Software Engineering I (02161)

Week 2: Class diagrams part 1

Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2010

Activities in Software Developement

- Understand and document what kind of the software the customer wants
  - → Requirements Analysis
- Determine how the software is to be built
  - → Design
- Build the software
  - → Implementation
- Validate that the software solves the customers problem
  - → Testing
  - → Verification
  - → Evaluation: e.g. User friendlieness

From Requirements to Design

Problem

Given a requirements model consisting of:

1. use case diagram
2. detailed use case descriptions
3. glossary
4. non functional requirements

how do I get a system design consisting of

a. class diagram with associations, attributes, and operations?

From Requirements to Design: Solution

Design process

1. The terms in the glossary give first candidates for classes, attributes, and operations
2. Take one use cases
   a. Take one main or alternative sceanrio
      i. Realize that scenario by adding new classes, attributes, associations, and operations so that you design can execute that scenario
   b. Repeat step a with the other scenarios of the use case
3. Repeat step 2 with the other use cases

- Techniques that can be used
  - Grammatical analysis of the text of the scenario
    → nouns are candidate for classes and attributes; verbs are candidates for operations, and adjectives are candidates for attributes
  - CRC cards (= Class Responsibility Collaboration cards)

Introduction CRC Cards

- Class Responsibility Collaboration (CRC)
- CRC cards were developed by Ward Cunningham in the late 80's
- Can be used for different purposes
  - Analyse a problem domain
  - Discover object-oriented designs
    - Learn to think objects

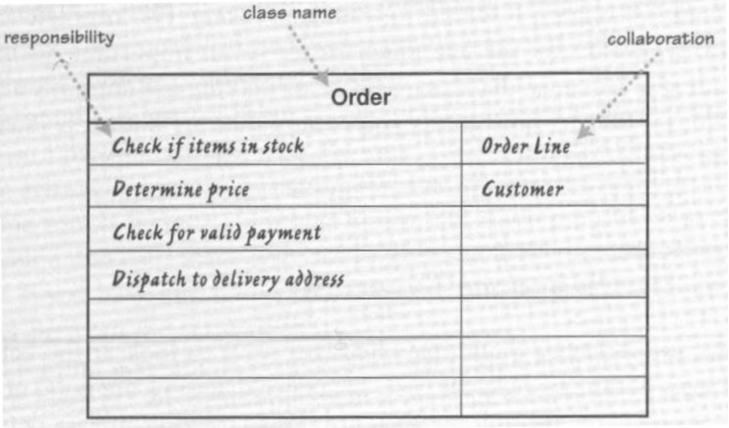
  → Objects
    - have structure and behaviour

    → both need to be considered at the same time
- Literature
  - http://c2.com/doc/oopsla89/paper.html
  - Martin Fowler: UML Destilled pages 62—63

CRC Card

- Class
  - Can be an object of a certain type
  - Can be the class of an object
  - Can be a component of a system
  - Index cards are used to represent classes (one for each class) (I use A6 paper instead of index cards)
- Responsibilities
  - Corresponds roughly to operations and attributes
  - Somewhat larger in scope than operations
  - ”do something”
  - ”know something”
- Collaborations
  - With whom needs this class to collaborate to realize its responsibilities

CRC Card Template



A larger example

- http://c2.com/doc/crc/draw.html

Process I

- Starting point
  - List of use-cases scenarios
    - Should be as concrete as possible
  - A group of up to 6 people (but can also be done alone)
- Brainstorming
  - Initial set of Classes (just enough to get started)
  - Assign Classes to persons
- Execute Scenarios
  - Simulate how the computer would execute the scenario
  - Each object/class is represented by one person
  - This person is responsible for executing a given responsibility
    - This is done by calling the responsibilities of other objects/persons he collaborates with
  - objects/classes can be created
  - responsibilitites can be added
  - collaborations can be added

