Begreber om Godt Software

Maintainability (vedligeholdelse): Softwarens evne til at blive ændret (funktionalitet, rettet, forbedrelser, miljø, krav).

- **Analyserbart**: Evnen til at blive fejldiagnosticeret, eller finde steder, der skal ændres
- **Forandringsevne**: Evnen til at implementere en specifik ændring
- **Stabilitet**: Evnen til at undgå uventede effekter pga. modifikationer
- **Testbarhed**: Evnen til at ændrede systemer valideres

Fleksibilitet: Evnen til at understøtte tilføjelser eller forhøjet funktionalitet ved kun at tilføje sw-enheder (og ikke ved at ændre nuværende).

Coupling: Hvor afhængig en software-enhed er af andre enheder.

- Lav: få dependencies
- Høj: mange ---
- Giv ansvar så vi får lav kobling → Lokale ændringer har ikke meget effekt, nemmere at forstå units i sig selv, større sandsynlighed for genbrug

Cohesion: Hvor stærkt relateret og fokuseret ansvar og givet opførsel af enhed er.

Law of demeter: Tal ikke med fremmede (ingen ”lange”/nestede metodekald).

+ Sænker direkte kobling

- Interfaces bliver fyldt med mange metoder

For at opdage fejl, må man teste.

Fejl, når software ikke opfører sig som forventet. En **defekt** er årsagen til fejl, da det er forkert implementation af kode. **Test cases** laves for at finde fejl, man giver noget input til UUT og tjekker på output. En **test suite** er en mængde test cases. **Unit under test (UUT)**, en del af systemet vi betragter som et hele. Man kan bruge **manuelle tests** hvilket er test cases der køres manuelt, eller **automatiserede tests**, test suites der bliver kørt og tjekket automatisk af et program. Man skal bruge **regression testing**, dvs kører test cases ofte, for at sikre at systemets opførsel ikke har ændret sig. **Produktions kode**, den kode der styrer opfylder kravene til programmet. **Test kode**, den kode der tester produktions koden.

1 Test-driven development

Test før produktionskode laves, med **automatiserede tests**. Der laves en **test liste**, hvoraf det fremgår, hvad der skal testes – en test vælges som vil **lære** en noget, og komme frem i udviklingen.

Der gås frem efter **TDD-rytmen**:

1. Test tilføjes hurtigt
2. Alle tests køres, og den nye fejler
3. Der laves en lille ændring i produktionskoden
4. Alle tests køres, og den nye går igennem
5. Refaktorer for at fjerne unødvendig kode.

Fake-it-princippet bruges (til en hvis grad), med mindre der kan laves en **åbenlys implementation**.

Triangulering bruges til at sikre at alle aspekter af fx algoritmer dækkes (ved at lave flere tests).

Generelt bruger vi **isolerede tests**, så de forskellige tests ikke påvirker hinanden.

Vi **refaktorerer** for at gøre vedligeholdelse og fleksibilitet større (uden at påvirke systemets opførsel ud af til).

I vores tests bruges **tydelig data** – vi skal lave testene så vi forstår dem (ikke regne dem ud for computeren). Vi bruger **repræsentativt data**, så vi ikke vælger tre tests med data inden for samme **ækvivalensklasse**. Desuden skal testene også være tydelige.

TDD-værdier: Hold fokus, tag små skridt (hurtigt), simpelt

Fordele:

- Ren (vedligeholdbar) kode, som virker (troværdig)
- Hurtigt feedback giver programmøren selvtillid (hvis fejler, ved man det er i skridtet før, eller før det, der er fejl i produktionskoden)
- Troværdig software (pga. testcases)
- Kun ønsket opførsel programmeres (da vi kun programmerer det testene kræver)
- Dokumentation
- Ingen ”driver”-kode, vi har testcasene
- Struktureret programmeringsproces

Ulemper: Hvis interfaces af en grund skal ændres, så skal alle tests ændres. Tests skal holdes simple, ellers er de værdiløse for læseren.

Relater til Black Box Testing samt Godt Software.

2 Systematic black-box testing

Systematisk testing er en planlagt og systematisk proces, som bruges til at finde fejl i veldefinerede dele i systemer.

Black-box testing behandler UUT (enhed under tester) som en sort boks! Det eneste vi ved om den er specifikationen – implementationen er skjult for os. Derudover kræves der viden omkring hyppige fejl lavet af programmører.

