dSoftArk Noter

Michael Lind Mortensen, 20071202, DAT4

23. januar 2009

Indhold

1 Test-Driven Development

1.1 Concept Overview

Følgende emner har relevans for Test-Driven Development.

- Testing, Test Case, Failure, Defect, Unit under test *(s. 9-10)*
- TDD Rythm *(s. 42-61)*
- TDD Principles *(s. 42-61)*
- Refactoring *(s. 51)*
- Unit Testing *(s. 136)*
- Integration Testing *(s. 136)*
- Depended-on unit (DOU) *(s. 181)*
- Direct-/Indirect input *(s. 181)*
- Test stubs *(s. 183)*

1.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

1.2.1 Definitions

Def. Testing
Testing is the process of executing software in order to find failures.

Def. Failure
A failure is the situation in which the behavior of the executing software deviates from what is expected.

Def. Defect
A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented.

Def. Test case
A test case is a definition of input values and expected output values for the unit under test.

Def. Unit under test (UUT)
The unit under test is some part of the system that we consider to be a whole.

Def. Unit Testing
Unit testing is the process of executing a software unit in isolation in order to find failures in the unit itself.

Def. Integration Test
Integration testing is the process of executing a software unit in collaboration with other units in order to find failures in their interactions.

Def. Refactoring
Refactoring is the process of changing a software system in such a way, that it does not alter the external behavior of the code, yet improves it's internal structure.

Def. Direct input
Direct input is values or data, provided directly by the testing code, that affect the behaviour of the unit under test (UUT).

Def. Indirect input
Indirect input is values or data, that cannot be provided directly by the testing code, that affect the behaviour of the unit under test (UUT).

Def. Depended-On Unit (DOU)
A unit in the production code that provides values or behaviour that affect the behaviour of the unit under test.

NOTE: Dette relaterer sig til Test-stubs, hvor DOU'en f.eks. er et ur eller en random generator der i produktionen giver input der varierer.

Def. Test Stub
A test stub is a replacement of a real *depended-on unit* (DOU) that feeds indirect input, defined by the test code, into the *unit under test* (UUT).

1.2.2 TDD Rythm

1. Quickly add a test
2. Run all tests and see the new test fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication
6. *(Run all tests again to check refactoring!)*

Så altså man tilføjer en test i forhold til ens testlist, kører testen og ser den fejler (for at sikre man rent faktisk har en test der tester noget), laver den mindst mulige ændring i produktionskoden for at det virker, kører tests igen for at se at det virker og refaktorerer derefter for at få clean code. Sidst men ikke mindst burde man, omend det ikke er en del af TDD Rythm, køre testene igen så man ved ens refaktorering ikke ødelagde noget.

1.2.3 TDD Principles

TDD Principperne minder lidt om patterns, men er egentlig mere regler for hvorledes vi udfører TDD på en effektiv måde.

Test First: When should you write your tests? Before you write the code that is to be tested.

Test List: What should you test? Before you begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests.

One Step Test: Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

Isolated Test: How should the running of tests affect one another? Not at all.

Evident Tests: How do we avoid writing defective tests? By keeping the testing code evident, readable and as simple as possible.

Fake It (Till You Make It): What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

Triangulation: How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

Assert First: When should you write the asserts? Try writing them first.

Evident Data: How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

Obvious Implementation: How do you implement simple operations? Just implement them.

Representative Data: What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or special computational processing.

Automated Test: How do you test your software? Write an automated test.

Test Data: What data do you use for test-first tests? Use data that makes the tests easy to read and follow. If there is a difference in the data, then it should be meaningful. If there isn't a conceptual difference between 1 and 2, use 1.

Child Test: How do you get a test case running that turns out to be too big? Write a smaller test case that represents the broken part of the bigger test case. Get the smaller test case running. Reintroduce the larger test case.

Regression Test: What is the first thing you do when a defect is reported? Write the smallest possible test that fails and that, once run, will be repaired.

Test Data og **Representative Data** ligger meget tæt op af hinanden og burde egentlig have en ekstra tilknytning: **Boundary Values:** Vælg værdier op af, og på den anden side af, grænserne for hvad der er "lovligt" i henhold til test casen.
Desuden skal siges, at hvis man bruger **Fake It**, bør man efter brugen skrive

en ny test til ens test list. Denne test skal således tvinge implementationen af Fake It til at blive fuldendt. Et eksempel på sådan en fremgangsmåde er *Triangulation*, hvor man udvider kravet til algoritmen, således det at returnere en konstant ikke bliver godt nok.

Der er to andre vigtige principper, som dog er mere relateret til generel udvikling end blot lige til TDD (til trods for hvad bogen siger):

Break: What do you do when you feel tired or stuck? Take a break.

Do Over: What do you do when you are feeling lost? Throw away the code and start over.

Break spørger han uden tvivl om til eksamen.

1.2.4 Unit-/Integration Testing

Vi kan ikke generelt regne med ingen fejl i delene af et system betyder ingen fejl i hele systemet. Derfor laver vi både unit testing og integration testing, som herunder:

```
1  @Test
2  public void shouldIntegrateProgressiveRateCorrectly()
3      throws IllegalCoinException {
4      // reconfigure ps to be the progressive rate pay station
5      ps = new PayStationImp( new ProgressiveRateStrategy() );
6      // add $2.: 1.5 gives 1 hours and next 0.5 gives 15 min
7      addOneDollar(); addOneDollar();
8
9      assertEquals( "Progressive Rate: 2$ should give 75 min",
10             75, ps.readDisplay() );
11 }
```

1.2.5 Test Stubs

En test stubs bruges til at kunne lave automatiske tests af Units under test der kræver input fra "upålidelige" kilder som et ur eller lignende. Sagt på en anden måde, en Unit under test (UUT) har en Depended-On Unit (DOU) der bidrager med indirekte input fra non-konsistente kilder.

2 Systematic Blackbox Testing

2.1 Concept Overview

Følgende emner har relevans for Systematic Blackbox Testing.

- Testing, Test Case, Failure, Defect, Unit under test *(s. 9-10)*
- No testing, Explorative or Systematic *(Systest s. 2)*
- Blackbox-/Whitebox Testing *(Systest s. 1)*
- Equivalence Classes, Soundness, Coverage, Representation and Disjointness *(Systest s. 2-3)*
- Partioning Heuristics *(Systest s. 3)*
- EC Table *(Systest s. 4)*
- Myers rules *(Systest s. 3)*
- Test Case Table *(Systest s. 5)*
- Conditions/Parameters *(Systest s. 5-6)*
- Boundary Value Analysis *(Slides uge 6)*

2.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

2.2.1 Definitions

Def. Testing
Testing is the process of executing software in order to find failures.

Def. Failure
A failure is the situation in which the behavior of the executing software deviates from what is expected.

Def. Defect
A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented.

Def. Test case
A test case is a definition of input values and expected output values for the unit under test.

Def. Unit under test (UUT)
The unit under test is some part of the system that we consider to be a whole.

Def. Black-box testing
The UUT is treated as a black box. The only knowledge we have to guide our testing is the specification of the UUT and a general knowledge of common programming techniques and algorithmic constructs.

Def. White-box testing
The full implementation of the UUT is known, so the actual code can be inspected in order to generate test cases.

Def. Equivalence class (EC)
A subset of all possible inputs to the UUT, 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.

Def. Soundness
For a set of equivalence classes to be *sound*, the ECs most uphold the criteria of Coverage, Representation and Disjointness.

Def. Coverage
Every possible input belongs to one of the equivalence classes.

Def. Representation
If a failure is demonstrated on a particular member of an equivalence class, the same failure is demonstrated by any other member of that class.

Def. Disjointness
No input belongs to more than one equivalence class.

Myers rule for valid ECs
Until all valid partitions have been covered, define a new EC (and test-case) covering as many uncovered valid partitions as possible.

Myers rule for invalid ECs
Until all invalid partitions have been covered, define a new EC (and test-case) that covers one, and only one, of the uncovered invalid partitions.

