Eksamensnoter

Softwarekonstruktion og -arkitektur

Martin Nørskov Jensen

201610882@post.au.dk

2. januar 2018

Indhold

1 Test-driven development

- Test-driven development er en del af ekstrem programmering
  - En af de første agile metoder
    - Ser software development som en team indsats
    - Fokuserer meget på samarbejde med brugerene
    - Fokuserer mere på at skrive høj kvalitets kode end at skrive dokumentation
  - Automatiseret testing er en vigtig del
  - Par programmering, hvor person sidder ved computeren og en anden person evaluere koden er central
- Alt TDD test produktionskode kommer fra test
- Er en iterativ process, hvor man følger TDD rhythm i hver iteration
  - Man kan ikke skrive en enkelt linje kode uden at der er en test case, som kræver det
- En test liste bruges til at skrive mangle test og tilføje nye test mens koden skrives

1.1 TDD rhythm

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

1.2 Values

- Take small steps
  - Da hvis man hopper over flere trin, kan man det medfører mange problemer.
  - Selv hvis det betyder, at man skal skrive midlertidig kode.
- Keep focus
  - Fokusere på et trin, et problem ad gangen
- Speed
  - at have en velstruktureret programmeringsprocess der bliver guidet fornuftige principper
  - test idéer tidligt
  - at holde koden maintainable og af høj kvalitet
- Simplicity
  - implementere kode der får produktet til at virke og ikke mere

1.3 Principles

- Automated test
  - use automated test to test your software
- Test First
  - Write your test before your write the code that should be tested
- Test List
  - Before you begin, write a list of all the tests you know you will have to write and add more later
- One Steps Test
  - Pick a test that will teach you something and that you can implement
- Fake It (Til You Make It)
  - What is your first implementation once you have a broken test? Return a constant and then gradually transform it
- Triangulation
  - Abstract only when you have two or more examples
- Isolated Test
  - The running of tests should not affect one another
- Evident Data
  - Make the expected and actual results relationship apparent.
- Representative Data
  - Select a small set of data where each element represents a conceptual aspect or a special computational processing
- Assert First
  - Write asserts first
- Obvious Implementation
  - Just implement simple operations
- Evident Tests
  - To avoid writing defective tests, we keep the testing code evident, readable and as simple as possible
- Child Test
  - If a test case turns out to be too big write smaller test cases that are part of the bigger test case and parses, then reintroduce the larger test case

- Do Over
  - When feeling lost, throw away code and start over
- Regression Test
  - When a defect is reported write the smallest test that fails and ones running will be repaired
- Break
  - When you feel tired or stuck, take a break,

1.4 Testing terminology

- En **defekt** (eller bug) er en aritmetisk fejl
- En **test case** er en definition af input værdier og forventede output værdier for et unit under test
- En **test suite** er et set af test cases (ofte repræsenteret af et test case table)
- En fejlet test bliver refereret til som en **broken test**
- **Manuel testing** er hvor test suites bliver eksekveret af mennesker
- **Automated testing** er en proces hvor test suiten bliver eksekveret af computer og verificeret af mennesker
- **Produktionskode** er kode der defineret softwaren
- **Testkode** er kode der defineret test case for produktionskoden

Unit under test: dayOfWeek	
Input	Expected output
year=2008, month=May, dayOfMonth=19	Monday
year=2008, month=Dec, dayOfMonth=25	Thursday
year=2010, month=Dec, dayOfMonth=25	Saturday

Figur 1: Test case table

2 Systematic black-box testing

- **Systematisk testing** er en planlagt og systematisk process med et eksplicit mål, som er at finde fejl i en fejldefineret del af systemet
- **Black-box testing**: unit-under-test bliver behandlet som en sort box
  - Den eneste viden man har til at guide testene er specifikationen af UUT, generale programmeringstekniker samt typisk programmeringsfejl
- **White-box testing:** hele implementationen er kendt og kan blive brugt til at generer test cases
- Der er følgende test-metoder baseret på kompleksiteten af UUT
  - **Ingen testing:** Mange metoder er så små, at de ikke er værd at test
    - Fx. assessor metoder
  - **Eksplorativ testing:** baseret på erfaring, men følger ikke nogen rigid metode
    - passere bedst til metoder af mellem kompleksitet
    - koster lidt
    - bliver brugt i TDD
  - **Systematisk testing:** her følges en rigid metode til at generer test cases
    - bekostlig, da en del arbejde bliver investeret i analyse af problemet
    - passer bedst til metoder af høj kompleksitet, hvor tiden brugt giver umagen værd

2.1 Equivalence class partitioning

- En **ækvivalens klasse** er et delsæt af alle mulig værdier for UUTen, som har den egenskab at hvis et element demonstrere en defekt i testing, så er det antaget at alle andre værdier producere samme defekt.
- Når man partitioner input element til ECer to egenskaber skal være korrekt for at partitioneringen er sound
  - **Coverage:** Alle mulige input tilhører mindst en af ækvivalensklasserne
  - **Representation:** Hvis en defekt er demonstreret af en element af en ækvivalensklasse, så er antaget den samme defekt kan blive demonstreret af et hvilket som helst andet element i ækvivalensklassen
- En **invalid EC** er et defineret til at være et input, der får algoritmen til at bail ud hurtigt
- **Boundry value analysis**, når man kigger på værdierne der ligger på kanten af en ækvivalensklasserne
  - bruges til at undgå of by one fejl
  - disse værdier er ofte smarte at bruge i test cases

