Design

Arkitektur og GRASP design pattern
Design af en CRUD funktionalitet
Design af interaktion
Design klassediagram

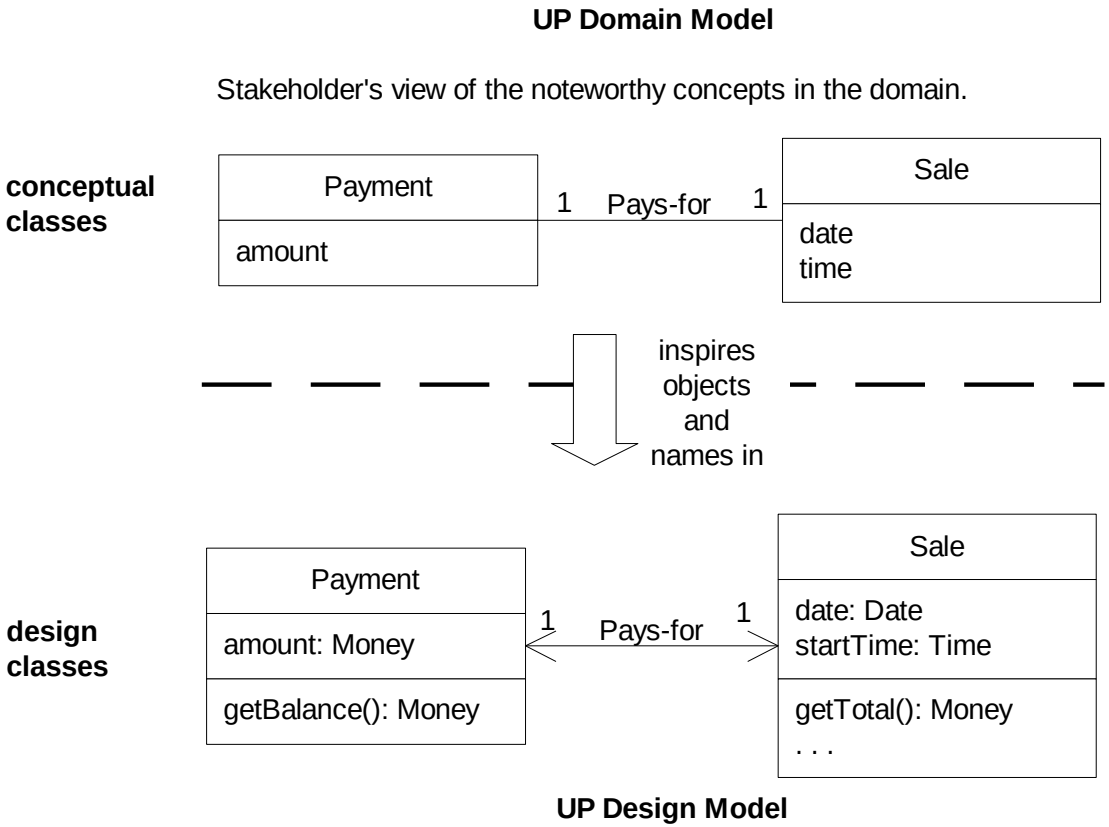
Fra Krav til Design

- Ved specifikation af krav betragtede vi systemet ***udefra,*** og opstillede kravene til systemet med udgangspunkt i systemets omgivelser
- I design betragtes systemet ***indefra***. Spørgsmålet er:
  - *Hvordan skal objekterne interagere eller samarbejde for at kunne understøtte kravene (use cases)?*
  - *Hvilket ansvar (operationer) skal de enkelte klasser have?*
  - *med udgangspunkt i en valgt teknisk platform?*

Fra Krav til Design

- I design bygges videre på følgende resultater fra kravs- og analyse fasen:
  - use case model (funktionerne i systemet)
  - domænemodellen (informationen i systemet)
    - synonym: model af problemområdet
- De vigtigste resultater i design
  - use case realiseringer i **interaktionsdiagrammer** (UML kommunikations- eller sekvensdiagrammer)
  - **design klassediagram**

Eksempel: Fra Krav til Design



The object developer has taken inspiration from the real-world domain in creating software classes. Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Designgrundlag

- Krav
  - Domænemodel
  - SystemSekvensDiagrammer (SSD)
  - Operationskontrakter
- Arkitektur
- Designmønstre (GRASP og GOF)
- De vigtigste resultater i design
  - use case realiseringer i **interaktionsdiagrammer** (UML kommunikations- eller sekvensdiagrammer)
  - **design klassediagram**

Oversigt

Krav

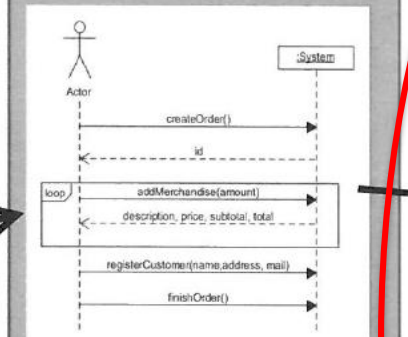
Fully dressed for de højest prioriterede use cases

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



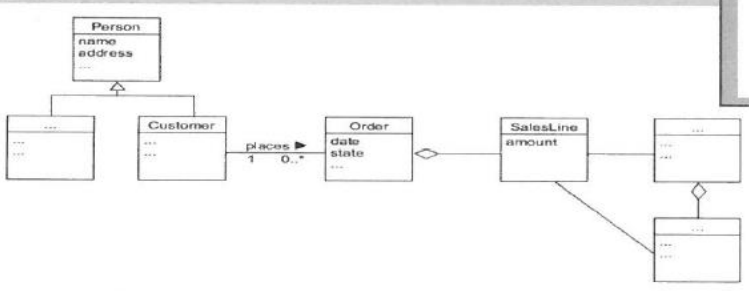
Operations kontrakter

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

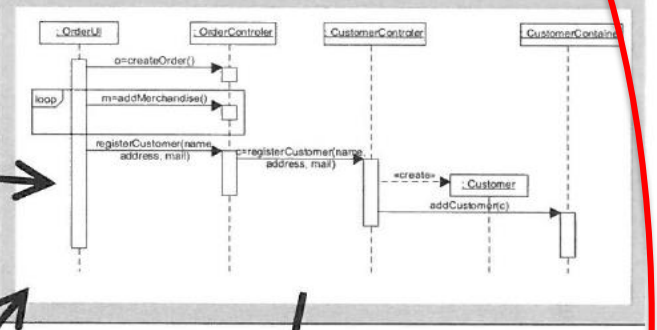
Domænemodel



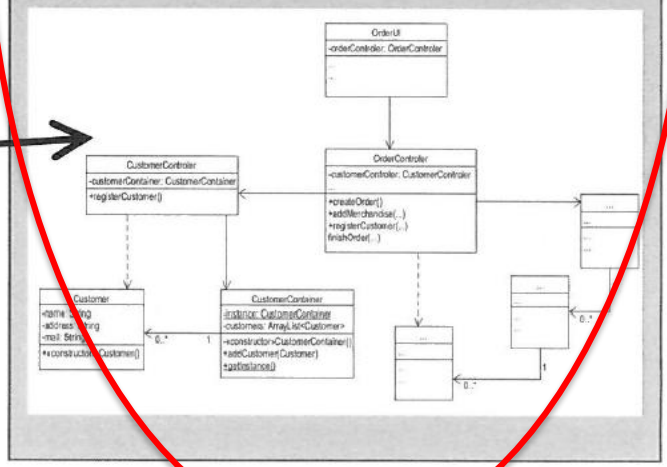
Design

Vi bruger også arkitekturen til design

Interaktionsdiagram for Registrer Ordre



Designklassediagram, laves fra interaktionsdiagram



Opgaverne i denne lektion

- Formålet med opgaverne i denne lektion er, at I skal lære at designe ud fra specificerede krav
- De første opgaver dækker krav man normalt ikke vil bruge tid på at specificere i dybden (CRUD), men I skal gøre det her for læringens skyld!!!
- Det er samme type opgaver I skal løse i tema design

Arkitektur

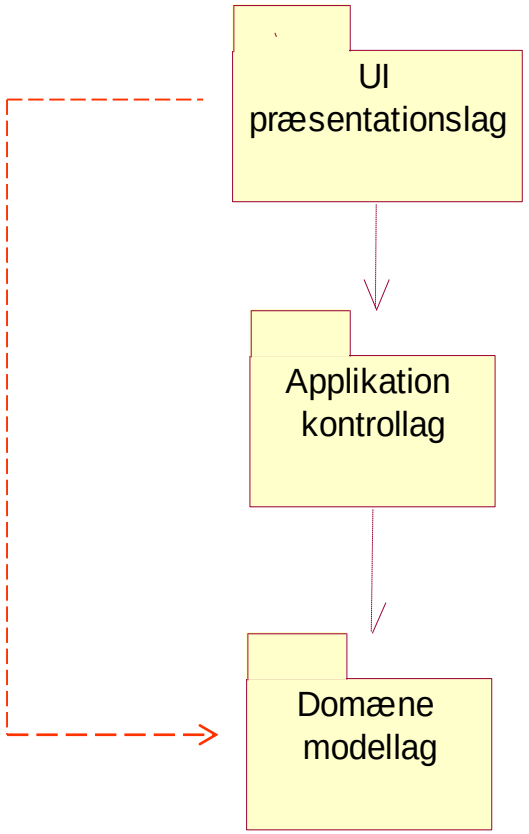
- Et systems arkitektur fastlægger et systems overordnede **struktur**, dvs. dets komponenter og deres sammenhæng
- En god arkitektur **reducerer kompleksiteten** og gør det lettere at **forstå** og **vedligeholde** systemet Arkitekturen danner rammerne for designet
- Komponenterne beskrives i **UML** som **pakker** og forbindelserne mellem dem beskrives som **afhængigheder**

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 (evt. ArrayList'er)

Realisering af use cases

- Design af use cases kaldes for **use case realisering**
- Dette sker i form af samarbejdende objekter
- Hvordan samarbejdet skal ske beskrives i et UML interaktions diagram(kommunikationsdiagram eller sekvensdiagram)
- Men først findes de designklasser som skal indgå i use case realiseringen

Designklasser

- **Designklasserne** som indgår i **use case realisering** findes med udgangspunkt i den **lagdelte arkitektur** samt domænemodellen
- Den lagdelte arkitektur kræver en designklasse for hvert lag:
  - **Grænsefladeklasser (Uiklasser)** der håndterer interaktionen mellem aktøren og grænsefladen. Sender systemhændelsen videre til controllerklassen.
  - **Controllerklasser** der får ansvaret for at danne ”limen” mellem grænseflade- og domæneklasserne. Håndterer forretningslogik. Typisk er der en per use case.
  - **Modelklasser** der findes ud fra domæne modellen af problemområdet. De modelklasser der understøtter use cases medtages. Her på 1. semester har vi også containerklasser med