2.2.2 Testing Approaches

No testing: Meget små metoder kan ikke betale sig at teste, da deres funktionalitet er så simpel at testkoden bliver større end produktionskoden. Dette drejer sig f.eks. om getters/setters.

Explorative testing: Explorative testing går ud på at bruge intuition og erfaring til at teste. Denne stil kan være meget nyttig ved middelkomplekse metoder og er bl.a. også den mest dominante stilart bag TDD.

Systematic testing: Systematisk testing bruges når vi har at gøre med meget komplekse metoder, eller metoder med en meget lav fejltolerance (software til rumfaretøjer f.eks.). Her finder vi, på systematisk vis, ækvivalensklasser for de conditions der påvirker systemet og bygger vore test-cases op på baggrund af disse.

2.2.3 Partioning Heuristics

Generelt skal der egentlig blot partioneres hver gang man er i tvivl om Representationsprincippet er overholdt. Dette er f.eks. hvis en værdi mellem $0-7$ tillægges en anden condition, så bør man skelne mellem tests der bruger 0 som input og tests der bruger $1-7$ som input, da tests med 0 ikke vil afsløre hvis tillæggelsen aldrig finder sted.

Ved en langsom og metodisk fremgangsmåde burde det være muligt at få et minimalt sæt test cases der tager alle aspekter i betragtning.

2.2.4 EC Table

Et eksempel på en EC tabel ses her:

Condition	Invalid ECs	Valid ECs
year	< 1900 [1]; > 3000 [2]	$1900 - 3000$ [3]
month	< 1 [4]; > 12 [5]	$1 - 12$ [6]

2.2.5 Test Case Table

Et eksempel på en test case table ses her:

ECs covered	Test case	Expected output
$[3a], [6a]$	$y = 2000; m = 2$	-
$[3b], [6b]$	$y = 1900; m = 5$	-
$[3c], [6b]$	$y = 2004; m = 10$	5
$[3d], [6a]$	$y = 1985; m = 1$	-
$[1]$	$y = 1844; m = 4$	[exception]
$[2]$	$y = 4231; m = 8$	[exception]
$[4]$	$y = 2004; m = 0$	[exception]
$[5]$	$y = 2004; m = 13$	[exception]

2.2.6 Myers Rules for Valid/Invalid ECs

1. Until all **valid** ECs have been covered, define a test case that covers as many uncovered **valid** ECs as possible.

2. Until all **invalid** ECs have been covered, define a test case whose element only lies in a single **invalid** ECs.

2.2.7 Conditions/Parameters

Det er ikke nok at kigge på parametrene til den metode man tester. Man er nød til at kigge på conditions fremfor parametre. Eksempel:

```
1 public class PrettyStupid {
2     private int T;
3
4     // return true iff x+y < T
5     public boolean isMoreThanSumOf(int x, int y)
6     {
7         return (x+y < T);
8     }
9 }
```

2.2.8 Boundary Value Analysis

Boundary Value Analysis går ud på vælge test data på begge sider af grænserne mellem ækvivalensklasser. Således hvis en valid EC range går fra $1-5$, så ville man teste ved 0, 1, 5 og 6.
Kombineres med systematisk testing, men bruges egentlig også indirekte ved TDD når man vælger test data.

3 Frameworks

3.1 Concept Overview

Følgende emner har relevans for Frameworks.

- Framework definition *(s. 339)*
- Framework Characteristics *(s. 340)*
- Compositional Design: Delegation *(s. 116)*
- Framework Code, Framework Customization Code & Non-Framework code *(s. 342)*
- Variability Points *(s. 342)*
- Customization techniques *(s. 343)*
- Inversion of Control *(s. 343)*
- Template Method *(s. 345)*
- Unification/Separation *(s. 346)*
- Blackbox-/Whitebox framework *(s. 344)*
- Framework Composition *(s. 347)*

3.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

3.2.1 Definitions

Def. Framework
A framework is a set of cooperating classes that make up a reusable design for a specific class of software.

Def. Delegation
Delegation is when the behavior associated with some responsibility of an object is completely or partially handled by a subordinate object, called the delegate.

Def. Variability points
Code point where specialisation code can alter behaviour or add behaviour to a framework.

NOTE: Desuden kendt som "hooks" eller "hotspots"

Def. Inversion of Control
The framework dictates the protocol - the customization code just has to obey it!

NOTE: "Hollywood principle": Don't call us, we will call you.

Def. Unification
Both template and hook methods reside in the same class. The template method is concrete and invokes an abstract hook method that can be overridden in subclasses.

Def. Separation
The template method is defined in one class and the hook methods are in another class. The template method is concrete and delegates to hook methods via an object reference.

Def. Blackbox framework
A blackbox framework is a framework that is customized using composition, i.e. to define the specific behavior of the framework, you specify a composition of predefined instances.

Def. Whitebox framework
A whitebox framework is a framework that is customized using inheritance, i.e. to define the specific behavior of the framework, you subclass abstract classes in the framework/implement interfaces, and give these concrete class instances to the framework to use.

Def. Software Product Line
A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of

core assets in a prescribed way.

3.2.2 Framework definition

Ifølge Gang of Four (GoF):

Def. Framework
A framework is a set of cooperating classes that make up a reusable design for a specific class of software.

Altså er frameworks et spørgsmål om code reuse.

3.2.3 Framework characteristics

Der er en række karakteristika ved et framework. Disse er:

Skeleton Design: Et framework leverer applikationsadfærd på højt abstraktionsniveau.

Class of Software: Et framework er designet til et givent domæne og kan bruges til at skabe produkter indenfor det domæne.

Cooperating classes: Et framework definerer interaktionen mellem et veldefineret sæt af komponenter, således brugeren skal kende til disse patterns for at kode til frameworket.

Customizable: Et framework can customizes til en bestemt kontekst indenfor problemdomænet.

Implementation: Et framework er genbrug af fungerende kode og genbrug af design.

3.2.4 Framework komponenter

En applikation udviklet vha. et framework kan løst blive delt i følgende 3 dele:

1. Framework koden, som er givet på forhånd.
2. Customization koden, som programmøreren skriver for at tweake frameworkets funktionalitet til en given kontekst.
3. Non-Framework kode, hvilket er andre funktionalitetsaspekter frameworket ikke er ment til at styre.

Non-framework kode kan f.eks. være hvis man bruger Minidraw til at udvikle et spil, så er koden der bruges til at styre moves, attacks osv. ikke en del af Minidraw frameworket og derfor er det, fra dennes synspunkt, Non-framework kode.

3.2.5 Variability Points

Er defineret som:

Def. Variability points
Code point where specialisation code can alter behaviour or add behaviour to a framework.

Desuden kendt som "hooks" eller "hotspots". Der er adskillige måder at implementere sådanne variability points på.

3.2.6 Customization Techniques

Der er 4 standard metoder til at implementere variability points. Disse er:

Subclassing: Bruges til frameworks der indeholder abstrakte klasser. Du skal således extende disse abstrakte klasser og frameworket bruger så disse implementerede klasser som customization.

Interfaces: Samme princip som ved subclasses, bortset fra der her ikke er nogen implementation overhovedet. Så det eneste frameworket definerer her er design og ikke kode.

Composition: Frameworket kommer med et antal predefinerede klasser og det er så dit job at bruge composition til at implementere variability points. Både AWT og Swing bruger denne.

Parameterization: Frameworket har en sat funktionalitet, hvor du så sætter parametre for at manipulere variability punkterne.

3.2.7 Inversion of Control

Frameworks adskiller sig fra simple libraries ved at Clienten kalder library metoder, hvorimod frameworket kalder metoder som Client har defineret. Der sker derved en Inversion of Control!
Det eneste Client gør er at definere variability points af forskellig art, hvorefter frameworket vil kalde disse kodestumper når det passer ind i udførslen. Et framework bygger således på Hollywood princippet:

Don't call us - We'll call you!