White-box testing, vi kender hele implementationen.

Vi kan vælge **ikke** at teste overhovedet (for helt simple accessor og set metoder) – vi kan vælge "undersøgende" **(eksplorativ)** testing, hvor vi bruger vores erfaring – til sidst er der **systematisk** test, hvor vi går efter at finde fejl (for avancerede systemer, hvor sikkerheden er vigtig)!

Vi bruger **ækvivalensklasser** til dele input til UUT op. Input fra samme ækvivalensklasse skal give samme output (og samme fejl).

Vi finder **ECs** ved at kigge på betingelserne for input (og nogen gange output). Herefter kigger vi på vores ECs, og repartionerer evt. de fundne ECs. Herefter kigger vi på dem endnu engang, for at sikre, alt er dækket.

Man deler ofte op i gyldige og ugyldige Ecs, afhængigt af gyldigheden af input. Men gyldigt input kan også være ugyldigt, hvis det fx betyder at metoden bailer ud (fx en if sætning i starten, der bare retunerer en fast værdi).

Herefter laver man testcases ud fra ECs. **Myers heuristik** er god:

1) Indtil alle gyldige ECs er dækket, laves en test, hvor så mange gyldige ECs dækkes som muligt.
2) Indtil alle ugyldige ECs er dækket, laves en testcase, hvis elementer kun ligger i én ugyldig EC.

Herudover er det meget godt at lave tests, som ligger lige på grænsen, **boundary analysis** – tre for hver grænse. Så der sikres, at der findes dumme programmør-fejl, hvor der fx mangler en = i en if-sætning.

Key points: Test ikke på preconditions. Test ikke på overdrevet dum programmering. Myers heuristics skal bruges velovervejet og ikke ukritisk.

Relater til TDD.

3 Variability management

Vi identificerer et punkt i vores kode, som varierer. Der er 4 forskellige fremgangsmåder:

Kildekodekopiering: Der kopieres simpelthen en hel kopi af den nuværende udgave, hvor vi så ændrer i kopien.

+ Simpelt
+ Hurtigt
+ Afkoblet (fejl i den ene udgave kommer ikke i den anden)
- Vedligeholdelsen af flere systemer besværlig (multiple maintenance problem)
- Varianterne glider fra hinanden, og ender med at blive forskellige produkter
- Besværligt at arbejde med mange udgaver af stort set samme kode

Parametrisk løsning: En if-sætning bruges til at bestemme, hvilken variant der skal bruges.

+ Simpelt
+ Multiple maintenance problem undgås
- Ændring ved modificering, troværdighedsproblem, evt. nye fejl (alle varianter skal testes)
- Jo flere krav der styres med denne løsning, jo svære er det at analysere koden (mange if-sætninger)
- Ansvarstillægelse – nu får den specifikke klasse pludselig ansvar for at skifte variant

Polymorf løsning: Vi nedarver og overskriver de metoder, som vi vil ændre i varianten.

+ Multiple maintenance problem undgås
+ Vi ændrer ikke eksisterende kode, tilføjer kun
+ Nemmere at læse ny kode
- Flere klasser at forholde sig til
- Kan kun nedarve fra én klasse (C# og Java)
- Svært at genbruge kode fra andre varianter (super og sub-klasser)
- Varianten bindes allerede ved compile-time, kan ikke ændres med mindre der compiles en ny udgave

Compositionel løsning: Vi uddelegerer ansvaret, der varierer, til andre objekter, som arbejder sammen.

+ Vi ændrer ikke i gammel kode, tilføjer kun nyt – reliable
+ Run-time binding, vi kan ændre variant undervejs
+ Separering af ansvar
+ Separering af tests – nemmere at teste specifikke varianter
+ Variant vælges lokalt og kun et sted
+ Endnu flere variabilitetspunkter kan tilføjes uden at påvirke dette
- Flere klasser og interfaces
- Klienter skal være opmærksomme på at vælge en strategi

Normalt identificeres noget opførsel, der varierer, ved at bruge **3-1-2-metoden**:

3. Find variabilitetspunkt
1. Flyt ansvar til interface
2. Uddeleger opførsel til konkrete objekter

Relater til Dessign Patterns og Godt Software.

4 Test stubs and unit/integration testing

Direkte input er data, som gives af vores testkode og som påvirker opførslen for vores UUT.

