1. Statisk analyse

Når vi ønsker at teste et program, findes der som udgangspunkt to metoder.

Test mens programmet kører / mens programmet ikke kører.

Allerede før programmet kører, er der en masse regler som skal være opfyldt, fordi vi ved at ellers kommer programmet til at fejle. Dette er f.eks. symbolerne {( som skal være lukkede, samt ; som afslutter linjer

Mange af disse ting bliver automatisk detekteret af vores compiler, da den laver en statisk analyse, altså en test af koden uden at programmet kører.

Resharper er et andet eksempel på et statisk analyse værktøj, som udover de meget kritiske fejl, også informerer om mindre fejl der kan gøre programmøren opmærksom på at noget måske ikke er som det skal være. Feks kan der gøres opmærksom på en variabel der ikke bliver brugt, eller ikke er blevet initializeret.

Det er også muligt at lave et index over hvor ”god” koden er, vha. en lang række værdier for f.eks. linjer i en metode. Dette beskriver hvor kompliceret koden er, og dermed også hvor nemt det vil være at vedligeholde eller at overdrage til en anden programmør.

Desuden kan regler om kode stil også vedligeholdes gennem et statisk analyse værktøj.

Fælles for alt dette er som sagt at det er en analyse der foregår uden at programmet kører. Der vil derfor altid være nogle ting som ikke bliver testet, og man kan ikke være sikker på at programmet forløber som det skal, alene på baggrund af den statiske analyse.

Men når jeg siger sådan, skyldes det alene at den statiske analyse aldrig vil være fyldestgørende nok. For at forklare, antager vi for et øjeblik at den statiske analyse foretages af en perfekt programmør. Denne person vil ifølge min overbevisning, kunne se alle former for fejl og usikre elementer i koden.

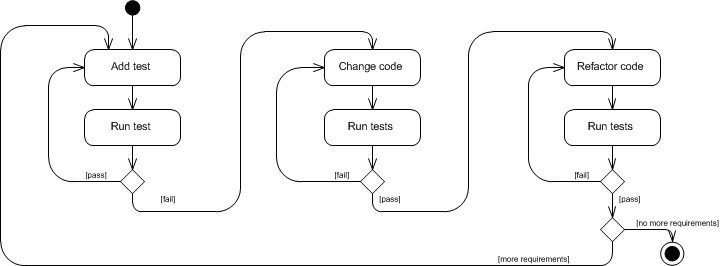
Hvorfor findes det perfekte statisk analyse værktøj så ikke? Fordi det i mange tilfælde er relativt hvad der er korrekt, og hvad der ikke er korrekt. Hvis et givent tal f.eks. skal ende med at være 10, så vil analyse værktøjet kun vide dette, hvis det programmeres således i det konkrete projekt. Det vil altså kræve en masse ekstra unødvendigt arbejde.

* + Quantitative – Software Metrics
    - Various metrics based on code analysis
  + Qualitative – Rules and errors
    - Violation reports and warnings

2. Test-Driven Development (TDD)

I TDD arbejder vi udfra vores unitstests.

Al kode er skrevet for at opfylde en specifik testcase.



Jeg starter med at definere en metode, for at tilfredsstille compileren. Der er ingen implementering.

void DetectConflicts(List<Track> tracksInAir)

Vi ved altså hvad metoden skal modtage, og hvad den skal gøre.

Dette giver os mulighed for at skrive en test, før vi implementerer.

Testen kan sammenlignes med en kravspecifikation.

For at få opfyldt alle krav til metoden, startes med at lave en featureliste:

* Detektere en konflikt jvf. systemets regler
* Må ikke udgive en konflikt som ikke er til stede jvf. systemets regler
* Hver konflikt skal resultere i et event

Vi tager første punkt på listen og skriver en test

public void DetectConflicts\_ConflictingTracks\_ReceivedSeperationEvent()

{

\_incomingListOfTracks = new List<Track>()

{

new Track() {XCoordinate = 5000, YCoordinate = 4000, Altitude = 500},

new Track() {XCoordinate = 5000, YCoordinate = 4000, Altitude = 500}

};

\_eventAggregator.Publish(\_incomingListOfTracks);

\_eventAggregator.Received(1).Publish(Arg.Any<Event>());

}

EventAggregatoren er en substitute vi har initializeret i setup metoden.

