1 SOFTWARE QUALITY METRICS

Indledning: Sætte tal på ens software, hvor godt er det skrevet og svært er det at vedligeholde

Kvantitativ: Måler tal på softwaren
Kvalitative: Giver kvaliteten af softwaren

	Quantitative	Qualitative
Static	Complexity metrics Cyclomatic Maintainability ...	Code style and standards Deducible errors or warnings Pointer errors Memory leaks Uninitialized variables
Dynamic	Profiling resource usage Time/CPU/Memory/OS Leaks Memory OS resources Client/Server Bandwidth Scalability Bottlenecks Memory/CPU/ bandwidth	Testing! Indexing errors Uninitialized variables NULL pointers Double delete Uncaught exceptions Crash analysis Stack trace Memory dump Logging/Tracing

Statisk analyse: Hvad kan man finde ud af om sit program uden at køre det

- Kvantitative
  - Maintainability
    - Index mellem 1 og 100 → fortæller hvor holdbar og effektiv kode er, og hvor let den er at vedligeholde

20-100	
10-19	
0-9	

  - Cyclomatic Complexity (M)
    - Måler strukturelle kompleksitet af kode
    - Giver billede af hvor svær koden er at test, vedligeholde og fejlfinding

If	Plus 1 i kompleksitet
While	Plus 1 i kompleksitet
Switch Csae	Plus 1 i kompleksitet for hver case

    - Antal testcases for få 100% coverage (Antal af testcases >= M)
  - Lines of code: Antal af CI linjer kode
    - Bruges til at se som en metode ”laver” for meget og dermed burde opdeles
  - Class Coupling: Beskriver kobling mellem klasser
    - Høj kobling → svært at genbruge og vedligeholde
    - Lav kobling og høj sammenhørlighed
  - Depth of Inheritance: Angiver hvor mange klasser, der hænger sammen i klassernes arveheiraki
    - Jo dybere heiraki, desto svære er at finde ud af hvor klasserne er defineret.
- Kvalitative
  - Kodestil og kode standatd
  - Fejl: Løse pointer, memory leaks, uinitialiseret variabler

Værktøj: Resharper + compiler → sætte til højeste warning så man ikke overser noget

Dynamisk analyse: Hvad kan man finde ud af når man kører programmet.

- Kvalitativ:
    - Unhandles exceltions, index error, double delete, stack ovwerflow… fanges ved DA.
    - Hvordan?
        - Logging/tracing: udskriv på skærm
        - Stack trace: Fortæller hvilken metode har kaldt hvilken metode
        - Memory dumpe: Skriver indhold fra RAM i fil
- Kvantitativ
    - Absolutte tid: Tid før man får svar tilbage
    - CPU-cyklus: Antal sammentidelige brugere på en server
    - Memory footprint: Hvor meget RAM bruges det
    - OS Ressourcer: Hvor mange filer kan man have åben
    - Memory leaks: Glemmer delete
    - Andre flaskehalse (Virtuel vs. Fysik hukommelse)

    - Profilers: Program som overvåger kørslen af et program og foretager målinger undervejs
        - Instrumented: Høj påvirkning af kode → Høj præcision
            - Compiler indsætter kode i hver funktion/linje
            - Programmet køres og data opsamles
            - Data reporteres tilbage
        - Sampled Lille påvirkning → medium præcision
            - Compiler laver et ”codemap”
            - Programmet køreres overvåget
                - Data opsamles på bestemte tidspunkter (faste intervaller)
                - Data reporteres.

    - God kandidat for Continuous integration
        - Lange tinge byg kan med fordel køre om natten
            - Uden spild af tid og penge

2 FAKES

Indledning: Det objekt som skal testes afhænger typisk af andre objekter som vi ikke har kontrol over.