Indirekte input er data, som vi ikke kan give direkte i vores testkode, som ændrer opførslen for vores UUT.

Depended-on Unit (afhængig af denne): En del i produktionskoden, som giver data til UUT og påvirker dennes opførsel.

Igen et variabilitetsproblem:

- Under test bruger vi data, som vi bestemmer (**test stub**)
- Under normal brug, bruges den rigtige varierende data (som vi ikke har kontrol over) (**DOU**)

Der findes forskellige **test doubles**:

- **Test stub:** Få indirekte input under kontrol
- **Test spy:** Indirekte *output* optages – for at kontrollere om UUT giver de rigtige kommandoer
- **Mock objekt:** En test spy med fail fast egenskab (fejl på først brud af protokol), også stub
- **Fake objekt:** En hurtig men realistisk erstating (når UUT-DOU er langsom)

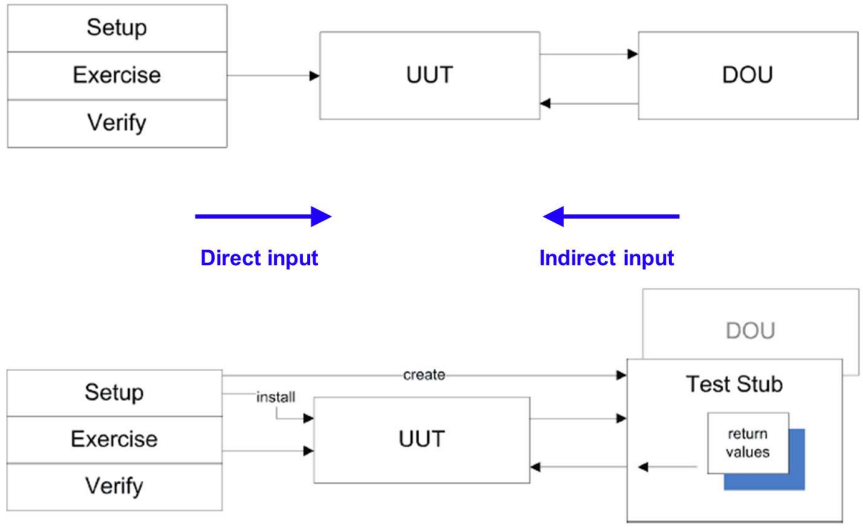
Man skal vide, hvornår man ikke skal teste – fx bør man stole på at firmaer som Sun og Microsoft selv tester, deres software.

Unit-testing: Software-enheden udføres i isolation for at finde defects (fejl-implementationer).

Integration-testing: Software-enhed udføres i samarbejde med andre enheder, for at finde fejl i deres interaktion.

System-testing: Hele systemet udføres for at finde fejl i forhold til kravene.

Relater til Godt Software, TDD og Compositionelt Design.



5 Design patterns

Patterns er beskrivelser af kommunikerende objekter og klasser, som ændres til at løse generelle problemer i en bestemt kontekst.

Becks def.: Et designpattern er en information om noget, som har virket godt i før, som kan bruges til lignende situationer i fremtiden.

Patterns er defineret ud fra de problemer de løser! Mange kan ligne hinanden.

Patterns organiserer koden på to måder:

- Statisk: Interfaces og klasser
- Dynamisk: Give ansvar og interagere

Valg af pattern sker på baggrund af fordele/ulemper-analyse i forhold til problem. At bruge pattern er ikke et mål i sig selv, men en måde at opnå kvalitativ kode.

Pattern fragility: Hvis ikke et pattern implementeres korrekt, kan vi ende op med kun at for ulemperne fra et pattern.

Deklaration: Brug ikke klassenavne, men interface navne i deklarationer.

Binding det rigtige sted: Objekter skal laves og kobles i produktionskoden, hvor ansvaret præcist er at konfigurere og binde.

Beslut dig for en design strategi, og hold dig til denne (ingen evt nemme løsninger).

