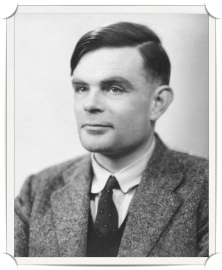
Test Driven Development



1930

- “Turing Maskine” koncepted blev opfundet 1936
- Atanasoff-Berry Computer: Første elektriske & digitale computer (Clifford Berry) Påbegyndt 1937, Prototype 1939 (2 tromler med 1600 kondensatorer. 60 Hz fra AC strøm. 50 bit fixed-point)
- Plan-Do-Study-Act: Metode til kvalitetssikring i 1939 af Walter Shewhart, Bell Labs. Forgænger til Iterative & Incremental Development

1960

- X-15 var en milepæl inden for IID. Stor succes, takket være IID. Fløj op til grænsen til det ydre rum. Holder stadig hastighedsrekorden for bemandede fartøj.
- Folk med IID erfaring (bla. Jerry Weinberg) fortsatte i Mercury. Time-boxed iterationer på en halv dag; Test-First Development; Code review af alle ændringer; Top-down udvikling med stubs (forgænger til mocking - symbolsk manipulering i OS’et hjalp).
- Mercury lærte også fra IBM (John von Neumann?) og Motorola (simulering udviklet med noget der ligner XP forbløffende meget)
- Man brugte unit testing på Mercury, men det havde vist ikke noget navn den gang; man antog bare at enhver professionel ville gøre det på den måde. Vel-struktureret og test-bart design var helt centralt.
- Man skrev testen først: Man skitserede sine test cases før man satte pennen til sin “coding pad.”

I think if people did these things today, we'd be in a lot less trouble. I know because there are people who do these things today, and they tend to stay out of trouble.

On the other hand, people who just throw code at a compile and say "it's unit tested" as soon as they see a green dot, tend to be in trouble all the time.

— Gerald M. Weinberg

- Deres “software development” var time-boxed, iterativ, inkrementiel, test-first, tested med stubs og med masser af code review. Fordi det var den mest naturlige måde at udvikle på.

1990

- Kent Beck læser om test ved sammenligning af hulbånd i en af sin fars bøger, i 1989. Resultatbåndet laves først, så skrives og køres programmet, og de to bånd sammenlignes.
- Dette gøres for hver funktion, mens man udvikler. Han prøver først for sig selv i lille skala på en række små projekter.
- I 1989 skrev Ward Cunningham også den første version af FIT (i Smalltalk). Et test framework som læste og kørte test cases skrevet i et regneark.
- I 1994 skriver/frigiver Kent den første version af SUnit.
- I 1996 kommer han ind som konsulent på C3 projektet hos Chrysler, som blev det første rigtige XP projekt. Et payroll system som skulle erstatte et gammelt COBOL system. Det gik egentlig fint, men man blev dog aldrig færdig med at udfase det gamle og endte i 1999 med at gå tilbage til det gamle system.

Get on with it!

TDD

- **Red:** Write a failing unit test — stop as soon as it fails! (not compiling is failing)
- **Green:** Write product code to make the test pass — stop as soon as it passes!
- **Refactor:** Remove duplication and fix bad names.
- Repeat!

- Farven matcher end-state af dine tests i det enkelte step
- Man skal dog ikke glemme at tænke sig om. Det er en god idé at tænke over hvor man gerne vil hen med sin løsning, og lave en liste over tests man gerne vil skrive. Så får man det ud af hovedet, så man kan fokusere på en ting ad gangen.

Why not test-after?

- Segregate failures
- Pull design from desire to test

- Hvis man skriver koden, og så testen, og testen fejler, er fejlen så i testen eller i koden? Man skal først finde ud af hvad man skal rette, og så rette det. Det tager mere tid. Hvis du skriver testen først bliver dette problem kraftigt reduceret.
- Hvis man først skriver koden og så går igang med at skrive testen, men man finder ud af at det er svært. Så kan man blive nød til at lave om i sin løsning så den er testbar. Først bruger man tid på at prøve at skrive den svære test, derefter bruger man tid på at lave sine ting om og først da kan man endelig få skrevet sin test. Det er spild. Hvis man skriver testen først, kan man skrive den til de testbare API'er man ønsker sig. Dette forøger testability.

Why micro-iterations?

- Faster feedback is faster decision making
- Reduce sunk cost = Reduce risk

- Hver iteration giver feedback, som kan bruges til at traffe beslutninger. Jo hurtigere man får feedback jo hurtigere kan man træffe beslutninger. Det her er feedback om kode og API'er, mens disse udvikles. Iterationerne er beslutningernes hjerteslag.
- Feedback er også indlæring. Man lærer om problemet og løsningen mens den udvikles. Hvis man er på vej ud af en forkert tangent vil man gerne vide det så hurtigt som muligt, så man kan gøre det om. Jo før man ved at noget skal være anderledes, jo mindre når man at investere i noget forkert. Der er andre ting som angriber dette, men det her hjælper.

Why refactor?

- Defer design to when you know enough
- Continuously improve design

- Godt design kræver indsigt. Vi lærer om problemet og løsningen når vi skriver tests og produkt-kode. Vi kan få et bedre design ved at vente lidt med at designe til vi kender problem og løsning bedre.
- Dårlige beslutninger kan ikke undgås. Når vi refakturere løbende undgår vi at designet sander til.