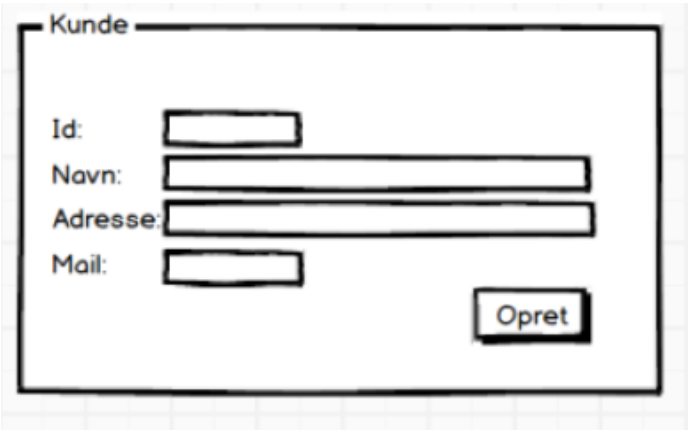
Eksempel: Design af Kundekartotek

- Principperne i design illustreres i det følgende gennem design af CRUD funktionalitet: *Håndter kunde – CRUD* **normalt er det kun komplekse der specificeres på denne måde, her blot for eksemplets skyld**
- Fremgangsmåden er følgende:
  - Der tages **udgangspunkt i kravene** dvs. **domænemodel, SSD** og **kontakter**
  - Først findes de relevante **designklasser**, som indgår i designet af use casen med udgangspunkt i 3 lags arkitekturen (som I også har hørt om i programmering)
  - Dernæst designes **interaktionen** mellem de involverede objekter
  - Endelig udarbejdes et **designklassediagram**

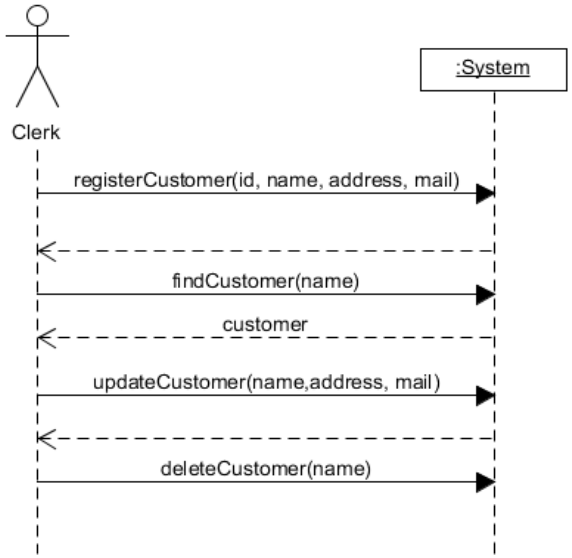
Kravene der skal designes ud fra
Eksempel: Kundekartotek

Domæne model

Mock up



Systemsekvensdiagram



Use case –Fully dressed

Use case: *Håndter kunde – CRUD*

Registrer kunde

Præcondition: Ingen

Postcondition: En kunde er blevet registreret

Successcenarie:

1. Sekretæren angiver kundeoplysninger(navn, adresse, mail…)
2. Systemet accepterer oplysningerne og opretter en ny person

FindKunde

……

OpdaterKunde

…..

SletKunde

…….

Operationskontrakt

Operation: registerCustomer(id, name, address, mail)

Use case: Håndter kunde CRUD

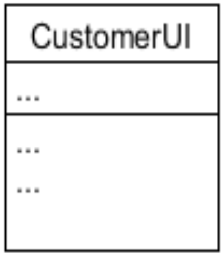
Pre betingelse: Ingen

Postbetingelse:

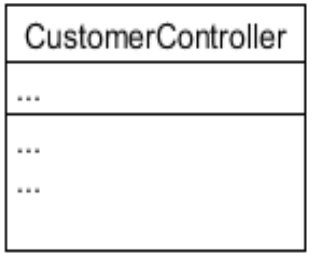
- et Customer objekt c blev oprettet
- c.id, k.name, c.address, c.mail blev tilskrevet værdier

Valg af designklasser
Eksempel: Kundekartotek

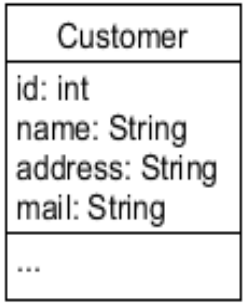
UI lag:
Der tilføjes et UI objekt til at håndtere UI funktionalitet, dvs. felter, knapper events til Controller lag

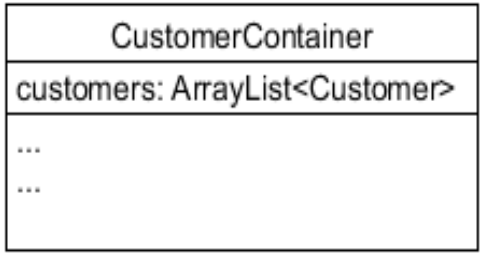


Controller lag:
Der tilføjes et kontroller objekt til at håndtere systemhændelsen fra UI laget og sørge for udførelser ift. modellaget



Modellag:
Relevante klasser fra domænemodellen dvs. Customer klassen samt klasser til håndtering af samlingen af personer, dvs en CustomerContainer med en ArrayList <Customer>.





CRC Øvelse: Klassernes ansvar?

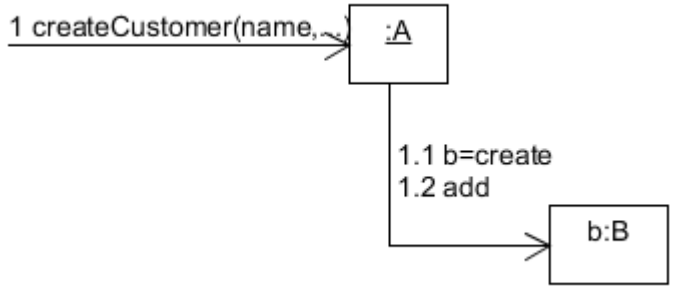
Eksempel: Kundekartotek

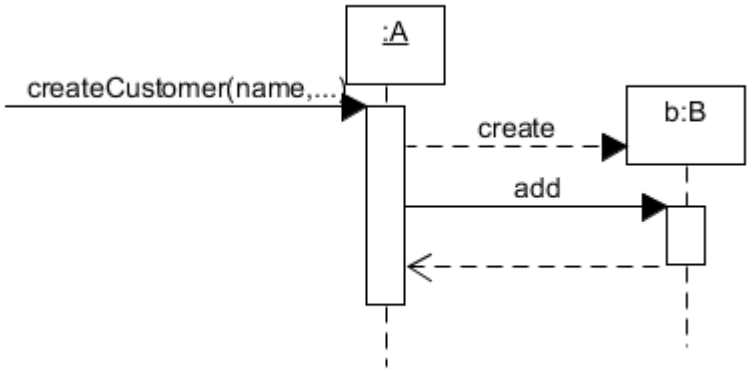
- Opgave: Design af interaktion for systemhændelsen: *registerCustomer(id, name, address, mail)*
- Deltagende objekter: CustomerUI, CustomerController, CustomerContainer og Customer (se forrige slide)
- Øvelsen starter med at CustomerUI får ansvaret for at udføre metoden *registerCustomer(id, name, address, mail)* dvs registrere parametre fra UI i et objekt.
  - *Hvilke andre objekter* skal *CustomerUI* samarbejde med for at kunne fuldføre dette ansvar?
  - Hvilket ansvar skal de samarbejdende objekter have?
- CRC står for ”Class Responsibility Collaboration”

Design af interaktion
UML interaktionsdiagrammer

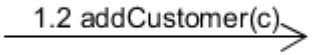
Diagramalternativer:

- Kommunikations-diagrammer:
  - objekt interaktion i graf eller netværks format, Larman s.240
- Sekvens diagrammer
  - interaktionen vises I stakit format, Larman s. 227
- Styrke/svagheder
  - Larman s. 224
  - Jeg foretrækker kommunikationsdiagrammer, men vil bruge begge typer





”Message” i UML:



Eksempel: Der sendes en besked *addCustomer(c)* fra et objekt af klassen A til en objekt af klassen B (A kalder metoden *addCustomer* på B)

Design af interaktion
Eksempel: Kundekartotek

Domæne model:

Operationskontrakt

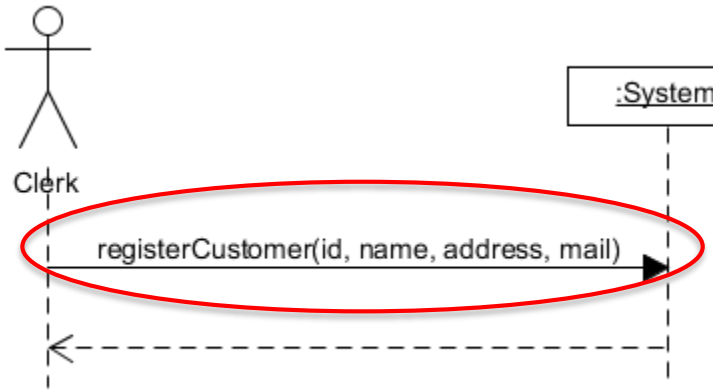
Præcondition: Ingen

Postcondition:

–En instans af Customer c blev oprettet

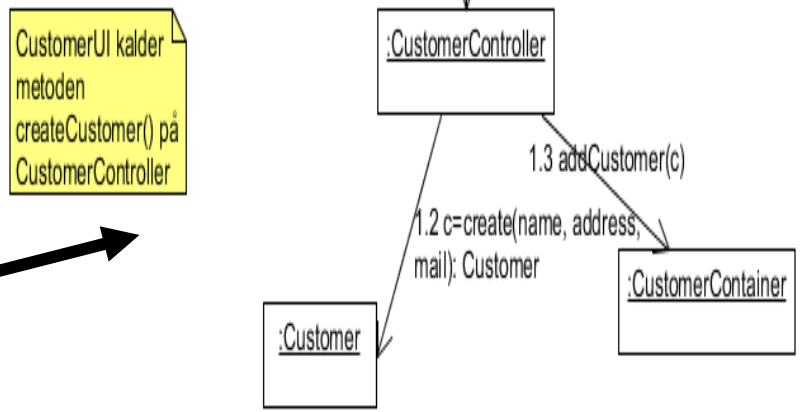
–c.customerNo, c.name, c.address, c.mail tilskrives værdier

SSD:



Design af interaktion: registerCustomer

UML kommunikationsdiagram



Lidt UML syntaks

- Meddelser, Larman s. 227
  - return = message(parameter: parameterType):returnType
  - eksempler:
    - p=getPerson(id)
    - p=getPerson(id:personId)
    - p=getperson(id:personId):Person
- create/destroy, larman s. 230 og 242
  - create(parameterliste) – fortolkes som *new*
  - *destroy er ikke relevant i Java da vi ikke har deconstructors*

