Föreläsning 4:
Design och praktisk testning

ETSA01 Ingenjörsprocessen 1 - Metodik VT15 | Markus Borg

Min frånvaro

Spårbarhet för säkerhetskritiska programvarusystem
- Bilindustri - ISO 26262
- Processautomation - IEC 61511
- Flygindustri - DO178c
- Kärnkraft - IEC 61513
- Tågtrafik - IEC 62279
- Medicinteknik - IEC 62304
- Saknades: försvarsindustri, rymdfart och sjöfart

Åt vilken intressent utvecklas säkerhetsstandarder?
- Utvecklarna?
- Certifierarna?
- Samhället i stort?

Standarder

Utvecklare

Samhället

Certifierare

Agenda

Programvarudesign
- Arkitekturdesign
- Objektorienterad design
- Design av användargränssnitt

Programvarutestning - del 2
- White-box testing
- Demo – verktyg för testning
  - Hexawise – kombinatorisk testning
  - JUnit – enhetstestning
  - CodeCover - kodtäckning

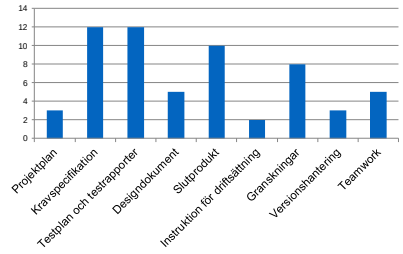
Projektinfo

Ingen deadline idag!

- Jobba vidare mot L4 (Mån 27/4 kl. 23.59)
- L4 = Kravspecifikation 1.0
  - Baseline och formell ändringshantering

Kort om projektbetygen

60 poäng totalt, fördelade på 9 kategorier



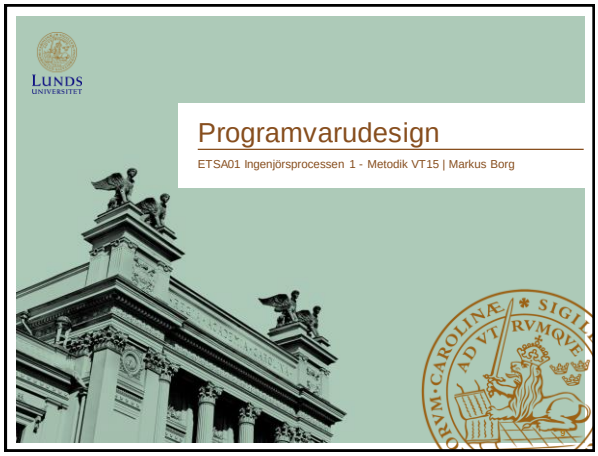
Några riktlinjer

Generellt: +välskrivet +struktur +förtroendeingivande
Projektplan: +tydlig +realistiska risker –avsnitt saknas
Kravspecifikation: +"krav med kvalitet" +matchar målen
Designdokument: -matchar inte slutprodukten
Test: +heltäckande +enkla att utföra –överlappande tester
Slutprodukt: +bra UI –krav ej uppfyllda –går att krascha
Instruktion: +tydlig
Granskningar: +olika typer av brister –protokoll saknas
Versionshantering: +referenser stämmer +ändringshantering korrekt
Teamwork: +feedback +kommunikation –deadlines

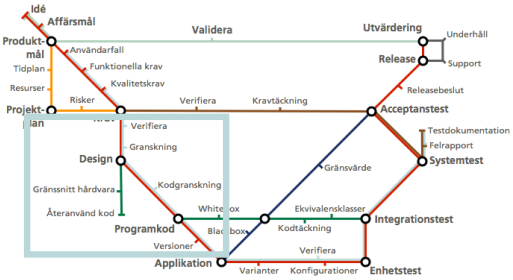
Övning 4a och 4b

Samma upplägg som första kursveckan
Övning 4a på onsdag kl. 13-15 eller 15-17
Hemarbete 2 h
Övning 4b på torsdag kl. 8-10 eller 10-12

Sista övningarna!