3.2.8 Template Method

Template Method er hele grundlaget i hvorledes Frameworks fungerer. Template Method Pattern er dog oprindeligt et inheritance pattern, men kan ændres til et compositionelt pattern.
Basalt set: Definer et skelet af funktionalitet, men efterlad visse del til at blive customized af andre.

Intent Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure.

Problem Some steps in the algorithm are fixed but some steps I would like to keep open for future modifications to the behaviour.

Solution Can be implemented by Unification (Subclassing) or Separation (Composition).

Template Method kan blive implementeret på to forskellige måder:

3.2.9 Unification/Separation

Def. Unification
Both template and hook methods reside in the same class. The template method is concrete and invokes an abstract hook method that can be overridden in subclasses.

Def. Separation
The template method is defined in one class and the hook methods are in another class. The template method is concrete and delegates to hook methods via an object reference.

Så basalt set:

$$Unification = Inheritance$$
$$Separation = Composition$$

Blackbox-/Whitebox Framework Overordnet har vi to forskellige slags frameworks:

Def. Blackbox framework
A blackbox framework is a framework that is customized using composition, i.e. to define the specific behavior of the framework, you specify a composition of predefined instances.

Def. Whitebox framework
A whitebox framework is a framework that is customized using inheritance, i.e. to define the specific behavior of the framework, you subclass abstract classes in the framework/implement interfaces, and give these concrete class instances to the framework to use.

HotCiv er derved et Blackbox framework.

3.2.10 Framework Composition

Hvis man ønsker at koble mere end et framework op omkring sin kode, så skal man også være opmærksom på, at valget af en Unification Template Method gør, at dette ikke kan lade sig gøre i Java eller C#, da ingen af disse kan lave multiple inheritance.
Med en Separation Template Method er det dog intet problem, da man sagtens kan implementere flere interfaces.

4 Design for Variability Management

4.1 Concept Overview

Følgende emner har relevans for Design for Variability Management.

- Flexibility, Maintainability, Analysability, Changeability, Stability & Testability *(s. 26, dSoftArk Compilation og Slides uge 6)*
- Change by modification/Change by addition *(s. 106 og 113)*
- Code bloat, Switch creep & Feature creep *(s. 108)*
- Variability point *(s. 101)*
- Source code copy *(s. 102)*
- Parametric solution *(s. 104)*
- Polymorphic solution *(s. 111)*
- Compositional solution *(s. 115)*
- Delegation *(s. 116)*
- The 3-1-2 process *(s. 170)*
- Strategy Pattern *(s. 125)*
- Abstract Factory *(s. 203)*

4.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

4.2.1 Definitions

Def. Change by modification
Change by modification is when software changes are introduced by modifying existing production code.

Def. Code bloat
Code bloat is the production of code that is perceived as unnecessarily long, slow or otherwise wasteful of resources.

Def. Switch creep
Switch creep is the tendency that conditional statements become more and more complex as software ages.

Def. Change by addition
Change by addition is when software changes are introduced by adding new production code instead of modifying existing.

Def. Flexibility
The capability of the software product to support added/enhanced functionality purely by adding software units and specifically not by modifying existing software units.

Def. Maintainability (ISO 9126)
The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

Def. Analysability (ISO 9126)
The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be indentified.

Def. Changeability (ISO 9126)
The capability of the software product to enable a specified modification to be implemented.

Def. Stability (ISO 9126)
The capability of the software product to avoid unexpected effects from modifications of the system.

Def. Testability (ISO 9126)
The capability of the software product to enable modified system to be validated.

4.2.2 Variability Point

Et variability point er en sektion af kode der skal variere. Vi bruger dette når vi skal lave en eller anden form for reuse, enten i form af ekstra krav til funktionaliteten (e.g. algoritmen skal være anderledes hver lørdag nat) eller hvis nye kunder vil have samme produkt med en lille ændring.

4.2.3 Source Code Copy

Den groveste løsning er at lave direkte source code copy og så kun ændre det lille sted hvor variationen er ønsket. Fordele og ulemper er:

- Fordele

  Det er simpelt!

  Det er hurtigt!

  Decoupler varianter fuldstændig!

- Ulemper

  "Multiple Maintenance Problem!"

  At rette fejl fundet i gammel kode er svært (det skal rettes i alle varianter).

4.2.4 Parametric Solution

En anden løsning er at styre varianterne vha. If eller Switch statements i koden. Fordele og ulemper er her:

- Fordele

  Det er simpelt!

  Intet "Multiple Maintenance Problem"

- Ulemper

  "Change by modification" (stor risiko for at introducere fejl i gammel kode)

  Code bloat (Hvis der er 43 varianter bliver det meget uoverskueligt)

  Switch creep (if's skaber nye if's, svært at læse!)

  Feature creep (Reponsibility: "Håndter variants" opstår)

4.2.5 Polymorphic Solution

En tredje løsning er at styre varianter ved at extende klasser og override de metoder man ønsker. Fordele og ulemper er her:

- Fordele

  Intet "Multiple Maintenance Problem"

  Kun "Change by modification" første gang, derefter "Change by addition" for hver subclass

  Nemt at læse

- Ulemper

  Forhøjet antal klasser

  Inheritance kan kun bruges en gang

  Genbrug henover varianter er svært

  Objecter bindes på compile-time (Variant kan ikke ændres på run-time)

4.2.6 Compositional Solution

En fjerde og foretrukken løsning er at styre varianter ved at delegere varianten til sin egen abstraktion i form af et interface, der composeres sammen med produkterne. Fordele og ulemper er her:

- Fordele

  Nemt at læse

  Run-time binding

  Separation af responsibilities

  Separation af testing

  Variant valg er lokaliseret

  Inheritance mulig hvis behovet opstår

- Ulemper

  Forhøjet antal interfaces og klasser

  Brugere må vide gøres bekendt med løsningen

4.2.7 The 3-1-2 process

3-1-2 processen er en generel opskrift på at håndtere variationer på forskellig vis. Den lyder således:

3. Identificer en operation der vil variere

1. Skab en responsibility der omfatter denne operation og udtryk den i et interface.

2. Udfør nu denne operation ved at delegere opgaven til et underordnet objekt der implementerer interfacet.

4.2.8 Strategy Pattern

Strategy pattern er et af de bedste patterns til at håndtere variationer.

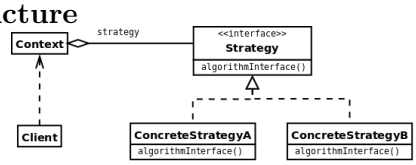
STRATEGY

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controling the variability.

Solution Separate selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithms realize this interface.

Structure



Roles

Strategy: Specifies the responsibility and interface of the algorithm.
ConcreteStrategy: Defines concrete behaviour fulfilling the responsibility.
Context: Performs the work for Client by delegating to a Strategy.

Cost/Benifit

Pros:
- Strategies eliminate conditional statements.

- Avoids subclassing.
- Avoids multiple maintenance.
- Change by addition, not by modification.
- Makes separate testing of Context and Strategy possible.

Cons:

- Increased number of objects.
- Clients must be aware of strategies.
- Mild resource loss by indirection.

4.2.9 Abstract Factory

Abstract Factory kan bruges til at samle konstruktionen af varianter et sted og styre det ved at lade factories samle komponenter til et samlet produkt.

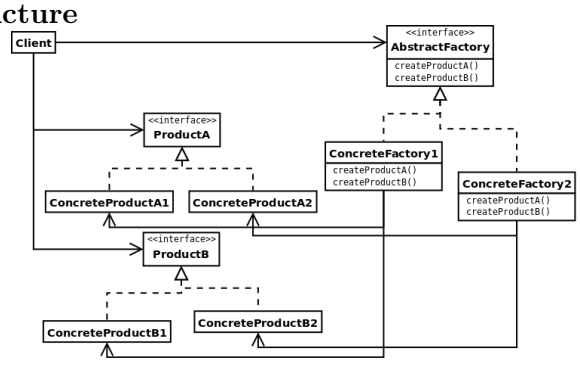
ABSTRACT FACTORY

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

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure



Roles

Abstract Factory: Defines a common interface for object creation.
ProductX: Defines the interface of an object ConcreteProductXY (product X in variant Y)
ConcreteFactoryY: Is responsible for creating Products that belong to a variant Y family of objects that are consistent with each other.