Design klassediagram

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

Design af klasser og deres relationer

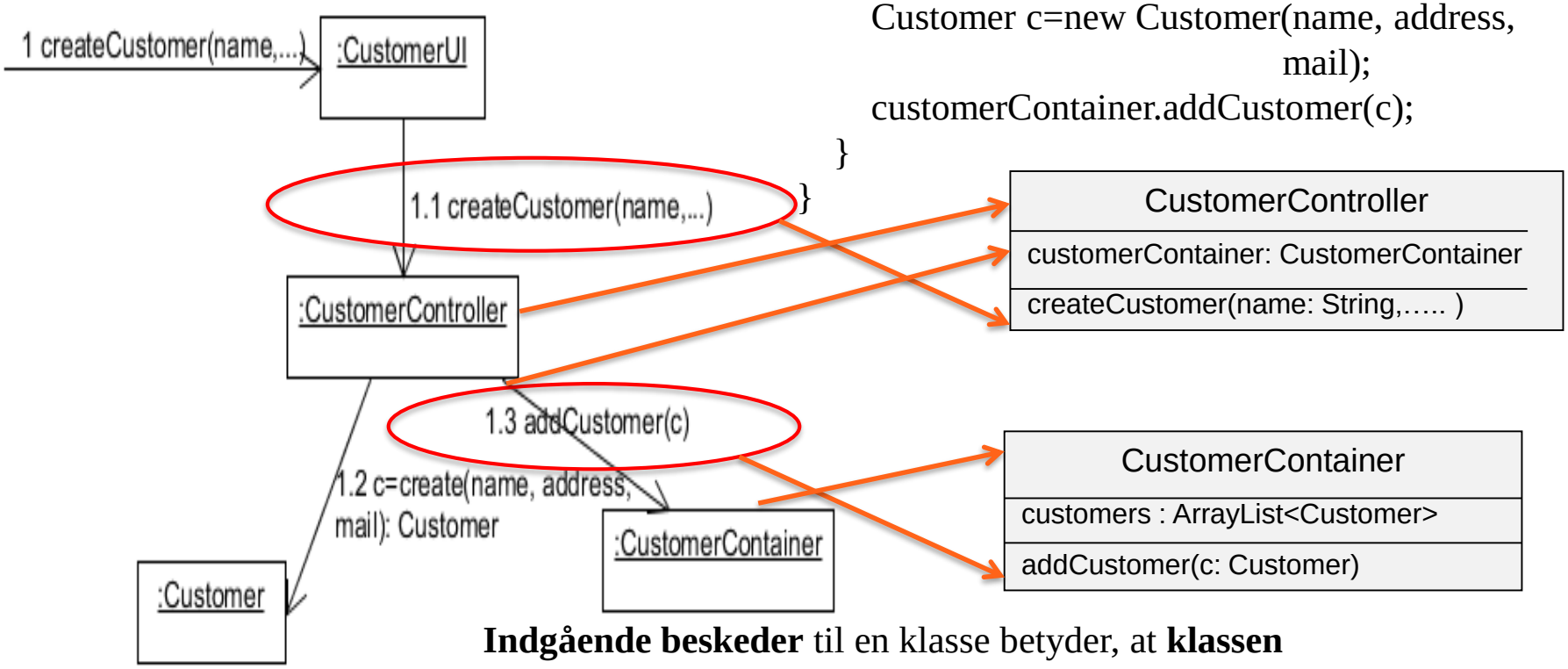
- Ved at gå interaktionsdiagrammerne igennem identificeres sammenhængen mellem de valgte designklasser ud fra krav om synlighed
- Evt. attributter overføres fra domænemodel
  - Id tilføjes hvor det er nødvendigt
- Herefter tilføjes metoderne på klasserne:
  - **Indgående beskeder** til en klasse betyder, at **klassen må definere en modsvarende metode**
  - Metoderne specificeres ved at nævne deres navn (beskeden) i operationsdelen på klassen

