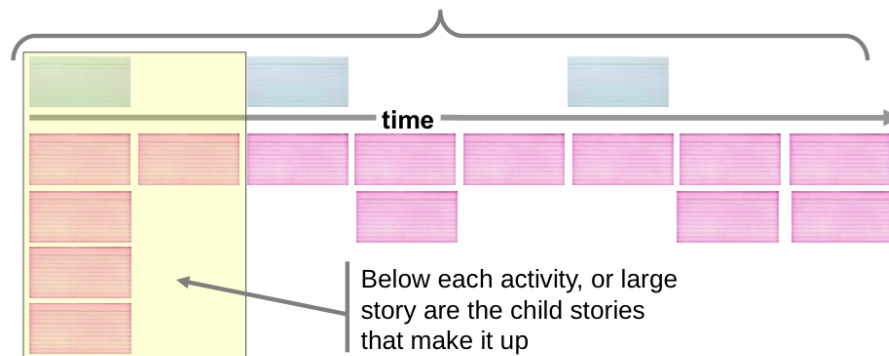
11/img/large-floor-map.jpg



Figure 20:

Brukeroppgaver overlapper vertikalt om en bruker vil kunne utføre en oppgave mer eller mindre på samme tid (tenk: flere funksjoner på én side). “Huskeregelen” her er at om en bruker “gjør X eller Y eller Z, deretter P”, vil alle tilfeller av “eller” være en vertikal stabling, “deretter” er horisontal.

Det resulterende kartet viser en dekomponering og typisk flyt gjennom hele systemet. Gjennom å lese aktivitetene over systemets topp vil en kunne bedre forstå end-to-end-bruken av systemet. 11/img/2.png



Videre vil en prioritere disse historiene basert på produktets mål. Dette målet beskriver utfallet eller nytten en organisasjon vil ha av å bruke produktet. Disse målene kan videre identifisere potensielle inkrementelle utgivelser, hvor hver enkelt utgivelse gir en form for nytte.

Del opp kartet inn i lag for å gruppere funksjoner inn i utgivelser. Arrangere så funksjonene vertikalt basert på nytte sett fra brukerens perspektiv. Del oppgaver inn i deler som kan utsettes til senere utgivelser. Benytt deretter produktmålene for å identifisere stykker som inkrementelt realiserer disse.

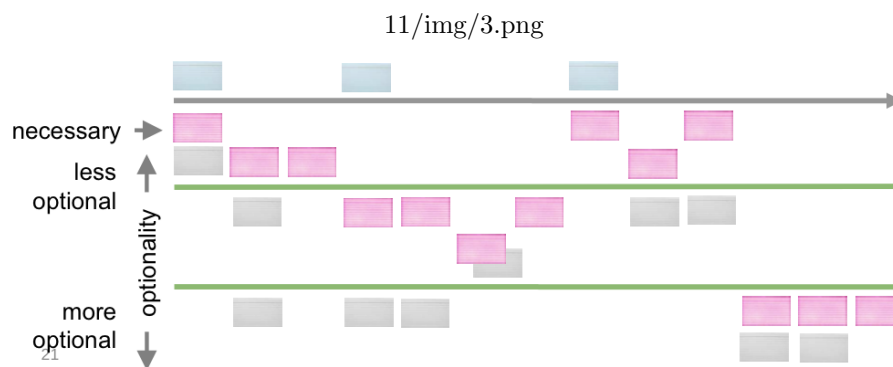


Figure 21:

Fra brukerhistorie til test-case

CUCUMBER

- Scenario - kort beskrivelse av testscenario
- Given - forutsetninger for testen
- When - input
- Then - output
- And - kan brukes for å inkludere mer enn én forutsetning, input eller output

Eksempel

- Scenarion - memory BIT
- When - we have inserted a memory fault
- And - run a memory BIT
- Then - the memory fault flag should be set
- And - the system should move to the error state

Utfordringer ved bruk av smidige krav

Aktive brukere kan være krevende, basert på brukerrepresentaters to. Krever og en mye forpliktelse gjennom hele prosjektets varighet.

Iterasjoner kan bidra til betydelig overhead dersom utgivelseskostnader er høye.

Smidige krav er så vidt tilstrekkelige som betyr at mindre informasjon tilgjengelig for nybegynnere i teamet om funksjoner og hvordan de skal fungere. (wut?)

Som regel vil ikke smidige krav være passende for prosjekter med høy utviklingsturnover med langtids-vedlikeholdskontrakter.

Ikke passende for sikkerhetskritiske systemer.

Oppsummering

- Brukerhistorier kombinerer skriftlig og verbal kommunikasjon, understøttet med bilde hvor det er mulig
- Brukerhistorer bør beskrive funksjoner som er av verdi til brukeren, skrevet på brukerens språk

- Brukerhistorer gir tilstrekkelig informasjon, men ikke mer
- Detaljer utsettes og samles opp gjennom samhandling JIT for utvikling
- Test-cases bør skrives før utviklingen starter, etter en brukerhistorie er skrevet
- Brukerhistorer bør være Independent, Negotiable, Valuable, Estimatable, Small og Testable.

Nyttige URLer

User story mapping

Advanced Use cases

Aktør : Eksterne parter som interagerer med systemet. Har en *rolle* (tilsvarer *ikke* jobbtittel). Denne rollen responderer til systemets spørringer. : En aktør behøver ikke være en person – det kan være et system eller et subsystem.

Use Case : En sekvens av handlinger som systemet utfører som resulterer i et observerbart resultat av en verdi til en aktør-

Use Case Model : Inneholder en aktørliste, pakker, diagrammer, use-cases og views. : Inkluderer strukturerte use-case-beskrivelse som er forankret i veldefinerte konsepter begrenset av krav og omfang.

Hvordan finne aktører

Viktige spørsmål en må spørre seg når en skal finne potensielle aktører er:

- Hvem bruker systemet?
- Hvem får informasjon fra systemet?
- Hvem gir informasjon til systemet?
- Hvilke andre systemer benytter seg av systemet?
- Hvem installerer, starter eller vedlikeholder systemet?

Ha først fokus på menneskelige og andre primæraktører. Grupper så individer etter felles oppgaver og systembruk – navngi og definér deres felles roller. Identifiser systemer som initierer interaksjoner med systemet, samt andre systemer som brukes for å utføre systemets oppgaver.