Cost/Benifit

Pros:

- Lowers coupling between client and products
- Makes exchanging product families easy
- Promotes consistency among products
- Change by addition, not by modification
- Client constructor parameter list stays intact

Cons:

- Introduces extra classes and objects
- Supporting new aspects of variation is difficult

5 Design Patterns

5.1 Concept Overview

Følgende emner har relevans for Design Patterns.

- The 3-1-2 Process *(s. 170)*
- Design Pattern Komponenter *(s. 212)*
- Roles, Responsibilities & Protocols *(s. 228-232)*
- Pattern Fragility *(s. 216)*
- Coupling/Cohesion *(s. 150-153)*
- Law of Demeter *(s. 154)*
- Alle patterns *(Patterns sektionen i noterne)*

5.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

5.2.1 Definitions

Def. Software Architecture
The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships among them.

Def. Design Pattern
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.

Def. Design Pattern (Gamma et al.)
Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Def. Design Pattern (Beck et al.)
A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future.

NOTE: "prose form" betyder template og denne template indeholder som min. altid: Name, Problem, Solution og Consequences.

Def. Pattern Fragility
Pattern fragility is the property of design patterns, that their benefits can only be fully utilized if the pattern's object structure and protocol is implemented correctly.

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

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

Law of Demeter
Do not collaborate with indirect objects.

NOTE: Also called "Don't Talk to Strangers"

Def. Object Orientation (Nygaard m.fl.)
A program execution is viewed as a physical model simulating the behaviour of either a real or imaginary part of the world.

NOTE: Desuden kendt som Model-centric focus

Def. Object Orientation (Budd)
An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community.

NOTE: Desuden kendt som Responsibility-centric focus

Def. Responsibility
The state of being accountable and dependable to answer a request.

Def. Protocol
A convention detailing the expected sequence of interactions or actions expected by a set of roles.

Def. Role (Software)
A set of responsibilities and associated protocol with associated roles.

5.2.2 The 3-1-2 Process

3-1-2 processen er en generel opskrift på at håndtere variationer på forskellig vis. Den lyder således:

3. Identificer en operation der vil variere

1. Skab en responsibility der omfatter denne operation og udtryk den i et interface.

2. Udfør nu denne operation ved at delegere opgaven til et underordnet objekt der implementerer interfacet.

5.2.3 Design Patterns Komponenter

Design patterns defineres på flere forskellige måder, men den vi følger her er:

Def. Design Pattern
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.

Vi bruger patterns til at implementere forskellige "best-practice" designkomponenter for at opnå en eller anden Quality (typisk fleksibilitet og maintainability).

Et design pattern er typisk beskrevet ved minimum 4 ting:

1. Et navn (Name)
2. Et problem patternet løser (Problem)
3. En løsning (Solution)
4. Konsekvenserne af løsningen (Consequences)

I bogen indeholder de fleste patterns også en betegnelse af roller (Roles), et UML diagram (Structure) og en hensigt med patternet (Intent).

5.2.4 Roles, Responsibilities & Protocols

Patterns er bygget op af Roles, Responsibilities for disse Roles og deres interne kommunikation (Protocols). Lad os definere disse tre termer:

Def. Role (Software)
A set of responsibilities and associated protocol with associated roles.

Def. Responsibility
The state of being accountable and dependable to answer a request.

Def. Protocol
A convention detailing the expected sequence of interactions or actions expected by a set of roles.

UML kan ikke vise roller, så det er derfor vigtigt at vurdere patterns ikke ud fra deres diagrammer, men ud fra deres **Intent** og **Roles** med dertil hørende **Responsibilities** og **Protocols**.

5.2.5 Pattern Fragility

Patterns er en måde at opnå et mål på og ikke et mål i sig selv! Et forkert implementeret pattern kan derfor lede til at man får alle ulemperne ved det pågældende Pattern og ingen af dets fordele.
Pattern Fragility defineres som:

Def. Pattern Fragility
Pattern fragility is the property of design patterns, that their benefits can only be fully utilized if the pattern's object structure and protocol is implemented correctly.

Typiske fejl er:

1. Brug af klassenavne ved deklaration fremfor interfacenavn
2. Binding til klasse på et forkert tidspunkt (i.e. et sted hvor man ikke kan variere klassen)
3. Pga. stress eller deadlines laver man et hurtigt fix til noget variability (typisk med parametre)

6 Behaviour, Responsibilities and Roles

6.1 Concept Overview

Følgende emner har relevans for Behaviour, Responsibilities and Roles.

- Object Orientation - Language centric view *(s. 226)*
- Object Orientation - Model centric view *(s. 227)*
- Object Orientation - Responsibility centric view *(s. 227)*
- Behaviour, Roles, Responsibilities & Protocols *(s. 228-232)*
- Who/What vs. What/Who *(Slides "roles.pdf")*
- Coupling/Cohesion *(s. 150-153)*
- Alle patterns *(Patterns sektionen i noterne)*

6.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

6.2.1 Definitions

Behaviour
Acting in a particular and observable way.

Def. Role (General)
A function or part performed especially in a particular operation or process.

Def. Role (Software)
A set of responsibilities and associated protocol with associated roles.

Def. Responsibility
The state of being accountable and dependable to answer a request.

Def. Protocol
A convention detailing the expected sequence of interactions or actions expected by a set of roles.

Def. Object Orientation (Nygaard m.fl.)
A program execution is viewed as a physical model simulating the behaviour of either a real or imaginary part of the world.

NOTE: Desuden kendt som Model-centric focus

Def. Object Orientation (Budd)
An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community.

NOTE: Desuden kendt som Responsibility-centric focus

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

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

6.2.2 Object Orientation generelt

Der er overordnet 3 forskellige måder at se Objekter på i OO programmering. Disse 3 forskellige tankegange leder så til komplet forskellige designs, men er ikke nødvendigvis i konflikt og bygger egentlig blot på hinanden. Forklaring på de 3 følger:

6.2.3 Object Orientation - Language centric

I den Language centriske tankegang er Objekter defineret som nogle **fields** + nogle **methods**.
Man kigger således ikke på den reele funktionalitet eller hvad objekterne eksisterer for, men i stedet på de reele byggesten, som havde du kigget på

systemet i Emacs eller Eclipse. (eller Vim :-P)

6.2.4 Object Orientation - Model centric

I den Model centriske tankegang er Objekter modeller for virkeligheden. Den fokuserer således på **Concepts** og **Relations**, hvor selve modellen er i et eller andet problem domæne. Et object er således en del af en model, hvor modellen er simulation af virkeligheden.
Den officielle definition på dette syn er:

Def. Object Orientation (Nygaard m.fl.)
A program execution is viewed as a physical model simulating the behaviour of either a real or imaginary part of the world.

På denne måde blev klasser modelleret som "Concepts" og objekter som "Phenomena" af disse "Concepts". Man laver altså en 1 til 1 translation mapping mellem virkeligheden og software.

Denne form for modellering holder dog ikke længe, da objekter i virkeligheden ikke er særlig interessante. Eksempelvis giver det ikke meget mening at kalde funktionen *addMoney()* på et *Account* objekt, da en konto ikke har intelligens til at tilføje penge til sig selv. Man giver altså virkelige fænomener uvirkelige evner og modellerer derfor i virkeligheden intet, men gør blot designet mere uoverskueligt.
Model perspektivets mantra er grundlæggende: **Define the classes, next define their methods**. *(Kaldes nogen gange who/what cycle, som I klasser før funktionalitet)*
Patterns bygger eksempelvis heller ikke på virkelige koncepter, men derimod på Roles, Responsibilities og Protocols.

