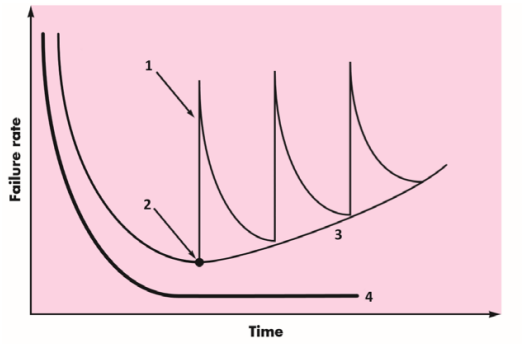
F1 Software engineering

Följande är *korrekta* påståenden om software engineering:
• Off-the-shelf-mjukvara är mjukvara som inte utvecklas på direkt beställning av en kund.
• Ett problem med att utveckla mjukvara är att mjukvara inte är en fysisk (tangible) produkt vilket gör det svårt att överblicka och mäta utvecklingen.
• Den kvalitet som en process för mjukvaruutveckling har återspeglas i kvalitén på mjukvaran.
• Det finns ett samband mellan kvalitet på utvecklingsprocess för mjukvara och kvalitet på mjukvaran i sig.
• Alla utvecklingsprocesser för mjukvara bör innehålla följande aktiviteter: kommunikation, planering, modellering, konstruktion och driftsättning.
• En mjukvaruprodukt kan sägas åldras även om koden för själva mjukvaruprodukten i sig inte ändras.
• Svårigheter med att konstruera mjukvara idag uppstår ur samma problematik som fanns för 40 år sedan.
• Det vi kallar mjukvara omfattar inte bara kod utan även dokumentation i olika former och data.
• För att utveckla mjukvara behöver vi ta hänsyn till vilka processer vi använder, vilka metoder vi använder, vilka verktyg vi använder och hur vi integrerar kvalitet i alla dessa steg.

Software Engineering: Vilka påståenden är korrekta gällande diagrammet?



Sant:
• Kurvan markerad med 4 visar den ideala kurvan för en mjukvaruprodukt.
• Software Engineering innebär att använda metoder i en strävan att få kurva 3 att närma sig det utseende som kurva 4 har.

Falskt:
• Spikarna i kurvan indikerar att testfasen i den använda processen startar vid punkt 2.
• Siffran 3 markerar en kurva som indikerar att en mjukvaruprodukt har dålig kvalitet.
• Spikarna i kurvan beror alltid på att något förändrats i mjukvaruproduktens kod. F. Kurva 4 är samma kurva som man kan påvisa för hårdvara.

Följande påståenden är *korrekta* för legacy systems:
• Ett legacy-system är dyrt att underhålla och kan vara riskabelt att byta ut.
• Mjukvaran i ett legacy system måste anpassas för att möta krav från nya datormiljöer och teknologier.
• Legacy systems är mjukvarusystem som behålls trots att de är kostsamma att underhålla men kostnaden eller risken att byta ut dem anses vara större än underhållskostnaden.

F2 Processmodeller

Följande är *sant* om iterativ utveckling:
• När man utvecklar iterativt upprepas vissa steg i processen minst en gång.
• Iterativ utveckling kan kombineras med prototyping.
• RUP – Rational Unified Process är en plandriven och iterativ utvecklingsmetod.
• Spiralmodeller kan ses som iterativa processer där alla steg upprepas i varje varv i spiralen.

Följande påståenden om utvecklingsprocesser för mjukvara är *sanna*:

• Det som skiljer olika processmodeller åt är hur man rör sig mellan olika ramverksaktiviteter.
• Ramverksaktiviteterna kompletteras av olika övergripande aktiviteter som ständigt pågår.
• Ramverksaktiviteter måste brytas ner i mindre uppgifter.
• Det finns ingen ideal utvecklingsprocess för mjukvara - alla utvecklingsprocesser har situationer där de är mer lämpliga att använda och andra situationer där de inte är lika lämpliga att använda.
• Alla mjukvaruprocesser tar inte samma hänsyn till alla aspekter av mjukvaruutveckling.
• Det är viktigt att skilja mellan process och processmodell när man utvärderar en organisation.
• Det är viktigt att skilja mellan processmodellen och den faktiska processen.
• Processmodeller kan betraktas som processmönster på en hög abstraktionsnivå.
• En processmodell är en abstrakt beskrivning av aktiviteter som ska utföras och processflödet beskriver hur dessa aktiviteter relaterar till varandra.
• Alla processmodeller, både agila och traditionella, behöver på något sätt förhålla sig till följande övergripande aktiviteter - specifikation, design, implementation, validering och evolution.
• Det är alltid mindre kostsamt att identifiera och åtgärda ett fel tidigt i utvecklingen än att identifiera och åtgärda samma fel senare i utvecklingen.
• En processmodell byggs upp av olika aktiviteter och beskriver hur dessa aktiviteter relaterar till varandra.
• En bra beskrivning av en aktivitet innehåller information om input, output, entry criteria/preconditions, exit criteria/postconditions och vilka steg som utförs i aktiviteten.
• Alla processmodeller innehåller på något sätt de olika stegen specifikation, design och implementation, validering och evolution (stegen kan dock beskrivas med olika termer beroende på författaren).