Hvordan finne use-cases

- Beskrive de funksjoner brukere ønsker fra systemet.
- Beskrive alle CRUD-operasjoner (operasjoner som skaper, leser, oppdaterer og sletter informasjon)
- Beskrive hvordan aktører varsles om endringer til den systemets interne tilstand.
- Beskrive hvordan aktører kommuniserer informasjon om hendelser som systemet må vite om.

Det å skape use-cases er en iterativ prosess hvor en involverer interessenter i hver operasjon. Det er svært sjeldent en klarer å skape de perfekte use-cases på første forsøk.

Det er vanlig å bruke UML-notasjon.

1. Definér aktører
 - Husk at også systemer kan defineres som aktører
 - Ta også med abuse-agenter
2. Definér nøkkelaktører
3. Definér use-cases for nøkkelaktørenes mål
4. Abstraher og gjenbruk av generiske handlinger som spesielle aktører kan arve handlinger fra.
 - Dependency (“Gjør en betaling” er avhengig av at “Logg inn” er utført)
 - Include (“Utfør en betaling” inkluderer “Validere midlers tilgjengelighet”)
 - Extends (“Gjør en periodisk betaling” og “Gjør en enkel betaling” extender “Gjør en betaling”)
 - Generalize
5. Legg til detaljer som *boundary object*, *control object* og *entity object*.

![Abstrahering av en generisk bruker]Forelesning 11/img/1.png)

Konvertering til sekvensdiagram

Use case index

Hvert use-case har flere attributter som relaterer til seg selv og til prosjektet. På prosjektnivå vil disse attributtene inkludere omfang, kompleksitet, status og prioritet

<i>Use Case ID</i>	<i>Use Case Name</i>	<i>Primary Actor</i>	<i>Scope</i>	<i>Complexity</i>	<i>Priority</i>
1	Places a bid	Buyer	In	High	1
2	Purchases an item	Buyer	In	High	1
3	Creates account	Generic User	In	Med	1
4	Searches listings	Generic User	In	Med	1
5	Provides feedback	Generic User	In	High	1
6	Creates an auction	Seller	In	High	1
7	Ships an item	Seller	In	High	2

Use-case-diagrammer er som regel lette å forstå og enkle å tegne. Mer komplekse diagrammer, av typen som bruker <> og <> kan være noe vanskeligere å forstå. Et use-case-diagram forteller heller ikke noe om en handlings sekvens.

Tekstlige use-cases

Use-case-element Beskrivelse Use-case-nummer ID som representerer use-caset Applikasjon Systemet eller applikasjonen denne hører til Use-case-navn Use-casets navn. Hold det konsist. Beskrivelse Utbrodere navnet, litt mer fyldig. Primæraktør Hovedaktøren som dette use-caset representerer Forutsetning Forutsetninger som må være tilstede for at dette use-caset kan starte Trigger Det som trigger dette use-caset Grunnleggende flyt Perfekt gjennomføring, uten feil. Alternativ flyt De mest signifikante alternativer og unntak fra den grunnleggende flyten Eksempler på alternativ flyt involverer at en kundes kredittkort ikke godkjennes, session timeout. Alternativ flyt kan håndteres av <> og <>.

Sette inn eksempler her? Bah, mer tabeller :/

Komplekse tekstlige use-cases er enklere å forstå enn komplekse UML-diagrammer, og kan og være enklere å oversette til sekvensdiagrammer da de viser sekvensen til en handling. På den andre siden tar de mer tidkrevende å produsere.

Mis-Use cases

Brukes for å identifisere mulige misuses av systemet. Brukes på samme måte som et ordinært use-case, men med markerte fiendtlige aktører. Angrepsvektorer markeres og med *threatens* fra fiendtlig handling til ordinær handling. Fiendtlige aktører kan være på grunn av utilsiktede feil fra ordinære brukere og fra kuk i computeren.

UC-name Respond to over-pressure User action System response Threats Mitigations Alarm operator o high pressure System fails to set alarm; Operator fails to notice alarm Have two independent alarms; Test alarms regularly; Use both audio and visual cues; Alarm also outside control room Operator gives command to empty tank Operator fails to react (e.g. ill, unconscious); Operator gives wrong command (e.g. filling tank) Alarm backup operator; Automatic sanity check; Diallow filling at high pressure System opens valve to sewer System fails to relay command to valve; Valve is stuck Operator reads pressure Operator misreads and stops emptying too soon Maintain alarm until situation is normal

Hvorfor misuse case?

Et misuse-case brukes på tre måter:

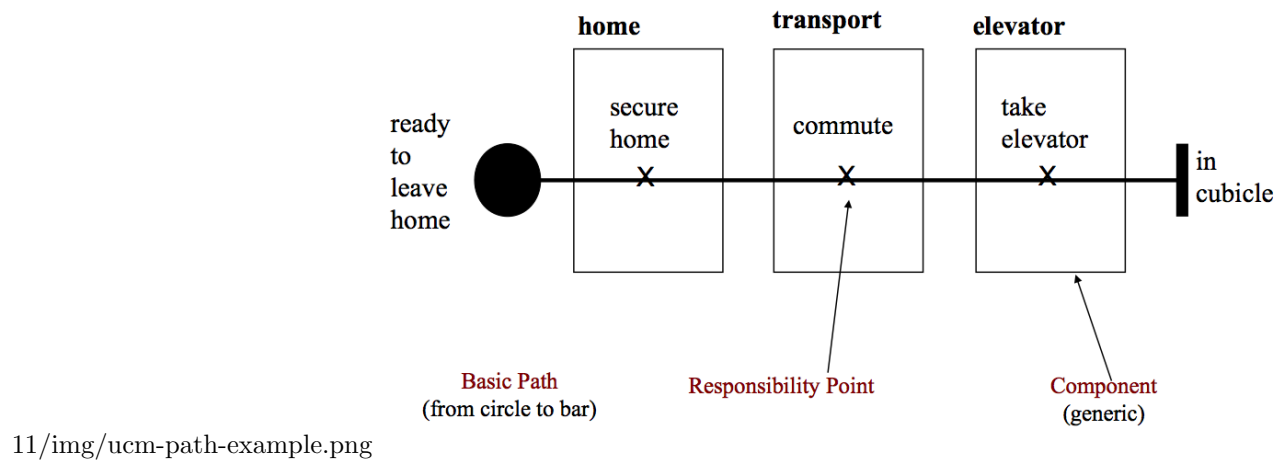
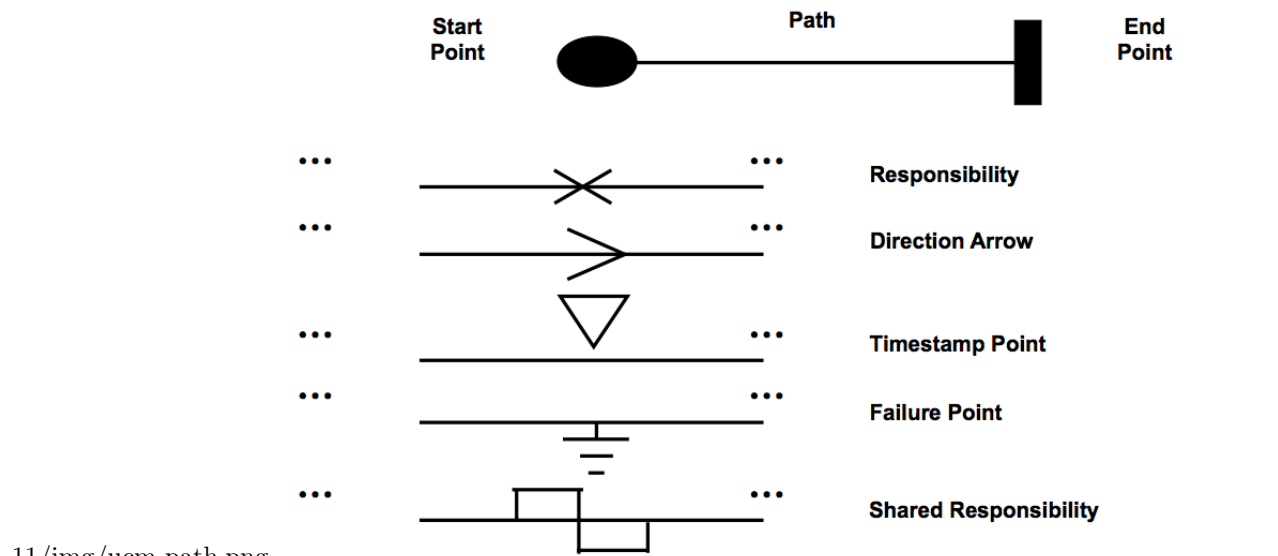
- Identifisering av trusler
- Identifisering av nye krav
- Identifisering av nye tester

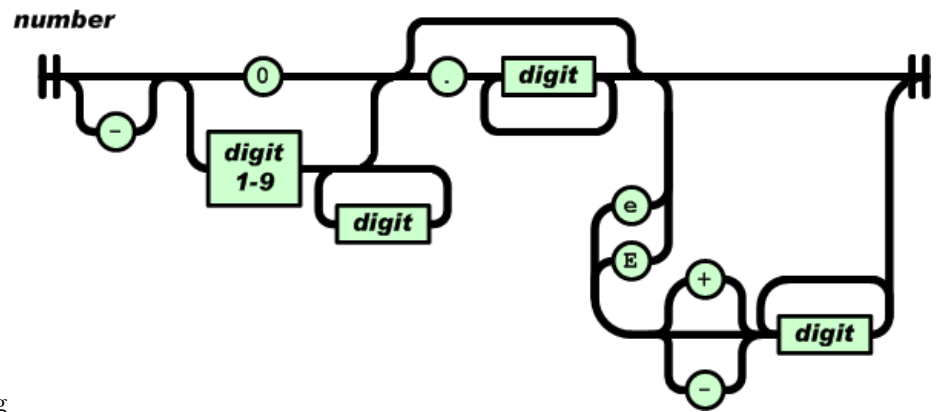
Fordeler med misuse-cases inkluderer muligheten til å fokusere på mulige problemer, og å se på mulige forsvars- og mitigasjonsstrategier. Misusediagrammer kan bli noe komplekse.

Use-case maps

Use case map : En visuell representasjon av et systems krav ved å bruke et presist definert sett av symboler for ansvar, systemkomponenter og sekvenser. : Lenker sammen oppførsel og struktur på en eksplisitt og visuell måte.

Use case map path : Arkitekturelle entiteter som beskriver uformell relasjon mellom ansvar, som bundet til underliggende organisatoriske strukturer av abstrakte komponenter. : Brukes for å skape en kobling mellom krav og detaljert design.





11/img/number.png

Cucumber – Cucumber testing

... cue eksempler i massemassevis

Hohoy denne notasjonen var full av fuck

Testdrevet utvikling

Fra et TDD-ståsted er det urimelig å separere testing og implementasjon. Når en implementasjonsstrategi er bestemt, har en og indirekte valgt en teststrategi.

TDD oppfordrer til å skrive tydelige krav. Dele opp utviklingen i mindre steg som kan testes separat. Minimalistisk kode, samt YAGNI (“You ain’t gonna need it”). En bør kun skrive kode som oppfyller kravet (fullfører testen), verken mer eller mindre.

Legg merke til at TDD ikke krever smidig utvikling. Det legges vekt på at testing og implementering er to parallelle aktiviteter.

Testing for greenfield-prosjekter

In many disciplines a greenfield is a project that lacks any constraints imposed by prior work. The analogy is to that of construction on greenfield land where there is no need to remodel or demolish an existing structure. – Wikipedia

I slike prosjekter står en i prinsippet fritt til å selv velge programmeringsspråk, arkitektur, utviklingsmetode og detaljert design. I realiteten vil dog company policy, tilgjengelige verktøy, samt egen kunnskap og erfaring være ting som setter begrensninger.

Detaljer eller det “store bildet”

En må velge mellom å starte med å løse alle små problemer for så å kombinere de til en komponent, eller en kan løse design problemet først for så å inkludere alle detaljer.

I de fleste tilfeller vil vi først starte med å skrive tester, og deretter kode etter bekymringer.

Usikkert terreng eller det kjente

Om en utforsker ukjent terreng vil en så godt en kan forsøke å redusere risiko. Det kjente vil på den andre siden gi en større fordeler raskere.

Se på dette som et kost/nytte-problem. Skriv tester først, kode deretter på en måte som gir en best kost/nytte-ratio.