- Test kan ikke kontrollere hvad afhængigheder returnerer til koden under testen
- Eller hvordan den opfører sig

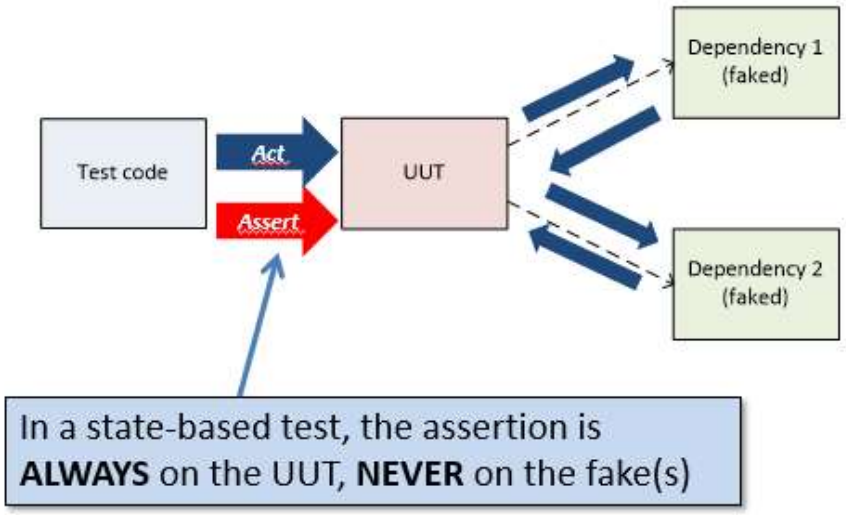
Fake: En falsk version af en klasses afhængighed

- Bruges når vi ønsker at isolere UUT fra systemet, således testen foregår isoleret.
- Eller hvis afhængigheden ikke er implementeret

Unit test opdeles i typer **Tilstandsbaserede test** og **Adfærdsbaserede test.**

Tilstandsbaserede test:

- Bruges for at teste om UUT er i den forventede tilstand efter påvirkning
- Her anvendes en type fake som kaldes "Stub"
- STUB: Snyde udgave af UUT afhængighed
    - Findes kun for at opfylde UUT behov for en afhængighed
- 1. Arrange: Sæt UUT og afhængigheder op
- 2. Act: Stimuler UUT
- 3. Assert: Tjek som tilstanden af UUT er som forventet.



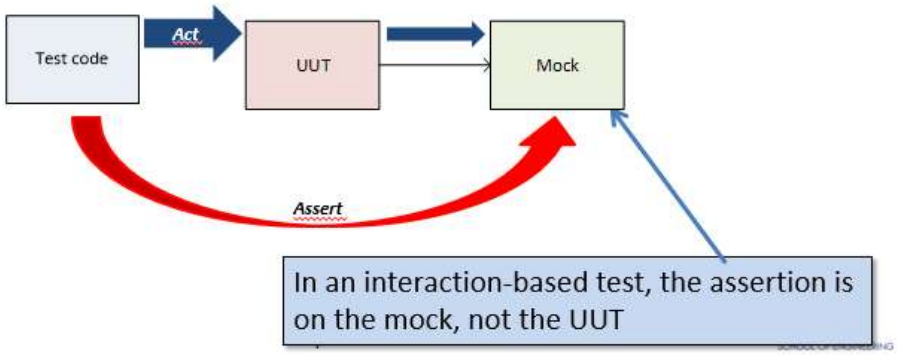
- STUB: Er en kontrollerbar erstatning for en afhængighed i systemet
    - At vende en stub kan koden testes uden at skulle håndtere afhængigheder direkte
    - Kan ALDRIG fejle en test, er der kun for at stimulere situationer.

Eksempel:
En publisher som sender beskeder til sine subscribers.
Hvordan skal publisher testes?
I denne type test kan den testes ved at verificere om publisher holder styr på sine subscribers.

- Vores UUT skal derfor tjekke på afhængigheder, som returner som de subcribed.

Adfærdsbasserede test:

- Bruges for at teste om UUT har haft den forventede interaktion med dens afhængigheder som et resultat efter en påvirkning.
- Her anvendes en type fake som kaldes MOCK
- MOCK: Snydeudgave af UUT afhængighed
    - Mere intelligent end STUB
    - Kan optage hvilken adfærd UUT har haft med den
- 1. Arrange: Sæt UUT og afhængigheder op
- 2. Act: Stimuler UUT
- 3. Tjek som MOCK klassen blev interageret med korrekt af UUT
    - UUT ændre ikke tilstand



- MOCK: Beslutter hvorvidt vores UUT har bestået eller fejlet.
    - Gøres ved at verificere hvorvidt UUT kaldede MOCK objektet som forventet.

Eksempel:
En publisher som sender beskeder til sine subscribers.
I denne type test skal det derimod testes om en meddelelse sendt af publisher er blevet modtaget af subscriberne. For at teste dette skal vi tjekke på UUT men derimod på afhængigheder.
Har de modtaget beskeden efter vores påvirkning af UUT.

