Sammanfattningar Essentials of Software Engineering

F10, Testning

Quality Assurance (QA) inkluderar testning. Testning är en aktivitet som handlar om att utvärdera produktens kvalitet, och att förbättra den, genom att identifiera defekter och problem.

Bästa sättet att uppnå kvalitet är genom att ha en väl-definierad process som passar organisationen och projektet, + att alla teammedlemmar ska vara engagerade och vilja göra sitt bästa samt använda rätt tekniker.
Högkvalitativ process → större chanser till högkvalitativ produkt.
Det är dock inte ofta som den ideala processen uppnås, därför är testning av artefakterna viktigt genom hela projektet.

Olika tekniker för att hitta fel i ett program:

- **Testning:** att designa testfall, köra programmet i en kontrollerad miljö och verifiera att output är korrekt.

- **Inspektioner och granskningar:** en eller flera personer utöver skaparen av produkten tittar på den och granskar den för att hitta fel. Bra att applicera på kravspecifikationer.

- **Formella metoder:** matematiska tekniker för att ”bevisa” att ett program är korrekt

- **Statiska analyser:** att analysera strukturen i ett program. Ofta automatiserade, hittar misstag och felbenägna villkor i programmet.

Vi skiljer på ***fault***, ***failure*** och ***error***.
Människor gör ***errors (misstag)*** som kan leda till att mjukvara innehåller ***faults (***buggar, defekter***)***. När man kör mjukvara med ***faults*** kan det leda till ***failure***.

Testning handlar om att ställa frågor till systemet och jämföra svaret med någon typ av ”korrekt” svar.

- **Hur ställer vi frågorna?**
- **Hur vet vi svaret (från systemet)?**
- **Hur vet vi vad som är det ”korrekta” svaret, att jämföra med?**
  → Är det en del av kraven?
  → Räknas det ut manuellt?
  → Kan man jämföra med tidigare versioner eller andra system?
  → Kan man använda en fördefinierade test suites (en samling testfall som är

menade för att testa mjukvara)?
→ Ska man använda ett ”oracle”? (eller Heuristics, dvs riktlinjer?)

Testning görs utifrån olika testfall som blir grunden för hur man ska utföra testningen.

För att bestämma vad ett förväntat resultat av ett testfall ska bli kan vi använda ett **test oracle.** Ett test oracle är en källa med information om huruvida output från ett testfall är korrekt eller inte. Det kan specificera korrekt output för alla möjliga inputs eller endast för specifika inputs. Det behöver inte specificera specifika output-värden utan kan också bara visa inom vilka begränsningar output-värdet behöver vara.

Testfall

De 3 delar som behöver ingå i ett testfall är:
- Ett set med input-kriterier
- ”Execution conditions”, alltså steg i testet, som inkluderar inputvärdena
- Förväntade resultatet

Scenario	Test Step	Expected Result	Actual Outcome
Verify that the input field that can accept maximum of 10 characters	Login to application and key in 10 characters	Application should be able to accept all 10 characters.	Application accepts all 10 characters.
Verify that the input field that can accept maximum of 11 characters	Login to application and key in 11 characters	Application should NOT accept all 11 characters.	Application accepts all 10 characters.

Varje testfall ska ha ett unikt ID.

Hur kan vi generera och välja testfall?

Antingen genom intuition, vilket innebär att man låter den som utför testet fritt välja vad de vill testa.

Eller genom specifikationerna. Man baserar alla testfall på vad som står i specifikationerna utan att kolla på koden. Tekniker som baseras på specifikationer kallas för **black-box testing**.

Ett annat sätt att välja testfall är genom att titta på koden. Man baserar testfallen på kunskaper om själva koden, och definierar ett visst mått av vad koden ska täcka

för funktioner för att sedan designa testfall för att uppnå det måttet. Tekniker baserade på koden kallas ofta för **white-box testing**.

Black-box testing: Tekniker baserade på specifikationerna

White-box testing: Tekniker baserade på koden

Vem är det som utför testningen?

Vi har tre alternativ: programmerare, testare eller användare.

Programmerare utför ofta unit testing. De skapar testfall och kör dem medan de skriver koden.

Testares jobb är att skriva testfall och se till att de körs. Här krävs andra kunskaper än endast programmeringskunskaper. Vissa professionella testare analyserar resultaten från testfallen och kan utifrån det bedöma kvalitetsnivån av produkten.