Processmodeller: Vad är *korrekta* påståenden om olika generella processmodeller?
KUGGFRÅGA - INGA SVAR ÄR KORREKTA.

A. Vattenfallsmodellen är en föråldrad processmodell som inte längre ändvänds i praktiken utan fungerar endast som utgångspunkt för att jämföra olika processmodeller.
B. Evolutionära processmodeller är inte iterativa.
C. Inkrementella processmodeller ger en mycket tydlig process.
D. Om man i sin processmodell någon gång låter slut-användare utvärdera produkten innan release så använder man sig av processmodellen prototyping.
E. RUP - Rational Unified Process är en agil processmodell med fokus på use cases/användningsfall.
F. De två angreppssätten top-down och bottom-up för inkrementella processmodeller bör inte kombineras.

Följande är *korrekta* påståenden gällande vattenfallsmodellen:

• Vattenfallsmodellen är den äldsta modellen för mjukvara.
• Vattenfallsmodellen kan vara bra att använda i ett litet projekt med väl definierade och stabila krav och utvecklingsteamet är sedan tidigare bekanta med att utveckla den givna typen av system.
• Vattenfallsmodellen är den grundmodell som vi relaterar andra processmodeller till.
• Vattenfallsmodellen kan i vissa varianter ha feedback-loopar som gör att man återgår till en tidigare aktivitet och på så viss jobbar med ett mer iterativt arbetssätt.
• Vattenfallsmodellen förknippas ofta med V-modellen för att beskriva hur testning relaterar till andra aktiviteter.
• Vattenfallsmodellen är den tidigaste processmodellen för mjukvaruutveckling men har med tiden blivit allt mer kritiserad.

Ex. på varför man använder protoyping:

• Det är billigare att justera en produkt tidigt i utvecklingsprocessen än att göra förändringar efter att man påbörjat implementeringen
• För att förstå kundens behov bättre.

Följande påståenden om evolutionära utvecklingsprocesser för mjukvara är *korrekta*:

• Evolutionära utvecklingsprocesser är alltid iterativa.
• Evolutionära processmodeller arbetar efter så kallad top-down-metod.
• Spiralmodeller kan ses som en typ av evolutionära processer.
• Spiralmodeller räknas in som en undergrupp till evolutionära processmodeller.
• Prototyping är en variant av evolutionära processer.

Följande påståenden om spiralmodeller för mjukvaruutveckling är *korrekta*:

• Spiralmodeller är risk-orienterade.
• Spiralmodeller drivs av riskanalyser.
• Spiralmodeller försöker kombinera de systematiska delarna från vattenfallsmodellen och flexibiliteten hos iterativa modeller.
• Spiralmodeller införlivar prototyping som en återkommande aktivitet.

Processmodeller: Vad är *korrekta* påståenden om inkrementella processer?

• Inkrementella processmodeller är alltid också iterativa.
• Med en inkrementell process så finns det en risk att helhetsgreppet om arkitekturen går förlorat.
• Inkrementella processer kan användas med en stor release när alla delar är klara eller med flera mindre releaser.

Vem är bl.a en intressent (stakeholder) till ett projekt som utvecklare av mjukvaruprodukt A?

• Utvecklaren som producerar mjukvaruprodukten A
• Den som köper mjukvaruprodukten A
• Den som använder mjukvaruprodukten A

Vad är ex. på en intressents roll (stakeholder's role) i ett projekt?

• Att ge förslag till krav
• Att diskutera produktmå
• Att köpa och använda produkter
• Att vara tillgänglig och ge input till utvecklingsteamet
• Att diskutera kraven
• Att validera produkten

F3 Agila metoder

Följande påståenden om den agila utvecklingsprocessen XP är *korrekta*:

• Ofta förekommande refactoring är en del av processen i XP.
• Det finns inga traditionellt utformade krav specificerade i naturligt språk i XP.
• Konstruktion av enhetstest sker alltid före annan kodning.
• XP utgår från att kunden alltid finns tillgänglig och arbetar tillsammans med utvecklingsteamet.