Forskellen mellem MOCK og STUB

- STUBS erstatter et objekt, således man kan teste et andet objekt uden problemer
- ASSERT laves på MOCK objektet
- STUB kan ikke fejle tests det kan MOCKS

Ulemper ved manuelle mock og stubs

- Tager tid at skrive
- Svært at skrive dem til klasser som har mange metoder, properties og events
- Svært at genbruge

Isolation Framework

- NSubstitute er et Isolation Framework
    - Et genbrugeligt bibliotek som kan lave fake objekter ved runtime på baggrund af interfaces
    - Gør komplekse test lettere, hurtigere og der laves færre fejl.
- Asserte på resultater af et funtkionskald (Stub)
- Asserte på om en funktion i en mock er blevet med Recieved() (Mock)

3 UNIT TEST

Indledning:
Unit test er en testmetode der verificerer, at de individuelle enheder i kildekoden virker efter hensigten.

- Unit test er den mindst testbare enhed i en applikation.

Black Box vs. White Box test: Grundlæggende to måder at teste
BlackBox: Test metode hvor den interne struktur/design/implementering ikke kendes af den som tester

- Anvendes hovedsageligt til højere testniveauer (Aceppttest og systemtest)
- Ansvar: Generelt laves dette er uafhængige software testere

WhiteBox: Test metode hvor den interne struktur/design/ implementering kendes af den som tester

- Anvendes hovedsageligt til lavere testniveauer (Unit og integration test)
- Ansvar: Laves af den som skrevet softwaren (udvikleren)

Test før → manuelle test (fx output på konsollen)

- Tager lang tid, mange gentagelser
- Ikke effektivt → skal selv verificere resultat

Unit Test Framework (NUnit): Mulighed for automatiserede test, fejlrapporter, og nemt at lave nye test

- Readability (næsten menneskesprog *"Assert.That(myvalue, Is.EqualTo(result)"*
- Framework skal køre og revidere unittesten og deres resultat
    - Skal bruge to attributter for at køre (findes flere!)
        - [TestFixure] - fortæller at klasse indeholder automatiserede test.
        - [Test] - sættes på en metode for at fortælle at det er en automatiseret test.
- Regler man følger for navngivning
- Opsætning af unittest, følger typisk de 3 A'er
    - Arrange: Arranger objekter, lave og sætte dem som er nødvendigt for testen.
    - Act: Handler på objektet
    - Assert: Tjekker at noget er som vi forventer

- Assertions modeller: To måder at asserte på
    - Classic model
        - Anvender AreEqual metoden
        - Tager to argumenter, en værdi samt metoden om ønskes testes
    - Constraint Model
        - Anvender Assert.That()
        - Bruger den faktiske værdi og en constraint

Green Zone:

- Adskil unittest fra test (integration test)
- Unittest kan derfor køres hurtigt og effektivt

Eksempel: Calculator opgave

4 TESTKVALITET (COVERAGE OG BVA)

Indledning: Handler om at lave test om er gode nok og overskuelige.
Coverage er en automatiserbar måling, som bruges til at indikere, om man har testet godt nok.

- White Box testing teknikker

Coverage:

- **Test Coverage**
  - Vil måle hvor god testen er
    - En måleenhed: Hvor meget koden er "dækket" af vores test?
  - Test Coverage måler kvaliteten af test (Ikke det faktiske produkt)
  - Processen af at bestemme hvilke dele af programmet som er dækket af tests
  - Bruger denne viden til systematisk at udvide testcases til den udækket del af programmet.
  - Giver en kvantitativ måling af code coverage
- **Line Coverage**
  - Tjekker hvorvidt alle "Statements" er blevet eksekveret

$$Line\ Coverage = \frac{Source\ lines\ reached\ by\ test}{Source\ lines\ in\ code} * 100\%$$

  - Ulemper:
    - Kan ikke tjekke falske betingelse
    - White Box test
    - Insensitiv overfor kontrolstrukturerer (Tjekker ikke om begge scenarier af en if-statement er kørt)
- **Branch Coverage**
  - Måler hvilke beslutningsoutput er blevet taget (hvilke veje er kørt)
  - Måler i hvilket omfang alle "Beslutningspunkter" (if, while, switch) er blevet fuldt udført.
  - Inkluderer også "asynchronous branches" som exeptions og interrupt handlers

$$Branch\ Coverage = \frac{Number\ af\ executed\ branches}{Total\ number\ og\ branches} * 100\%$$