Library Example: Problem Description and Glossary

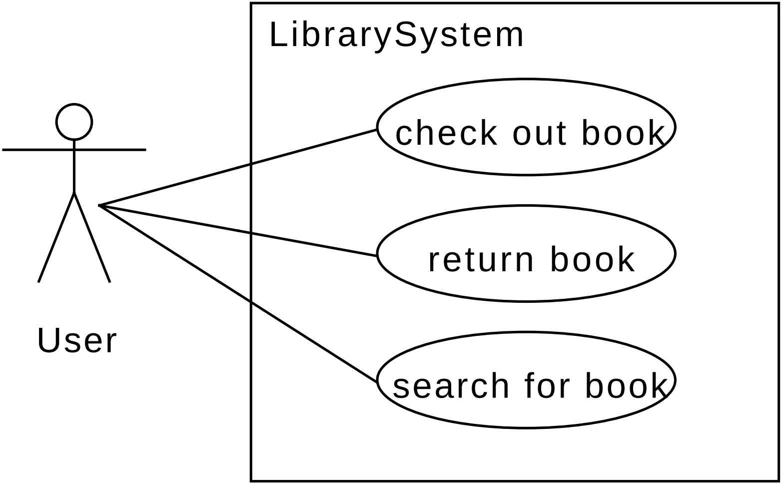
- Problem Description
  - Library system for checking out, returning, and searching for books. No more than 5 books can be loaned by one borrower at a time. And if a book is returned after its overdue date, a fine has to be paid.

Glossary

- Librarien
  - The object in the system that fulfills User requests to check out, check in, and search for library materials
- Book
  - The set of objects that represent Users who borrow items from the library
- Borrower
  - The set of objects that represent Users who borrow items from the library
- . . .

Library Example: Use Case Diagram

- Use Cases



Library Example: Detailed Use Case Check Out Book

- Name: Check Out Book
- Description: The user checks out a book from the library
- Actor: User
- Main scenario:
  1. A user presents a book for check-out at the check-out counter
  2. The system registers the loan
- Alternative scenarios:
  - The user already has 5 books borrowed
    - 2a The system denies the loan
  - The user has one overdue book
    - 2b The system denies the loan

Example II

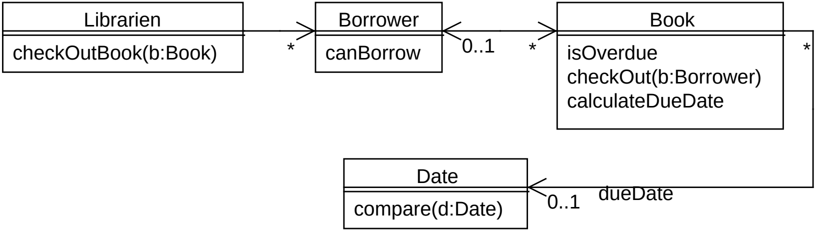
- Set of initial CRC cards
  - Librarien
    - The object in the system that fulfills User requests to check out, check in, and search for library materials
  - Borrower
    - The set of objects that represent Users who borrow items from the library
  - Book
    - The set of objects that represent items to be borrowed from the library
- Use case Check out book main scenario
  - ”What happens when Barbara Stewart, who has no accrued fines and one outstanding book, not overdue, checks out a book entitled Effective C++ Strategies+?”

Library Example: All CRC cards

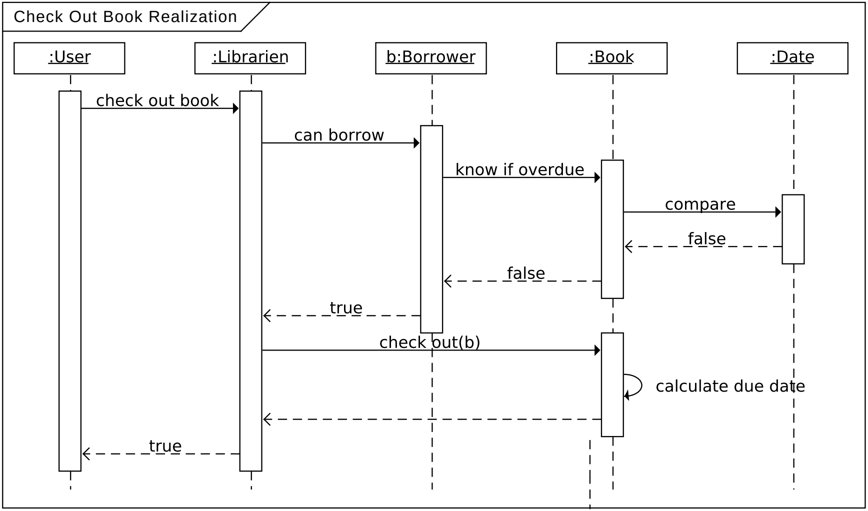
Process: Next Steps

- Review the result
  - Group cards
    - by collaborations
    - shows relationship between classes
  - Check responsibilities
  - Check correct representation of the domain
  - Refactor if needed
- Transfer the result
  - UML class diagrams
    - Responsibilities map to operations and attributes/associations
    - Collaborations map to associations and dependencies
  - The executed scenarios to UML interaction diagrams