Ex. på vilka principer som är viktiga inom den agila modellen XP:
• Par-programmering
• Testdriven utveckling
• Enhetstestning/Unit testing

Följande påståenden gällande agila utvecklingsprocesser för mjukvaruutveckling är *sanna*:
• Agila utvecklingsprocesser fokuserar ofta på ett smalare område eller färre aspekter av utveckling än traditionella utvecklingsprocesser.
• Med agil utveckling så är ofta kostnaden för förändring högre i början av ett projekt än vad kostnaden för förändring under samma tid skulle vara om en mer traditionella utvecklingsprocess användes.
• Agil utveckling försöker inte skapa mer kontroll över vad som sker utan försöker istället vara mer flexibel för när förändring sker.
• Agil utveckling kan skalas upp till stora projekt med fler än 10 personer men är inte alltid lika väl anpassade för detta som traditionella utvecklingsprocesser.
• Alla agila utvecklingsprocesser är inkrementella i sin natur.

Följande alternativ är exempel på agila principer:
• Välkomna ändrade krav, även sent i utvecklingen.
• Leverera fungerande mjukvara ofta, från med ett par veckors till ett par månaders mellanrum, ju oftare desto bättre.
• Fungerande mjukvara är det huvudsakliga sättet att mäta framsteg.
• Arbetsgruppen bör regelbundet reflektera över hur man kan bli mer effektiva och anpassa sitt arbetssätt efter detta.
• Enkelhet – man bör maximera mängden arbete som inte behöver göras.
• At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
• Business people and developers must work together daily throughout the project.
• The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
• Continuous attention to technical excellence and good design enhances agility.
• Working software is the primary measure of progress.
• Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Följande *antaganden* gällande projekt för utveckling av mjukvara ligger till grund för uppkomsten av det agila paradigmet:
• Det är svårare att förutsäga vad som bör ske under analys, design, konstruktion och test än vad vi från början trott.
• Design och konstruktion behöver alltid överlappa varandra i praktiken.
• Det är svårt att förutsäga hur krav kommer att förändras under projektets gång.

Följande *problem* kan uppstå med agila processer:
• Det kan i praktiken vara svårt att följa en princip om att inte förhandla om fast pris på kontrakt utan utveckla på löpande räkning om man inte vill sätta ett slutdatum.
• Självorganiserande grupper kräver stor disciplin av utvecklarna och kan skapa problem i en hierarkisk linjeorganisation. • Att uppnå den nivå av interaktion med kunden och/eller slutanvändaren som många agila processer förespråkar kan vara svårt då det är kostsamt för kunden.

Följande påståenden gällande utvecklingsprocessen Scrum är *korrekta*:

• Scrum-möten, i form av ca 15-minuter långa möten där man kort diskuterar vad man har arbetat med, planerar att arbeta med och vilka problem man har, ska hållas varje dag med alla i teamet närvarande.
• Scrum utgår från självstyrande grupper som till stor del själva avgör vilket arbete som genomförs i sprinten.
• Scrum definierar tydligt när och hur nya krav får införas i projektet.
• Scrum definierar ingen roll som direkt matchar mot en projektledare.
• Scrum utgår från att en ständig processförbättring sker via Scrum teamets retrospect-möten.
• I Scrum så utformas först en övergripande projektplan med planerade milstolpar och releaser.
• Processförbättring är en del av processen som beskrivs i Scrum.
• Product backlog och sprint backlog är två viktiga dokument i Scrum.
• Scrum är en lämplig processmodell om kraven är oklara/otydliga.

F4 Kravhantering

Vad är ett krav?

• En kort och koncis bit av information.
• Säger något om systemet.
• Alla intressenter har kommit överens om att det är giltigt.
• Det hjälper till att lösa kundens problem.

Följande punkter anses vara *god praktik* när man samlar in krav:

• Intervjua potentiella användare.
• Samla intressenter och brainstorma med dem.
• Använda en focus-grupp bestående av alla intressenter.
• Skicka ut formulär till potentiella användare.
• Analysera företagsdokumentationen.
• Observera användarna av systemet som ska utvecklas.
• Intervjua potentiella användare.
• Göra brainstorming med intressenter.
• Använda en focus grupp bestående av mjukvaru-utvecklare.

Kravhantering - Följande är *korrekta* påståenden om krav och kravhantering:

• Domänkrav kan vara både funktionella eller icke-funktionella/kvalitativa krav.
• Ett användarkrav/user requirement ger oftast upphov till flera olika systemkrav.

Följande ska man *undvika* när man formulerar krav i naturligt språk:

• Termer/beskrivningar med mer än en rimlig tolkning (ambiguity) • Synonymer.
• Långa meningar med komplexa bisatser • Flera krav som formuleras till ett enskilt krav.

Ex. på hur man ska *formulera* ett funktionellt krav i naturligt språk:

• Man ska skriva korta meningar som beskriver systemets funktionalitet
• Man ska använda modala hjälpverb såsom ska (shall), bör (should) och måste(must).
• Man ska använda bilder och exempel för att hjälpa till att förklara funktionalitet.

Kravhantering - Följande påståenden om kravdokumentation är *korrekta:*

• Krav bör prioriteras utifrån olika aspekter och prioriteringen dokumenteras.