- Målet skal være 100% coverage
  - Ved en coverage på 95% vides intet om de resterne 5%
  - Hvis noget ikke kan dækkes skal dette dokumenteres

Fordele og ulemper ved coverage

- Fordele:
  - Simples objektive målinger for kvaliteten af testen
  - Hjælper med at vise hvor der mangler test
  - Detekterer trends (hvis coverage bliver lavere eller højere)
- Ulemper:
  - 100% coverage er ikke en garanti for at koden virker perfekt
  - Dyrt og afhænger meget af værktøjer (kan derfor heller ikke flyttes)
  - Fortæller intet om omission fejl

Automatisering

- Anvender dotCover → line coverage
- Tager ekstra tid → ekstra kode skal køres
- Ideal til Continuous Integration
  - Laves ved hvert push
  - Kan køre om natten

BVA: Boundary Value Analyse

- Bruges til at bestemme hvilke værdier som bør testes
- Grænseværdier af gyldige og ugyldige værdier testes
- Grænseværdierne har større for at fejle → bedre sandsynlighed for at finde fejl

Boundary Value Analyse		
Ugyldig (min-1)	Gyldig (min, +min, -max, max)	Ugyldigt (max+1)

EP: Equalicalence partitions

- Alle de værdier har input hvor output har den samme værdi
- Opdeler test betingelser op i grupper som kan regnes for at være ens
- Vælges et tal i hver gruppe som testes
    - Hvis det fejler ses det som at hele gruppen (som enhed) fejler

5 DESIGN FOR TESTABILITY

Indledning: Test funktionalitet af klasse → skal den fjernes fra resten af systemet (afhængigheder)

- Afhængigheder gør test besværlige/umulige → ingen kontrol over afhængighederne
    - Hvorfor fejler testen?
    - Kan derfor ikke teste en ting af gangen…

Men, med mindre systemet er designet specielt til dette, vil dette ikke være en nem opgave.

Afhængigheder:

- I design fase → finde objekt afhængigheder og bryde dem.
- Ved at bryde dem kan vi under tests manipulere med afhængigheden → stub eller mock.

Interface:

- Afhængigheder brydes ved at bruge interfaces → abstraherer en afhængighed væk

Kontrol over afhængigheder:

- Et testbart design tillader os at fjerne en enkelt klasse (The UUT) fra resten af systemet
- Derefter kan vi kontrollere hvilke afhængigheder UUT skal bruge. ("Fake" versioner af afhængigheden!)

- Der er 3 steps mod kontrol (Triple I)
    - IDENTIFY: Identificer de eksterne afhængigheder
    - INTERFACE: Introducer et interface ved afhængigheden
    - INJECT: Erstat (inject) afhængigheden

Dependency injection: Kontroller typen af afhængigheder: Fakes eller virkelige
Construktor injection: Afhængigheder ind i constructor → så construktor ikke oprette egne afhængigheder

- Contructoren kan nu isoleres fra resten af systemet.

Proproty injection: Implementeres med {get; set;}

- Indsættes afhængigheden efter instantialiseringen af objektet.

Afhængigheder udlades ikke, men udsættes:

- Ved test isoleres UUT → så meget som muligt
- Når alt er testes kan de virkelige afhængigheder indsættes igen.

Eksempel:
Teste en **House** Klasse, som er afhængig af 3 klasser; Bedroom, Kitchen og FrontDoor.
Scenarie: Når ejer forlader house → skal lys slukkes, udstyr slukkes og dør låses.

Første skridt er at lave interfaces mellem klassen som vi tester og afhængigheder

- Interfaces gør koblingen løsere

Der efter skal afhængigheder fakes for at kontrollere House.

Målet: Lav kobling - høj samhørighed → gør koden testbar

6 INTEGRATIONSTEST

Indledning: Integration test er næste skrift fra unit test.
- Udføres for verificere at modulerne også virker sammen.
- Moduler sættes sammen og testes som en helhed.

Formål: Er at teste interaktiviteten og interfaces imellem moduler, og dermed verificere at modulerne også virker sammen.