Design

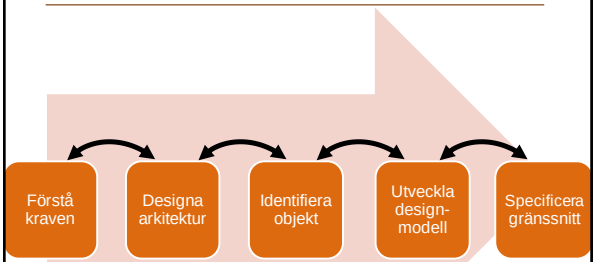


Design

är både en aktivitet

och ett resultat

Modell för design av objektorienterad programvara

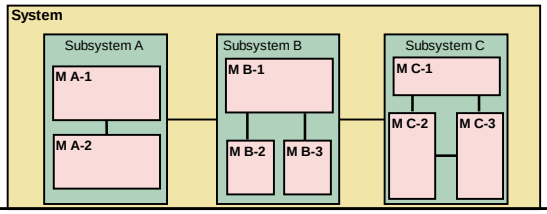


(Sommerville, 2015)

Arkitekturdesign

Nedbrytning av systemets övergripande struktur

- System - helheten
  - Subsystem – enhet som ej beror på andra subsystem
    - Moduler – enhet som verkar ihop med andra moduler
      - (Komponenter – en eller flera klasser)



Syfte med arkitekturdesignen

- Länk mellan kraven och detaljerad design
  - Grov ritning för implementation
- Kommunicerar designbeslut i organisationen
- Grund för systemanalys
  - Säkerhet
  - Prestanda
- Underlättar återanvändning
  - Använda delar i andra system
  - Utveckla produktlinjer

Val av arkitekturdesign

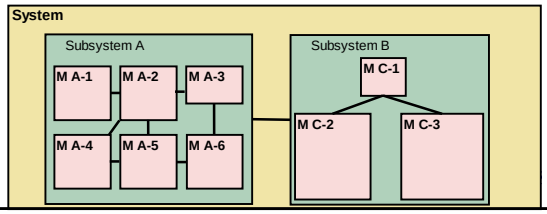
- Förståelse för kontext och intressenter nödvändig för bra beslut
- Kvalitetskraven avgör ofta beslutet
  - Vad vill vi uppnå? Motsättningar vanligt!

Övriga faktorer som kan avgöra

- Organisationens tekniska kompetens och erfarenhet
- Återanvändning av tidigare arkitektur
- Standarder som behöver uppfyllas

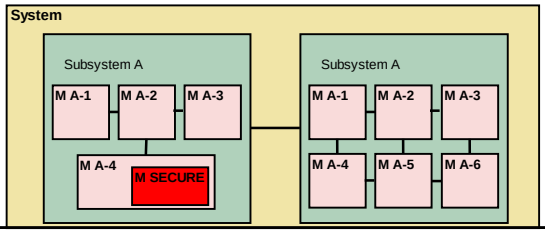
Prestanda?

- Kommunikation är en prestandatjuv!
- Samla tunga beräkningar i moduler som kommunicerar minimalt utåt
- Acceptera att beräkningsmoduler blir stora



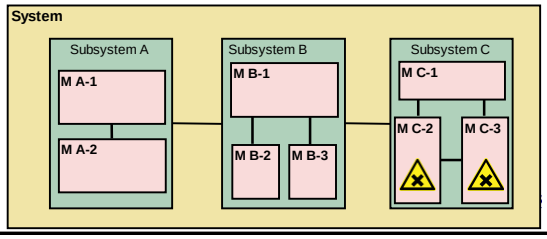
Säkerhet? (security)

- Åtkomstbegränsning viktigt
- Introducera säkerhet i olika lager
- Hantera den känsligaste informationen innerst



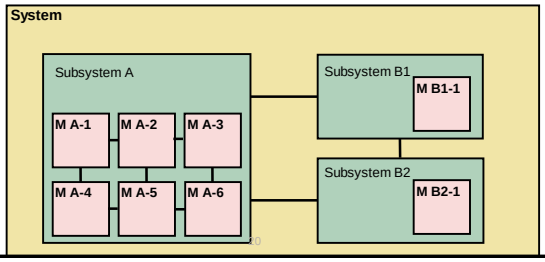
Säkerhet? (safety)

- Att verifiera säkerhetskrav är svårt och dyrt
- Samla alla säkerhetskritiska operationer i separat subsystem



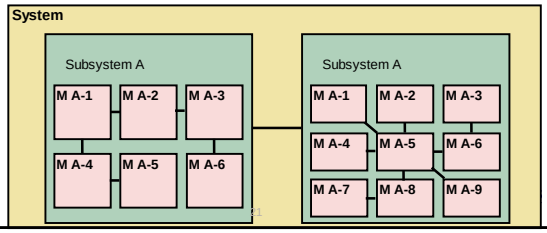
Tillgänglighet?