Hovedprinsippet er å utsette viktige beslutninger så lenge som mulig – *men aldri lenger*. I mellomtiden vil en samle informasjon som gjør en i stand til å møte utfordringene så godt forberedt som mulig.

Highest value vs. the low-hanging fruits

Også et kost/nytte-problem.

Om en går for de lavthengende fruktene vil en raskt og billig kunne demonstrere mye funksjonalitet tidlig i prosjektet. Det er også lettere å både kode og teste slike alternativer.

Happy paths vs. error situations

Selv om krav til robusthet er ekstreme vil en først behøve å teste og kode noe som er av nytte for brukeren. Feilhåndtering kommer (som regel) etterpå.

Unntak til denne regelen er situasjoner hvor en behøver feilhåndtering tidlig, som for eksempel i innloggings-funksjoner. Her er kravet til feilhåndtering vedt så tett inn at det må taes med én gang.

Essensielle TDD-konsepter

Fixtures : Sett med objekter som vi har instansiert for å bruk i testene : En fixture er et stykke kode vi ønsker å teste

Test doubles : “*Object stand-in*”. Ser ekte ut fra utsiden, men kjører fortere og er enklere å utvikle og vedlikeholde. : Brukes for to typer testing: tilstandsbasert testing og interaksjonsbasert testing. : *stubs, fakes, mocks*.

Test doubles brukes for å teste et objekt uten å måtte skrive all kode som objektet skal benytte seg av. Dette gjør en i stand til å etablere en stegvis implementasjon etter hver som en suksessivt erstatter stadig flere doubles. Om en implementerer doubles korrekt kan en være sikker på at eventuelle problemer i koden ikke stammer fra miljøet, men fra det som testes.

Tilstandsbasert testing : Brukes for å teste hvor godt et objekt lytter. Responderes det korrekt på input?

Interaksjonsbasert testing : Brukes for å verifisere hvordan et objekt snakker med sine kollaboratører.

Retningslinjer for et testable design

- **Unngå komplekse private metoder**

Det er vanskelig å teste private metoder direkte, men om det er mulig kan disse testes indirekte.

- **Unngå final metoder**

Veldig få programmer trenger final metoder, så oddsene er at du også ikke trenger det.

- **Unngå static metoder**

De fleste metoder skal IKKE være static, men enkelte har hatt en praksis at utility classes full av static metoder fordi metodene ikke skal være relevante til enkelte objekter.

- **Bruk “new” med forsiktighet**

Bruk av new er den vanligste formen for hardkoding, hver gang new skrives så opprettes et nytt objekt. Derfor burde vi bare instansiere objekter som vi ikke vil subsidiere med [test doubles](#).

- **Unngå logikk i constructors**

Constructors er nærmest umulig å unngå siden subklassers constructor vil alltid kalle på minst en superklasse constructor. Derfor burde vi unngå test kritisk kode eller logikk i constructors.

- **Unngå Singleton**

Bruk av Singleton skaper problemer for testing, særlig da det skal kjøres tester på kode som benytter seg av en singleton klasse. Da må denne være inkludert i testen for at resultatet skal være gyldig, dette er ikke ønskelig da vi ønsker å teste så lav nivå som overhode mulig.

- **Komposisjon fremfor arv**

Arv er greit for polymorphism, men om det gjøres kun for gjenbruk av funksjonalitet så er det ofte bedre å benytte seg av komposisjon; ved å benytte seg av et annet objekt istedenfor å arve fra klassen dens.

- **Pakk inn eksterne bibliotek**

Det er viktig å ikke bare slenge inn kall til eksterne bibliotek i koden din, men tenke deg om før du gjør dette. Vi har allerede tatt for oss arv, men dette er verre da du heller ikke har kontroll på koden du arver eller kaller direkte på.

- **Unngå tjeneste kall**

De fleste tjeneste kall (som å kalle på en Singleton instanse) er

Legacy code Det meste av utvikling er endringer på eksisterende kode. [TDD](#) forventer at vi skriver tester, så kode som tilfredstiller denne testen. Denne fremgangsmåten er ikke tilfredstillenden for Legacy code, da koden allerede eksisterer. Derfor er følgende foretrukket istedet:

1. Identifiser hvor det må gjøres endringer.
2. Identifiser et vendepunkt.
3. Dekk identifiserte vendepunkt.
 - Fjern interne avhengigheter.
 - Fjern eksterne avhengigheter.
 - Skriv tester
4. Gjør endringer
5. Refactor dekket koden

Enhetstesting-patterns

Assertion `assertEqual(java.lang.String expected, java.lang.String actual)`
throws `AssertionFailedError`

- `assertFalse(...)`
- `assertTrue(...)`
- `assertEquals(...)`
- `assertNotEquals(...)`
- `assertNotSame(...)`
 - kontrollerer at to objekter ikke refererer til det samme objektet
- `fail(...)`

En assertion bør plasseres så nært den nye koden som mulig, samt rett etter hver nye vikige kodesnutt. Intensjonen er å oppdage feil så raskt som mulig for å kunne redusere debugging-innsats.

Det er svært viktig å oppdage feil så raskt som mulig, da en da har mindre kode å gå gjennom for å lokalisere den. For å oppnå dette trenger vi assertions der hvor vi trenger å kontrollere at vi er på riktig kurs.

Assertionens type avhenger av dens bruk. De viktigste typene er:

Resulting state assertion

1. Eksekvér en funksjonalitet
2. Kontrollér at den resulterende tilstanden er som forventet ved bruk av en eller flere assertions.

Guard assertion Brukes for å kontrollere at våre forutsetninger om status før eksekvering av ny kode er korrekte.

```
assertTrue(...)
```

Ny kode for testing

```
assertNotSame(...)
```

Delta assertion Istedet for å kontrollere absolutte verdier kontrollér forandringer i verdien. Dette gjør at en ikke trenger et “magisk nummer” i testen, og testen vil være mer robust i forhold til forandringer i koden.

Custom assertion Dekker et bredt spekter tilpassede assertion-patterns. Kalles gjerne “fuzzy patterns”. Benytter seg av mer komplekse predikater og uttrykk (`custom_assertInRange(...)`).

Disse må skreddersys til den assertion som skal gjøres, men gjør en i stand til å dekke et bredt område viktige testsituasjoner. Over tid vil en kunne skape et bibliotek assertions som kan gjenbrukes.

