dSoftArk notater

18. januar 2007

1 Test-driven development

Testing: The process of executing software in order to find failures.

Test case: A test case is a definition of input values and expected output values for the unit under test. (kan være manuel/automatisk)

1.1 Principper

- Software is produced by writing code
- Take small steps
- Keep focus
- Tests should be automated

1.2 Patterns

- Test first
- Test list (kompromis mellem "moving fast" og "keeping focus")
- Fake it (for at lave små trin)
- Triangulation (for at fixe fake it implementationer)
- Isolated test (for at forhindre at tests påvirker hinanden)
- Evident data/tests (skær det ud i pap, så du altid kan forstå det)
- Obvious implementation (balance mellem fake it og simpel implementation)
- Break (træt i hovedet? tag en pause!)

1.3 TDD rytme

1. Tilføj hurtigt en test
2. Kør alle tests, og se den nye test fejle
3. Lav en lille ændring
4. Kør alle tests, og se dem alle virke
5. Refaktoriser for at fjerne duplikeret kode

> Refactoring: Refactoring is the process of changing a software system in such a way that is does not alter the external behaviour of the code yet improves its internal structure.

1.4 Fordele/ulemper

- + Clean code that works
- + Structured programming process
- + Fast feedback gives confidence
- + Strong focus on reliable software
- + Playing with the interface from the clients side (=>YAGNI)
- + No driver code
- - Interface changes are expen$ive

2 Systematic blackbox testing

> Software reliability: the probability that a software system will not cause the failure of the system for a specified time under specified conditions.
>
> Testing: The process of executing software in order to find failures.

2.1 Fejl

- Failure: any deviation of the observed behavior from the specified behavior.
- Error: the system is in a state such that further processing by the system will lead to a failure.
- Fault/bug: the mechanical or algorithmic cause of an error.

2.2 Testteknikker

- Ingen test
- Eksplorativ test
- Systematisk test

2.3 Systematisk test

> Systematic testing: is a planned and systematic process with the explicit goal of finding defects in some well defined part of the system.
>
> Test case: A test case is a definition of input values and expected output values for the unit under test. (kan være manuel/automatisk)

2.4 Trin

1. Identify dimensions in the input space that can be partitioned, and define this partitioning.

   > Equivalence class: A subset of all possible inputs to the UIT that has the property that if one element in the subset demonstrates a defect during testing then we assume that all other elements in the subset will demonstrate the same defect.

Vores ævkivalensklasser skal have egenskaberne

- Coverage (skal dække hele input-rummet)
- Representation (en fejl fundet i en ækvivalensklasse, skal også findes i alle andre elementer i ækvivalensklassen)
- Disjointness (intet element må være i to ækvivalensklasser)

1. Use the partitioning to establish the equivalence classes, and use heuristics to pick the equivalence classes with highest probability of detecting defects.
2. Generate test cases from the selected ECs.

2.5 Partioneringsregler

- Range of values: Three partitions: A) within range B) above range and C) below range.
- Set of values: a partition for each member + one for nonmembers.
- A must be situation (a requirement on input): two partitions: A) requirement fulfilled and B) requirement not fulfilled.

- Any reason to question representation of a partition leads to the partition being split!

Paritioneringstabel: dimension | ugyldig partition | gyldig partition

Myers: Gyldige = så mange som muligt. Ugyldige = en af gangen (masking)

"Test case"-tabel: ækvivalensklasse | test case | forventet output

2.6 Boundary Analysis

- Range of values: Define test cases for the ends of the range and just beyond.
- Numbered values: Define test cases for one above and one beneath maximum and minimum values. Example: 1-255 would be 0, 1, 255, and 256
- Ordered sets: focus on the first and last element i.e. are they ordered properly?

3 Frameworks

Framework: A set of cooperating classes that make up a reusable design for a specific class of software. [GoF]

Hotspot: A code point where specialization code can alter behavior or add behavior to a framework.

3.1 Customization techniques

- Subclassing an abstract class or an interface
- Configuration/composition of existing classes
- Parameterization

Inversion of control: the framework defines the flow of control, not you. Klassebibliotek vs. framework

3.2 Typer

- Whitebox: Udvide ved nedarvning/implementation af interfaces (JHotDraw)
- Blackbox: Udvide ved komposition af eksiterende klasser (Swing)

3.3 Template pattern

- Unification: Template + hook i samme klasse, ergo nedarvning (whitebox)
- Seperation: Template + hook i hver deres klasse, ergo komposition (blackbox)

Framework komposition er bedst med blackbox, eg. GoF princip 1.