6.2.5 Object Orientation - Responsibility centric

I den Responsibility centriske tankegang er Objekter service-providers med en række Responsibilities og en eller flere klare Roles. Man har derfor stor fokus på Behaviour fremfor klasser.
Den officielle definition på denne tankegang er:

Def. Object Orientation (Budd)
An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a

service or performs an action that is used by other members of the community.

Man designer således software ud fra et ønske om en given funktionalitet, hvor hver Role har sin Responsibility og intern Behaviour, og hvor en Role så kan modtage en request, som man så kan antage denne imødekommer såfremt det er dennes Responsibility. *(Kaldes nogen gange what/who cycle, i det man først definerer funktionaliteten og herefter definerer en rolle til at håndtere denne.)*

6.2.6 Roles & Responsibilities

I den Responsibility centriske tankegang har klasser bestemte Roles i systemet, en række Responsibilities og Protocols der bestemmer kommunikationen imellem disse komponenter.
Behaviour derimod er de reelle operationer et objekt kan udføre - altså selve den basiske funktionalitet. Responsibilities fungerer så som en abstraktion ovenpå Behaviour, således vi ikke er interesseret i hvordan et subordinate Role udfører et stykke arbejde, blot at den gør det.

OBS: Et given objekt kan sagtens have flere Roles i et system og en Role kan sagtens være fordelt på flere objekter.

7 Principles for Flexible Design

7.1 Concept Overview

Følgende emner har relevans for Principles for Flexible Design.

- Flexibility *(s. 26)*
- GoF principperne *(s. 237)*
- The 3-1-2 Process *(s. 170)*
- Coupling/Cohesion *(s. 150-153)*

7.2 Concept Details

Her gives detaljer på nogle ting, således de hurtigt kan genopfriskes. Disse informationer er dog ikke udtømmende. Konsulter bogen såfremt der ønskes mere.

7.2.1 Definitions

Def. Flexibility
The capability of the software product to support added/enhanced functionality purely by adding software units and specifically not by modifying existing software units.

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

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

7.2.2 GoF principperne

Der er 3 overordnede principper i at lave et fleksibelt design:

1. Program to an interface, not an implementation
2. Favor object composition over class inheritance

3. Consider what should be variable in your design

Det er disse 3 principper, der i bogen er omrokeret til 3-1-2 processen, for at give en operationel måde at gøre sit design fleksibelt.

1. princip
Første princip siger basalt set man bør kode op mod et interface (en kontrakt af responsibilities) fremfor op imod den reele implementation. På den måde kan implementationen ændre sig uden interfacet gør det og vi kan benytte en række patterns vi ellers ikke ville have kunnet.

Hvis dette princip ikke følges opnås **High Coupling**, hvor klienten pludselig bliver afhængig af en operation udføres på en given måde. Herefter kan variation kun opnås ved Inheritance.

Hvis princippet følges derimod, opnås **Low Coupling**, og variationer er nemmere at håndtere, da man eksempelvis nu kan koble sig til third-party kode (Adapter pattern).

2. princip
Dette princip siger grundlæggende der er 2 måder at genbruge kode på: **Composition** og **Inheritance**.
Her anbefales at bruge Composition for at lave fleksible designs, da inheritance har ulemper som:

1. Bindes på compile-time *(man skriver decideret "extends" i sin kode)*
2. Du får hele funktionaliteten som du så tweaker lidt
3. Risiko for at skulle ændre en masse kode *(Kan undgås ved at skabe en Abstract Class med fælles funktionaliet, men så laver man Change by modification)*
4. Inheritance bryder encapsulation *(Da det at ændre i superklassens implementation påvirker alle subclasses)*

Modsat Inheritance har Composition de fordele at:

1. Bindes på run-time
2. Loose coupling da man blot kommunikere med interfacet
3. Klar afgrænsning af Responsibilities
4. Nemmere at teste i isolation *(Da krav står direkte i interfacet - i teorien)*
5. Change by addition

3. princip
Dette princip siger at du skal finde de dele af din kode du forventer vil variere, hvorefter du skal enkapsulere denne del i et passende pattern.

Encapsulate the behaviour that varies!

7.2.3 The 3-1-2 Process

Man bruger så GoF principperne vha. bogens 3-1-2 proces, som beskrevet her:

3. Identificer kode der vil variere

1. Skab en responsibility der omfatter denne kode og udtryk den i et interface.

2. Udfør nu denne kode ved at delegere opgaven til et underordnet objekt der implementerer interfacet.

8 Definitioner

8.1 Testing & TDD

Def. Testing
Testing is the process of executing software in order to find failures.

Def. Failure
A failure is the situation in which the behavior of the executing software deviates from what is expected.

Def. Defect
A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented.

Def. Test case
A test case is a definition of input values and expected output values for the unit under test.

Def. Unit under test (UUT)
The unit under test is some part of the system that we consider to be a whole.

Def. Unit Testing
Unit testing is the process of executing a software unit in isolation in order to find failures in the unit itself.

Def. Integration Test
Integration testing is the process of executing a software unit in collaboration with other units in order to find failures in their interactions.

Def. Refactoring
Refactoring is the process of changing a software system in such a way, that it does not alter the external behavior of the code, yet improves it's internal structure.

Def. Direct input
Direct input is values or data, provided directly by the testing code, that affect the behaviour of the unit under test (UUT).

Def. Indirect input
Indirect input is values or data, that cannot be provided directly by the testing code, that affect the behaviour of the unit under test (UUT).

Def. Depended-On Unit (DOU)
A unit in the production code that provides values or behaviour that affect the behaviour of the unit under test.

NOTE: Dette relaterer sig til Test-stubs, hvor DOU'en f.eks. er et ur eller en random generator der i produktionen giver input der varierer.

Def. Test Stub
A test stub is a replacement of a real *depended-on unit* (DOU) that feeds indirect input, defined by the test code, into the *unit under test* (UUT).

8.2 Variability Management

Def. Change by modification
Change by modification is when software changes are introduced by modifying existing production code.

Def. Code bloat
Code bloat is the production of code that is perceived as unnecessarily long, slow or otherwise wasteful of resources.

Def. Switch creep
Switch creep is the tendency that conditional statements become more and more complex as software ages.

Def. Change by addition
Change by addition is when software changes are introduced by adding new production code instead of modifying existing.

8.3 Architecture & Design

Def. Software Architecture
The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships among them.

Def. Design Pattern
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.

Def. Design Pattern (Gamma et al.)
Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Def. Design Pattern (Beck et al.)
A design pattern is a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future.

NOTE: "prose form" betyder template og denne template indeholder som min. altid: Name, Problem, Solution og Consequences.

Def. Pattern Fragility
Pattern fragility is the property of design patterns, that their benefits can only be fully utilized if the pattern's object structure and protocol is implemented correctly.

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

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

Law of Demeter
Do not collaborate with indirect objects.

NOTE: Also called "Don't Talk to Strangers"

Def. Object Orientation (Nygaard m.fl.)
A program execution is viewed as a physical model simulating the behaviour of either a real or imaginary part of the world.

NOTE: Desuden kendt som Model-centric focus

Def. Object Orientation (Budd)
An object-oriented program is structured as a community of interacting agents called objects. Each object has a role to play. Each object provides a service or performs an action that is used by other members of the community.

NOTE: Desuden kendt som Responsibility-centric focus

Behaviour
Acting in a particular and observable way.

Def. Role (General)
A function or part performed especially in a particular operation or process.

Def. Role (Software)
A set of responsibilities and associated protocol with associated roles.

Def. Responsibility
The state of being accountable and dependable to answer a request.

Def. Protocol
A convention detailing the expected sequence of interactions or actions expected by a set of roles.

8.4 Compositional Design

Def. Delegation
Delegation is when the behavior associated with some responsibility of an object is completely or partially handled by a subordinate object, called the delegate.