Användare är bra att ha med i testningen, för de hittar ofta användbarhetsproblem + att de använder väldigt många olika inputs som är bra för att testa systemet mot riktiga scenarion.

Vad är det som testas?

- **Unit testing**: att testa de individuella enheterna av funktionalitet, t. ex. en enda funktion.
- **Functional testing**: att testa så att de individuella enheterna fungerar som en funktionell enhet när man slår samman dem.
- **Integration and system testing**: att testa den integrerade funktionaliteten av det kompletta systemet.

Enligt föreläsningen är nivåerna istället dessa:

- **Unit:** testar de individuella enheterna (som ovan)
- **Integration**: testa två eller fler enheter tillsammans (som ovan men annat namn)
- **System**: testa funktionella och icke-funktionella egenskaper i hela systemet.

Skillnaden mellan boken och föreläsarens är sista nivån (System eller Integration and system testing). Föreläsarens nivå handlar inte bara om att testa funktionaliteten utan även om att testa icke-funktionella egenskaper hos systemet.

Två olika sätt att mäta kvalitet på ett program:

Verifiering: Gör vi produkten på rätt sätt? Handlar om att kontrollera att produkten uppfyller kraven och specifikationerna. Man spårar en mjukvaruprodukts krav genom utvecklingsfasen, och när mjukvaran ska gå från en fas till en annan så verifierar man att kraven hittills är mötta.

Validering: Gör vi rätt produkt? Att kontrollera att den färdiga mjukvaruprodukten möter användarnas krav och specifikationerna. Kan oftast bara göras i slutet på projektet med ett fullständigt mjukvarusystem.

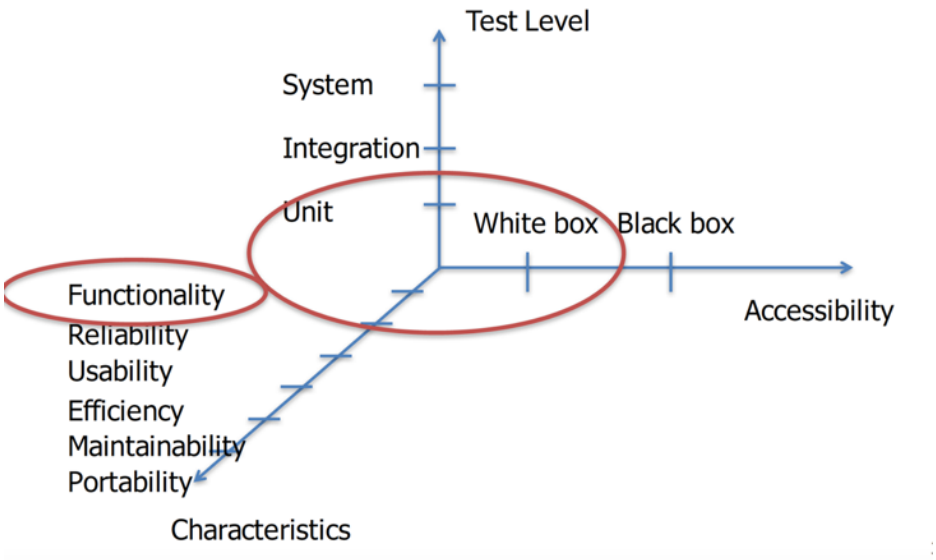
Olika tekniker för att utföra verifiering: inspektion och granskning, testning, statisk analys och formella metoder.

Olika tekniker för att utföra validering: testning och inspektioner.

F11, Unit-testing

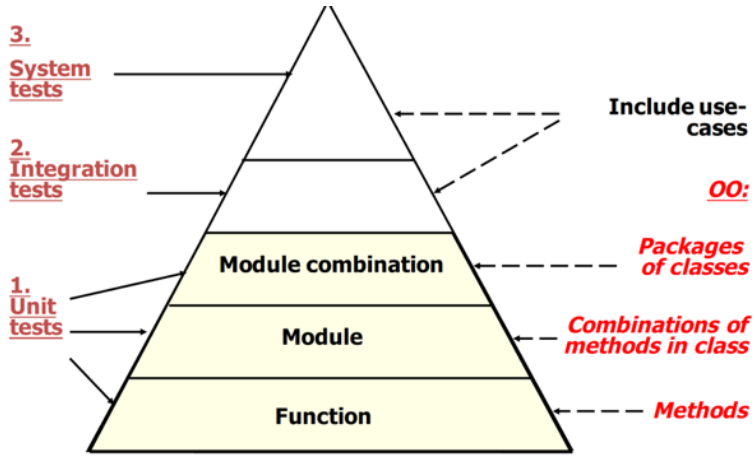
Det finns tre olika nivåer av testning:

- **Unit**
- Integration
- System



En unit kan i ett testsammanhang vara lite olika saker:

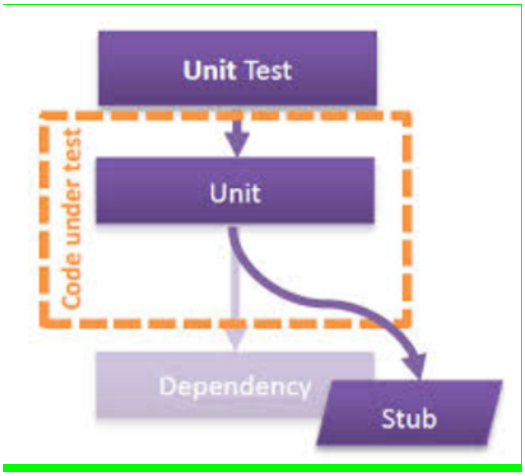
- Individuella funktioner / metoder inom en viss klass
- Objektklasser med flera attribut och metoder
- Sammansatta komponenter, som består av flera klasser, med definierade gränssnitt som används för att få tillgång till deras funktionalitet



Definitionen av unit testing är att man ska testa individuella kodenheter i isolering. Detta innebär att alla beroenden med andra kodenheter måste brytas. Detta kan

man göra genom att använda sig av så kallade **stubs** (stubbar) eller **drivers.** **Stubbar** är de moduler som **blir kallade på**, medan **drivers** är de funktioner **som kallar på andra funktioner**.

En stubbe innehåller samma gränssnitt som den beroende kodenheten, men ingen funktionalitet. Istället kan den förse med förbestämda svar på förfrågningar.



Olika tekniker för att ta fram testfall:

Equivalence Class Partitioning

En black-box teknik (alltså baserad på specifikationerna). Görs denna alltså som white-box teknik för unit testing då, och som black-box när det gäller integration och system testing?

Grundtanken med denna teknik är att input och output från en komponent kan delas in i klasser, som enligt komponentens specifikation, kommer att behandlas likadant av komponenten. Då räcker det med att testa ett enda värde från varje klass, som kan representerar hela klassen.

Det är viktigt att ta med både **giltiga** och **ogiltiga** värden i testfallen.

Ex. om en tenta har följande gränser: under 10p = U, 10-15 = G, 16 och uppåt = VG. Maxpoäng man kan få är 20.
→ Ekvivalensklasserna blir då 0-9, 10-15, 16-20. Då väljer man ett värde från varje klass, t. ex. 6, 11 och 17 att testa.
Sedan måste man även testa ogiltiga värden. I detta fall allt som är mindre än 0 och större än 20, samt icke-heltal. Dessa kan vara 1.55, -2, 23 och ”tjoho”.

Om testfallet misslyckas för ett värde så förväntar vi oss alltså att det misslyckas för alla andra medlemmar i den klassen också, och tvärtom om den lyckas så lyckas den för alla.

Border Value Analysis

BVA är baserad på Equivalence Partitioning och är därmed också en black-box teknik. Grundtanken är att programmerare är benägna att göra fel när de ska programmera för gränserna i klasserna.
Därför bör man testa gränsvärdena i varje klass, alltså; precis under min-värdet, min-värdet själv samt precis över min-värdet; precis under max-värdet, max-värdet och precis över max-värdet.

Anledningen till att det är användbart att testa på gränserna är:
- För att det är en sannolik källa till programmerings-fel, t. ex. att man råkar skriva < istället för <=.
- För att kraven ofta är lite luddiga när det gäller beteendet vid gränsvärden
- För att man kan upptäcka interna dolda begränsningar i koden

All pairs

En metod för att testa kombinationer av input.
Den grundläggande principen för denna metod är att de flesta fel orsakas av interaktioner mellan som mest två faktorer.

