Design 2

Repetition
GRASP +
Design af kompleks interaktion
GoF (singleton)

Oversigt

Krav

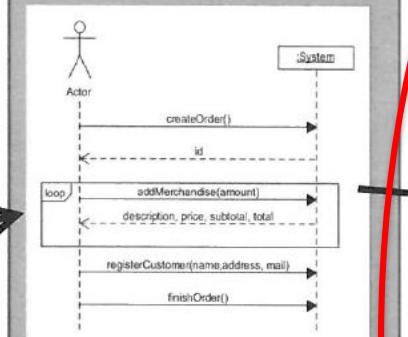
Fully dressed for de højest prioriterede use cases

Benyt RoI og kompleksitet som faktorer

Use case:	Registrer Ordre	
Aktører	Ekspedient	
Præcondition	De ønskede varer er registrerede og på lager	
Postcondition	Ordren er oprettet og varer og kunde er tilføjet	
Flow of events	Aktør handling	System svar
	1 En kunde ringer for at bestille varer	
	2 Ekspedienten starter en ny ordre	3 Systemet opretter en ny ordre
	4 Ekspedienten angiver id på ønskede vare	5 Systemet returnerer vareinfo og deltotal
	6 …	7 …

Analyse

SSD for use case Registrer Ordre



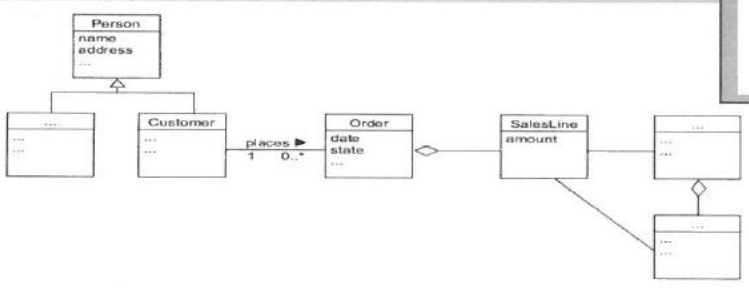
Operations kontrakter

Operation: createOrder
…

Operation: addMrchandise(id, amount)
…

Operation: registerCustomer(name, address, mail)
præcondition: ingen
postcondition:
-en instans c af Customer blev oprettet
-c blev tilskrevet værdierne c.name, c.address og c.mail

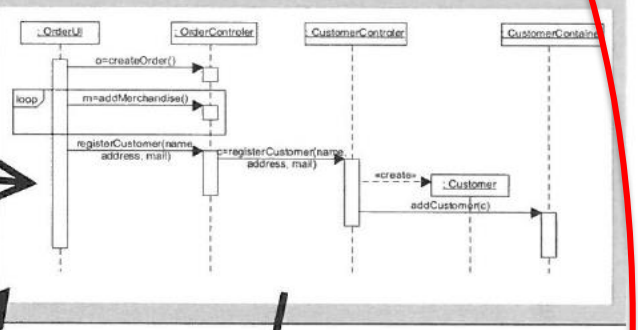
Domænemodel



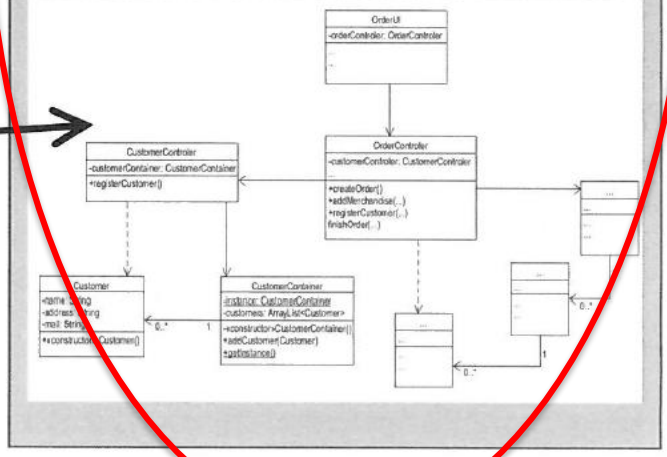
Design

Vi bruger også arkitekturen til design

Interaktionsdiagram for Registrer Ordre



Designklassediagram, laves fra interaktionsdiagram

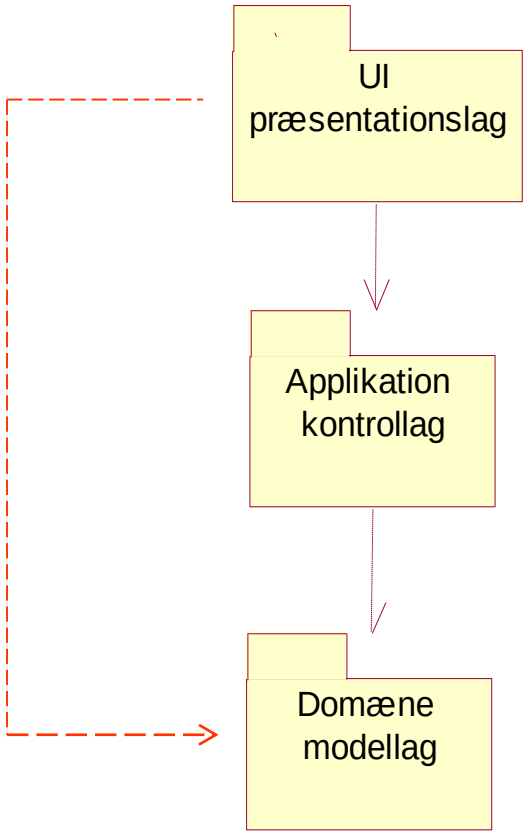


(repetition) En lagdelt grundarkitektur

Larman kap. 13.6

I praksis blive man ofte nødt til indgå kompromisser, som bryder med idealmodellen, fx at UI laget kan læse modellerne i domænelaget (åben arkitektur)

Pilene viser afhængigheden (synligheden) mellem lagene



håndterer interaktionen mellem aktøren og grænsefladen – sender systemhændelsen videre

håndterer afviklingen af use cases – typisk én kontroller klasse per use case

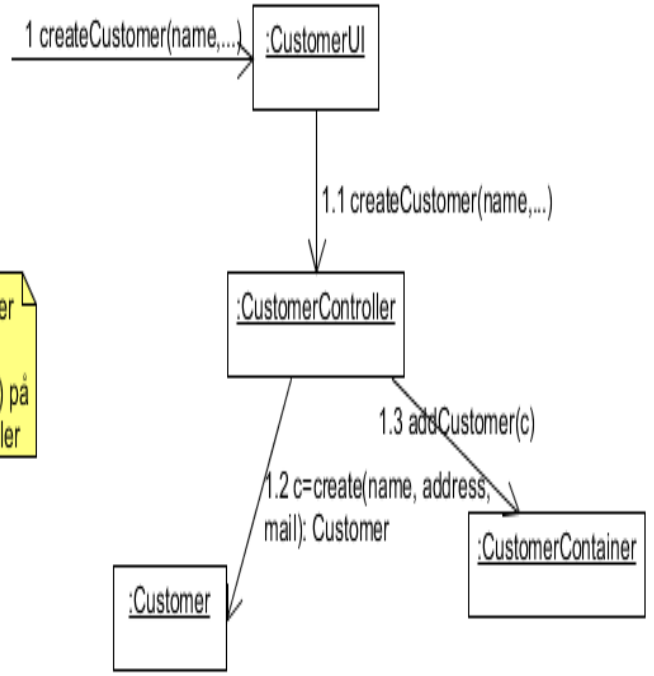
de domæneklasser som use casene bruger plus deres containerklasser (f.eks. ArrayList'er)

(repetition) Design af interaktion
Eksempel: Kundekartotek

Domæne model:

