Foundation Level Syllabus
Certified Tester

Version 4.0

International Software Testing Qualifications Board

Svensk kursplan v1.0

Upphovsrättsmeddelande

Översättning av engelsk Syllabus för International Software Testing Qualifications Board (ISTQB®), originaltitel: Certified Tester, Foundation Level Syllabus, Version 4.0.

Denna kursplan Certified Tester, Foundation Level, Version ISTQB ® CTFL V4.0 är skyddad av upphovsrätt. Swedish Software Testing Board, SSTB® innehar ensamrätt till kursplanen.

Revisionshistorik

För revisionshistorik för Syllabus se Syllabus-dokumentet.

Version	Datum	Kommentarer
SSTB 2005-2006 (1.1)	2006-03-07	Första svenska utgåvan
SSTB 2007 (2.0)	2007-12-20	Harmonisering med ”Syllabus CTFL ver. 2007”.
SSTB 2010 (2.1)	2010-10-26	Första version av översättning av ändringar introducerad i Syllabus 2010 inkl. ”Errata to CTFL 2010 8-Aug-10”
SSTB 2011 (2.2)	2011-04-29	Uppdaterad efter ISTQB® CTFL Syllabus 2011
SSTB 2018	2018-11-01	Uppdaterad efter ISTQB® CTFL Syllabus 2018
SSTB 2018	2019-06-28	Tydligare skillnad på säkerhet och informationssäkerhet (funktionssäkerhet borttaget) Några felöversättningar korrigerade
SSTB 2018 V3.1	2020-01-28	Uppdaterad efter mindre ändringar i Syllabus V 3.1 – se release notes. Språkmässiga uppdateringar
SSTB 4.0 v1.0	2023-09-19	Svensk utgåva av CTFL 4.0

Innehållsförteckning

Tack

Den engelskspråkiga förlagan godkändes formellt av ISTQB® 20223-04-21

Den skapades av ett team från ISTQB Working Group på Foundation Level & Agile Working Group: Laura Albert, Renzo Cerquozzi (vice chair), Wim Decoutere, Klaudia Dussa-Zieger, Chintaka Indikadahena, Arnika Hryszko, Martin Klonk, Kenji Onishi, Michaël Pilaeten (co-chair), Meile Posthuma, Gandhinee Rajkomar, Stuart Reid, Eric Riou du Cosquer (co-chair), Jean-François Riverin, Adam Roman, Lucjan Stapp, Stephanie Ulrich (vice chair), Eshraka Zakaria.

Teamen tackar Stuart Reid, Patricia McQuaid och Leanne Howard för deras tekniska granskning och medlemsorganen (MB) för deras förslag och input.

Swedish Software Testing Board (SSTB) tackar följande för bidrag till den svenska kursplanen: Tobias Ahlgren, Beata Karpinska, Johan Klintin, Johan Meivert, Ingvar Nordström, Daniel Säther, Mattias Östholm

Följande personer deltog i granskning, kommenterandet och omröstning av den engelska förlagan: Adam Roman, Adam Scierski, Ágota Horváth, Ainsley Rood, Ale Rebon Portillo, Alessandro Collino, Alexander Alexandrov, Amanda Logue, Ana Ochoa, André Baumann, André Verschelling, Andreas Spillner, Anna Miazek, Armin Born, Arnd Pehl, Arne Becher, Attila Gyúri, Attila Kovács, Beata Karpinska, Benjamin Timmermans, Blair Mo, Carsten Weise, Chinthaka Indikadahena, Chris Van Bael, Ciaran O'Leary, Claude Zhang, Cristina Sobrero, Dandan Zheng, Dani Almog, Daniel Säther, Daniel van der Zwan, Danilo Magli, Darvay Tamás Béla, Dawn Haynes, Dena Pauletti, Dénes Medzihradszky, Doris Dötzer, Dot Graham, Edward Weller, Erhardt Wunderlich, Eric Riou Du Cosquer, Florian Fieber, Fran O'Hara, François Vaillancourt, Frans Dijkman, Gabriele Haller, Gary Mogyorodi, Georg Sehl, Géza Bujdosó, Giancarlo Tomasig, Giorgio Pisani, Gustavo Márquez Sosa, Helmut Pichler, Hongbao Zhai, Horst Pohlmann, Ignacio Trejos, Ilia Kulakov, Ine Lutterman, Ingvar Nordström, Iosif Itkin, Jamie Mitchell, Jan Giesen, Jean-Francois Riverin, Joanna Kazun, Joanne Tremblay, Joëlle Genois, Johan Klintin, John Kurowski, Jörn Münzel, Judy McKay, Jürgen Beniermann, Karol Frühauf, Katalin Balla, Kevin Kooh, Klaudia Dussa-Zieger, Klaus Erlenbach, Klaus Olsen, Krisztián Miskó, Laura Albert, Liang Ren, Lijuan Wang, Lloyd Roden, Lucjan Stapp, Mahmoud Khalaili, Marek Majernik, Maria Clara Choucair, Mark Rutz, Markus Niehammer, Martin Klonk, Márton Siska, Matthew Gregg, Matthias Hamburg, Mattijs Kemmink, Maud Schlich, May Abu-Sbeit, Meile Posthuma, Mette Bruhn-Pedersen, Michal Tal, Michel Boies, Mike Smith, Miroslav Renda, Mohsen Ekssir, Monika Stocklein Olsen, Murian Song, Nicola De Rosa, Nikita Kalyani, Nishan Portoyan, Nitzan Goldenberg, Ole Chr. Hansen, Patricia McQuaid, Patricia Osorio, Paul Weymouth, Pawel Kwasik, Peter Zimmerer, Petr Neugebauer, Piet de Roo, Radoslaw Smilgin, Ralf Bongard, Ralf Reissing, Randall Rice, Rik Marselis, Rogier Ammerlaan, Sabine Gschwandtner, Sabine Uhde, Salinda Wickramasinghe, Salvatore Reale, Sammy Kolluru, Samuel Ouko, Stephanie Ulrich, Stuart Reid, Surabhi Bellani, Szilard Szell, Tamás Gergely, Tamás Horváth, Tatiana Sergeeva, Tauhida Parveen, Thaer Mustafa, Thomas Eisbrenner, Thomas Harms, Thomas Heller, Tobias Letzkus, Tomas Rosenqvist, Werner Lieblang, Yaron Tsubery, Zhenlei Zuo and Zsolt Hargitai.

ISTQB Working Group Foundation Level (Edition 2018): Klaus Olsen (chair), Tauhida Parveen (vice chair), Rex Black (project manager), Eshraka Zakaria, Debra Friedenberg, Ebbe Munk, Hans Schaefer, Judy McKay, Marie Walsh, Meile Posthuma, Mike Smith, Radoslaw Smilgin, Stephanie Ulrich, Steve Toms, Corne Kruger, Dani Almog, Eric Riou du Cosquer, Igal Levi, Johan Klintin, Kenji Onishi, Rashed Karim, Stevan Zivanovic, Sunny Kwon, Thomas Müller, Vipul Kocher, Yaron Tsubery and all Member Boards for their suggestions.

ISTQB Working Group Foundation Level (Edition 2011): Thomas Müller (chair), Debra Friedenberg. The core team thanks the review team (Dan Almog, Armin Beer, Rex Black, Julie Gardiner, Judy McKay, Tuula Pääkkönen, Eric Riou du Cosquier Hans Schaefer, Stephanie Ulrich, Erik van Veenendaal), and all Member Boards for the suggestions for the current version of the syllabus.

ISTQB Working Group Foundation Level (Edition 2010): Thomas Müller (chair), Rahul Verma, Martin Klonk and Armin Beer. The core team thanks the review team (Rex Black, Mette Bruhn-Pederson, Debra Friedenberg, Klaus Olsen, Judy McKay, Tuula Pääkkönen, Meile Posthuma, Hans Schaefer, Stephanie Ulrich, Pete Williams, Erik van Veenendaal), and all Member Boards for their suggestions.

ISTQB Working Group Foundation Level (Edition 2007): Thomas Müller (chair), Dorothy Graham, Debra Friedenberg, and Erik van Veenendaal. The core team thanks the review team (Hans Schaefer, Stephanie Ulrich, Meile Posthuma, Anders Pettersson, and Wonil Kwon) and all the Member Boards for their suggestions.

ISTQB Working Group Foundation Level (Edition 2005): Thomas Müller (chair), Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhäjärvi, Geoff Thompson and Erik

0. Inledning

0.1. Syfte med kursplanen

Denna kursplan utgör grunden för ackreditering och certifiering på ISTQB® Foundation Level. SSTB tillhandahåller kursplanen med följande syften:

1. Till kurshållare för att ta fram kursmaterial och som vägledning för att hitta lämpliga undervisningsmetoder för att sedan ackreditera sig.
2. Till de som vill certifiera sig så att de kan förbereda sig för certifieringsexamineringen (antingen som del av en utbildning eller fristående).
3. Till programvaru- och systemtekniker som ett hjälpmedel för att öka den samlade kunskapen om programvaru- och systemtestning, och som grund för böcker och artiklar.

ISTQB® och SSTB kan också tillåta andra organ och enskilda att använda kursplanen i andra syften förutsatt att de inhämtar skriftligt godkännande från ISTQB® och SSTB i förväg.

0.2. Certifierad testare på Foundation Level för programvarutestning

Foundation Level-nivån riktar sig till alla som är inblandade i programvarutestning. Den kan omfatta personer i roller som testare, testanalytiker, testingenjörer, testkonsulter, testledare, programvaruutvecklare och teammedlemmar inom utveckling. Foundation Level-nivån är också lämplig för alla som vill ha en grundläggande förståelse för programvarutestning, såsom projektledare, kvalitetschefer, produktägare, programvaruutvecklingschefer, affärsanalytiker, verksamhetsanalytiker, IT-chefer och managementkonsulter. Innehavare av Foundation-certifikatet kommer att kunna gå vidare till högre kunskapsnivåer inom programvarutestning.

0.3. Karriärväg för testare

ISTQB®-programmet ger stöd för yrkeskunniga testare i alla skeden i deras karriärer och erbjuder både bred och djup kunskap. Individer som erhållit ISTQB® Foundation-certifikatet kan också vara intresserade av Advanced Level (Test Analyst, Technical Test Analyst, och Test Manager och därefter Expert Level (Test Management eller Improving the Test Process). Alla som vill utveckla sina testkunskaper för en agil miljö kan överväga certifieringarna Agile Technical Tester eller Agile Test Leadership at Scale. Specialistvägen erbjuder en djupdykning i områden som har specifika testmetoder och testaktiviteter (t.ex. inom testautomatisering, AI-testning, modellbaserad testning, mobilappstestning), som är relaterade till specifika testområden (t.ex. prestandatestning, användbarhetstestning, acceptanstestning, säkerhetstestning), eller testkunnande för vissa branschdomäner (t.ex. fordon eller spel). Besök www.istqb.org för den senaste informationen om ISTQB:s Certified Tester-programmet.

0.4. Verksamhetsresultat

Nedan listas de 14 verksamhetsresultat (Business Outcomes, BO) som förväntas av en person som har Foundation Level-certifikatet.

En certifierad testare på Foundation Level kan:

FL-BO1	Förstå vad testning är och varför den är värdefull
FL-BO2	Förstå grundläggande koncept för programvarutestning
FL-BO3	Identifiera teststrategi och aktiviteter som ska genomföras beroende på testsammanhanget
FL-BO4	Bedöma och förbättra kvaliteten på dokumentation
FL-BO5	Förbättra effektiviteten hos testningen
FL-BO6	Anpassa testprocessen till programvaruutvecklingens livscykel
FL-BO7	Förstå testhanteringsprinciper
FL-BO8	Skriva och kommunicera tydliga och begripliga felrapporter
FL-BO9	Förstå de faktorer som påverkar prioriteringar och arbete relaterade till testning
FL-BO10	Arbeta som en del av ett tvärfunktionellt team
FL-BO11	Känna till risker och fördelar med testautomatisering
FL-BO12	Identifiera nödvändiga färdigheter som krävs för testning
FL-BO13	Förstå riskers påverkan på testningen
FL-BO14	Effektivt rapportera om testframsteg och kvalitet

0.5. Utbildningsmål som kan examineras och kognitiva kunskapsnivåer

Utbildningsmålen (Learning Objectives, LO) stödjer verksamhetsresultaten (BO) och används för att skapa de examineringar som används för att certifiera testare på Foundation Level. Rent allmänt kan man säga att allt innehåll i kapitel 1-6 i den här kursplanen är examinerbart på K1-nivå. Det innebär att kandidaten ska identifiera, komma ihåg eller minnas ett nyckelord eller koncept som nämns i något av de sex kapitlen. Utbildningsmålen som finns i början av varje kapitel klassificeras enligt följande:

- K1: komma ihåg
- K2: förstå
- K3: tillämpa

Mer information och exempel på utbildningsmål finns i bilaga A. Definitioner av alla termer som listas som nyckelord strax under kapitelrubrikerna ska kommas ihåg (K1), även om de inte uttryckligen nämns i utbildningsmålen.

0.6. Certifieringsexamen på Foundation Level

Certifieringsexamen för Foundation Level bygger på den här kursplanen. Svaren på examineringsfrågorna kan kräva användning av material som bygger på mer än ett kapitel i kursplanen. Alla kapitel i kursplanen, förutom inledning och bilagor, kan examineras. Standarder och böcker inkluderas som referenser (Kapitel 7), men innehållet i sådana standarder, böcker är inte att examinerbara utöver vad som sammanfattas i den här kursplanen. För detaljer se dokumentet Exam Structures and Rules for the Foundation Level

0.7. Ackreditering

Ett ISTQB®-nationellt organ, t.ex. SSTB, kan ackreditera kurshållare vars kursmaterial följer denna kursplan. Kurshållaren ska införskaffa ackrediteringsriktlinjer från det organ som utför ackrediteringen. En kurs är ackrediterad då den överensstämmer med den här kursplanen och får ha en ISTQB®-examinering i anslutning till kursen. Ackrediteringsriktlinjerna för denna kursplan följer General Accreditation Guidelines som publicerats av Processes Management and Compliance Working Group.

0.8. Hantering av standarder

I kursplanen finns referenser till standarder (t.ex. IEEE eller ISO-standarder). Dessa referenser tillhandahåller ett ramverk (som i referenserna till ISO 25010 angående kvalitetsegenskaper) eller en källa till ytterligare information om så önskas av läsaren. Standarddokumenten är inte avsedda för examinering. Se kapitel 7 för mer information om standarder.

0.9. Hålla information aktuell

Programvaruindustrin förändras snabbt. För att hantera dessa förändringar och för att ge intressenterna tillgång till relevant och aktuell information har ISTQB:s arbetsgrupper skapat länkar på webbplatsen www.istqb.org som hänvisar till stödjande dokument och ändringar av standarder. Denna information är inte examinerbar.

0.10. Detaljnivå

Detaljnivån i syllabus och den här kursplanen ger möjlighet till att skapa kurser och examineringar som överensstämmer internationellt. För att uppnå det målet innehåller kursplanen:

- Allmänna instruktionsmål som beskriver syftet med Foundation Level
- En lista med termer (nyckelord) som eleverna måste komma ihåg
- Utbildningsmål (Learning Objectives) för varje kunskapsområde, som beskriver det kognitiva utbildningsresultatet som ska uppnås
- En beskrivning av nyckelorden, inklusive referenser till erkända källor

Kursplanens innehåll är inte en beskrivning av hela kunskapsområdet programvarutestning; den återspeglar den detaljnivå som ska täckas i kurser för Foundation Level. Den fokuserar på testkoncept och testtekniker som kan tillämpas i alla programvaruprojekt, oberoende av vilken SDLC som används.

0.11. Kursplanens upplägg

Det finns sex kapitel med innehåll som kan examineras. Rubriken på den översta nivån för varje kapitel innehåller kurstiden för kapitlet. Inga tider anges på lägre kapitelnivåer. För ackrediterade kurser kräver kursplanen en tid för undervisning på minst 1135 minuter (18 timmar och 55 minuter) som fördelas över de sex kapitlen enligt följande:

- Kapitel 1: Grunderna inom testning (180 minuter)
    - Eleven lär sig de grundläggande principerna relaterade till testning, skälen till varför testning behövs och vad syftet med testningen är.
    - Eleven förstår testprocessen, de större testaktiviteterna och testvaran.
    - Eleven förstår de väsentliga färdigheterna för att kunna testa.
- Kapitel 2: Testning i programvarans utvecklingslivscykel (130 minuter)
    - Eleven lär sig hur testning ingår i olika utvecklingsstrategier.
    - Eleven lär sig begreppen test-first och DevOps.
    - Eleven lär sig om olika testnivåer, testtyper och underhållstestning.
- Kapitel 3: Statisk testning (80 minuter)
    - Eleven lär sig om grunderna för statisk testning, om återkoppling och om granskningsprocessen
- Kapitel 4: Testanalys och design (390 minuter)
    - Eleven lär sig hur man tillämpar black-box, white-box och erfarenhetsbaserade testtekniker för att härleda testfall från olika programvaruprodukter.
    - Eleven lär sig om samarbetsbaserat testangreppssätt.
- Kapitel 5: Hantering av testaktiviteterna (335 minuter)
    - Eleven lär sig hur man planerar testning i allmänhet och hur man uppskattar testarbetsmängden.
    - Eleven lär sig hur risker kan påverka omfattningen av testning.
    - Eleven lär sig att övervaka och kontrollera testaktiviteter.
    - Eleven lär sig hur konfigurationshantering stödjer testning.
    - Eleven lär sig att rapportera fel på ett tydligt och begripligt sätt.
- Kapitel 6: Testverktyg (20 minuter)
    - Eleven lär sig att klassificera testverktyg och att förstå risker och fördelar med testautomatisering.

1. Grunderna inom test – 180 minuter

Nyckelord

debugging, defekt, felsymptom, grundorsak, kvalitet, kvalitetssäkring, misstag, testanalys, testavslut, testbas, testdata, testdesign, testexekvering, testfall, testimplementation, testmålsättning, testning, testobjekt, testövervakning, testplanering, testprocedur, testresultat, teststyrning, testvara, testvillkor, validering, verifiering

Utbildningsmål för kapitel 1:

1.1 Vad är testning?

FL-1.1.1 (K1) Identifiera typiska målsättningar för testningen

FL-1.1.2 (K2) Skilja på testning och debugging

1.2 Varför är testning nödvändigt?

FL-1.2.1 (K2) Ge exempel på varför testning är nödvändigt

FL-1.2.2 (K1) Komma ihåg relationen mellan testning och kvalitetssäkring

FL-1.2.3 (K2) Skilja mellan grundorsak, misstag, defekt och felsymptom

1.3 Testprinciper

FL-1.3.1 (K2) Förklara de sju testprinciperna

1.4 Testaktiviteter, testvara och testroller

FL-1.4.1 (K2) Summera de olika testaktiviteterna och testuppgifterna

FL-1.4.2 (K2) Förklara sammanhangets inverkan på testprocessen

FL-1.4.3 (K2) Skilja på testvara som ger stöd åt testprocessen

FL-1.4.4 (K2) Förklara värdet av att bibehålla spårbarheten

FL-1.4.5 (K2) Jämföra de olika rollerna i testningen

1.5 Viktiga färdigheter och god praxis vid testning

FL-1.5.1 (K2) Ge exempel på de generiska färdigheter som krävs för att testa

FL-1.5.2 (K1) Komma ihåg fördelarna med hela teamets ansvar (whole team approach)

FL-1.5.3 (K2) Särskilja för- och nackdelar med oberoende testning

1.1. Vad är testning?

Programvarusystem är en viktig del av vardagen. De flesta av oss har upplevt att programvaran inte har fungerat som förväntat. Om den inte fungerar på rätt sätt kan det leda till många problem, inklusive förlust av pengar, tid, affärsrykte och värsta fall personskador eller dödsfall. Programvarutestning utvärderar programvarans kvalitet och hjälper till att minska risken för felsymptom i drift.

Programvarutestning är en uppsättning aktiviteter för att upptäcka defekter och utvärdera kvaliteten hos programvaruartefakter. Dessa artefakter benämns testobjekt när de testas. En vanlig missuppfattning om testning är att den bara består av att exekvera tester (dvs köra programvaran och kontrollera testresultaten). Men programvarutestning omfattar också andra aktiviteter och måste därför anpassas till programvarans utvecklingslivscykel (se kapitel 2).

En annan vanlig missuppfattning om testning är att den enbart fokuserar på att verifiera testobjektet. Även om testning innebär verifiering, det vill säga att kontrollera om systemet uppfyller specificerade krav, omfattar den också validering, dvs. kontroll om systemet uppfyller användarnas och andra intressenters behov i sin operativa miljö.