Följande krav-modeller är *mest använda* under kravdokumentering:
• Data flow diagrams
• Scenario-based models
• Class diagrams
• Entity relationship diagrams

Nedanstående alternativ är kravstilar (olika sätt att skriva krav):
• Virtuella fönster
• Användningsfall
• Kontext diagram
• Datamodell

Följande är funktionella krav:
• Krav på utdata produkten ska producera.
• Krav på beräkningar produkten ska göra.
• Systemet ska registrera en betalning.
• Systemet ska beräkna ett resultat.
• Systemet ska kunna skriva ut en faktura.
• Användaren ska kunna välja hur många uppgifter som ska visas per sida.
• Systemet ska första dagen i varje månad generera en rapport som visar vilka som är de mest använda söktermerna under den föregående månaden.
• Användaren ska kunna ange vikt i både kilogram och hekto.

Följande är icke-funktionella krav:
• Krav på hur gränssnittet ska se ut
• Krav på hur effektiv produkten ska vara
• Krav på hur dokumentationen ska se ut
• Krav på vilket operativsystem produkten kan köras på
• Systemet ska klara av att 100 samtidiga användare gör en förfrågan till systemet var 5:e minut utan att svarstiden blir längre än 0,5 sekunder.
• Systemet ska använda sig av element definierade enligt HTML5 så som beskrivet i rekommendation av W3C.
• Systemet ska vara kompatibelt med grafikkort ur serie UX-89 av Notel.

F5 Use cases/Spårbarhet

Följande är *korrekta* påståenden om användningsfall/use cases:
• Stor täckning av kraven.
• Användningsfall beskriver vad ett system ska göra men inte hur detta utförs i kod.
• En aktör kan vara en person som interagerar med systemet men kan också representera ett annat system.
• Utförliga scenarion eller användningsfallsbeskrivningar kan användas som testfall.
• Begreppet extend används för att beskriva hur ett användningsfall eventuellt omfattar funktionalitet från ett annat användningsfall.

Följande påståenden om use case modeling är *sanna:*
• Varje use case representerar en uppgift som involverar en interaktion med ett system
• Use cases stödjer spårbarhet.
• Use cases är en teknik för att modellera krav.
• Aktörer i use cases kan vara personer eller system.
• Use cases är bra för att få en bild över krav på projektet.

Följande är *sant* gällande spårbarhets(-matris):
• Spårbarhet används till exempel när man vill göra ändringar i kraven.
• ID behövs för att kunna spåra.
• Spårbarhet är bra för underhållning (maintenance), det blir då lättare att underhålla systemet.
• Spårbarhet kan användas för att tydliggöra samband mellan kraven och testfallen.
• Spårbarhet kan användas för att tydliggöra samband mellan krav och design.
• Spårbarhet kan användas för att tydliggöra samband mellan krav och källan(intressenter).

Varför är spårbarhet viktigt? Bl.a:
• Det blir lättare att göra ändringar i kraven.
• För att det underlättar underhåll (maintenance).
• För att det underlättar verifiering.

Följande *fördelar* finns det ex. med att använda användningsfall:
• De underlättar design av testfall.
• De stödjer spårbarhet.
• De är lätta att förstå för icke-tekniska intressenter.
• De ger ett dynamiskt perspektiv på kraven.

Följande alternativ kan vara ett användningsfall:
• Anställa personal
• Checka in
• Beställa paket
• Ta ut pengar
• Reservera biljett

F6 Arkitektur

Följande påståenden gällande olika arkitekturella stilar/mönster är korrekta:
• I en datacentrerad/repository arkitektur så utgör databehållaren en svag punkt.
• En nackdel med en lagerarkitektur är att den kan bli långsam.
• Prestandan hos en klient-server-arkitektur är beroende på det nätverk som används.
• En nackdel för pipe-and-filter-arkitekturer är att gränssnitten för dataformaten måste bestämmas på förhand och kan inte bara bytas ut i ett filter.
• En fördel med datacentrerade/repository arkitekturer är att de olika modulerna är oberoende av varandra.
• En fördel med en lagerarkitektur är att säkerheten kan bli hög om varje lager ansvarar för att filtrera indata.
• Referensarkitekturer används för att visa mönster för lämplig arkitektur för en viss typ av system.
• Arkitekturen för ett system måste stödja alla aspekter av systemet.

Vilken arkitektonisk stil lämpar sig bäst vid transformering av input-data till output-data genom en serie av beräknande eller manipulativa komponenter?
• Data flow

Vilken arkitektonisk stil lämpar sig bäst vid hantering av stor mängd av data?
• Data-centered (repository or blackboard)

I system-arkitektur pratar man om hur man ska strukturera ett system. Följande element använder man sig av när man bygger en sådan struktur:
• Componentents
• Connectors
• Properties

Vad menas med Architectural styles/patterns?
• Det är ett mönster man kan applicera till designen.
• Det används inom arkitektur för att rita/beskriva.

