SWT\_Eksamensforberedelse

# GeneralTesting

## BlackboxTesting

* Ved Blackbox testing ved man ikke hvad funktioner indeholder eller andre properties. Man kan kun se den klasse man er i gang med at teste's interface.
* Alle acts og assertions virker ind på interfacet
* Hvis klassen's interface er svær at teste på så skal man nok hav ændret lidt i sin klasse.

## WhiteboxTesting

* Ved whitebox testing kan man vælge om act går på klassen selv eller dens interface
* Det samme gælder for assertions
* Whitebox test gør test mega skrøblige
* Man tænker sig ikke så meget om når man laver test og kommer derfor tit til at lave whitebox test
* Alt afhængig af klassens interface er det måske ikke interessant at teste vise dele af klassen. Man skal derfor tænke på hvad klassen skal kunne og ikke hvad den gør.
* Hvis ens test knækker efter refaktorering af sin kode er det fordi man tester sin implementering og ikke sit interface.

## Test Types and Fake Types

* Fakes er navnet for afhægigheder. Her findes 2 typer Mocks og Stubs.
* Hvis en UUT har brug for afhængigheder, så er det der at man laver en fake
* Identify, Interfaces, Inject - de 3 I'er
* Man skal huske at lave et interface til hver klasse så man kan lave en ny exact klasse som er en fake

* Test Types:
* Value based tests.
* State based tests.
* State based test hvis man ikke giver et fuxk om tidligere test. Eks. tal i en liste
* Value based test er når man bruger tal der ligger i en liste og er afhængig af dem.

* I state based test er alle fakes en stub
* En fake der retunerer noget er en stub
* The assertion is always on the UUT and never on the stub

* For interaction based test så bruges der mocks (og måske nogle stubs)
* A mock exist to "record" that the interaction took place, because you want to test if the UUT had expected interactions with the dependencies
* Mocks kan blive FUXKING! COMPLEX! Skriv dem ikke selv nedefra! BRUG Nsubstitute
* **REMEMBER!** Man tester på mocks i interaction based test

* Smarte Isolationframeworks som Nsubstitute er noice fordi den laver mocks eller stubs alt afhængig af hvad du har brug for ud fra interface.

* Events går ikke under mocks eller stub, men er en helt tredje("No idea what the name is" - Frank 2019)

## Unittest/Unittest af Events

* En unit test indeholder 3 dele Arange-Act-Assert
* Arange er opstillingen af din test, instantieringer etc.
* Act er kaldet på den/de funktioner der skal testes på
* Assert er selve din assert sætning, som tester et udsagn er rigtigt eller forkert.
* Det er ikke altid nødvendigt at teste hver en funktion i en klasse.
* Man skal til tider være kritisk overfor om det giver mening at teste en funktion/ specifik del af en klasse.
* Det er helveds vigtigt at være kritisk overfor sin kode og optimerer den så godt som muligt således at testing bliver nemt
* INTERFACES SKAL ALTID MED! EN TIL HVER KLASSE HVIS DU HAR TANKER OM AT TESTE SENERE
* God regel er 3 testcases for hver funktion

**Events:**

* *interface*-*implementeret klasse*- *klasse der vil reagere på event*(The Trio)

* Events er en implementation af observer pattern

* Vigtigt at der laves en event klasse med alt det data man har brug for
* Herefter et interface til den klasse som skal bruge eventet med at initialiserer en event funktion med den først benævnte eventklasse
* Til sidst laves den klasse som hører til interfacet. Det er i denne klasse vigtig at lave en virtual funktion til at kunne invoke eventet.
* **Det kan ses i caffeine eksemplet**

**Hvordan bruges det så?:**

* En eventhandler skal helst være private så man ikke kommer til at lave whitebox testing.
* Herudover skal man lave en funktion som som subscriber på eventet.