8.5 Build Management Systems

Def. Build management
The process of managing and constructing an executable software system from it's parts in a reliable and cost-efficient way.

Def. Build description
A description of the goals and means of managing and constructing an executable software. A build description states *targets*, *dependencies*, *procedures*, and *properties*.

8.6 Frameworks

Def. Framework
A framework is a set of cooperating classes that make up a reusable design for a specific class of software.

Def. Variability points
Code point where specialisation code can alter behaviour or add behaviour to a framework.

NOTE: Desuden kendt som "hooks" eller "hotspots"

Def. Inversion of Control
The framework dictates the protocol - the customization code just has to obey it!

NOTE: "Hollywood principle": Don't call us, we will call you.

Def. Unification
Both template and hook methods reside in the same class. The template

method is concrete and invokes an abstract hook method that can be overridden in subclasses.

Def. Separation
The template method is defined in one class and the hook methods are in another class. The template method is concrete and delegates to hook methods via an object reference.

Def. Blackbox framework
A blackbox framework is a framework that is customized using composition, i.e. to define the specific behavior of the framework, you specify a composition of predefined instances.

Def. Whitebox framework
A whitebox framework is a framework that is customized using inheritance, i.e. to define the specific behavior of the framework, you subclass abstract classes in the framework/implement interfaces, and give these concrete class instances to the framework to use.

Def. Software Product Line
A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

8.7 Systematic Testing

Def. Black-box testing
The UUT is treated as a black box. The only knowledge we have to guide our testing is the specification of the UUT and a general knowledge of common programming techniques and algorithmic constructs.

Def. White-box testing
The full implementation of the UUT is known, so the actual code can be inspected in order to generate test cases.

Def. Equivalence class (EC)
A subset of all possible inputs to the UUT, 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.

Def. Soundness
For a set of equivalence classes to be *sound*, the ECs most uphold the criteria of Coverage, Representation and Disjointness.

Def. Coverage
Every possible input belongs to one of the equivalence classes.

Def. Representation
If a failure is demonstrated on a particular member of an equivalence class, the same failure is demonstrated by any other member of that class.

Def. Disjointness
No input belongs to more than one equivalence class.

Myers rule for valid ECs
Until all valid partitions have been covered, define a new EC (and test-case) covering as many uncovered valid partitions as possible.

Myers rule for invalid ECs
Until all invalid partitions have been covered, define a new EC (and test-case) that covers one, and only one, of the uncovered invalid partitions.

8.8 Quality Attributes

Def. Quality Attributes
It is the mapping of a system's functionality onto software structures that determines the architecture's support for quality attributes.

NOTE: As in you can get the same functionality using many different architectures. Functionality and architecture are orthogonal.

Def. Reliability (IEEE 610)
The ability of a software system to perform it's required functions under stated conditions for a specified period of time.

Def. Flexibility
The capability of the software product to support added/enhanced functionality purely by adding software units and specifically not by modifying existing software units.

Def. Maintainability (ISO 9126)
The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

Def. Analysability (ISO 9126)
The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be indentified.

Def. Changeability (ISO 9126)
The capability of the software product to enable a specified modification to be implemented.

Def. Stability (ISO 9126)
The capability of the software product to avoid unexpected effects from modifications of the system.

Def. Testability (ISO 9126)
The capability of the software product to enable modified system to be validated.

9 Patterns

Patterns and their purposes:

ABSTRACT FACTORY: Provide an interface for creating families of related or dependent objects, without specifying their concrete classes. Page 203

ADAPTER: Convert the interface of a class into another interface clients expect. Adapter lets classes work together, that couldn't otherwise because of incompatible interfaces. Page 283

BUILDER: Separate the construction of a complex object from its representation, so that the same construction process can create different representations. Page 287

COMMAND: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. Page 290

DECORATOR: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Page 279

FACADE: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Page 271

ITERATOR: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Page 293

MODEL-VIEW-CONTROLLER: Define a loosly coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse and keyboard. Page 315

NULL OBJECT: Null Object er et alternativ til 'null' til at indikere fravaeret af et objekt at delegere et metodekald til. Istedet for at skulle teste for 'null' hver gang, bruges et objekt der ingenting goer. Available at http://www.cs.au.dk/dSoftArk/

OBJECT SERVER: Define a central access point for references to variability points/objects that may be changed at run-time. Available at http://www.cs.au.dk/dSoftArk/

OBSERVER: Define a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically. Page 308

PROXY: Provide a surrogate or placeholder for another object to control access to it. Page 297

STATE: Allow an object to alter its behaviour when its internal state changes. Page 178

STRATEGY: Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it. Page 125

TEMPLATE METHOD: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure. Available at http://www.cs.au.dk/dSoftArk/

9.1 Pattern details

Here will be details concerning the 15 patterns:

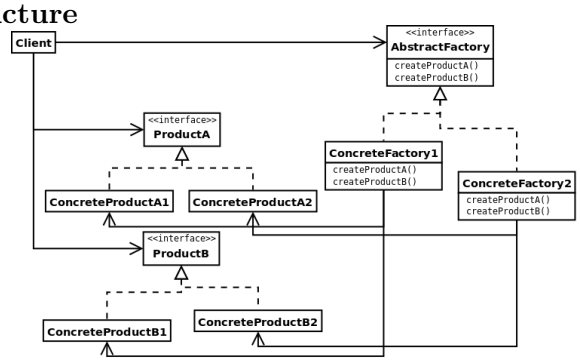
9.1.1 ABSTRACT FACTORY

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

Problem Families of related objects need to be instantiated. Product variants need to be consistently configured.

Solution Define an abstraction whose responsibility it is to create families of objects. The client delegates object creation to instances of this abstraction.

Structure



Roles

Abstract Factory: Defines a common interface for object creation.
ProductX: Defines the interface of an object ConcreteProductXY (product X in variant Y)
ConcreteFactoryY: Is responsible for creating Products that belong to a variant Y family of objects that are consistent with each other.

Cost/Benifit

Pros:
- Lowers coupling between client and products
- Makes exchanging product families easy
- Promotes consistency among products
- Change by addition, not by modification
- Client constructor parameter list stays intact

Cons:
- Introduces extra classes and objects
- Supporting new aspects of variation is difficult

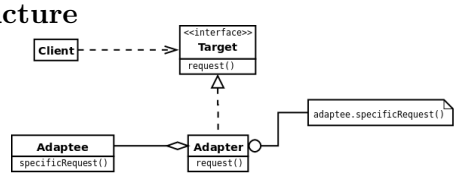
9.1.2 ADAPTER

Intent Convert the interface of a class into another interface clients expect. Adapter lets classes work together, that couldn't otherwise because of incompatible interfaces.

Problem You have a class with desirable functionality but its interface and/or protocol does not match that of the client it should be used by.

Solution You put an intermediate object, the adapter, between the client and the class with the functionality. The adapter conforms to the interface used by the client and delegates actual computation to the adaptee class, potentially performing parameter and protocol translations in the process.

Structure



Roles

Target: Is some interface defining some behaviour used by the Client.

Adapter: Implements the Target role and delegates actual processing to the Adaptee.
Adaptee: Performs parameter and protocol translations.

Cost/Benifit

Pros:
- Lets objects collaborate that otherwise are incompatible.
- A single adapter can work with many adaptees

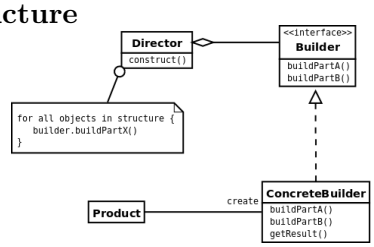
9.1.3 BUILDER

Intent Separate the construction of a complex object from its representation, so that the same construction process can create different representations.

Problem You have a single defined construction process, but the output format varies.

Solution Delegate the construction of each part in the process to a builder object. Define a builder object for each output format.