Design af interaktion: registerCustomer

UML kommunikationsdiagram



Operationskontrakt

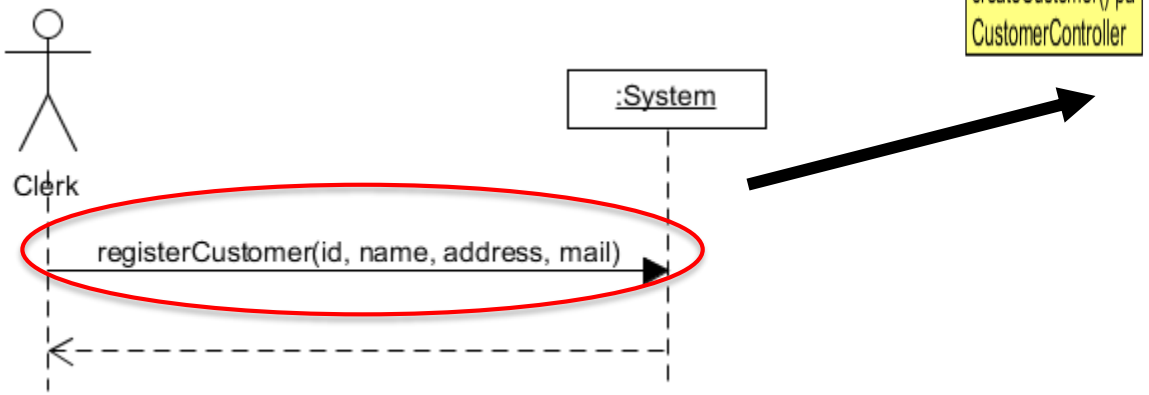
Præcondition: Ingen

Postcondition:

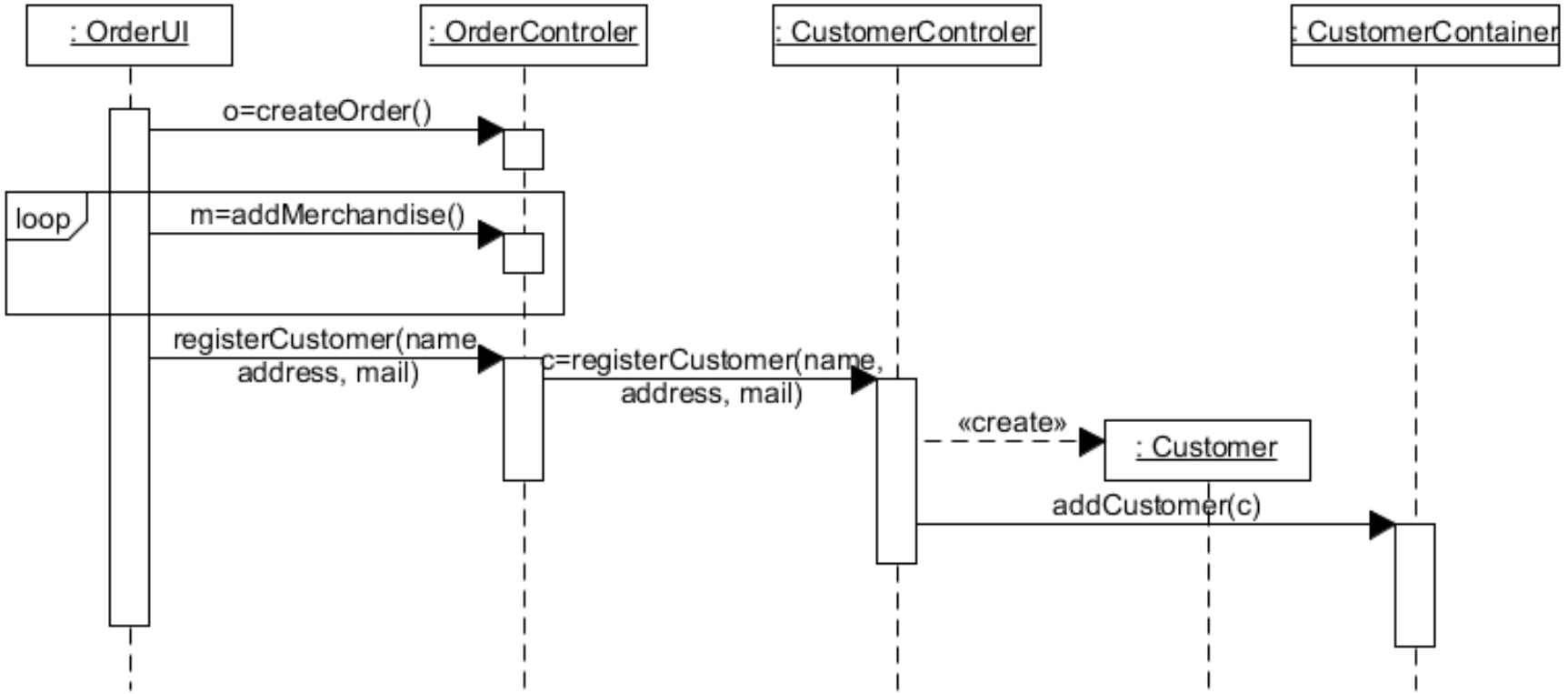
–En instans af Customer c blev oprettet

–c.customerNo, c.name, c.address, c.mail tilskrives værdier

SSD:



(repetition) Alternativt interaktionsdiagram UML sekvensdiagram (mit foretrukne)



(repetition) Design klassediagram

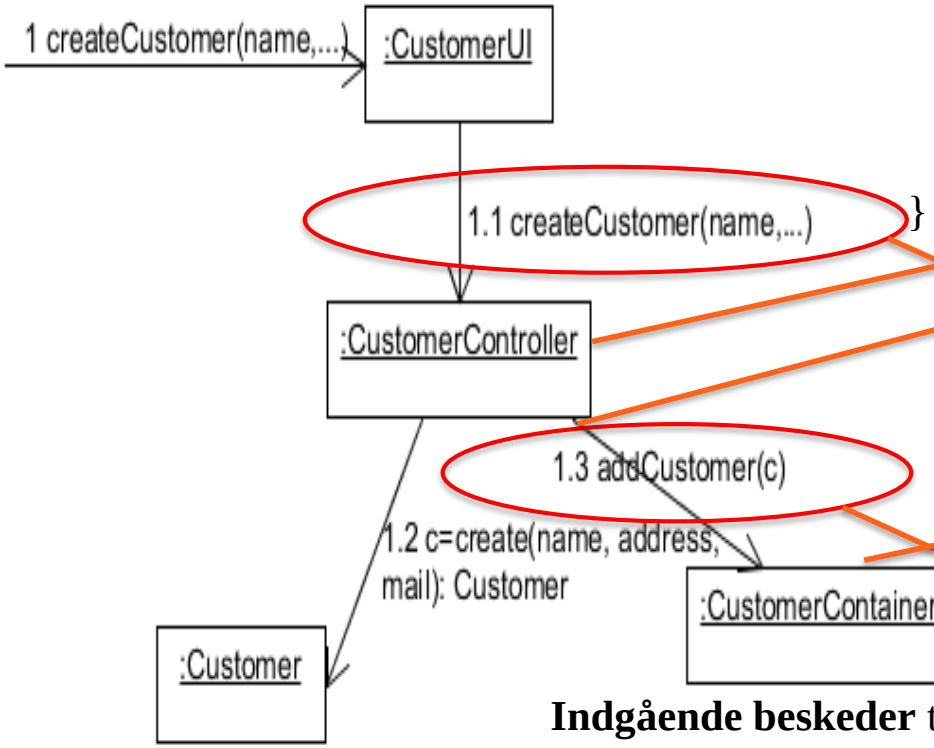
- Et design klassediagram indeholder følgende:
  - klasser, associeringer og attributter
  - interfaces
  - metoder
  - attributter og deres datatyper
  - synlighed

(repetition) Design af klasser fra kommunikationsdiagram

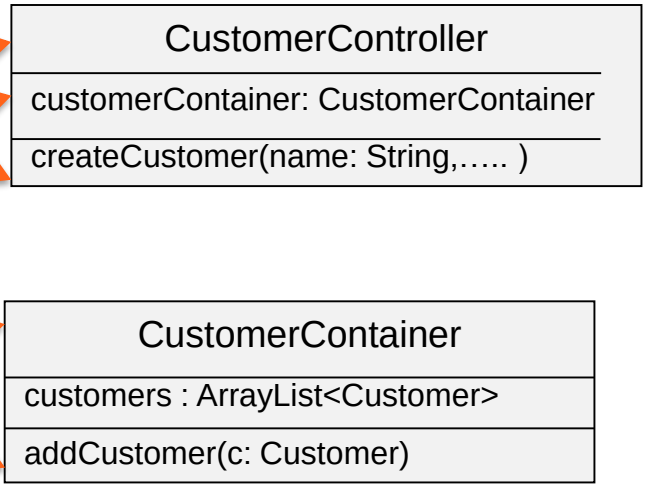
Eksempel: Kundekartotek

Use case interaktion

Kommunikationsdiagram




```
public class CustomerController {
   private CustomerContainer customerContainer;
   public  ….
   public void createCustomer(String name, String
                                address, String mail) {
      Customer c=new Customer(name, address,
                                        mail);
      customerContainer.addCustomer(c);
   }
}
```