Example: Class Diagram (so far)



Example: Sequence Diagram for Check-out book



Summary

- Further scenarios give more detail
- The scenarios are now quite easy to implement
- CRC process can be repeated on a more detailed level, e.g., to design the database interaction, or user interface
- Helps one to think in objects (structure and behaviour)
- Humans playing objects help to get a better object-oriented design as it helps delegating responsibilities

Computing the price of an order

- Task
  - Calculate the price of an order
  - Take into account if the customer has any discounts
- Initial CRC cards

Order	
calculate price	Order Line
knows order lines	Customer
knwos custommer	

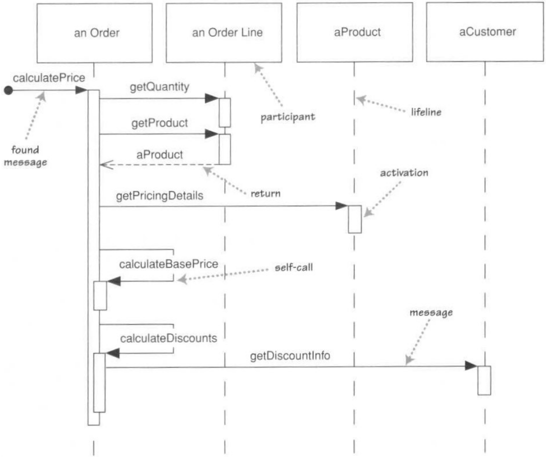
Customer	
knows name	
knows discount info	

OrderLine	
knows quantity	Prodcut
knows product	

Product	
knows price	
knows name	

Two possible solutions: Centralised Control

- The order computes the price by asking its collaborators about data



→ centralised control

Centralised Control: CRC cards

Order	
calcuate price calcuate base price calcuate discounts knows order line knows customer	Order Line Product Customer

Customer	
knows name knows discount info	

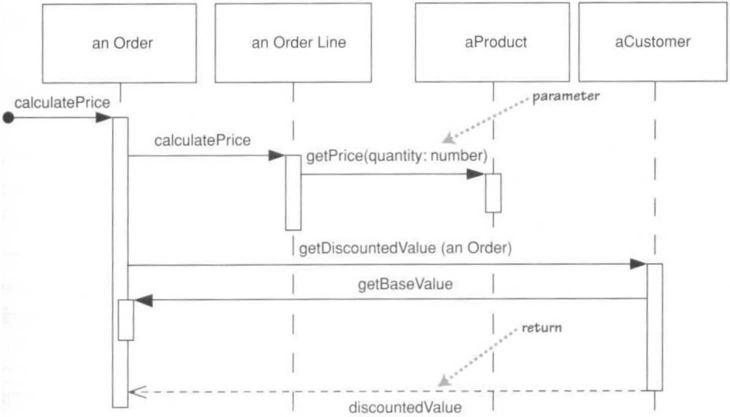
OrderLine	
knows quantity knows product	Prodcut

Product	
knows price knows name	

- Only class Order has any interesting behaviour
- OrderLine, Customer, and Product are purly data classes

Two possible solutions: Distributed Control

- The order computes the price by delegating part of the price calculation to order line and customer



→ distributed control

Distributed Control: CRC cards

Order	
calcuate price knows base price calcuate discounts knows order line knows customer	Order Line Customer

Customer	
knows name knows discount info calculate discount for order	Order

OrderLine	
knows quantity knows product calculate price	Prodcut

Product	
knows price knows name get price for quantity	

- More customer types can be added
- Each computing the discount differently

- The product know calculates the price depending on quantity
- One could now have products that are cheaper the more one buys

Centralised vs Distributed Control

- Centralised control
  - One method does all the work
  - The remaining objects are merely data objects and usually don't have their own behaviour
  - Typical for a procedural programming style
- Distributed control
  - Objects collaborate to achieve one task
    - "Instead of doing myself the work, I delegate work to other object"
  - Each object in a collaboration has behaviour (= is a "real" object)
  - Typical object-oriented style
    - Each object has its own responsibilities
    - Facilitates polymorphism

Object-Orientation

Distributed Control is a characteristic of object orientation

Class Diagram I

Class diagrams can be used for different purposes

1. to give an overview over the domain concepts
   - as part of the requirements analysis (e.g. a graphical form representation supplementing the glossary)
2. to give an overview over the system
   - as part of the design of the system
3. to give an overview over the systems implementation
4. ...