Interaction assertion Kontrollerer ikke at koden fungerer som forventet, men at den samarbeider med våre kollaboratører som forventet (COTS) basert på foreliggende dokumentasjon. Benytter seg av de samme asserts diskutert over.

Ta vare på eller forkaste?

Skal en fjerne assertions fra produksjonskode? Avhenger av patterns. Delta og custom er mer robuste for forandringer i koden.

Testing for forandring av legacy-kode

Mesteparten av utvikling gjort vil forholde seg til foreliggende legacy-kode.

Prosess

1. Identifisere hvor i koden det skal forandres
2. Identifisere et inflection-punkt
3. Dekke identifisert inflection-punkt
 - Brekk interne avhengigheter
 - Brekk eksterne avhengigheter
 - Skriv tester
4. Gjør forandringer
5. Refaktorere dekket kode

Inflection-punkt er et punkt downstream fra forandringspunkt hvor en kan oppdage relevante forandringer i koden. Vi foretrekker inflection-punkter så nært som mulig til forandringspunktet.

Etter hver som vi forandrer legacy-kode vil vi bruke både karakteriseringstester og ny-kode-tester. Førstnevnte for å kontrollere at vi ikke bryr noe av oppførsel vi ønsker å beholde. Sistnevnte for å kontrollere at de forandringer vi gjør har ønsket effekt.

TDD og akseptansetesting

Valg av user story

Prioritering i forhold til forretningsverdi og teknisk risiko, om der ikke foreligger en prioritering fra før.

$$\text{Leverage} = (\text{Business value} - \text{Tech. risk}) / \text{Tech. risk}$$

Ellers vil en på dette stadiet ha en god innsikt i brukerens krav og behov gjennom uformelle diskusjoner o.l.

12/img/1.png

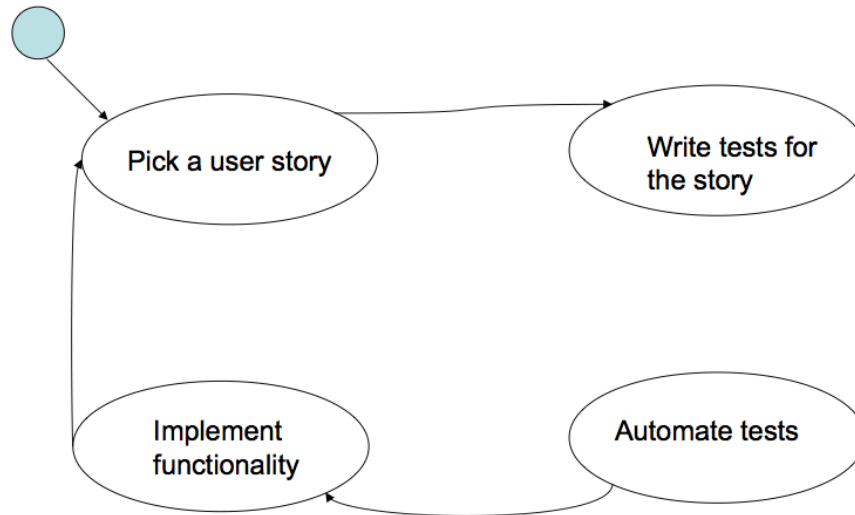


Figure 22:

Skrive tester for historien

Sitt sammen med kunde og beskriv de viktigste scenarioer for denne user-storien. Kom så til en avtale om hva som behøves implementert for å kunne realisere hvert scenario.

Resultatene fra disse samtalene vil være et sett godt definerte tester og forventede resultater (akseptansekriterier). Sistnevnte inkluderer forventede resultater og systemets forventede tilstand etter endt test.

Automasjon av testene

Tabellrepresentasjon (ala (FITNESS)[#fitness] brukes mye. Det en da trenger er:

- Funksjonens ID
- Input-parametre
- Forventede resultater

Pro TDD

- Forbedre refaktoring. Endre strukturen, ikke oppførselen
- Gjør ditt beste for å oppnå høy dekningsgrad på testene (Strive for high test coverage)
- Skriv små snutter av både kode og tester
- Fokus på manglende funksjonalitet- skriv tester som feiler

Regresjonstesting

Regresjonstesting er testing gjort for å kontrollere at en systemoppdatering ikke reintroduserer feil som har blitt fikset tidligere. Dette gjøres via black-box-testing (for å teste funksjonalitet) og grey-box-testing (for å teste arkitekturen).

Ettersom regresjonstester er ment å teste all funksjonalitet og alle forandringer, vil slike tester være store. Av den grunn må regresjonstesting eksekveres og kontrolleres automatisk.

Automatisk regresjonstesting

Parameterisering av testresultater. I stedet for å bruke en komplett output-fil som orakel vil vi bruke et verktøy for å ekstrahere *relevant* data fra filen. Disse dataene vil vi så bruke sammen med de data som er lagret i orakelet.

Da en regresjonstest er stor vil det alltid være et behov for å identifisere de deler av en regresjonstest som behøves å kjøre etter en forandring. Traceability er derfor viktig av to grunner: mindre eksekveringstid; vite hvilke tester som må forandres når funksjonalitet endres.

Implementerte regresjonstester må holdes oppdatert hver gang koden oppdateres. Skapes det varianter av systemet må der også skapes parallelle varianter av regresjonstest-suiten.

Bug fixing

1. Rapportering av bug
2. Reprodusering av bug, helst med en så enkel input-sekvens som mulig
3. Lag en ny oppføring i regresjonstesten, sammen med
 - Rapportert inputsekvens (realistisk, men kan være stor)
 - Enklest mulig inputsekvens (enkel å forstå, men er kunstig)

Firewall for regresjonstesting

Her brukes konseptet om en firewall for å redusere settet klasser eller komponenter som behøves testes. Dette da de fleste regresjonsteseter er store og tidkrevende.

En firewall i denne sammenhengen er en separator mellom de klasser som beror på en klasse som endres fra resten.

Det finnes to sentrale konsepter, Dependency og encoding.