- Minimera risken att systemet är otillgängligt
- Introducera redundans

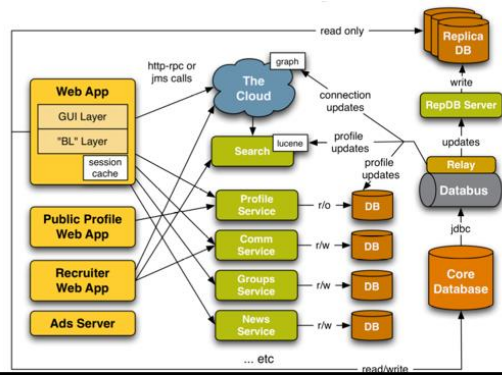


Enkelt underhåll/evolution?

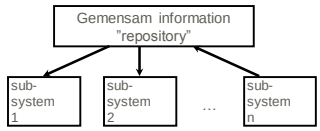
- Fokusera på små oberoende moduler
- Minimal kommunikation gör det lättare att ersätta moduler i framtiden



LinkedIn: Java-arkitektur



Typexempel – Repository (delad data)



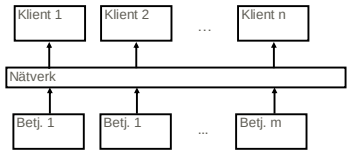
Fördelar:

- Effektivt med mycket data
- Data-producent måste inte veta så mycket om konsument
- Operationer på all data underlättas, t.ex. backup

Svagheter:

- Alla subsystem måste använda samma dataformat
- Vidareutveckling kan vara svårt eftersom mycket bygger på en viss datamodell

Typexempel - Client-server



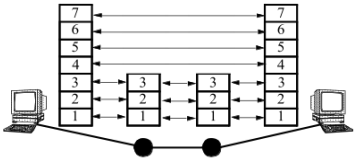
Fördelar:

- Distribuerad arkitektur
- Lätt att lägga till nya klienter och betjänare

Svagheter:

- Uppdatering av klient eller betjänare kan kräva uppdatering av samtliga
- Ingen gemensam datamodell

Typexempel – Abstract machine



Varje lager utgör en ”abstrakt maskin” som används av nästa lager

Fördelar:
- Stöd för inkrementell utveckling
- Underlättar portabilitet

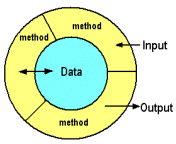
Svagheter:
- Kan uppstå beroenden mellan flera lager
- Kan bli sämre prestanda

Objektorienterad design

- Implementationsnära design
- Beskrivning av hur komponenter implementeras på klassnivå
- Klasser beskriver meningsfulla entiteter i problemdomänen
  - Substantiv i beskrivningen blir klasser
  - Operationer implementeras i metoder

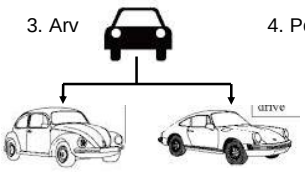
Objektorientering - Fundamentala koncept

1. Inkapsling

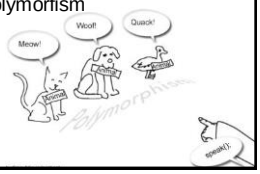


2. Abstraktion

3. Arv



4. Polymorfism

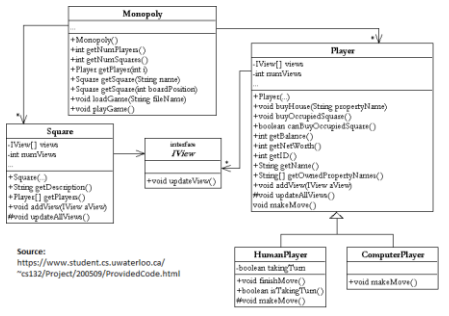


Unified Modeling Language - UML

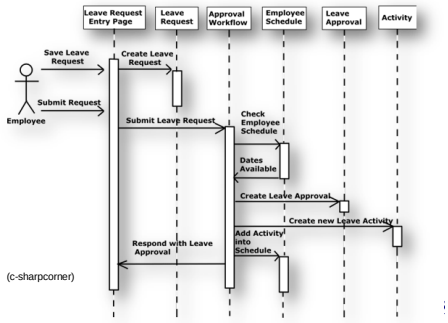
- Generellt språk för att modellera programvarusystem
  - Standardiserat av ISO
  - Tre skapare, varav en svensk: Ivar Jacobson
- Statisk modell av systemet
  - Klassdiagram
- Dynamisk modell av systemet
  - Sekvensdiagram

