Introduction to Software Engineering

The Specification Phase

2.3 Object Modeling How to build a good model?

Prof. Walter F. Tichy

ქუთაისის
საერთაშორისო
უნივერსიტეტი

Wait—I´m lost!
What are all those diagrams for?

- Planning phase: functional Model
  - What are the scenarios, use cases??
  - recorded in requirements document
- How do you do that?
  - ask customer, (future) user
  - study work flows, processes, forms, existing (software or manual) system
  - study application area
  - brain storm with others
  - use your own world knowledge, imagination

But we are in the specification phase!

- Now we add the object model
  - What are the classes?
  - Which associations exist? And their multiplicities?
  - Any special associations? (aggregation, composition, qualified associations, 3-way associations?)
  - Which attributes are needed?
  - What are the methods?
  - Lastly: design inheritance
    - pull common elements of several classes up into super class
    - observe substitution principle
- Object model is a **static model.** No dynamic behavior.

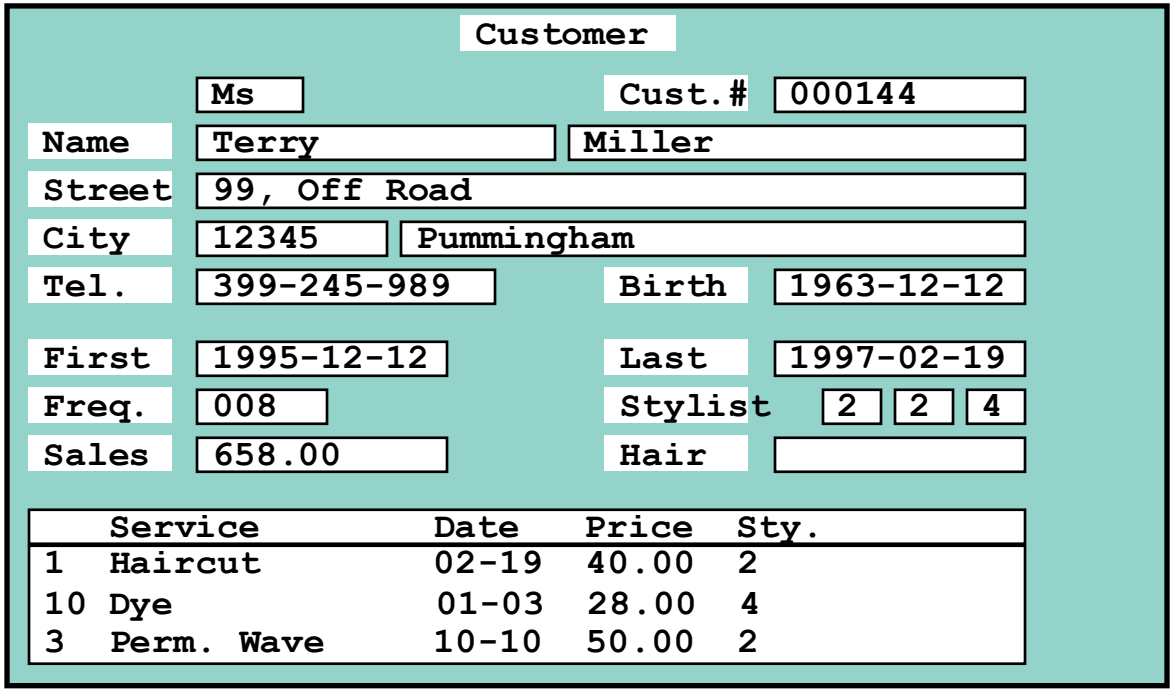
Why do we need the other diagrams?

- **Dynamic models**
  - Activity diagram
    - describes sequential and parallel processes
  - Sequence diagram
    - Identifies messages being sent among objects
    - (add newly identified methods to class diagram)
    - Shows calling relationship among several objects
    - Refines use cases, scenarios
  - State chart
    - defines state changes within a single object.

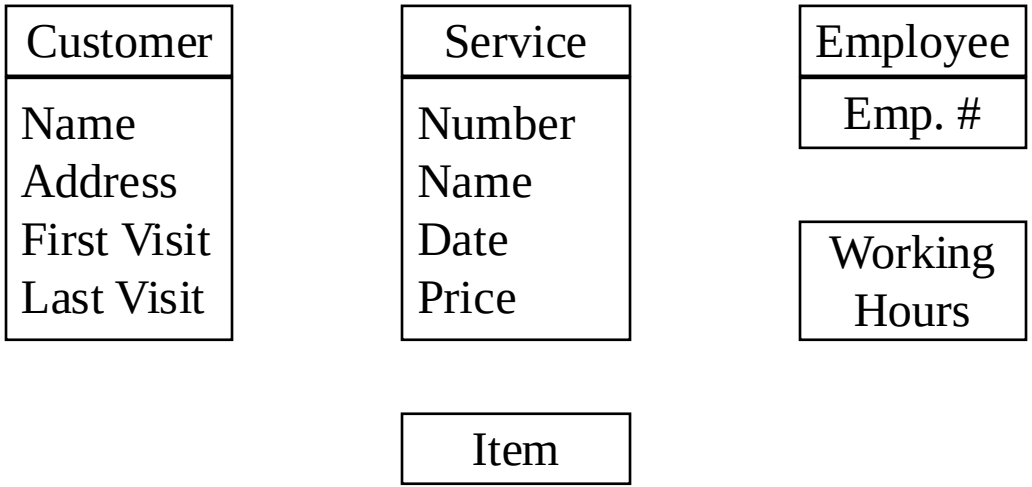
How to find classes

- Can you identify concrete objects in the application?
  - In technical systems, you can use **real objects** as a starting point. Define classes for these objects.
  - For example, if you are modeling an anti-blocking brake, you will need to model the brake, the actuator, and the sensor that detects the wheel (not) turning.
  - In business systems, you can often identify classes and attributes in **forms** that are filled in or in existing **screens**. Combine related attributes into classes. This is especially useful when reengineering an existing system.
  - Are there actors or objects, for which one needs to record something? These are potential classes