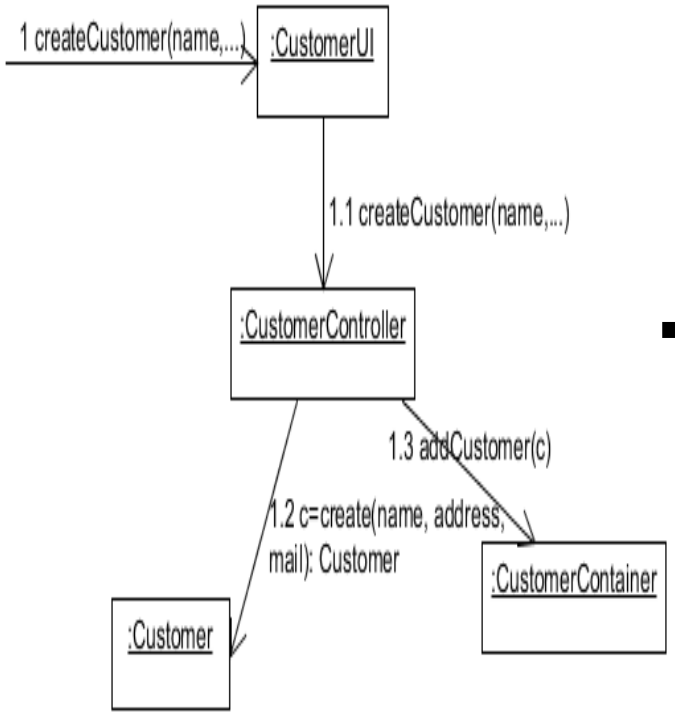


Indgående beskeder til en klasse betyder, at **klassen må definere en modsvarende metode**

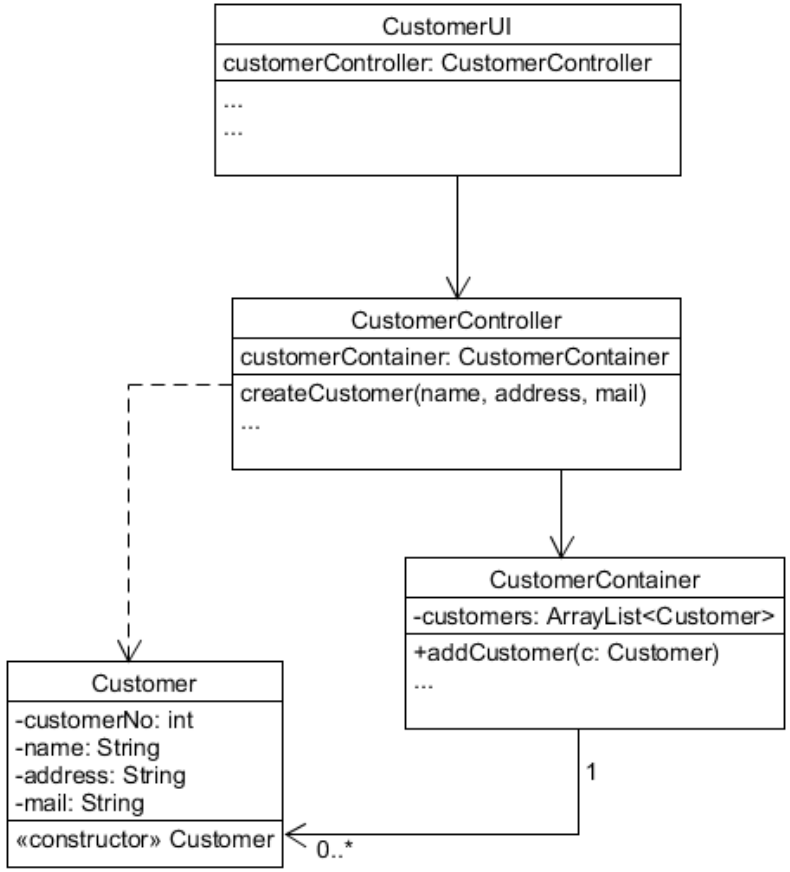
(repetition) Det endelige designklassediagram
Eksempel: Kundekartotek