Mer i kursen ”Objektorienterad design och modellering” eller ”Objektorienterad design och diskreta strukturer”

Klassdiagram

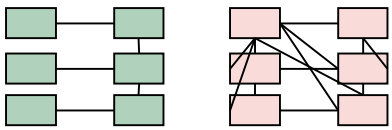


Sekvensdiagram



Designmål: Låg koppling (coupling)

- I hur stor utsträckning är klasser i programmet kopplade till varandra?
- Låg koppling underlättar underhåll och evolution



Designmål: Hög samhörighet (cohesion)

Hur väl innehållet i en klass ”hänger samman”

Exempel:

- **Logisk** – t.ex. en klass som sköter all utmatning av data
- **Temporal** – t.ex. en klass som sköter all ”uppstart” eller ”avslut”
- **Procedurbaserad** – aktiviteter som utförs efter varandra slås ihop
- **Kommunikationsbaserad** – delar som behandlar samma data slås ihop
- **Sekventiell** – kedja av aktiviteter som hänger ihop i sekvens
- **Funktionell** – innehållet står för en enstaka funktion, t ex ”sortera vektor”

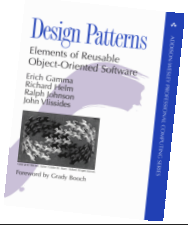
Designmönster

Beprövade lösningar på återkommande problem

- Ursprungligen från arkitektur
- Inom programvara används objektorienterade koncept

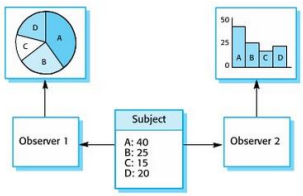
Några exempel:

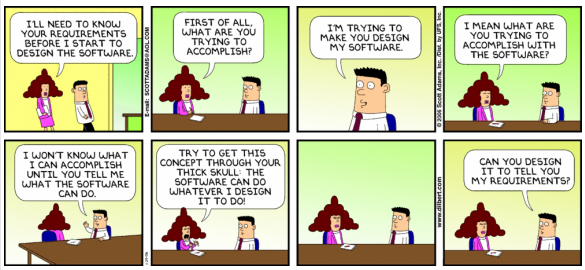
- **Singleton** garanterar ett unikt objekt av en klass
- **Iterator** traverserar en samling objekt
- **Visitor** gör en operation på samtliga objekt i en samling



Exempelmönster: Observer

- Separera ett objekts tillstånd från presentation
- Möjliggör flera olika vyer





Design av användargränssnitt

Textuellt gränssnitt

- Kraftfullt för expertanvändare

Grafiska användargränssnitt

- Lättare att lära sig och använda

Några grundläggande designprinciper

Igenkänning
- Använd välkänd terminologi från domänen
- Följ beprövade koncept (fönster, flikar, dialogfönster etc.)

Konsistens i GUIt
- Använd samma grafiska komponenter överallt
- Erbjud samma kortkommandon överallt

Inga överraskningar
- Användaren ska kunna förutspå vad som händer

Återhämtning
- Användare gör fel: tillåt återgång till föregående tillstånd

Guida användaren
- Hjälp användaren och presentera feedback

I projektet: Designdokumentet

Ingen mall finns
- Hitta ett format som funkar

Alla klasser ska beskrivas
- Namn, konstruktorer, publika metoder, ärvning
- Kommentarer
- Ange även ansvarig för klasser/metoder

Rita klassdiagram
- Behöver inte vara UML
- Relationer och multiplicitet
- Alla GUI-klasser behöver ej visas (en ”GUI”-box räcker)