Om man ska testa varje kombination av varje typ av input så får vi ett väldigt stor antal kombinationer att testa. Kan bli tusentals, tiotusentals… Därför finns det denna teknik, All Pairs, för att generera endast så många kombinationer som behövs för att täcka alla par av värden. Varje värde från varje variabel, ska paras ihop minst en gång men varje värde från varje annan variabel.

Det finns All-pairs verktyg som genererar ett passande set med testfall.

White-box testning
= metoder baserade på den interna strukturen av koden.

Används oftast i unit testing.

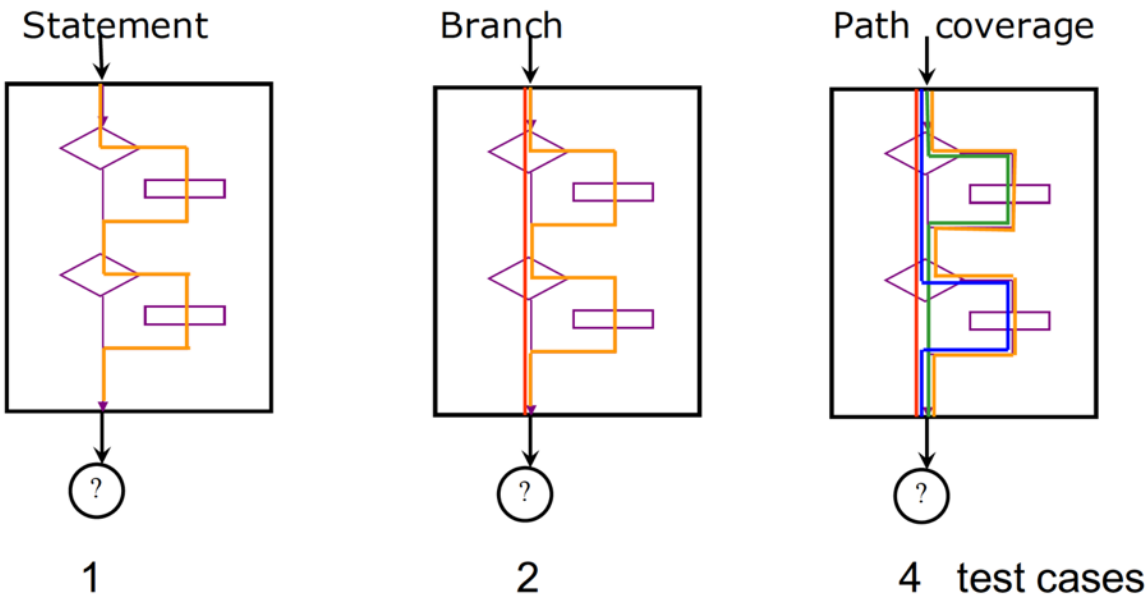
Path Analysis

Två uppgifter i path analysis:
1. Analysera antalen paths som existerar i ett system eller ett program
2. Bestäm hur många vägar som ska inkluderas i testet.

Hur många vägar som ska inkluderas i de olika testfallen bestäms genom att man ska se till att varje statement är täckt av något test och varje branch är täckt av något test.

Detta kallas för **coverage-based testing**.
Det finns tre grundläggande typer av code-coverage:

- Statement coverage: se till att varje statement utförs åtminstone en gång i något av testfallen
- Branch coverage: se till att för varje beslutspunkt i grafen så ska varje branch väljas åtminstone en gång
- Path coverage: se till att varje unik path genom hela control-flow grafen utförs åtminstone en gång i något av testfallen



Man kan använda sig av **cyklomatisk komplexitet** för att få en indikation för hur många testfall som behövs för att täcka en bit kod.
CC är då den övre gränsen för Branch coverage (BC), och den lägre gränsen för Path coverage (PC).

Fördelar med coverage-based testing:
- Det är ett systematiskt sätt att utveckla testfall
- Det finns mätbara resultat – alltså hur täckande det blir
- Det finns många stödverktyg att använda, t. ex. testdata-generatorer

Nackdel:
- Koden måste vara tillgänglig

Olika verktyg för code coverage mäter olika saker, så det är viktigt att ta reda på vad verktyget man använder mäter.

Test-driven development

= att skriva unit tests innan man skriver koden, och sedan använda dessa tester för att se till att koden fungerar. Detta gör att du först får ange kraven som en testfall, och sedan implementera dem.