F7 Metrics

Följande meningar är *sanna* för mjukvaru-metrics:
• Mätning används för att förutsäga en framtida situation.
• Mätning används för att identifiera felaktiga komponenter.
• Mätning tillåter att man kvantifierar programvaran och processen.
• Mätning används för att utvärdera den aktuella situationen.
• En bra metric har matematiska egenskaper som gör att värden har betydelse i förhållande till storlek.
• En bra metric kan eller har valideras empiriskt i olika sammanhang.
• En bra metrics värde förhåller sig relativt till den egenskap som mäts. Ökar värdet så ökar graden av egenskapen.
• En indikator är en metric som ger någon insikt om en process eller produkt.

Följande påståenden om olika metrics är *korrekta:*
• McCabes Cyclomatic Complexity kan användas för analys av risken för fel i koden.
• Cohesion och coupling är två metrics som kan användas för både objektorienterad och procedurell kod.
• Coupling som uppstår för att två kod-moduler använder varandras variabler är sämre än coupling som uppstår för att funktioner anropas mellan kodmodulerna.

F8 Implementation/Underhåll

Bl.a följande aktiviteter gör man under konfiguration hantering (configuration management):
• Change management
• Version management
• System building
• Release management

Följande påståenden gällande configuration management är *korrekta:*
• En mainline byggs upp av olika baselines • I en software configuration ingår dokumentation, kod och data
• En build innebär att skapa ett exekverbart system utifrån en mainline.

Följande aktiviteter stöds av versionshantering: (kommer enligt Kristina inte på tentan)
• Version and release identification
• Storage management
• Independent development (cooperation and collaboration support)
• Project support

Följande är *sant* om versionshanteringssystem:
• Alla förändringar som görs i koden listas och sparas.
• Git är ett versionshanteringssystem.
• Man kan hämta en gammal version av programmet/systemet.
• Alla förändringar som görs i koden listas och sparas.

Varför är det viktigt att spara varje version av systemet som man testar och att även hålla koll på vilken version som är vilken?
• För att om man stöter på problem i en senare version är det bra jämföra med en tidigare version för att se var det gick fel.
• För att lättare hålla koll på vilka versioner som har blivit testade och vilka buggar som har blivit fixade.

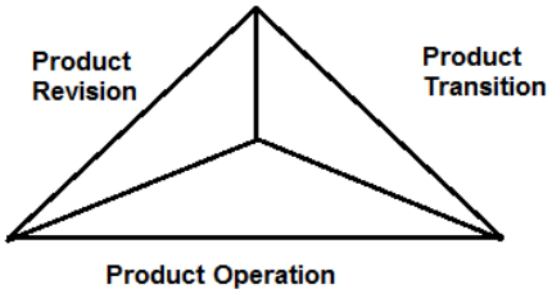
Följande alternativ brukar räknas med som en kostnad inom "Buisness Value of testing":
• "Cost of Prevention"
• "Cost of Detection"

Följande påståenden om underhåll av mjukvara är *korrekta:*
• Supportability är en kvalitetsfaktor som relaterar till hur lätt det är att hålla en mjukvara i drift under hela dess livstid.
• Så mycket som 70 % av ett företags resurser kan gå till att underhålla mjukvara man släppt tidigare.
• Underhåll innebär inte endast att uppdatera kod och fixa buggar utan även att uppdatera och återskapa dokumentation.
• Underhåll av mjukvara startar så fort vi släppt en version för användning av slutanvändare.
• Underhåll av mjukvara bör vara integrerat i samma team som utförde eller utför nyutveckling av produkten.
• I vissa fall kan det vara mer kostnadseffektivt att utföra en total nyutveckling för att fylla ett behov än att med mindre ändringar underhålla den programvara som i nuläget är avsedd att täcka behovet.
• En konsekvens av en ändringsförfrågan kan vara att man inte gör någon åtgärd även om det är en legitim bug. • Underhåll kan delas in i att hantera defekter och att hantera support och service.
• Begreppet usage month används för att lättare kunna avgöra när problem med en släppt mjukvaruprodukt kommer att nå sin topp efter release.

Följande påstående gällande implementation av mjukvara är *korrekt*:
• Generellt så bör performance hos kod utvärderas endast när detta verkligen behövs och under utveckling så bör andra kvalitetsaspekter förmodligen prioriteras framför performance.