Enkle regler for fire wall i et objektorientert system

1. Gitt to suksessive versjoner av et objektorientert system, identifiser klasser som har blitt endret.
2. Dersom en klasse er en del av et arvshierarki må en også anse etterkomere av den endrede klassen som endret.
3. For hver endrede klasse, identifiser alle klasser som sender eller mottar meldinger til/fra den endrede klassen og inkluder de inne i firewallen.

regel 4 og 5 identifiserer en utvidet fire wall

4. Identifiser alle datastier til og fra den endrede klassen
5. Finn alle eksterne klasser i den modifiserte klassens omfang og inkluder de i den utvidede firewallen

Dependency

To viktige spørsmål kommer opp her

1. Benytter komponenten seg vanlige verdier av inputen?
2. Må komponenten gjøre rede for hvordan denne inputen ble generert?

Noen grunnregler for disse testee er at det bare skal bli brukt for å teste oppdateringer og endringer. Det antas også at det innledende testsettet og softwaren som er testet er av høy kvalitet.

IKKE FERDIG

15/img/firewall.png

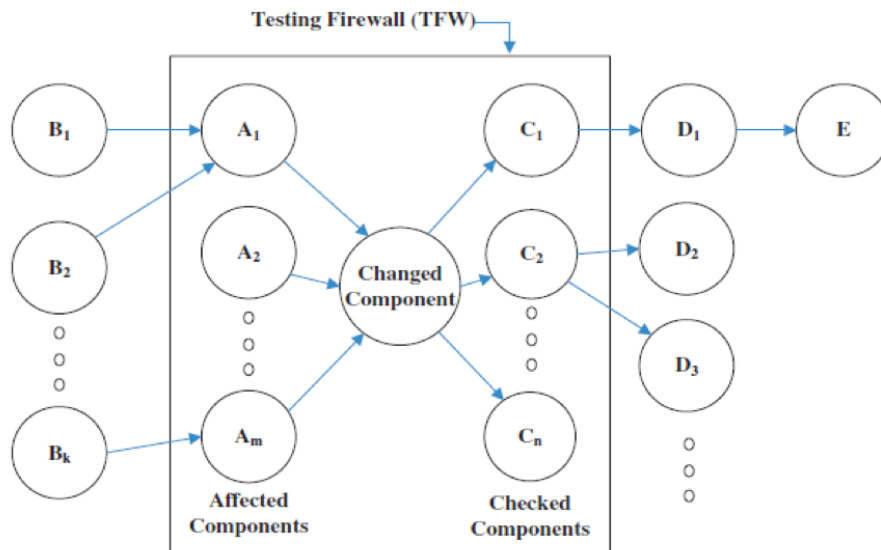


Figure 23: Testing firewall (TFW)

Ikke-funksjonelle krav

Standardisert i ISO-9126 (og mange andre steder) – en *faktor-kriterie-metrikk*-modell. Nært koblet til “quality in use” – dvs. brukernes erfaring av systemet i bruk. Brukernes erfaringer er subjektive og derfor vil og kvalitetsfaktorer være subjektive.

Functionality

The capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions.

Suitability

Programvarens evne til å gi et hensiktsmessig sett funksjoner for spesifiserte oppgaver og brukermål.

Accuracy

... to provide the right or agreed.

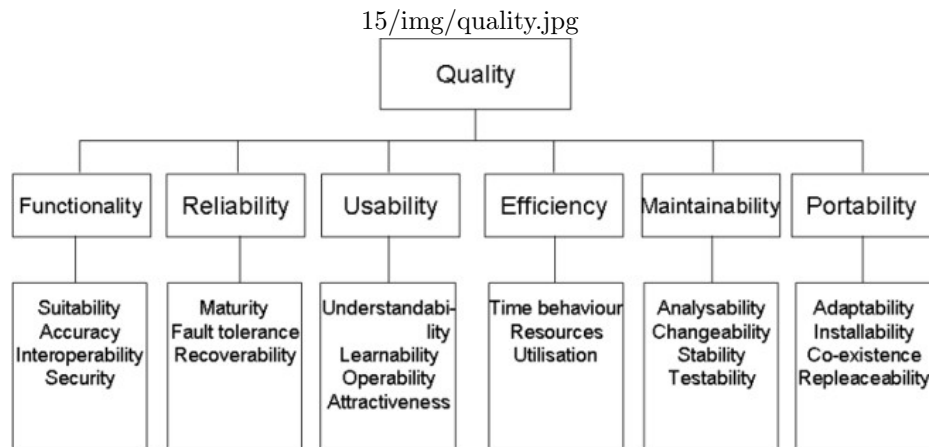


Figure 24: Et nesten riktig diagram

Interoperability

Evne til å interagere med et eller flere spesifiserte systemer.

Compliance

Evne til å overholde applikasjonsrelaterte standarder, konvensjoner og reguleringer i forhold til lover og forskrifter.

Security

Forhindre uønsket adgang, motstå angrep, uønskede modifikasjoner til informasjon, gi angriper en mulighet til å forhindre adgang til legitime brukere (DOS).

Reliability

The capability of the software to maintain the level of performance of the system when used under specified conditions.

Merk at slitasje og aldring ikke er med i denne beregningen, men kun begrensninger gitt i kraft av mangler i krav, design og implementasjon.

Maturity

Unngå failure som et resultat av faults.

Fault tolerance

Opprettholde et spesifisert ytelsesnivå i tilfeller med store faults, eller krenkelser av spesifisert grensesnitt.

Recoverability

Reetablering av ytelsesnivå, gjenopprette data rett etter failure.

Availability

Evne til å være i en fungerende tilstand til å utføre sine funksjoner på et gitt tidspunkt, under gitte bruksforhold.

Usability

The capability of the software to be understood, learned, used and liked by the user, when used under specified conditions.

Understandability

Gjøre en bruker i stand til å forstå om programvaren er passende og hvordan den kan brukes for særlige oppgaver og bruksforhold.

Learnability

Gjør bruker i stand til å lære sin applikasjon.

Operability

Gjøre bruker i stand til å operere og kontrollere seg.

Likeability

Applikasjonens evne til å bli likt av en bruker.

Efficiency

The capability of the software to provide the required performance relative to the amount of resources used, under stated conditions.

Time behaviour

Gi hensiktsmessige respons- og prosesseringstider og throughput-rater når den utfører sine funksjoner, under gitte forhold.

Resource utilisation

Bruke hensiktsmessige ressurser (annen programvare, HW, materialer) på hensiktsmessig tidspunkt når den utfører sine funksjoner, under gitte forhold.

Maintainability

The capability of the software to be modified (korrigeringer, forbedringer, adapteringer).