Use case interaktion

Kommunikationsdiagram

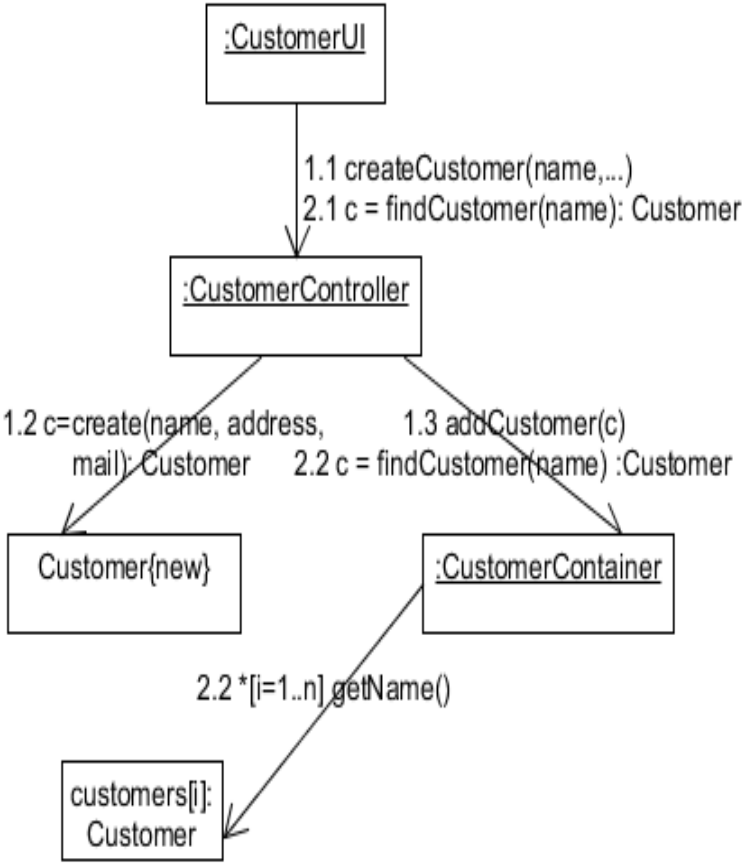


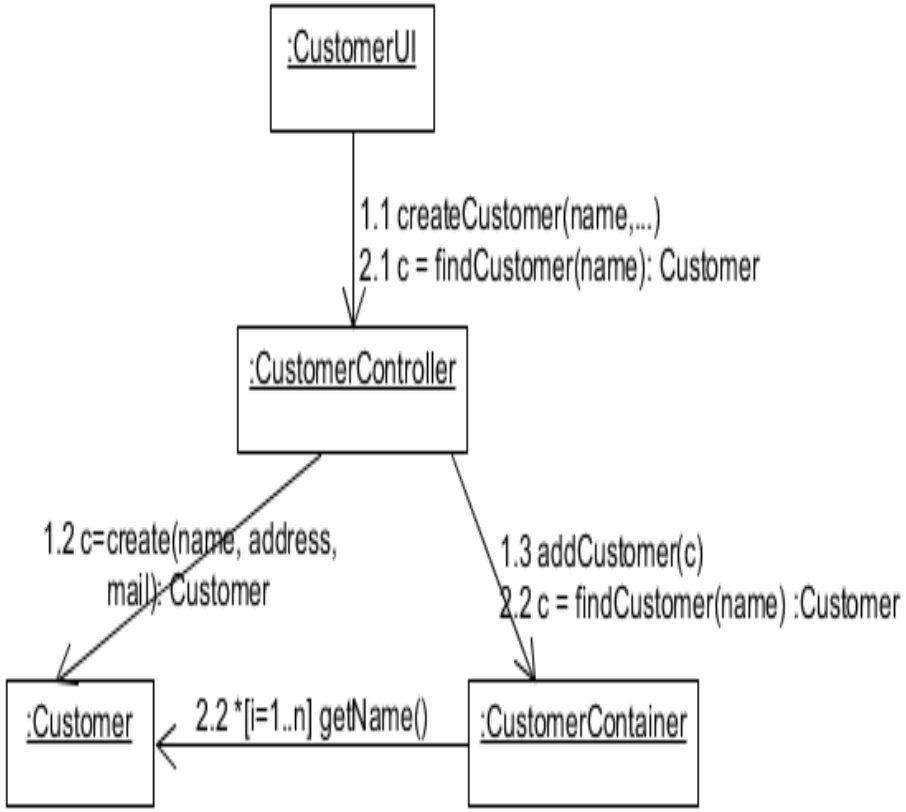
Designklassediagram



(repetition) Design af søgning
Systemhændelse: *findCustomer(name)*

Alternative visninger



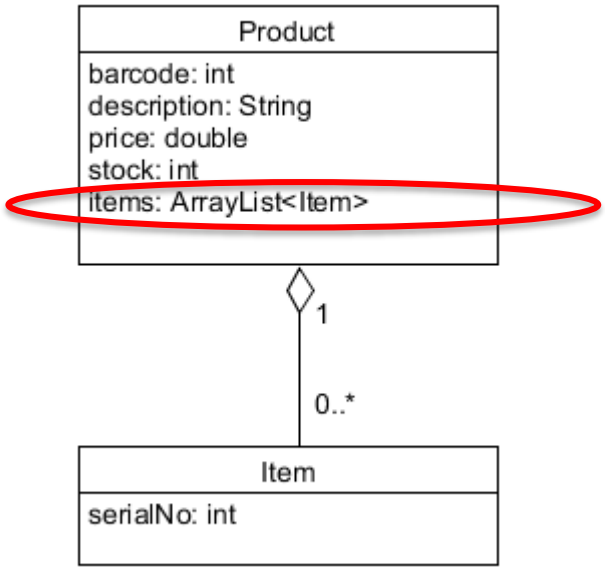


Design af aggregeringer (Helhed -> del)

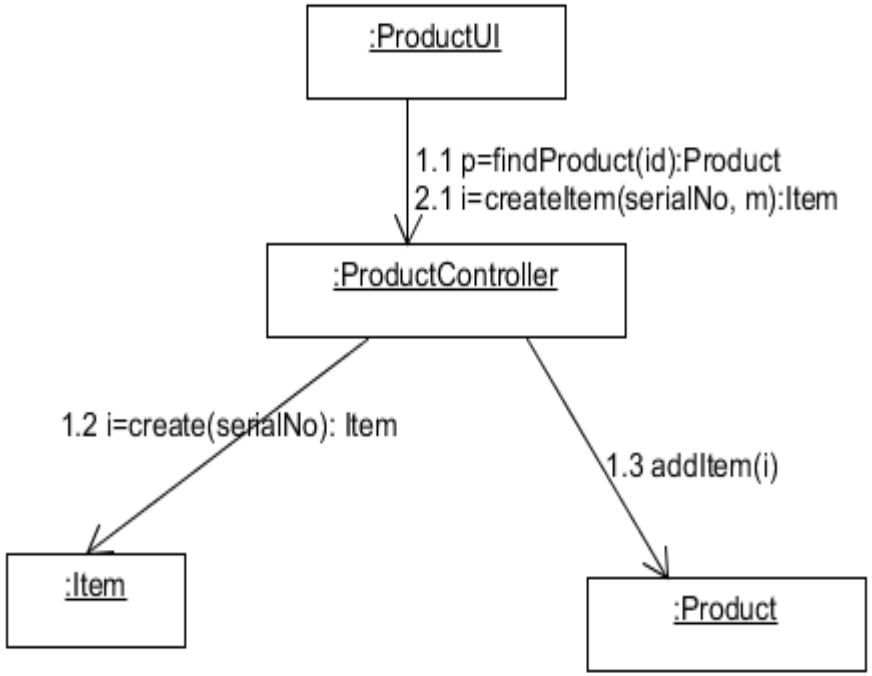
Registrering af ”delene”, fx *opretEksemplar*

Operationskontrakt: createItem(serialNo)
Use case: handle item CRUD
præcondition: en instans p af Product er fundet
postcondition:
- en instans i af Item blev oprettet
- i blev aggregeret til p
- i.serialNo blev tilskrevet værdi

Product klassen fra domænemodellen bliver container for instanser af Item.
Der tilføjes en ArrayList af Item.



Design af interaktionen for createItem
UML komunikationsdiagram anvendt



Design mønstre

- Et mønster er en løsning på et typisk (design)problem, som med mindre modifikationer kan anvendes i mange sammenhænge

> Most simply, a pattern is a named problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations

Design mønstre

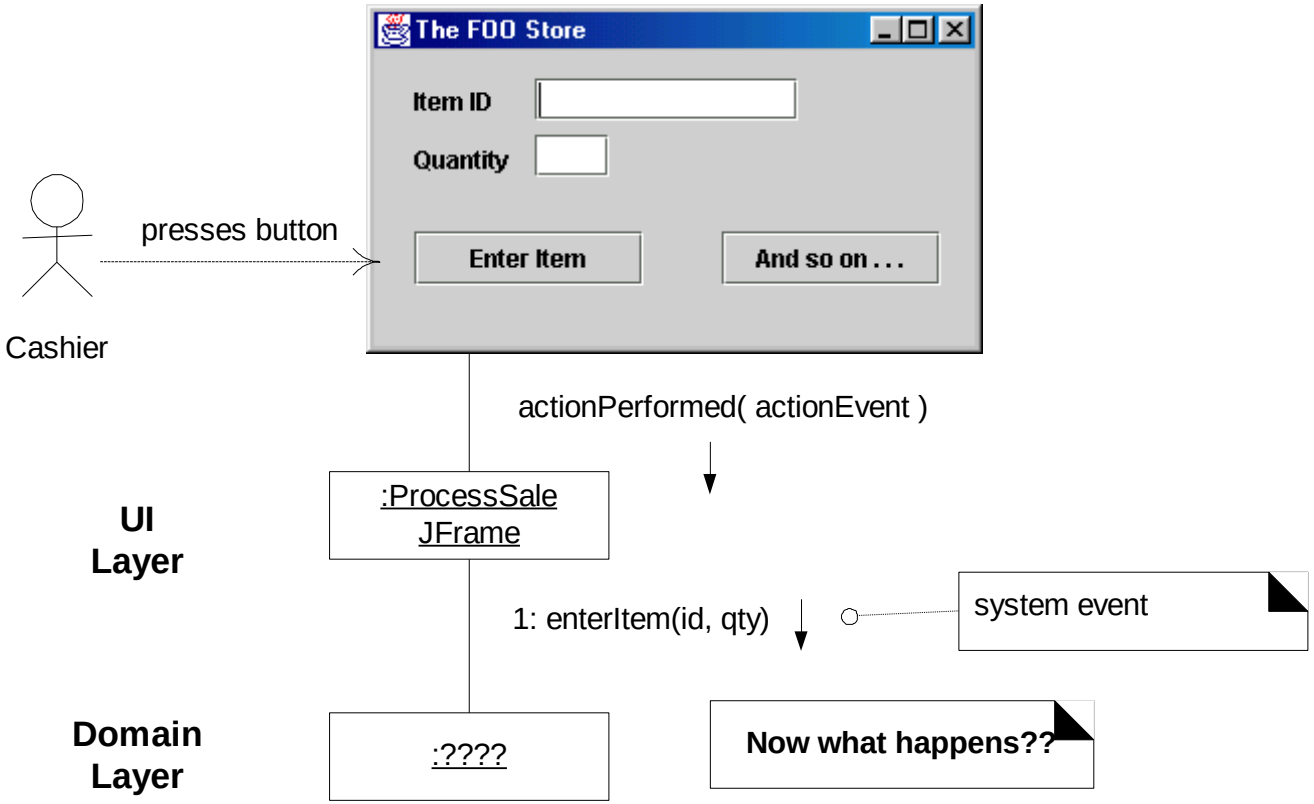
- Mønster beskriver en løsning på et problem
- GRASP mønstre: **General Responsibility Assignment Software Patterns**
  - Principper for hvordan ansvar skal tildeles objekter
- GoF mønstre: **Gang of Four**
  - Singleton

Design mønstre I skal kunne

- GRASP
  - Controller
  - Informationseksperten (indirekte ellers 2 semester)
  - Creator (delvis – ellers 2 semester)
  - Lav kobling og høj samhørighed
- GoF
  - Singleton