Vi kører derefter testen for at bekræfte at den fejler når metoden ikke er implementeret.

Vi implementerer metoden, indtil testen er grøn – Fake it till you make it.

\_eventAggregator.Publish<Event>(new SeperationEvent(tracksInAir[0], tracksInAir[1]));

Her får vi altså en kode som på den simplest mulige måde opfylder den første testcase.

Det er altså nødvendigt at lave flere testcases, så alle elementer på featurelisten bliver implementeret. Alle de foregående tests, skal naturligvis fortsat virke.

TDD - kræver disciplin og evt. 2 mand der samarbejder, for at holde styr på featurelisten, da denne eventuelt skal opdateres under arbejdet.

Det gode er at alt bliver testet og intet unødvendig kode bliver lavet.

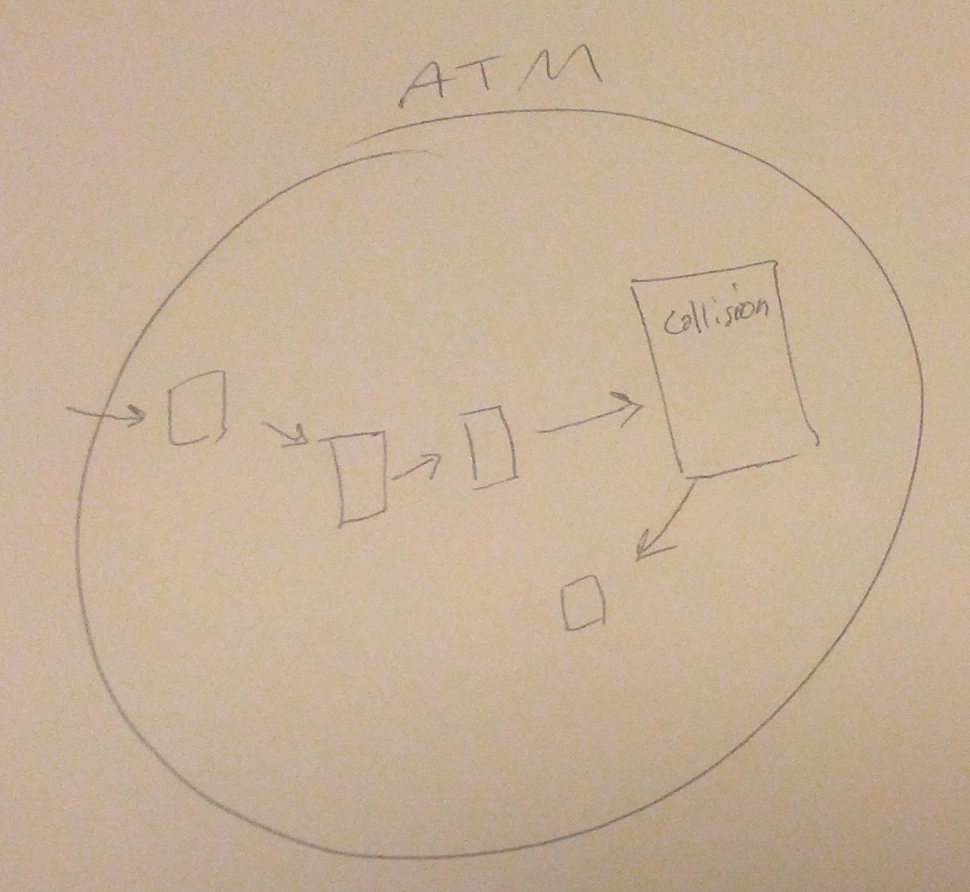
Det dårlige. tager længere tid at kode, kan være besværligt.

3. Fakes

Formålet med fakes er at isolere den klasse man arbejder med.

For at lave fakes, indsættes interfaces ved de klasser som klassen kommunikerer med. Ved hjælp af disse interfaces, kan der udvikles ”tomme” metoder, som f.eks. bare returnerer en værdi. Dette kan yderligere gøres automatisk via et isolation framework, f.eks. NSubstitute

ATM system



CollisionDetector klassen er afhængig af de andre

Ved at isolere den, kan klassen laves samtidig med de andre, og testes uafhængigt af de andre.

