Kodekvalitet

Buzzwords, det store billede, tanker, ideer, **handling**

Hvad er kodekvalitet – og hvorfor er det dyrt at ignorere?

Hvad er kodekvalitet

- Kodekvalitet:
  - Forudsigelig adfærd
  - Let at ændre (modificerbarhed)
  - Let at forstå (læsbarhed)
  - Lav risiko ved ændringer (regression/skrøbelighed)
  - Tidlig fejldetektion
- Buzzwords
  - Change Amplification
    - Hvor mange steder i koden fylder en enkelt ændring
  - Defect Density
    - Antal fejl i forhold til størrelse af integration
  - Mean Time Between Failure
    - Hvor ofte den fejler
    - If it aint broken, dont fix it
  - Mean Time To Repair
    - Hvor lang tid fra fejl opdaget til prod er stabilt igen

Vores ansvar

- ”Koden er i forvejen ikke god, men…”
- ”Det virker vist nok”
- ”Den her skal måske bruges”
- ”Det er nok lavet på bedste måde i forvejen”
- Hurtig ind, hurtig ud

- Boy Scout Rule
  - Efterlad koden pænere end du fandt den
- YAGNI
  - You Ain’t Gonna Need It
- Bit Rot / Software Entropi / Teknisk gæld
  - Forfald over tid
- Sustainability over Maintainability
  - Kun bæredygtigt hvis man er i stand til at reagere på forandringer i hele dens levetid

"Code is a liability, not an asset. The best code is the code that is easy to delete when it no longer serves its purpose."

Tankegang

- Detection levels:
  - Compile-time, test-time, startup-time, runtime
- Økonomi i fejl
  - Testbart < feedback
  - Kompleks < manuelt/automatisk udførsel/vedligeholdelse
  - Før deploy, under deploy, efter deploy

- Shift Left
  - Fra detection til prevention
- The Paradox of Prevention
  - Hero Culture (reaktiv) og Operational Excellence (proaktiv)
  - Reaktive helt = Brandmanden
    - Reparerer symptomer, høj synlighed, lille langsigtet værdi. Her og nu. Bugfixing øger kompleksiteten.
  - Proaktive helt = Ingeniøren
    - Reparerer årsag (root cause). Helst før problemet opstår, bliver ikke ”set”, har stor værdi. Refaktorering.
- ”The parable of the Fence and the Ambulance”

Godt design handler primært om at flytte fejl fra runtime til compile-time og fra produktion til udvikling og fra src/main til src/test

Refaktorering er vigtig, fordi..

You can't change the world,

But you can change the facts,

And when you change the facts,

You change points of view,

If you change points of view,

You may change a vote,

And when you change a vote,

You may change the world

Depeche Mode, New Dress, 1986

Refaktorering er vigtig, fordi..

You can't change the world,

But you can change the facts,

And when you change the facts,

You change points of view,

If you change points of view,

You may change a vote,

And when you change a vote,

You may change the world

Depeche Mode, New Dress, 1986

- Al vores kode er forfærdelig
- Men jeg fjerner lige `public static final String Version = "$Id";`
- Fordi det er ikke relevant mere (svn repository)
- Og det gør de andre også i code reviews
- Og hvis de andre også gør det
- Er det måske "en god ting"™
- Så summen af alle de gode ting
- Gør koden lidt bedre

Grundprincip: Sammenhæng og kobling

Høj intern binding (High Cohesion = sammenhæng)

