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?:**

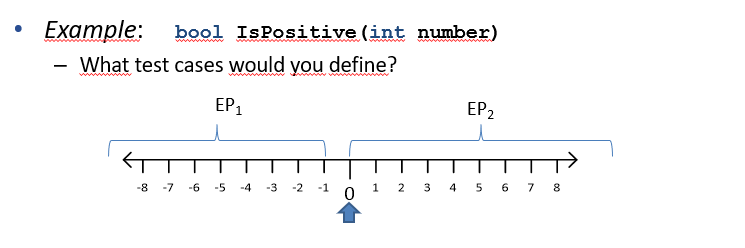
* The source - the class which have the event property
* The receiver - the class that connects to the property
* I testen skal der i setup funktionen laves en fake subscriber med en anonym lambdafunktion.
* Sikrer dig i setup at eventet er sat til null is starten så man ikke ligepludselig ender i en 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

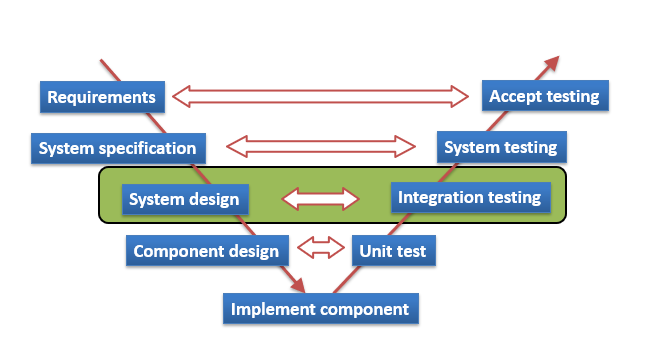
# OtherTypesOfTesting

## Accepttest

Meeeeh. Specflow, har ikke tænkt mig at sætte mig ind i det. De virker ikke som det skal. Selv Frank kunne ikke få det itl at virke.

## Integrationstest

* Når der arbejdes med Integrationstest skal udviklingen struktureres efter V-modellen



* En god idé at lave integrationstest for hver iteration i udviklingen af et projekt
* The purpose of integration test is to test the **interactions** and **interfaces** **between** several modules
* The aim is to verify correct interaction of the tested modules
  + Classes
  + Packages
  + Components
  + Subsystems
* Additionally – the interaction **between** the **low level modules** (HW drivers) and the **actual hardware:** that is **hardware-software integration**
* Verification requires 100% *interface coverage* – is hard to measure and can be hard to obtain
* Integration tests requires knowledge of system architecture to partition the system into testable chunks

What to know before integrationtesting?:

* Unit testing of all modules is complete
* System architecture (dependencies) is known
* Integration test plan is defined
  + Integrated modules
  + System Under Test (SUT) structure?
  + Test fixtures / environment
  + Test cases – SUT stimuli and expected responses