Testning kan vara dynamisk eller statisk. Dynamisk testning innebär exekvering av programvaran, medan statisk testning inte gör det. Statisk testning inkluderar granskningar (se kapitel 3) och statisk analys. Dynamisk testning använder olika typer av testtekniker och testangreppssättför att härleda testfall (se kapitel 4).

Testning är inte bara en teknisk aktivitet. Den måste också planeras, hanteras, uppskattas, övervakas och styras på rätt sätt (se kapitel 5).

Testare använder verktyg (se kapitel 6), men det är viktigt att komma ihåg att testning till stor del är en intellektuell aktivitet, vilket kräver att testarna har specialiserade kunskaper, använder analytiska färdigheter, tillämpar kritiskt tänkande och systemtänkande (Myers 2011, Roman 2018).

ISO/IEC/IEEE 29119-1-standarden ger ytterligare information om koncept avseende programvarutestning.

1.1.1. Målsättning med testningen

Typiska målsättningar med testningen:

- Utvärdera arbetsprodukter, till exempel krav, användarberättelser, design och kod
- Provocera fram felsymptom och upptäcka defekter
- Säkerställa erforderlig täckning av ett testobjekt
- Minska risknivån för bristfällig programvarukvalitet
- Verifiera om specificerade krav har uppfyllts
- Verifiera att ett testobjekt följer avtalsmässiga, juridiska och lagstadgade krav
- Tillhandahålla information till intressenter så att de kan fatta välgrundade beslut
- Bygga förtroende för testobjektets kvalitet
- Validera om testobjektet är komplett och fungerar som förväntat enligt intressenterna

Målsättningen med testningen kan variera beroende på sammanhanget, vilket inkluderar arbetsprodukten som testas, testnivån, risker, programvaruutvecklingslivscykeln (SDLC) som följs och faktorer relaterade till affärskontexten, t.ex. företagsstruktur, konkurrensavvägande eller tid till marknaden.

1.1.2. Testning och debugging

Testning och debugging är olika aktiviteter. Testning kan ge upphov till felsymptom som förorsakats av defekter (fel) i programvaran (dynamisk testning) eller kan hitta defekter direkt i testobjektet (statisk testning).

När dynamisk testning (se kapitel 4) utlöser ett felsymptom innebär debugging att hitta orsakerna till felsymptomet (dvs. defekter), analysera orsakerna och eliminera dem. Den typiska felsökningsprocessen i detta fall innefattar:

- Återskapande av felsymptomet
- Diagnos (att hitta grundorsaken)
- Åtgärda orsaken

Efterföljande omtestning kontrollerar om åtgärderna löste problemet. Helst ska omtestning göras av samma person som utförde det första testet. Efterföljande regressionstestning kan också utföras för att kontrollera om åtgärderna orsakat felsymptom i andra delar av testobjektet (se avsnitt 2.2.3 för mer information om omtestning och regressionstestning).

När statisk testning identifierar en defekt handlar debugging om att eliminera den. Det finns inget behov av återskapande eller diagnos, eftersom statisk testning hittar defekter direkt och inte kan orsaka felsymptom (se kapitel 3).

1.2. Varför är testning nödvändigt?

Testning, som en form av kvalitetskontroll, hjälper till att uppnå de överenskomna målen gällande omfattning, tid, kvalitet och budgetbegränsningar. Testningens bidrag till framgång ska inte begränsas till testteamets aktiviteter. Alla intressenter kan använda sina testkunskaper för att föra projektet framåt. Att testa komponenter, system och tillhörande dokumentation hjälper till att hitta defekter i programvaran.

1.2.1. Testningens bidrag till ett lyckat resultat

Testning är ett kostnadseffektivt sätt att upptäcka defekter. Dessa defekter kan sedan elimineras (genom debugging – en icke-testande aktivitet), så testning bidrar indirekt till högre kvalitet hos testobjekten.

Testning är ett sätt att direkt utvärdera kvaliteten på ett testobjekt i olika stadier i programvaruutvecklingslivscykeln. Dessa åtgärder ingår som en del av en större projektledningsaktivitet och bidrar till beslut om att gå vidare till nästa steg i programvaruutvecklingslivscykeln, som t.ex. releasebeslut.

Via testningen blir användarna indirekt representerade i utvecklingsprojektet. Testare säkerställer att deras förståelse av användarnas behov beaktas under hela utvecklingslivscykeln. Alternativet är att involvera en representativ uppsättning användare som en del av utvecklingsprojektet, vilket vanligtvis inte är möjligt på grund av de höga kostnaderna och bristen på lämpliga och tillgängliga användare.

Testning kan också krävas för att uppfylla avtalsenliga eller juridiska krav, eller för att följa lagstadgade standarder.

1.2.2. Testning och kvalitetssäkring (QA)

Även om många ofta använder termerna "testning" och "kvalitetssäkring" (QA) omväxlande, är testning och QA inte samma sak. Testning är en form av kvalitetskontroll (QC).

QC är ett produktorienterat, korrigerande tillvägagångssätt som fokuserar på de aktiviteter som stödjer uppnåendet av lämpliga kvalitetsnivåer. Testning är en viktig del av kvalitetskontroll, men inkluderar även formella metoder (modellkontroll och bevis på korrekthet), simulering och prototyper.

QA är ett processorienterat, förebyggande arbetssätt som fokuserar på implementering och förbättring av processer. Det utgår från att om en bra process följs korrekt, så kommer den att generera en bra produkt. QA gäller både utvecklings- och testprocesserna och är allas ansvar i ett projekt.

Testresultaten används av QA och QC. I QC används de för att åtgärda defekter, medan de i QA ger återkoppling om hur väl utvecklings- och testprocesserna fungerar.

1.2.3. Misstag, defekter, felsymptom och grundorsaker

Människor gör fel (misstag), som kan leda till defekter (fel, buggar), som i sin tur kan resultera i felsymptom. Människor gör fel av olika anledningar, som tidspress, komplexitet i arbetsprodukter, processer, infrastruktur eller interaktioner, eller helt enkelt för att de är trötta eller saknar adekvat utbildning.

Defekter kan hittas i dokumentation, till exempel en kravspecifikation eller ett testskript, i källkod eller i en stödjande artefakt som en byggfil. Defekter i artefakter som producerats tidigare i SDLC leder, om de inte upptäcks, ofta till defekta artefakter senare i livscykeln. Om en defekt kod exekveras kan systemet misslyckas med att göra vad det borde göra, eller göra något det inte borde, vilket orsakar ett felsymptom. Vissa defekter kommer alltid att resultera i ett felsymptom om de exekveras, medan andra endast kommer att resultera i ett felsymptom under specifika omständigheter, medan vissa defekter kanske aldrig resulterar i ett felsymptom.

Misstag och defekter är inte den enda orsaken till felsymptom. Felsymptom kan också orsakas av miljömässiga skäl, till exempel när strålning eller elektromagnetiska fält orsakar defekter i firmware.

En grundorsak är ett grundläggande skäl till att ett problem uppstår (t.ex. en situation som leder till ett misstag). Grundorsaker identifieras genom grundorsaksanalys, som vanligtvis utförs när ett felsymptom upptäcks eller en defekt identifieras. Detta bygger på kunskapen att ytterligare liknande felsymptom eller defekter kan förebyggas eller deras frekvens minskas genom att åtgärda grundorsaken, till exempel genom att eliminera den.

1.3. Testprinciper

Ett antal testprinciper innehållande allmänna riktlinjer som är tillämpliga på all testning har föreslagits genom åren. Denna kursplan beskriver sju sådana principer.

1. Testning visar närvaron, inte frånvaron, av defekter. Testning kan visa att det finns defekter i testobjektet, men kan inte bevisa att det inte finns några defekter (Buxton 1970). Testning minskar sannolikheten för att defekter förblir oupptäckta i testobjektet, men även om inga defekter hittas kan testning inte bevisa att testobjektet är korrekt.

2. Uttömmande testning är omöjligt. Att testa allt är inte möjligt utom i triviala fall (Manna 1978). I stället för att försöka testa uttömmande bör testtekniker (se kapitel 4), prioritering av testfall (se avsnitt 5.1.5) och riskbaserad testning (se avsnitt 5.2) användas för att fokusera på testinsatserna.

3. Tidig testning sparar tid och pengar. Defekter som elimineras tidigt i processen kommer inte ge upphov till efterföljande defekter i senare arbetsprodukter. Kostnaden för kvalitet kommer att minska eftersom färre fel kommer att inträffa senare i SDLC (Boehm 1981). För att hitta defekter tidigt bör både statisk testning (se kapitel 3) och dynamisk testning (se kapitel 4) påbörjas så tidigt som möjligt.

4. Ansamling av fel. Ett litet antal systemkomponenter innehåller vanligtvis de flesta av de upptäckta defekterna eller är ansvariga för de flesta driftstörningarna (Enders 1975). Detta fenomen är exempel på Pareto-principen. Förväntade defektkluster och faktiska defektkluster som observerats under testning eller i drift, är en viktig input för riskbaserad testning (se kapitel 5.2).

5. Tester slits ut. Om samma tester upprepas många gånger blir de allt mindre effektiva vad gäller att upptäcka nya defekter (Beizer 1990). För att övervinna denna effekt kan befintliga tester och testdata behöva modifieras och nya tester kan behöva skrivas. Men i vissa fall kan upprepning av samma tester ge ett fördelaktigt resultat, t.ex. vid automatiserad regressionstestning (se avsnitt 2.2.3).

6. Testning beror på sammanhang. Det finns ingen enskild universellt tillämplig metod för testning. Testning görs på olika sätt i olika sammanhang (Kaner 2011).

7. Frånvaro-av-fel-fallgropen. Det är felaktigt (dvs en missuppfattning) att förvänta sig att verifiering av programvaran räcker för att säkerställa ett framgångsrikt system. Att noggrant testa alla specificerade krav och åtgärda alla defekter som hittats kan ändå producera ett system som inte uppfyller användarnas behov och förväntningar, som inte hjälper till att uppnå kundens affärsmål och som är sämre jämfört med andra konkurrerande system. Därför ska, utöver verifiering, även validering genomföras (Boehm 1981).

1.4. Testaktiviteter, testvara och testroller

Testning är beroende av sitt sammanhang, men på en hög nivå finns det generella uppsättningar av testaktiviteter utan vilka det är mindre sannolikt att testning skulle uppnå testmålen. Dessa uppsättningar av testaktiviteter utgör en testprocess. Testprocessen kan skräddarsys till en given situation utifrån olika faktorer. Vilka testaktiviteter som ingår i denna testprocess, hur och när de genomförs är normalt en del av testplaneringen för den specifika situationen (se avsnitt 5.1).

Följande kapitel beskriver de allmänna aspekterna av denna testprocess i termer av testaktiviteter och testuppgifter, påverkan av sammanhanget, testvaran, spårbarhet mellan testbas och testvaran samt roller i testningen .

ISO/IEC/IEEE 29119-2-standarden ger ytterligare information om testprocesser.

1.4.1. Testaktiviteter och uppgifter

En testprocess består vanligtvis av de huvudgrupper av aktiviteter som beskrivs nedan. Även om många av dessa aktiviteter kan tyckas följa en logisk sekvens, genomförs de ofta iterativt eller parallellt. Dessa testaktiviteter behöver vanligtvis anpassas till systemet och projektet.

Testplanering består av att definiera testmålen och sedan välja ett angreppssätt som bäst uppnår målen inom de begränsningar som det övergripande sammanhanget ställer. Testplanering förklaras ytterligare i kapitel 5.1.

Testövervakning och styrning. Testövervakning innebär löpande kontroll av alla testaktiviteter och jämförelse av faktiska framsteg mot planen. Teststyrning innebär att vidta de åtgärder som är nödvändiga för att uppnå testmålen. Testövervakning och styrning förklaras ytterligare i kapitel 5.3.

Testanalys omfattar analys av testbasen för att identifiera testbara funktionaliteter och för att definiera och prioritera associerade testvillkor tillsammans med relaterade risker och risknivåer (se kapitel 5.2). Testbasen och testobjekten utvärderas också för att bedöma deras testbarhet och för att identifiera defekter som skulle kunna hittas. Testanalys stöds ofta genom användning av testtekniker (se kapitel 4). Testanalys svarar på frågan "vad ska man testa?" i termer av mätbara täckningskriterier.

Testdesign omfattar utveckling av testvillkoren till testfall och annan testvara (t.ex. testcharter). Denna aktivitet omfattar ofta en identifiering av täckningsobjekt, som fungerar som en guide för att specificera indata för testfall. Testtekniker (se kapitel 4) kan användas för att stödja denna aktivitet. Testdesign inkluderar också att definiera krav på testdata, designa testmiljön och identifiera eventuell annan nödvändig infrastruktur och verktyg. Testdesign svarar på frågan "hur testar man?".

Testimplementation omfattar skapande eller inskaffande av den testvara som krävs för testexekvering (t.ex. testdata). Testfall kan organiseras i testprocedurer och sätts ofta ihop till

testsviter. Manuella och automatiserade testskript skapas. Testprocedurer prioriteras och ordnas inom ett testexekveringsschema för effektiv testexekvering (se kapitel 5.1.5). Testmiljön byggs och verifieras så att den är korrekt.

Testexekvering innebär att köra testerna i enlighet med testexekveringsschemat. Testexekvering kan vara manuell eller automatiserad. Testexekvering kan ta många former, inklusive kontinuerlig testning eller sessioner med partestning. Faktiska testresultat jämförs med de förväntade resultaten. Testresultaten loggas. Avvikelserna analyseras för att identifiera deras troliga orsaker. Denna analys gör det möjligt för oss att rapportera avvikelserna baserat på de observerade felsymptomen (se kapitel 5.5).

Testavslut. Testavslutsaktiviterna inträffar vanligtvis vid projektmilstolpar (t.ex. release, slut på en iteration eller en testnivå) efter att eventuella olösta defekter, ändringsbegäran eller produktbacklogobjekt som skapats. All testvara som kan vara användbar i framtiden identifieras och arkiveras eller lämnas över till lämpliga team. Testmiljön stängs av till ett överenskommet tillstånd. Testaktiviteterna analyseras för att identifiera lärdomar och förbättringar för framtida iterationer, releaser eller projekt (se kapitel 2.1.6). En sammanfattande testrapport skapas och kommuniceras till intressenterna.

1.4.2. Testprocess i sitt sammanhang

Testning utförs inte isolerat. Testaktiviteterna är en integrerad del av de utvecklingsprocesser som genomförs i en organisation. Testning finansieras också av intressenter och dess slutmål är att hjälpa till att uppfylla intressenternas affärsbehov. Därför kommer behovet av hur testningen utförs att bero på ett antal kontextuella faktorer, inklusive:

- Intressenter (behov, förväntningar, krav, vilja att samarbeta, etc.)
- Teammedlemmar (färdigheter, kunskap, erfarenhetsnivå, tillgänglighet, utbildningsbehov etc.)
- Företagsdomän (testobjektets kritikalitet, identifierade risker, marknadsbehov, specifika juridiska regler, etc.)
- Tekniska faktorer (typ av programvara, produktarkitektur, använd teknik, etc.)
- Projektbegränsningar (omfattning, tid, budget, resurser, etc.)
- Organisatoriska faktorer (organisationsstruktur, befintliga policyer, praxis som används, etc.)
- Livscykeln för programvaruutveckling (teknikpraxis, utvecklingsmetoder, etc.)
- Verktyg (tillgänglighet, användbarhet, efterlevnad, etc.)

Dessa faktorer kommer att ha en inverkan på många testrelaterade frågor, inklusive teststrategi, använda testtekniker, grad av testautomatisering, erforderlig täckningsnivå, detaljnivå på testdokumentation, rapportering, osv.

1.4.3. Testvara

Testvara är arbetsprodukter som resultat av de testaktiviteter som beskrivs i kapitel 1.4.1. Det finns en betydande variation i hur olika organisationer producerar, utformar, namnger, organiserar och hanterar sina arbetsprodukter. Korrekt konfigurationshantering (se kapitel 5.4) säkerställer konsistens och integritet för arbetsprodukterna. Se följande lista över arbetsprodukter för testaktiviteter, den är inte komplett:

- **Testplanering:** Testplan, testschema, riskregister, start- och avslutskriterier (se kapitel 5.1). Riskregister är en lista över risker tillsammans med risksannolikhet, riskpåverkan och information om riskreducering (se kapitel 5.2). Testschema, riskregister, start- och avslutskriterier är ofta en del av testplanen.

- **Testövervakning och styrning:** Teststatusrapporter (se kapitel 5.3.2), dokumentation av styrdirektiv (se kapitel 5.3) och riskinformation (se kapitel 5.2).
- **Testanalys:** Prioriterade testvillkor (t.ex. acceptanskriterier, se kapitel 4.5.2) och felrapporter om defekter i testbasen (om de inte åtgärdats direkt).
- **Testdesign.:** Prioriterade testfall, testcharter, täckningsobjekt, testdatakrav och testmiljökrav.
- **Testimplementation:** Testprocedurer, testskript för automatisering, testsviter, testdata, testexekveringsschema och testmiljöelement. Exempel på testmiljöelement är stubbar, drivrutiner, simulatorer och tjänstevirtualiseringar.
- **Testexekvering:** Testloggar och felrapporter (se kapitel 5.5).
- **Testavslut:** Sammanfattande testrapport (se kapitel 5.3.2), åtgärdspunkter för förbättring av efterföljande projekt eller iterationer, dokumenterade lärdomar och ändringsbegäran (t.ex. som produktbackloggobjekt).

1.4.4. Spårbarhet mellan testbas och testvara

För att implementera effektiv testövervakning och styrning är det viktigt att etablera och upprätthålla spårbarhet genom hela testprocessen mellan testbaselementen, testvara associerad med dessa element (t.ex. testvillkor, risker, testfall), testresultat och upptäckta defekter.

En korrekt spårbarhet stödjer täckningsutvärdering, så det är mycket användbart om mätbara täckningskriterier har definierats i testbasen. Täckningskriterierna kan fungera som nyckeltal för att användas i de aktiviteter som visar i vilken utsträckning testmålen har uppnåtts (se kapitel 1.1.1). Till exempel:

- Spårbarhet av testfall till krav kan verifiera att kraven är täckta av testfall.
- Spårbarhet av testresultat till risker kan användas för att utvärdera nivån på kvarvarande risker i ett testobjekt.

Förutom att utvärdera täckningen, gör god spårbarhet det möjligt att avgöra påverkan av förändringar, underlättar testaudits och hjälper till att uppfylla IT-styrningskriterier. God spårbarhet gör också teststatusrapporter och sammanfattande testrapporter lättare att förstå genom att inkludera status för testbaselementen. Detta kan också hjälpa till att kommunicera de tekniska aspekterna av testning till intressenter på ett begripligt sätt. Spårbarhet ger information för att bedöma produktkvalitet, processkapacitet och projektframsteg mot företagets mål.

1.4.5. Roller i testningen

I denna kursplan tas två huvudroller inom testning upp: en testledningsroll och en testroll. De aktiviteter och uppgifter som tilldelas dessa två roller beror på faktorer som projekt- och produktkontext och kompetensen hos personerna i rollerna och organisationen.

Testledningsrollen tar ett övergripande ansvar för testprocessen, testteamet och ledarskapet för testaktiviteterna. Testledningsrollen är huvudsakligen inriktad på aktiviteterna testplanering, testövervakning och styrning samt testavslut. Sättet på vilket testledningsrollen utförs varierar beroende på sammanhanget. Till exempel kan i agil programvaruutveckling vissa av testhanteringsuppgifterna hanteras av det agila teamet. Uppgifter som spänner över flera team eller hela organisationen kan utföras av en testledingsroll utanför utvecklingsteamet.

Testrollen tar ett övergripande ansvar för den tekniska aspekten av testning. Testrollen är främst inriktad på aktiviteterna testanalys, testdesign, testimplementation och testexekvering.

Olika personer kan ta på sig dessa roller vid olika tidpunkter. Till exempel kan testledningsrollen utföras av en teamledare, en testledare eller testchef, en utvecklingschef etc. Det är också möjligt för en person att samtidigt ta rollen som testare och testledning.

1.5. Viktiga färdigheter och god praxis vid testning

Färdighet är förmågan att göra något bra som kommer från egen kunskap, praktik och begåvning. Bra testare bör ha vissa nödvändiga färdigheter för att göra sitt jobb bra. Bra testare ska vara effektiva lagspelare och ska kunna utföra testning på olika nivåer av testoberoende.