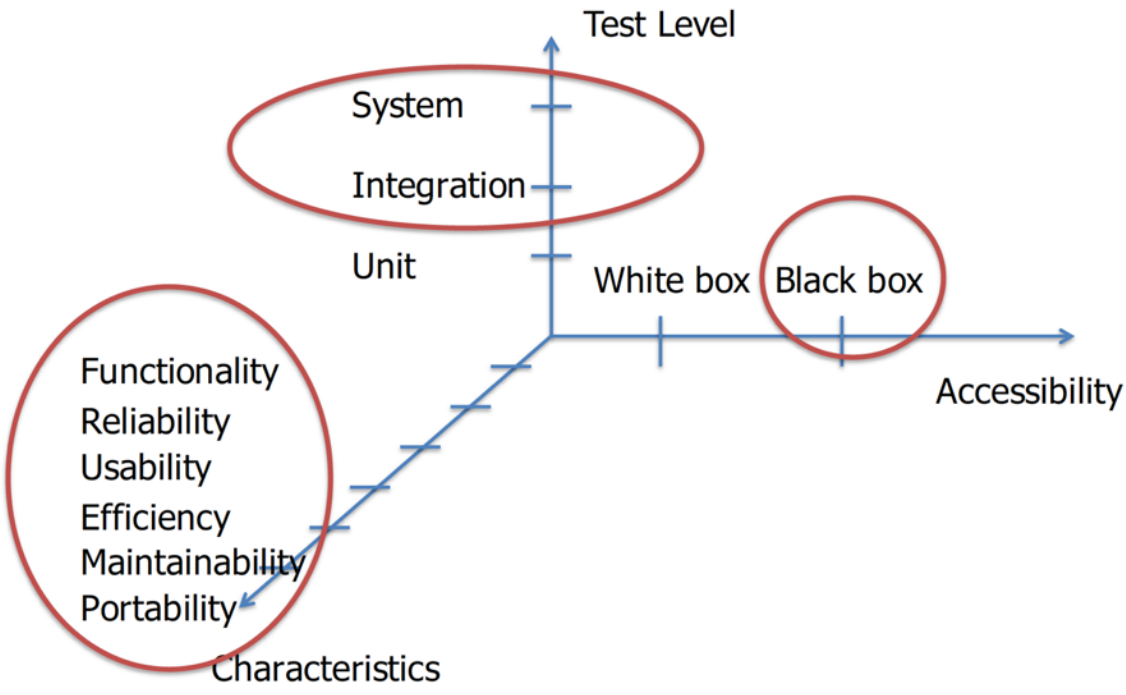
Process för detta:

1. Skriv ett testfall
2. Verifiera att det testfallet misslyckas
3. Ändra korden så att det testfallet lyckas (målet är att skriva så enkel kod som möjligt, utan att bry sig om framtida tester eller krav)
4. Kör testfallet för att verifiera att det nu fungerar och kör de tidigare testfallen för att verifiera att koden inte har fått någon annan ny defekt.
5. Refactor the code, för att göra den bättre.

Fördelar med TDD:

- Man skriver endast skriver små kodsnuttar, testar, sedan fortsätter skriva.
- Man får först ange kraven som ett testfall, och sedan implementera dem. Detta gör att du måste förstå kraven först innan du implementerar dem.
- Leder ofta till att man skriver fler tester och enklare kod. Uppnår ofta åtminstone statement coverage.
- Det är en effektiv teknik som hjälper programmerare att snabbt bygga pålitlig kod.

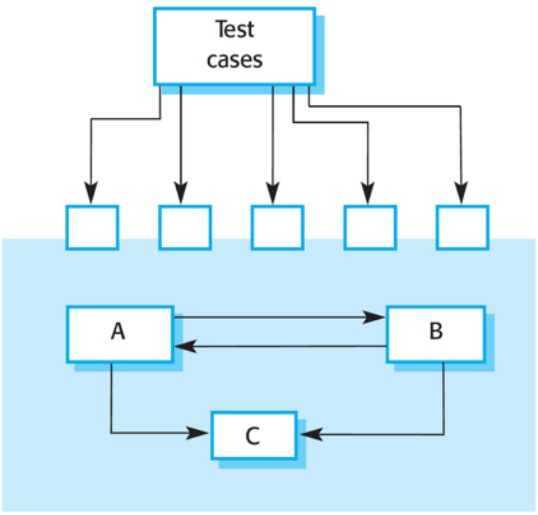
F12, Integration testing, system testing, reviews and test reporting



Integration testing

= man testar två eller fler enheter tillsammans, för att se till att de individuella enheterna fungerar som en funktionell enhet

Här skapar man testfall som testar gränssnittet mellan moduler.