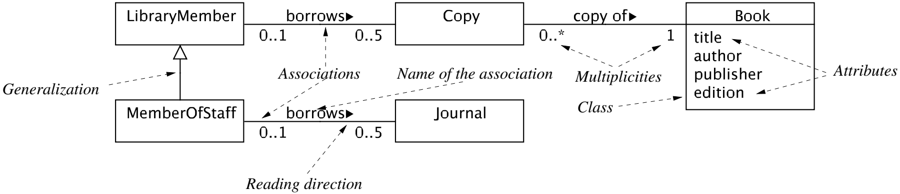
Level of detail of a class description depends on the purpose of the class diagram

Domain Modelling : typically low level of detail

⋮

Implementation : typically high level of detail

Class Diagram Example

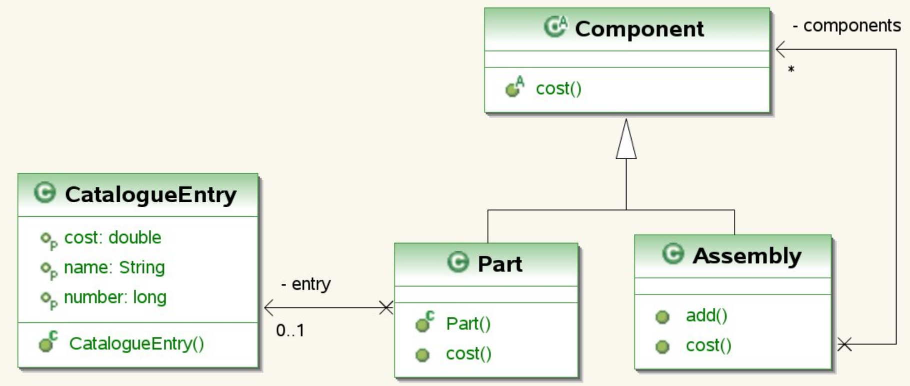


Basic concepts of class diagrams

- Classes with attributes and operations
- Associations with multiplicities and possibly navigability
- Generalization of classes (corresponds in principle to subclassing in Java)

Why class diagrams?

- Example of a class diagram



Why class diagrams? (cont.)

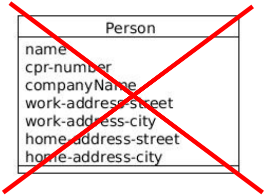
- The same information as in the class diagram presented as Java source code
- But with the class diagram the relationship between the classes is easier to understand

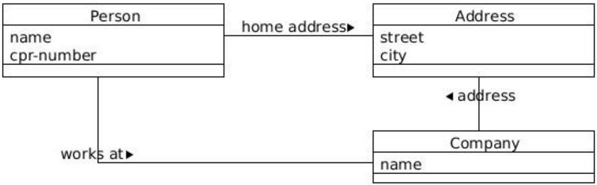
```
public class Assembly
           extends Component {
 public double cost() {   }
 public void add(Component c) {}
 private Collection<Component>
      components;
}

public class CatalogueEntry {
 private String name = "";
 public String getName() {}
 private long number;
 public long getNumber() {}
 private double cost;
 public double getCost() {}
}
```

```
public abstract class Component {
 public abstract double cost();
}

public class Part extends Component {
 private CatalogueEntry entry;
 public CatalogueEntry getEntry() {}
 public double cost(){}
 public Part(CatalogueEntry entry){}
```

Classes

- A class describes a collection of objects that have a common characteristics regarding
  - state (attributes)
  - behaviour (operations)
  - relations to other classes (associations and generalisations)
- A class ideally should represent only one concept
  - All the attributes should related to that concept
  - All the operations should make sense for that concept





Class Description

'-' : private

'+' : public

'#': protected

```
KlasseNavn
-----------------------------------
+navn1: String = "abc"
-navn2: int
#navn3: boolean
-----------------------------------
-f1(a1:int,a2:String []): float
+f2(x1:String,x2:boolean): void
#f3(a:double): String
```