ADFÆRDSBASERET TEST – MOCK FAKE. I dette tilfælde ændrer klassen sig ikke, men sender i stedet noget videre til en anden klasse. Det er derfor nødvendigt at lave en adfærdsbaseret test, hvor der testes (assertes) på en anden klasse, altså ikke selve den klasse der er under test, men en fake (mock)

TILSTANDSBASERET TEST – STUB FAKE. Et andet eksempel kunne være hvis selve klassen ændrede tilstand, og f.eks gik i en alarm state, så ville det være nødvendigt at teste (asserte) på klassen under test. Hvis klassen er afhængig af at arbejde sammen med en anden klasse, f.eks. for at lave nogle beregninger, kan der udarbejdes en stub som sender et fast forudbestemt svar tilbage. Dette kaldes en fake (stub)

Ved disse metoder opnår man altså:

* Fjerne afhængigheder
* Udarbejde klassen ”alene”
* Sikre at andre klassers fejl, ikke rammer denne når den testes.
* ”Mindre” arbejdsområde – enklere tests.

4. Unit tests

Hvorfor tester vi?

Tests før SWT – console output ulemper

* Langsomt
* Manuelt, en test af gangen
* Evt. ændring af kode
* meget mere

For at gøre tests mere effektivt vil vi gerne at det er

* Nemt
* Hurtigt
* Udføres automatisk
* Genbruges til enhver tide
* Ingen ændring af kode

For at opnå dette er det vigtigt at lave meget enkle tests, hvor der kun testes et enkelt parameter i hver test. Desuden er det vigtigt at vi er klar over hvilket resultat vi ønsker af testen, og at vi specificerer dette. Et eksempel på en primitiv test kan være:

bool test = false;

if (add(2,2) == 4)

test = true;

assert???

Nunit – Test framework

Giver os et miljø hvor vi nemt og overskueligt kan opsætte vores tests, så vi får overblik over hvilke der fejler.

Black box vs. White box tests

Det er vigtigt at vi tester systemet “set udefra”, så vores test ikke er afhængig af vores implementation. Det dur derfor ikke at lave diverse variabler og lignende public for at kunne teste dem.

[Test]

public void AltitudeDistanceLessThan300\_TracksCloser\_ReceivedConflict()

{

\_incomingListOfTracks = new List<Track>()

{

new Track() { XCoordinate = 5000, YCoordinate = 4000, Altitude = 800},

new Track() { XCoordinate = 5000, YCoordinate = 4000, Altitude = 500},

};

\_eventAggregator.Publish(\_incomingListOfTracks);

\_eventAggregator.Received(1).Publish(Arg.Any<Event>());

}

5. Testkvalitet (coverage og BVA)

Når man tester sin kode, er det vigtigt at gøre det så effektivt som muligt.

Vi vil gerne blive hurtigt færdige med tests, men samtidigt skal vi gerne sikre at alt er testet.

Til dette er coverage et fremragende værktøj.

Coverage går i bund og grund ud på at undersøge om al code er testet, hvilket vil sige alle linjer kode, samt alle forskellige beslutningspunkter. Dette kan dog af og til være vanskeligt at opnå uden at nærme sig en whitebox test, hvilket vi ikke ønsker.

Det er vigtigt at forstå at coverage som udgangspunkt ikke siger noget om kvaliteten af ens test, men kun om hvor stor dækning ens test har.

En af metoderne til at få god coverage, og samtidig skrive gode effektive tests, er at anvende Boundary Value Analysis. (Analyse af grænseværdier)

Tag f.eks. vores ATM system. Her er et eksempel hvor to fly med mindre end 300 meter imellem sig, skal give en konflikt. Derfor vælger vi at teste ved præcis 300 meter, og 301 meter. Alle andre input vil tilhøre en af disse såkaldte equivalens klasser. Med andre ord et andet input, som giver det samme resultat.

[Test]

public void AltitudeDistanceLessThan300\_TracksCloser\_ReceivedConflict()

{

\_incomingListOfTracks = new List<Track>()

{

new Track() { XCoordinate = 5000, YCoordinate = 4000, Altitude = 800},

new Track() { XCoordinate = 5000, YCoordinate = 4000, Altitude = 500},

};

\_eventAggregator.Publish(\_incomingListOfTracks);

\_eventAggregator.Received(1).Publish(Arg.Any<Event>());

}

Normalt vil det også være en god ide at teste med ikke gyldige input, hvis ens metode f.eks. kun modtager tal mellem 1-10, eller at teste med negative tal.