2.1.1 Partitioning heuristics

- Lad vær med at generer test cases til inputs der ikke opfylder en precondition
- Givet en betingelse, det første set af ECer kan blive udledt ved hjælp af følgende retningslinjer
  - **Range:** Hvis en betingelse er specificeret for en interval
    - Definer en valid EC dækker intervallet og to invalid en over intervallet og en under
  - **Set:** Hvis en betingelse er specificeret for et sæt af værdier
    - Definer en EC for hver værdi i sættet og en for alle værdier uden for sættet
  - **Boolean:** Hvis en betingelse er specificeret som en skal være betingelse
    - Definer en EC for true og en for false
- Givet at specifikationen af UUT defineret en aritmetisk operation, så
  - **Addition og subtraktion:** hvis computeringen indeholder addition eller subtraktion
    - Vælg en valid EC for det neutrale element 0 og en valid EC for alle andre elementer
  - **Multiplikation og division:** hvis computeringen indeholder multiplikation eller division
    - Vælg en valid EC for det neutrale element 1 og en valid EC for alle andre
- Heuristikkerne for aritmetiske operation, skal bruges på alle elementerne i en computeringen

2.1.2 Myers heuristics

- For at begrænse antallet af test cases **Myers heuristikkerne** kan blive brugt
  1. Indtil alle valid ECer er dækket, definere en test case, som dækker så mange valide ECer som muligt
  2. Indtil alle invalide ECer er dækkes, definere en test case som kun ligger i en invalid ECn
- For beregninger definere test cases der kun indeholder ECer med ikke-neutrale elementer

2.1.3 Processen

1. Gennemgå kravene for UUTen og identificere betingelser og brug heuristikker til at finde ECer for hver betingelse
2. Gennemgå de produceret ECer og overvej repræsentationsegenskaben for elementerne i hver EC
   - Hvis man er i tvivl om nogle af elementerne er repræsentative for ECen, så reparationerne ECen
   - Brug en EC tabel til dette
3. Gennemgå de produceret ECer for at verificeret at coverage egenskaben er opfyldt
4. Generer test case fra ECerne
   - Brug Myyers heuristikker for at generer et minimalt sæt af test cases
   - Brug et test case table til dette

2.2 Examples of equivalence class and testcase tables

Condition	Invalid ECs	Valid ECs
absolute value of x	–	$x > 0[a1]$ $x \leq 0[a2]$

Figur 2: Ækvivalensklasse tabel

ECs covered	Test case	Expected output
$[a3a], [a3e], [g2]$	('f',5) to ('f',4)	valid
$[a3b], [a3f], [g2]$	('f',5) to ('g',5)	valid
$[a3c], [a3d], [g2]$	('f',5) to ('e',6)	valid
...		

Figur 3: Test case tabel

3 Variability management

- A **variability point** er en veldefineret del af produktionskoden, hvis opførelse det skal være muligt at variere
- ③①② metoden

  ③ Identificere noget adfærd hvad der varierer

  ① Definer et ansvar der dækker, denne adfærd

  ② Deleger dette ansvar til en delegate
- **Change by addition:** er adfærd ændringer, der er introduceret ved at tilføje kode uden at ændre i eksisterende produktionskode.
- **Change by modification:** er adfærd ændringer, der er introduceret ved at modificere eksisterende produktionskode
- **Open/closed principle:** siger at software skal være åben for udvidelse og lukkede for modifikation

3.1 Source tree copy solution

- En kopi af den original taget til den nye source kode og det der variere ændres i dette
  - Bliver kaldet **cut'n'paste reuse**, når man ændre kopier kode for ændre en del af det
- Fordele:
  - **Hastighed:** Det er hurtigt, der skal kun bruges en enkelt kopieringsoperation, samt modificering i testkoden
  - **Simpelt:** Idéen er simpel og hurtig og nem at forklarer til nye programmøre
  - **Ingen implementation indblanding:** To implementation har igen indflydelse på hinanden
- Ulemper:
  - **Multiple maintenance problem:** Der er flere kode baser at vedligeholde. Hvis en bug er opdaget i den fælles kode, skal den rettes i begge kodebaser

3.2 Parametric solution

- Et if statement bliver brugt til at vælge hvilken blok af kode, der skal eksekveres baseret på hvilken variation der kører
- Fordele:
  - **Simpelt:** Conditionals er nemme at forstå og er derfor nemme at beskrive til nye programmøre
  - **Multiple maintenance problemet bliver undgået:** Der er kun en kode base at vedligeholde, derfor er bug fixing meget billigere og fejl er mindre sandsynlige.
- Ulemper:

  - **Pålidelighedsbekymring:** Nye krav er håndteret ved brug af *change by modification*, hvilket øget riskoen for nye fejl
  - **Analysability bekymring:** Når flere og flere krav bliver håndteret ved brug af den parametriske løsning, bliver koden svære at analysere (code bloat)
  - **Ansvars erosion:** Klassen får ekstra ansvar i at håndtere de forskellige versioner

3.3 Polymorphic solution

- Subclassing bilver brugt til at lave ny variation af koden
- Fordele:
  - **Multiple maintenance problemet bliver undgået:** der er kun en kode base
  - **Pålidelighedsbekymring:** Nye krav bliver håndteret ved brug af *change by addition*
  - **Kodes analysability:** Dette forslag gør ikke at koden bliver bloated
- Ulemper:
  - **Stigende antal klasser:** Nye krav til introducere en ny subklasse
  - **Nedarvningsrelationen bliver brugt til en enkelt type af variation:** Man kan ikke bruge nedarvningsrelationen til at håndtere andre typer relationer
  - **Genbrug mellem varianter er svært:** Det er svært at bruge algoritmer defineret i en variant i en anden
  - **Compile-time binding:** Binding mellem en bestemt variant og klassen bliver bestemt, når koden kompileres.