Klassens navn

Attributter

Operationer

'navn3' og 'f1' er statiske størrelser

private : only visible in the same class
protected : visible also in subclasses
public : visible also in other classes
package : (∼) visible for other classes in the same package

- Attributes and operations that are underlined are static
  - Attributes can be accessed without that an instance of that class exists
  - Operations can be called without that an instance of that class exists

Associations between classes

- A association between classes means, that the objects belonging to the two classes have knowledge of each other
- Associations can be navigable in one direction or two directions (a.k.a. bidirectional association)

→ this means that knowledge can be only on one side or on two sides

Navigable associations in one direction I

Example: Persons and their employers:

- every person is associated to an employer (company/firma)
- every company has 0, 1, or more ('*') employees (persons)

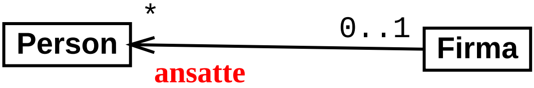
The arrow means that a company has knowledge about all employees (persons)

→ a company object has a reference to all the objects representing employees

- Conversely, a person object does not need to have a reference to the company object
- A × at an arrow denotes that it should not be possible to navigate in this direction
- *Navigability* in the direction of the arrow
- 0 and $*$ are called *multiplicities* or *cardinality*

Navigable associations in one direction II

Example: Persons and their employers:

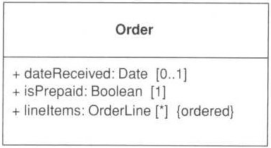


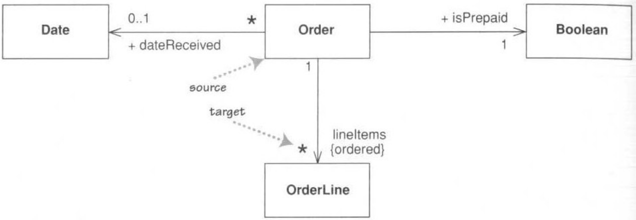
- a *role* (here **ansatte**) describes objects (here persons) at the end of an association, seen from the objects belonging to the classes at the opposite end of the (here company)
- default role name: name of the associated class (e.g. person)
- in an implementation a role name is typically a variable. For example:

```
public class Firma
{    ....
     private Collection<Person> ansatte;
     ....
}
```

Attributes and Associations

- There is in principle no distinction between attributes and associations
- Associations can be drawn as attributes and vice versa





Attributes versus Associations

When to use attributes and when to use associations?

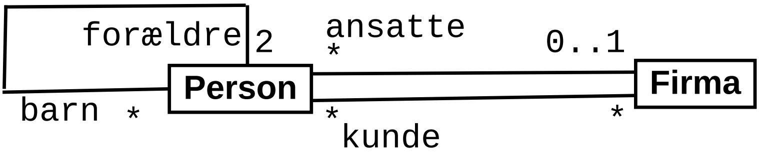
- Associations
  - When the target class of an association is shown in the diagram
  - The target class of an association is a major class of the model
    - e.g. Part, Assembly, Component, . . .
- Attributes
  - When the target class of an associations is not shown in the diagram
  - With datatypes / Value objects
    - Datatypes consists of a set of values and set of operations on the values
    - In contrast to classes are datatypes stateless
    - e.g. int, boolean, String . . .
  - Library classes
- However final choice depends on what one wants to express with the diagram
  - E.g. Is it important to show a relationship to another class?

Multiplicities

- At the end of associations (or at other places) one often denotes how often an object can appear. E.g. how many employees a company can have
- These denotations are called multiplicities or cardinalities.
- Typical values used as cardinalities:

values	**meaning**
`0`	0
`1`	1
m..*n*	integer interval *m* to *n* (*m* and *n* inclusive)
$\star$	$0, 1, 2, \ldots$
m..$\star$	$m, m+1, m+2, m+3, \ldots$

Bi-directional associations



when associations don't have any arrows, can this be understood

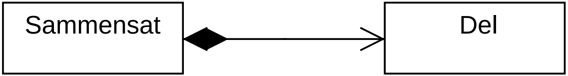
- as bi-directional, i..e. navigable in both directions, where on has decided not to show navigability, e.g.
  - every person object has a reference to his employer
  - every company object has a reference to his employee
- or as an under specification of navigability