What TDD is not

- A substitute for thinking
- A replacement for architecture
- All the testing you need

- TDD er ikke en erstatning for at tænke selv, men kan måske påpege hvis du ikke har tænkt nok. Du skal have en idé om hvor det er du er på vej hen. Du skal have en retning, så du ikke tester og refakturere i blinde.
- Idéen med TDD som arkitektur kom fra de tidlige XP dage, men det design du hiver ud af TDD er meget tæt på koden. TDD arbejder ikke på arkitektur niveau, og hjælper dig ikke her. Derfor kan man nemt ende med at få en ringere arkitektur hvis TDD er det eneste man læner sig op af. Scopet for TDD er sidenhen blevet reduceret til den disciplin vi kender i dag. Coplien kalder det “revisionistisk TDD” men jeg kalder det “moderne TDD.”
- Selvom TDD i princippet giver 100% coverage, er det ikke nok. Især data-race bugs kan nemt skjule sig, men misforståelser i krav eller API’er er også et problem. Det er også et problem med tests der mangler — glemsel, misforståelser, ingen sad-path tests, etc.

What’s a Good Test?

- Når man går igang med TDD bliver det hurtigt klart for en, at det at skrive tests er en rimelig central del af processen.
- Da det er testne der driver udviklingen, og enhver test man skriver indgår som en del af vedligeholdelsesbyrden for system, skal de helst være gode tests.

Good Tests

- Readable
- Trustworthy
- Maintainable
- (Fast)

- Roy Osherove siger at gode tests har tre sammenhængende egenskaber. De er letlæselige, troværdige og vedligeholdelsesvenlige. Det er også rart hvis de er hurtige, men dette er ikke en central egenskab.
- Når tests er letlæselige bliver de nemmere at vedligeholde. Når de er nemmere at vedligeholde, kan de holdes opdaterede og letlæselige. Når de er vedligeholdte og letlæselige er det nemmere at stole på at de tester det de påstår de tester.
- Det er rart at have hurtige tests, thi man nemt ender med at have mange af dem.

Readable Tests

- Name tests after the behaviour they check
- Separate setup, action & checking
- Don't DRY out the context
- Don't use magic values

- Når man siger at man kun skal teste én ting in en test, så menes der en "unit of behaviour" i forhold til abstraktionsniveauet for den unit man tester. Hvis du kan sige med ord, hvad det er for en opførelse du vil teste, så har du et navn til din test.
- Det er ofte nemmere at overskue en test når der er mindst en linie til hver af disse steps. Kan godt samle dem for meget små tests, men pas på det ikke bliver rodet.
- Factory metoder er gode, men man skal også kunne se hvad der foregår. Balancen mellem for mange og for få detaljer i en test er svær.
- Undgå magic numbers, strings og booleans. Put dem i konstanter eller variabler med et godt navn så man kan se hvorfor de er der.

Maintainable Tests

- Factory methods
- Only test the public API
- Avoid logic like if's and for-loops
- Don't use dynamic values
- Decouple tests

- Vedligeholdelsesvenlige tests knækker mindre når man ændre koden. Løs kobling og høj samnhørighed. Factory metoder afkobler fra konstruktører.
- Tests kun på public API afkobler fra impl. detaljer.
- Undgå logik. Hvor der er logik kan der også være bugs.
- Beregn ikke den forventede værdi i testen. Forventede værd skal være en konstant. Ellers kan test og produkt kode have samme bugs. Eks. undgå stregn-konkatenering.
- Lad være med at lade en test kalde en anden test, eller have nogen anden afhængighed mellem tests. De skal være helt afkoblede fra hindanden. De skal kunne køres i vilkårlig rækkefølge.

Trustworthy Tests

- Use deterministic values
- Check coverage
- Separate unit and integration tests

- Undgå Random, System.currentTimeMillis, DateTime.NOW, etc. De skal køre med de samme værdier hver gang. Brug konstanter i stedet for.
- Kan du stole på dine test? Jo højere coverage jo større er chancen for at det bliver opdaget hvis du laver en fejl. Prøv at introducere fejl og se om en test fejler.
- Dine unit tests skal helst være fuldstændigt deterministiske. Ellers hvis du checker koden ud og kører testne og de fejler - hvad så? Er det en bug? Er der noget galt med dit miljø? False-positives kan gøre det svært at tage en test seriøst. PHP lavede for nylig en release med en kritisk sikkerhedsbug som blev fanget af deres unit tests, men de releasede alligevel.

Basic Correctness

- Nu jeg nævner coverage og hvor vigtigt det er for at man kan stole på sine tests

Basic Correctness

- For every unit:
  - Verify correctness of exposed APIs
  - Verify correct use of dependent APIs

- Vi kan ikke verificere alle veje gennem et system. Kompleksiteten stiger kombinatorisk.
- Men givet en unit fungere korrekt internt, og bruger eksterne APIer korrekt, så er vi nået meget langt.
- Kompleksitet stiger linært for den enkelte unit.

Unit Test Reviews

Test Reviews (like code reviews, but on tests) can offer you the best process for teaching and improving the quality of your code and your unit tests while implementing unit testing into your organization.

— Roy Osherove

http://artofunittesting.com/unit-testing-review-guidelines/

- Roy mener at unit test reviews er vigtigere end code reviews, fordi de er tættere på kravne og er eksempler på hvordan designet bruges. Implementation af et krav er i princippet mindre vigtig.

Conclusion

- Plan ahead a little
- Listen to the tests
- Maintainable, readable, trustworthy
- Basic correctness
- Unit test reviews

?