3.4 Compositional solution

- Variabilitetspunktet bliver delegeret ud til et andet objekt og klassen og dette objekt samarbejde om at fuldføre opgaven
  - Den varierende del bliver indkapslet i et interface
- Fordele:
  - **Reliability:** Den originale kode bliver refaktoreret til at tage højde for variationen and nye variation af dette variantions punkt kan blive håndteret ved at lave nye klasser
  - **Run-time binding:** Bindingen mellem klassen og variationen kan blive ændre på run-time
  - **Separation af ansvar:** Ansvar er tydeligt adskilt og tildelt til nemt identificerbar abstraktioner i designet
  - **Separation af test:** Variationer kan testes separeret fra hovedklassen.
  - **Variations udvælgelse er lokaliseret:** Koden der afgøre hvilken variation, der skal bruges er kun et sted i koden.
  - **Kombinatorisk:** Nye typer af variation kan tilføjes uden at det får indflydelse på eksisterende
- Ulemper:

- **Stigende antal klasser og interfaces:** Antallet af klasser og interfaces stiger
- **Klienter må opmærksom på strategierne:** Variant udvælgelse er rykket til klient objekterne

4 Test stubs and unit/integration testing

- **Direct input:** er de værdier eller data givet af test koden direkte, som har en effekt på opførelsen af unit under test (UUT)
- **Indirect input:** er de værdier eller data, der ikke kan blive givet direkte af test koden, som har en effekt på opførelsen af UUT
- **Depended-On unit (DOU):** et unit i produktionskoden, der giver værdier eller opførsel som har en effekt på UUT
- Test double typer (Meszaros (2007))
  - **Test stub:** En double, som er en erstatning af DOU og giver indirekte input, der er defineret af test koden til UUT.
    - Test stubbe gøre koden testbar, da de erstatter DOU og gør at test koden kan kontrollere indirekte input.
  - **Test spy:** En double, som skal gemme UUT'ens indirekte input, til senere bekræftelse af test casen.
  - **Mock object:** En double, der er skabt og programmeret dynamisk af et mock bibliotek, der både må fungere som en stub og en spy
    - Man undgår at skrive test stub
    - Skal fortælles hvad den skal returnerer og i hvilken rækkefølge metoder skal kaldes
    - Kan være meget besværlige at programmere
    - Fejler hurtigt, på den første fejl i protokollen
  - **Fake object:** En double, hvis formål det er erstatte en krævende og dyr DOU
    - bliver ofte brugt til at erstatte en database
- Der er tre niveauer af testing:
  - **Unit test:** er processen, hvor man eksekvere software i isolation for at finde defekter
  - **Integrationstest:** er processen, hvor eksekvere software, som samarbejder får at finde fejl defekter i deres interaktion
  - **System test:** er processen, hvor man eksekvere et helt software system for at finde afvigelser fra dens krav

5 Design patterns

- Definitioner af design mønster
  - **Gamma et al. :** Et design mønster er en beskrivelse af kommunikerende objekter og klasser, som er blevet designet til at løse et generalt design problem i en bestemt kontekst.
  - **Beck et al. :** Et design mønster er bestemt prosa form af at gemme design information, så at designs der har fungerede godt i fortiden kan bruges i fremtiden.
  - **Role view:** Et design mønster er defineret af et set af roller, som har et specifikt set af ansvar og en veldefineret protokol mellem disse roller
  - **Roadmap view:** Design mønster strukturere, dokumentere og giver et overblik over roller og protokoller i komplekse, kompositionel designs. Et design mønster fungere, som en plan (roadmap) af en del af designet.
- **Rule of three:** En løsning skal være set mindst tre gang i tre forskellige systemer får at kunne betragtes som et mønster
- *Objekt samarbejde definerer kompositionel designs:* Derfor betyder det at når man designer software kompositionel, får de objekter til at samarbejde får at få komplekst opførelse
- *Design mønstre er defineret af de problemer de løser:* Det er et problems kendetegn, et design mønster har til formål at løse.

5.1 Compositional design principles

① Programmer til et interface, ikke en implementation

- Klienterne kan frit bruge en hvilken som helst service udbyder klasse (lav kobling)
- Interfaces giver en mere *fine grained* adfærd abstraktioner, fx. `Comparable` interfacet
- Interfaces udtrykker roller bedre, fx. `Comparable` interfacet
- Klasser definere implementation, såvel som et interface

② Foretræk objekt kompostion over klasse nedarvning

- Indkapsling: Klasse nedarvning ødelægger klassen indkapsling og skaber en høj kobling, hvor imod interface giver en god indkapsling samt en lav kobling
- Man kan kun tilføje ansvar til klassen, når man bruger nedarvning, ikke fjerne ansvar
- Man kan ændre opførelse på run-time ved brug af kompositionel designs, hvor imod nedarvning skal det gøres på compile-time
- Klasse hierarkier bliver ofte ændret
- Separat testing: Objekt kompostion gør det nemmere at teste forskellig software units i isolation (unit testing)
- Bedre mulighed for genbrug
- Antallet af klasser, når man bruger objekt komposition stiger og derfor er det vigtigt at have et overblik over klasserne
- Delegation giver mere boilerplate kode