1.5.1. Generiska färdigheter som krävs för testning

Även om de är generiska så är följande färdigheter särskilt relevanta för testare:

- Testkunskap (för att öka effektiviteten i testningen, t.ex. genom att använda testtekniker)
- Grundlighet, noggrannhet, nyfikenhet, uppmärksam på detaljer, metodisk (för att identifiera defekter, särskilt de som är svåra att hitta)
- Bra kommunikationsförmåga, aktivt lyssnande, att vara en lagspelare (att interagera effektivt med alla intressenter, att förmedla information till andra, att bli förstådd och att rapportera och diskutera defekter)
- Analytiskt tänkande, kritiskt tänkande, kreativitet (för att öka effektiviteten i testningen)
- Teknisk kunskap (för att öka effektiviteten i testningen, t.ex. genom att använda lämpliga testverktyg)
- Domänkunskap (för att kunna förstå och kommunicera med slutanvändare eller företagsrepresentanter)

Testare är ofta de som kommer med dåliga nyheter. Det är tyvärr ett vanligt mänskligt drag att skylla på den som kommer med dåliga nyheter. Detta gör kommunikationsförmågan avgörande för testare. Att kommunicera testresultat kan uppfattas som kritik mot produkten och dess författare. Bekräftelsebias kan göra det svårt att acceptera information som inte stämmer överens med nuvarande uppfattningar. Vissa människor kan uppfatta testning som en destruktiv aktivitet, även om den i hög grad bidrar till projektframgång och produktkvalitet. För att försöka förbättra denna syn bör information om defekter och fel kommuniceras på ett konstruktivt sätt.

1.5.2. Hela teamets ansvar (Whole Team Approach)

En av de viktiga färdigheterna för en testare är förmågan att arbeta effektivt i ett teamsammanhang och att bidra positivt till teamets mål. Hela teamets ansvar – en praxis som kommer från Extreme Programming (se kapitel 2.1) – bygger på denna färdighet.

I hela teamets ansvar kan varje teammedlem med nödvändiga kunskaper och färdigheter utföra vilken uppgift som helst, och alla är ansvariga för kvaliteten. Teammedlemmarna delar samma arbetsplats (fysisk eller virtuell), eftersom samlokalisering underlättar kommunikation och interaktion. Hela teamets ansvar förbättrar dynamiken, kommunikationen och samarbetet inom teamet och skapar synergi genom att tillåta de olika kompetenserna inom teamet att utnyttjas till förmån för projektet.

Testare arbetar nära andra teammedlemmar för att säkerställa att de önskade kvalitetsnivåerna uppnås. Detta innebär samarbete med företagsrepresentanter för att hjälpa dem att skapa lämpliga acceptanstester och att arbeta med utvecklare för att komma överens om teststrategin och besluta om metoder för testautomatisering. Testare kan därmed överföra testkunskaper till andra teammedlemmar och påverka utvecklingen av produkten.

Beroende på sammanhanget kanske hela teamets ansvar inte alltid är lämpligt. Det kan exempel i vissa situationer, som säkerhetskritiska, behövas en hög nivå av testoberoende.

1.5.3. Oberoende testning

En viss grad av oberoende gör testaren mer effektiv i att hitta defekter på grund av skillnader mellan författarens och testarens kognitiva synsätt (jfr Salman 1995). Oberoende är dock inte en ersättning för förtrogenhet, t.ex. kan utvecklare på ett effektivt sätt hitta många defekter i sin egen kod.

Arbetsprodukter kan testas av deras författare (inget oberoende), av författarens kollegor från samma team (viss oberoende), av testare utanför författarens team men inom organisationen (högt oberoende), eller av testare utanför organisationen (mycket hög oberoende). För de flesta projekt är det vanligtvis bäst att utföra testningen med flera nivåer av oberoende (t.ex. utvecklare som utför komponent- och komponentintegreringstestning, testteam som utför system- och systemintegreringstester och företagsrepresentanter som utför acceptanstestning).

Den största fördelen med oberoende av testning är att oberoende testare sannolikt känner igen olika typer av felsymptom och defekter jämfört med utvecklare på grund av deras olika bakgrund, tekniska perspektiv och synsätt. Dessutom kan en oberoende testare verifiera, ifrågasätta eller motbevisa antaganden som gjorts av intressenter under specificering och implementering av systemet.

Men det finns också några nackdelar. Oberoende testare kan vara isolerade från utvecklingsteamet, vilket kan leda till bristande samarbete, kommunikationsproblem eller en fientlig relation med utvecklingsteamet. Utvecklare kan tappa känslan av ansvar för kvaliteten. Oberoende testare kan ses som en flaskhals eller skyllas på vid förseningar i releasen.

2. Testning under programvarans utvecklingslivscykel – 130 minuter

Nyckelord

acceptanstestning, black-box-testning, funktionstestning, icke-funktionell testning, integrationstestning, komponentintegrationstestning, komponenttestning, omtestning, regressionstestning, shift-left, systemintegrationstestning, systemtestning, testnivå, testobjekt, testtyp, underhållstestning, white-box-testning

Utbildningsmål för Kapitel 2:

2.1 Testning inom ramen för en programvaruutvecklingslivscykel

FL-2.1.1 (K2) Förklara vilken inverkan den valda utvecklingslivscykeln för programvara har på testningen

FL-2.1.2 (K1) Komma ihåg bra testpraxis som gäller för alla utvecklingslivscykler för programvara

FL-2.1.3 (K1) Komma ihåg exempel på angreppssättet för test-first i utvecklingen

FL-2.1.4 (K2) Summera hur DevOps kan ha en påverkan på testningen

FL-2.1.5 (K2) Förklara angreppssättet shift-left

FL-2.1.6 (K2) Förklara hur retrospektiv kan användas som en mekanism för processförbättringar

2.2 Testnivåer och testtyper

FL-2.2.1 (K2) Särskilja de olika testnivåerna

FL-2.2.2 (K2) Särskilja de olika testtyperna

FL-2.2.3 (K2) Skilja på omtestning och regressionstestning

2.3 Underhållstestning

FL-2.3.1 (K2) Sammanfatta underhållstestning och dess utlösande faktorer

2.1. Testning inom ramen för en programvaruutvecklingslivscykel

En modell för en programvaruutvecklingslivscykel (SDLC) är en abstrakt högnivå-representation av programvaruutvecklingsprocessen. En SDLC-modell definierar hur olika utvecklingsfaser och typer av aktiviteter som utförs inom denna process relaterar till varandra både logiskt och kronologiskt. Exempel på SDLC-modeller inkluderar sekventiella utvecklingsmodeller (t.ex. vattenfallsmodell, V-modell), iterativa utvecklingsmodeller (t.ex. spiralmodell, prototyping) och inkrementella utvecklingsmodeller (t.ex. Unified Process).

Vissa aktiviteter inom programvaruutvecklingsprocesser kan också beskrivas med mer detaljerade programvaruutvecklingsmetoder och agila praxis. Exempel är acceptanstestdriven utveckling (ATDD), beteendedriven utveckling (BDD), domändriven design (DDD), extreme programming (XP), funktionsdriven utveckling (FDD), Kanban, Lean IT, Scrum och testdriven utveckling (TDD).

2.1.1. Programvaruutvecklingslivscykelns påverkan på testningen

Testning måste anpassas till SDLC för att lyckas. Valet av SDLC påverkar:

- Omfattning och samordning av testaktiviteter (t.ex. testnivåer och testtyper)
- Detaljnivå på testdokumentationen
- Val av testteknik och testangreppssätt
- Omfattning av testautomatisering
- En testares roll och ansvar

I de inledande faserna i sekventiella utvecklingsmodeller deltar testare vanligtvis i kravgenomgångar, testanalys och testdesign. Den exekverbara koden skapas vanligtvis i de senare faserna, så därför kan dynamisk testning normalt inte utföras tidigt i SDLC.

I vissa iterativa och inkrementella utvecklingsmodeller antas det att varje iteration resulterar i en fungerande prototyp eller ett produktinkrement. Detta innebär att både statisk och dynamisk testning i varje iteration kan utföras på alla testnivåer. Frekventa leveranser av inkrement kräver snabb återkoppling och omfattande regressionstestning.

Agil programvaruutveckling förutsätter att förändringar kan ske under hela projektet. Därför föredras lättviktsdokumentation för arbetsprodukter och omfattande testautomatisering, detta för att göra regressionstestning enklare. Dessutom tenderar de flesta manuella tester att utföras med erfarenhetsbaserade testtekniker (se kapitel 4.4) som inte kräver att omfattande testanalys och design genomförts innan.

2.1.2. Programvaruutvecklingslivscykeln och bra testpraxis

Bra testpraxis, oberoende av den valda SDLC-modellen, inkluderar följande:

- För varje programvaruutvecklingsaktivitet finns en motsvarande testaktivitet, så att alla utvecklingsaktiviteter är föremål för kvalitetskontroll
- Olika testnivåer (se kapitel 2.2.1) har specifika och olika testmål, vilket gör att testningen blir tillräckligt omfattande samtidigt som man undviker redundans
- Testanalys och design för en given testnivå påbörjas under motsvarande utvecklingsfas av SDLC, så att testning kan följa principen om tidig testning (se kapitel 1.3)
- Testare är involverade i att granska arbetsprodukter så snart utkast till dokumentation finns tillgängligt, så att tidig testning och felupptäckt kan stödja strategin för shift-left (se kapitel 2.1.5)

2.1.3. Testning som en drivkraft för programvaruutveckling

TDD, ATDD och BDD är liknande angreppssätt för utvecklingen, där tester definieras som ett sätt att styra utvecklingen. Dessa angreppssätt realiserar principen om tidig testning (se kapitel 1.3) och följer ett shift-left-angreppssätt (se kapitel 2.1.5), eftersom testerna definieras innan koden skrivs. De stöder en iterativ utvecklingsmodell. Dessa angreppssätt kännetecknas enligt följande:

Testdriven utveckling (TDD):

- Styr kodningen genom testfall (istället för omfattande programvarudesign) (Beck 2003)
- Tester skrivs först, sedan skrivs koden för att uppfylla testerna, och sedan omstruktureras testerna och koden

Acceptanstestdriven utveckling (ATDD) (se kapitel 4.5.3):

- Tester härleds från acceptanskriterier som en del av systemdesignprocessen (Gärtner 2011)
- Tester skrivs innan den del av applikationen som ska uppfylla testerna utvecklas

Beteendedriven utveckling (BDD):

- Uttrycker det önskade beteendet hos en applikation med hjälp av testfall skrivna i en enkel form av naturligt språk som är lätt att förstå av intressenter – vanligtvis enligt formatet Given/When/Then. (Chelimsky 2010)
- Testfall översätts sedan automatiskt till exekverbara tester

För alla ovanstående angreppssätt gäller att testfallen kan bibehållas som automatiserade testfall för att säkerställa kodkvaliteten vid framtida anpassningar/omstruktureringar.

2.1.4. DevOps och testning

DevOps är ett organisatoriskt angreppssätt som syftar till att skapa synergi genom att få utveckling (inklusive testning) och drift att samverka för att uppnå gemensamma mål. DevOps kräver en kulturell förändring inom en organisation för att överbrygga klyftorna mellan utveckling (inklusive testning) och drift samtidigt som deras funktioner behandlas lika. DevOps främjar teamautonomi, snabb återkoppling, integrerade verktygskedjor och teknisk praxis som kontinuerlig integration (CI) och kontinuerlig leverans (CD). Detta gör det möjligt för teamen att bygga, testa och releasa högkvalitativ kod snabbare genom en DevOps-leveranspipeline (Kim 2016).

Ur ett testperspektiv är några av fördelarna med DevOps:

- Snabb återkoppling på kodens kvalitet och om ändringar påverkar befintlig kod negativt
- CI främjar ett shift-left-synsätt vid testning (se kapitel 2.1.5) genom att uppmuntra utvecklare att tillgängliggöra sin högkvalitativa kod tillsammans med komponenttester och statisk analys
- Främjar automatiserade processer som i CI/CD, vilka handhar etablering av stabila testmiljöer
- Ökar insikten om icke-funktionella kvalitetsegenskaper (t.ex. prestanda, tillförlitlighet)
- Automatisering genom en leveranspipeline minskar behovet av upprepade manuella tester
- Risken vid regression minimeras på grund av omfattningen av automatiserade regressionstester

DevOps är inte utan sina risker och utmaningar, som omfattar:

- DevOps leveranspipeline måste definieras och upprättas
- CI/CD-verktyg måste introduceras och underhållas

- Testautomatisering kräver ytterligare resurser och kan vara svår att etablera och underhålla

Även om DevOps kommer med en hög nivå av automatiserad testning, så kommer manuell testning – särskilt ur användarens perspektiv – fortfarande att behövas.

2.1.5. Angreppssättet shift-left

Principen för tidig testning (se kapitel 1.3) benämns ibland som shift-left eftersom det är ett angreppssätt där testning genomförs tidigare i SDLC. Shift-left föreslår normalt att testning ska göras tidigare (dvs. inte vänta på att kod ska implementeras eller på att komponenter ska integreras), men det betyder inte att testning senare i SDLC ska negligeras.

Det finns några bra metoder som illustrerar hur man uppnår “shift-left” i testningen:

- Granska specifikationer ur ett testperspektiv. Dessa granskningsaktiviteter hittar ofta potentiella defekter som oklarheter, ofullständigheter och inkonsekvenser
- Skriva testfall innan koden skrivs och exekvera koden på en testexekveringsplattform under kodimplementeringen
- Använda kontinuerlig integration (CI), eller ännu bättre kontinuerlig leverans (CD), eftersom de levererar snabb återkoppling och automatiserade komponenttester som följer med källkoden när den skickas till en koddatabas
- Slutföra statisk analys av källkoden innan dynamisk testning, eller som en del av en automatiserad process
- Genomföra icke-funktionella tester med början på komponenttestnivå, där så är möjligt. Detta är en form av shift-left eftersom dessa icke-funktionella testtyper tenderar att utföras senare i SDLC när ett komplett system och en representativ testmiljö är tillgänglig

Ett shift-left-angreppssätt kan resultera i mer utbildning, arbetsinsats och/eller kostnader tidigare i processen men förväntas spara insatser och/eller kostnader senare i processen.

För att genomföra ett angreppssättet shift-left är det viktigt att intressenterna övertygas och tror på detta koncept.

2.1.6. Retrospektiver och processförbättringar

Retrospektiver (även kända som "post-projektmöten" eller projektretrospektiv) hålls ofta i slutet av ett projekt eller en iteration, vid en release-milstolpe men kan också hållas vid behov. Timing och organisation av retrospektiverna beror på den SDLC-modell som följs. I dessa möten diskuterar deltagarna (inte bara testare, utan även t.ex. utvecklare, arkitekter, produktägare, företagsanalytiker) följande:

- Vad var framgångsrikt och bör bibehållas?
- Vad var inte framgångsrikt och vad kan förbättras?
- Hur införlivar vi förbättringarna och bibehåller effekten av dem framöver?

Resultaten ska dokumenteras och är normalt en del av den sammanfattande testrapporten (se kapitel 5.3.2). Retrospektiver är avgörande för ett framgångsrikt genomförande av ständiga förbättringar och det är viktigt att alla rekommenderade förbättringar följs upp.

Typiska fördelar med retrospektiv för testningen inkluderar:

- Ökad testeffektivitet (t.ex. genom att implementera processförbättringsförslag)
- Ökad kvalitet på testvaran (t.ex. genom att gemensamt granska testprocesserna)

- Förstärkta relationerna i teamet och lärande (t.ex. som ett resultat av möjligheten att ta upp frågor och föreslå förbättringspunkter)
- Förbättrad kvalitet på testbasen (eftersom brister i t.ex. kravens omfattning och kvalitet skulle kunna åtgärdas och lösas)
- Bättre samarbete mellan utveckling och testning (bland annat genom att samarbetet regelbundet ses över och optimeras)

2.2. Testnivåer och testtyper

Testnivåer är grupper av testaktiviteter som organiseras och hanteras tillsammans. Varje testnivå är en instans av testprocessen, som utförs i relation till programvaran i ett givet utvecklingsskede, från enskilda komponenter till kompletta system eller, i förekommande fall, system av system.

Testnivåer är relaterade till andra aktiviteter inom SDLC. I sekventiella SDLC-modeller är testnivåerna ofta definierade så att avslutskriterierna för en nivå är del av startkriterierna för nästa nivå. I vissa iterativa modeller kanske detta inte gäller. Utvecklingsaktiviteter kan sträcka sig över flera testnivåer. Testnivåer kan tidsmässigt överlappa varandra.

Testtyper är grupper av testaktiviteter relaterade till specifika kvalitetsegenskaper och de flesta av dessa testaktiviteter kan utföras på varje testnivå.

2.2.1. Testnivåer

I denna kursplan beskrivs följande fem testnivåer:

- **Komponenttestning** (även känd som enhetstestning) fokuserar på att testa isolerade komponenter. Den kräver ofta specifikt stöd, som testexekveringsplattform eller enhetstestramverk. Komponenttestning utförs normalt av utvecklare i sina utvecklingsmiljöer.
- **Komponentintegreringstestning** (även känd som enhetsintegrationstestning) fokuserar på att testa gränssnitt och interaktioner mellan komponenter. Komponentintegreringstestning är starkt beroende av integrationsstrategins angreppssätt som bottom-up, top-down eller big bang.
- **Systemtestning** fokuserar på det övergripande beteendet och förmågan hos ett helt system eller produkt, ofta med testning av end-to-end-funktionalitet och icke-funktionell testning av kvalitetsegenskaper. För vissa icke-funktionella kvalitetsegenskaper är det att föredra att testa dem i ett komplett system i en representativ testmiljö (t.ex. användbarhet). Det är också möjligt att använda simuleringar av delsystem. Systemtestning kan genomföras av ett oberoende testteam och är relaterat till specifikationer för systemet.
- **Systemintegrationstestning** fokuserar på att testa gränssnitten mellan det system som testas och andra system eller externa tjänster. Systemintegrationstestning kräver lämpliga testmiljöer som helst liknar den operativa miljön.
- **Acceptanstestning** fokuserar på validering och på att påvisa att systemet är redo för driftsättning, vilket innebär att systemet uppfyller användarens affärsbehov. Helst bör acceptanstestning utföras av de kommande användarna. De huvudsakliga formerna av acceptanstestning är användaracceptanstestning (UAT), driftsacceptanstestning, acceptanstestning av kontrakt och förordningar samt alfa- och betatestning.

Testnivåer utmärks av följande (inte kompletta) lista över attribut, i syfte att undvika överlappning av testaktiviteter:

- Testobjekt
- Testmål
- Testbas
- Defekter och felsymptom
- Angreppssätt och ansvar

2.2.2. Testtyper

Det finns många testtyper som kan användas i projekt. I denna kursplan tas följande fyra testtyper upp:

Funktionstestning utvärderar de funktioner som en komponent eller ett system ska utföra. Funktionerna är "vad" testobjektet ska göra. Huvudsyftet med funktionstestningen är att kontrollera funktionell kompletthet, funktionell korrekthet och funktionell ändamålsenlighet.

Icke-funktionell testning utvärderar andra attribut än funktionella egenskaper hos en komponent eller ett system. Icke-funktionell testning är testning av "hur väl systemet beter sig". Huvudsyftet med icke-funktionell testning är att kontrollera programvarans icke-funktionella kvalitetsegenskaper. ISO/IEC 25010-standarden tillhandahåller följande klassificering av programvarans icke-funktionella kvalitetsegenskaper:

- Prestandaeffektivitet
- Kompatibilitet
- Användbarhet
- Tillförlitlighet
- Säkerhet
- Underhållbarhet
- Portabilitet

Det är ibland lämpligt att börja med icke-funktionell testning tidigt i livscykeln (t.ex. som en del av granskningar och komponenttestning eller systemtestning). Många icke-funktionella tester härleds från funktionstesterna eftersom de testar samma funktion, men fokuserar då på icke-funktionella egenskaper när funktionen exekveras (t.ex. att kontrollera att en funktion utförs inom en angiven tid, eller att en funktion kan överföras till en ny plattform). En sen upptäckt av icke-funktionella defekter kan utgöra ett allvarligt hot mot ett projekts framgång. Icke-funktionell testning behöver ibland en mycket specifik testmiljö, till exempel ett användbarhetslabb för användbarhetstestning.

Black-box-testning (se kapitel 4.2) är specifikationsbaserad och härleder tester från dokumentation utanför testobjektet. Huvudsyftet med black-box-testning är att utvärdera systemets beteende mot dess specifikationer.

White-box-testning (se kapitel 4.3) är strukturbaserad och härleder tester från systemets implementering eller interna struktur (t.ex. kod, arkitektur, arbetsflöden och dataflöden). Huvudsyftet med white-box-testning är att med tester täcka den underliggande strukturen till en acceptabel nivå.