F9 Kvalitet/SPI/Riskhantering

Följande kvalitetsfaktorer för mjukvara tillhör kategorin Product Operation i McCalls modell:
• Correctness
• Integrity
• Reliability
• Efficiency
• Usability



Följande påståenden gällande kvalitet och kvalitetskostnader är *korrekta:*
• Felkostnader/Failure costs kan delas in i interna och externa felkostnader.
• Kostnader för uppskattningar/Appreciasial costs genereras bland annat av testning.
• Att planera testning är en typ av förebyggande kvalitetskostnad.
• Kvalitet kan vara olika saker beroende på synvinkel.
• Kvalitet på en mjukvaruprodukt är ofta starkare förknippat med icke-funktionella krav än med funktionella krav.
• Kvalitetsegenskaper för en mjukvaruprodukt tar hänsyn till andra intressenter än slutanvändaren.
• Kvalitetsfaktorer behöver brytas ner till mätbara mål för det specifika projektet.
• Standarder är ett verktyg att använda för kvalitetskontroll.
• Det finns ett överlapp för de kvalitetsfaktorer som anges i ISO 9126 och av McCall.

Följande alternativ innebär så kallade appreciasial costs/kostnader för uppskattning för kvalitet:
• Testning
• Reviews
Inte: • Planering • Kostnader för att åtgärda fel innan release • Utbildning • Koordinering av kvalitetskontroll.

Följande aktiviteter genererar så kallade förebyggande kvalitetskostnader (prevention costs):
• Att planera testning
• Utbildning av personal

Följande påståenden för riskhantering är *korrekta:*
• Kända risker identifieras genom att utvärdera det aktuella projektet. Förutsägbara risker identifieras utifrån erfarenhet från tidigare projekt.
• En händelse kan innebära både risk för produkt, projekt och organisation samtidigt.
• Att planera för riskhantering innebär att planera hur en risk undviks, hur en risk kan minimeras och att ta fram en plan för att risken inträffar/contingency plan.
• En process för riskhantering bör innehålla identifiering, analys, planering och övervakning.
• Ett projekt bör alltid ha färdiga åtgärdsplaner att implementera om en identifierad risk inträffar.
• För att utvärdera hur allvarlig en risk är behöver man uppskatta sannolikheten att risken ska inträffa och hur allvarliga konsekvenser som uppstår om risken inträffar.
• Den grundläggande idén är att hitta de viktiga buggarna först.
• Höga risk-nummer innebär att det behövs omfattande testning för den funktionen eller det möjliga problemet.

Följande är *korrekt* om riskbaserad teststrategi:
(obs! skillnad på riskbaserad testning/teststrategi och risker generellt)
• Risker används för planering och rapportering av status.
• Hitta de viktigaste buggarna först.

I förebyggande riskhantering ingår:
• Ta fram åtgärder för att minska sannolikheten för att en identifierad risk inträffar.
• Övervaka identifierade risker för att kunna avgöra om status för risken ändras.
• Ta fram en åtgärdsplan för om en identifierad risk inträffar.
• Arbeta för att identifiera nya risker efter att projektet startat.

Följande påståenden är *sanna* om Border value analysis:
• Denna teknik kan tillämpas på alla nivåer av tester: unit-, integration- och systemnivå.
• Hjälper testaren att välja de mest effektiva värdena från värdeintervall.
• Det är ofta optimalt att ska skapa tre testfall för varje gränsvärde.

Följande alternativ är *riktlinjer* för en border value analysis:
• Om inmatningsförhållandena specificerar ett antal värden så bör tester utvecklas som använder max- och minimivärdena.
• Om inmatningsförhållandena specificerar ett intervall mellan värdena a och b, prova vad som händer om värden precis över och under dessa värden matas in.
• Om interna datastrukturer har fasta gränser, utforma ett test för att prova vad som händer när dessa gränser är nådda.

Border Value Analysis: Ett matteprov har följande gränser: under 10p ger U, 10-15 ger G och 16 eller högre ger VG (max är 20. Endast hela poäng ges). Vilka värden måste testas för att Border Value Analysis testet ska vara fullständigt?
• 0, 10, 16, 20
• "hello"
• 9, 15, 19
• -1, 21
• 1, 11, 17
Inte: 5, 12, 18

Följande påståenden om SPI/processförbättring är *korrekta:*
• En organisations mognadsgrad är avgörande för vilken typ av förbättringar som man kan lyckas införa.
• För att lyckas genomföra en förändring i en process så måste hela organisationen stötta denna. Det går inte att tvinga igenom en ändring uppifrån eller nerifrån i en hierarki.
• SPI är en iterativ process utan direkt slutmål.
• Att identifiera mätvärden/metrics är en viktig del av SPI.
• Alla delar i en förbättringsstrategi bör inte implementeras direkt utan införas en i taget för att kunna mäta dess effektivitet.
• Utvärdering av en organisations mognadsgrad är en viktig del av SPI.
• Där existerar ett samband mellan kvalitet på process och kvalitet på produkt vilket innebär att förbättra processen innebär en förbättring av produkten.
• Processförbättringar kan inte drivas igenom i en organisation endast underifrån eller ovanifrån.
• SPI kan användas i alla typer av organisationer.