Design af klasser ud fra kommunikationsdiagram

Eksempel: Kundekartotek

Use case interaktion

Kommunikationsdiagram



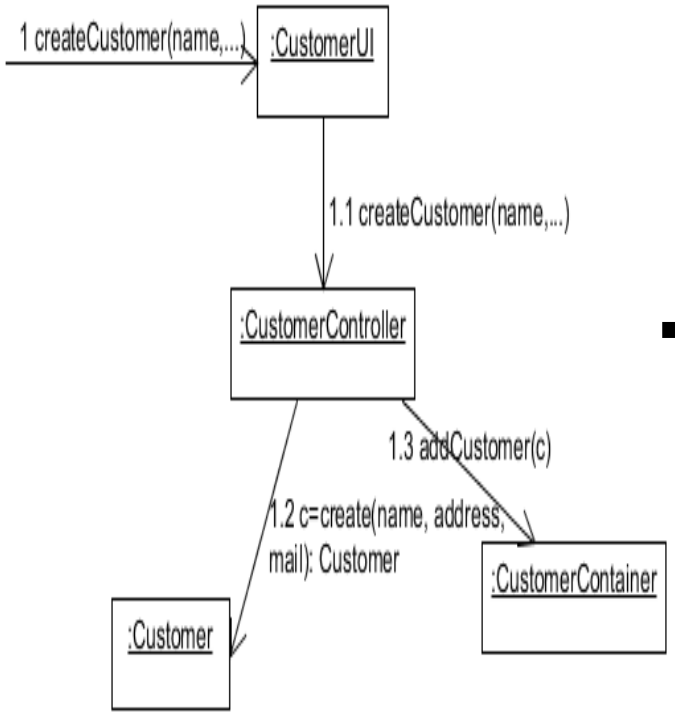

```
public class CustomerController {
   private CustomerContainer customerContainer;
   public  ….
   public void createKunde(String name, String
                                address, String mail) {
      Customer c=new Customer(name, address,
                                      mail);
      customerContainer.addCustomer(c);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

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

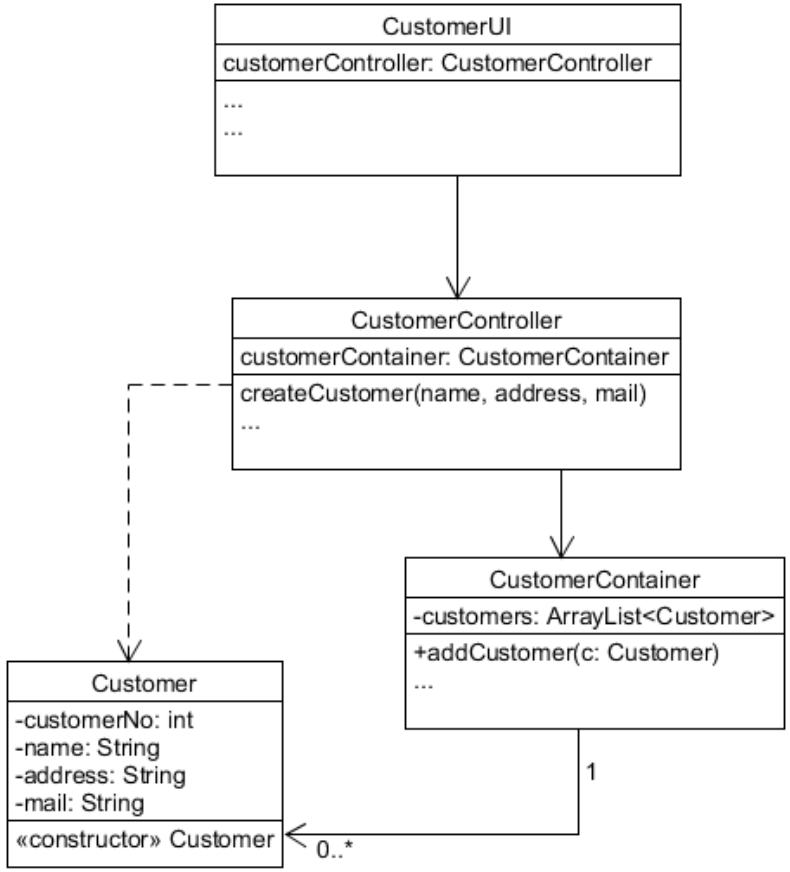
Det endelige designklassediagram
Eksempel: Kundekartotek

Use case interaktion

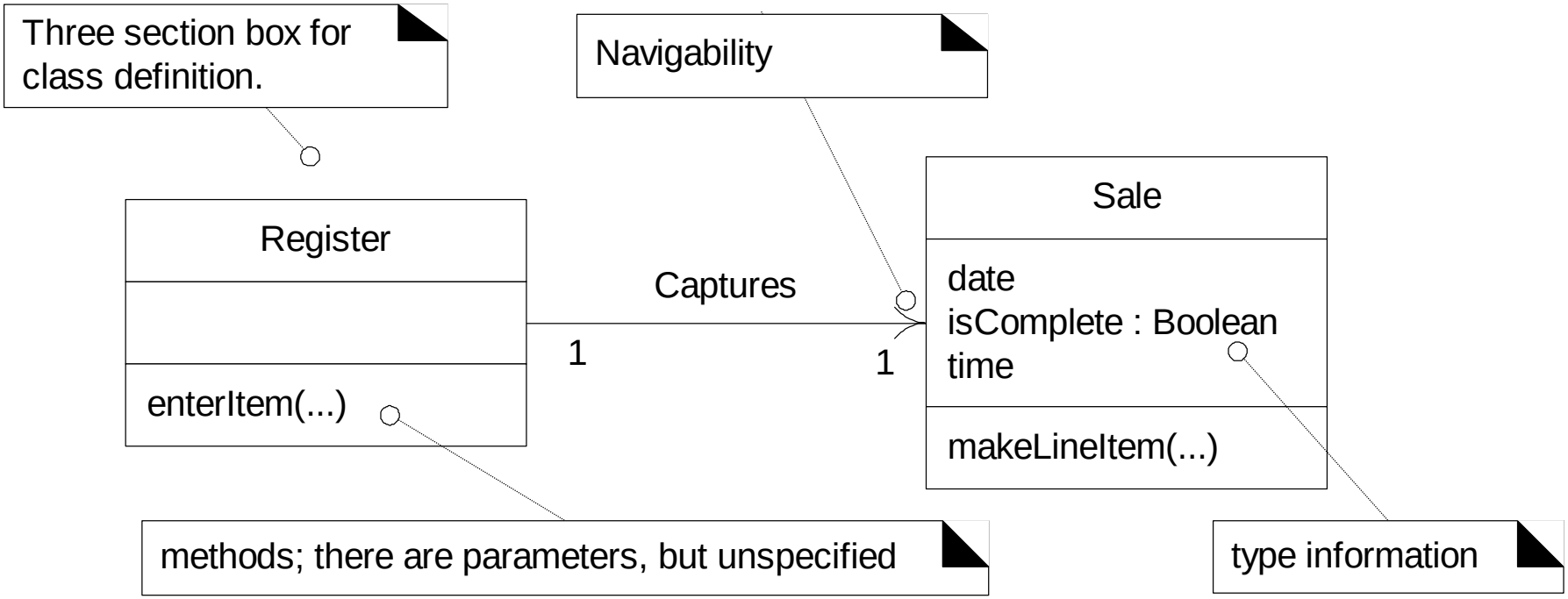
Kommunikationsdiagram



Designklassediagram



UML notation for design klasse diagram



UML – Attributreferencer (synlighed)

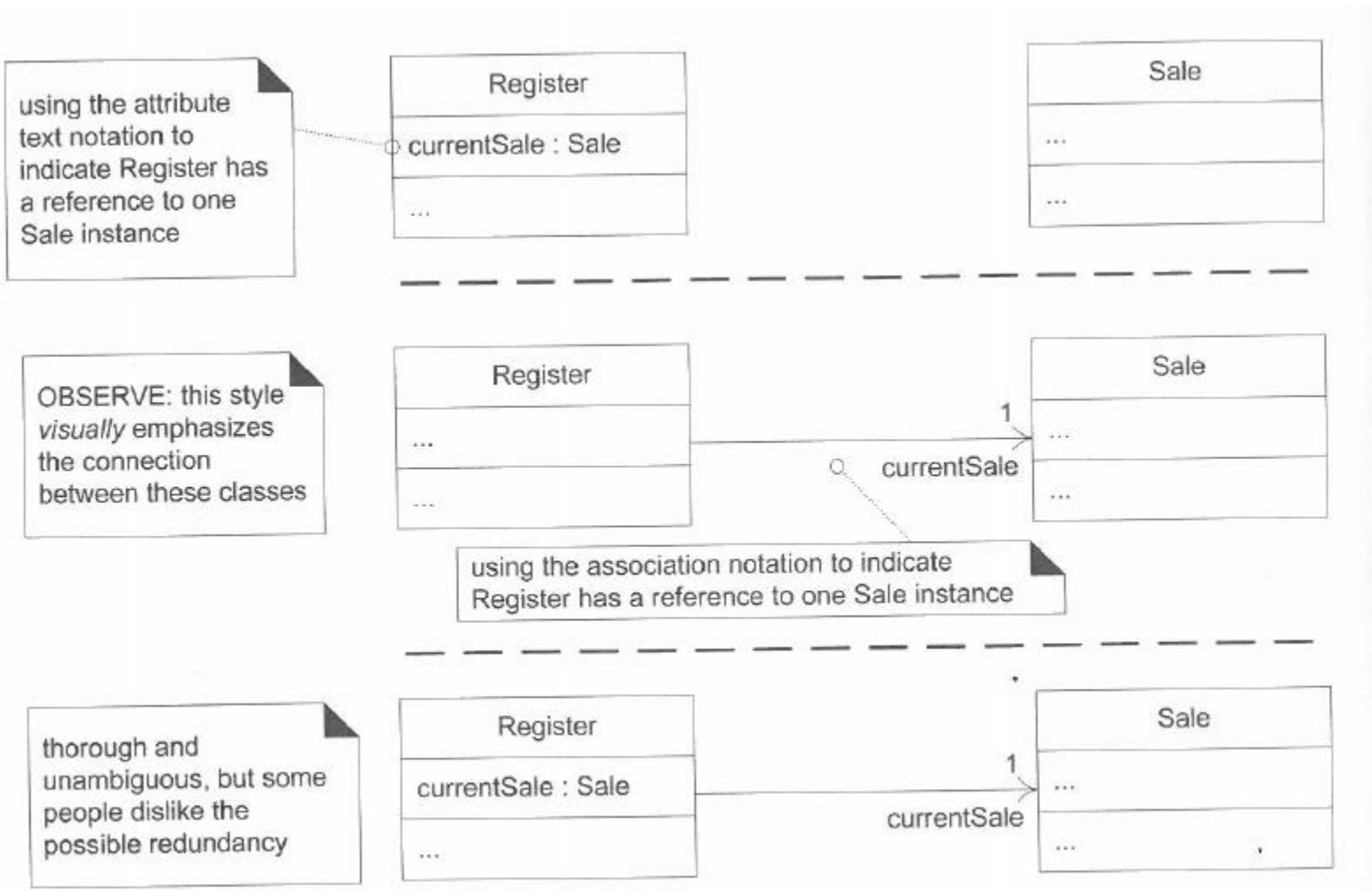


Figure 16.3 Attribute text versus association line notation for a UML attribute.

Synlighed

- **Synlighed** er et objekts evne til at ”se” eller have reference til et andet objekt
- For at et afsender objekt A kan sende en besked til et modtager objekt B, skal A have synlighed til B
- Former for synlighed
  - Attributsynlighed: B er en attribut i A
  - Parameter synlighed: B er en parameter i en metode i A
  - Lokal synlighed: B er et lokalt objekt i en metode i A
- **I klassediagrammet** tilføjes en **associering** mellem A og B og der tilføjes **navigeringsretning** (pil) som fortæller at A sender en besked til B, Larman s. 252-257 og 260. Den stiplede linie bruges ved parametersynlighed.

Impl. af lagdelt arkitektur

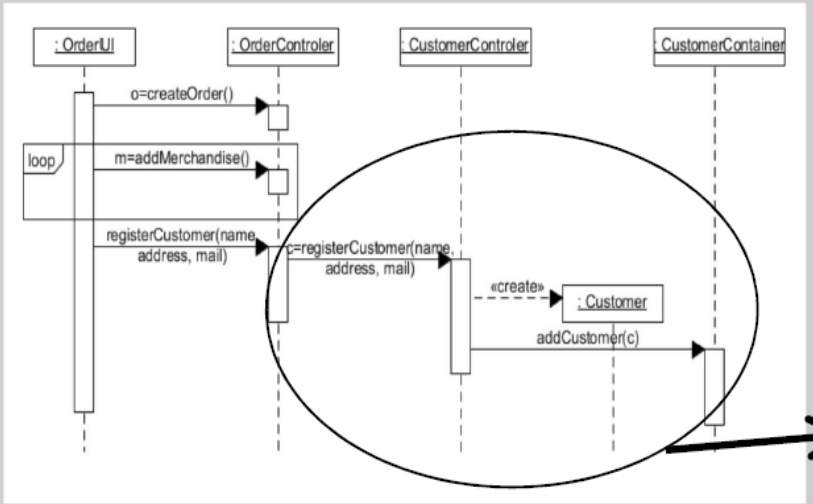
- Klasserne skal placeret i det lag de tilhører i den lagdelte arkitektur. De arkitektoniske lag realiseres ved brug af pakker:
  - Opret pakkerne: *edit-> new package*, husk at i Java standarden er pakkenavne med småt. Nogle toolchains stejler fuldstændigt ved store bogstaver i package navne!)
  - Tilføj klasser til pakken: *Dobbeltklik på pakken, vælg new*
- Synlighed til en underliggende pakke etableres ved *import*, fx:
  - package controllerlayer;
  - import modellayer.*;
  - import modellayer.Customer;

Tilføjes automatisk når klassen tilknyttes pakken

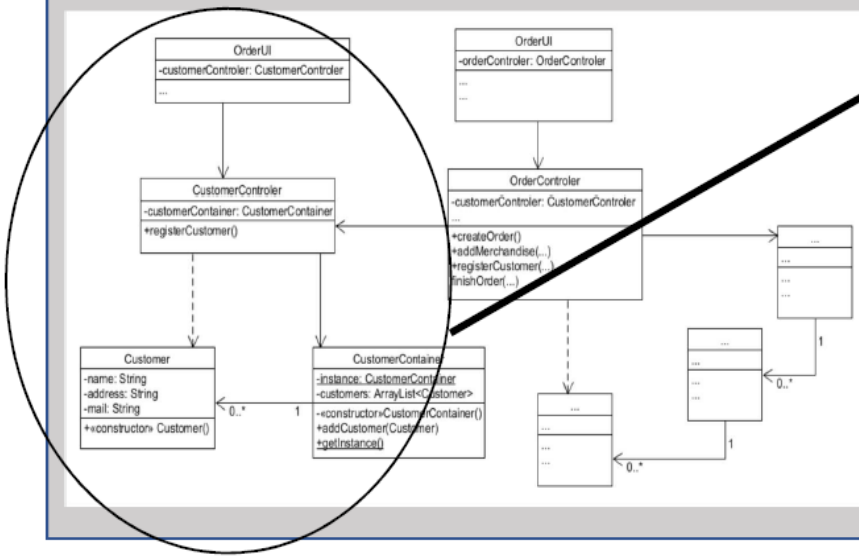
Synlighed til underliggende lag - skal tilføjes manuelt, her hele laget, vi kan også tilføje specifikke klasser

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

}
```

Opgave

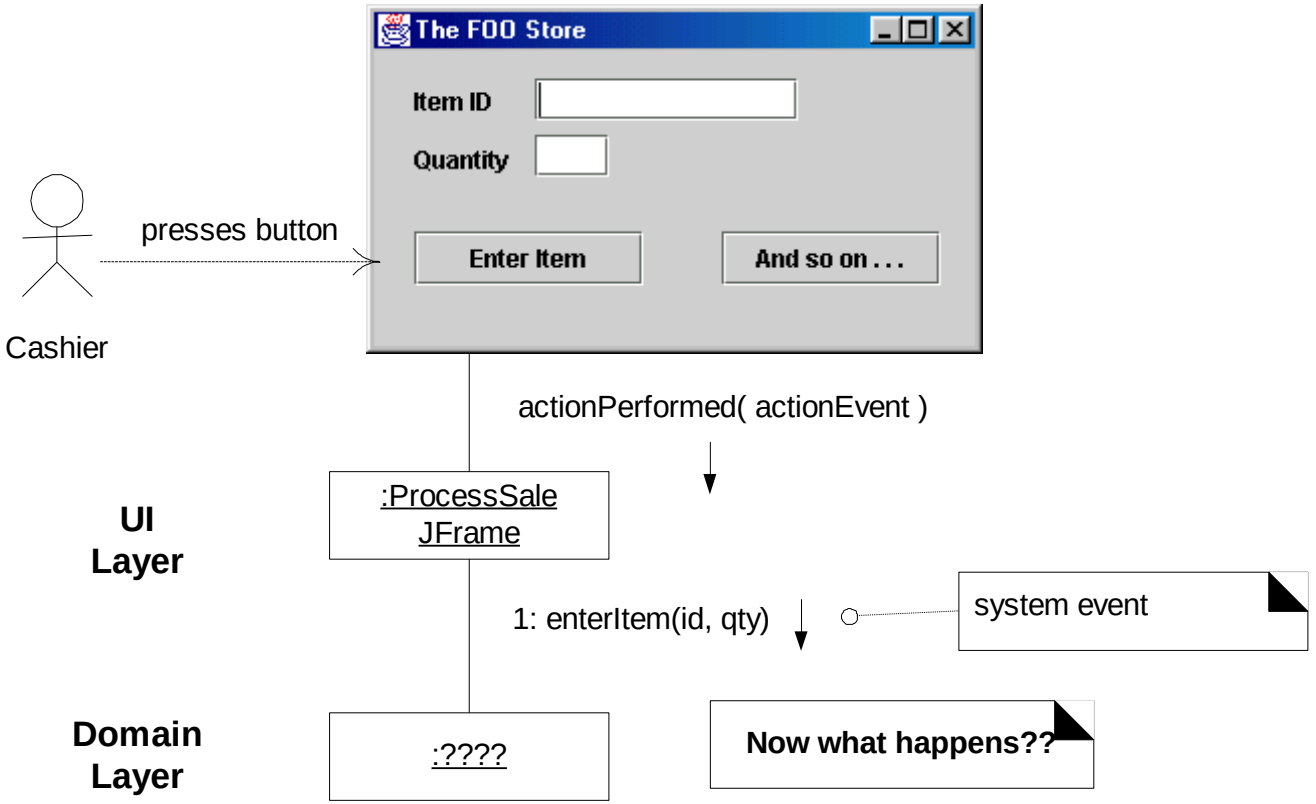
- Lav opgave 1 og 2 på dagens opgaveliste
  - Fokus skal være på at få orden i diagrammerne og have konsistent navngivning i både diagram og kode

Design mønstre

- GRASP (General Responsibility Assignment Software Patterns) beskriver fundamentale principper i at tilskrive ansvar til objekter, og er således en hjælp til at bestemme ansvar
- GRASP anvendes bl.a. som argument for den 3-delte arkitektur.