Changeability

Evne til å muliggjøre implementasjon av en spesifisert modifikasjon.

Stability

Minimisere ønskede bieffekter av modifikasjon. (ripple-effekt)

Testability

Muliggjøre testing av modifisert programvare.

Portability

The capability of software to be transferred from one environment (organisational, HW, SW env) to another.

Adaptability

Evne til å bli modifisert til et annet spesifisert miljø uten å gjøre noe annet enn de handlinger som applikasjonen tillater.

Installability

Evne til å bli installert i et gitt miljø.

Co-existence

Evne til å sameksistere med annen uavhengig programvare i et felles miljø med deling av felles ressurser.

Conformance

Evne til å overholde standarder og konvensjoner relatert til portability.

Replaceability

Evne til å bli brukt i sted for annen spesifisert programvare dets miljø.

Setting av krav

Minst tre måter:

- Brukernivå – hvordan systemet oppfører seg
- Possessnivå – hvordan produktet utvikles
- Metrikknivå – hvordan programvaren er

For å ha testbare kriterier må man gå ned på kriterienivå.

MbO - Management by Objectives

Starter med krav (systemet skal være enkelt å vedlikeholde). Spør så *“hva mener du med ...”* til du har noe observerbart og testbart. I slike tilfeller må en ha en bruker som kan delta i testen på en eller annen måte. Der vil nå være en sterk kobling mellom krav og test – i mange tilfeller vil kravet være testen.

Det er relativt enkelt å age krav som omhandler reliability, maintainability og andre “objektive” ikke-funksjonelle krav på grunn av at de er målbare. Subjektive faktorer som f.eks. usability er vanskeligere. Men bruk MbO, skap tall der det er mulig (skal kunne læres <1 uke).

Testen sier mer om kravet enn det kravet i seg selv sier.

Scenario testing

Der finnes to typer scenariotesting, *type 1* hvor scenarioer benyttes for å definere input/output-sekvenser, og *type 2* hvor scenarioer brukes som et skript for sekvenser realistiske handlinger i ekte eller simulerte omgivelser.

Mye av det en behøver for å skrive gode krav må brukes for å skrive gode scenarioer. En av årsakene bak at scenario-testing er så effektivt kan være at en i effekt går gjennom kravprosessen nok en gang, men involverer andre mennesker.

Noen regler for å skrive gode scenarioer

- List opp mulige brukere
 - Analyse av mål og interesser
- List opp systemhendelser
 - Definerings/analyse av hvordan systemet håndterer dette
- List opp spesielle hendelser
 - Hvordan imøtekommer systemet disse hendelsene?
- List opp nytte
 - Lag en fremgangsmåte for hvordan en skal oppnå disse
- Jobb sammen med bruker
 - Studér hvordan de arbeider, hva de gjør
- Les deg opp hva systemer av denne typen er ment å gjøre
- Lag en liksomforretning
 - Mat systemet med realistiske data.

Brukere

De som vil bruke systemet som er under utvikling.

List opp mulige brukere

For hver identifiserte bruker - identifiser interesser. Fokuser på én interesse om gangen og identifiser brukerens mål.

Finn så ut hvordan man best kan teste at hvert av målene er enkle å oppnå med programvaren.

List opp systemhendelser

Hendelse : Enhvert tilfelle som systemet er ment å svare på.

For hver hendelse må vi forstå: dets *formål*; hva systemet er ment å gjøre med den; og regler relatert til hendelsen.

List opp spesielle hendelser

Forutsigbare, men uvanlige. Disse behøver spesiell håndtering.

Da disse behøver spesielle omstendigheter for å trigges må man sørge for at scenarioene inkluderer de viktigste av disse omstendighetene.

List opp nytte

Hvilken nytte skal systemet gi brukerne? Dette må interessentene svare på. Her er det viktig å passe på eventuelle misforståelser og konflikter.

Jobb sammen med bruker

Det er svært viktig å jobbe sammen med systemets framtidige brukere. Her kan en finne informasjon om hvordan de utfører sitt arbeide som gir en peker om hvordan systemet best kan oppfylle de mål og ønsker en bruker har. Se spesielt etter dagens arbeidsmønster og hva de har problemer med.

Her vil en finne gode ideer til hvordan utforme scenarioer.

Les om denne typen systemer

Før en skriver scenarioer er det viktig å ha kunnskap om hva det nye systemet skal gjøre, samt ha et innblikk i hva lignende systemer gjør. Dette gir kunnskap om hva brukeren kan forvente av ens system. Denne kunnskapen finner en gjerne i bøker, manualer, etc.

Lag en liksomforretning

Dette krever en god del kunnskap om hvordan forretningen fungerer. Det er viktig at liksomforretningen er realistisk for å kunne få best mulig resultat av øvelsen, og det kan være nødvendig å hente inn eksterne konsulenter. Tar mye ressurser, men kan gi veldig verdifulle resultater.

Risiko

Passer ikke til testing av ny kode, da bugs i implementasjonen kan føre til forsinkelser da testen må utsettes til disse er fikset. Derfor bør scenariotesting kun brukes som en akseptansetest.

Scenarier skal dekke alle systemets funksjonaliteter, ikke kodedekning. Avdekker designfeil heller enn kodefeil, og bør derfor ikke brukes den til regresjonstesting, eller for testing av nye fixer.

Type 1 scenariotesting

Bruker scenarier for å skrive transaksjoner som sekvenser av input og forventet output. Resultatet kan være et ekstremt detaljert tekstlig usecase.

Type 2 scenariotesting

Brukes om en vil ha realisme. Her vil en teste hvordan systemet vil oppføre seg i de real-world-situasjoner beskrevet av scenarioene, med reelle brukere gitt av produkteier og (om nødvendig) med reelle kunder.

En type 2 scenariotest utføres på følgende måte:

1. Omgivelser settes opp i ifølge scenariobeskrivelsen. Kunder blir instruert i deres oppgaver.
2. En person – game master – leser hvert steg av scenarioene
3. Brukere og kunder reagerer på situasjonene GM skaper
4. Hendelsene som resulterer fra hvert scenario dokumenteres (f.eks. på video) for senere analyser

Om det er for mange scenarier (slik det ofte er), og en må prioritere vil de scenarier som involverer en sterk interaksjon med systemets omgivelser (brukere, kunder, nettverk, filservere, stressende arbeidssituasjon etc.) være de mest effektive.