Hur gör man ett border value analysis?

• Man kontrollerar att programmet hanterar gränsvärdena på alla olika gränser som är inkluderade i programmet på ett korrekt sätt.

F10 Testning

Nedanstående påståenden om testning *stämmer:*

• Vid statisk testning behöver man inte köra någon kod.
• Verifikation innebär att man kontrollerar att man bygger produkten rätt.
• Validering innebär att man kontrollerar att man bygger rätt produkt.

Följande påståenden är *korrekta* för "Psychology of testing":

• "Ad hoc"-tester är ostrukturerade.
• Det är inte möjligt att verifiera att allt fungerar.
• I verifiering, frågar vi oss om vi bygger produkten rätt.
• I validering frågar vi oss om vi bygger rätt produkt.
• Ber oss göra den bästa möjliga testningen som är möjlig på den tid som är tillgänglig.
• Används primärt för att hitta fel i koden.

Följande påståenden *gäller* för "black box testing":

• Programmeringskunskaper är ej nödvändiga för "black box test".
• Kunskap om systemets funktionalitet är nödvändigt för att göra "black box test".
• "Black box test" utförs oftast i slutet av projektet.
• "Black box test" fokuserar på att testa systemet så att det uppfyller de krav som ställts.
• Programmeringskunskaper är ej nödvändiga för black box test.
• Black box testet fokuserar på att testa systemet så att det uppfyller de krav som givits.

Följande påstående är *korrekta* om White-box testning och Black-box testning:

• Med White-box-testning har man kunskap om kod och struktur.
• Kod måste finnas tillgänglig för att utföra en White-box-testning.

Vilka av följande alternativ räknas som en kostnad inom Business Value of testning?

• Cost of Prevention
• Cost of Detection

Granskning - Följande påståenden är *sanna:*

• Granskning är en form av .sk. "Statisk testning"
• Om man ber en kompis kolla igenom ens kod, så är detta en form av granskning.
• "Inspektion" är en mer formell typ av granskning än "Walkthrough

F11 Unit testing

Följande påståenden är *korrekta* om Unit test:

• Unit test innebär att man testar ett systems minsta exekverbara kod isolerat.
• Hittar felen tidigt.
• Ett unit test kan innebära att man testar en ihopsatt komponent, bestående av flera klasser som inte kan exekvera individuellt.

• All pairs är en testmetod för att testa kombinationer av indata.
• Junit är ett program för automatiserad testning där man kan initiera ett system med testfall och automatiskt jämföra resultat med förväntade resultat.
• Inmatning av felaktig indata är ett sätt för att spåra fel i ett program. Vid test-driven utveckling bör alla tester av en ny funktionalitet.

Följande påståenden är *korrekta* om Code Coverage:
• En nackdel med code coverage-baserad testning är att man måste ha tillgång till koden.
• Koden måste vara tillgänglig.
• En fördel med code coverage-baserad testning är att det är lätt att mäta graden av täckning.
• Det finns många verktyg för att stödja code coverage-baserad testning. Det behövs 3 testfall för att få full branch coverage i nedanstående bild.
Det är ett systematiskt sätt att skapa ”test case”
• Code Coverage kan användas för att se vilka delar i din funktion/program som faktiskt körs.
• Code coverage är ett mått på hur mycket av din kod som exekveras beroende på vilka tester som körs.
• Ett program med hög code coverage är mer noggrant testat och har en lägre sannolikhet att innehålla programvarufel än ett program med låg kodtäckning.
• Det finns flera olika sätt att mäta Code coverage på, ett exempel är branch coverage.
• Path coverage innebär att man testa att varje distinkt väg genom koden exekveras minst en gång inom testet.
• Med full ”Path Coverage” måste alla möjliga vägar i systemet köras minst en gång
• Det går mäta resultatet av hur mycket av systemet som är testat.
• Kan användas för att se om alla delar i din funktion/ditt program faktiskt körs.
• Code coverage är ett mått på hur många rader/block/delar av din kod exekveras beroende på vilka tester som körs.
• Ett program med hög kodtäckning (code coverage) är oftast mer noggrant testad och har en lägre chans att innehålla programvarufel än ett program med låg kodtäckning.
• Det finns flera olika sätt att mäta Code coverage på, ett exempel är branch coverage.
• Path coverage innebär att testa att varje distinkt väg genom koden exekveras minst en gång inom testet.
•”Path Coverage” går *inte* att testa i black-box.

Vad stämmer om ”Statement coverage”, ”Branch coverage” och ”Path coverage”?
• Resultaten blir mätbara om man använder dem.
• Det finns gott om verktyg som stöder dem
• De ingår i systematiska metoder för att skapa testfall.