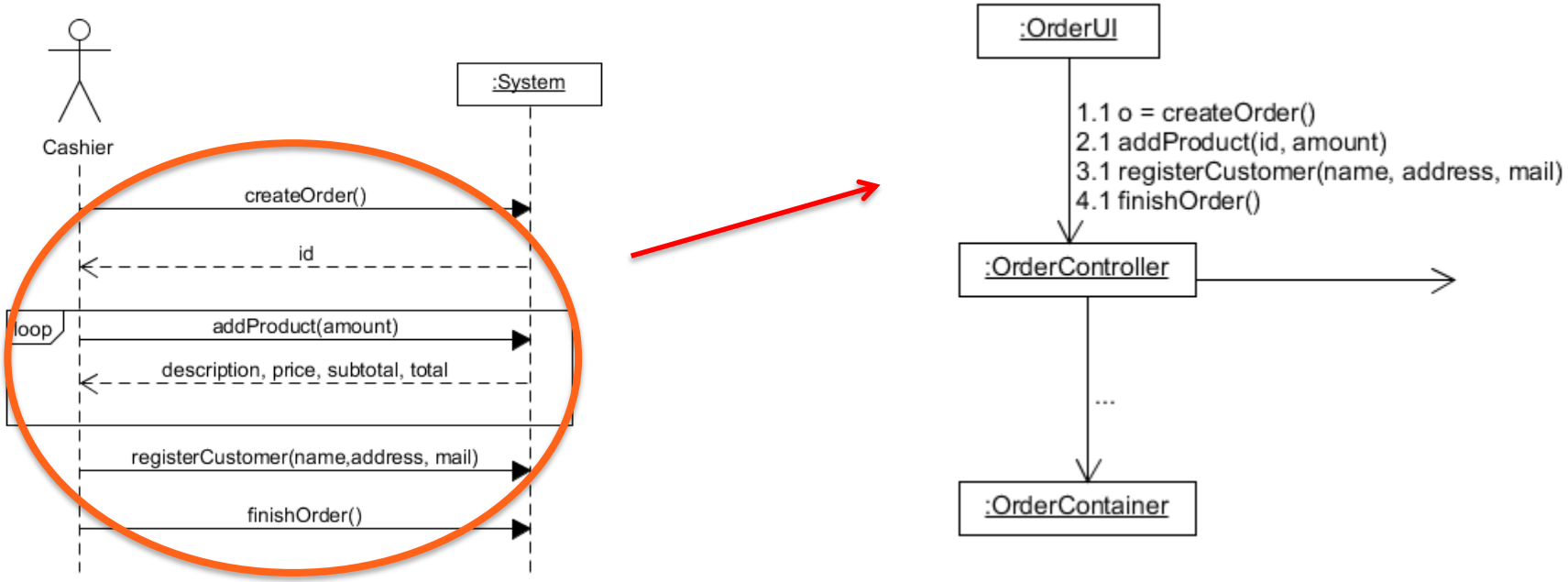
Eksempel: GRASP controller

- 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



Konsekvens af GRASP controller

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



Design af interaktion
Eksempel: Kundekartotek

Domæne model:

Operationskontrakt

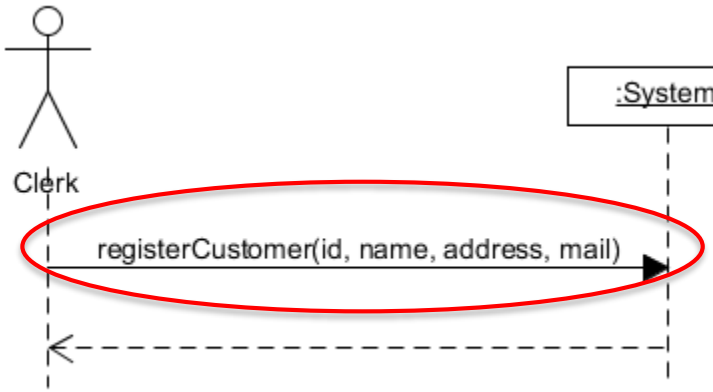
Præcondition: Ingen

Postcondition:

–En instans af Customer c blev oprettet

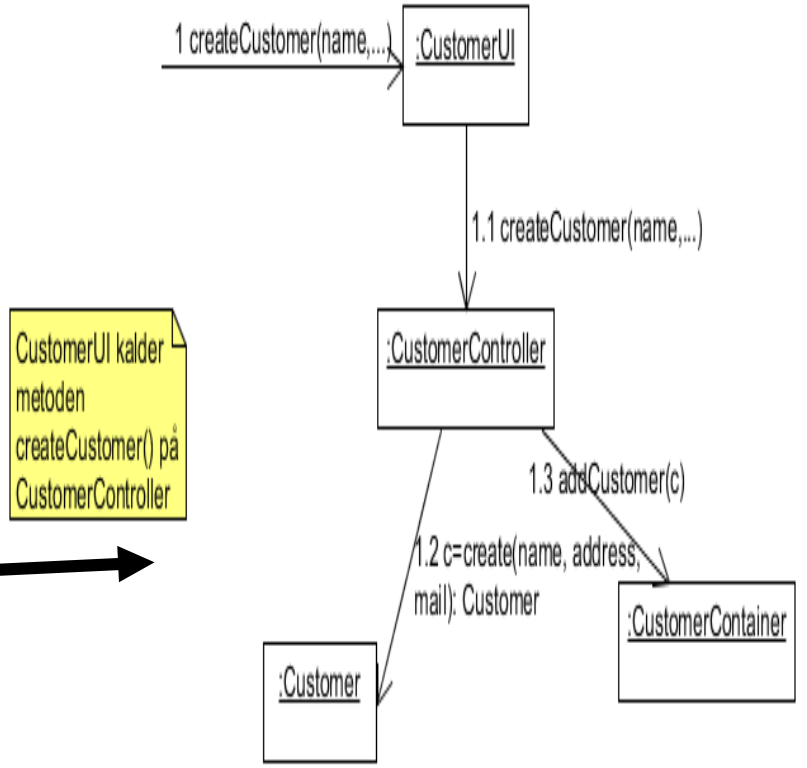
–c.customerNo, c.name, c.address, c.mail tilskrives værdier

SSD:

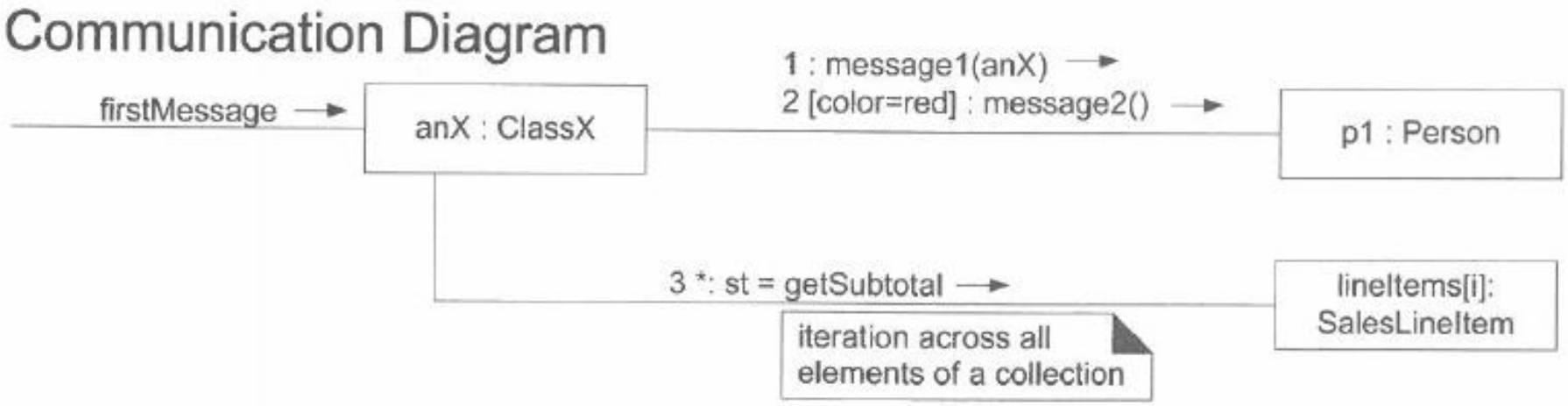


Design af interaktion: registerCustomer

UML kommunikationsdiagram



UML notation: Kommunikationsdiagram



UML notation for beskeder (metodekald)

- Beskeder/meddelelser/kommandoer, Larman s. 227
  - return = message(parameter: parameterType):returnType
  - eksempler:
    - p=getPerson(id)
    - p=getPerson(id:personId)
    - p=getperson(id:personId):Person
- create/destroy, larman s. 242
  - create(parameterliste) – fortolkes som *new*,

UML Create muligheder (kald af konstruktør) [Larman s. 242]