Alla de fyra ovannämnda testtyperna kan tillämpas på alla testnivåer, även om fokus kommer att skilja sig mellan nivåerna. Olika testtekniker kan användas för att härleda testvillkoren och testfall för alla nämnda testtyper.

2.2.3. Omtestning och regressionstestning

Ändringar görs vanligtvis i en komponent eller i ett system för att förbättra det genom att lägga till en ny funktion eller genom att åtgärda en defekt. Testning bör då också omfatta omtestning och regressionstestning.

Omtestning bekräftar att ett ursprungligt fel har åtgärdats. Beroende på risken kan man testa den korrigerade versionen av programvaran på flera sätt, inklusive:

- exekvera alla testfall som tidigare har misslyckats på grund av defekten, men även
- lägga till nya testfall för att täcka eventuella ändringar som behövdes för att åtgärda defekten

Men när tiden eller pengarna är knappa för att åtgärda defekter kan omtestning begränsas till att helt enkelt utföra de steg som ska återskapa felsymptomen som orsakades av defekten för att kontrollera att felet inte längre finns.

Regressionstestning bekräftar att inga negativa konsekvenser har orsakats av en förändring, inklusive den rättning som redan har omtestats. Dessa negativa konsekvenser kan påverka samma komponent där ändringen gjordes, andra komponenter i samma system eller till och med andra anslutna system. Regressionstestning behöver inte vara begränsad till själva testobjektet utan kan också relateras till miljön. Det är lämpligt att först göra en påverkansanalys för att visa vilka delar av programvaran som påverkats och för att optimera omfattningen av regressionstestningen.

Regressionstestsviter körs många gånger och i allmänhet kommer antalet regressionstestfall att öka med varje iteration eller release. Regressionstestning är därför en stark kandidat för automatisering. Automatisering av dessa tester bör starta tidigt i projektet. Där CI används, som i DevOps (se kapitel 2.1.4), är det också god praxis att inkludera automatiserade regressionstester. Beroende på situationen kan regressionstesterna vara på olika nivåer.

Omtestning och/eller regressionstestning av testobjektet behövs på alla testnivåer när defekter har åtgärdats och/eller ändringar gjorts på dessa testnivåer.

2.3. Underhållstestning

Det finns olika kategorier av underhåll, det kan vara korrigeringar, anpassningar till förändringar i miljön och förbättringar av prestanda eller underhållbarhet (se ISO/IEC 14764 för detaljer). Därför kan underhåll omfatta planerade releaser/driftsättning och oplanerade releaser/driftsättningar (hot fixes). En påverkansanalys kan göras först för att avgöra om förändringen ska genomföras, baserat på de potentiella konsekvenserna i andra delar av systemet. Att testa ändringar i ett system i produktion innefattar både att utvärdera resultatet av implementeringen av förändringen och att kontrollera eventuella regressioner i de delar av systemet som är oförändrade (vilket vanligtvis är större delen av systemet).

Omfattningen av underhållstestning beror vanligtvis på:

- Graden av risk vid förändringen
- Storleken på det befintliga systemet
- Storleken på förändringen

Utlösande faktorer för underhåll och underhållstestning kan klassificeras enligt följande:

- Ändringar, såsom planerade förbättringar (d.v.s. releasebaserade), korrigerande ändringar eller snabbkorrigeringar (hot fixes).
- Uppgradering eller migrering av den operativa miljön, till exempel från en plattform till en annan, kan kräva testning kopplat till den nya miljön såväl som av den ändrade

programvaran, eller testning av datakonvertering när data från en annan applikation migreras in i systemet som ska underhållas.

- Avveckling, till exempel när en applikation når slutet av sin livslängd. När ett system är avvecklat kan detta kräva testning av dataarkivering om långa datalagringsperioder krävs. Testning av återställnings- och hämtningsprocedurer efter arkivering kan också behövas om vissa data senare skulle behövas under arkiveringsperioden.

3. Statisk testning – 80 minuter

Nyckelord

avvikelse, dynamisk testning, formell granskning, genomgång, granskning, informell granskning, inspektion, statisk analys, statisk testning, teknisk granskning

Utbildningsmål för kapitel 3:

3.1 Grunder för statisk testning

FL-3.1.1 (K1) Identifiera de produkttyper som kan granskas med de olika statiska testteknikerna

FL-3.1.2 (K2) Beskriva värdet med statisk testning

FL-3.1.3 (K2) Jämföra likheter och skillnader mellan statisk och dynamisk testning

3.2 Granskningsprocess och återkoppling

FL-3.2.1 (K1) Identifiera fördelarna med tidig och frekvent återkoppling från intressenter

FL-3.2.2 (K2) Summera aktiviteterna i granskningsprocessen

FL-3.2.3 (K1) Känna igen vilka ansvarsområden de viktigaste rollerna har vid granskningar

FL-3.2.4 (K2) Jämföra likheter och skillnader mellan de olika granskningstyperna

FL-3.2.5 (K1) Känna igen de faktorer som bidrar till en framgångsrik granskning

3.1. Grunder för statisk testning

I motsats till dynamisk testning behöver man inte exekvera programvaran vid statisk testning. Kod, processbeskrivning, systemarkitekturspecifikation eller andra arbetsprodukter utvärderas genom manuell kontroll (t.ex. granskning) eller med hjälp av ett verktyg (t.ex. statisk analys). Testmålsättningen är att förbättra kvaliteten, upptäcka defekter och bedöma egenskaper som läsbarhet, fullständighet, korrekthet, testbarhet och konsistens. Statisk testning kan användas både vid verifiering och validering.

Testare, verksamhetsrepresentanter och utvecklare samarbetar under exempelkartläggningar, kollaborativt skrivande av användarberättelser och förfiningssessioner av backloggen för att säkerställa att användarberättelser och relaterade arbetsprodukter uppfyller definierade kriterier, t.ex. Definition of Ready (se kapitel 5.1.3). Granskningstekniker kan användas för att säkerställa att användarberättelser är fullständiga och begripliga och att de inkluderar testbara acceptanskriterier. Genom att ställa rätt frågor kan testare utforska, utmana och hjälpa till att förbättra de föreslagna användarberättelserna.

Statisk analys kan identifiera problem tidigare än dynamisk testning och kräver ofta en mindre arbetsinsats eftersom inga testfall behövs och man vanligtvis använder verktyg (se kapitel 6). Statisk analys är ofta en del av ett CI-ramverk (se kapitel 2.1.4). Även om statisk analys till stor del används för att upptäcka specifika defekter i koden, så kan den också användas för att utvärdera underhållbarhet och säkerhet. Stavningskontroll och utvärdering av läsbarheten är andra exempel på statiska analysverktyg.

3.1.1. Arbetsprodukter som kan utvärderas med statisk testning

Nästan alla arbetsprodukter kan utvärderas med statisk testning. Som exempel kan nämnas kravspecifikationsdokument, källkod, testplaner, testfall, produktbacklog, testcharter, projektdokumentation, avtal och modeller.

Varje arbetsprodukt som kan läsas och förstås kan bli föremål för en granskning. Men för statisk analys behöver arbetsprodukterna ha en struktur mot vilken de kan kontrolleras (t.ex. modeller, kod eller text med en formell syntax).

Arbetsprodukter som inte är lämpliga för statisk testning är sådana som är svåra att tolka av människor och som inte bör analyseras med verktyg (t.ex. exekverbar kod från tredje part på grund av juridiska skäl).

3.1.2. Värdet med statisk testning

Statisk testning kan upptäcka defekter i de tidigaste faserna av SDLC, vilket uppfyller principen om tidig testning (se kapitel 1.3). Statisk testning kan också identifiera defekter som inte kan upptäckas genom dynamisk testning (t.ex. oåtkomlig kod, designmönster som inte implementeras som önskat, defekter i icke exekverbara arbetsprodukter).

Statisk testning möjliggör utvärdering av kvaliteten på arbetsprodukterna och att bygga förtroende för dem. Genom att verifiera de dokumenterade kraven kan intressenterna försäkra sig om att dessa krav beskriver deras faktiska behov. Eftersom statisk testning kan utföras tidigt i SDLC, kan en gemensam förståelse skapas mellan de inblandade intressenterna. Kommunikationen förbättras också mellan dessa. Av den anledningen rekommenderas det att engagera en bred mångfald av intressenter i statisk testning.

Även om granskningar kan vara kostsamma att genomföra är de övergripande projektkostnaderna vanligtvis mycket lägre än när inga granskningar utförs, eftersom mindre tid och arbete behöver läggas på att åtgärda defekter senare i projektet.

Defekter i koden kan upptäckas mer effektivt med statisk analys än vid dynamisk testning, vilket vanligtvis resulterar i både färre defekter i koden och lägre total utvecklingsinsats.

3.1.3. Skillnader mellan statisk och dynamisk testning

Statisk testning och dynamisk testning kompletterar varandra. De har liknande syften, som att stödja upptäckten av defekter i arbetsprodukter (se kapitel 1.1.1), men det finns också vissa skillnader, som:

- Statisk och dynamisk testning (med analys av felsymptom) kan båda leda till upptäckt av defekter, men det finns vissa defekttyper som endast kan hittas genom antingen statisk eller dynamisk testning.
- Statisk testning upptäcker defekter direkt, medan dynamisk testning upptäcker felsymptom från vilka de orsakande defekterna härleds genom efterföljande analys
- Statisk testning kan lättare upptäcka defekter som ligger i kodgrenar som sällan exekveras eller är svåra att nå med dynamisk testning
- Statisk testning kan tillämpas på icke-exekverbara arbetsprodukter, medan dynamisk testning endast kan tillämpas på exekverbara arbetsprodukter
- Statisk testning kan användas för att mäta kvalitetsegenskaper (t.ex. underhållbarhet) som inte är beroende av exekvering av kod medan dynamisk testning kan användas för att mäta kvalitetsegenskaper (t.ex. prestandaeffektivitet) som är beroende av exekvering av kod

Typiska defekter som är lättare och/eller billigare att hitta genom statisk testning inkluderar:

- Defekter i krav (t.ex. inkonsekvenser, tvetydigheter, motsägelser, utelämnanden, felaktigheter, dubbletter)
- Designfel (t.ex. ineffektiva databasstrukturer, dålig modularisering)
- Vissa typer av koddefekter (t.ex. variabler med odefinierade värden, odeklarerade variabler, oåtkomlig eller duplicerad kod, överdriven kodkomplexitet)
- Avvikelser från standarder (t.ex. bristande efterlevnad av namnkonventioner i kodningsstandarder)
- Felaktiga gränssnittsspecifikationer (t.ex. felaktigt antal, typ eller ordning av parametrar)
- Specifika typer av säkerhetsbrister (t.ex. buffertöverskridning)
- Luckor eller felaktigheter i täckning av testbasen (t.ex. saknade testfall för ett acceptanskriterium)

3.2. Granskningsprocess och återkoppling

3.2.1. Fördelar med tidiga och frekventa återkopplingar från intressenter

Tidig och frekvent återkoppling möjliggör tidig kommunikation om potentiella kvalitetsproblem. Om det finns lite engagemang från intressenterna under SDLC kanske produkten som utvecklas inte uppfyller intressentens ursprungliga eller nuvarande vision. Ett misslyckande med att leverera vad intressenten önskar kan resultera i kostsamma omarbetningar, missade deadlines, skuldbeläggningar och kan till och med leda till fullständigt projektmisslyckande.

Frekventa återkopplingar från intressenter genom hela SDLC kan förhindra missförstånd om krav och säkerställa att kravändringar förstås och implementeras tidigare. Detta hjälper utvecklingsteamet att förbättra förståelsen för vad de bygger. Det låter dem fokusera på de funktioner som ger störst värde för intressenterna och som har den mest positiva inverkan på identifierade risker.

3.2.2. Aktiviteter i granskningsprocessen

ISO/IEC 20246-standarden definierar en generisk granskningsprocess som ger ett strukturerat men flexibelt ramverk från vilket en specifik granskningsprocess kan skräddarsys för en viss situation. Om den begärda granskningen är mer formell, kommer fler av de beskrivna arbetsuppgifterna för de olika aktiviteterna att behövas.

Storleken på många arbetsprodukter kan göra dem för stora för att genomföras i en granskning. Granskningsprocessen kan då upprepas flera gånger för att slutföra granskningen för hela arbetsprodukten.

Aktiviteterna i granskningsprocessen är:

- **Planering**. Under planeringsfasen ska omfattningen av granskningen definieras. Det gäller syftet, arbetsprodukten som ska granskas, kvalitetsegenskaper som ska utvärderas, områden att fokusera på, avslutskriterier, stödjande information som standarder, arbetsinsats och tidsramar för granskningen.
- **Initiera granskningen**. Vid granskningsinitieringen är målet att se till att allt och alla inblandade är förberedda för att påbörja granskningen. Det innebär att se till att varje deltagare har tillgång till arbetsprodukten som granskas, förstår sin roll och sitt ansvar och har fått allt som behövs för att utföra granskningen.
- **Individuell granskning.** Varje granskare genomför en individuell granskning för att bedöma kvaliteten på arbetsprodukten och för att identifiera avvikelser, rekommendationer och frågor genom att tillämpa en eller flera granskningstekniker (t.ex. checklistebaserad granskning, scenariobaserad granskning). ISO/IEC 20246-standarden ger mer djupgående information om olika granskningstekniker. Granskarna loggar alla sina identifierade avvikelser, rekommendationer och frågor.
- **Kommunikation och analys.** Eftersom de avvikelser som identifierats under en granskning inte nödvändigtvis är defekter måste alla dessa avvikelser analyseras och diskuteras. För varje avvikelse ska beslut fattas om dess status, ägande och nödvändiga åtgärder. Detta görs vanligtvis i ett granskningsmöte, där deltagarna också beslutar om kvalitetsnivån på granskad arbetsprodukt och vilka uppföljningsåtgärder som krävs. En uppföljande granskning kan krävas för att slutföra åtgärderna.
- **Åtgärda och rapportera.** För varje defekt ska en felrapport skrivas så att korrigerande åtgärder kan följas upp. När avslutskriterierna har uppnåtts kan arbetsprodukten accepteras. Granskningsresultaten redovisas.

3.2.3. Roller och ansvar i granskningar

Granskningar involverar olika intressenter, som kan ta på sig flera roller. De huvudsakliga rollerna och deras ansvarsområden är:

- Chef – bestämmer vad som ska granskas och tillhandahåller resurser, såsom personal och tid för granskning
- Författare – skapar och ordnar arbetsprodukten som granskas
- Moderator (även känd som facilitator) – säkerställer effektivt genomförande av granskningsmöten, inklusive medling, tidshantering och en säker granskningsmiljö där alla kan tala fritt
- Sekreterare – sammanställer avvikelser från granskarna och dokumenterar granskningsinformation, såsom beslut och nya avvikelser som upptäcktes under granskningsmötet

- Granskare – utför granskningar. En granskare kan vara någon som arbetar i projektet, en ämnesexpert eller någon annan intressent
- Granskningsledare – tar ett övergripande ansvar för granskningen som att bestämma vem som ska delta i mötet och organisera när och var granskningen ska ske

Andra, mer detaljerade roller är möjliga, som beskrivs i ISO/IEC 20246-standarden.

3.2.4. Granskningstyper

Det finns många granskningstyper, allt från informella till formella granskningar. Den önskade formalitetsnivån beror på faktorer, till exempel vilken SDLC som följs, utvecklingsprocessens mognad, den kritiska och komplexa egenskapen hos den arbetsprodukt som granskas, juridiska eller reglerade krav och behovet av en revisionslogg. Samma arbetsprodukt kan granskas med olika granskningstyper, t.ex. först en informell och senare en mer formell.

Att välja rätt granskningstyp är nyckeln till att uppnå de önskade granskningsmålen (se kapitel 3.2.5). Urvalet baseras inte bara på målen utan också på faktorer som projektbehov, tillgängliga resurser, typ av arbetsprodukt, risker, affärsdomän och företagskultur.

Några vanliga granskningstyper är:

- **Informell granskning.** Informella granskningar följer inte en definierad process och kräver inte något formellt dokumenterade utdata. Huvudsyftet är att upptäcka avvikelser.
- **Genomgång**. En genomgång, som leds av författaren, kan tjäna många syften, som att utvärdera kvaliteten och bygga förtroende för arbetsprodukten, utbilda granskare, uppnå konsensus, generera nya idéer, motivera och göra det möjligt för författare att förbättra sig samt upptäcka avvikelser. Granskare kan utföra en individuell granskning innan genomgången, men detta är inget krav.
- **Teknisk granskning.** En teknisk granskning utförs av tekniskt kvalificerade granskare och leds av en moderator. Syftet med en teknisk granskning är att uppnå konsensus och fatta beslut angående ett tekniskt problem, men också att upptäcka avvikelser, utvärdera kvalitet och bygga förtroende för arbetsprodukten, generera nya idéer och att motivera och möjliggöra för författare att förbättra sig.
- **Inspektion**. Eftersom inspektioner är den mest formella typen av granskning följer den kompletta generiska processen (se kapitel 3.2.2). Huvudsyftet är att hitta maximalt antal avvikelser. Andra målsättningar är att utvärdera kvalitet, bygga förtroende för arbetsprodukten och att motivera och göra det möjligt för författarna att förbättra sig. Mätvärden samlas in och används för att förbättra SDLC, inklusive inspektionsprocessen. Vid en inspektion kan författaren inte fungera som granskningsledare eller sekreterare.

3.2.5. Framgångsfaktorer för granskningar

Det finns flera faktorer som avgör om granskningar är framgångsrika:

- Att definiera tydliga mål och mätbara avslutskriterier. Utvärdering av deltagare ska aldrig vara ett mål
- Att välja lämplig granskningstyp för att uppnå de satta målen och för att anpassa arbetsprodukttyp, granskningsdeltagare, projektets behov och till sammanhanget
- Att genomföra granskningar i små bitar, så att granskarna inte tappar koncentrationen under en individuell granskning och/eller granskningsmötet (när det hålls)

- Att ge återkoppling från granskningar till intressenter och författare så att de kan förbättra produkten och sina aktiviteter (se kapitel 3.2.1)
- Att ge deltagarna tillräckligt med tid för att förbereda sig för granskningen
- Att få stöd från ledningen under granskningsprocessen
- Att göra granskningar till en del av organisationens kultur för att främja lärande och processförbättringar
- Att ge tillräcklig utbildning till alla deltagare så att de vet hur de ska uppfylla sin roll
- Att moderera eller facilitera möten

4. Testanalys och design – 390 minuter

Nyckelord

acceptanskriterier, acceptanstestdriven utveckling, black-box-testteknik, checklistebaserad testning, ekvivalensklassindelning, erfarenhetsbaserad testteknik, felgissning, gränsvärdesanalys, kodgrenstäckning, kodsatstäckning, samarbetsbaserat testangreppssätt, täckningsobjekt, testning med hjälp av beslutstabeller, testteknik, tillståndsbaserad testning, utforskande testning, white-box-testteknik

Utbildningsmål för kapitel 4:

4.1 Översikt över testtekniker

FL-4.1.1 (K2) Skilj mellan black-box-, white-box- och erfarenhetsbaserade testtekniker

4.2 Black-box-testtekniker

FL-4.2.1 (K3) Använda ekvivalensklassindelning för att härleda testfall

FL-4.2.2 (K3) Använda gränsvärdesanalys för att härleda testfall

FL-4.2.3 (K3) Använda testning med hjälp av beslutstabeller för att härleda testfall

FL-4.2.4 (K3) Använda tillståndsbaserad testning för att härleda testfall

4.3 White-box-testtekniker

FL-4.3.1 (K2) Förklara kodsatstestning

FL-4.3.2 (K2) Förklara kodgrenstestning

FL-4.3.3 (K2) Förklara värdet med white-box-testning

4.4 Erfarenhetsbaserade testtekniker

FL-4.4.1 (K2) Förklara felgissning

FL-4.4.2 (K2) Förklara utforskande testning

FL-4.4.3 (K2) Förklara checklistebaserad testning

4.5. Samarbetsbaserade testangreppssätt

FL-4.5.1 (K2) Förklara hur användarberättelser skrivs i samarbete med utvecklare och verksamhetsrepresentanter

FL-4.5.2 (K2) Klassificera de olika alternativen för att skriva acceptanskriterier

FL-4.5.3 (K3) Använda acceptanstestdriven utveckling (ATDD) för att härleda testfall

4.1. Översikt över testtekniker

Testtekniker stödjer testaren i testanalys (vad man ska testa) och i testdesign (hur man testar). Testtekniker hjälper till att utveckla en relativt liten, men tillräcklig, uppsättning testfall på ett systematiskt sätt. Testtekniker hjälper också testaren med att definiera testvillkor, identifiera täckningsobjekt och identifiera testdata under testanalys och design. Ytterligare information om testtekniker och deras motsvarande mätetal finns i ISO/IEC/IEEE 29119-4-standarden och i Beizer 1990, Craig 2002, Copeland 2004, Koomen 2006, Jorgensen 2014, Ammann 2016, Forgács 2019.