③ Overvej hvad der variere i dit design (eller Indkapsle det der variere)

- Forekommer i mange design mønstre fx. `STRATEGY`

5.2 Design patterns explanation

- `Strategy`
  - Bruges når man skal understøtte varierende algoritmer eller regler
  - Algoritmen ansvar bliver givet i et interface, som implementationer af algoritmen implementere
- `State`
  - Bruges når man skal understøtte skiftende algoritmer, når objektets indre tilstand ændre
  - Konteksten har et tilstandsobjekt for hver tilstand.
  - Konteksten afgøre hvilket tilstandobjekt den skal kalde afhængig af tilstanden
- `Abstract Factory`
  - Bruges når man har en familie af relateret produkter der skal instantiereres på bestemte måder
  - Gøres ved at lave en abstraktion, hvis ansvar det er at lave familier af objekter.
  - Objektet delagere objekt oprettelse til dette objekt
- `Facade`
  - Bruges når kompleksiteten af subsystemet ikke skal vises for klienterne
  - Gøres ved at definere et interface, der giver simpelt adgang til det komplekse subsystem
  - Klienter der bruges subsystemet skal ikke have adgang til det direkte
- `Decorator`
  - Bruges når man vil tilføje ansvar og opførsel til individuelle objekter under at ændre i klassen
  - Gøres ved at lave et objekt der implementere det samme interface.
  - Delagere metode kaldene videre til det rigtige objekt og tilføjer noget opførsel i nogle af metoderne
- `Proxy`
  - Bruges når læsning af objektet er krævene og man derfor først vil loade billedet når det er klart eller når man vil styrer klienters adgang til objektet så som logging, login eller pay-by-access.
- `Builder`
  - Bruges når man har en bestemt måde at konstruere et objekt på men output formatet variere
  - Delagere en del af konstruktionen til et `Builder` objekt
- `Command`

  - Bruges, hvis man gerne vil konfigurere objekter med opførsel på runtime og/eller understøtte undo
  - Gøres ved at indkapsle handling i et interface.

- `Null Object`
  - Bruges i steddet for `null`, når man gerne vil have et objekt til at gøre ingenting
  - Gøres ved at definere en implementation af samme interface som objektet der ikke gør noget

- `Adapter`
  - Bruges når man har en klasse med en ønsket funktionalitet, men dens interface og eller protokol svarer ikke overens med det klienten har brug for
  - Gøres ved at man laver et objekt `Adapter`, som passer til klientens krav, og som kalder de krævede metoder på en instans af klassen med den ønskede funktionlitet
  - `Adapter` implementere `Target` interface, som er det interface klienten skal bruge
  - `Adaptee` er klassen med den ønskede funktionalitet

- `Composite`
  - Bruges når man gerne vil håndtere træ datastrukturer
  - Gøres ved at et fælles interface for `Leaf` og `Composite` kaldet `Component` med delt funktionalitet
  - `Composite` har en form for datastruktur med `Component` typer og deres metoder bliver kaldt med nogle tilføjelser, når metoder bliver kaldt på `Composite`

- `Observer`
  - Bruges når objekt gerne vil blive notificeret omkring tilstandsændringer på et andet objekt
  - Gøres ved at lave et observer interface, som subjektet har en liste af og disse kaldes når den ønskede tilstandsændring kommer
  - Har to varianter
    * **Push variant:** subjekt giver besked til observerne med den del af tilstanden der er blevet ændret
    * **Pull variant:** subjekt giver besked til observerne uden den del af tilstanden der er blevet ændret

- `Model-View-Controller`
  - Bruges når man skal understøtte flere vinduer med grafiske brugeflade, der tilgår samme data
  - `Model`
    * Gemmer applikationstilstanden
    * Holder styr på de associerede views
    * Giver besked til alle views i tilfælde af tilstandsændring
  - `View`

    - ∗ Visualisere models tilstand grafisk
    - ∗ Acceptere user input og delegere dem til den associerede controller
    - ∗ Hold styr på et sæt af kontrollere og lad brugeren sætte hvilken kontroller er aktiv
  - `Controller`
    - ∗ Tolke brugerens input og oversæt dem til tilstandændring til modellen

- `Template-Method`
  - Bruges når man vil have forskellig overførelse i punkter af en algoritme, men algoritmens struktur er den samme
  - Unification
    - ∗ Både template metoden og hook metoder er inde i den samme klasse
    - ∗ Template metoden er konkret og involvere abstrakte hook metoder der kan bliver overskrevet i en subklasse
  - Seperation
    - ∗ Template metoden er defineret i en klasse og hook metoderne er defineret i en eller flere interfaces
    - ∗ Template metoden er konkret og delegere til implementationer af hook interface(s)