Test livscyklus
Unit test → **Integration test** → System test → Accepttest

Før integration test:
- Alle unit test af alle moduler skal være færdige
  - Ikke fejl i moduler → så ved vi ikke hvor fejlen kommer fra
- Systemarkitekturen for system skal være kendt (Dependency Tree)
- Plan for integrationen (Hvilken type skal anvendes)

Dependency tree: Visualisere afhængigheder i systemet.

Drivers vs. Stubs
- Driver: En fake som kalder et modul (kalder)
- Stub: En fake som returner data til modul (bliver kaldt)

Integration test patterns: Findes forskellige mønstre som kan avendes til integration test

Big Bang Integration
- Test type som må bære eller briste (prøv og håb på det bedste)
- Alle systemets moduler sættes sammen og der testes.
- Ulemper:
  - Kan først testes sent i udviklingsprocessen fordi alle moduler skal være færdige
    - Sen test → dyr at finde og fixe fejl
  - Det vides ikke hvor fejlen kommer fra → det hele testes i en stor sort boks
    - Finder fejl, men ikke hvor disse kommer fra
  - Tager lang tid
- Fordele:
  - ”Virker” til små, ukomplicerede og stabile systemer
  - Ingen planlægning → det hele sættes bare sammen.

Bottom-Up Integration
- Starter nedefra i dependency træet → tester det nederste lag først
- Ved hvert trin bevæger vi os op igennem træet
- Ulemper:
  - Kræver mange drivers
  - Udsætter test af kritiske kontrol moduler (det øverste lag)
- Fordele:
  - Ingen eller få stubbe → moduler har ikke afhængigheder nedad.
  - Tester få interfaces af gangen → hvis fejl er det tydeligt mellem hvilke moduler
    - Og dermed let at finde og fixe.

Top-Down Integration:

- Starter ovenfra i depencency træet → tester det øverste lag først
- Ved hvert trin bevæger vi os ned gennem træet
- Ulemper:
    - Lave mange stubbe (isolation gør det lettere)
    - Svært at teste low-level interfaces, fordi driveren er i toppen af træet
- Fordele:
    - Få drivere
    - Man udsætter ikke det kritiske øverste lag (tidlig kontrol over kontrolkomponenterne)
    - God til HW og SW samarbejde
        - SW kan teste de øverste lag
        - Mens HW udvikler og laver selve HW

Collaboration Integration:

- Der vælges en "vej/gren" igennem dependency træet
    - Moduler som samarbejder for fuldføre en opgave testes sammen (Use-Cases)
- Ulemper:
    - Lille smule Big Bang (tester flere moduler på en gang)
    - Hvis der opstår fejl kan det være i flere moduler
- Fordele:
    - Intuitivt for kunden → kan følge med i Use-Casen
    - Iterativ udvikling (En mindre del færdiggøres og testes)
    - Højniveau systemer

Sandwich Integration:

- Det bedste fra Top-Down og Bottom-Up kombineres
- Ulemper:
    - Skal bruge lang tid på planlægning
- Fordele:
    - Kombinerer det bedste fra Top-Down og Bottom-Up
    - God til store projekter
    - Et hold kan starte fra bunden et kan starte fra toppen → mødes på midten

Efter man har valgt det mønster man ønsker at anvende, skal der laves en plan (tabel) som fortæller hvad der skal testes, stubbes og hvor skal der laves driver.

Continuous Integration:

- Hyppig integration: Får det udført hele tiden
    - Nattelige Continuous Integration byg for hvert testcase for at tjekke integrationen
- Gør at integration testen ikke kun bliver udført til sidst men i hvert sprint gennem hele sprintet
- Hver gang der afleveres noget og unittesten har bestået → testes om det virker sammen.
- Gør det komplekse opgave (IT) mere overskuelig at det opdeles i mindre opgaver.

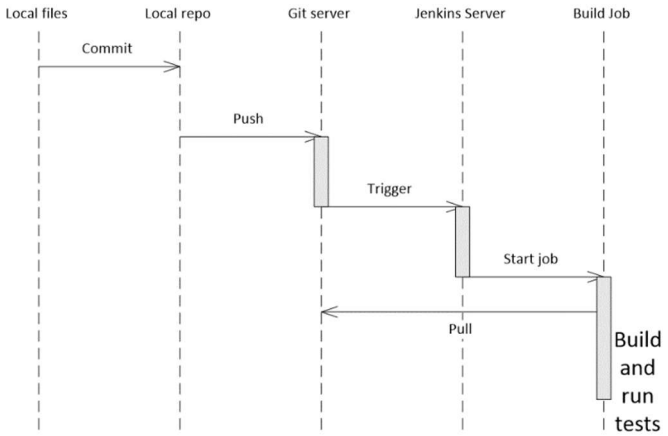
7 CONTINUOUS INTEGRATION

Indledning: Hver gang et gruppemedlem commiter kode til fælles repository skal alle test køres.