I denna kursplan klassificeras testteknikerna som black-box, white-box och erfarenhetsbaserade.

Black-box-testtekniker (även kända som specifikationsbaserade tekniker) baseras på en analys av testobjektets specificerat beteende, utan referens till dess interna struktur. Därför är testfallen oberoende av hur programvaran är implementerad. Följaktligen är testfallen fortfarande användbara även om implementeringen ändras men det kravställda beteendet förblir detsamma,.

White-box-testtekniker (även kända som strukturbaserade tekniker) baseras på en analys av testobjektets interna struktur och exekvering. Eftersom testfallen är beroende av hur programvaran är designad kan de bara skapas efter design eller implementering av testobjektet.

Erfarenhetsbaserade testtekniker använder effektivt testarnas kunskap och erfarenhet för design och implementering av testfall. Effektiviteten hos dessa tekniker beror mycket på testarens förmågor. Erfarenhetsbaserade testtekniker kan upptäcka defekter som kan ha missats med black-box- och white-box-testteknikerna. Följaktligen är erfarenhetsbaserade testtekniker komplementära till dessa.

4.2. Black-Box-testtekniker

Vanligt använda black-box-testtekniker som tas upp i följande delkapitel är:

- Ekvivalensklassindelning
- Gränsvärdesanalys
- Testning med hjälp av beslutstabeller
- Tillståndsbaserad testning

4.2.1. Ekvivalensklassindelning

Ekvivalensklassindelning (equivalence partition, EP) delar in data i klasser (kallas även ekvivalensklasser) baserat på förväntningen att alla element i en given klass ska behandlas på samma sätt av testobjektet. Teorin bakom denna teknik är att om ett testfall, som testar ett värde från en ekvivalensklass, upptäcker en defekt, bör denna defekt också upptäckas av testfall som testar något annat värde från samma klass. Därför räcker det med ett testfall för varje klass.

Ekvivalensklasser kan identifieras för alla dataelement som är relaterade till testobjektet, inklusive indata, utdata, konfigurationsobjekt, interna värden, tidsrelaterade värden och gränssnittsparametrar. Klasserna kan vara kontinuerliga eller diskreta, ordnade eller oordnade, ändliga eller oändliga. Klasserna får inte överlappa varandra och får inte vara tomma.

För enkla testobjekt kan ekvivalensklassindelningen vara enkel, men i praktiken är det ofta komplicerat att förstå hur testobjektet kommer att behandla olika värden. Därför måste klassindelningen göras med eftertanke.

En klass som innehåller giltiga värden kallas en giltig klass och en klass som innehåller ogiltiga värden kallas en ogiltig klass. Definitionerna av giltiga och ogiltiga värden kan variera mellan team och organisationer. Till exempel kan giltiga värden tolkas som de som ska bearbetas av testobjektet eller som de för vilka specifikationen definierar deras bearbetning. Ogiltiga värden kan tolkas som de

som ska ignoreras eller avvisas av testobjektet eller som de för vilka ingen bearbetning är definierad i testobjektspecifikationen.

I ekvivalensklassindelning är täckningsobjekten ekvivalensklasserna. För att uppnå 100 % täckning med denna teknik måste testfall innehålla alla identifierade klasser (inklusive ogiltiga klasser) genom att täcka varje klass minst en gång. Täckningsgraden mäts som antalet klasser som testats av minst ett testfall, dividerat med det totala antalet identifierade klasser, och uttrycks i procent.

Många testobjekt inkluderar flera uppsättningar av klasser (t.ex. testobjekt med mer än en indataparameter), vilket innebär att ett testfall kommer att täcka klasser från olika uppsättningar av klasser. Det enklaste täckningskriteriet i fallet med flera uppsättningar av klasser kallas Each Choice-täckning (Ammann 2016). Each Choice-täckning kräver att testfall testar varje klass från varje uppsättning klasser minst en gång. Each Choice-täckning tar inte hänsyn till kombinationer av klasser.

4.2.2. Gränsvärdesanslys

Gränsvärdesanslys (Boundary Value Analysis, BVA) är en teknik som bygger på att testa gränserna för ekvivalensklasser. Därför kan BVA endast användas för ordnade klasser. Lägsta och högsta värdena i en klass utgör dess gränsvärden. För BVA gäller att om två element tillhör samma klass så måste alla element mellan dem också tillhöra den klassen.

BVA fokuserar på klassernas gränsvärden för klasserna eftersom utvecklare är mer benägna att göra fel vid hantering av dessa gränsvärden. Typiska defekter som hittas av BVA är gränser som är felplacerade till positioner över eller under deras avsedda positioner, eller är helt utelämnade.

Denna kursplan omfattar två versioner av BVA: 2-värdes och 3-värdes BVA. De skiljer sig åt vad gäller täckningsobjekt per gräns som måste testas för att uppnå 100 % täckning.

I 2-värdes BVA (Craig 2002, Myers 2011) krävs det två täckningsobjekt för varje gränsvärde: detta gränsvärde och dess närmaste granne som tillhör den intilliggande klassen. För att uppnå 100 % täckning med 2-värdes BVA måste testfall testa alla täckningsobjekt, d.v.s. alla identifierade gränsvärden. Täckningen mäts som antalet testade gränsvärden dividerat med det totala antalet identifierade gränsvärden, och uttrycks i procent.

I 3-värdes BVA (Koomen 2006, O'Regan 2019) krävs det tre täckningsobjekt för varje gränsvärde: detta gränsvärde och båda dess grannar. Därför kan vissa av täckningsobjekten här inte vara gränsvärden. För att uppnå 100 % täckning med 3-värdes BVA måste testfall testa alla täckningsobjekt, d.v.s. identifierade gränsvärden och deras grannar. Täckningsgraden mäts som antalet testade gränsvärden och deras grannar, dividerat med det totala antalet identifierade gränsvärden och deras grannar, och uttrycks i procent.

3-värdes BVA är mer rigoröst än 2-värdes BVA eftersom det kan upptäcka defekter som förbises av 2-värdes BVA. Till exempel, om beslutspunkten "if (x ≤ 10) ..." är felaktigt implementerat som "if (x = 10) ...", kan inga testdata härledda från 2-värdes BVA (x = 10, x = 11) detektera defekten. Men x = 9, härledd från 3-värdes BVA, kommer sannolikt att upptäcka det.

4.2.3. Testning med hjälp av beslutstabeller

Beslutstabeller används för att testa implementeringen av systemkrav som anger hur olika kombinationer av villkor ger olika utfall. Beslutstabeller är ett effektivt sätt att registrera komplex logik, såsom verksamhetsregler.

När beslutstabeller skapas så definieras villkoren och de resulterande åtgärderna som ska utföras i systemet. Dessa bildar raderna i tabellen. Varje kolumn motsvarar en beslutsregel som definierar en unik kombination av villkor, tillsammans med tillhörande åtgärder. I beslutstabeller med begränsat indata visas alla värden för villkoren och åtgärderna (förutom irrelevanta eller omöjliga, se nedan) som booleska värden (sant eller falskt). I beslutstabeller med utökat indata kan alternativt vissa eller alla

villkor och åtgärder också anta flera värden (t.ex. numeriska intervall, ekvivalensklasser, diskreta värden).

Beteckningen för villkor är följande: "T" (true/sant) betyder att villkoret är uppfyllt. "F" (falskt) betyder att villkoret inte är uppfyllt. "–" betyder att värdet av villkoret är irrelevant för åtgärdsresultatet. "N/A" betyder att villkoret är omöjligt för en given regel. För åtgärder: "X" betyder det att åtgärden ska ske, Blank betyder att åtgärden inte ska ske. Andra notationer kan också användas.

En fullständig beslutstabell har tillräckligt med kolumner för att täcka varje kombination av villkor. Tabellen kan förenklas genom att ta bort kolumner som innehåller omöjliga kombinationer av villkor. Tabellen kan också minimeras genom att slå samman kolumner till en enda kolumn, där vissa förhållanden inte påverkar resultatet,. Algoritmer för minimering av beslutstabeller ligger utanför denna kursplan.

Vid testning med hjälp av beslutstabeller är täckningsobjekten de kolumner som innehåller möjliga kombinationer av villkor. För att uppnå 100 % täckning med denna teknik måste testfall testa alla dessa kolumner. Täckningen mäts som antalet testade kolumner, dividerat med det totala antalet möjliga kolumner, och uttrycks i procent.

Styrkan med testning med hjälp av beslutstabeller är att det ger ett systematiskt angreppssätt för att identifiera alla kombinationer av villkor, av vilka några annars skulle kunna förbises. Det hjälper också till att hitta eventuella luckor eller motsägelser i kraven. Om det finns många villkor kan det vara tidskrävande att testa alla beslutsregler eftersom antalet regler växer exponentiellt med antalet villkor. I ett sådant fall kan en minimerad beslutstabell eller ett riskbaserat angreppssätt användas för att minska antalet regler som behöver testas.

4.2.4. Tillståndsbaserad testning

Ett tillståndsdiagram modellerar beteendet hos ett system genom att visa dess möjliga tillstånd och giltiga tillståndsövergångar. En övergång initieras av en händelse, som dessutom kan vara styrd av ett logiskt villkor (vaktvillkor). Övergångarna antas vara omedelbara och kan ibland leda till att programvaran vidtar åtgärder. Den vanliga syntaxen för övergångar är: "händelse [vaktvillkor] / åtgärd". Vaktvillkor och åtgärder kan utelämnas om de inte finns eller är irrelevanta för testaren.

En tillståndstabell är en modell som motsvarar ett tillståndsdiagram. Dess rader representerar tillstånd och dess kolumner representerar händelser (tillsammans med vaktvillkor om de finns). Tabellposter (celler) representerar övergångar och innehåller måltillståndet, såväl som vaktvillkoren och resulterande åtgärder, om de är definierade. I motsats till tillståndsdiagrammet visar tillståndstabellen uttryckligen ogiltiga övergångar, som representeras av tomma celler.

Ett testfall baserat på ett tillståndsdiagram eller tillståndstabell representeras vanligtvis i form av en sekvens av händelser, vilket resulterar i en sekvens av tillståndsändringar (och åtgärder, om det behövs). Ett testfall kan, och kommer vanligtvis att, täcka flera övergångar mellan tillstånd.

Det finns många täckningskriterier för tillståndsbaserad testning. Denna kursplan tar upp tre av dem.

I **täckning av alla tillstånd** är täckningsobjekten tillstånden. För att uppnå 100 % täckning för alla tillstånd måste testfallen säkerställa att alla tillstånd besöks. Täckningen mäts som antalet besökta tillstånd dividerat med det totala antalet tillstånd, och uttrycks i procent.

I **täckningar av giltiga övergångar** (även kallad 0-switch-täckning) är täckningsobjekten enskilda giltiga övergångar. För att uppnå 100 % giltig övergångstäckning måste testfall testa alla giltiga övergångar. Täckningen mäts som antalet exekverade giltiga övergångar dividerat med det totala antalet giltiga övergångar, och uttrycks i procent.

I **täckning av alla övergångar** är täckningsobjekten alla övergångar som visas i en tillståndstabell. För att uppnå 100 % täckning av alla övergångar måste testfallen exekvera alla giltiga övergångar och försöka exekvera ogiltiga övergångar. Att testa endast en ogiltig övergång i ett enda testfall hjälper till att undvika felmaskering, dvs en situation där en defekt förhindrar upptäckten av en annan.

Täckning mäts som antalet giltiga och ogiltiga övergångar som exekverats eller försökt täckas av genomförda testfall, dividerat med det totala antalet giltiga och ogiltiga övergångar, och uttrycks i procent.

Täckningen av alla tillstånd är svagare än täckning av giltiga övergångångar, eftersom den vanligtvis kan uppnås utan att exekvera alla övergångar. Giltig övergångstäckning är det mest använda täckningskriteriet. Att uppnå full giltig övergångstäckning garanterar full täckning av alla tillstånd. Att uppnå full täckning av alla övergångar garanterar både full täckning av alla tillstånd och full giltig täckning av övergångar och bör vara ett minimikrav för verksamhets- och säkerhetskritisk programvara.

4.3. White-Box-testtekniker

På grund av deras popularitet och enkelhet fokuserar det här delkapitlet på två kodrelaterade white-box-testtekniker:

- Kodsatstestning (statement testing)
- Kodgrenstestning (branch testing)

Det finns mer noggranna tekniker som används i vissa säkerhetskritiska, verksamhetskritiska eller högintegritetsmiljöer för att uppnå en mer grundlig kodtäckning. Det finns också white-box-testtekniker som används på högre testnivåer (t.ex. API-testning) eller använder täckning som inte är relaterad till kod (t.ex. neurontäckning vid testning av neurala nätverk). Dessa tekniker tas inte upp i denna kursplan.

4.3.1. Kodsatstestning och kodsatstäckning

Vid kodsatstestning är täckningsobjekten exekverbara kodsatser. Syftet är att designa testfall som exekverar kodsatser tills en acceptabel täckningsgrad uppnås. Täckning mäts som antalet kodsatser som exekverats av testfallen dividerat med det totala antalet exekverbara kodsatserna och uttrycks i procent.

När 100 % kodsatstäckning uppnås säkerställer det att alla exekverbara kodsatser i koden har testats minst en gång. Detta innebär alltså att varje kodsats med en defekt kommer att exekveras, vilken kan resultera i ett felsymptom som visar närvaron av defekten. Men att exekvera en kodsats med ett testfall kommer inte att upptäcka defekter i alla situationer. Defekter som är databeroende (t.ex. en division med noll som bara misslyckas när en nämnare är noll) kanske inte upptäcks. Dessutom säkerställer inte 100 % kodsatstäckning att all beslutslogik har testats eftersom den kanske inte har exekverat alla grenar (se kapitel 4.3.2) i koden.

4.3.2. Kodgrenstestning och kodgrenstäckning

En kodgren är en överföring av kontrollen mellan två noder i ett kontrollflödesdiagram, som visar de möjliga sekvenserna i vilka källkodssatserna exekveras i testobjektet. Varje kontrollöverföring kan vara antingen ovillkorad (dvs. linjär kod) eller villkorad (d.v.s. ett beslutsutfall).

Vid kodgrentestning är täckningsobjekten kodgrenar och syftet är att designa testfall för att exekvera kodgrenar tills en acceptabel täckningsnivå uppnås. Täckningen mäts som antalet kodgrenar som exekverats av testfallen dividerat med det totala antalet kodgrenarna, och uttrycks i procent.

När 100 % kodgrenstäckning har uppnåtts så har alla kodgrenar, ovillkorade och villkorade, exekverats av testfall. Villkorade kodgrenar motsvarar vanligtvis ett sant eller falskt utfall från ett "if...then"-beslut, ett resultat från en switch/case-sats eller ett beslut att avsluta eller fortsätta i en loop. Men att exekvera en kodgren med ett testfall kommer inte att upptäcka defekter i samtliga fall. Det kanske inte upptäcker defekter som kräver exekvering av en specifik väg i en kod.

Kodgrentäckning inbegriper kodsatstäckning. Detta innebär att varje uppsättning testfall som uppnår 100 % kodgrenstäckning också uppnår 100 % kodsatstäckning (men inte vice versa).

4.3.3. Värdet med white-box-testning

En grundläggande styrka som alla white-box-tekniker delar är att hela programvaruimplementeringen beaktas under testning, vilket underlättar felupptäckt även när programvaruspecifikationen är vag, föråldrad eller ofullständig. En motsvarande svaghet är att om programvaran inte implementerar ett eller flera krav så kan white-box-testningen inte upptäcka de resulterande felen på grund av utelämnandet (Watson 1996).

White-box-tekniker kan användas vid statisk testning (t.ex. under torrexekvering av kod). De lämpar sig väl för att granska kod som ännu inte är klar för exekvering (Hetzel 1988), samt pseudokod och annan högnivå- eller top-down logik som kan modelleras med ett kontrollflödesdiagram.

Att endast utföra black-box-testning kan inte ge ett mått på faktisk kodtäckning. White-box-täckningsmätning ger ett objektivt mått på täckningen och ger nödvändig information för att skapa ytterligare tester för att öka täckningen, och därefter öka förtroendet för koden.

4.4. Erfarenhetsbaserade testtekniker

Vanligt använda erfarenhetsbaserade testtekniker som tas upp i följande avsnitt är:

- Felgissning
- Utforskande testning
- Checklistebaserad testning

4.4.1. Felgissning

Felgissning är en teknik som används för att förutse förekomsten av misstag, defekter och felsymptom, baserat på testarens kunskap, inklusive:

- Hur applikationen har fungerat tidigare
- Vilka typer av misstag utvecklarna tenderar att göra och vilka typer av defekter som är resultatet av dessa misstag
- Vilka typer av felsymptom som har inträffat i andra liknande applikationer

I allmänhet kan misstag, defekter och felsymptom vara relaterade till indata: (t.ex. korrekt indata accepteras inte, felaktiga eller saknade parametrar), utdata (t.ex. fel format, fel resultat), logik (t.ex. saknade fall, fel operator), beräkning (t.ex. felaktig operand, felaktig beräkning), gränssnitt (t.ex. icke matchade parametrar, inkompatibla typer) eller data (t.ex. felaktig initiering, fel typ).

Felattacker är ett metodiskt angreppssätt för implementering av felgissning. Denna teknik kräver att testaren skapar eller skaffar en lista över möjliga misstag, defekter och felsymptom och designar tester för att identifiera defekter som är förknippade med misstagen, och sedan avslöja defekterna eller orsakerna till felsymptomen. Skapandet av dessa listor kan baseras på erfarenhet, defekt- och felsymptomdata eller från allmän kunskap om varför programvaran misslyckas.

Se (Whittaker 2002, Whittaker 2003, Andrews 2006) för mer information om felgissning och felattacker.

4.4.2. Utforskande testning

I utforskande testning designas, exekveras och utvärderas testerna samtidigt som testaren lär sig om testobjektet. Testningen används också för att lära sig mer om testobjektet, att utforska testobjektet djupare med fokuserade tester och för att skapa tester för otestade områden.

Utforskande testning utförs ibland som sessionsbaserad testning för att strukturera testningen. I ett sessionsbaserat arbetssätt genomförs utforskande testning inom en definierad tidsram (time-box). Testaren använder en testcharter som innehåller testmålen för att styra testningen. Efter testsessionen genomförs vanligtvis en debriefing med en diskussion mellan testaren och intressenter som är intresserade av testresultaten från testsessionen. Med detta arbetssätt kan testmålen hanteras som högnivå-testvillkor. Täckningsobjekten identifieras och exekveras under testsessionen. Testaren kan använda testsessionsblad för att dokumentera de steg som följs och de upptäckter som gjorts.

Utforskande testning är användbar när det finns få eller otillräckliga specifikationer eller vid en betydande tidspress på testningen. Utforskande testning är också användbart för att komplettera andra mer formella testtekniker. Den kommer att bli effektivare om testaren är erfaren, har domänkunskaper och har en hög grad av väsentliga färdigheter, som analytiskt tänkande, nyfikenhet och kreativitet (se kapitel 1.5.1).

Utforskande testning kan innefatta användning av andra testtekniker (t.ex. ekvivalensklassindelning). Mer information om utforskande testning finns i (Kaner 1999, Whittaker 2009, Hendrickson 2013).

4.4.3. Checklistebaserad testning

I checklistebaserad testning designar, implementerar och exekverar en testare tester för att täcka testvillkoren utgående från en checklista. Checklistor kan byggas utifrån erfarenhet, kunskap om vad som är viktigt för användaren, eller en förståelse för varför och hur programvara fallerar. Checklistor ska inte innehålla objekt som kan kontrolleras automatiskt, objekt som är bättre lämpade som start-/avslutskriterier eller objekt som är för generella (Brykczynski 1999).

Poster i en checklista är ofta formulerade som en fråga. Det ska vara möjligt att kontrollera varje post separat och direkt. Posterna kan hänvisa till krav, egenskaper för grafiskt gränssnitt, kvalitetsegenskaper eller andra former av testvillkor. Checklistor kan skapas för att stödja olika testtyper, inklusive funktionell och icke-funktionell testning (t.ex. 10 heuristics for usability testing (Nielsen 1994)).

Vissa checklisteposter kan gradvis bli mindre effektiva med tiden eftersom utvecklarna lär sig att undvika att göra samma misstag. Nya poster kan också behöva läggas till för att återspegla nyfunna defekter med hög allvarlighetsgrad. Därför bör checklistor uppdateras regelbundet baserat på defektanalyser. Man bör dock undvika att checklistan blir för lång (Gwande 2009).