6. Integrationstest

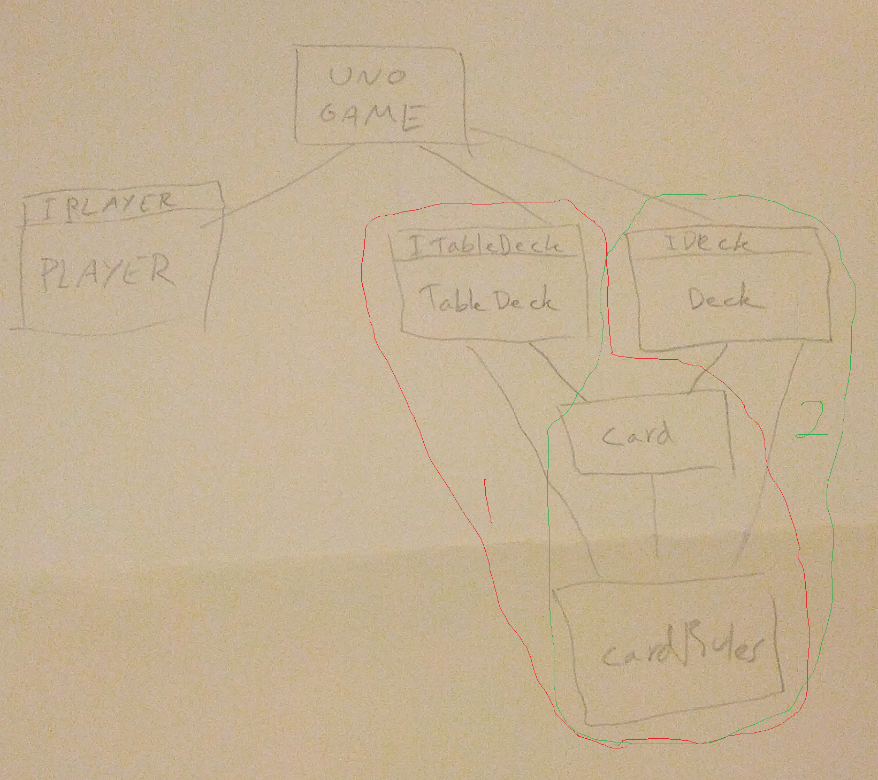
Formålet med integrationstest er at samle systemets dele til et helt system.

En vigtig forudsætning for integrationstest, er at der allerede er udført unittest. Vi ønsker at teste om f.eks. 2 klasser kan kommunikere med hinanden korrekt, og derfor er det yderst vigtigt at vi er sikre på de virker hver for sig først.

Det første skridt i en integrationstest er at lave et dependency træ.

Dette hjælper os til at opdage hvilke klasser der har brug for at kommunikere med hinanden

UNO spil



I dette tilfælde anvender jeg primært bottom up, selvom det måske kan minde lidt om collaboration. Jeg starter nedefra og arbejder mig op, da jeg på denne måde slipper for at lave en masse fakes og da det er et lille system er der næsten ingen ulemper.

Nævn Top Down

Collaboration – Use Case lignende integrations test

Big Bang – wtf? :D

Sandwich model

Ofte er det godt at bruge en kombination af de forskellige integrations modeller.

7. Continuous Integration

Når forskellige elementer i et system udvikles individuelt, kan det hurtigt blive uoverskueligt.



Ved hele tiden at holde det samlet opnås en lang række fordele (git)

* Bedre overblik
* Alle er hele tiden updateret
* Versions kontrol (hvornår virkede det sidst?)
* Det hele skal ikke sættes sammen til sidst, da dette er foregået kontinuerligt

Desuden opnås en række andre fordele idet systemet kan køres automatisk på f.eks. Jenkins

Hver gang en person commiter og pusher kode, køres tests automatisk for hele systemet

* Hurtig feedback og detektering af fejl (unit tests)
* Analyse værktøjer f.eks. ved brug af jenkins