**Hvordan testes det?:**

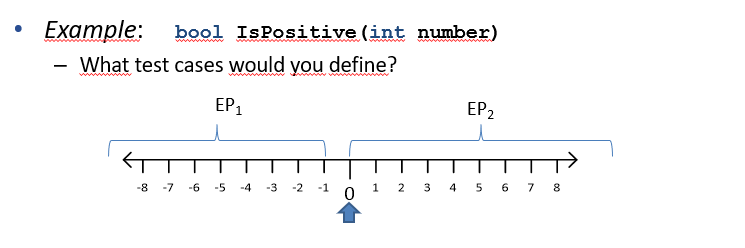
* The source - the class which have the event property
* The receiver - the class that connects to the property
* I testen skal der i setup funktionen laves en fake subscriber med en anonym lambdafunktion.
* Sikrer dig i setup at eventet er sat til null is starten så man ikke ligepludselig ender i en grim situation.
* Når man kalder en funktion der invoker et event gemmes dataene i eventet og man kan kalde funktionen og teste efter om eventet indeholder data, ved at kigge efter om eventet ikke er null. Men man kan også se på dataen inde i eventet med dot oberatoret.

* Når receiver skal testes så fakes source ved at lave den om til en substitute.
* Din substitute skal så bruges i stedet for en instantiering af din source.
* Når man skal raise event med en sub skal man bruge en funktion der hedder raise.eventwith.

# TEST QUALITY

## ZOMBIE-Analysis

* ZOMBIE er et akronym
* **Z**ero-**O**ne-**M**any-**B**oundary Value Analysis-**I**nterfaces-**E**xceptional Behavior
* Z - input or zero output or zero actions:
  + Test on the correct state of a newly created object – ZERO calls
  + Test with the empty input for collection – ZERO elements
  + Make tests that should return the empty collection – ZERO elements
* O - One input or One output or One action:
  + Test on the correct state of a newly created object after one call of each method
  + Test input with a collection with: – ONE element
  + Make tests that should return a collection with exactly: – ONE element
* M - inputs or Many outputs or Many actions
  + Test on the correct state of an object after several calls of a each method
  + Test on the correct state of an object after calls of several mixed methods
  + Test input with a collection with: – 2 or more elements
  + Make tests that should return a collection with: – 2 or more elements
* B – Boundaries
  + Use BVA (Boundary Value Analysis) for selecting the correct parameters for tests – both valid and invalid.
  + Eks. Vi har en grænse på mellem 0 og 100. Vi tester 0 og 100, 1 og 99, -1 og 101, -100 og 200.
  + Use **EP (Equivalence Partitions)** to keep this to a workable level



* + Del op som vist på billedet og derefter tag få test fra hver del. AKA lad være med at være dum og teste alle mulige kombinationer.
  + Make tests that investigate the specified or implicit boundaries of sizes – as far as practical
  + BVA and EPs are black box test tools – we only consider input and expected output
  + but uses general knowledge about how programs are built
  + EPs reduce the amount of tests through analysis – not too many!
  + BVA helps to select those tests in a way that makes it more probable that errors are found – not too few
* I – Interfaces
  + Test all methods
  + Test all overloaded versions of methods
  + Test all thrown exceptions
  + Use Coverage to be sure
  + Exercise all called interfaces (Interaction-based testing and Integration testing) to dependencies
  + Test all events used from dependencies
  + Test all events UUT provides
* E – Exceptional behavior
  + This is not just Exceptions as defined in the specifications
  + This is about robustness
  + What happens on faulty
    - Input?
      * (BVA)
    - Call sequence?
    - Dependencies (timeout, unexpected result, exception, …)?
  + IN SHORT! TEST EXCEPTIONS TO MAKE PROGRAM ROBUST!

## Coverage

**General:**

* The process of determining which areas of a program that are exercised by given set of test cases.
* Using that knowledge to systematically expand test cases to test “untouched” parts of the program.
* Having a quantitative measure of code coverage
* Many different kinds of coverage exist
* We want a measure of how “good” our test is.
  + One measure: How much application code is “covered” by tests?
* *Test coverage* measure quality of *test (*not of the actual product)
* We want to cover our own ass from an angry customer, by covering the whole software with test.
* %
* Remember to check if-statements, both the case where it’s false and the case where it’s true

**Ups about coverage:**

* Simple, objective(Repeatable)measure for quality of test
* Helps to decide where to spend time testing
* Helps to detect trends (increasing or decreasing coverage in specific parts of the code) and hence take appropriate action

**Downs about coverage:**

* 100% coverage is no guarantee of zero defects
* Very tool dependent (expensive, non-portable)
* Test coverage says nothing of *omission errors*
* Test coverage works best for new projects where it can be applied from the start

**How to automate:**

* We use dotCover to do line coverage calculations
* dotCover must run an application
* Running coverage calculation takes time because extra code is executed in the UUT
* Ideal for Continuous Integration
  + on each push
  + nightly