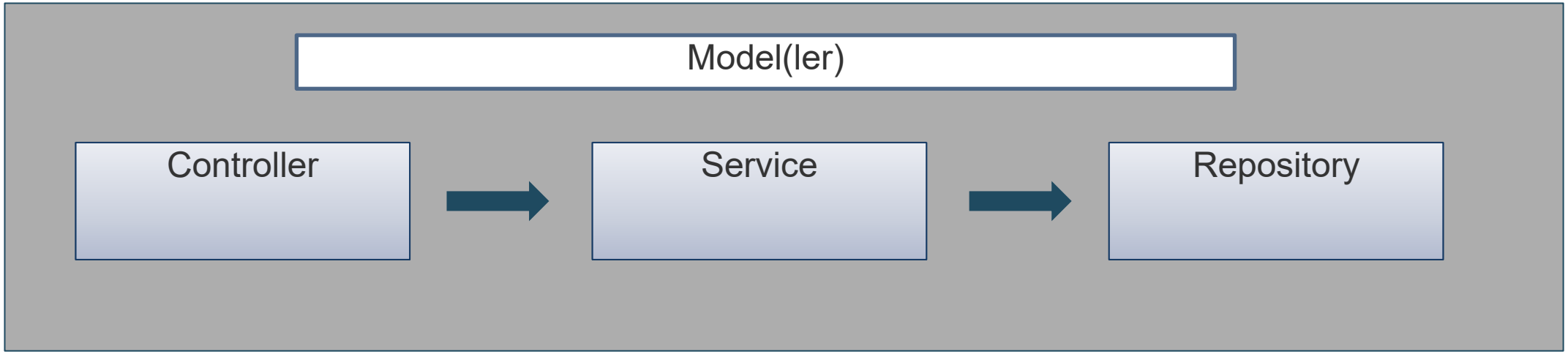
- Modularisering, objektorienteret
  - Klasser, packages m.m. har ét ansvar
  - Metoder relaterer til samme domæne
  - Data og adfærd hører naturligt sammen
- Eksempel
  - Utilityklasse med 25 statiske metoder -> Lav cohesion
  - OrderValidator der kun validerer ordrer -> High cohesion
  - Alle klasser for en use-case samlet i en package/module

Lav ekstern kobling (Low Coupling = løs kobling)

- Målbar via
  - Antal imports og deres type
  - Antal afhængigheder
  - Antal transitive afhængigheder
  - Direkte adgang til fremmede strukturer (getter efter getter) og gør-det-selv kode
- Strukturer:
  - Abstraktioner / Interfaces
  - Pakker
  - Arkitekturlag

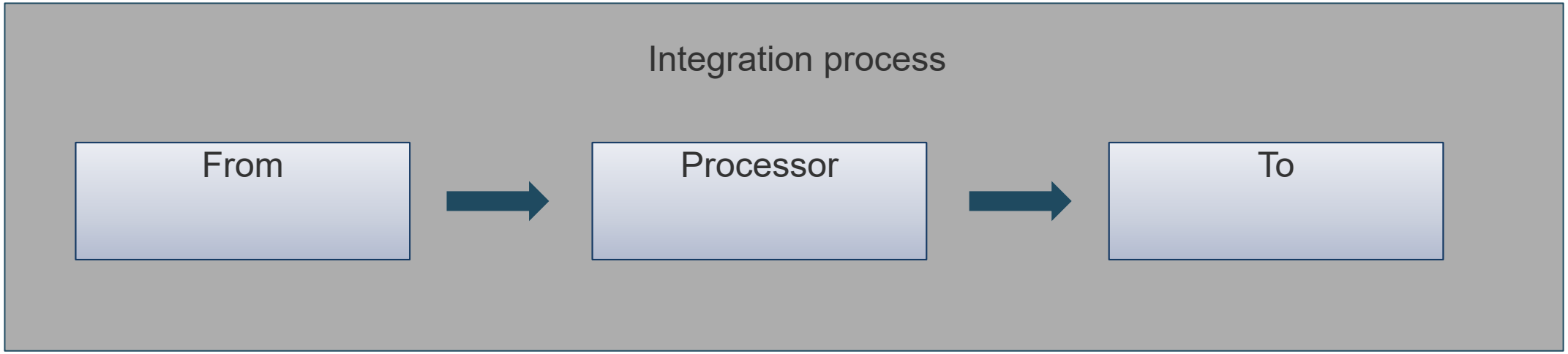
S103-DPR2Epic:dk.regionh.integraiton.dpr2epic.servlet.RunServlet

Web Service Arkitektur



Weblaget	Forretningslaget	persistenslaget
http request Parsing json/XML validering	”Forretningslogik” Transaktioner	Data repositories DAO Eksterne systemer

Camel arkitektur?



Konfiguration/Code

Header + body
routing
Validering/Transformering
Forretningslogik
Persistens

Konfiguration/Code

Arkitektonisk Sundhed

- Separation of Concerns
  - Prøv at adskille teknologispecifik kode fra domænespecifik kode
  - Hold Camel-specifik kode væk fra din domænemodel. Forretningslogik burde kunne køres uden en kørende Camel kontekst
- Law of Demeter (Paper Boy)
  - Ikke hente alle data via service og repository og iterere over dem selv. Bed service om at levere dem (Et objekt må kun tale med sine ”nære bekendte”)
  - Order.getCustomer().getAddress().getZipCode() vs Order.getCustomerZipCode()
  - i.e. push-back ind i abstraktioner(eller ”rig domænemodel” vs ”anæmisk model”)