### Dependencytree:

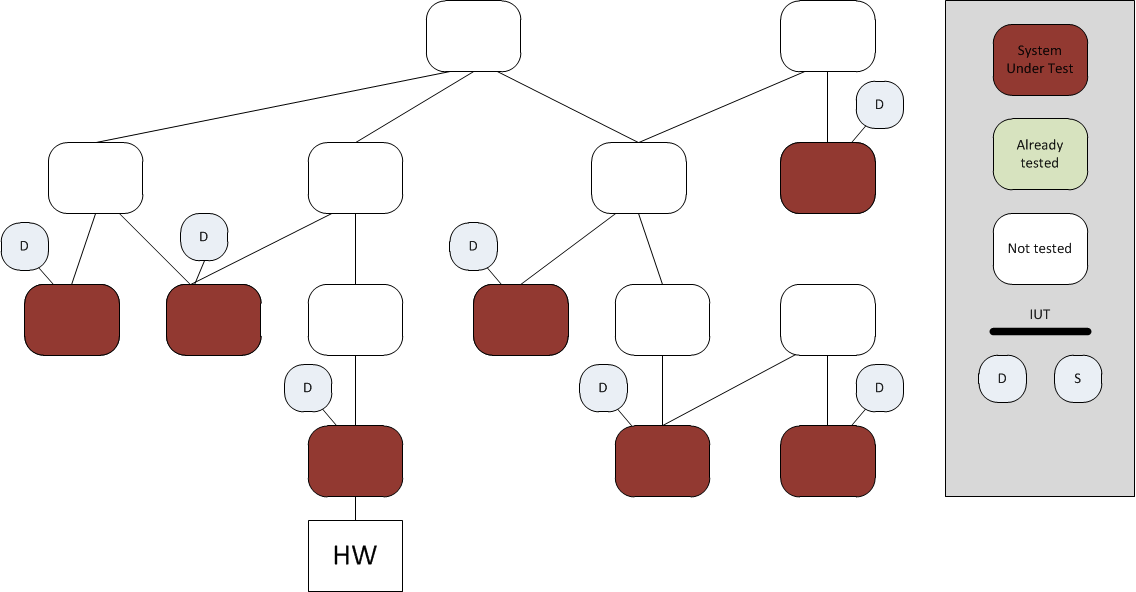
* Et træ hvor det øverste element afhænger af alle nedenstående elementer. Nederste element har ingen afhængigheder.
* Ingen horisontale streger
* Klassediagrammer, sekvensdiagrammer og andre slags diagrammer bruges til at afgører hvilke dele af et system der afhænger af hinanden.
* Loops i et program kan blive erstattet med stubs

Hvilke typer dependencytrees findes?:

* Big Bang Integration



* + Downs ved Bigbang:
    - ”Fire it up, see it fail”
    - Only possible late in developmenterrors costly to fix
    - Works (sometimes) for small, low-complexity, stable, systems
    - Very low probability of detecting errors
    - Very little feedback
  + Middles ved Bigbang:
    - Works (sometimes) for small, low-complexity, stable, systems
* Bottom-up Integration



* + Downs ved Bigbang
    - Requires many drivers at different levels
    - Postpones test of critical control com-ponent interfaces
  + Middles ved Bigbang
    - Reflects very ”engineering-like” mindset
  + Ups ved Bigbang
    - No (few) stubs to develop
    - Easy to cover interfaces at all levels
* Top-down



* + Downs ved Top-Down
    - Hard to exercise low-level interfaces from the top
  + Middles ved Top-Down
    - Needs lots of stubs (OK with isolation framework)
  + Ups ved Top-Down
    - Early feedback on controller compo-nents
    - Facilitates concurrent HW and SW development
* Collaboration Integration



* + Downs ved Collaboration
    - Hard to exercise low-level interfaces
    - Participants not exercised separa-tely – mini Big-Bang!
  + Middles ved Collaboration
    - Needs lots of stubs (OK with isolation framework)
  + Ups ved Collaboration
    - Intuitive for users (may follow use cases)
    - Especially useful for higher-level system tests (component, subsystem)
    - Models iterative development with UCs as unit
* Sandwich Integration



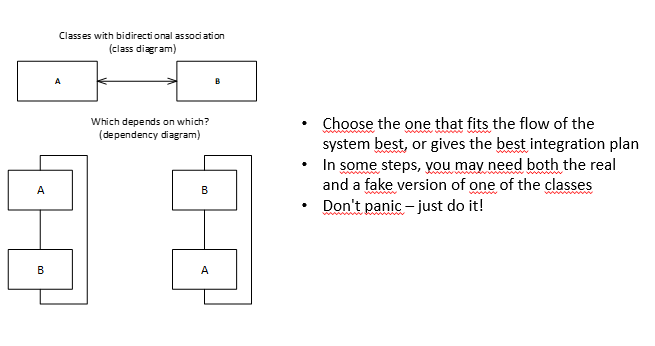


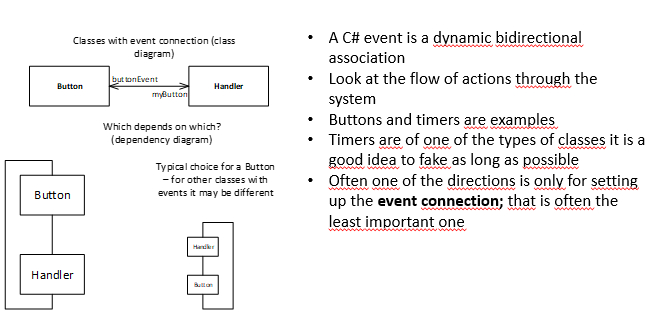
* + Middles ved Sandwhich
    - Takes lots of planning
  + Ups ved Sandwhich
    - The best of top down and bottom up models
    - Many of the disadvantages of TD and BU are alleviated