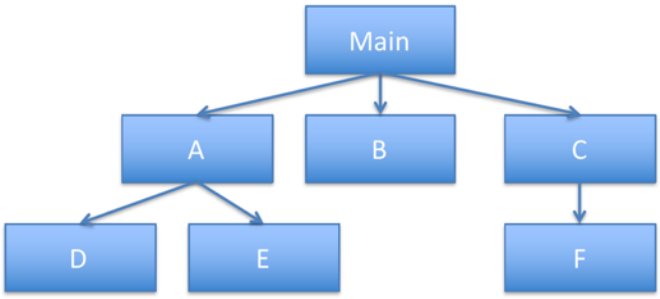
Problem som kan finnas i gränssnittet mellan två komponenter:

- Interface misuse: en komponent kallar på en annan komponent men använder dess gränssnitt felaktigt. T. ex. att parametrarna används i fel ordning.

- Komponenten som kallar på den andra har inbäddade antaganden om beteendet hos den andra komponenten som är felaktiga.

- Komponenten som kallar och komponenten som blir kallad på opererar i olika hastighet och utgången information blir hämtad.

Man vill alltså försöka hitta fel i gränssnittet. För att kunna göra det behöver man en s.k. **”call graph”**, eller **”dependency graph”**.



Anledningen till att man behöver en call graph är för att man ska få en förståelse av systemet, och kunna spåra flödet av värden mellan olika funktioner.

Olika strategier för integration testing:

- ”Big Bang” = Alla moduler/komponenter integreras samtidigt. Sedan testar man allt, man har alltså inga mellanliggande test. Inga moduler integreras förrän alla moduler är färdiga.

- ”Top Down” = Här börjar man med huvudmodulen och lägger sedan till moduler genom att följa call graph:en. Man integrerar de viktigaste modulerna först.
- ”Bottom Up” = Här börjar man med löven i call graph:en, och lägger sedan till moduler genom att följa call graph:en uppåt i hierarkin.
- Use Case / Scenario = Man integrerar alla delar av koden som tillhör ett visst scenario och testar dessa.

Moduler som inte är färdiga än och alltså inte går att testa med använder man stubbar eller drivers för att ersätta.

Stubbar / stubs används för att simulera funktionalitet hos en modul som man behöver **kalla på** för att kunna testa en modul. = called

Drivers används för att simulera funktionalitet hos en modul som **ska kalla på** modulen man testar. = callers

System Testing

= systemtestning under utvecklingen handlar om att skapa en version av de i princip färdiga systemet och testa detta

Använder nästan alltid black-box tekniker.

Det är viktigt att ha en bra **versionshantering** för att kunna veta vilken version som testats vart buggar har blivit fixade.

Man kan använda **use cases** som identifierar systemets interaktioner som en bas för att testa **funktionalitet** under systemtestningen.