I avsaknad av detaljerade testfall kan checklistebaserad testning ge riktlinjer och en viss grad av enhetlighet i testningen. Om checklistorna är på hög nivå kommer sannolikt viss variation i själva testningen att uppstå, vilket kan resultera i potentiellt större täckning men mindre repeterbarhet.

4.5. Samarbetsbaserade testangreppssätt

Var och en av de ovan nämnda teknikerna (se kapitel 4.2, 4.3, 4.4) har ett särskilt syfte med avseende på defektdetektering. Samarbetsbaserade angreppssätt, å andra sidan, fokuserar också på att undvika defekter genom samarbete och kommunikation.

4.5.1. Skriva användarberättelser i samarbete

En användarberättelse representerar en funktion som kommer att vara värdefull för antingen en användare eller köpare av ett system eller programvara. Användarberättelser har tre kritiska aspekter (Jeffries 2000), som tillsammans kallas för "3 C:na":

- Kort (Card) – mediet som beskriver en användarberättelse (t.ex. ett registerkort, en post på en elektronisk tavla)
- Konversation (Conversation) – förklarar hur programvaran kommer att användas (kan dokumenteras eller verbalt)
- Bekräftelse (Confirmation) – acceptanskriterierna (se kapitel 4.5.2)

Det vanligaste formatet för en användarberättelse är "Som en [roll] vill jag att [målet ska uppnås], så att jag kan [resulterande affärsvärde för rollen]", följt av acceptanskriterierna.

Vid samarbetande författarskap av användarberättelser kan tekniker som brainstorming och mindmapping användas. Samarbetet gör att teamet får en gemensam vision om vad som ska levereras, genom att ta hänsyn till tre perspektiv: verksamhet, utveckling och testning.

Bra användarberättelser bör vara Oberoende, Förhandlingsbara, Värdefulla, Uppskattningsbara, Små och Testbara (INVEST: Independent, Negotiable, Valuable, Estimable, Small and Testable). Om en intressent inte vet hur man testar en användarberättelse kan det tyda på att användarberättelsen inte är tillräckligt tydlig, eller att den inte speglar något värdefullt för denne, eller att intressenten bara behöver hjälp med att testa (Wake 2003).

4.5.2. Acceptanskriterier

Acceptanskriterier för en användarberättelse är de villkor som en implementering av användarberättelsen måste uppfylla för att accepteras av intressenter. Ur detta perspektiv kan acceptanskriterier ses som de testvillkor som bör exekveras av testerna. Acceptanskriterier är vanligtvis ett resultat av Konversationen (se kapitel 4.5.1).

Acceptanskriterier används för att:

- Definiera omfattningen av användarberättelsen
- Uppnå konsensus bland intressenterna
- Beskriva både positiva och negativa scenarier
- Fungera som bas för acceptanstestning av användarberättelser (se kapitel 4.5.3)
- Möjliggöra rätt planering och uppskattning

Det finns flera sätt att skriva acceptanskriterier för en användarberättelse. De två vanligaste formaten är:

- Scenarioorienterat (t.ex. formatet Given/When/Then som används i BDD, se kapitel 2.1.3)
- Regelorienterat (t.ex. verifieringslista med punkter, eller tabellform med input-output-mappning)

De flesta acceptanskriterier kan dokumenteras i något av dessa två format. Teamet kan dock använda ett annat anpassat format, så länge acceptanskriterierna är väldefinierade och entydiga.

4.5.3. Acceptanstestdriven utveckling (ATDD)

ATDD är en test-first-metod (se kapitel 2.1.3). Testfall skapas innan användarberättelsen implementeras. Testfallen skapas av teammedlemmar med olika perspektiv, t.ex. kunder, utvecklare och testare (Adzic 2009). Testfallen kan exekveras manuellt eller automatiserat.

Det första steget är en specifikationsworkshop där användarberättelsen och (om inte ännu definierad) dess acceptanskriterier analyseras, diskuteras och skrivs av teammedlemmarna. Ofullständigheter, oklarheter eller defekter i användarberättelsen löses under denna process. Nästa steg är att skapa testfallen. Detta kan göras av teamet som helhet eller av testaren individuellt. Testfallen baseras på acceptanskriterierna och kan ses som exempel på hur programvaran fungerar. Detta kommer att hjälpa teamet att implementera användarberättelsen korrekt.

Eftersom exempel och tester är detsamma används dessa termer ofta omväxlande. Under testdesign kan de testtekniker som beskrivs i kapitel 4.2, 4.3 och 4.4 tillämpas.

Vanligtvis är de första testfallen positiva, och bekräftar det korrekta beteendet utan undantag (exception) eller feltillstånd, och omfattar sekvensen av aktiviteter som utförs om allt går som förväntat. Efter att de positiva testfallen är genomförda ska teamet utföra negativa tester. Slutligen bör teamet också täcka icke-funktionella kvalitetsegenskaper (t.ex. prestandaeffektivitet och användbarhet). Testfall ska skrivas på ett sätt som är förståeligt för intressenterna. Vanligtvis innehåller testfallen meningar på naturligt språk som inkluderar de nödvändiga förutsättningarna (om några), indata och sluttillstånd.

Testfallen måste täcka alla egenskaper hos användarberättelsen och ska inte gå utanför berättelsen. Acceptanskriterierna kan dock beskriva några av de problem som beskrivs i användarberättelsen. Dessutom ska inte fler testfall beskriva samma egenskaper hos användarberättelsen.

När utvecklarna vägleds av ett format som stöds av ett ramverk för testautomatisering kan de automatisera testfallen genom att använda vägledningen vid implementationen av funktionen som beskrivs av en användarberättelse. Acceptanstesterna blir då exekverbara krav.

5. Hantering av testaktiviteterna – 335 minuter

Nyckelord

avslutskriterier, felhantering, felrapport, produktrisk, projektrisk, risk, riskanalys, riskbaserad testning, riskbedömning, riskhantering, riskidentifiering, riskkontroll, risknivå, riskövervakning, riskreducering, sammanfattande testrapport, startkriterier, testangreppssätt, testkvadranter, testövervakning, testplan, testplanering, testpyramid, teststatusrapport, teststyrning

Utbildningsmål för kapitel 5:

5.1 Testplanering

FL-5.1.1 (K2) Exemplifiera syftet med och innehållet i en testplan

FL-5.1.2 (K1) Känna igen hur en testare kan ge mervärde till iterations- och releaseplanering

FL-5.1.3 (K2) Jämföra likheter och skillnader mellan start- och avslutskriterier

FL-5.1.4 (K3) Använda uppskattningstekniker för att beräkna önskad testarbetsinsats

FL-5.1.5 (K3) Tillämpa prioritering av testfall

FL-5.1.6 (K1) Komma ihåg testpyramidens egenskaper

FL-5.1.7 (K2) Sammanfatta testkvadranterna och deras samband med testnivåer och testtyper

5.2 Riskhantering

FL-5.2.1 (K1) Identifiera risknivåer genom att använda sannolikhet och påverkan för risker

FL-5.2.2 (K2) Skilja mellan projektrisker och produktrisker

FL-5.2.3 (K2) Beskriva hur produktriskanalyser kan påverka testningens fullständighet och omfattning

FL-5.2.4 (K2) Förklara vilka åtgärder som kan vidtas som svar på analyserade produktrisker

5.3 Testövervakning, teststyrning och testavslut

FL-5.3.1 (K1) Komma ihåg mätetal som används i testningen

FL-5.3.2 (K2) Summera syften med, innehåll och mottagare för testrapporter

FL-5.3.3 (K2) Exemplifiera hur man kommunicerar teststatus

5.4 Konfigurationshantering

FL-5.4.1 (K2) Sammanfatta hur konfigurationshanteringen stödjer testningen

5.5 Felhantering

FL-5.5.1 (K3) Skriva en felrapport

5.1. Testplanering

5.1.1. Syftet med och innehållet i en testplan

En testplan beskriver mål, resurser och processer för ett testprojekt genom att:

- Dokumentera medel och tidsplan för att uppnå testmålen
- Hjälpa till att säkerställa att de utförda testaktiviteterna kommer att uppfylla de fastställda kriterierna
- Fungera som ett kommunikationsmedel för teammedlemmar och andra intressenter
- Visa att testningen kommer att följa den befintliga testpolicyn och teststrategin (eller förklarar varför testningen kommer att avvika från dem)

Testplanering vägleder testarna och tvingar dem att möta framtida utmaningar relaterade till risker, tidsplaner, människor, verktyg, kostnader, insats etc. Processen att skriva en testplan är ett sätt att tänka igenom de arbetsinsatser som behövs för att uppnå testprojektets mål.

Vanligtvis innehåller en testplan följande:

- Testningens sammanhang (t.ex. omfattning, testmål, begränsningar, testbas)
- Testprojektets antaganden och begränsningar
- Intressenter (t.ex. roller, ansvar, relevans till testningen, rekrytering och utbildningsbehov)
- Kommunikation (t.ex. former och frekvens för kommunikation, dokumentmallar)
- Riskregister (t.ex. produktrisker och projektrisker)
- Testangreppssätt (t.ex. testnivåer, testtyper, testtekniker, testleverabler, startkriterier och avslutskriterier, oberoende testning, mätvärden som ska samlas in, testdatakrav, testmiljökrav, avvikelser från organisationens testpolicy och teststrategi)
- Budget och tidsplan

Mer information om testplanen och dess innehåll finns i standarden ISO/IEC/IEEE 29119-3.

5.1.2. Testarens medverkan i iterations- och releaseplanering

I iterativa SDLC:er förekommer vanligtvis två typer av planering: releaseplanering och iterationsplanering.

Releaseplanering planerar för release av en produkt, definierar och omdefinierar produktbackloggen och kan innebära förfining av större användarberättelser till en uppsättning mindre. Den fungerar också som grund för testangreppssätt och testplaner för alla iterationer. Testare som är involverade i releaseplanering deltar genom att skriva testbara användarberättelser och acceptanskriterier (se kapitel 4.5). De deltar också i projekt- och kvalitetsriskanalyser (se kapitel 5.2), uppskattar testarbetsinsatsen för användarberättelser (se kapitel 5.1.4), bestämmer testangreppssättet och planerar testningen inför releasen.

Iterationsplanering planerar för slutet av en enstaka iteration och fokuserar på iterationsbackloggen. Testare som är involverade i iterationsplanering deltar i den detaljerade riskanalysen av användarberättelser, bestämmer testbarheten av användarberättelser, bryter ner användarberättelser till uppgifter (speciellt testuppgifter), uppskattar testinsatsen för allt testarbete och identifierar och förfinar funktionella och icke-funktionella aspekter av testobjektet.

5.1.3. Start- och avslutskriterier

Startkriterier definierar förutsättningarna för att bedriva en viss aktivitet. Om startkriterierna inte är uppfyllda är det sannolikt att aktiviteten kommer att bli svårare, mer tidskrävande, kostsammare och mer riskfylld. Avslutskriterierna definierar vad som måste uppnås för att definiera en aktivitet som avslutad. Startkriterier och avslutskriterier ska definieras för varje testnivå och kommer att vara olika beroende på testmålsättningarna.

Typiska startkriterier är tillgång till resurser (t.ex. personal, verktyg, miljöer, testdata, budget, tid), tillgång till testvara (t.ex. testbas, testbara krav, användarberättelser, testfall) och initial kvalitetsnivå för ett testobjekt (t.ex. alla smoketester har godkänts).

Typiska avslutskriterier är mått på fullständighet (t.ex. uppnådd täckningsnivå, antal olösta defekter, defektdensitet, antal underkända testfall) och slutförandekriterier (t.ex. planerade tester har utförts, statiska tester har utförts, alla hittade defekter rapporteras, alla regressionstester är automatiserade).

Att tid eller budget tar slut kan också ses som giltiga avslutskriterier. Även om andra avslutskriterier inte är uppfyllda kan det vara acceptabelt att avsluta testning under sådana omständigheter om intressenterna har granskat och accepterat risken att gå i drift utan ytterligare testning.

I agil programvaruutveckling kallas avslutskriterier ofta för Definition of Done, som definierar teamets objektiva mätvärden för ett releasebart objekt. Startkriterier som en användarberättelse måste uppfylla för att påbörja utvecklings- och/eller testaktiviteter kallas Definition of Ready.

5.1.4. Uppskattningstekniker

Uppskattning av ett testarbete innebär att förutsäga mängden testrelaterat arbete som behövs för att uppfylla målen för ett testprojekt. Det är viktigt att göra det klart för intressenterna att testuppskattningen bygger på ett antal antaganden som alltid kan vara föremål för felberäkning. Beräkning av små uppgifter är vanligtvis mer exakt än för stora. När man därför uppskattar en stor uppgift så kan det vara bra att dela upp den i ett antal mindre uppgifter som sedan i sin tur kan beräknas.

I denna kursplan beskrivs följande fyra uppskattningstekniker.

Uppskattning baserad på förhållanden. I denna mätetalsbaserad teknik samlas fakta in från tidigare projekt inom organisationen, vilket gör det möjligt att härleda ”standard”-förhållanden från liknande projekt. Förhållanden mellan en organisations egna projekt (t.ex. hämtade från historiska data) är i allmänhet den bästa källan att använda i uppskattningsprocessen. Dessa standardförhållanden kan sedan användas för att uppskatta testinsatsen för det nya projektet. Till exempel, om förhållandet utveckling-till-testinsats i det föregående projektet var 3:2, och i det aktuella projektet förväntas utvecklingsinsatsen vara 600 persondagar, kan testinsatsen uppskattas till 400 persondagar.

Extrapolering. I denna mätetalsbaserad teknik görs mätningar så tidigt som möjligt i det aktuella projektet för att samla in data. Med tillräckligt många observationer kan arbetsinsatsen som krävs för det återstående arbetet approximeras genom att extrapolera dessa data (vanligtvis genom att tillämpa en matematisk modell). Denna metod är mycket lämplig i iterativa SDLC:er. Till exempel kan teamet extrapolera testarbetsinsatsen i den kommande iterationen som den genomsnittliga arbetsinsatsen från de tre senaste iterationerna.

Wideband Delphi. I denna iterativa, expertbaserade teknik gör experter erfarenhetsbaserade uppskattningar. Varje expert uppskattar insatsen isolerat. Resultaten samlas in och om det finns avvikelser som ligger utanför de överenskomna gränserna diskuterar experterna sina aktuella uppskattningar. Varje expert uppmanas sedan att göra en ny uppskattning baserat på den återkopplingen, återigen isolerat. Denna process upprepas tills konsensus uppnås. Planning Poker är en variant av Wideband Delphi, som vanligen används i agil programvaruutveckling. I Planning Poker görs uppskattningar vanligtvis med kort med siffror som representerar arbetets storlek.

Trepunktsuppskattning. I denna expertbaserade teknik görs tre uppskattningar av experterna: den mest optimistiska uppskattningen (a), den mest sannolika uppskattningen (m) och den mest pessimistiska uppskattningen (b). Den slutliga skattningen (E) är deras vägda aritmetiska medelvärde. I den mest populära versionen av denna teknik beräknas uppskattningen som E = (a + 4*m + b) / 6. Fördelen med denna teknik är att den tillåter experterna att beräkna mätfelet: SD = (b – a) / 6. Om uppskattningarna (i persontimmar) t.ex. är: a=6, m=9 och b=18, är den slutliga uppskattningen 10±2 persontimmar (dvs. mellan 8 och 12 personer) -timmar), eftersom E = (6 + 4*9 + 18) / 6 = 10 och SD = (18 – 6) / 6 = 2.

Se (Kan 2003, Koomen 2006, Westfall 2009) för dessa och många andra testuppskattningstekniker.

5.1.5. Prioritering av testfall

När testfallen och testprocedurerna har specificerats och sammanställts i testsviter kan dessa testsviter ordnas i ett testexekveringsschema som definierar i vilken ordning de ska köras. Vid prioritering av testfall kan olika faktorer beaktas. De vanligast använda prioriteringsstrategierna för testfall är följande:

- Riskbaserad prioritering, där testexekveringsordningen baseras på resultaten i en riskanalys (se kapitel 5.2.3). Testfall som täcker de viktigaste riskerna exekveras först.
- Täckningsbaserad prioritering, där testexekveringsordningen baseras på täckning (t.ex. kodsatstäckning). Testfall som uppnår den högsta täckningen exekveras först. I en annan variant, kallad ytterligare täckningsprioritering, exekveras testfallet som uppnår högst täckning först; varje efterföljande testfall är det som sedan uppnår den då högsta täckningen.
- Kravbaserad prioritering, där testexekveringsordningen baseras på kravprioriteringarna som spåras tillbaka till motsvarande testfall. Kravprioriteringar definieras av intressenterna. Testfall relaterade till de viktigaste kraven exekveras först.

Helst skulle exekvering av testfallen baseras på deras prioritetsnivåer genom att t.ex. använda en av de ovannämnda prioriteringsstrategierna. Denna praxis kanske dock inte fungerar om testfallen eller funktionerna som testas har beroenden. Om ett testfall med högre prioritet är beroende av ett testfall med lägre prioritet, måste testfallet med lägre prioritet exekveras först.

Ordningen för testkörning måste också ta hänsyn till tillgången på resurser. Exempel kan vara testverktyg som krävs, testmiljöer eller personer som kanske bara är tillgängliga under en viss tidsperiod.

5.1.6. Testpyramid

Testpyramiden är en modell som visar att olika tester kan ha olika detaljeringsgrad. Testpyramidmodellen stödjer teamet i testautomatisering och i fördelning av testarbetsmängden genom att visa att olika mål stöds av olika nivåer av testautomatisering. Lagren i pyramiden representerar grupper av tester. Ju högre upp lagret befinner sig, desto mindre testdetaljeringsgrad, testisolering och testexekveringstid. Tester i det undre lagret är små, isolerade, snabba och kontrollerar en liten del av funktionaliteten, så vanligtvis behövs det många av dem för att uppnå en rimlig täckning. Det översta lagret representerar komplexa end-to-end-tester på hög nivå. Dessa högnivåtester är i allmänhet långsammare än testerna från de lägre lagren, och de kontrollerar vanligtvis en stor del av funktionaliteten, så oftast behövs bara ett fåtal av dem för att uppnå en rimlig täckning. Antal och namn på lagren kan skilja sig åt. Till exempel definierar den ursprungliga testpyramidmodellen (Cohn 2009) tre lager: "enhetstester", "servicetester" och "UI-tester". En annan populär modell definierar enhets(komponent)-tester, integrations (komponent integration)-tester och end-to-end-tester. Andra testnivåer (se kapitel 2.2.1) kan också användas.

5.1.7. Testkvadranter

Testkvadranterna, definierade av Brian Marick (Marick 2003, Crispin 2008), grupperar testnivåerna med lämpliga testtyper, aktiviteter, testtekniker och arbetsprodukter i agil programvaruutveckling. Modellen stödjer testhanteringen genom att visualisera dessa och säkerställa att alla lämpliga testtyper och testnivåer ingår i SDLC och för att förstå att vissa testtyper är mer relevanta för vissa testnivåer än andra. Den här modellen ger också ett sätt att särskilja och beskriva testtyperna för alla intressenter, inklusive utvecklare, testare och verksamhetsrepresentanter.

I modellen kan tester vara affärsinriktade eller teknikinriktade. Tester kan också stödja teamet (dvs vägleda utvecklingen) eller kritisera produkten (dvs mäta dess beteende mot förväntningarna). Kombinationen av dessa två synpunkter bestämmer de fyra kvadranterna:

- Kvadrant Q1 (teknikinriktade, stöder teamet). Denna kvadrant innehåller komponent- och komponentintegrationstester. Dessa tester bör automatiseras och inkluderas i CI-processen.
- Kvadrant Q2 (affärsinriktade, stöder teamet). Denna kvadrant innehåller funktionella tester, till exempel tester av användarberättelser, prototyper för användarupplevelser, API-tester och simuleringar. Dessa tester kontrollerar acceptanskriterierna och kan vara manuella eller automatiserade.
- Kvadrant Q3 (affärsinriktade, kritiserar produkten). Denna kvadrant innehåller utforskande testning, användbarhetstestning, användaracceptanstestning. Dessa tester är användarorienterade och ofta manuella.
- Kvadrant Q4 (teknikinriktade, kritiserar produkten). Denna kvadrant innehåller smoketester och icke-funktionella tester (förutom användbarhetstester). Dessa tester är ofta automatiserade.

5.2. Riskhantering

Organisationer står inför många interna och externa faktorer som gör det osäkert om och när de kommer att uppnå sina mål (ISO 31000). Riskhantering tillåter organisationerna att öka sannolikheten för att uppnå målen, förbättra kvaliteten på sina produkter och öka intressenternas förtroende och tillit.