4 Design for variability management / Principles of flexible design

4.1 Design techniques

- Parametrisk løsning
- Polymorfi
- Kompositionelt design

4.2 Parametrisk løsning

- + relativt simpelt
- + ingen multiple maintenance
- - ALTID change by modification
- - code bloat / switch creep
- - ansvarsområde rykket

4.3 Polymorfi

- + INGEN multiple maintenance
- + Change by addition
- + INGEN code bloat / switch creep
- - mange MANGE klasser
- - eneste nedarvning brugt på én dimension af varians
- - genbrug af varianter problematisk
- - compile-time binding

4.4 Kompositionelt design

- + change by addition
- + runtime binding
- + tests kan sepereres
- + et ansvarsområde DELEGERES UD
- + variant vælges ét specifikt sted
- + vi bruger ikke nedarvning, så vi kan let tilføje andre varianspunkter
- - flere klasser, dog ikke så mange som med polymorfi (i længden)
- - klienter skal kende til strategierne, som skal udvælges

4.5 Gang of Four principper (3-1-2)

- 3. Consider what should be variable in your design
- 1. Program to an interface, not an implementation
- 2. Favor object composition over class inheritance

... eller alternativt ...

- 3. Identify some behavior that is likely to change
- 1. State well-defined responsitiblity that covers this behavior and express it as an interface
- 2. Instead of implementing the behavior ourselves, we deletagate to an object implementing the interface

4.6 Første princip (program to an interface)

- + Klienter kan bruge en /hvilken som helst/ serviceudbyder
- + Partielle klasser giver ikke mening. Interfaces giver mere fin-kornet kontrol over abstraktioner
- + Interfaces udtrykker roller bedre

4.7 Andet princip (favor object composition)

- + Indkapsling
- + Compile-time vs. run-time binding
- + Kbe hele pakken vs. fin-kornet kontrol over abstraktioner
- + Seperat (unit)test
- + Mere sandsynligt/nemmere at genbruge kode
- - Flere klasser/objekter

Combinatorial explosion. Design patterns. Frameworks.

5 Design patterns

> Design pattern: Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

5.1 Pattern-skabalon

- navn
- problem
- løsning
- konsekvenser
- (+ form & roller)

> A design pattern is defined by a *set of roles*, each role having a specific set of responsibilities, and by a well defined protocol (interaction pattern) between these roles.

Patterns som kommunikationsmiddel? Annotering? Pattern roadmap?

5.2 Strategy

Intent: Define a family of business rules or algorithms, encapsulate each one, and make them interchangible. Strategy lets the algorithms vary independantly from the clints that use it.

Problem: Ens produkt skal supportere variable algoritmer eller "business rules" og give dig en fleksibel og pålidelig måde at kontrollere variabiliteten.

Roles:

- Strategy: Specificerer ansvar og interface for algoritmen.
- ConcreteStrategy: Definrerer konkret opførsel, opfylder ansvaret givet i Strategy.
- Context: Udfører dets arbejde for Client ved at delegere til en instans af Strategy.

Cost - Befinit:

- Strategier eleminerer 'conditional statements'.
- Det er et alternativ til nedarvning.
- Der er seperat testning

5.3 State

Intent: Allow an object to alter it's behavior when it's internal state changes. The object will appear to change it's class.

Problem: Et objekts opførsel afhænger af dets stadie. Disse stadier har meget kompleks opførsel associeret.

Roles:

- Context: Definerer et interface interessant for klienter, og indeholder en instans af ConcreteState, som indeholder det nuværende stadie.
- State: Definerer et interface som indkapsler opførslen associeret med et bestemt stadie af Context.
- ConcreteState: Implementation af State.

Cost - Benifit:

- Lokaliserer stadie-specifik opførsel og partionerer opførsel for forskellige stadier.
- Stadietransitioner er mere eksplicitte, når hvert stadie har sin egen klasse.

5.4 Abstract factory

Intent: Provide an interface for creating families of related or dependant objects without specifying their concrete classes.

Problem: Famillier af relaterede objekter skal instantieres. Produktvarianter skal konfigureres konsistent.

Roles:

- Abstract Factory: Definerer et fælles interface for objekt-instantiering.
- Product: Definerer et interface for et objekt.
- ProductY: Variant Y af Product.
- ConcreteFactoryY: Ansvarlig for at konfigurere ProductY.

Cost - Benefit:

- + Lavere kopling mellem klient og produkter (ingen new i produkt)
- + Let at konfigurere varianter af produkter
- + Øger konsistents mellem produkter (al instantskode samlet)
- - Svært at tilføje nye slags produkter (opdatere alle factories)

5.5 Decorator

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem: YOu want to add responsibility and behavior to individual objects without modifying it's class.