6 Compositional design

- Definitioner af objekt orientering
  - **Sprog-specifikke:** Objekter er beholder for metoder og instansvariabler
  - **Model:** Et program ses som en fysisk model der simulere opførelse er en rigtigt en imaginær del af verdenen
  - **Ansvars:** Et program er struktureret, som en gruppe af integrerende agenter kaldet objekter, hvor hvert objekt har en rolle og udføre en service eller opgave brugt af de andre medlemmer.
- **Adfærd (behaviour):** at opføre sig på en bestem og observerbar måde
- **Ansvar (responsibility):** Et stadie hvor man bliver ansvarlig og pålidelig for at svare en forespørgsel
  - Skal ikke være for programmeringsspecifik så som at have en `addPayment` metode
  - Skal ikke være for åbne og uklare så som *"Opfør sig som en paystation"*
- **Rolle (General):** En funktion eller del som bliver udført i en bestem operation eller proces
- Relationen mellem rolle og objekt er en mange til mange relation
  - En rolle kan bliver udført af mange forskellige slags objekter
  - Et objekt kan have flere roller i et system
- **Rolle (Software):** Et sæt af ansvar og associerede protokoller
- Det er vigtig at definere små og cohesive roller
- **Protokol:** En funktion eller delfunktion der er udført i en bestemt process
  - Komplekse roller kan defineres ved hjælp af simple roller
- ③①② metoden

  ③ Identificere noget adfærd hvad der varierer

  ① Definer et ansvar der dækker, denne adfærd

  ② Deleger dette ansvar til en delegate

6.1 Compositional design principles

① Programmer til et interface, ikke en implementation

- Klienterne kan frit bruge en hvilken som helst service udbyder klasse (lav kobling)
- Interfaces giver en mere *fine grained* adfærd abstraktioner, fx. `Comparable` interfacet
- Interfaces udtrykker roller bedre, fx. `Comparable` interfacet
- Klasser definere implementation, såvel som et interface

② Foretræk objekt kompostion over klasse nedarvning

- Indkapsling: Klasse nedarvning ødelægger klassen indkapsling og skaber en høj klasser, hvor imod interface giver en god indkapsling samt en høj kobling

- Man kan kun tilføje ansvar til klassen, når man bruger nedarvning, ikke fjerne ansvar
- Man kan ændre opførelse på run-time ved brug af kompositionel designs, hvor imod nedarvning skal det gøres på compile-time
- Klasse hierarkier bliver ofte ændret
- Separat testing: Objekt kompostion gør det nemmere at teste forskellig software units i isolation (unit testing)
- Bedre mulighed for genbrug
- Antallet af klasser, når man bruger objekt komposition stiger og derfor er det vigtigt at have et overblik over klasserne
- Delegation giver mere boilerplate kode

③ Overvej hvad der variere i dit design (eller Indkapsle det der variere)

- Forekommer i mange design mønstre fx. `STRATEGY`

7 Frameworks

- Framework kendetegn
  - **Skelet / design / high-level sprog / skabelon**
    - Et framework giver applikation adfærd på et høj niveau af abstraktion
  - **Applikation / klasse af software / indenfor et bestemt domæne**
    - Et framework giver en bestemt adfærd i et veldefineret domæne
  - **Samarbejde / samarbejdende klasser**
    - Et framework definere protokollen mellem et sæt af veldefineret komponenter / objekter
    - For at bruge frameworket skal man forestå protokollen
  - **Tilpasning / abstrakte klasse / genbruglighed / specialisering**
    - Et framework er fleksibel, så man kan tilpasse det til en bestem kontekst, så længe at den kontekst er inde i domænet af frameworket
  - **Klasser / implementation / skelet**
    - Et framework er genbrug af kode såvel som design
- **Inversion of control**
  - Frameworket definere the flow of control ikke dig
  - Bliver ofte beskrevet af Hollywood princippet *"Don't call us we'll call you"*
  - Modsat et bibliotek
- På grund af Inversion of control egenskaben definere frameworket også
  - Objekt protokoller
  - Variabilitets punkter
- **Software produkt linje:**
  - er et set af software krævende systemer, som deler et fælles set af features
  - imødekommer de konkrete krav af et bestemt marked, eller mission
  - er blevet udviklet fra de samme hoved assets på en beskrevet måde

7.1 Frozen and hot spots

- **Frozen spot**
  - en del af frameworkets kode som ikke kan blive ændret
  - definere det basale design og objekt protokoller i den endelige applikation
- **Hot spot** (hook variability points)
  - en defineret del af frameworket
  - specialiseringskode kan ændre eller tilføje adfærd til applikationen
- Frameworkets bliver ikke tilpasset af kode modifikation

  - Frameworks er en lukket sort boks, hvor source koden ikke må blive ændret
  - Tilpasning må kun ske gennem den givne adfærd i hot spots, af de mekanismer givet af framework programmørene
  - Change by addition ikke modification
- Mange frameworks kommer med standard implementationer for de fleste eller alle variabilitetspunkter

7.2 Defining Variability Points

- **Framework kode:** Koden der definere frameworket
- **Applikationskode:** Koden der definere specialiseringen af variabilitetspunkterne
- Et framework skal bruge dependency injection
  - Frameworket selv kan ikke instantierer objekter som definere variabilitetspunkter
  - Variabilitetspunkterne selv skal blive instantiereret af applikationskoden og injected ind i frameworket
- En regel siger, at framework kode ikke må bruge et `new` statement på klasser der har hotspot metoder defineret
- Et framework giver en række muligheder for at definere hot spot objekter
  - Eksisterende konkrete klasser
    * Et framework kommer med en række prædefineret klasser
    * Det er applikationsprogrammørens job at lave de prædefineret klasser til noget brugbart
    * Fx. AWT og swing til GUI
  - Subklasser og abstrakte klasser
    * Frameworket har abstrakte klasser
    * Største delen af implementation med høj kvalitet er allerede givet
    * Udvikleren skal give de sidste detaljer
  - Implementation af et interface
    * Frameworket indeholder et interface
    * Giver applikationsudvikleren muligheden for at have fuld kontrol over hot spots
- Et framework skal understøtte spekteret fra igen implementation (interface) til delvis (abstrakt) til fuldt (konkret) implementation for variabilitetspunkterne.