Design - sammanfattning

- Design är både en aktivitet och ett resultat
- Arkitekturdesign är en övergripande nedbrytning av systemstruktur: System → Subsystem → Moduler
  - Val av arkitektur beror på kvalitetskraven
  - Exempel: repository, client-server, abstract machine
- Objektorienterad design beskriver hur komponenter implementeras av klasser
  - Beskrivs vanligtvis med UML
  - Sträva efter låg koppling och hög sammanhållning
- Utveckla GUI som matchar användarens mentala modell

Från F3

Enhetstest

- Test av minsta testbara komponent
  - Ofta klass eller metod

Från F3

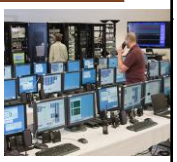
Integrationstest

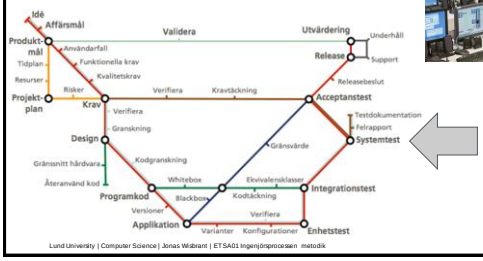
- Testfall bestäms t ex baserat på specifikationer av gränssnitt i design
  - Test av subsystem

Från F3

Systemtest

- Testar det fullständiga systemet
- Är kravspecifikationen uppfylld?





Från F3

Acceptanstest

- Test för att säkerställa att utlovat system har utvecklas
  - Kan utföras av beställaren

Från F3

Black-box vs. White-box

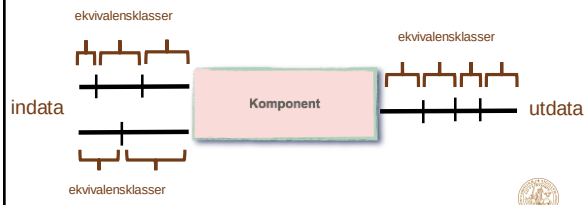
Black-box
- Programmet ses som en ”svart låda” och man utnyttjar inte någon kunskap om koden i samband med definition av testfall
- Kravspecifikationen används för att ta fram testfall
- Testar utfall/resultat

White-box
- Kräver tillgång till koden
- Testar utfall och inre funktion
  - täcker vi koden?
  - täcker vi vägarna?

Från F3

Ekvivalenspartitionering

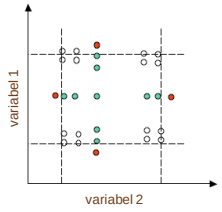
Hitta värden för in- och utdata som behandlas på inbördes enhetligt sätt



Från F3

Gränsvärdestestning

Vanliga gränsvärden	(gröna)
Robust-test	(röda)
Kombinationer av gränsfall	(vita)



Från F3

Parvis testning

Vissa fel ”single mode”
- T ex lägga in viss typ av kurs

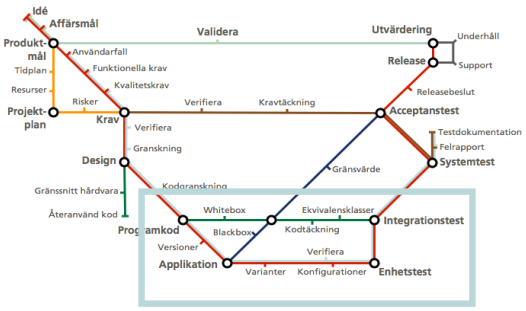
Andra fel uppstår som kombination av två parametrar:
- T ex lägga in viss typ av kurs för viss institution
- Parvis testning: täck alla möjliga kombinationer av värden för alla möjliga par av parametrar

Eller ännu fler parametrar
- T ex lägga in viss typ av kurs för viss institution för visst år och viss studentgrupp

Antag ett system för att hantera kurser och studenter, med följande parametrar:

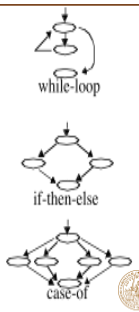
- Kurstyp (G1, G2, A)
- Institution (CS, EIT, Math,…)
- År (2006, 2007, 2008, 2015)
- Studentgrupp (C, D, E,…)

White-box testing



White-box testing

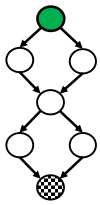
- Kräver tillgång till koden
- Tanken är att testfallen ska täcka all kod
  - Men vad menas med ”all kod”?
- Betrakta kodens kontrollflödesgraf