Roles:

- Component: Definerer et interface til en abstraktion
- ConcreteComponent: er implementation heraf
- Decorator: Tilføjer opførsel til Component.

Cost - Benifit:

- + Tilføje og fjerne ansvar på objekter på run-time
- + Inkrementalt tilføje ansvarsområder i udviklingsprocessen
- + Kompleks opførsel via kædning af decoraters
- - Du kan ende op med mange små objekter der ikke gør meget (kæde svær at forstå)
- - Delegering er lidt besværlig at skrive

5.6 Template method

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Problem: ???

Roles:

- AbstractClass: Definerer abstrakte *primitive operationer* som den konkrete subklasse definerer for at implementerer algoritmen. Implementerer også en *template metode* som definerer skelettet af en algoritme. Template metoden kalder primitive operationer såvel som operationer defineret i AbstractClass eller andre objekter.
- ConcreteClass: Implementerer de primitive operationer som skal bruges i algoritmen.

Cost - Benifit:

- Ikke-varierende kode bliver genbrugt i template metoden.
- Inversion of control ala. frameworks.

5.7 Facade

Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higer-level interface that makes te subsystems easier to use.

Problem: The complexity of a subsystem should not be exposed to clients.

Roles:

- Facade: Definerer et simpelt interface til et subsystem.
- Client: tilgår kun systemet via Facade.

Cost - Benifit:

- + Beskytter klienter fra subsystemets objekter
- + Hjælper lav kopling mellem klienter og subsystemet.
- - Facade kan svulme op med mange metoder, for at klienterne kan tilgå alt i subsystemet.

5.8 Observer

Intent: Define a one-to-many depedancy between objects so that when one object changes state, all it's dependancies are notified and updated automatically.

Problem: Der ønskes konsistents mellem samarbejdende klasser uden tæt kopling, da dette medfører at det bliver sværere at genbruge disse klasser.

Roles:

- Subject: Kender sine observere. Et hvilket som helst antal observere kan observere subject. Definerer også et interface så observere kan tilføjes/fjernes.
- Observer: Definerer et opdaterende interface for objekter som ønsker at blive underrettet når der er ændringer i subject.
- ConcreteSubject & ConcreteObserver: implementationer af ovenstående.

Cost - Benifit:

- Abstract kopling mellem subject og observer
- Understøtter broadcast kommunikation (modtager ligegyldig)
- Små ændringer i subject kan medføre kaskade af opdateringer i observere (dyrt).

6 Metrics for Software Architecture

Maintainability: Maintainability is a measure of the cost of introducing change in software.

Coupling: Coupling is a measure of how strongly dependent one software /unit/ is on other software units.

Cohesion: Cohesion is a measure of how strongly related and focused the responsibilities of a software /unit/ is.

- lokalisering
- kocenptuel integritet (at gøre den samme ting, på den samme måde)

Law of Demeter: Snak ikke med fremmede

6.1 Kvalitative attributter

Forskellige grene af datalogien bruger forskellig terminologi, og snakker derfor dårligt sammen. For at stille krav til software, er det nødvendigt at have fælles terminologi og en fælles måde at måle kvalitative attributter.

Opstiller scenarier for kvalitative scenarier (s.75)

- Kilde for stimulus
- Stimulus
- Miljø
- Artifakt
- Respons
- Reponsmåling (responstid)

Kan gøres for bl.a.

- Tilrådighed / availability
- Hvor svært det er at ændre et program / modifiability
- Hvor 'hurtigt' et system er / performance
- Sikkerhed / security
- Hvor let det er at teste et system / testability
- Brugbadhed / usability

Findes også mere økonomiske attributter

- Time to market
- Cost and benifit
- Projected lifetime of the system
- Targeted market
- Rollout schedule
- Integration with legacy systems

7 Behaviour, Responsibility and Roles

Findes flere syn på objekt-orientering...

- Sprogcentrisk
- Modelcentrisk
- Ansvarscentrisk

Behavior $<$ Reponsibility $<$ Roles

> Role: (General) A function or part performed especially in a particular operation or process

This definition embody the dual requirement: both function performed (responsibilities) as well as in a particular process (collaboration pattern/protocol).

> Protocol: A convention detailing the expected sequence of interactions or actions expected by a set of roles.
>
> Role: (Software) A set of responsibilities and associated protocol with associated roles

This leads to a concept that we may term *contractual responsibility* which is the entry point from the outside; and *sub responsibilities* which are the simpler tasks. The deliver responsibility of the florist is actually the contractual responsibility relying on subresponsibilities of another florist and a delivery person.