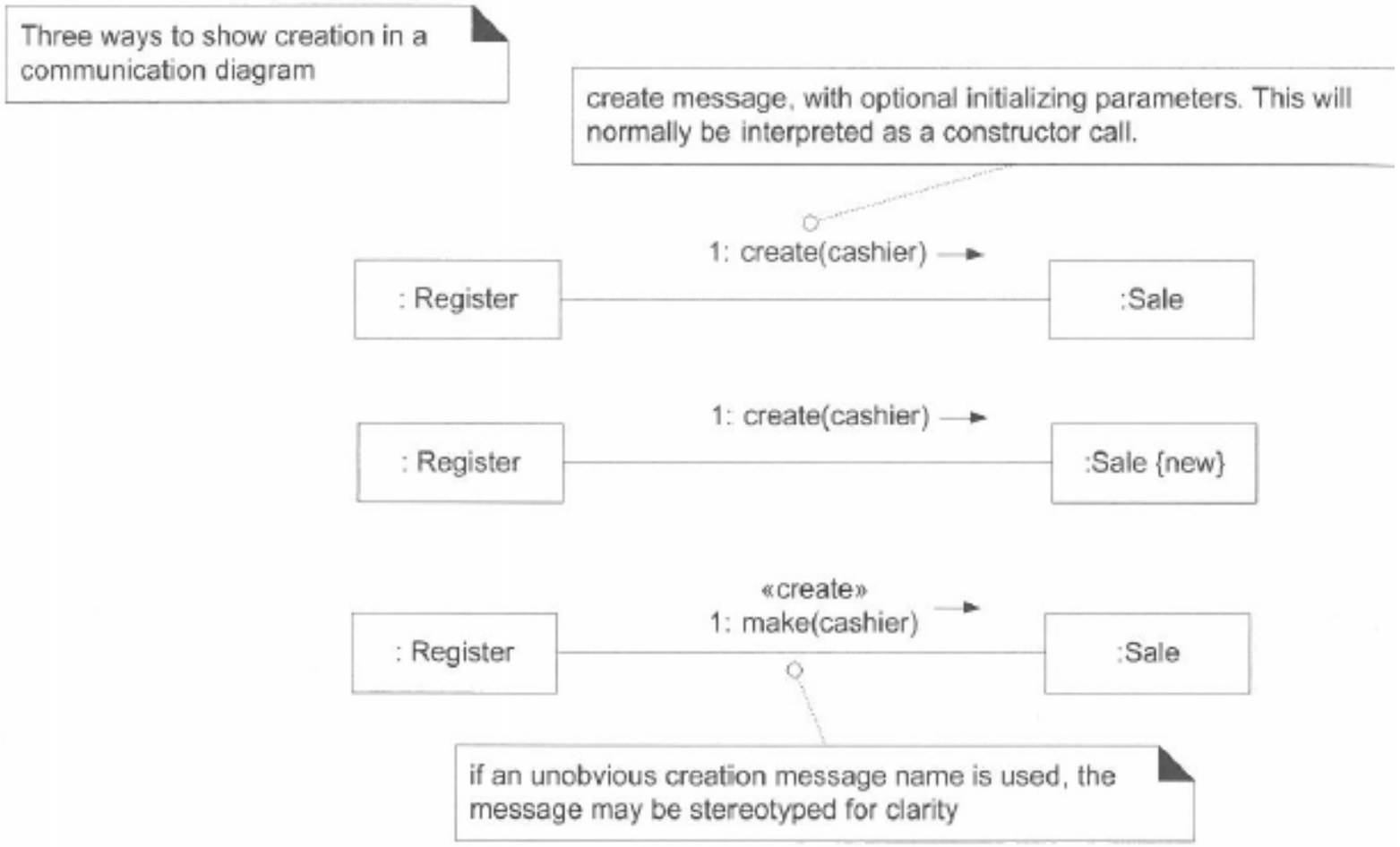
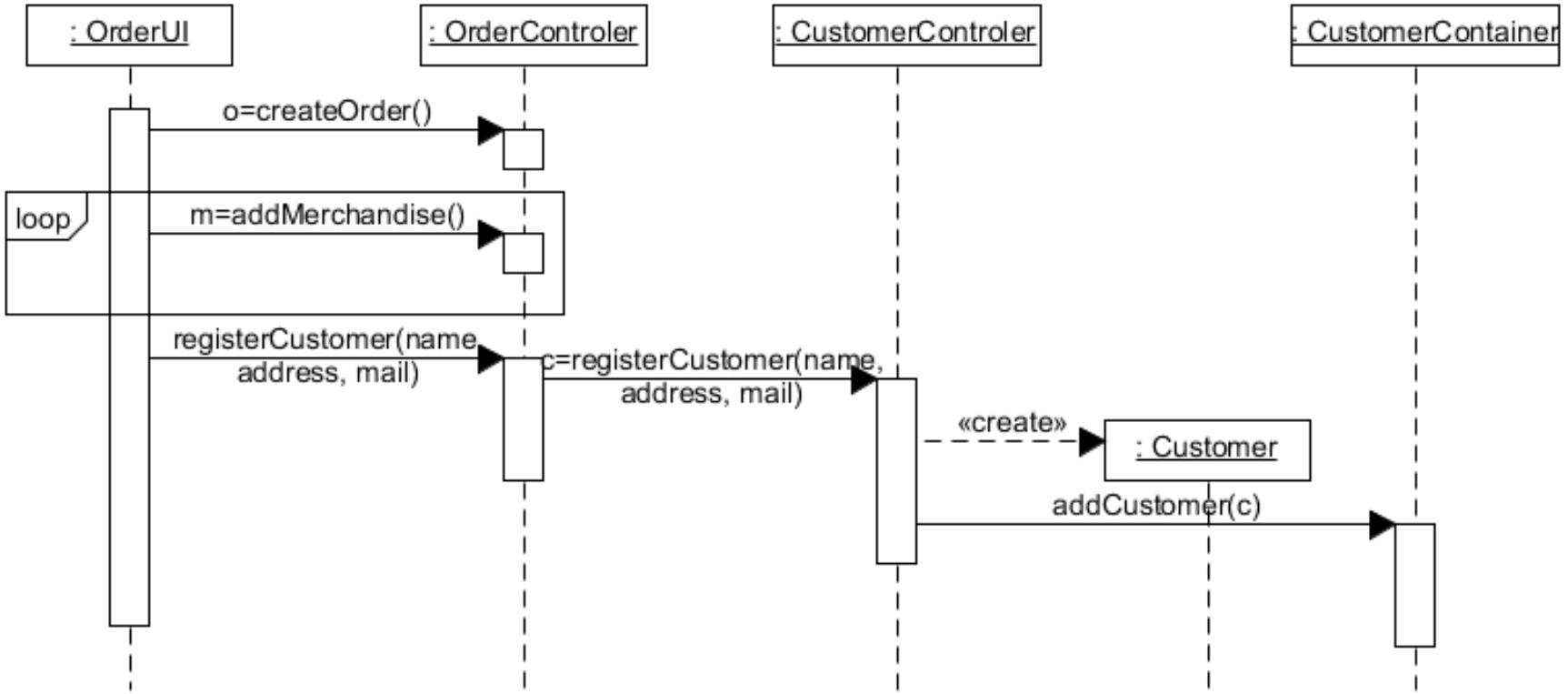


Figure 15.26 Instance creation.

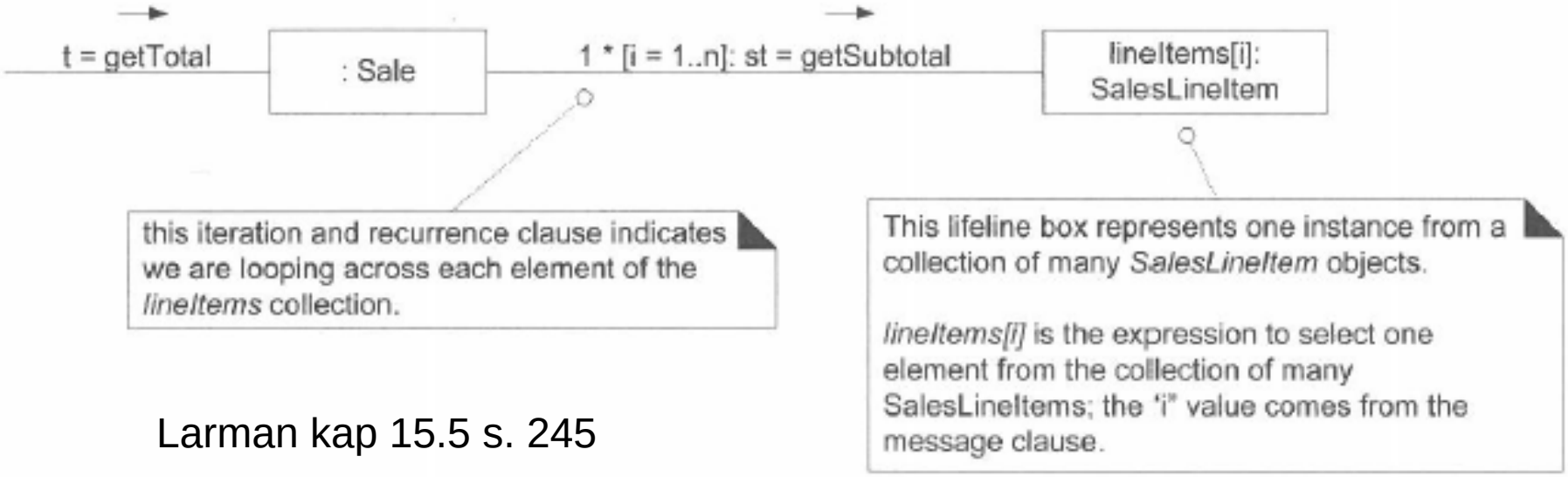
Alternativ interaktionsdiagram
UML sekvensdiagram



Design af søgninger – Iterationer gennem collections

- De fleste use cases indeholder søgninger
- I kundekartoteket kunne det være at finde en bestemt kunde ved angivelse af kundens navn dvs gennem en systemhændelse: *findKunde(navn)*
- Objekter er samlet i collections
- Når man skal finde et konkret objekt itereres gennem dets collection
- For hver objekt kaldes *getNavn()* der sammenlignes med parametren navn.
- Er der match er objektet fundet

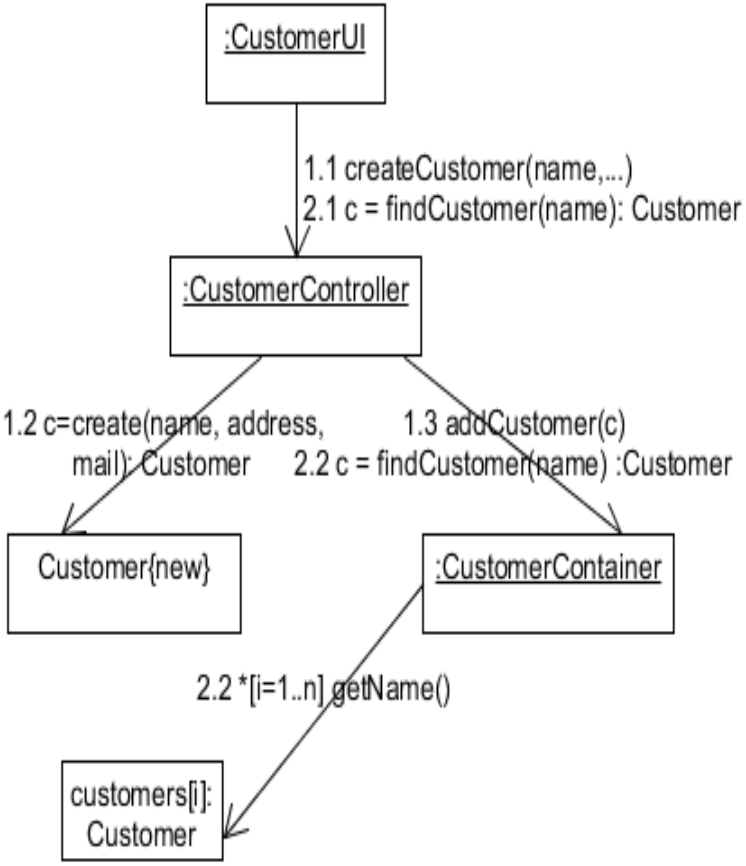
UML – Iterationer gennem ”Collections”