Structure



Roles

Director: Defines a building process by delegating.
Builder: Constructs the particular parts delegated from Director.
ConcreteBuilders: A set of ConcreteBuilders are responsible for building concrete Products

Cost/Benifit

Pros:
- Easy to define new products
- Construction code and representation are isolated
- Finer control over the construction than with e.g. ABSTRACT FACTORY

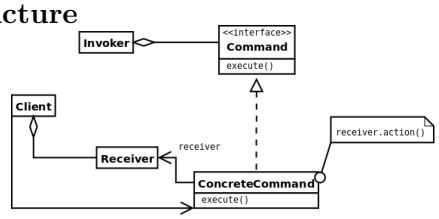
9.1.4 COMMAND

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem We want to support undo and want users to be able to dynamically associate commands with specific buttons, menu-items and shortcut keys.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an *execute* method. This way requests can be associated user interface elements dynamically, stored and replayed etc.

Structure



Roles

Invoker: Is the user interface related object that may execute a Command.

Command: Defines the responsibility of being an executable operation.

ConcreteCommand: Defines the concrete operations that manipulates the Receiver.

Receiver: The actual target of manipulation from the command.

Cost/Benifit

Pros:

- Commands can be manipulated like objects.
- Commands can be assembled into composite commands.
- Easy to add new commands.

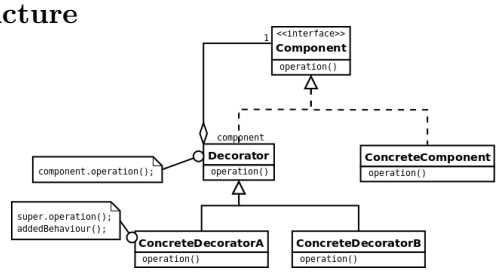
9.1.5 DECORATOR

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

Problem You want to add responsibilities and behaviour to individual objects without modifying its class.

Solution You create a decorator class that responds to the same interface. The decorator forwards all requests to the decorated object, but may provide additional behaviour to certain requests.

Structure



Roles

Component: Defines the interface of some abstraction.
ConcreteComponent: Implementations of Components.
Decorator: Adds behaviour to a Component

Cost/Benifit

Pros:
- Allows adding and removing responsibilities at run-time.
- Allows incrementally adding responsibilities
- Can provide complex behaviour by chaining decorators.

Cons:
- Long chains of small objects that look alike.
- Can be difficult to understand purpose of decorators.
- Delegation code for each method in decorator is tedious to write.

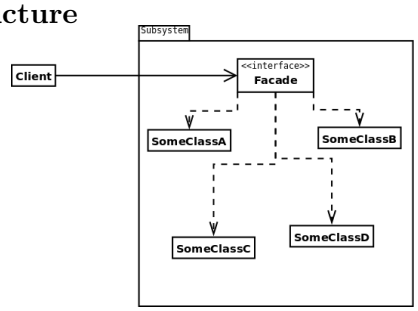
9.1.6 FACADE

Intent Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

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

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have access to the subsystem objects directly.

Structure



Roles

Facade: Defines the simple interface to the subsystem.
Client: Accesses the subsystem only via the Facade.

Cost/Benifit

Pros:
- Shields clients from subsystem objects.
- Promotes weak coupling between client and subsystem.

Cons:
- Facade becomes very bloated with methods.

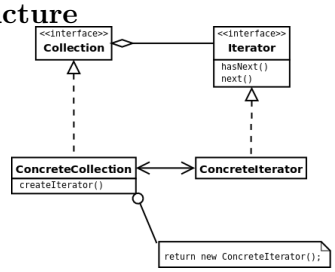
9.1.7 ITERATOR

Intent Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Problem You want to iterate a collection without worrying about the implementation details of it.

Solution Encapsulate iteration itself into an object whose responsibility it is to allow access to each element in the collection.

Structure



Roles

Collection: Is some data structure that can be iterated.
Iterator: Defines the iteration responsibility.
ConcreteIterator: Can be returned on request and knows how to iterate ConcreteCollection.

Cost/Benifit

Pros:
- Decouples iteration from the collection.
- Supports variations in the iteration.
- Slims collection class.
- Several iterators can work simultaniously.

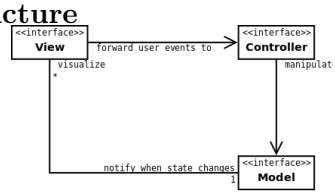
9.1.8 MODEL-VIEW-CONTROLLER

Intent Define a loosly coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse and keyboard.

Problem A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.

Solution A Model contains the application's state and notifies all Views when state changes happen. The Views are responsible for rendering the model when notified. User input events are received by a View but forwarded to its associated Controller. The Controller interprets events and makes the appropriate calls to the Model.

Structure



Roles

Model: Maintains application state and updates all associated Views.
View: Renders the model graphically and delegates user events to the Controller.
Controller: Is responsible for reacting on user events and modifying the Model.

Cost/Benifit

Pros:

- Loose coupling between all three roles.
- Multiple views supported.
- Possible to change event processsing at run-time.

Cons:

- Unexpected/Multiple updates.
- Design complexity.

9.1.9 NULL OBJECT

Intent Null Object er et alternativ til 'null' til at indikere fravaeret af et objekt at delegere et metodekald til. Istedet for at skulle teste for 'null' hver gang, bruges et objekt der ingenting goer.

Motivation En langsommelig algoritme boer have en 'progress-indicator'. Denne kan have forskellige udforminger: grafisk, tekstuel, osv. Ergo bruger vi en Strategy:

```
public void slowAlgorithm() {
 ...; progress.report( "done A" );
 ...; progress.report( "done B" );
 ...; progress.report( "done C" ); }
```

Under aftestning oensker vi ikke output; en "NullReporter" har bare en tom metodekrop for 'report' metoden.

Structure

Cost/Benifit

Pros:

- Koden mere sikker overfor programmoerfejl.

Cons:

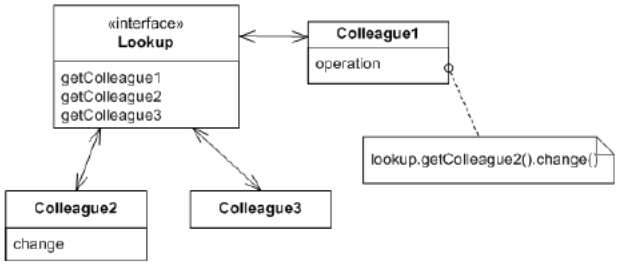
- Boer ikke implementeres hvis AbstractOperation ikke allerede findes.

9.1.10 OBJECT SERVER

Intent Define a central access point for references to variability points/objects that may be changed at run-time.

Motivation If objects that define variability in a design can be changed at run-time to provide run-time variability there must be global consensus in a system about which objects to use.
A typical example is objects playing the ConcreteStrategy role from the Strategy pattern. If a ConcreteStrategy is changed, all objects/subsystems that uses the strategy must immediately use the new strategy object.

Structure



Cost/Benifit

Pros:
- Object references only stored in one place.

Cons:
- Breaks Demeter's Law

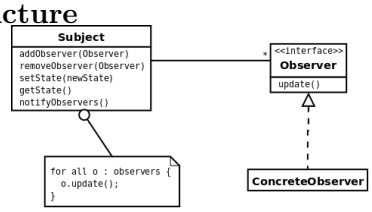
9.1.11 OBSERVER

Intent Define a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.

Problem A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.

Solution All objects that must be notified (Observers) implements an interface containing an *update* method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the *update* method on each object in the list. Thereby all observing objects are notified of state changes.

Structure



Roles

Observer: Specifies the responsibility and interface for being able to be notified.

Subject: Is responsible for holding state information, a list of observers and for notifying observers.

ConcreteObserver: Defines concrete behaviour for how to react to state changes.

Cost/Benifit

Pros:

- Loose coupling between Subject and Observer.
- Support broadcast communication.

Cons:

- Unexpected/Multiple updates.