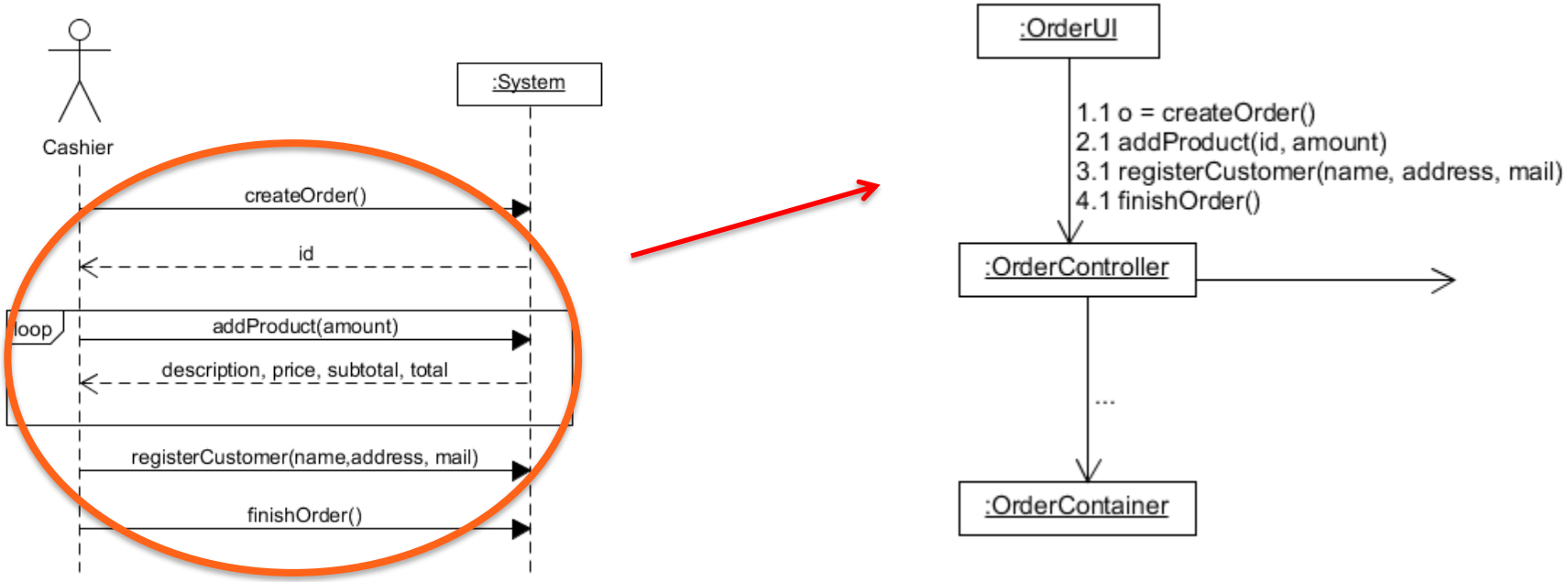
(repetition) Eksempel: GRASP controller

- Hvilket objekt skal modtage systemhændelsen fra grænsefladen?
- Tildel ansvaret til et **kontroller objekt** der repræsenterer et af følgende valg:
  - En systemklasse
  - En klasse der repræsenterer use case funktionaliteten



(repetition) Konsekvens af controller

- Et controller layer
- Systemhændelserne fra SSD skal overføres som metodekald (beskeder) fra UI til Controller



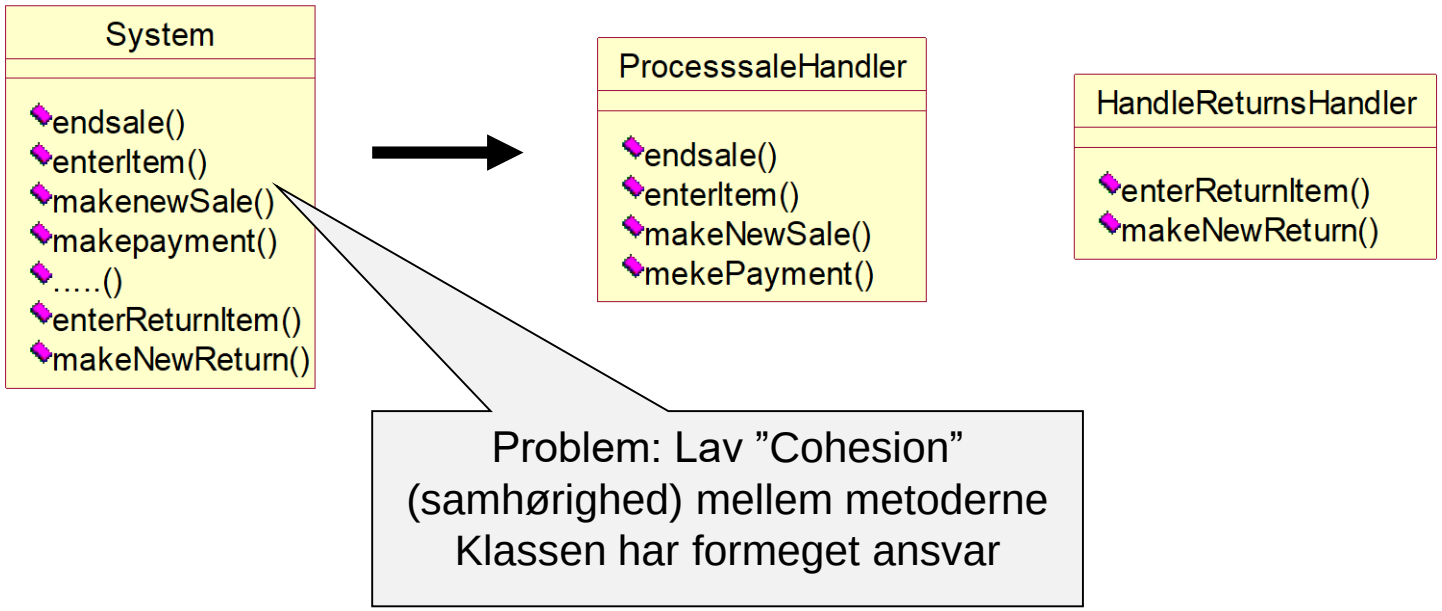
GRASP høj samhørighed mønstret

- **Problem**: Hvordan holdes kompleksiteten lav?
  - høj samhørighed udtrykkes ved, at en klasses ansvar er tæt sammenhængende og kan beskrives kort og klar
- **Løsning**:
  - Tilskriv ansvar sådan at samhørigheden forbliver høj
- **Fordele:**
  - let forståelig
  - lettere vedligeholdelse
  - betyder ofte lav kobling (men ikke altid)
  - fremmer genbrug