7.3 TEMPLATE METHOD

- Unification
  - Både template metoden og hook metoder er inde i den samme klasse
  - Template metoden er konkret og involvere abstrakte hook metoder der kan bliver overskrevet i en subklasse

- Seperation
  - Template metoden er defineret i en klasse og hook metoderne er defineret i en eller flere interfaces
  - Template metoden er konkret og delegere til implementationer af hook interface(s)

7.4 Compositional design principles

① Programmer til et interface, ikke en implementation

- Klienterne kan frit bruge en hvilken som helst service udbyder klasse (lav kobling)
- Interfaces giver en mere *fine grained* adfærd abstraktioner, fx. `Comparable` interfacet
- Interfaces udtrykker roller bedre, fx. `Comparable` interfacet
- Klasser definere implementation, såvel som et interface

② Foretræk objekt kompostion over klasse nedarvning

- Indkapsling: Klasse nedarvning ødelægger klassen indkapsling og skaber en høj kobling, hvor imod interface giver en god indkapsling samt en lav kobling
- Man kan kun tilføje ansvar til klassen, når man bruger nedarvning, ikke fjerne ansvar
- Man kan ændre opførelse på run-time ved brug af kompositionel designs, hvor imod nedarvning skal det gøres på compile-time
- Klasse hierarkier bliver ofte ændret
- Separat testing: Objekt kompostion gør det nemmere at teste forskellig software units i isolation (unit testing)
- Bedre mulighed for genbrug
- Antallet af klasser, når man bruger objekt komposition stiger og derfor er det vigtigt at have et overblik over klasserne
- Delegation giver mere boilerplate kode

③ Overvej hvad der variere i dit design (eller Indkapsle det der variere)

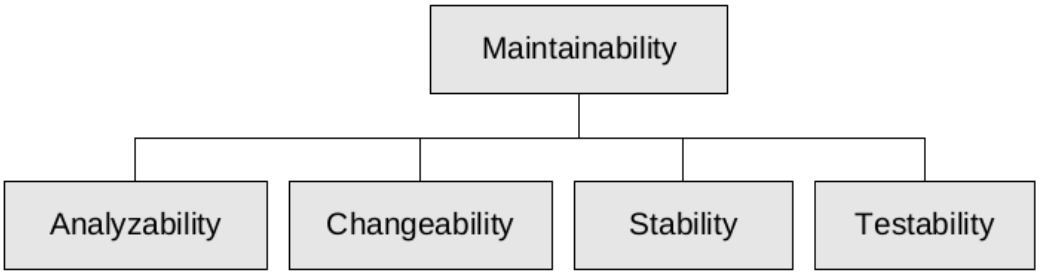
- Forekommer i mange design mønstre fx. `STRATEGY`

8 Clean Code and Refactoring

- **Small**
  - Indent niveauet af en funktion må aldrig være mere end en eller to
- **Do One Thing**
  - En funktion skal kun gøre en ting.
- **One Level of Abstraction**
  - En funktion skal bruge det samme niveau af abstraktion
- **Use Descriptive Names**
  - En funktion navn skal tydelig beskrive hvad den ene ting den gør
- **Keep the Number of Arguments Low**
  - Færrest muligt argumenter er bedst
  - 0-1-2 argumenter måske 3, hvis funktionen bruger flere argumenter gør den mere end en ting
- **Avoid Flags Arguments**
  - Man skal ikke bruge flag argumenter til funktioner, da det betyder at funktionen gør noget forskellige afhængige af flag argumentet
- **Have No Side Effects**
  - Funktionen skal kun gøre det, den siger den gør.
  - Det skal altid være sikkert at kalde en funktion.
- **Command Query Separation**
  - Accessors and mutators
  - **Query:** Ingen state ændring
  - **Command:** Ingen retur værdi
  - Metoder skal enten give et resultat eller ændre noget i begge ting.
- **Prefer Exceptions to Error Codes**
  - Man skal heller smide en exception end at returnere fejl koder
- **Don't Repeat Yourself**
  - Man skal ikke have det samme kode to gange i ens kode base
  - For at undgå Multiple maintanance problem
- **Do the Same Thing the Same Way**
  - Man skal gør den samme ting på samme måde
  - Gør ens kode nemmere at bruge

- **Name Boolean Expressions**
  - Navngiv Boolean expression, gør koden nemmere at læse
- **Bail Out Fast**
  - Tjek boolean expression individuelt og returnere `false`, hvis en fejler
- **Arguments in Argument Lists**
  - Argumenter høre til i argument lister ikke som en del af en funktion
  - Undgå at et argument er en del af metode navnet fx `moveUnitForBluePlayer`