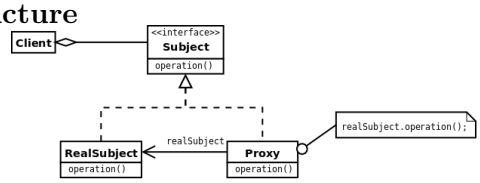
9.1.12 PROXY

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively effect the client's resource requirements even if the object is not used at all. Or we need different types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure



Roles

Client: A Client only interacts via a Subject interface.
RealSubject: Implements the resource-demanding operations.
Proxy: Implements the Subject interface and provides access control to the RealSubject.

Cost/Benifit

Pros:
- Strengthens reuse.
- Can act as proxy for several types of real subjects.

Cons:
- Introduces an extra level of indirection.

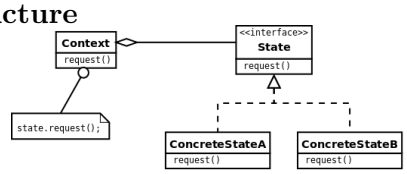
9.1.13 STATE

Intent Allow an object to alter its behaviour when its internal state changes.

Problem Your product's behaviour varies at run-time depending upon some internal state.

Solution Describe the responsibilities of the dynamically varying behaviour in an interface and implement the concrete behaviour associated with each unique state in an object, the state object, that implements t-his interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed refer to the corresponding state object.

Structure



Roles

State: Specifies the responsibilities and interface of the varying behaviour.

ConcreteState: Objects define the specific behaviour associated with a state.
Context: The Context delegates to its current state object.

Cost/Benifit
Pros:
- State specific behaviour is localized.
- Makes state transitions explicit.
Cons:
- Increased number of objects and interactions.

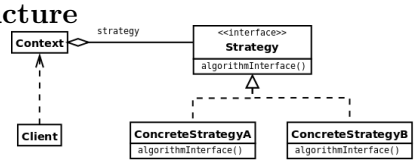
9.1.14 STRATEGY

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controling the variability.

Solution Separate selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithms realize this interface.

Structure



Roles
Strategy: Specifies the responsibility and interface of the algorithm.
ConcreteStrategy: Defines concrete behaviour fulfilling the responsibility.
Context: Performs the work for Client by delegating to a Strategy.

Cost/Benifit
Pros:
- Strategies eliminate conditional statements.
- Avoids subclassing.
- Avoids multiple maintenance.
- Change by addition, not by modification.
- Makes separate testing of Context and Strategy possible.

Cons:
- Increased number of objects.
- Clients must be aware of strategies.

9.1.15 TEMPLATE METHOD

Intent Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure.

Problem Some steps in the algorithm are fixed but some steps I would like to keep open for future modifications to the behaviour.

Solution Can be implemented by Unification (Subclassing) or Separation (Composition).

10 Pattern Code Examples

10.1 ABSTRACT FACTORY

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

PayStation Code

```
1  public class PayStationImpl implements PayStation {
2
3      PayStationFactory factory;
4      RateStrategy rateStrategy;
5
6      public PayStationImpl( PayStationFactory factory )
7      {
8          this.factory = factory;
9          this.rateStrategy = factory.createRateStrategy();
10     }
11
12     public Receipt buy()
13     {
14         Receipt r = factory.createReceipt(timeBought);
15         timeBought = insertedSoFar = 0;
16         return r;
17     }
18 }
```

PayStationFactory Interface

```
1  public interface PayStationFactory {
2      // Create an instance of the rate strategy to use
3      public RateStrategy createRateStrategy();
4
5      /** Create an instance of the receipt
6        * @param parkingTime the number of minutes
7        * parking time the receipt is valid for
8      */
9      public Receipt createReceipt(int parkingTime);
10 }
```

10.2 ADAPTER

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together, that couldn't otherwise because

of incompatible interfaces.

10.3 BUILDER

Intent: Separate the construction of a complex object from its representation, so that the same construction process can create different representations.

10.4 COMMAND

Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

10.5 DECORATOR

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

```
1  public class TranscriptGame implements Game {
2      private Game g;
3      private PrintStream p;
4      ...
5      public TranscriptGame(Game g, PrintStream p) {
6          this.g = g;
7          this.p = p;
8      }
9
10     private void transcribe(String line) {
11         p.println(line);
12     }
13     ...
14     public void endOfTurn() {
15         transcribe(g.getPlayerInTurn() + "␣ends␣turn");
16         g.endOfTurn();
17     }
18     ...
19 }
```

10.6 FACADE

Intent: Provide a unified interface to a set of interfaces in a subsystem.

Facade defines a higher-level interface that makes the subsystem easier to use.

```
1 public class Main {
2     public static void main(String[] args) {
3         RateStrategy rs = new LinearRateStrategy();
4         Paystation ps = new PayStationImpl(rs);
5         new PayStationGUI(ps);
6     }
7 }
```

10.7 ITERATOR

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

10.8 MODEL-VIEW-CONTROLLER

Intent: Define a loosly coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse and keyboard.

10.9 NULL OBJECT

Intent: Null Object er et alternativ til 'null' til at indikere fravaeret af et objekt at delegere et metodekald til. Istedet for at skulle teste for 'null' hver gang, bruges et objekt der ingenting goer.

```
public void slowAlgorithm() {
  ...; progress.report( "done A" );
  ...; progress.report( "done B" );
  ...; progress.report( "done C" ); }
```

10.10 OBJECT SERVER

Intent: Define a central access point for references to variability points/objects that may be changed at run-time.

CompositePricingStrategy

```
public int calculateParkingTime(int amount)
{
    PricingStrategy which;
    if ( pm.getWeekdayDeterminationStrategy().isWeekend() )
    {
        which = linear;
    }
    else
    {
        which = progressive;
    }
    return which.calculateParkingTime(amount);
}
```

10.11 OBSERVER

Intent: Define a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.

```
public class StandardGame implements Game {
    ...
    private List<GameObserver> observers;
    ...
    public StandardGame(GameFactory factory) {
        ...
        // initialize a list of GameObservers
        observers = new ArrayList<GameObserver>();
    }
    ...
    private void endOfTurn() {
        ...
        ...
        // Notify end-of-turn
        notifyEndOfTurn();
    }
    ...
    private void notifyEndOfTurn() {
        // Make even calls
        for (GameObserver g : observers) {
            g.endOfTurn();
        }
    }
    ...
}
```

10.12 PROXY

Intent: Provide a surrogate or placeholder for another object to control

access to it.

10.13 STATE

Intent: Allow an object to alter its behaviour when its internal state changes.

```
1  public class PayStationImpl implements PayStation {
2      ...
3      // The strategy for rate calculations
4      private RateStrategy rateStrategyWeekday;
5      private RateStrategy rateStrategyWeekend;
6
7      // Construct a pay station
8      public PayStationImpl( RateStrategy rateStrategyWeekday,
           RateStrategy rateStrategyWeekend )
9      {
10         this.rateStrategyWeekday = rateStrategyWeekday;
11         this.rateStrategyWeekend = rateStrategyWeekend;
12     }
13
14     public void addPayment(int coinValue) throws IllegalCoinException
15     {
16         ...
17         if( isWeekend() ) {
18             timeBought = rateStrategyWeekend.calculateTime(
                   insertedSoFar);
19         } else {
20             timeBought = rateStrategyWeekday.calculateTime(
                   insertedSoFar);
21         }
22     }
23     ...
24     private boolean isWeekend()
25     {
26     ...
27     }
28 }
```

10.14 STRATEGY

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

```
1  public class PayStationImpl implements PayStation {
2      private int insertedSoFar;
3      private int timeBought;
4
5      // The strategy for rate calculation
6      private RateStrategy rateStrategy;
```

```
7
8       public PayStationImpl(RateStrategy rs) {
9           rateStrategy = rs;
10      }
11
12      public void addPayment(int coinValue) throws IllegalCoinException
            {
13          ...
14          insertedSoFar += coinValue;
15          timeBought = rateStrategy.calculateTime(insertedSoFar);
16      }
17  }
```