Formell inspeksjon

Den formelle kodeinspeksjonen består av fem deler:

- Planlegging
- Kick-off

- Individuell kontroll
- Loggføring
- Redigering og oppfølging

Planlegging

Hvem skal delta i inspeksjonen? Hvem er interessert? Hvem har tid? Hvem har kunnskap om språk, applikasjon, verktøy og metoder?

Verdiområde: 3-5% Typisk verdi: 4%

Kick-off

Distribuerer av nødvendige dokumenter er viktig her. Relevante dokumenter er de dokumenter som skal inspiseres, krav, samt relevante standarder og sjekklister. Andre ting som skal med her er rolle- og arbeidsfordeling. En vil også sette mål for ressursbruk, deadlines, etc.

Verdiområde: 4-7% Typisk verdi: 6%

Individuell kontroll

Individuell kontroll er hovedaktiviteten ved inspeksjonen. Hver av inspeksjonens deltakere leser gjennom alle relevante dokumenter for å lete etter:

- Potensielle feil
 - Inkonsistens med kravsspesifikasjon eller felles applikasjonsopplevelse.
- Mangel på etterlevelse i forhold til firmas standarder, eller god arbeidsskikk

Verdiområde: 20-30% Typisk verdi: ?

Loggføring

En ønsker å loggføre de feil som de enkelte deltakere finner, feil som oppdages gjennom diskusjoner og den faktiske loggføringen (eks. mismatch mellom deltakers oppfattelse av applikasjonen), samt vil en identifisere potensielle forbedringer til inspeksjons- og/eller utviklingsprosessen.

Verdiområde: 20-30% Typisk verdi: 25%

Redigering og oppfølging

Alle loggens oppføringer kategoriseres som enten:

- Feil i forfatters dokument
 - Gjør nødvendige korrigeringer
- Feil i andres dokument
 - Informere dokumentets eier
- Misforståelser innad i inspeksjonsteamet
 - Forbedre dokument for å forhindre fremtidige misforståelser

Inspeksjonsleder er ansvarlig for at hver oppføring i loggen gjøres noe med på en tilstrekkelig god nok måte.

Verdiområde: 15-30 Typisk verdi: 20

Fordeler og ulemper ved formell inspeksjon

- Fordeler
 - Kan brukes for å formelt akseptere dokumenter
 - Inkluderer prosessforbedringer
- Ulemper
 - Krever mye tid
 - Dyrt
 - Behøver ekstensiv planlegging for å lykkes

Testing

- Enhetstesting
- Systemtesting (aka. verifikasjonstesting av funksjoner)
- Akseptansetesting (aka. verifikasjonstesting av systemet)

Enhetstesting

Fordeler og ulemper

- Fordeler:
 - Enkel måte å kontrollere at kode fungerer som den skal
 - Kan iterativt bli til sammen med kode
- Ulemper
 - Tester kun utviklerens forståelse av spesifikasjon
 - Kan kreve stubber eller drivere for å kunne teste

Systemtesting

1. Basert på krav, identifisere
 - Tester for hvert enkelt krav, inklusiv feilhåndtering
 - Initiell tilstand, forventet resultat og slutttilstand
2. Identifisere avhengigheter mellom tester
3. Identifisere akseptansekriterier for hver test-suite
4. Utføre tester og kontrollere resultater opp mot
 - Akseptansekriterier for hver test
 - Akseptansekriterier for hele test-suiten

Fordeler og ulemper

- Fordeler
 - Tester systemets oppførsel opp mot kundes krav
- Ulemper
 - Black-box. Finner man en feil må der ekstensiv debugging til å for å finne hvor feilen oppstår.

Akseptansetesting

Akseptansetesting har som regel tre aktiviteter, og involverer både kunde eller dens representanter:

- Utføre systemtesting på kundens område
- Bruke systemet for å utføre realistiske oppgaver
- Forsøke å brette systemet gjennom bruk av store mengder data og ulovlig inndata.

Fordeler og ulemper

- Fordeler
 - Skaper en tillit til systemet er brukbart for kunde
 - Viser systemets evne til å operere i kundes miljø
- Ulemper
 - Kan tvinge systemet til å handle inndata som det ikke var designet for, og dermed skape et ufordelaktig inntrykk.

Del 2

Testing og inspeksjon - En kort dataanalyse

To viktige termer: defekt-**typer** og -**termer**

Defekt-kategorier

Åtte ulike defektkategorier: * Feil eller manglende data * Feil eller manglende datavalidering * Feil i algoritme * Feil timing eller sekvensiering * Interfaceproblemer * Funksjonelle feil * Bygging-, pakking- eller sammenslåingsproblemer * Dokumentasjonsproblemer

Inspeksjonstriggere

- Konformitet til design
- Forståelse av detaljer
 - Operasjon og semantikk

- Bivirkninger
- Konkurrens
- Bakoverkompabilitet
- Lateral kompabilitet
- Sjeldne situasjoner
- Dokumenters konsistens og kompletthet
- Språkavhengigheter

Inspeksjonsdata Tre utviklingsaktiviteter i fokus:

- Høynivådesign
 - Dokumentasjon - 45,10%
 - Funksjon - 24,71%
 - Grenesnitt - 14,12%
 - Algoritme - 20,72%
- Lavnivådesign
 - Funksjon - 21,17%
 - Dokumentasjon
- Implementasjon.

Testdata Enhetstesting, systemtesting, akseptansetesting.

Testing og inspeksjon er forskjellige aktiviteter. De behøver: Forskjellige triggere, forskjellig mindset, og finner ulike typer defekter. En trenger derfor begge aktiviteter for å skape et produkt av høy kvalitet.

Kompendie i TDT4242

Hello world

Kompendiets kapitler er nummerert etter forelesningsnummer. Benyttet syntaks er [markdown](#).

Diskusjoner skjer på IRC - #etkulnavn på freenode.

Kontributører

[kartoffelmos ekun aeven](#)

For å bygge

Installere pandoc

Installere Ruby

Kjøre `ruby build.rb`

Ferdige kapitler

15, 12, 11, 9, 8, 6, 5 (5.2 må korrekturleses), 3(under review),

Gjenstår å gjøre

- Alle kapitler
- Build-skript som samler alle kapitler og bygger .mobi, .epub, .pdf (via LaTeX)og HTML.
- 3.2
- 8.2