GRASP høj samhørighed: En kontroller klasse per use case

Konsekvens af valg af én system controller

Konsekvens af valg af een controller per use case



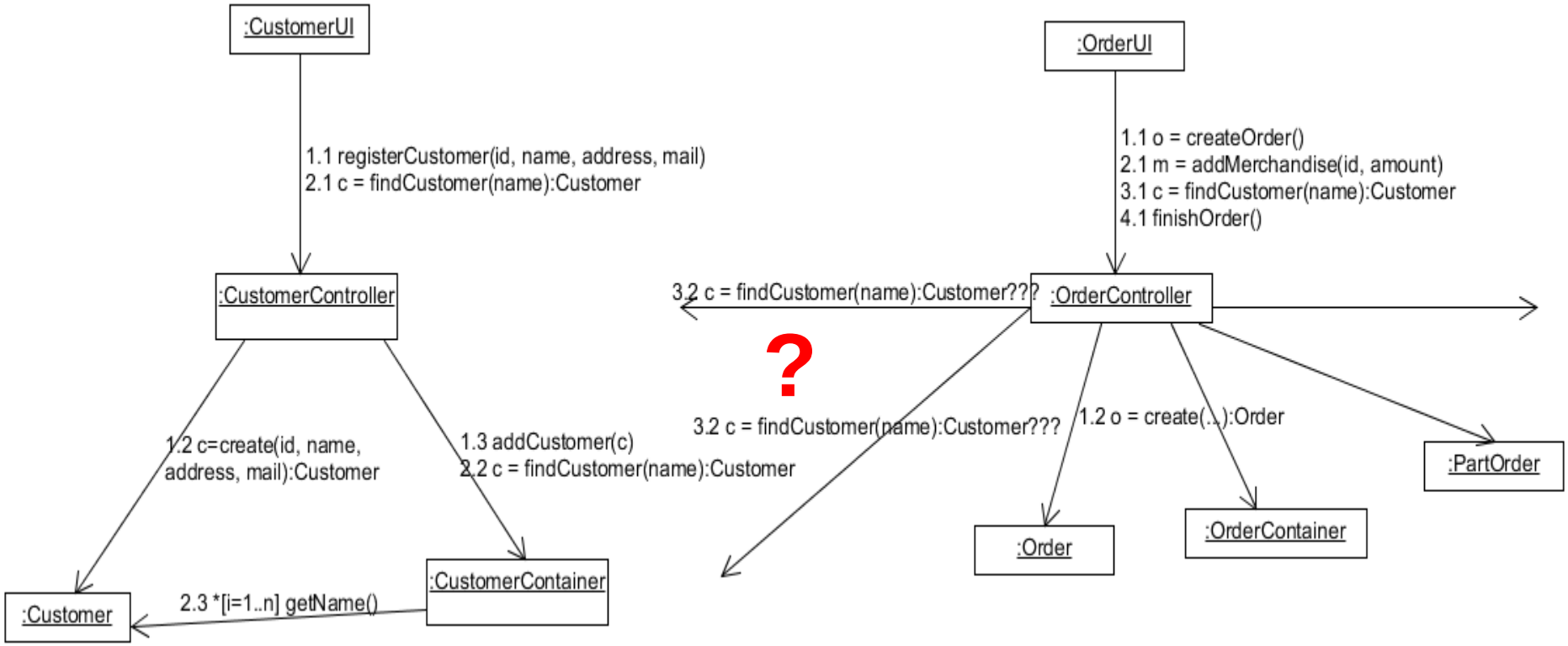
GRASP lav kobling mønstret

- **Problem som løses:**
  - Hvordan understøttes lav afhængighed mellem systemets dele og hvordan fremmes genbrug?
- **Løsning**
  - Tilskriv ansvar så at kobling i systemet forbliver lav
- **Fordele:**
  - lav kobling modvirker ”vandrende fejl”
  - fremmer genbrug
  - fremmer forståelse af den enkelte klasse

GRASP lav kobling overvejelser

Design af use casen: Opret ordre. Hvor skal kaldet 3.1 *findKunde* gå til:

1. Direkte til *KundeContaineren* i modellaget?
2. Via *KundeCtr*, hvor der allerede er et kald *findKunde* til *KundeContaineren*?



Design af mere komplekse use cases

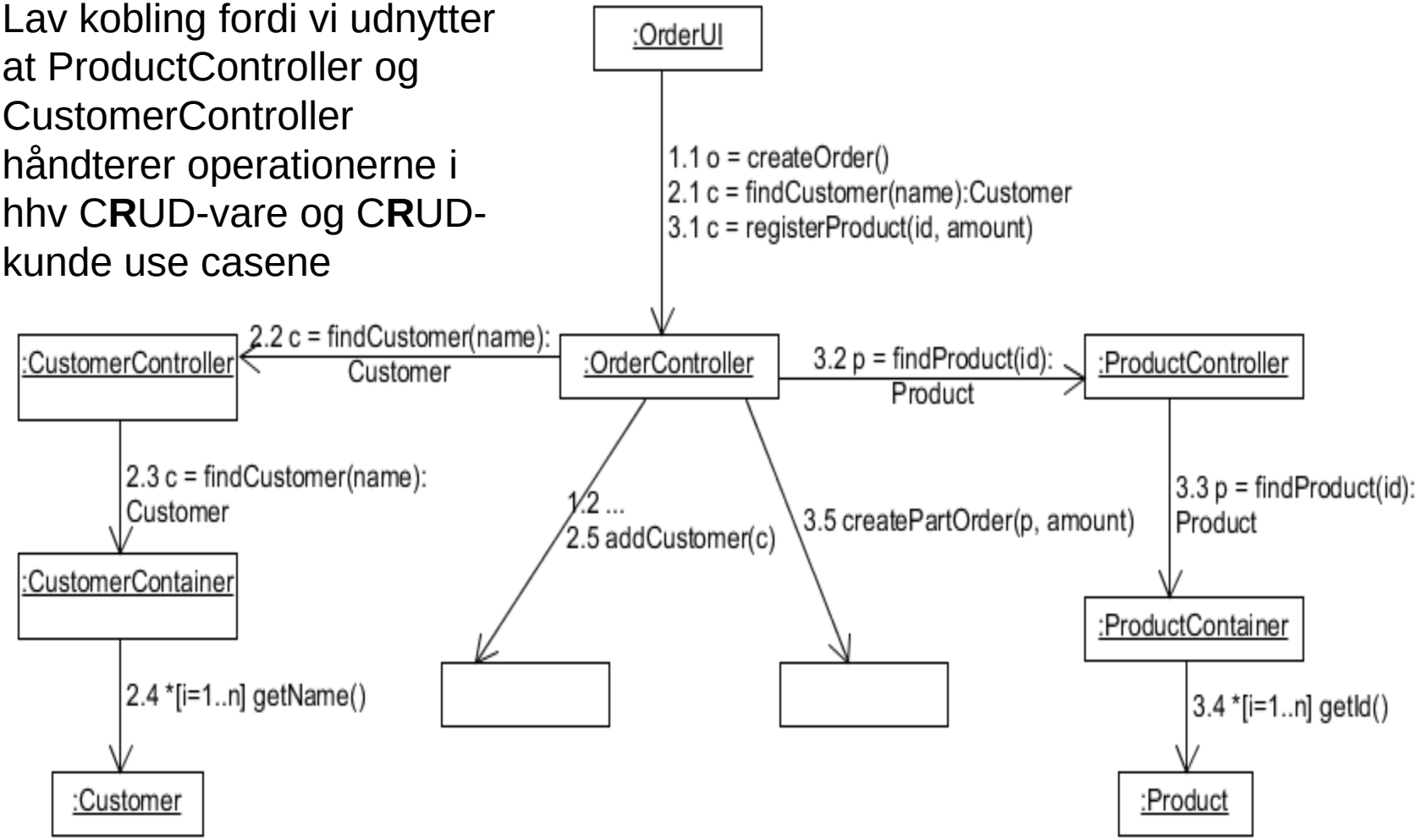
- De mere komplekse use cases opererer på mere end en klasse i domænemodellen
- CRUD funktionalitet indgår ofte som trin i en eller flere af de mere komplekse use cases
- Fx anvender use casen: *Opret Salg* systemhændelsen:
  - *findCustomer(id)* som også indgår i *Håndter kunde - CRUD* use casen (måske man skulle søge på navn i stedet)
  - *findProduct(id)* som også indgår i *Håndter vare – CRUD use* casen
- For at opfylde GRASP håndteres dette ved genbrug af Controller klasser

Genbrug af controllere

- Reglen om at der i udgangspunktet skal anvendes en controller per use case betyder fx at der i forbindelse med et salgssystem fx skal være:
  - En *CustomerController* til håndtering af use casen: *Håndter kunde-CRUD*
  - En *ProductController* til håndtering af use casen: *Håndter vare –CRUD*
  - En *SaleController* til håndtering af use casen: *Registrer Salg*
- For at opnå lav kobling mellem lagene genanvendes CRUD controllere – dvs at **SaleController genbruger de respektive CRUD controller klasser**
  - lavere kobling mellem controllerlayer og modellayer
  - rettes i modelklasserne har dette kun konsekvens for *een* controller klasse

Eksempel på genbrug af controllere
(viser princippet – ellers er diagrammet mangelfuldt)

Lav kobling fordi vi udnytter at ProductController og CustomerController håndterer operationerne i hhv C**R**UD-vare og C**R**UD-kunde use casene



GRASP creator og GOF singleton

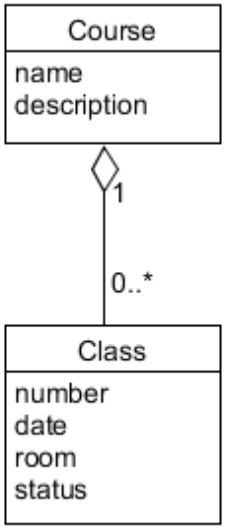
- Nye mønstre der betyder, at vi skal overveje at lave lidt om i det design vi allerede har lavet
- Creator siger fx at vi kan uddelegere ansvaret for oprettelse af objekter, hvis vi har aggregeringer i domænemodellen, og derved opnå en lavere kobling
- Det har dog også ulemper!