Täcka rader (statement coverage)

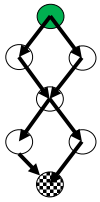
Exekvera alla rader minst en gång

- Alla noder i kontrollflödesgrafen



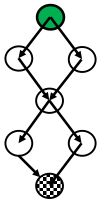
Täcka grenar (branch coverage)

Exekvera alla grenar minst en gång
- Alla bågar i kontrollflödesgrafen
- Innefattar även komplett radtäckning

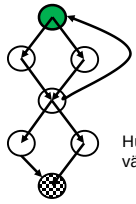


Täcka vägar (path coverage)

Exekvera samtliga vägar från startnod till slutnod
- Innefattar även komplett grentäckning
- Antalet testfall exploderar med loopar!



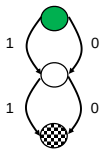
4 vägar



Hur många vägar?

Täcka enkla vägar (simple path coverage)

Exekvera samtliga linjärt oberoende vägar från startnod till slutnod
- Innefattar komplett grentäckning
- Hanterar problematiken med loopar
- Ofta en rimlig kompromiss



Vägar:
(1,1)
(0,0)
(1,0)
(0,1)

Ej linjärt oberoende!

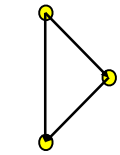
McCabe's Cyclomatic Complexity (CC)

Används för att beräkna antalet enkla vägar i en kontrollflödesgraf

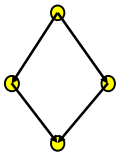
Antal linjärt oberoende vägar:
CC = #bågar - #noder + 2

Krav
- En enda komponent i grafen
- En unik startnod samt en unik slutnod

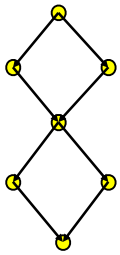
Exempel (CC = #bågar - #noder + 2)



If:
CC = 3-3+2 = 2



If-else:
CC = 4-4+2 = 2

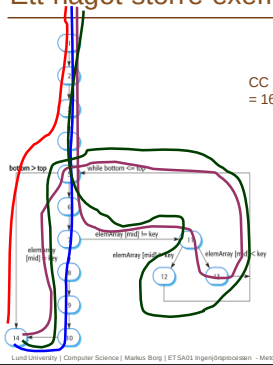


3 linjärt oberoende vägar

2 linjärt oberoende vägar

CC = 8-7+2 = 3

Ett något större exempel



CC = #bågar - #noder +2 =
= 16-14+2 = 4

CC används också som ett mått på komplexitet

Praktisk testning: Demo av testverktyg

1. Motivering av parvis testning
   - Systemtestning av en diskmaskin
2. Praktiskt exempel
   - Kvalitetssäkring av personnummerklass
     1. Webbtjänst Hexawise: Optimala testfall för parvis testning
     2. Optimala testfall → testfall i JUnit
     3. Mäta kodtäckning med CodeCover i Eclipse

Parvis testning för systemtest av diskmaskin

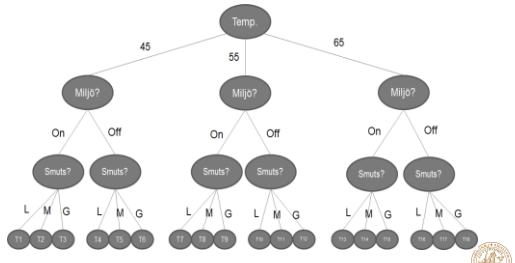
Testparametrar:

Temperatur	(45, 55, 65)
Miljöläge	(on, off)
Nedsmutsningsgrad	(lätt, måttlig, grov)

Utfall:

Diskresultat	(rent, smutsigt)

Parvis testning för systemtest av diskmaskin



→ **3 x 2 x 3 = 18 möjliga testfall**

Parvis testning

Samtliga kombinationer av parameterpar testas
- En black box-teknik

Färre testfall än vad som krävs för uttömmande testning
- Men en rimlig nivå för att hitta defekter

Generera testdata som uppnår parvis täckning med ett minimalt antal testfall är svårt
- ett kombinatoriskt optimeringsproblem