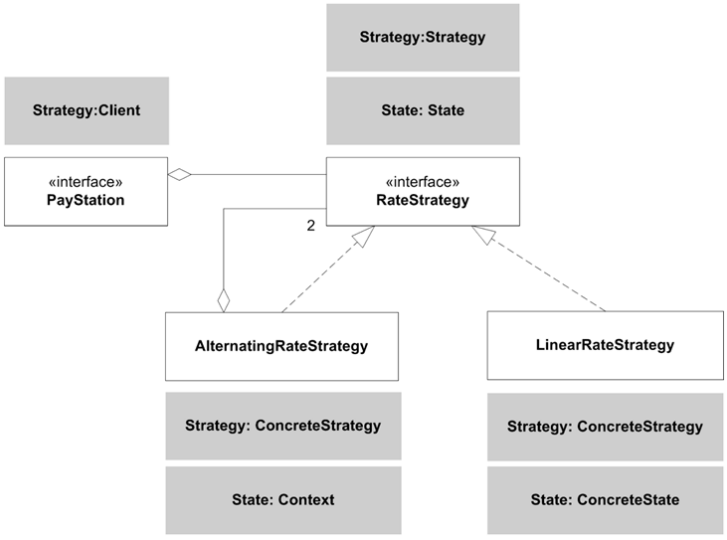
Undgå ansvarserodering – hvis der er nye krav, så flyt ansvar ud i nye klasser (følg pattern).

Patterns som roller: Pattern definerer et sæt roller med noget ansvar, og en veldefineret protokol imellem disse roller. Der kan så laves et **rolle diagram**, hvor de forskellige patterns roller (interface/objekt-navne) også står, udover det alm. UML-diagram.

Patterns som roadmap: Patterns strukturerer, dokumenterer og giver overblik over roller og protokoller i komplekse kompositionelle designs.

Generelt bruges **3-1-2-metoden** til at finde designpatterns.

Relater til Godt Software og Compositionelt Design.



6 Compositional design

3 grundlæggende principper for fleksibelt design.

1. Programmer til et interface, ikke en implementation.
    a. Kig kun på ansvar, ikke konkret implementationsdetaljer.
    b. Interfaces udtrykker specifikt ansvar, klasser koncepter (flere ansvar).
2. Favoriser objekt sammensætning (composition) frem for nedarvning.
    a. Nedarvning: compile-time binding, man får al opførsel fra super-klassen (ansvar kan kun tilføjes, ikke fjernes), data-strukturer defineret, ændring i super-klasse fører til gennemtestning af sub-klasser, høj kobling, indkapsling brydes
    b. Lav kobling, ansvar deles tydeligt op
3. Betragt, hvad der skal variere i designet (eller sammenfat den varierende opførsel).
    a. Betragt, hvad man gerne vil være i stand til at ændre uden at redesigne.
    b. Kode der skal være stabil identificeres, og kode der skal variere identificeres.
        i. Change by add, not by mod.

Dette er principper, ikke love. Brug kun, hvis fordelene kan realiseres. Ellers hold sig til TDD-værdien: simpelhed.

Compositionel løsning: Vi uddelegerer ansvaret, der varierer, til andre objekter, som arbejder sammen.

+ Vi ændrer ikke i gammel kode, tilføjer kun nyt – reliable
+ Run-time binding, vi kan ændre variant undervejs
+ Separering af ansvar
+ Separering af tests – nemmere at teste specifikke varianter
+ Variant vælges lokalt og kun et sted
+ Endnu flere variabilitetspunkter kan tilføjes uden at påvirke dette
- Flere klasser og interfaces
- Klienter skal være opmærksomme på at vælge en strategi

Sprog-centriske perspektiv: Objekter er felter og metoder, konkret kode.

Model-centriske perspektiv: Fokus på koncepter og relationer (nemmest for noget ”real-world”). Objekter skal gøre noget – vi animerer dem, da et flysæde fx ikke har en opførsel. Først defineres klasser, derefter deres opførsel.

Ansvars-centriske perspektiv: Objekters opførsel og ansvar er centrale. Et OO-program består af nogle interagerende objekter, som hver især indtager nogle roller, og udfører en service.

Opførsel: Noget gøres på en observerbar måde. Når vi kalder en metode på et objekt.

Ansvar: Ansvar abstraherer konkret opførsel væk. Interfaces bedre til at beskrive ansvar end konkrete klasser, da disse har implementation med. Metode signaturer deklarerer ansvaret, men selve opførslen (metode kroppen) dikteres ikke.

Rolle: Fokuserer både på funktionalitet og sammenhængen. Definerer noget ansvar og måden der samarbejdes med andre roler.

Protokol: Konvention omkring den forventede (og krævede) interaktion roller imellem.