God skik:

* Automate, automate, automate
  + Higher complexity 🡪 lower probability
  + Use High Frequency integration: Nightly CI builds for each test scenario to check integration (common code repo)
  + Use Unit Test and isolation frameworks as much as possible
* Easier to code stubs than drivers
* Use a little bit of all patterns (except Big Bang)

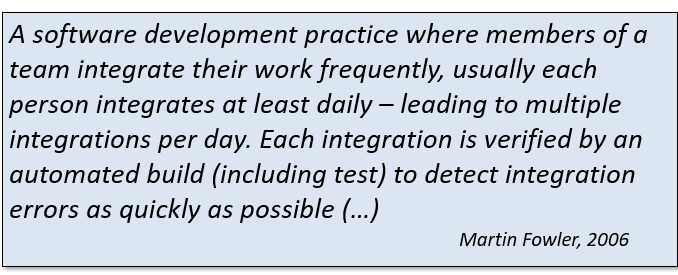
Hvordan fikser man bidirectional dependencytree?:





# Continues Integration

Hvad er det?:



Benefits?:

* No ”integration hell” at the ”end” of a project
* Never more than hours from working (deployable) build
* Rapid feedback
* Several graded builds (continuous, nightly, …)
* Metrics for code and test quality never out of date
* Certified and controlled runtime environment

Vigtigste af alt er at du sætter et jenkins job op som giver dig en fejl når du integrer en ny del af dit system ind på din main branch. Idéen er at dine unittest eller integrations test kan fejle hvis den nye del ikke passer sammen og af den grund er det godt at have sat et Jenkins job på til at udfører Continues Integration.

Sig det så skal underviseren nok blive ophidset til eksamen 😉

# Code Metrics

## General

 **Maintainability Index** - Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. Color coded ratings can be used to quickly identify trouble spots in your code. A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 and indicates low maintainability. For more information, see the [Maintainability index range and meaning](https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/) blog post.

 **Cyclomatic Complexity** - Measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program. A program that has complex control flow requires more tests to achieve good code coverage and is less maintainable. For more information, see the [Wikipedia entry for cyclomatic complexity](https://wikipedia.org/wiki/Cyclomatic_complexity).

 **Depth of Inheritance** - Indicates the number of different classes that inherit from one another, all the way back to the base class. Depth of Inheritance is similar to class coupling in that a change in a base class can affect any of its inherited classes. The higher this number, the deeper the inheritance and the higher the potential for base class modifications to result in a breaking change. For Depth of Inheritance, a low value is good and a high value is bad.

 **Class Coupling** - Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration. Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types. For more information, see the [Class coupling](https://blogs.msdn.microsoft.com/zainnab/2011/05/25/code-metrics-class-coupling/) blog post.

 **Lines of Code** - Indicates the approximate number of lines in the code. The count is based on the IL code and is therefore not the exact number of lines in the source code file. A high count might indicate that a type or method is trying to do too much work and should be split up. It might also indicate that the type or method might be hard to maintain.

Source: <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2017>

## Static analysis

Hvad kan man finde ud af om sit program, UDEN at køre det? Denne disciplin kaldes Static Analysis, og kan give feedback til programmøren, der er ligeså værdifuld som en faktisk test. Vi gennemgår en række metoder og områder, som Static Analysis kan dække.

HER MÅ DU SELV LIGE LÆSE I SLIDESNE #FatterHat

## Dynamic analysis

Hvad kan man finde ud af om sit program, NÅR man kører det? Denne disciplin kaldes Profiling eller Dynamic Analysis, og giver feedback til programmøren, når programmet kører, fx. i lang tid. Der kommer en række metoder og værktøjer til Dynamic Analysis.

HER MÅ DU SELV LIGE LÆSE I SLIDESNE #FatterHat