Systemtest diskmaskin: Möjliga par

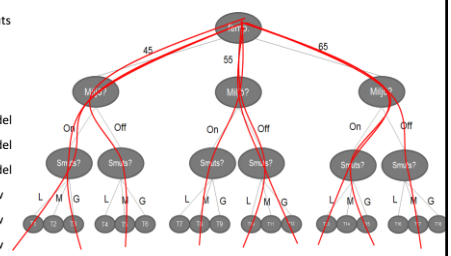
Temperatur-Miljöläge (3 x 2 komb.)
- 45-on, 45-off
- 55-on, 55-off
- 65-on, 65-off

Temperatur-Nedsmutsningsgrad (3 x 3 komb.)
- …

Miljöläge-Nedsmutsningsgrad (2 x 3 komb.)
- …

Parvis testning för systemtest av diskmaskin

	Temp.	Miljö.	Smuts
T1	45	On	Lätt
T2	55	Off	Lätt
T3	65	On	Lätt
T4	45	Off	Medel
T5	55	On	Medel
T6	65	Off	Medel
T7	45	On	Grov
T8	55	Off	Grov
T9	65	On	Grov



→ **9 testfall räcker för att testa alla parameterpar**

Verifiera extern modul för personnummer

Nytt krav på cykelgaraget!
- Beställaren vill att ha stöd för personnummer
- Del av betalningsmodellen

Projektet bestämmer sig för att integrera funktionalitet hittad i öppen källkod
- Kvalitetssäkring av den externa koden krävs

Hur gör man?

Verifieringsapproach

Skapande av testfall genom black box-tekniker
- Ekvivalenspartitionering
- Parvis testning

White box-testning för att avgöra testkvalitet
- Kodtäckningstestning

Personnummerkoll för användare

Personnummer anges enligt format:
- **XXXXXX-XXXX eller XXXXXX+XXXX** (standard: + betyder > 100 år)
- **XXXXXXXX-XXXX (long format)**
- **XXXXXXXXXX (short format)**
- **Sista siffran är en kontrollsiffra (Luhn-algoritmen)**
- **Istället för – accepteras även /**
- **Personer klassificeras i en ålderskategori**
  - **<10, 10-18, 18-65, 66-99, >99**

Vi identifierar följande ekvivalensklasser:

User category	(child, teen, adult, senior, golden)
ID format	(standard, short, long)
Separator	(-, /)
Only digits	(true, false)
Input date	(valid, future date, invalid month, invalid day)
Checksum	(correct, incorrect)

Personnummerkoll för användare

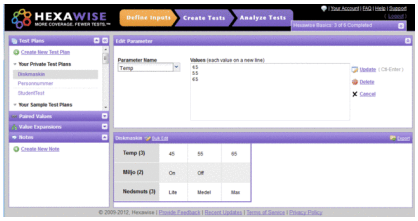
Vi identifierar följande ekvivalensklasser:

User category	(child, teen, adult, senior, golden)
ID format	(standard, short, long)
Separator	(-, /)
Only digits	(true, false)
Input date	(valid, future date, invalid month, invalid day)
Checksum	(correct, incorrect)

Vi kan inte testa 5 x 3 x 2 x 2 x 4 x 2 = 480 kombinationer

Hexawise

Verktyg för att generera testfall för parvis täckning



20 testfall täcker alla exvivalensklasspar

	Category	Format	Sep	Only digits	Birth date	Checksum
Test 1	child	standard	-	TRUE	correct	correct
Test 2	child	short	/	FALSE	future date	incorrect
Test 3	child	long	/	TRUE	invalid month	correct
Test 4	child	standard	-	FALSE	invalid day	incorrect
Test 5	teen	short	-	TRUE	correct	incorrect
Test 6	teen	standard	/	TRUE	future date	correct
Test 7	teen	standard	-	FALSE	invalid month	correct
Test 8	teen	long	-	TRUE	invalid day	correct
Test 9	adult	long	/	FALSE	correct	incorrect
Test 10	adult	standard	-	TRUE	future date	correct
Test 11	adult	short	/	FALSE	invalid month	correct
Test 12	adult	short	/	FALSE	invalid day	incorrect
Test 13	senior	standard	-	TRUE	correct	correct
Test 14	senior	long	/	FALSE	future date	incorrect
Test 15	senior	short	-	FALSE	invalid month	incorrect