Följande är *korrekt* gällande equivalence partitioning:
• Tekniken kan användas när som helst under en testfas och kräver därmed ingen färdig produkt.
• Equivalence partitioning innebär att en uppsättning indata kan delas in i grupper där medlemmarna i gruppen förväntas bedömas likvärdigt av systemet.
• Om en grupp med likvärdig indata ger ett förväntat resultat kan övriga grupper med indata därmed förväntas ge samma resultat

Ett program låter användaren mata in ett heltal, och skriver sedan ut huruvida talet är positivt, negativt eller noll. Programmet ska testas enligt ”equivalence partitioning”.
Följande alternativ innehåller testdata som borde placeras inom samma ”equivalence class”:
”minus sju”, ”noll”, ”fyra”
47, 13, 991
-4, -13, -48,9
1, 2, 3

Ett program låter användaren mata in ett heltal, och programmet skriver sedan ut huruvida talet är positivt, negativt eller noll. Programmet ska testas enligt ”equivalence partitioning”. Vilka av följande alternativ innehåller data inom samma ”equivalence group”?
”minus sju”, ”noll”, ”fyra”
47, 13, 991
1,2,3

F12 Integration/Systemtest

Följande påståenden är *korrekta* om integrationstest:
• ”Big Bang” är en av de kända principerna för Integrationstestning.
• Processen av ”Top-Down”-integration innebär att delar av koden ersätts med så kallade ”stubbar”.
• Integrationstest testar om fungerande självständiga komponenter fungerar tillsammans.
• Integrationstest nämns i V-modellen.
• Integrationstest är en av ett antal olika testnivåer som används för att testa mjukvaran.
• Integrationstest tar enhetstestade komponenter och testar om de fungerar tillsammans enligt systemets design.
• Det är fasen i programvarutestning i vilken individuella mjukvarukomponenter kombineras och testas i en grupp.
• En fas där fel och buggar upptäcks i ett delvis färdigt system.

Följande är *sant* om så kallad "Smoke testing" inom mjukvarutveckling:
• Ett test som vanligtvis körs dagligen (eller varje gång systemet byggts).
• Ett bra sätt att ta reda på om det uppstått något fel i det senaste "bygget".
• Inte ett komplett test utan testar enbart av de viktigaste funktionerna.

Koncept: Integrationstest. Följande steg gäller för ”Bottom-Up”-integration:
• Tester körs varje gång en komponent integrerats.
• ”Drivers” tas bort och ersätts succesivt med komponenter allteftersom integrationen fortsätter ”uppåt” i strukturen. Den sista Drivern ersätts av huvudprogrammet.

Följande påståenden är *sanna* för systemtester:
• Systemtester är nästan alltid black-box tester.
• Systemtester är något man oftast gör i grupp.
• Systemtester kan göras med hjälp av användare

Följande begrepp förekommer inom system-testning:
• ”Smoke test”
• ”Black-box”
• ”Use-cases”

• ”Performance test”
• ”Soap Opera test”

System-test: Varför är det viktigt att spara varje version av systemet som man testar och att även hålla koll på vilken version som är vilken?
• För att om man stöter på problem i en senare version är det bra att jämföra med en tidigare version för att se var det gick fel.
• För att lättare hålla koll på vilka versioner som har blivit testade och vilka buggar som har blivit fixade.

Påståendena nedan *stämmer* för reverse engineering:
• Begreppet completeness/kompletthet handlar om hur mycket detaljer som finns på en viss abstraktionsnivå.
• Reverse engineering kan vara en envägs-process eller en tvåvägs-process där information används för att förbättra befintlig kod.
• Automatiska verktyg kan vara till stor hjälp vid reverse engineering för att få fram information på olika abstraktionsnivåer.
• Code restructuring kan vara lämpligt att tillämpa på den gamla koden innan man påbörjar reverse engineering.

Ex. på vad ett kontext diagram är:
• Ett diagram som representerar programvarans helhet samt hur det interagerar med utomstående komponenter.
• Ett diagram som representerar alla input som matas in och produceras i programmet.

Vad är ex. på varför man behöver rita ett kontext-diagram?
• För att visa alla input som matas in och produceras i systemet
• För att visa hur systemet interagerar med utomstående komponenter

Följande ”entity-attribute-metrics” associationer är rätt: (kommer enligt Kristina inte på tentan)
OO Code, size, number of classes
Code, structure, number of statements
Code, quality, defect density
Requirements specification, stability, number of requirements changes
Source code, lenght, LOC (lines of code)
Programmer, productivity, LOC/per day
Code, quality, defect density
Requirements specification, stability, number of requirements changes

***Sortera* följande steg inom requirements-based testing i ordning med start på ”Definiera färdigställda testkriterier”:**
1. Designa testfall.
2. Prioritera testfall
3. Utföra tester.
4. Verifiera testresultat.
5. Verifiera testtäckning.
6. Spåra och hantera fel..