Creator mønstret

- **Navn: Creator**
- **Problem:**
  - Hvilket objekt skal have ansvaret for at oprette nye forekomster **A** af en klasse ?
- **Løsning:**
  - Vælg et objekt **B,** så at :
    - B indeholder eller **aggregerer** A (et alternativ)
    - B registrer A
    - B bruger ofte A
    - **B har de initierende data for A - dvs controller klasser** (kontrolobjektet har initierende data fra grænsefladen og er i udgangspunktet creator – vores praksis)

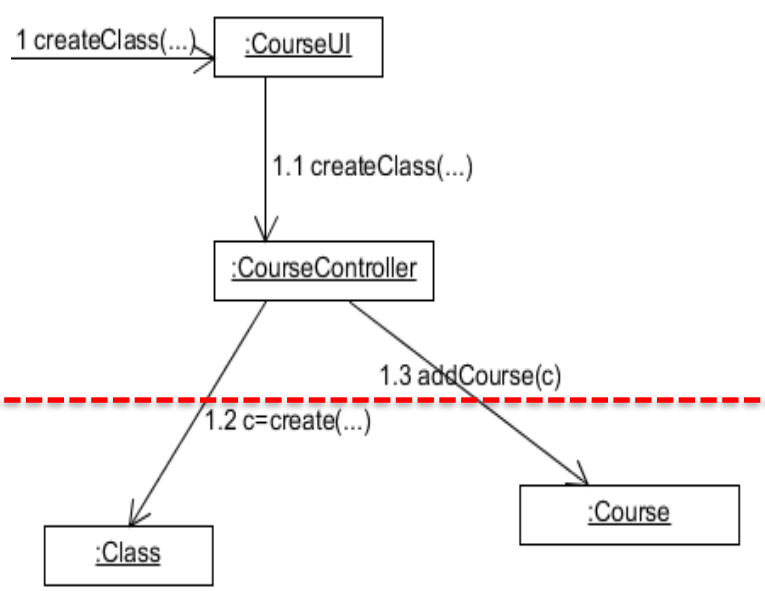
Hvis der er aggregeringer i domænemodellen kan I overveje at ændre i den praksis vi hidtil har anvendt!!!

Hvor bibeholdes koblingen lavest?



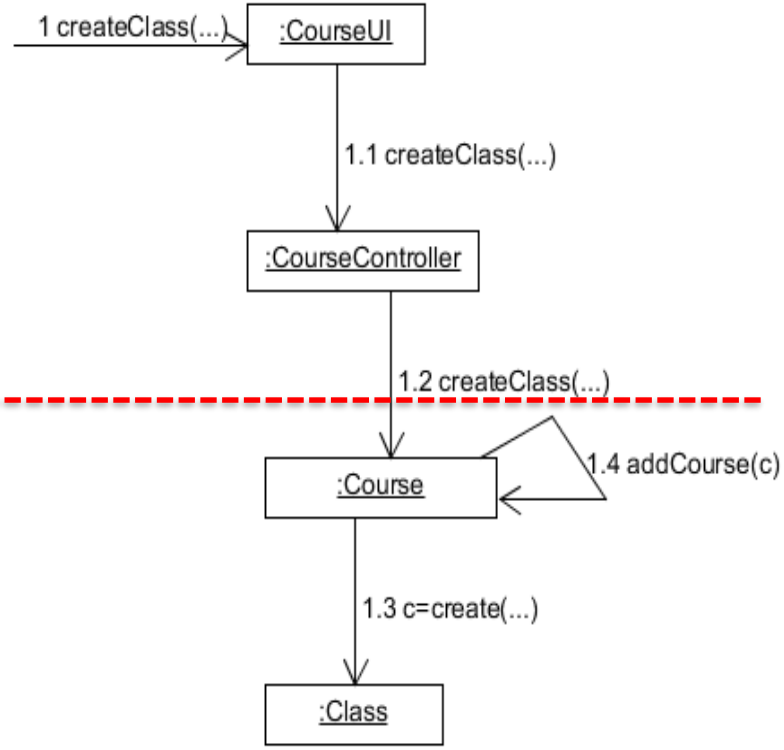
Creator Controller

Controller står for create

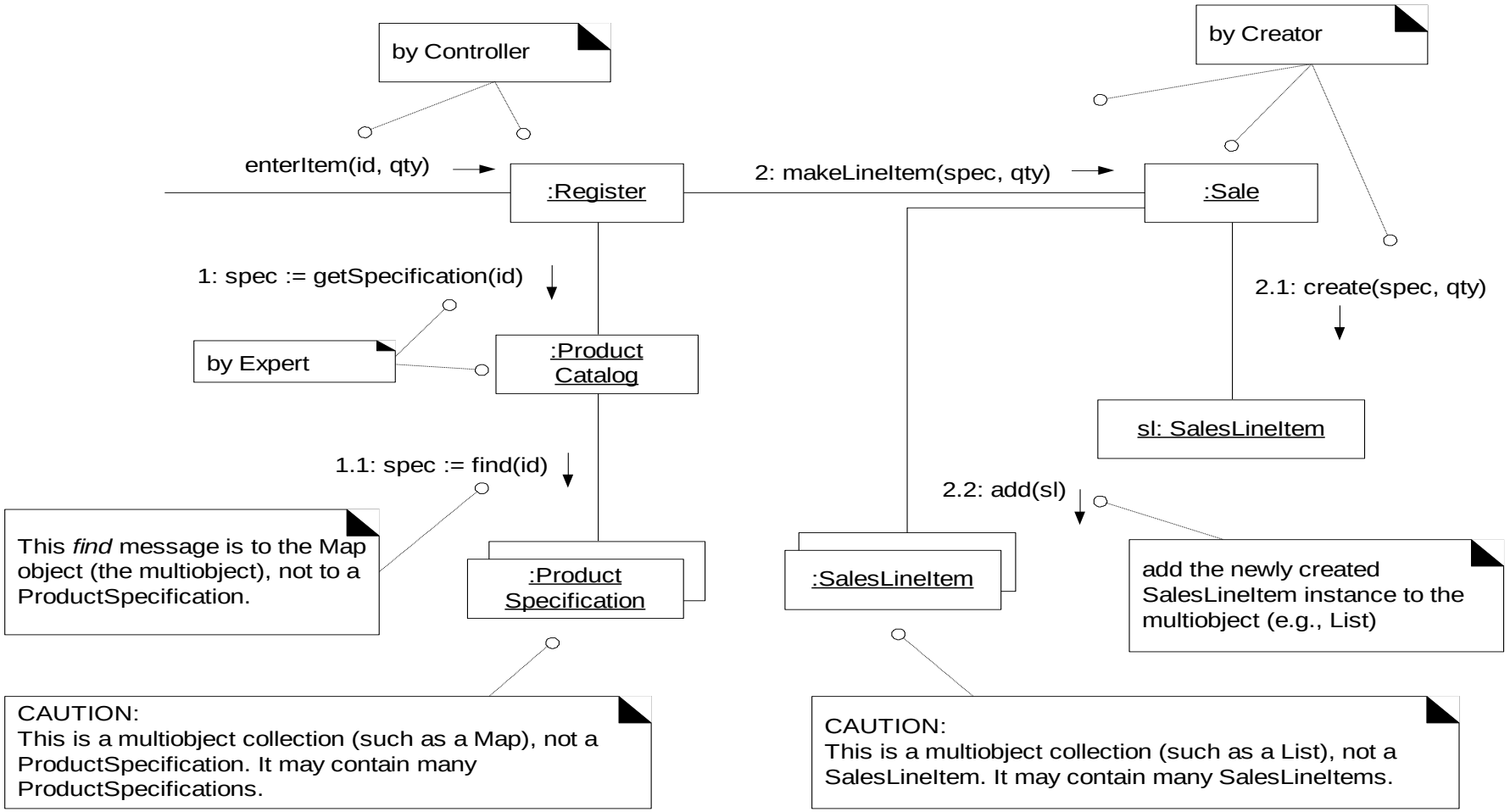


Creator Aggregering

Create uddelegeres til modellen klassen der ”aggregerer”



Eksempel: Design af enterItem(id,qty) med GRASP påført



GRASP opsummeret

- Creator
- Informations expert
- Low coupling
- Controller
- High cohesion

GOF Singleton pattern

- Problem: Der tillades kun en instans af en klasse – en singleton (et globalt ”access point”)
- Løsning: Definer en statisk metode af en klasse som returnerer en singleton

The ”Highlander” pattern:
There can be only one!