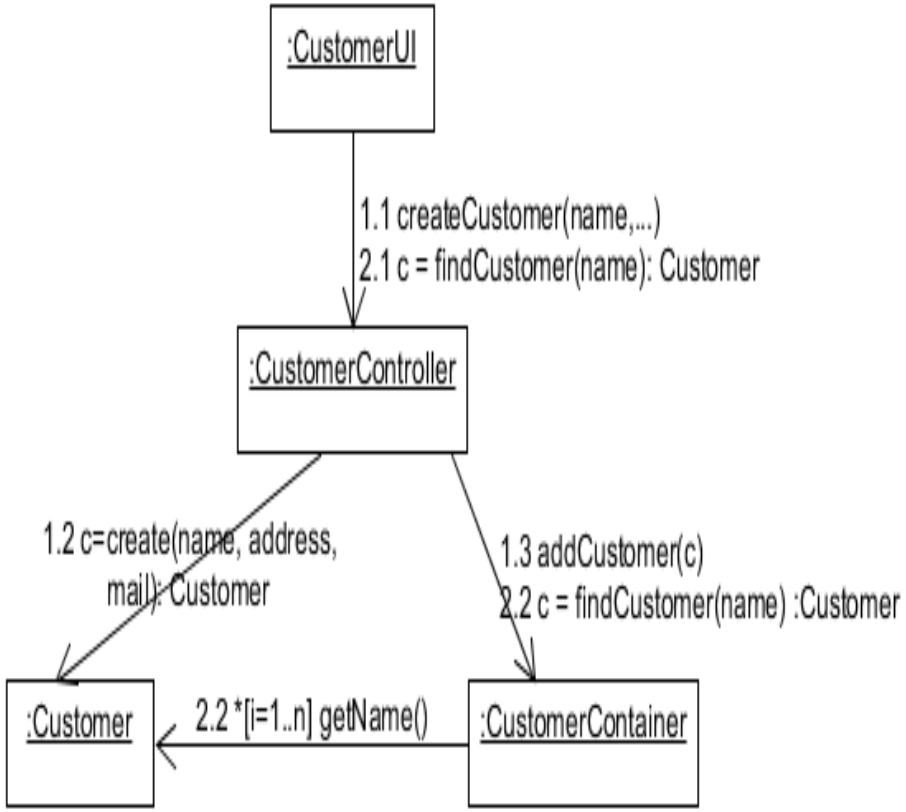


Larman kap 15.5 s. 245

Eksempler: Design af søgning
Systemhændelse: *findCustomer(id)*

Alternative visninger



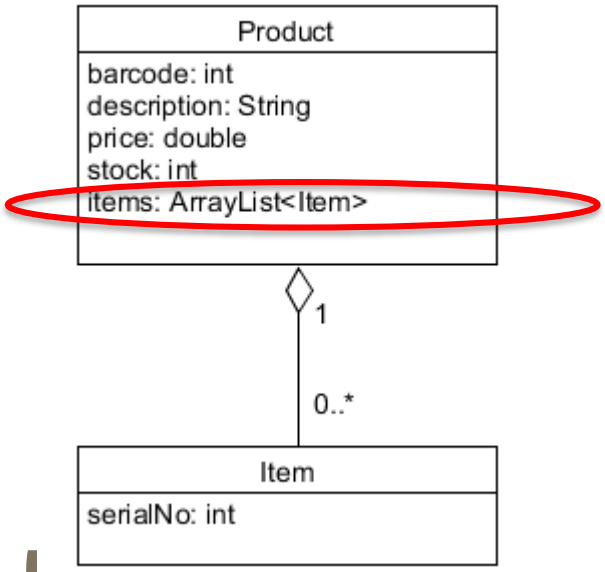


Design af aggregeringer (Helhed -> del)

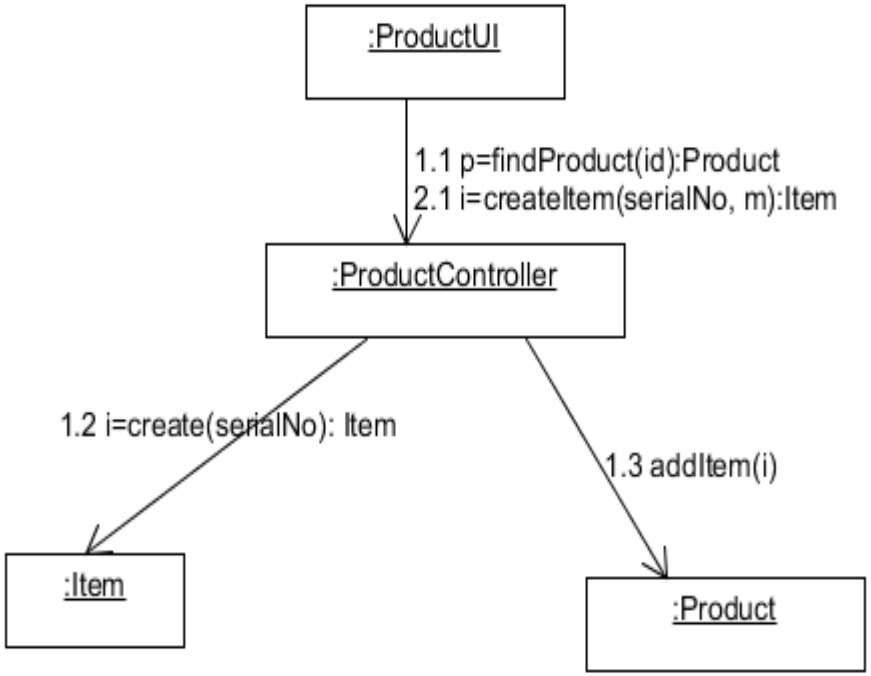
Registrering af ”delene”, fx *opretEksemplar*

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.

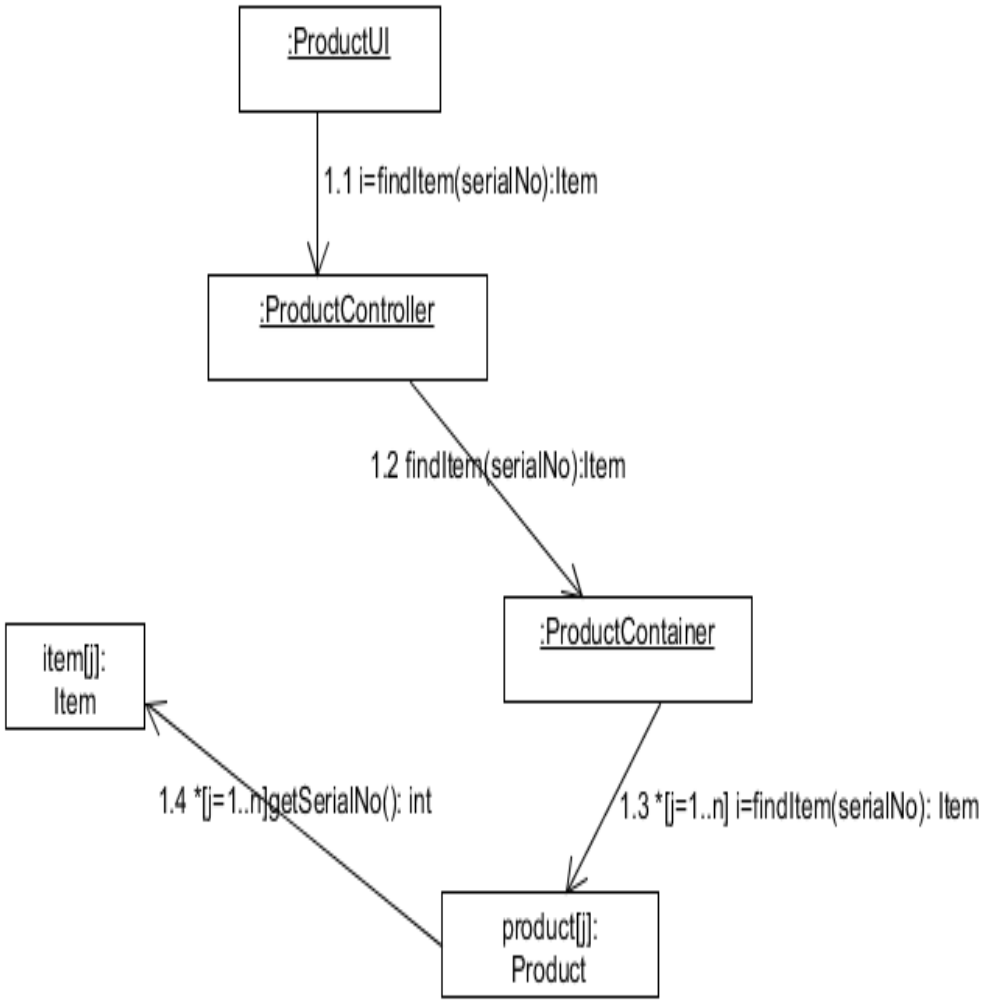


Design af interaktionen for createItem
UML komunikationsdiagram anvendt



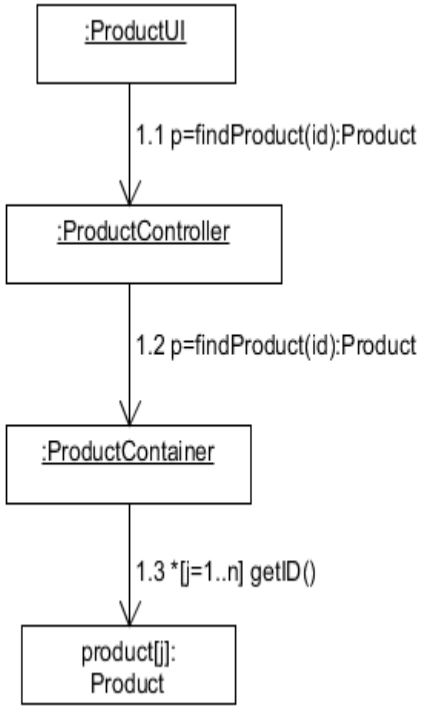
Design af aggregeringer
Alternative søgninger

Direkte søgning på serialNo



Design af operationen: findItem(serialNo)
Først itereres gennem containeren med Product
For hvert Product objekt skal der nu
itereres over dets Items.
Operationen stopper når der er match på serialNo
og item.getSerialNo() på Item på
product.findItem(serialNo)

Søgning på productId: I p der returneres til UI findes en arrayList<Item>. Brugeren vælger så fra listen.