The example shows also

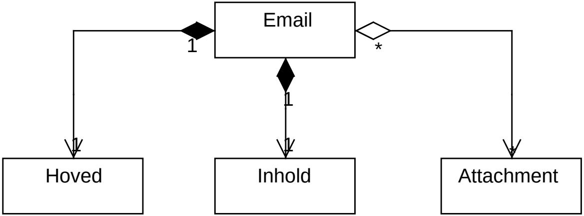
- two different associations between person and companies
- and a self-association

Composite Aggregation (I)

A special relation between "part-of" between objects



Example: An email consists of a header, a content and a collection of attachments



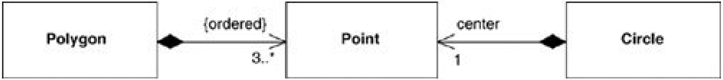
Composite Aggregation (II)

- The basic two properties of a composite aggregation are:
  - A part can only be part of one object
  - The of the part object is tied to the life of the containing object

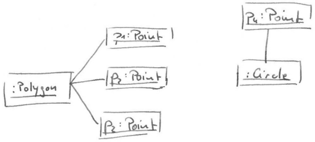
$\longrightarrow$ This results in requirements to the implementation

Composite Aggregation (III)

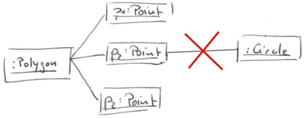
- A part can only be part of one object



- Allowed

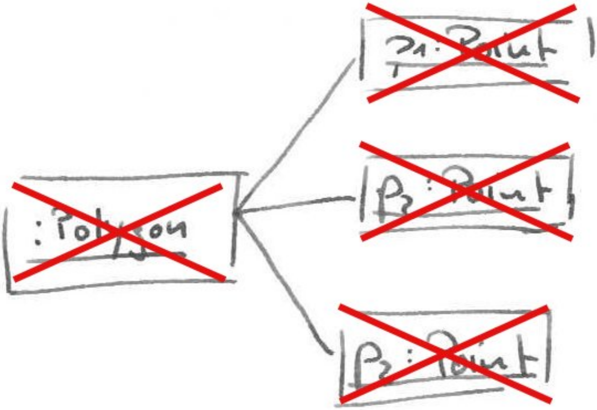


- Not Allowed



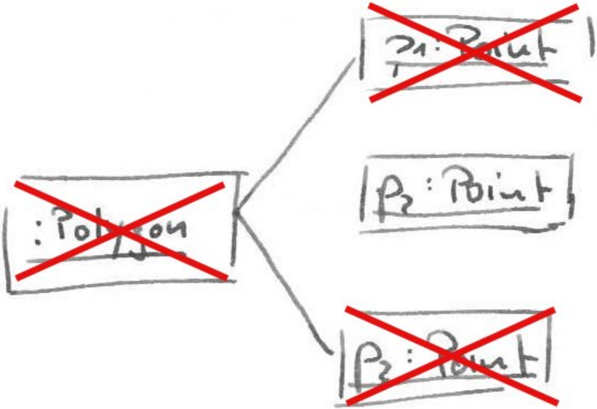
Composite Aggregation (IV)

- The life of the part object is tied to the life of the containing object
- If the containing object dies, so does the part object



Composite Aggregation (V)

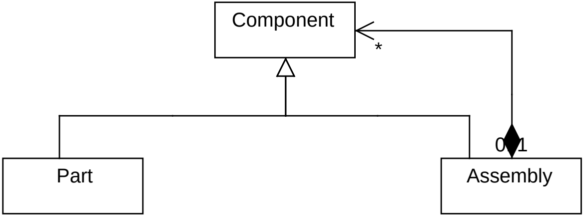
- But: A part can be removed before the composite object is destroyed



Composite Aggregation (VI): Styklister

- A complex component (assembly) cannot have itself as a component

→ Styklister can therefore be modelled using aggregation



Shared Aggregation

- Shared Aggregation
  - General ”part of” relationship
  - Notation: empty diamond

  - ”Precise semantics of shared aggregation varies by application area and modeller.” (from the UML 2.0 standard)

Summary

- From Requirements to Design
  - Class Responsibility Collaboration (CRC) cards
  - Object-orientation and distributed control
- Introduction to Class Diagrams
  - Classes
  - Associations
    - uni- / bi-directional
    - aggregation and composition
- Next week
  - Implementing class diagrams
  - More class diagram concepts