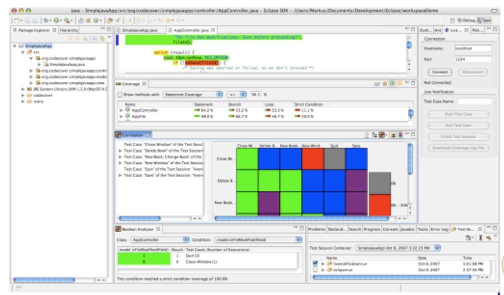
Formulera testfallen i JUnit

```
public class TestIDNumber {

    /*
     * TEST PARAMETERS:
     * User category: (child, teen, adult, senior, golden)
     * Format: (standard, short, long)
     * Separator: (-, /)
     * Only digits: (true, false)
     * Valid birth date: (valid, future date, invalid month, invalid date)
     * Correct checksum: (correct, incorrect)
     */

    /***
     * Test 1: (child, standard format, -, only digits, valid birth date, correct checksum)
     */
    @Test
    public void test1_Child_Standard() {
        IDNumber id = new IDNumber("051124-8833");
        assertEquals(IDNumber.USER_CHILD, id.getUserCategory());
    }

    /***
     * Test 2: (child, short format, /, only digits, future birth date, incorrect checksum)
     */
    @Test
    public void test2_Child_Short_FutureBirth_Checksum() {
        IDNumber id = new IDNumber("151124@137");
        assertTrue(id.getUserCategory() < 0);
        /*assertTrue(id.getUserCategory() == IDNumber.ERR_BORN_FUTURE ||
                   id.getUserCategory() == IDNumber.ERR_WRONG_CHECKSUM);*/
    }

    /***
     * Test 3: (child, long format, /, only digits, invalid month, correct checksum)
     */
```

	Category	Format	Sep	Only digits	Birth date	Checksum
Test 1	child	standard	-	TRUE	correct	correct
Test 2	child	short	/	FALSE	future date	incorrect
Test 3	child	long	/	TRUE	invali	
Test 4	child	standard	-	FALSE	invali	
Test 5	teen	short	-	TRUE	corre	
Test 6	teen	standard	-	TRUE	future	
Test 7	teen	standard	/	FALSE	invali	
Test 8	teen	long	-	TRUE	invali	
Test 9	adult	long	-	FALSE	corre	
Test 10	adult	standard	-	TRUE	future	
Test 11	adult	short	/	FALSE	invalid month	correct
Test 12	adult	short	/	FALSE	invalid day	incorrect
Test 13	senior	standard	-	TRUE	correct	correct
Test 14	senior	long	/	FALSE	future date	incorrect
Test 15	senior	short	-	FALSE	invalid month	incorrect
Test 16	senior	short	-	TRUE	invalid day	incorrect

Några av förslagen är omöjliga – man måste hitta dem själv =(

Mäta kodtäckning med CodeCover



Summering: Parvis testning och kodtäckning

1. Generera testfall för parvis testning
2. Implementera de möjliga testfallen i JUnit
3. Exekvera testfallen med CodeCover

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
ETSA01_Coverage	83,0 %	73,8 %	?	73,1 %	?	?
IDTest	83,0 %	73,8 %	?	73,1 %	?	?
IDNumber	83,0 %	73,8 %	?	73,1 %	?	?
IDNumber	100,0 %	–	?	–	?	?
assignUserCategory	80,0 %	71,4 %	?	66,7 %	?	?
calculateUserCategory	55,6 %	64,3 %	?	62,5 %	?	?
checkDateFormat	62,5 %	64,3 %	?	57,1 %	?	?
checkDigits	100,0 %	100,0 %	?	100,0 %	?	?
checkDigits	100,0 %	100,0 %	?	100,0 %	?	?
getUserCategory	100,0 %	–	?	–	?	?
registerDateOfBirth	90,0 %	83,3 %	?	83,3 %	?	?
verifyCheckSum	90,9 %	87,5 %	?	91,7 %	?	?

Kodtäckning 83% - vad innebär det?

Testning är dyrt!

Testning ofta den enskilt dyraste aktiviteten i ett utvecklingsprojekt

$$$

När är testningen färdig?

Finns inga garantier att alla defekter är hittade!

- Man kan alltid testa mer…

Avbryta för sent

Avbryta för tidigt

- Slösade resurser
- Försenad release
- Ökade kostnader

- Defekter finns kvar
- Missnöjda användare
- Dyr support
- Dyrt underhåll

Snabbguide på hemsidan