Opgave 3-4

- I skal nu løse opgave 3 og 4.
- Husk at fokusere på diagrammeringen

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 (næste gang)

Design mønstre I skal kunne

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

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
  - *findMerchandise(id)* som også indgår i *Håndter vare – CRUD use* casen
- For at opfylde GRASP håndteres dette ved genbrug af Controller klasser

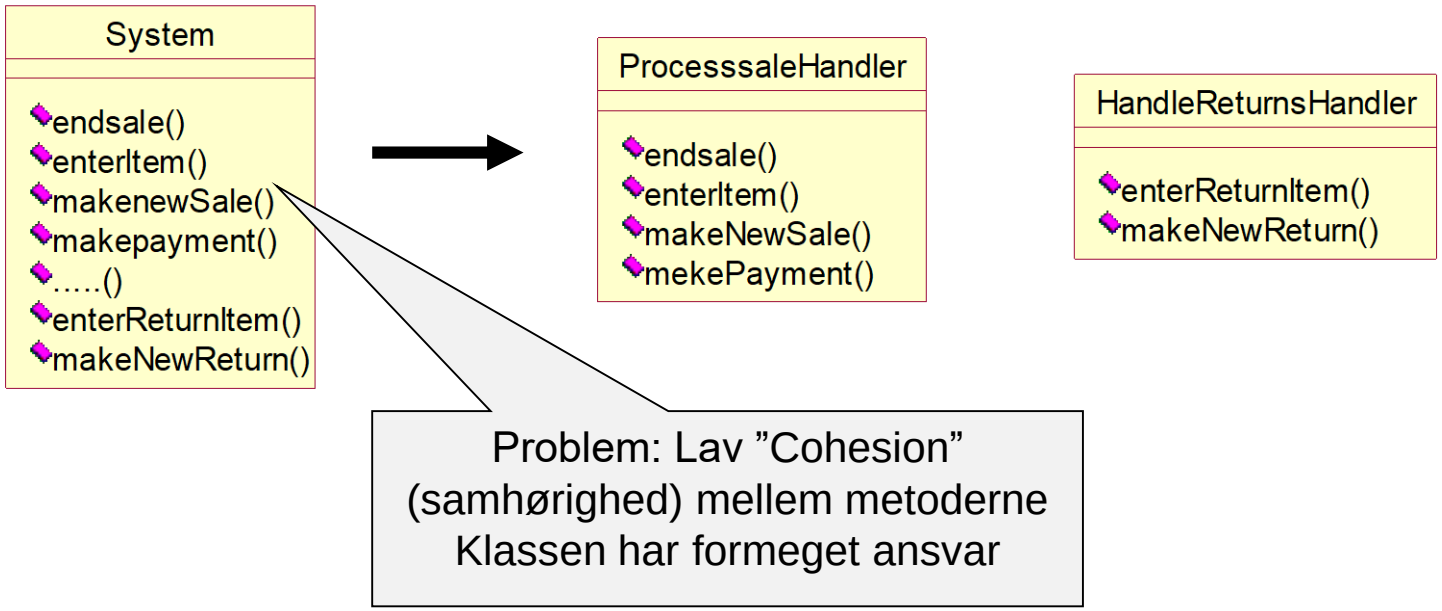
GRASP høj samhørighed mønstret

- **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 samhørinheden forbliver høj
- **Fordele:**
  - let forståelig
  - letter vedligehold
  - betyder ofte lav kobling
  - 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



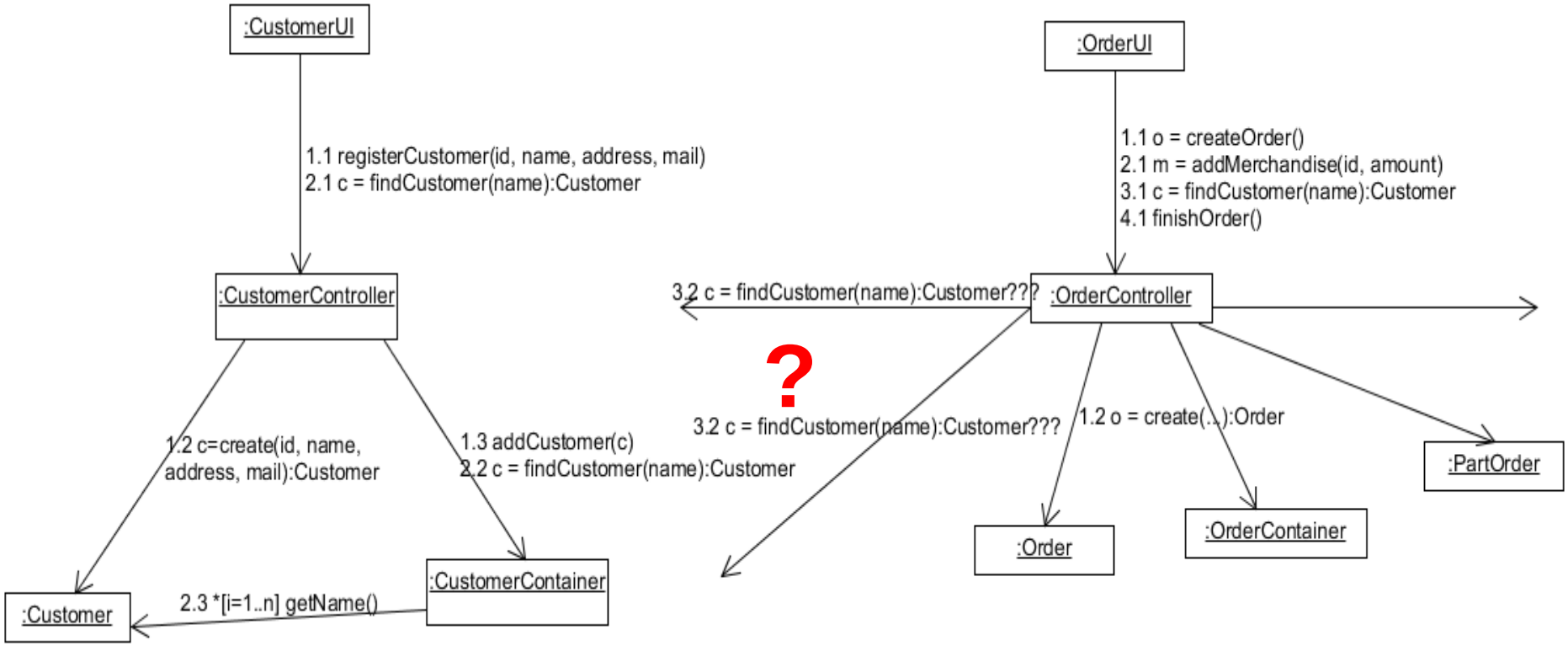
GRASP lav kobling mønstret

- **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/de

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*?

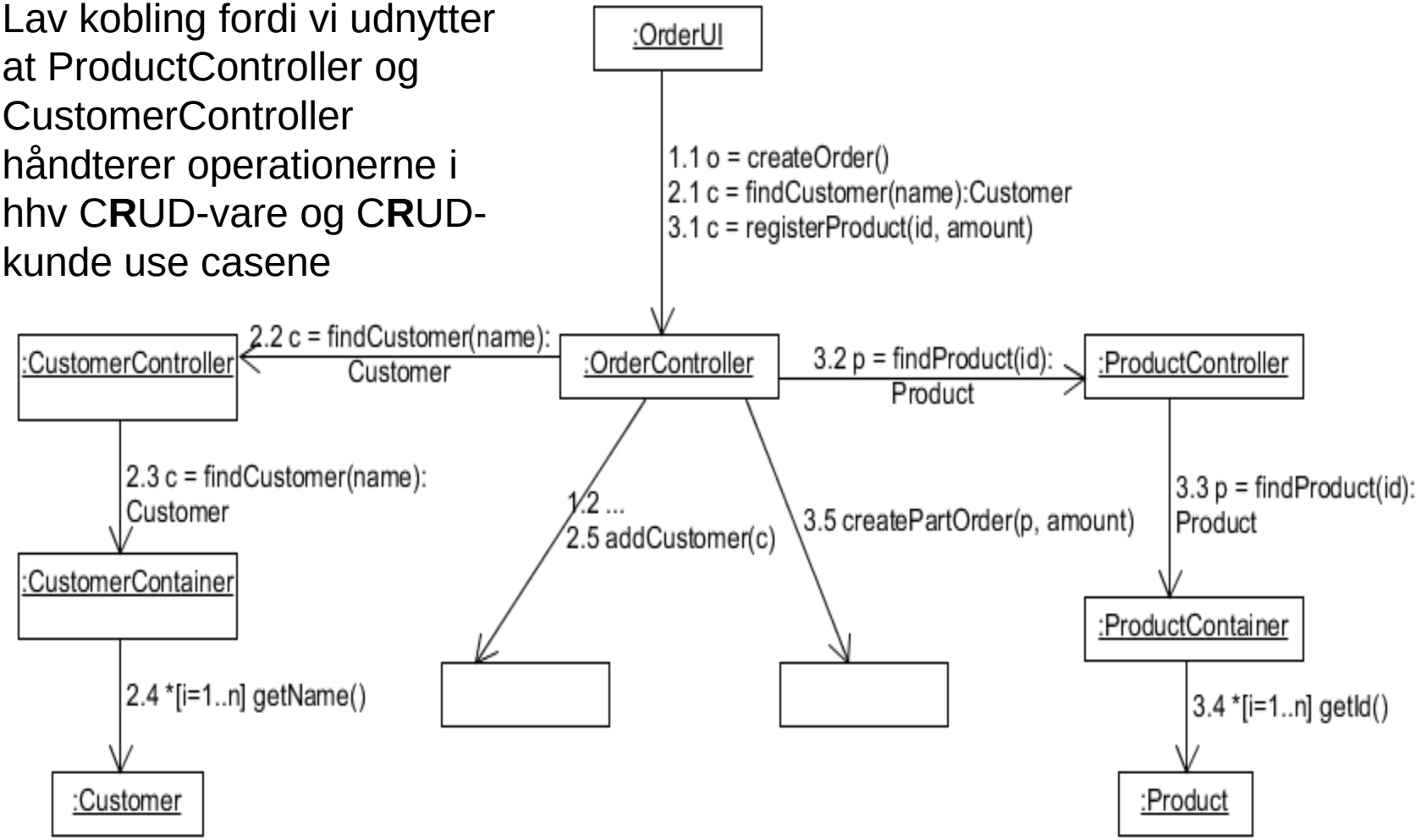


Genbrug af controllere

- 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 *MerchandiseController* 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 mangelfuld)

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



Opgave 5 - 9

- Hvis tiden tillader det. Vi kan vende tilbage til dette næste gang.