Sammenhæng og kobling – praktisk og konkret

Refaktorering mod læsbarhed

- Gode råd kan f.eks. være
  - Giv betingelser et navn
    - Ikcommon5:WsHealthViewHandler$WsInfo#getWsStatus:191 (request timeout)
  - Læs en metode som en usecase (extract method)
    - Ikcommon5:BackupProcessor#configureBackup:72
  - Lav Enums / Domain Types
  - Brug Records / Value Objects

Indkapsling = Ydre sikkerhed

- Visibility Matters
  - Placer ting efter use-case ikke efter teknologi (*Feature-based packaging*)
  - Public bør være din sidste udvej. Brug package-private til interne komponenter
- Beskyttelse af data og tilbyd metoder (Law of Demeter)
- Postel's Law (Be Conservative in what you send, liberal in what you accept)
- **Hver public metode eller klasse er en forpligtelse**
  - Her er IDEA IKKE altid din ven

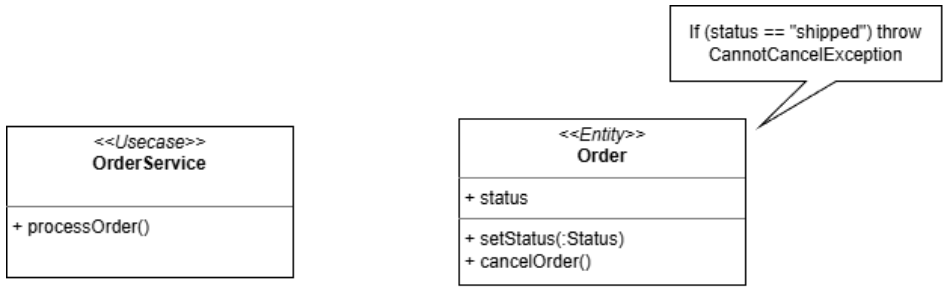
Gør-det-selv metoden

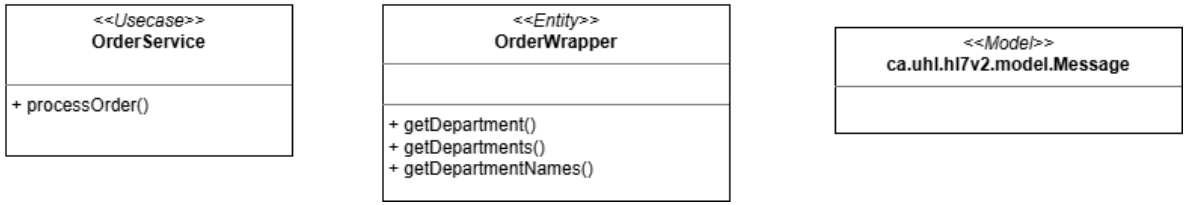
- Mange klasser har en tendens til at gøre noget praktisk for at løse et problem
  - Klassen har mange direkte afhængigheder
  - Der tager data ind og selv gør noget
  - Dvs høj grad af orkestrering
- Giver en høj kobling
- The Middle Man er ofte en løsning (Facade pattern / Service Facade)
  - Koreografi/delegering af delopgaver til en ”intermediate” eller ”Coordinator”

Anæmisk vs rig domænemodel

- Anæmisk model er blot data pakket i en objekt-orienteret struktur
- Regler for ændringer ligger i logik andre steder
  - e.g. setStatus vs cancelOrder
    - (cannot cancel when state==shipped)
  - Objektet ejer IKKE sin tilstand
  - Invarians er IKKE beskyttet
  - Ændringer til logik er spredt i koden (logic duplication og change amplification)
- Anæmiske modeller er acceptable når
  - DTO'er til transport, JPA entiteter med lav kompleksitet
  - Enkle CRUD-systemer uden domænekompleksitet
- Hvis man ikke ejer domænemodellen (f.eks. HL7 Message)
  - E.g. lokal Wrapper med invarians

Law of Demeter og rige modeller





Kompleks at teste

Nem at teste

Skal ikke testes

Designværktøj: CRC (Class-Responsibility-Collaborators)

- Drop komplekse UML diagrammer
- Fokuser på
  - Hvad ved klassen?
  - Hvad gør klassen?
  - Hvem taler den med?
  - Blander den forretning og teknologi?
  - Supergodt som klassens javadoc (documentation as code quality)
- S103-DPR2Epic som workshop udgangspunkt
  - DPRCprStatusFix/CprChangeHelper/FileErrorBean/…