8.1 Flexibility and Maintainability



- **Maintainability:** Softwarens evne til at blive ændret
  - Ændringer kan være rettelser, forbedringer, tilpasning til ændringer, krav og funktionelle specifikationer
  - Kan blive målt i hvor meget det koster at ændre softwaren
- **Analysability:** Softwarens evne til blive diagnosticere mangler eller årsagen til fejl
  - Evnen til at forstå softwaren
  - Det er vigtig at kunne forestå og læse softwaren
- **Changeability:** Softwarens evne til muligøre specifik modifikation
  - Evnen til at tilføj, ændre og forbedre en feature i et system til en fornuftig pris
  - Der er to forskellige dele
    - De aspekter der kan ændres på compile time
    - De aspekter der kan ændres på runtime
- **Stability:** Softwarens evne til at undgå uforudset effekter fra modifikationer af systemet
- **Testability:** Softwarens evne til at validere et modificeret system
  - Evenen til at teste software
- **Flexibility:** Softwaren evne til at understøtte tilføjet/forbedret funktionalitet udelukkende ved at tilføje software dele
  - Evnen til at tilføje funktionalitet ved at tilføj dele
  - Et specielt tilfælde af changeability

8.2 Coupling and Cohesion

- **Kobling:** Det måler hvor meget en software unit afhænger af andre software units
  - Afhængigheder mellem software kommer i mange former
    - Metoder afhænger af klassen instans variabler
    - Klasser afhænger af hinanden, når de er refereret i kode
    - Mellem pakker
  - En software enhed kan have en høj eller lav kobling
  - En høj kobling formindsker maintainability
    - Den formindsker også reliability, da en bug er mere sandsynlig
    - Koden bliver svære at læse
    - Det bliver svære at genbruge
  - En lav kobling gør generelt kode mere pålidelig
    - Gør koden nemmere at genbruge
- **Kohæsion:** Det måler hvor meget relateret og fokuseret krav og de givne adfærd i en software unit er.
  - Det betyder om koden er organiseret
  - En kvalitet, man som udvikler skal gå efter
  - En høj kohæsion betyder at software enheder har få tæt relateret krav
    - Hjælper på reliability og maintainability
    - Forøger analysability
    - Gør det nemmere at undgå fejl
- **Law of Demeter:** Den siger, at man ikke skal samarbejde med indirekte objekter
  - Bliver også kaldt *"do not talk to strangers"*
  - I en metode skal man kun kalde metoder på
    - `this`
    - En metodes parameter
    - En attribut af `this`
    - Et element i en samling hvilken er en attribut af `this`
    - Et objekt lavet inden i metoden
  - Det er en mere en design regel end lov
  - Hjælper imod høj kobling

9 Distribution and Broker

9.1 Distributed computing

- Distribuerede computing er et felt i Datalogi, der studerer distribuerede systemer
- **Distribuerede system:** Et system, hvor komponenter, lokaliseret på netværk computere, kommunikere og kordinere deres handlinger kun ved passere beskeder
- **Klient-server arkitektur:**
  - To komponenter har brug for at snakke sammen, og er uafhængige af hinanden.
  - En af komponenter indikere er forbindelse den anden udbyder
  - Den komponent der udbyder servicen skal kunne håndter et antal forbindelser på samme tid
  - Den der anmoder komponent skal kunne håndtere forskellige resultater
- CLIENT-SERVER mønsteret skelner mellem to typer komponenter klienter og serverer
  - Klienten anmoder information eller services fra en server hvilket betyder at den skal kende
    - Hvordan man forbinder til serveren, hvilken kræver et id eller en adresse til serveren
    - Serverens interface
  - Serveren svarer på klientens request
    - processer hver request for sig selv
    - kender ikke idet klientens adresse eller id før interaktionen
  - Klienter er optimeret til deres applikationsopgave
  - Serverer er optimeret til at håndtere flere klienter
- **Request-Reply protocol:** simulere synkron kald mellem klient og server ved parvis udveksling af beskeder
  - En former request beskeden fra klient til serveren
  - Anden former reply beskeden fra serveren tilbage til klienten
  - Klienten sender request beskeden for venter/blokere indtil reply besked er blevet modtaget
- **Marshalling:** Det er processen, hvor man tager en kollektion af struktureret data elementer og samler dem til et byte array, som kan bruges til at sende netværks beskeder
- **Unmarshalling:** Det er processen, hvor man laver et byte array modtaget over en netværk besked til en ækvivalent samling af struktureret data elementer.
- Marshalling og demarshalling processen skal være enig om formatet, marshalling formatet
- **Named services:** De kan bliver brugt til at oversætte et navn for et fjern objekt til en konkret location og identitet
  - Java RMI giver et register hvor man kan binde et navn til en location
  - DNS servere binder urler så som `google.dk` til en konkret ip-adresse