De huvudsakliga riskhanteringsaktiviteterna är:

- Riskanalys (bestående av riskidentifiering och riskbedömning; se avsnitt 5.2.3)
- Riskkontroll (bestående av riskreducering och riskövervakning; se avsnitt 5.2.4)

Testangreppssättet där testaktiviteter väljs, prioriteras och hanteras utifrån riskanalys och riskkontroll, kallas riskbaserad testning.

5.2.1. Riskdefinition och riskattribut

Risk är en potentiell händelse, fara, hot eller situation vars inträffande orsakar en negativ effekt. En risk kan karakteriseras av två faktorer:

• Risksannolikhet – sannolikheten för att risken inträffar (större än noll och mindre än ett)

• Riskkonsekvens (skada) – konsekvenserna av denna händelse

Dessa två faktorer uttrycker risknivån, som är ett mått på risken. Ju högre risknivå, desto viktigare är hanteringen.

5.2.2. Projekt- och produktrisker

Vid programvarutestning är man generellt intresserad över två typer av risker: projektrisker och produktrisker.

Projektrisker är relaterade till projekthantering och kontroll. Projektrisker avser:

- Organisatoriska frågor (t.ex. förseningar i leveranser av arbetsprodukter, felaktiga uppskattningar och kostnadsbesparingar)
- Personalfrågor (t.ex. otillräckliga kunskaper, konflikter, kommunikationsproblem, brist på personal)
- Tekniska frågor (t.ex. okontrollerade ändringar, dåligt verktygsstöd)
- Leverantörsfrågor (t.ex. fel i leverans från tredje part, konkurs hos det stödjande företaget)

Projektrisker kan, när de uppstår, ha en inverkan på projektets tidsplan, budget eller omfattning, vilket påverkar projektets förmåga att uppnå sina mål.

Produktriskerna är relaterade till produktens kvalitetsegenskaper (t.ex. beskrivna i kvalitetsmodellen ISO 25010). Exempel på produktrisker är saknad eller felaktig funktionalitet, felaktiga beräkningar, run-time-fel, dålig arkitektur, ineffektiva algoritmer, otillräckliga svarstider, dålig användarupplevelse, säkerhetssårbarheter. Produktrisker kan, när de uppstår, resultera i olika negativa konsekvenser, som:

- Missnöjda användare
- Förlust av intäkter, förtroende, rykte
- Skador på tredje part
- Höga underhållskostnader, överbelastning av helpdesk
- Straffrättsliga påföljder
- I extrema fall, fysisk skada, personskador eller till och med dödsfall

5.2.3. Produktriskanalys

Ur ett testperspektiv är målet med en produktriskanalys att ge en medvetenhet om produktrisker för att fokusera testinsatserna på ett sätt som minimerar den kvarvarande nivån av produktrisk. Helst börjar man med en produktriskanalys tidigt i SDLC.

Produktriskanalysen består av riskidentifiering och riskbedömning. Riskidentifiering handlar om att skapa en omfattande lista över risker. Intressenter kan identifiera risker genom att använda olika tekniker och verktyg, t.ex. brainstorming, workshops, intervjuer eller orsak-verkan-diagram. Riskbedömning innefattar kategorisering av identifierade risker, fastställande av deras risksannolikhet, riskpåverkan och risknivå, prioritering samt förslag till sätt att hantera dem. Att kategorisera dem underlättar tilldelning av reducerande åtgärder, eftersom risker som faller i samma kategori vanligtvis kan begränsas med ett liknande angreppssätt.

Riskbedömning kan använda en kvantitativ eller kvalitativ metod, eller en blandning av dem. I den kvantitativa metoden beräknas risknivån som multiplikationen av risksannolikhet och riskpåverkan. I den kvalitativa metoden kan risknivån bestämmas med hjälp av en riskmatris.

Produktriskanalysen kan påverka testningens grundlighet och omfattning. Dess resultat används för att:

- Bestämma omfattningen av testningen som ska göras

- Bestämma de specifika testnivåer och föreslå testtyper som ska användas
- Bestämma vilka testtekniker som ska användas och vilken täckning som ska uppnås
- Uppskatta den testinsats som krävs för varje uppgift
- Prioritera testningen i ett försök att hitta de kritiska defekterna så tidigt som möjligt
- Bestämma om några andra aktiviteter utöver testning ska göras för att minska risken

5.2.4. Produktriskkontroll

En produktriskkontroll omfattar alla åtgärder som vidtas som svar på identifierade och bedömda produktrisker. Produktriskkontrollen består av riskreducering och riskövervakning. Riskreducering innebär att de åtgärder som föreslås i riskbedömningen genomförs för att minska risknivån. Syftet med riskövervakning är att säkerställa att åtgärderna är effektiva för att reducera risken, att få ytterligare information för att förbättra riskbedömningen och att identifiera nya risker.

Under produktriskkontrollen kan, när en risk har analyserats, flera alternativ vara möjliga, t. ex. riskreducering genom testning, riskacceptans, risköverföring eller åtgärdsplan (Veenendaal 2012). Åtgärder som kan vidtas för att minska produktriskerna genom testning är följande:

- Välja testare med rätt erfarenhet och kompetens, lämpade för en given risktyp
- Använda en lämplig nivå av oberoende testning
- Genomföra granskningar och statisk analys
- Tillämpa lämpliga testtekniker och täckningsnivåer
- Tillämpa lämpliga testtyper för att ta itu med de påverkade kvalitetsegenskaperna
- Genomföra dynamisk testning, inklusive regressionstestning

5.3. Testövervakning, teststyrning och testavslut

Testövervakning handlar om att samla in information om testningen. Informationen används för att bedöma testningens framsteg och för att mäta om testavslutskriterierna eller testarbetet som är förknippade med avslutskriterierna är uppfyllda. Exempel på uppfyllda kriterier kan vara målen för täckning av produktrisker, krav eller acceptanskriterier.

Teststyrning använder informationen från testövervakningen för att, i form av styrdirektiv, ge vägledning och nödvändiga korrigerande åtgärder för att uppnå den mest effektiva och ändamålsenliga testningen. Exempel på styrdirektiv inkluderar:

- Omprioritering av testerna när en identifierad risk behöver hanteras
- Omvärdera om ett testobjekt uppfyller startkriterier eller avslutskriterier på grund av omarbetning
- Justera testschemat för att åtgärda en försening i leveransen av testmiljön
- Utöka med nya resurser när och där det behövs

Testavslut samlar in data från genomförda testaktiviteter för att ta vara på erfarenheter, testvara och annan relevant information. Testavslutsaktiviteter sker vid projektmilstolpar t.ex. när en testnivå är klar, en agil iteration är klar, ett testprojekt har slutförts (eller avbrutits), ett programvarusystem releasas eller en underhållsrelease slutförs.

5.3.1. Mätetal som används i testning

Mätresultat från testningen samlas in för att visa framsteg mot den planerade tidsplanen och budgeten, den aktuella kvaliteten på testobjektet och effektiviteten hos testaktiviteterna med avseende på målsättningen eller ett iterationsmål. Testövervakning samlar in en mängd olika mätvärden för att stödja teststyrningen och testavslutet.

Vanliga testmätetal kan vara:

- Mätetal för projektframsteg (t.ex. slutförande av uppgift, resursanvändning, testinsats)
- Mätetal för testframsteg (t.ex. framsteg för implementering av testfall, framsteg för förberedelse av testmiljö, antal körda/inte körda testfall, godkända/misslyckade testfall, testexekveringstid)
- Mätetal för produktkvalitet (t.ex. tillgänglighet, svarstider, medeltid till fel)
- Mätetal för defekter (t.ex. antal och prioriteringar för defekter som hittats/åtgärdats, defektdensitet, defect detection percentage, DDP)
- Mätetal för risker (t.ex. kvarstående risknivå)
- Mätetal för täckningar (t.ex. kravtäckning, kodsatstäckning)
- Mätetal för kostnader (t.ex. kostnad för testning, organisationskostnad för kvalitet)

5.3.2. Syften, innehåll och målgrupper för testrapporter

Testrapportering sammanfattar och kommunicerar testinformation under och efter testning. Teststatusrapporter stödjer den pågående styrningen av testningen och måste ge tillräckligt med information för att kunna göra ändringar (om de behövs) i testschemat, resurserna eller testplanen. Behovet kan vara att planen inte kunde följas eller andra ändrade förhållanden.. Sammanfattande testrapporter för ett specifikt testskede (t.ex. testnivå, testcykel, iteration) kan ge information för kommande testningar.

Under testövervakning och styrning producerar testteamet teststatusrapporter till intressenterna för att hålla dem informerade. Dessa rapporter produceras vanligtvis regelbundet (t.ex. dagligen, veckovis, etc.) och inkluderar:

- Testperiod
- Testframsteg (t.ex. före eller efter planerade tiden), inklusive eventuella noterbara avvikelser
- Hinder för testning och deras tillfälliga lösningar (workarounds)
- Testmätetal (se kapitel 5.3.1 för exempel)
- Nya och förändrade risker inom testperioden
- Planerad testning för nästa period

En sammanfattande testrapport förbereds under testningens slutförande, när ett projekt, testnivå eller testtyp är klar och när (idealiskt) dess avslutskriterier har uppfyllts. Rapporten använder teststatusrapporter och annat data. En typisk sammanfattande testrapport inkluderar:

- Testsammanfattning
- Utvärdering av testningen och produktkvaliteten baserat på den ursprungliga testplanen (d.v.s. testmål och avslutskriterier)
- Avvikelser från testplanen (t.ex. skillnader från det planerade tidsplanen, varaktighet och arbete).

- Testhinder och tillfälliga lösningar (workarounds)
- Testmätningar baserat på teststatusrapporter
- Risker som inte reducerats, defekter som inte åtgärdats
- Lärdomar som är relevanta för testningen

Olika målgrupper kräver olika information i rapporterna och detta kan påverka graden av formalitet och rapporteringsfrekvens. Rapportering om testframsteg till andra i teamet är ofta frekvent och informell, medan testrapportering för ett avslutat projekt följer en mall och sker endast en gång.

ISO/IEC/IEEE 29119-3-standarden innehåller mallar och exempel för teststatusrapporter och sammanfattande testrapporter (test status reports, test completion reports).

5.3.3. Teststatuskommunikation

Det bästa sättet att kommunicera teststatus varierar beroende på testhanteringsfrågor, organisationens teststrategier, regulatoriska standarder eller, i fallet med självorganiserande team (se kapitel 1.5.2), på själva teamet. Alternativen inkluderar:

- Verbal kommunikation med teammedlemmar och andra intressenter
- Dashboards (t.ex. CI/CD-dashboards, aktivitetstavlor och burn-down charts)
- Elektroniska kommunikationskanaler (t.ex. e-post, chatt)
- Onlinedokumentation
- Formella testrapporter (se kapitel 5.3.2)

Ett eller flera av dessa alternativ kan användas. Mer formell kommunikation kan vara mer lämplig för distribuerade team där direkt kommunikation öga mot öga inte alltid är möjlig på grund av geografiskt avstånd eller tidsskillnader. Vanligtvis är olika intressenter intresserade av olika typer av information, så kommunikationen bör anpassas efter det.

5.4. Konfigurationshantering

Konfigurationshantering (configuration management, CM) är en disciplin som hjälper testningen att identifiera, kontrollera och spåra arbetsprodukter som testplaner, teststrategier, testvillkor, testfall, testskript, testresultat, testloggar och testrapporter som konfigurationsobjekt.

För ett komplext konfigurationsobjekt (t.ex. en testmiljö), registrerar CM de objekt som den består av, deras relationer och versioner. Om konfigurationsobjektet godkänns för testning blir det en baslinje och kan endast ändras genom en formell ändringskontrollprocess.

Konfigurationshanteringen håller ett register över ändrade konfigurationsobjekt när en ny baslinje skapas. Det är möjligt att återgå till en tidigare baslinje för att återskapa tidigare testresultat.

För att korrekt stödja testning säkerställer CM följande:

- Alla konfigurationsobjekt, inklusive testelement (enskilda delar av testobjektet), är unikt identifierade, versionshanterade, spårbara för ändringar och relaterade till andra konfigurationsobjekt så att spårbarheten kan bibehållas genom hela testprocessen
- All identifierad dokumentation och programelement är entydigt refererade till i testdokumentationen

Kontinuerlig integration (CI), kontinuerlig leverans (CD), kontinuerlig driftsättning och tillhörande testning implementeras vanligtvis som en del av en automatiserad DevOps-pipeline (se kapitel 2.1.4), i vilken automatiserad CM normalt ingår.

5.5. Felhantering

Eftersom ett av de viktigaste testmålen är att hitta defekter, är en etablerad felhanteringsprocess väsentlig. Även om vi hänvisar till "defekter" här, kan de rapporterade avvikelserna visa sig vara verkliga defekter eller något annat (t.ex. falsk positiv, ändringsbegäran) - detta löses under felhanteringsprocessen. Avvikelser kan rapporteras under vilken fas som helst i SDLC och formen beror på SDLC. Processen måste följas av alla inblandade intressenter. Felhanteringsprocessen innehåller, som ett minimum, ett arbetsflöde för hantering av enskilda avvikelser (från upptäckt till stängning) och regler för deras klassificering. Arbetsflödet omfattar vanligtvis aktiviteter för att logga de rapporterade avvikelserna, analysera och klassificera dem, besluta om ett lämpligt svar som att åtgärda eller behålla som det är och slutligen att stänga felrapporten. Det är tillrådligt att hantera defekter från statisk testning (speciellt statisk analys) på liknande sätt.

Typiska felrapporter har följande syften:

- Att förse de ansvariga för hantering och lösning av rapporterade defekter med tillräcklig information för att lösa problemet
- Att tillhandahålla ett sätt att spåra kvaliteten på arbetsprodukten
- Att ge idéer till förbättring av utvecklings- och testprocessen

En felrapport som loggas under dynamisk testning inkluderar vanligtvis:

- Unik identifierare
- Titel med en kort sammanfattning av den avvikelse som rapporteras
- Datum när avvikelsen observerades, utfärdande organisation och författare, inklusive deras roll
- Identifiering av testobjekt och testmiljö
- Defektens sammanhang (t.ex. testfall som kördes, testaktivitet som utfördes, SDLC-fas och annan relevant information som testteknik, checklista eller testdata som används)
- Beskrivning av felsymptomet, för att kunna reproducera och lösa det, inklusive stegen som upptäckte avvikelsen och eventuella relevanta testloggar, databasdumpar, skärmdumpar eller inspelningar
- Förväntade resultat och faktiska resultat
- Defektens allvarlighetsgrad (grad av påverkan) enligt intressenterna eller kraven
- Åtgärdsprioritet
- Status för defekten (t.ex. öppen, uppskjuten, dubblett, väntar på att åtgärdas, väntar på omtestning, återöppnad, stängd, avvisad)
- Referenser (t.ex. till testfallet)

En del av dessa data kan inkluderas automatiskt när felhanteringsverktyg används (t.ex. identifierare, datum, författare och initial status). Dokumentmallar för en felrapport och exempel på felrapporter finns i standarden ISO/IEC/IEEE 29119-3 där felrapporter beskrivs som incident reports.

6. Testverktyg – 20 minuter

Nyckelord

testautomatisering

Utbildningsmål för kapitel 6:

6.1 Verktygsstöd för testning

FL-6.1.1 (K2) Förklara hur olika typer av testverktyg stödjer testning

6.2 Fördelar och risker med testautomatisering

FL-6.2.1 (K1) Komma ihåg fördelar och risker med testautomatisering

6.1. Verktygsstöd för testning

Testverktyg stödjer och underlättar många testaktiviteter. Exempel inkluderar, men är inte begränsade till:

- Hanteringsverktyg – ökar testprocessens effektivitet genom att underlätta hanteringen av SDLC, krav, tester, defekter och konfiguration
- Verktyg för statisk testning – stödjer testaren i att utföra granskningar och i statisk analys
- Testdesign- och testimplementationsverktyg – underlättar generering av testfall, testdata och testprocedurer
- Testexekverings- och testtäckningsverktyg – underlättar automatiserad testexekvering och täckningsmätning
- Verktyg för icke-funktionell testning – tillåter testaren att genomföra icke-funktionella tester som är svåra eller omöjliga att utföra manuellt
- DevOps-verktyg – stödjer DevOps leveranspipeline, uppföljning av aktiviteter i ett arbetsflöde, automatiserade byggprocesser, CI/CD
- Samarbetsverktyg – underlätta kommunikation
- Verktyg som stöder skalbarhet och standardisering av driftsättning (t.ex. virtuella maskiner, verktyg för att kapsla in kod i containers)
- Alla andra verktyg som stödjer vid testning (ett kalkylblad är t.ex. ett testverktyg i testsammanhang)

6.2. Fördelar och risker med testautomatisering

Att bara skaffa ett verktyg garanterar inte framgång. Varje nytt verktyg kommer att kräva ansträngningar för att uppnå verkliga och varaktiga fördelar (t.ex. verktygsintroduktion, underhåll och utbildning). Det finns också vissa risker som behöver analyseras och hanteras.

Potentiella fördelar med att använda testautomatisering inkluderar:

- Tidsbesparing genom att minska manuellt repetitivt arbete (t.ex. exekvera regressionstester, mata in samma testdata igen, jämföra förväntade resultat med faktiska resultat och jämföra med kodningsstandarder)
- Förebyggande av enkla mänskliga misstag genom större konsistens och repeterbarhet (t.ex. att testfall härleds konsekvent från krav, testdata skapas på ett systematiskt sätt och tester exekveras av ett verktyg i samma ordning med samma frekvens)
- Mer objektiv bedömning (t.ex. täckning) och tillhandahållande av åtgärder som är för komplicerade för människor att ta fram
- Enklare åtkomst till information om testning för att stödja testhantering och testrapportering (t.ex. statistik, grafer och aggregerat data om teststatus, felfrekvens och exekveringstid)
- Reducerade testexekveringstider för att möjliggöra tidigare defektdetektering, snabbare återkoppling och snabbare tid för att nå ut till marknaden
- Mer tid för testare att designa nya, djupare och mer effektiva tester

Potentiella risker med att använda testautomatisering inkluderar:

- Orealistiska förväntningar om fördelarna med ett verktyg (inklusive funktionalitet och användarvänlighet)
- Felaktiga uppskattningar av tid, kostnader, arbete som krävs för att introducera ett verktyg, underhålla testskript och ändra den befintliga manuella testprocessen
- Att använda ett testverktyg när manuell testning är lämpligare
- Att förlita sig för mycket på ett verktyg, t.ex. ignorera behovet av mänskligt kritiskt tänkande
- Beroendet av verktygsleverantören som kan gå i konkurs, kan avveckla verktyget, sälja verktyget till en annan leverantör eller ge dålig support (t.ex. svar på frågor, uppgraderingar och felkorrigeringar)
- Att använda en öppen källkodsprogramvara som kan överges vilket innebär att inga ytterligare uppdateringar blir tillgängliga, eller att dess interna komponenter kan kräva relativt frekventa uppdateringar som vid en vidareutveckling
- Automatiseringsverktyget är inte kompatibelt med utvecklingsplattformen
- Att välja ett olämpligt verktyg som inte överensstämde med regulatoriska krav och/eller standarder för funktionssäkerhet

7. Referenser

Standarder

ISO/IEC/IEEE 29119-1 (2022) Software and systems engineering – Software testing – Part 1: General Concepts

ISO/IEC/IEEE 29119-2 (2021) Software and systems engineering – Software testing – Part 2: Test processes

ISO/IEC/IEEE 29119-3 (2021) Software and systems engineering – Software testing – Part 3: Test documentation

ISO/IEC/IEEE 29119-4 (2021) Software and systems engineering – Software testing – Part 4: Test techniques

ISO/IEC 25010, (2011) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models

ISO/IEC 20246 (2017) Software and systems engineering – Work product reviews

ISO/IEC/IEEE 14764:2022 – Software engineering – Software life cycle processes – Maintenance

ISO 31000 (2018) Risk management – Principles and guidelines

Böcker

Adzic, G. (2009) Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, Neuri Limited

Ammann, P. and Offutt, J. (2016) Introduction to Software Testing (2e), Cambridge University Press

Andrews, M. and Whittaker, J. (2006) How to Break Web Software: Functional and Security Testing of Web Applications and Web Services, Addison-Wesley Professional

Beck, K. (2003) Test Driven Development: By Example, Addison-Wesley

Beizer, B. (1990) Software Testing Techniques (2e), Van Nostrand Reinhold: Boston MA

Boehm, B. (1981) Software Engineering Economics, Prentice Hall, Englewood Cliffs, NJ