Singleton 101 (jf. programmeringsundervisning)

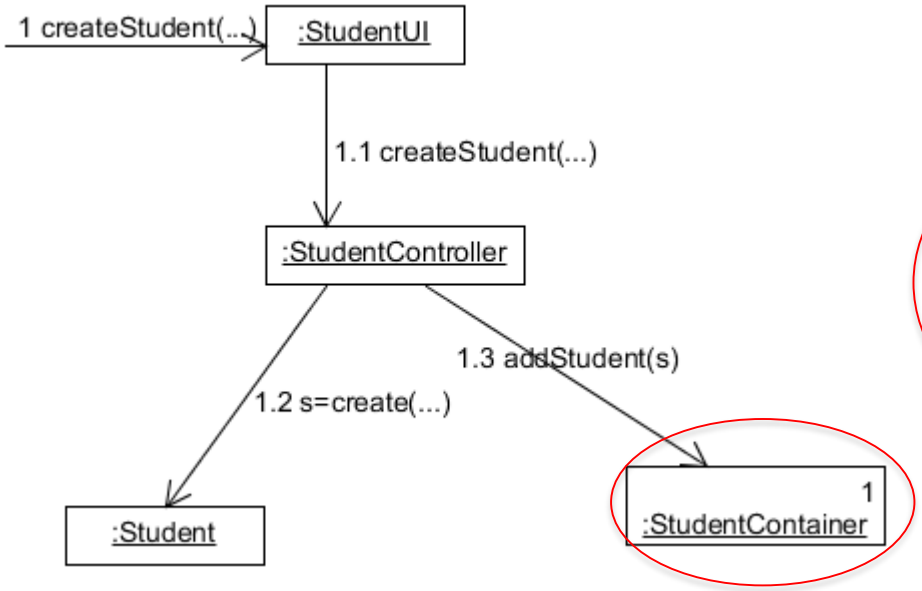
- Private constructor
- Private static variable **instance**
- Public static **getInstance()** metode

- instance er af samme type som klassen

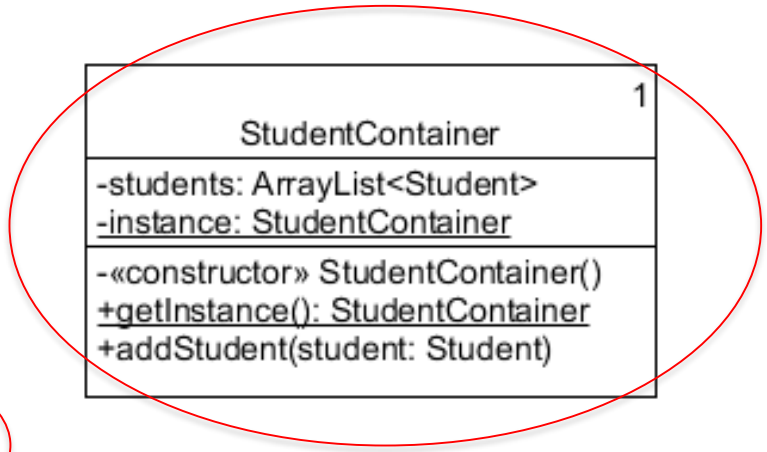
Eksempel: UML Singleton notation

Interaktion: *createStudent(…)*

(kommunikationsdiagram)



Designklasse: StudentContainer



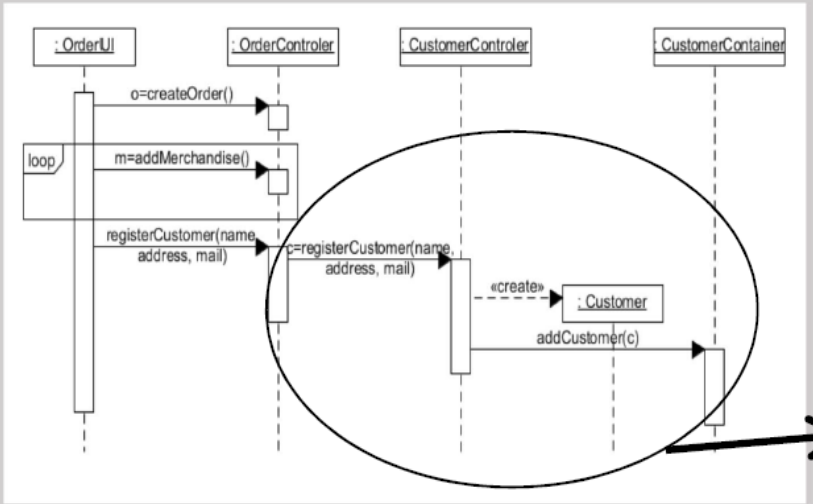
Kodeeksempel: Containerklasse som Singleton

```
package modellayer;
import java.util.*;

public class StudentContainer {
  private List<Student> students;
  private static StudentContainer instance;

  private StudentContainer() {
    students = new ArrayList<>();
  }

  public static StudentContainer getInstance() {
    if (instance == null) {
      instance = new StudentContainer();
    }
    return instance;
  }
  …
}
```

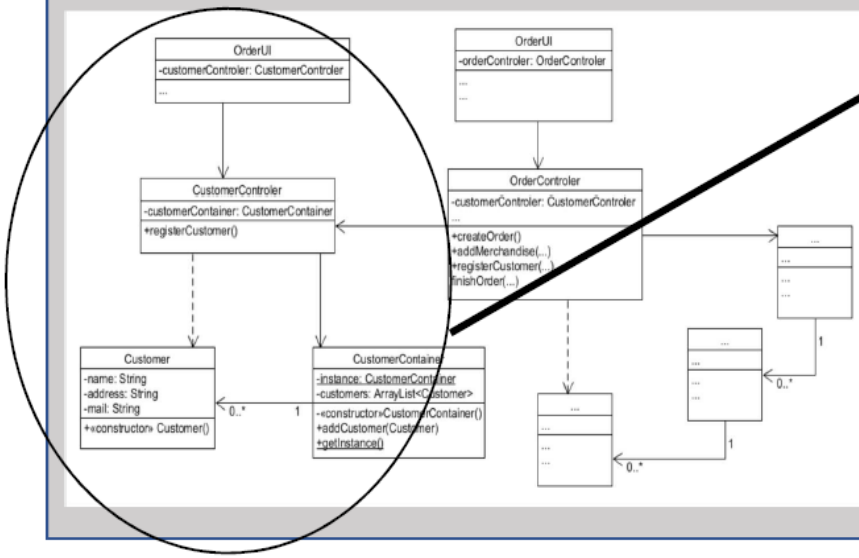
Instansen hentes fra Controller klassen ved:
StudentContainer studentContainer = StudentContainer.getInstance()

Design

Interaktionsdiagram for use case Registrer Ordre



Designklassediagram. Laves fra interaktionsdiagrammer



Kode

```
package uilayer;

public class CustomerUI {

 private CustomerController customerController;

 public CustomerUI() {

  customerController = new CustomerController();

 }

 private void registerCustomer() {

    int id = inputId();

    String name = inputName();

    String address = inputAddress();

    String mail = inputMail();

    customerController.registerCustomer(id, name, address, mail);

 }

}
```

```
package controllerlayer;

public class CustomerController {

 private CustomerContainer customerContainer;


 public CustomerControler() {

  customerContainer = CustomerContainer.getInstance();

 }

 public void registerCustomer(int id, String name, String address, String mail) {

  Customer customer = new Customer(id, name, address, mail);

  customerContainer.addCustomer(customer);

 }

}
```

```
package modellayer;

public class CustomerContainer {

 private static CustomerContainer instance;

 private ArrayList<Customer> customers;

 private CustomerContainer() {

  customers = new ArrayList<Customer>();

 }

 public static CustomerContainer getInstance(){

  if (instance == null) {

   instance = new CustomerContainer();

  }

  return instance;

 }

 public void addCustomer(Customer customer){

     customers.add(customer);

  }
```

```
package modellayer;

public class Customer{

 private int id;

 private String name;

 private String address;

 private String mail;

 public Customer(int id, String name, String address, String mail){

   this.id = id;

  this.name = name;

  this.address = address;

  this.mail = mail;

 }

 //getters and setters for all variables

}
```

Opgaver

- I skal løse opgaverne 1 – 6
  - Husk fokus skal ligge på diagrammer og sikring af at der er overensstemmelse mellem interaktionsdiagrammer og designklassediagrammer