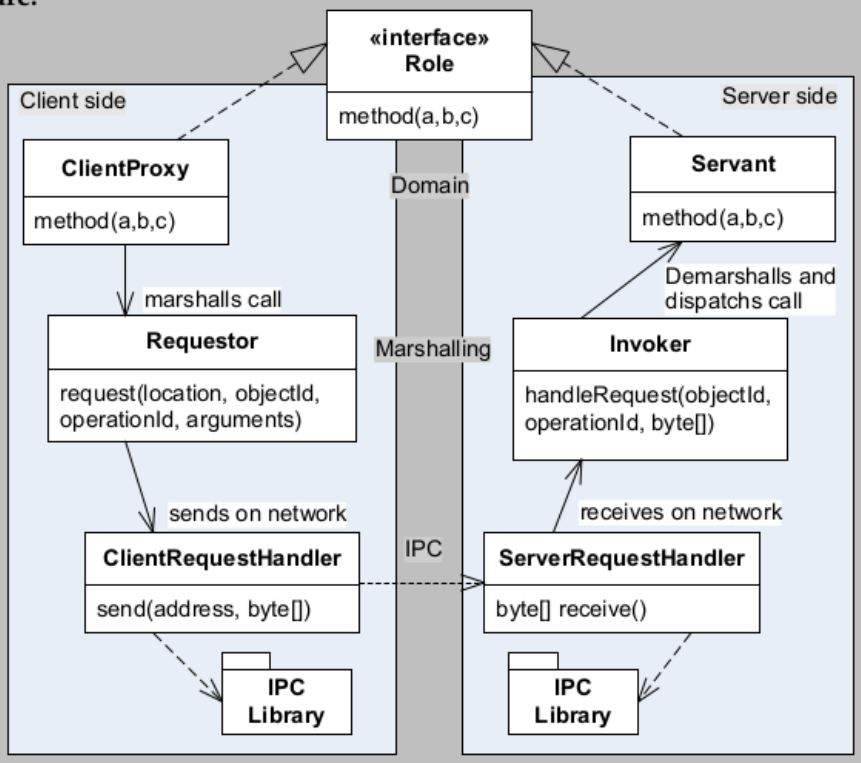
[37.1] Design Pattern: Broker

Intent Define an loosely coupled architecture that allows methods to be called on remote objects while having flexibility in choice of operating system, communication protocol, and marshaling format.

Problem [network only allow binary data to be sent] [Higher abstraction level of methods called on objects]

Solution A **RoleProxy** represents the client implementation of a **Role** that is actually residing on a remote computer. Bla bla

Structure:



Roles Pending

Cost - Benefit The benefits are: *loose coupling between all roles* that provides flexibility in choosing the marshaling format, the operating system, the communication protocol, as well as programming language on client and server side. The liabilities are: *pending :*

9.2 Broker

- `ClientProxy`
  - `PROXY` for remote servant objekt
  - Implementere det samme interface som `Servant`
  - Oversætte alle metoder kald til kald til den associerede `Reqestor`'s `request` metode
- `Requestor`
  - Laver mashalling af metode navn og argumenter til et `RequestObject`
  - Kalder den associerede `ClientRequestHandler`'s `send` metode
  - Demarshalling returneret `ReplyObject` til retur værdi(er)
  - Lav klientside expections, hvis der er sket fejl på server siden
- `RequestObject`
  - Indkapsler alt information omkring metode kald, inkluderende objekt identitet, server location og parameter i et marshalling format
- `ClientRequestHandler`
  - Udfører alt *inter process communication* for klientsiden
- `ServerRequestHandler`
  - Udfører alt *inter process communication* for serversiden
  - Indeholder en event loop tråd der venter på request fra netværket
  - Når den modtager en besked, kalder den `Invoker` med det modtagende `RequestObject`
- `Invoker`
  - Udfører demarshalling af `RequestObject`er
  - Afgøre hvilke objekter, metoder, argumenter og kald på den identificeret `Servant` objekt
  - Udfører marshalling af retur værdien fra `Servant` objektet til et `ReplyObject`
- `ReplyObject`
  - Indkapsler alt information omkring retur værdier, inkluderende exception smidt og fejl beskeder, fra metode kald i et marshalling format
- `Servant`
  - Domæne objekt med domæne implementation på serversiden
- *Altid inkludere format versionen, når man bruger marshalling*

9.3 Example implementation

- ClientProxy:

```
1  package breakthrough.client;
2
3  public class BreakthroughProxy implements Breakthrough,
       ClientProxy {
4      private Requestor requestor;
5
6      public BreakthroughProxy(Requestor requestor) {
7          this.requestor = requestor;
8      }
9
10     @Override
11     public Color getPieceAt(Position p) {
12         return requestor.sendRequestAndAwaitReply(
13                 p.toString(),
14                 OperationNames.GET_PIECE_AT_OPERATION,
15                 Color.class,
16                 p
17         );
18     }
19
20 }
```

- Invoker:

```
1  package breakthrough.marshall.json;
2
3  public class StandardJSONInvoker implements Invoker {
4
5      private Breakthrough breakthrough;
6      private Gson gson;
7
8      public StandardJSONInvoker(Breakthrough
           breakthrough) {
9          this.breakthrough = breakthrough;
10         gson = new Gson();
11     }
12
13     @Override
14     public ReplyObject handleRequest(String objectId,
           String operationName, String payload) {
15         ReplyObject replyObject = null;
16
17         JsonParser parser = new JsonParser();
18         JsonArray array =
               parser.parse(payload).getAsJsonArray();
19
```

```
20          switch (operationName){
21              case OperationNames.GET_PIECE_AT_OPERATION:
22                  Position position =
                        gson.fromJson(array.get(0),Position.class);
23                  replyObject = new ReplyObject(
24                          HttpServletResponse.SC_OK,
25                          gson.toJson(breakthrough.getPieceAt(position))
26                  );
27                  break;
28              case
                    OperationNames.GET_PLAYER_IN_TURN_OPERATION:
29                  replyObject = new ReplyObject(
30                       HttpServletResponse.SC_OK,
31                       gson.toJson(breakthrough.getPlayerInTurn())
32                  );
33                  break;
34              default:
35                  replyObject = new
                        ReplyObject(HttpServletResponse.
36                          SC_NOT_IMPLEMENTED,
37                          "Server received unknown operation
                                name: '"
38                                  + operationName + "'.");
39                  break;
40          }
41
42          return replyObject;
43      }
44  }
```