Buxton, J.N. and Randell B., eds (1970), Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969, p. 16

Chelimsky, D. et al. (2010) The Rspec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends, The Pragmatic Bookshelf: Raleigh, NC

Cohn, M. (2009) Succeeding with Agile: Software Development Using Scrum, Addison-Wesley

Copeland, L. (2004) A Practitioner's Guide to Software Test Design, Artech House: Norwood MA

Craig, R. and Jaskiel, S. (2002) Systematic Software Testing, Artech House: Norwood MA

Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams, Pearson Education: Boston MA

Forgács, I., and Kovács, A. (2019) Practical Test Design: Selection of traditional and automated test design techniques, BCS, The Chartered Institute for IT

Gawande A. (2009) The Checklist Manifesto: How to Get Things Right, New York, NY: Metropolitan Books

Gärtner, M. (2011), ATDD by Example: A Practical Guide to Acceptance Test-Driven Development, Pearson Education: Boston MA

Gilb, T., Graham, D. (1993) Software Inspection, Addison Wesley

Hendrickson, E. (2013) Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing, The Pragmatic Programmers

Hetzel, B. (1988) The Complete Guide to Software Testing, 2nd ed., John Wiley and Sons

Jeffries, R., Anderson, A., Hendrickson, C. (2000) Extreme Programming Installed, Addison-Wesley Professional

Jorgensen, P. (2014) Software Testing, A Craftsman's Approach (4e), CRC Press: Boca Raton FL

Kan, S. (2003) Metrics and Models in Software Quality Engineering, 2nd ed., Addison-Wesley

Kaner, C., Falk, J., and Nguyen, H.Q. (1999) Testing Computer Software, 2nd ed., Wiley

Kaner, C., Bach, J., and Pettichord, B. (2011) Lessons Learned in Software Testing: A Context-Driven Approach, 1st ed., Wiley

Kim, G., Humble, J., Debois, P. and Willis, J. (2016) The DevOps Handbook, Portland, OR

Koomen, T., van der Aalst, L., Broekman, B. and Vroon, M. (2006) TMap Next for result-driven testing, UTN Publishers, The Netherlands

Myers, G. (2011) The Art of Software Testing, (3e), John Wiley & Sons: New York NY

O'Regan, G. (2019) Concise Guide to Software Testing, Springer Nature Switzerland

Pressman, R.S. (2019) Software Engineering. A Practitioner's Approach, 9th ed., McGraw Hill

Roman, A. (2018) Thinking-Driven Testing. The Most Reasonable Approach to Quality Control, Springer Nature Switzerland

Van Veenendaal, E (ed.) (2012) Practical Risk-Based Testing, The PRISMA Approach, UTN Publishers: The Netherlands

Watson, A.H., Wallace, D.R. and McCabe, T.J. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, U.S. Dept. of Commerce, Technology Administration, NIST

Westfall, L. (2009) The Certified Software Quality Engineer Handbook, ASQ Quality Press

Whittaker, J. (2002) How to Break Software: A Practical Guide to Testing, Pearson

Whittaker, J. (2009) Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design, Addison Wesley

Whittaker, J. and Thompson, H. (2003) How to Break Software Security, Addison Wesley

Wiegers, K. (2001) Peer Reviews in Software: A Practical Guide, Addison-Wesley Professional

Artiklar och Websidor

Brykczynski, B. (1999) "A survey of software inspection checklists," *ACM SIGSOFT Software Engineering Notes, 24(1), pp. 82-89*

Enders, A. (1975) "An Analysis of Errors and Their Causes in System Programs," *IEEE Transactions on Software Engineering 1(2), pp. 140-149*

Manna, Z., Waldinger, R. (1978) "The logic of computer programming," *IEEE Transactions on Software Engineering 4(3), pp. 199-229*

Marick, B. (2003) Exploration through Example, http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1

Nielsen, J. (1994) "Enhancing the explanatory power of usability heuristics," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence, ACM Press, pp. 152–158*

Salman. I. (2016) “Cognitive biases in software quality and testing,” *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16), ACM, pp. 823-826.*

Wake, B. (2003) “INVEST in Good Stories, and SMART Tasks,” https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

8. Bilaga A – Utbildningsmål/Kognitiva kunskapsnivåer

Följande definitioner av utbildningsmål (learning objectives) tillämpas i denna kursplan. Varje avsnitt i kursplanen kommer att examineras enligt utbildningsmålet för det. Utbildningsmålen börjar med ett handlingsverb som motsvarar dess kognitiva kunskapsnivå enligt listan nedan.

Nivå 1: Kom ihåg (K1) – kandidaten kan komma ihåg, känna igen och erinra sig en term eller ett begrepp.

Handlingsverb: identifiera, erinra sig, komma ihåg, känna igen.

Exempel:
- "Identifiera typiska testmål."
- "Komma ihåg begreppen i testpyramiden."
- "Erinra sig hur en testare tillför mervärde till en iteration och vid releaseplanering."

Nivå 2: Förstå (K2) – kandidaten kan välja skäl eller förklaringar till påståenden relaterade till ämnet och kan sammanfatta, jämföra, klassificera och ge exempel på testkoncept.

Handlingsverb: klassificera, jämföra, kontrastera, särskilja, exemplifiera, förklara, ge exempel, tolka, sammanfatta.

Exempel:
- "Klassificera de olika alternativen för att skriva acceptanskriterier."
- "Jämföra de olika rollerna i testning" (visalikheter, skillnader eller båda).
- "Särskilja projektrisker och produktrisker”.
- "Exemplifiera syftet med och innehållet i en testplan."
- "Förklara hur sammanhanget påverkar testprocessen."
- "Sammanfatta aktiviteterna i granskningsprocessen."

Nivå 3: Tillämpa (K3) – kandidaten kan genomföra en procedur när han ställs inför en välbekant uppgift, eller välja rätt procedur och tillämpa den i ett givet sammanhang.

Handlingsverb: tillämpa, implementera, förbereda, använda.

Exempel:
- "Tillämpa prioritering av testfall" (ska hänvisa till en procedur, teknik, process, algoritm etc.).
- "Förbereda en felrapport."
- "Använda gränsvärdesanalys för att härleda testfall."

Referenser för de kognitiva nivåerna av utbildningsmål:
Anderson, L. W. och Krathwohl, D. R. (red) (2001) A Taxonomy for Learning, Teaching Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Allyn & Bacon

9. Bilaga B – Spårbarhetsmatris BO-LO

Bilagan listar antalet utbildningsmål (Learning Objectives, LO) relaterade till Verksamhetsmålen (Business Outcomes, BO) och spårbarheten mellan dem.

Business Outcomes: Foundation Level		FL-BO1	FL-BO2	FL-BO3	FL-BO4	FL-BO5	FL-BO6	FL-BO7	FL-BO8	FL-BO9	FL-BO10	FL-BO11	FL-BO12	FL-BO13	FL-BO14
BO1	Förstå vad testning är och varför den är värdefull	6													
BO2	Förstå grundläggande koncept för programvarutestning		22												
BO3	Identifiera teststrategi och aktiviteter som ska genomföras beroende på testsammanhanget			6											
BO4	Bedöma och förbättra kvaliteten på dokumentation				9										
BO5	Förbättra effektiviteten hos testningen					20									
BO6	Anpassa testprocessen till programvaruutvecklingens livscykel						6								
BO7	Förstå testhanteringsprinciper							6							
BO8	Skriva och kommunicera tydliga och begripliga felrapporter								1						
BO9	Förstå de faktorer som påverkar prioriteringar och arbete relaterade till testning									7					
BO10	Arbeta som en del av ett tvärfunktionellt team										8				
BO11	Känna till risker och fördelar med testautomatisering											1			
BO12	Identifiera nödvändiga färdigheter som krävs för testning												5		
BO13	Förstå riskers påverkan på testningen													4	
BO14	Effektivt rapportera om testframsteg och kvalitet														4

Chapter/ section/ subsection	Learning objective	K-level	BUSINESS OUTCOMES													
			FL-BO1	FL-BO2	FL-BO3	FL-BO4	FL-BO5	FL-BO6	FL-BO7	FL-BO8	FL-BO9	FL-BO10	FL-BO11	FL-BO12	FL-BO13	FL-BO14
Kapitel 1	**Grunderna inom test**															
1.1	**Vad är testning?**															
1.1.1	Identifiera typiska målsättningar för testningen	K1	X													
1.1.2	Skilja på testning och debugging	K2		X												
1.2	**Varför är testning nödvändigt?**															
1.2.1	Ge exempel på varför testning är nödvändigt	K2	X													
1.2.2	Komma ihåg relationen mellan testning och kvalitetssäkring	K1		X												
1.2.3	Skilja mellan grundorsak, misstag, defekt och felsymptom	K2		X												
1.3	**Testprinciper**															
1.3.1	Förklara de sju testprinciperna	K2		X												
1.4	**Testaktiviteter, testvara och testroller**															
1.4.1	Summera de olika testaktiviteterna och testuppgifterna	K2			X											
1.4.2	Förklara sammanhangets inverkan på testprocessen	K2			X			X								
1.4.3	Skilja på testvara som ger stöd åt testprocessen	K2			X											
1.4.4	Förklara värdet av att bibehålla spårbarheten	K2				X	X									
1.4.5	Jämföra de olika rollerna i testningen	K2										X				

1.5	**Viktiga färdigheter och god praxis vid testning**															
1.5.1	Ge exempel på de generiska färdigheter som krävs för att testa	K2												X		
1.5.2	Komma ihåg fördelarna med hela teamets ansvar (whole team approach)	K1										X				
1.5.3	Särskilja för- och nackdelar med oberoende testning	K2			X											
Kapitel 2	**Testning under programvarans utvecklingslivscykel**															
2.1	**Testning inom ramen för en programvaruutvecklingslivscykel**															
2.1.1	Förklara vilken inverkan den valda utvecklingslivscykeln för programvara har på testningen	K2						X								
2.1.2	Komma ihåg bra testpraxis som gäller för alla utvecklingslivscykler för programvara	K1						X								
2.1.3	Komma ihåg exempel på angreppssättet för test-first i utvecklingen	K1					X									
2.1.4	Summera hur DevOps kan ha en påverkan på testningen	K2					X	X			X	X				
2.1.5	Förklara angreppssättet shift-left	K2					X	X								
2.1.6	Förklara hur retrospektiv kan användas som en mekanism för processförbättringar	K2					X					X				
2.2	**Testnivåer och testtyper**															
2.2.1	Särskilja de olika testnivåerna	K2		X	X											
2.2.2	Särskilja de olika testtyperna	K2		X												
2.2.3	Skilja på omtestning och regressionstestning	K2		X												
2.3	**Underhållstestning**															
2.3.1	Sammanfatta underhållstestning och dess utlösande faktorer	K2		X					X							
Kapitel 3	**Statisk testning**															

3.1	**Grunder för statisk testning**															
3.1.1	Identifiera de produkttyper som kan granskas med de olika statiska testteknikerna	K1				X	X									
3.1.2	Beskriva värdet med statisk testning	K2	X			X	X									
3.1.3	Jämföra likheter och skillnader mellan statisk och dynamisk testning	K2				X	X									
3.2	**Granskningsprocess och återkoppling**															
3.2.1	Identifiera fördelarna med tidig och frekvent återkoppling från intressenter	K1	X			X						X				
3.2.2	Summera aktiviteterna i granskningsprocessen	K2			X	X										
3.2.3	Känna igen vilka ansvarsområden de viktigaste rollerna har vid granskningar	K1				X								X		
3.2.4	Jämföra likheter och skillnader mellan de olika granskningstyperna	K2		X												
3.2.5	Känna igen de faktorer som bidrar till en framgångsrik granskning	K1					X							X		
Kapitel 4	**Testanalys och design**															
4.1	**Översikt över testtekniker**															
4.1.1	Skilja mellan black-box-, white-box- och erfarenhetsbaserade testtekniker	K2		X												
4.2	**Black-box-testtekniker**															
4.2.1	Använda ekvivalensklassindelning för att härleda testfall	K3					X									
4.2.2	Använda gränsvärdesanalys för att härleda testfall	K3					X									
4.2.3	Använda testning med hjälp av beslutstabeller för att härleda testfall	K3					X									
4.2.4	Använda tillståndsbaserad testning för att härleda testfall	K3					X									
4.3	**White-box-testtekniker**															
4.3.1	Förklara kodsatstestning	K2		X												

4.3.2	Förklara kodgrenstestning	K2		X												
4.3.3	Förklara värdet med white-box-testning	K2	X	X												
4.4	**Erfarenhetsbaserade testtekniker**															
4.4.1	Förklara felgissning	K2		X												
4.4.2	Förklara utforskande testning	K2		X												
4.4.3	Förklara checklistebaserad testning	K2		X												
4.5	**Samarbetsbaserade testangreppssätt**															
4.5.1	Förklara hur användarberättelser skrivs i samarbete med utvecklare och verksamhetsrepresentanter	K2				X						X				
4.5.2	Klassificera de olika alternativen för att skriva acceptanskriterier	K2										X				
4.5.3	Använda accteptanstestdriven utveckling (ATDD) för att härleda testfall	K3					X									
Kapitel 5	**Hantering av testaktiviteterna**															
5.1	**Testplanering**															
5.1.1	Exemplifiera syftet med och innehållet i en testplan	K2		X					X							
5.1.2	Känna igen hur en testare kan ge mervärde till iterations- och releaseplanering	K1	X									X		X		
5.1.3	Jämföra likheter och skillnader mellan start- och avslutskriterier	K2				X		X								X
5.1.4	Använda uppskattningstekniker för att beräkna önskad testarbetsinsats	K3							X		X					
5.1.5	Tillämpa prioritering av testfall	K3							X		X					
5.1.6	Komma ihåg testpyramidens egenskaper	K1		X												
5.1.7	Sammanfatta testkvadranterna och deras samband med testnivåer och testtyper	K2		X							X					

5.2	**Riskhantering**															
5.2.1	Identifiera risknivåer genom att använda sannolikhet och påverkan för risker	K1							X						X	
5.2.2	Skilja mellan projektrisker och produktrisker	K2		X											X	
5.2.3	Beskriva hur produktriskanalyser kan påverka testningens fullständighet och omfattning	K2					X				X				X	
5.2.4	Förklara vilka åtgärder som kan vidtas som svar på analyserade produktrisker	K2		X			X								X	
5.3	**Testövervakning, teststyrning och testavslut**															
5.3.1	Komma ihåg mätetal som används i testningen	K1									X					X
5.3.2	Summera syften med, innehåll och mottagare för testrapporter	K2					X				X					X
5.3.3	Exemplifiera hur man kommunicerar teststatus	K2												X		X
5.4	**Konfigurationshantering**															
5.4.1	Sammanfatta hur konfigurationshanteringen stödjer testningen	K2					X		X							
5.5	**Felhantering**															
5.5.1	Skriva en felrapport	K3		X						X						
Kapitel 6	**Testverktyg**															
6.1	**Verktygsstöd för testning**															
6.1.1	Förklara hur olika typer av testverktyg stödjer testning	K2					X									
6.2	**Fördelar och risker med testautomatisering**															
6.2.1	Komma ihåg fördelar och risker med testautomatisering	K1					X						X			

10. Bilaga C – Kommentarer till denna utgåva

ISTQB® Foundation Syllabus v4.0 är en stor uppdatering baserad på Syllabus för Foundation Level (v3.1.1) och Agile Tester 2014 Syllabus. Därför finns det inga detaljerade kommentarer per kapitel och avsnitt. En sammanfattning av de huvudsakliga förändringarna ges dock nedan. Dessutom tillhandahåller ISTQB® i ett separat Release Notes-dokument spårbarhet mellan utbildningsmålen (LO) i version 3.1.1 av Foundation Level Syllabus, 2014 versionen av Agile Tester Syllabus och utbildningsmålen i den nya Foundation Level v4 .0 Syllabus, som visar vilka LO som har lagts till, uppdaterats eller tagits bort.

När Syllabus 4.0 skrevs (2022-2023) har mer än en miljon människor i mer än 100 länder genomfört Foundation Level-examineringen, och mer än 800 000 är certifierade testare över hela världen. Med antagandet att alla dessa har läst Foundations Syllabus (eller dess översättningar) för att klara examineringen, gör detta att Foundation Syllabus sannolikt kommer att bli det mest lästa testdokumentet för programvara någonsin! Denna stora uppdatering görs med hänsyn till detta arv och för att förbättra synpunkten hos hundratusentals fler människor på den kvalitetsnivå som ISTQB® levererar till den globala testgemenskapen.

I den här versionen har alla LO redigerats för att göra dem atomära och för att skapa en-till-en-spårbarhet mellan LO och kursplansavsnitt, och därmed inte ha bara ett innehåll utan att också ha ett LO. Målet är att göra denna version lättare att läsa, förstå, lära sig och översätta, med fokus på att öka praktisk användbarhet och balansen mellan kunskap och färdigheter.

Denna release har följande ändringar:

- Minskning av hela kursplanen. Kursplanen är inte en lärobok, utan ett dokument som tjänar till att beskriva de grundläggande delarna av en introduktionskurs om programvarutestning, inklusive vilka ämnen som bör tas upp och på vilken nivå. Därför, speciell:
    - I de flesta fall exkluderas exempel från texten. Det är kurshållarens uppgift att tillhandahålla exemplen, såväl som övningarna, under utbildningen
    - Checklistan för att skriva Syllabus har följts Den föreslår maximal textstorlek för LO på varje K-nivå (K1 = max 10 rader, K2 = max 15 rader, K3 = max 25 rader)
- Minskning av antalet LO jämfört med Syllabi för Foundation v3.1.1 och Agile v2014
    - 14 K1 LO jämfört med 21 LO i FL v3.1.1 (15) och AT 2014 (6)
    - 42 K2 LO jämfört med 53 LO i FL v3.1.1 (40) och AT 2014 (13)
    - 8 K3 LO jämfört med 15 LO i FL v3.1.1 (7) och AT 2014 (8)
- Mer omfattande referenser till klassiska och/eller respekterade böcker och artiklar om programvarutestning och relaterade ämnen tillhandahålls
- Större ändringar i kapitel 1 (Grunderna inom test)
    - Avsnittet om testfärdigheter utökat och förbättrat
    - Avsnitt om hela teamet ansvar (K1) tillagt
    - Avsnitt om oberoende testning har flyttats från Kap 5 till Kap 1

- Större ändringar i kapitel 2 (Testning under SDLCs)
    - Avsnitt 2.1.1 och 2.1.2 har skrivits om och förbättrats, motsvarande LO har ändrats
    - Mer fokus på metoder som: test-first (K1), shift-left (K2), retrospektiv (K2)
    - Nytt avsnitt om testning i samband med DevOps (K2)
    - Integrationstestnivån är uppdelad i två separata testnivåer: komponentintegreringstestning och systemintegrationstestning
- Större ändringar i kapitel 3 (Statisk testning)
    - Avsnitt om granskningsteknik, tillsammans med K3 LO (tillämpa en granskningsteknik) borttagen
- Större ändringar I kapitel 4 (Testanalys och design)
    - Användningsfallstestning har tagits bort (men finns fortfarande i Kursplan för Advanced Test Analyst)
    - Mer fokus på samarbetsbaserad metod för testning: ny K3 LO om att använda ATDD för att härleda testfall och två nya K2 LO om användarberättelser och acceptanskriterier
    - Beslutstestning och täckning ersatt med kodgrenstestning och täckning (för det första är kodgrenstäckning vanligare i praktiken; för det andra definierar olika standarder ’beslut’ annorlunda i motsats till "gren"; för det tredje löser detta en subtil men allvarlig brist från den gamla FL2018 som hävdar att "100 % beslutstäckning innebär 100 % kodsatstäckning" – den här meningen är inte sann när det gäller program utan beslut)
    - Avsnittet om värdet av white-box-testning har förbättrats
- Större ändringar I kapitel 5 (Testhantering)
    - Avsnittet om teststrategier/angreppssätt har tagits bort
    - Nytt K3-LO om uppskattningstekniker för att uppskatta testinsatsen
    - Mer fokus på de välkända Agile-relaterade koncepten och verktygen inom testhantering: iterations- och releaseplanering (K1), testpyramid (K1) och testkvadranter (K2)
    - Avsnittet om riskhantering bättre strukturerat genom att beskriva fyra huvudaktiviteter: riskidentifiering, riskbedömning, riskreducering och riskövervakning
- Större ändringar I kapitel 6 (Testverktyg)
    - Innehållet om vissa testautomatiseringsproblem minskat eftersom det är för avancerat för grundnivån – avsnittet om verktygsval, genomförande av pilotprojekt och införande av verktyg i organisationen har tagits bort

11. Index