Ting, der ikke kan unit-testes

- Ofte fordi klassen gør for meget (Overtræder Single Responsibility)
- Eller fordi den er for teknologi-specifik (Burde ikke indeholde forretningskode)
- Eller måske bruger statiske metoder
- Eksempler på mønstre til gradvis forbedring
  - Humble Object
  - Result Object
  - Testable Static

Humble Object 1

- Et problem vi har ofte, primært af historiske årsager er at blande camel logik og forretningslogik

```
// ❌ Forretningslogik blandet med Camel-infrastruktur
public class OrderProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        Order order = exchange.getIn().getBody(Order.class);

        // Forretningslogik her - FORKERT STED!
        if (order.getTotalAmount() > 10000) {
            order.setRequiresApproval(true);
        }

        exchange.getIn().setBody(order);
    }
}
```

Humble Object 2

- Separation af logik i separat abstraktion
- Forretningslogik kan testes med unittests

```
public class OrderBusinessLogic {
    public Order applyBusinessRules(Order order) {
        if (order.getTotalAmount() > 10000) {
            order.setRequiresApproval(true);
        }
        return order;
    }
}

public class OrderRouteProcessor implements Processor {
    private final OrderBusinessLogic businessLogic;

    @Override
    public void process(Exchange exchange) throws Exception {
        Order order = exchange.getIn().getBody(Order.class);

        // Deleger til ren bean
        Order processedOrder = businessLogic.applyBusinessRules(order);

        // Teknisk håndtering
        exchange.getIn().setBody(processedOrder);
        exchange.getIn().setHeader("ProcessingComplete", true);
    }
}
```

Humble Object 3

- Camel annoteringer kan gøre mediator overflødig.
- Brug .bean() i stedet for .process()

```
@Component
public class OrderBusinessLogic {

    @Handler
    public Order applyBusinessRules(@Body Order order, @Header(”customerId”) {
        order.setCustomerId(customerId);
        if (order.getTotalAmount() > 10000) {
            order.setRequiresApproval(true);
        }
        return order;
    }
}

from(”direct:new-order”)
    .bean(OrderBusinessLogic.class)
    .to(”direct:next”);
```

Result Object

- Gå fra handling til hensigt og returner din hensigt
- Ydre processor kan lave send
- Skal ikke mocke for at teste (eller starte en rute)

```
public class MyProcessor implements Processor{

public void process(Exchange exchange) {
      ….
      business code
      ….
      if (someComplexLogic()) {
          exchange.getContext().createProducerTemplate().sendBody(”direct:other”, ..)
      }
      ….
}

public record ProcesssingResult(Message main, Optional<Message> sideEffect);

public class BusinessLogic {
     public ProcessingResult calculate(String input) {
        if(someComplexLogic()) {
            return new ProcessingResult(”done”, Optional.of(”other”));
        }
        return new ProcessingResult(”done”, Optional.empty());
     }
}
```

Testable Static 1

- Du bruger en statisk klasse til noget
- Flyt det ud i separat metode
- Test kan lave en anonym nedarvning

```
public class DoSomething{

   public void goDoSomething() {
      ….
      business code
      ….
      if (Config.getString(”token”)) {
          ….
      }

      //Erstat med
      String getConfigString(String key) {
        return Config.getString(key);
   }
}

public class DoSomethingTest{

   public void testDoSomething() {
      DoSomething testWith = new DoSomething() {
          @Override String getConfigString(String key) { return ”A54D-F65B”; }
      ..
      use test.with.goDoSomething() as usual
   }
}
```

Testable Static

- Alternativt med fuld strategy pattern
- Klassen kan skifte resolve strategy når det er nødvendigt (begrænset scope)

```
@FunctionalInterface public interface ConfigResolver {
    String resolve(String key);
}
public class DoSomething{
    private ConfigResolver resolver = Config::getString;

    void setConfigResolver(ConfigResolver resolver) {
        this.resolver = resolver;
    }

    public void goDoSomething() {
        if (resolver.resolve(”token”)) {
            ….
        }
    }
}
```

Konkrete takeaways

- Hvert klasse har ét klart ansvar
- Beskyt interne data – saml operationer og invarians
- Reducer imports – Mål kobling
- Adskil domæne fra teknologi
- Gør komplekse udtryk læsbare
- Design så fejl opdages tidligt
- Hvis det ikke kan testes, er designet forkert

Interessant læsning, måske

- Clean Code
- Effective Java
- Enterprise Integration Patterns