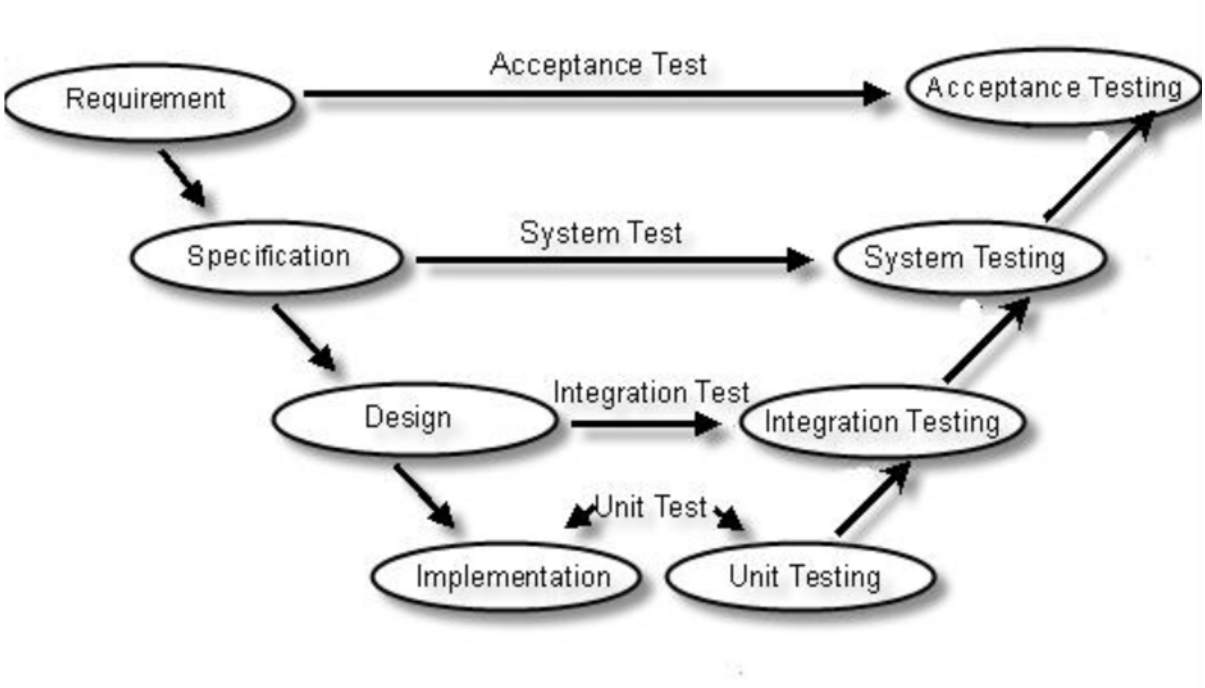
Man testar också **icke-funktionella krav** under systemtestning.

Vilka metoder används?

Man kan använda metoder så som **EP** och **BVA** för att utveckla testfall även här, men sedan finns det fler metoder. Exempelvis dessa:

- **Heuristics**, t. ex. felgissning. ”Vad brukar gå fel i den här typen av mjukvara?”
- **Bug taxonomy** – vilka är de vanligaste buggarna man har hittat hittills i projektet eller från de här utvecklarna?

The V-model:



Systemtester kan vara **scripted** eller **exploratory**.

- **Scripted** innebär att man följer ett set med instruktioner (skrivna i förväg) när man gör testet.

- **Exploratory** innebär att man skapar testfallen och de förväntade outputen under testsessionen.
  - dessa kan vara ”time boxed”
  - dessa kan vara baserade på en ”tour”, t. ex. att man går igenom alla funktioner.

Systemtester kan baseras på individuella funktionella krav, icke-funktionella krav, användarscenarion (use cases) eller scenarion.

Ett koncept för testning kallas för **”Soap Opera tests”** – det innebär att man testar en överdriven eller sammanträngd verklighet. Man använder sig av användarscenarion men gör dem mer komplicerade, precis som man gör med handlingen i soap operas.
→ Detta gör att funktioner och egenskaper testas snabbare och på en högre nivå än i riktiga situationer.

Icke-funktionella krav blir allt viktigare i systemtestningen

Man bör testa alla egenskaper i ISO9126.

Idag ligger ofta fokus på **performance testing**.
För att testa denna utför man ofta Load Tests och Stress Tests.

- Load test = testa om systemet kan hantera den belastning som det är gjort för
- Stress test = kan systemet hantera korta perioder av overload, och återhämta sig snyggt från detta?

Även andra tester kan prioriteras, så som **säkerhetstester** (penetrationstester, integritetstester) och **användbarhetstester** (kan användaren utföra alla operationer i systemet? Kan användare installera och börja använda systemet utan hjälp?)

Olika typer av tester:

Release testing: man testar en viss release av ett system, som ska användas utanför utvecklingsteamet

Acceptance testing: utförs ofta av kunden. Ibland är det dock kunden som definierar testfallen men sedan utförs de av utvecklingsteamet (under övervakning av kund).

User tests (INTE användbarhetstester): Uppdelade i två grupper:
- **Alpha tests**: en begränsad grupp testar tidigare versioner av systemet, för att få feedback tidigt
- **Beta tests**: en stor grupp testar den nästan färdiga produkten, för att hitta buggar relaterade till configuration eller för att testa specifika användarscenarion som är svåra att härma under labbtester.

Crowd testing: Liknar Alpha- och Beta testning, men här har de som utför testerna en formell roll i testningsorganisationen och kan få specifika instruktioner över vad som ska testas. Testarna kan betalas per aktivitet eller per bugg de hittar.