Hvad er det?

- Udviklingspraksis hvor medlemmer af et team ofte integrerer deres arbejde
    - Som regel en gang dagligt for hvert medlem
- Hver integration verificeres med et automatisk ”byg” (herunder test)
    - For at registrer integrationsfejl så hurtigt og tidligt som muligt.

Trigger sekvens:



Fordele ved Continuous Integration

- Ingen "integrations helvede" ved slutningen af et projekt
- Aldrig mere end timer fra en virkende (deployable) build
- Hurtig feedback
- Få test udført om natten
- Altid opdateret Metrics code quality
- Certificeret og kontrolleret runtime miljø
- Hyppige pushes: Små stykker kode bliver skrevet ad gangen

Jenkins: Bygger for at kunne køre test. På baggrund af resultatet fra denne test får brugeren besked således fejlende kan rettes hurtigst muligt.
Opsætning på Jenkins til at kun ar køre det næste ”Job” hvis det tidligere byg var stabilt.

- Pipeline
- Det betyder at integrationstesten ikke køre for unittesten er blevet stabilt.
    - Kan se hvor koden fejler (hvor er den kommet til)
    - Hvis det hele køres skal konsolvinduet tjekkes
        - Eller vides det ikke hvad/hvor der er fejl… Spild af tid.

Anvendes mange steder:
Dynamisk analyse → lange tunge byg kan køre om natten
Testkvalitet → disse beregnes og køres ligeledes om natten
Integration test → hyppig integrering, mere overskuelig

8 TEST AF APPLIKATIONER MED GUI

Indledning: Test af hvorvidt den grafiske bruger grænseflade (GUI) i produktet lever op til specifikationer. Det vil sige at det verificeres hvorvidt front-end delen af systemet er godt og rigtig designt.

Test livscyklus
Unit test → Integration test → **System test** → Accepttest

Systemtest:
- Minder om andre test → kræver et SetUp, evaluering samt forventet og faktisk output.
- Men er anderledes for den kører i et komplet system → sværere at validere
- Kan teste mange aspekter som er irrelevante for unitest og integrationstest

Man ønsker at automatiser disse test, dette kan gøres på 3 måder:
- GUI bitmap level: Interaktionerne er optaget med x- og y-koordinater
- GUI object level: Interaktionerne er optaget med komponenter
- Visuel GUI test: Interaktionerne er optaget med referencer til billeder

Teste UI på 4 måder:
- **F5**: Bygges og manuelt tester hele applikationen gennem UI
- **Typisk ikke CUIT**: Tester hele applikationen, dog ikke gennem IU, med (integration og unitest)
- **CUIT**: Tester hele applikationen gennem UI automatisk
- **Test som verificerer UI**: Tester UI i isolation

Coded User Interface Test tilbyder to fremgangsmetoder; **"Record & Play"** og **Programmatisk**

Record & Play:
- Optager skærm med applikationen kørende → derefter generer VS kode
  - Museklik, tekstindtastning osv.)
- Opstilles testcases
- Initialize (svarer til SetUp i NUnit) → åbner applikationen
- CleanUp (svarer til TearDown i NUnit) → lukker applikationen igen.
  - Sker hver gang test cases kører

Programmatisk fremgang:
- Testen skrives manulet.
- Her har programmøren kontrol over testen

Fordele og ulemper:
Fordele:
- Automatiske test udføres på en måde således de minder om en bruger som bruger systemet
- God til validering af brugerens "rejse" gennem applikationen
- Viser "flowet" igennem programmet

Ulemper:
- Kan være ustabile (hvis UI ændres uden funktionaliteten bag ved ændres)
- Langsom udførsel
- Record & Play supporterer ikke altid de teknologier der bruges i applikationerne, og genkender derfor ikke nødvendigvis alle UI-objekterne.