Rolle-objekt relation:

En rolle – mange objekter: Fx Comparable i Java

Mange roller – et objekt: En personer er både ven, studerende, kæreste, osv.

7 Frameworks

Et framework bruges til at lave flere varianter af et produkt.

Et framework er et sæt samarbejdende klasser, som tilsammen udgør et design, der kan gebruges.

Genbrug af både design og kode, genbrug i et veldefineret domæne, høj genbrugsprocent, skal tilpasses efter kundens behov.

- **Skelet:** Framework giver opførsel på et højt plan af abstraktionen.
- **Domæne:** Frameworket giver en bestemt opførsel inden for et veldefineret domæne.
- **Samarbejdende klasser:** Frameworket bestemmer protokollen, der fortæller hvordan klasserne i frameworket skal arbejde sammen. Disse skal forstås for at man kan bruge det.
- **Tilpasning/abstrakte klasser:** Det kan skræddersyes til en bestemt kontekst (så længe det er i domænet).
- **Implementation:** Genbrug af kode og design.

Applikationer lavet ud fra framework består af framwork koden, den tilpassede kode (vores kode) og ikke framework-relateret kode (vores kode).

Frozen spot: Et sted i frameworket, vi ikke kan ændre på – grundlæggende design og protokoller.

Hot spot: Et tydeligt defineret sted, hvor vores egen specialiserede kode kan sættes ind. Flere metoder til dette:

- Komposition (dependency injection – objekter der definerer opførsel placeres i fw), parametrisering, nedarvning
- Et framework skal understøtte lige fra ingen (interfaces), partiel (abstrakte klasser) til fuld (konkrete klasser) implementation.

Frameworks kræver at de **forståes**, ellers er de nytteløse. Deres **protokol** skal forstås.

Inversion of control handler om, at vores framework der bestemmer flow-of-control. Don't call us, we'll call you.

Frameworks er en **black-box**, så vi tilpasser ikke frameworks via modifikationer men ved at tilføje.

Design patterns

Strategy

Når der ønskes flere forskellige varianter af en algoritme. Den overordnede algoritme lægges ud i et interface, og der implementeres konkrete udgaver.

State

Når der ønskes en ændring af opførsel, når den interne tilstand ændres. Beskriv ansvar for den dynamiske ændrede opførsel i et interface, og implementer den konkrete opførsel i klasser. Når den interne tilstand ændres, flyttes objekt referencen til en anden konkret tilstandsklasse.

Abstract Factory

Der ønskes at kunne lave relaterede objekter, uden at specificere deres konkrete klasser. Der laves en abstraktion hvis ansvar er at lave ”familier” af objekter. Clienten bruger så denne factory, som så laver klasserne, i stedet for selv at skulle lave dem.

Template Method

Der er brug for forskellig opførsel I nogen af skridtene I en algoritme, ellers er algoritmens struktur fast. To måder: Definer algoritmens struktur og ikke-varierende opførsel i en metode, og kald hook-metoder, som indeholder de varierende skridt. Hook-metoderne kan enten være abstrakte metoder (unification) eller de kan være uddelegeret i andre objekter, som implementerer interfaces, der indeholder hook-metoderne (separation).

Facade

Kompleksiteten af et subsystem må ikke vises for klienter. Et interface defineres (facaden), som giver simpel adgang til subsystemet. Klienter tilgår så facaden i stedet for objekterne i subsystemet direkte.

Decorator

Man vil tilføje ansvar og opførsel til et individuelt objekt uden at modificere dens klasse. Vi laver en decorator-klasse ud fra samme interface. Decoratoren videresender alle forespørgsler til det dekorerede objekt, men kan derudover tilføje ekstra opførsel til specielle requests.

Adapter

Konverterer interface for en klasse til et andet interface, som der ønskes. Adapter lader klasser arbejder sammen, som ellers ikke kunne pga. inkompatible interfaces.

Builder

Separer konstruktionen af et komplekst objekt fra dets repræsentation, så den same konstruktionsproces kan generere forskellige repræsentationer.

Null Object

Definer et no-operation object til at repræsentere null. Bruges så man ikke skal tjekke for null hele tiden.

Observer

En-til-mange. Når et objekt ændrer tilstand, skal alle andre objekter gøres opmærksom på dette. Alle obervers indeholder en update()-metode, som subjectet sørger for at kalde, specifikke (de ønskede) steder i koden.

Iterator, Composite, MVC