Static vs Dynamic Testing

Note: Both types of testing are important.

Static testing	Dynamic testing
Without executing the program	By executing the program
Analysis process	Verification process
Prevention of defects	Finding and fixing the defects
Evaluates code/documentation	Finds bugs/bottlenecks in the system
Cost of defect is less	Cost of defect is high
Return on investment will be high (early stage)	Return on investment will be low (after development phase)

Static testing

Automatisk statisk analys (ASA)

Ett verktyg analyserar koden och producerar varningsmeddelanden och felmeddelanden.

Här kan det finns både **false positives**, varningar när det egentligen inte är något problem (false alarm), och **false negatives**, problem som verktyget inte hittar.

Låg kostnad men har hög påverkan.

Reviews

Människor läser och kommenterar koden och dokumentationen. Kräver inte att programmet körs.
Fördelar med reviews:

- Man hittar buggar
- Man minskar risker
- Man ser till att alla kan förstå dokumentationen

Alla typer av dokument kan granskas. Kravspecar, designdokument, kod, testfall, användarmanualer osv.

Det finns olika typer av reviews:

- Tekniska reviews (Peer review är en förenklad sådan.)
- Inspections

- Walk-through

Den *minst* formella typen av review är när två personer, författaren och en kollega, diskuterar koden.

Peer review = författaren av dokumentet ger det till en kollega som får granska det, sedan gör kollegan granskningen självständigt och skriver formella kommentarer till författaren av dokumentet.

Walkthrough = möte. Författaren går igenom dokumentet och så får några granskare ge kommentarer. Efter detta får författaren fundera över kommentarerna och sedan uppdatera dokumentet.

Inspection = möte. Mer *formell* än de andra två. Man har formella roller, t. ex. chairman och recorder. Författaren kan vara närvarande men det är inte hen som presenterar dokumentet. Materialet som ska granskas skickas ut i förväg så att alla individuellt kan granska det först. En reader (inte författaren) leder gruppen genom materialet. Inspektörerna ger kommentarer som diskuteras.
Här kan man använda sig av checklistor när man går igenom dokumentet.

En jämförelse av olika tekniker

	Static	White Box	Black Box
What	Test without running	Test how system works	Test what system does
Why	Identify bugs before they're built	Identify bugs in functions or interfaces	Identify bugs in system results and behavior
Who	All stakeholders	Typically developers	Typically testers
Phase	During specification and development	Usually unit, component, integration	Usually integration, system, and acceptance
Tools	Simulators, code analyzers, spell- and grammar checkers, graphic/diagramming, spreadsheets	Profilers, coverage analyzers, harnesses, debuggers, data generators, test case generators	Intrusive and non-intrusive GUI tools, load generators, performance tools, data generators

Dokument från testning

- Testplan
- Testfall
- Buggrapporter
- Testverktyg och automatiserad testning
- Metrics, statistik och sammanfattning

Vad kan ingå i en testledares rapport till projektledningen?

- Kvalitetsrisker – hur många av de identifierade riskerna har testats hittills?
- Defekter – hur många har hittats, hur många har fixats, m.m.

- Testfall – hur många har skapats, hur många har utförts, inte utförts, osv.
- Coverage – hur mycket kod har blivit täckt av testning hittills?
- Kostnad – hur mycket tid har vi spenderat hittills?

Dessa olika aspekter visar man ofta i en graf.

Att avsluta testning

Att hålla koll på testresultaten och observera statistiken. Olika sätt:

Problem find rate – antalet problem man hittat per timma varje dag i en graf, se hur de minskar med tiden. Man kan ha ett mål i planen som säger att när problem find rate har minskat till en viss nivå så slutar man testa.

S-curve – ett annat sätt att mäta när man kan sluta testa. Visar en kurva över hur antalet defekter ökar, och när den kurvan har nått en viss stabilitet så slutar man testa.

Pepper code with defects – sedan sortera kodens defekted i seeded defects och i riktiga defekter. Utifrån detta kan man sedan anta ungefär hur många fel som finns kvar i koden.

De kostnader som vägs mot varandra i en beräkning av ROI är kostnaderna för att förebygga testning och kostnaderna för att ha dålig kvalitet på slutprodukten. ROI för testning är oftast flera hundra procent.