Barbershop Management System - Screen

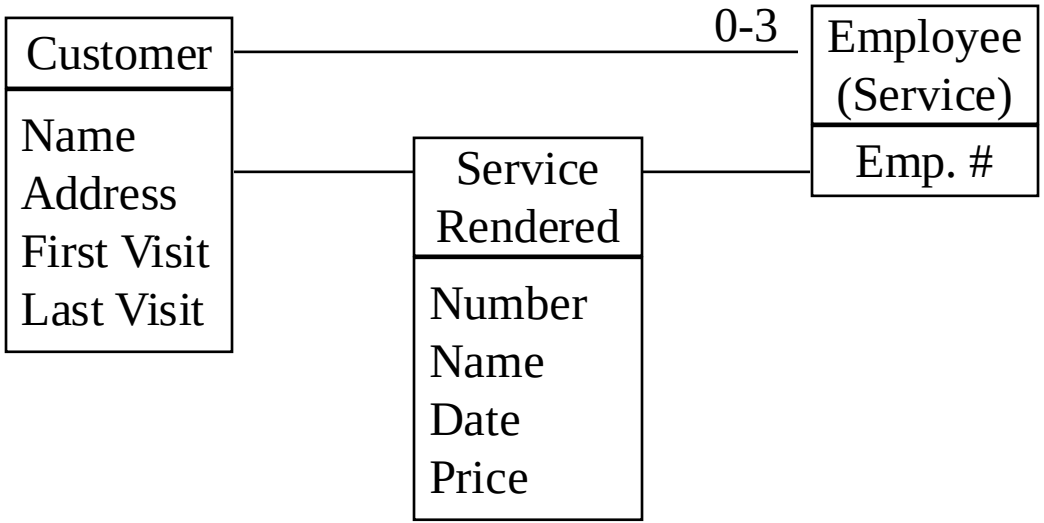


Results from Forms Analysis (Screen of Existing System)



Finding Associations

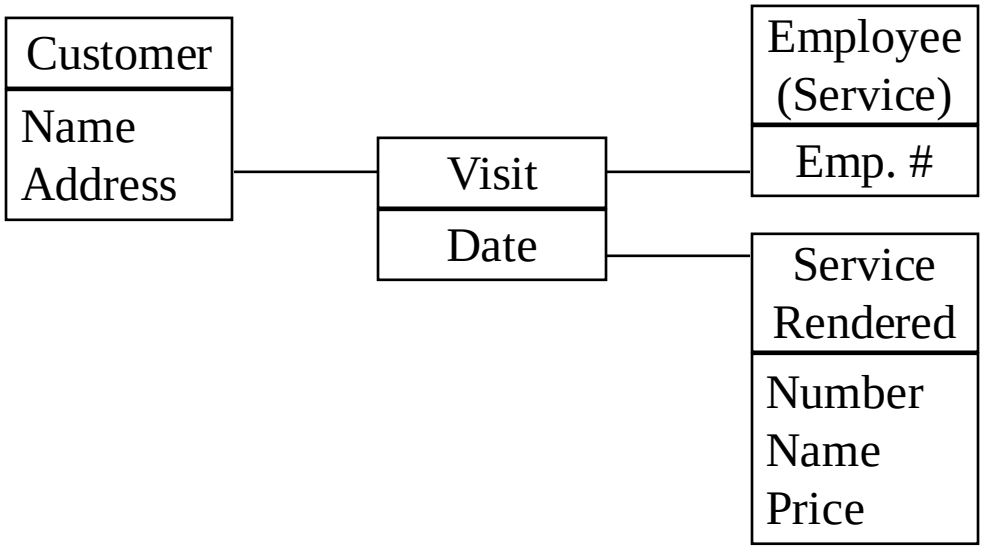
The old version stores only the details of the last visit, which leads to this structure:



Finding associations

- Look for relations between classes in the specification documents.
- New requirements dictate that *all* services be recorded. Therefore, introduce class *Visit* to represent the connection between customer, stylist, and service.

Changes by Specifications Analysis (new system will be built this way)



Specification Analysis: use the requirements to find classes, associations, etc.

Example:

“The lift closes its door before it moves to another floor.“

Potential classes: lift, door, floor.

- Model the door as a class, if it has methods or attributes.
- The methods “closeDoor“ and “openDoor“ are probably contained in the class “Lift“, therefore “Door“ is probably not a class by itself.
- (classes that have no attributes, no methods, and no associations can normally be eliminated)
- Search the requirements text for additional classes, methods, etc.

Specification Analysis according to Abbott

Part of Speech	Modelled as:	Example
Noun	class	car, dog, lift
Name	instance	Peter
Intransitive verb	method	to do, to run, to think,....
Transitive verb	association	love someone,
	or method	eat something (w. parameter “something”)
Verb „is a“	inheritance	is a (kind of), are, comprises, includes, there are two types of X: X1 and X2
Verb „have s.t.“	aggregation	has, consists of, contains, possesses composed of, component of, made up of, belongs to, is building block, element, component
	or attribute	
(Modal verb	assertion	must, should)
Adjective	attribute	3 years old, green, 20 seats

Use only as first approximation!

And what about the methods?

Example:

“Alice changes the module with the editor“

- *Alice:* an instance of class *Person* or *Programmer*
- *Module:* a class, probably subclass of *File*
- *Editor:* a class, probably subclass of *Tool* or *Instrument*

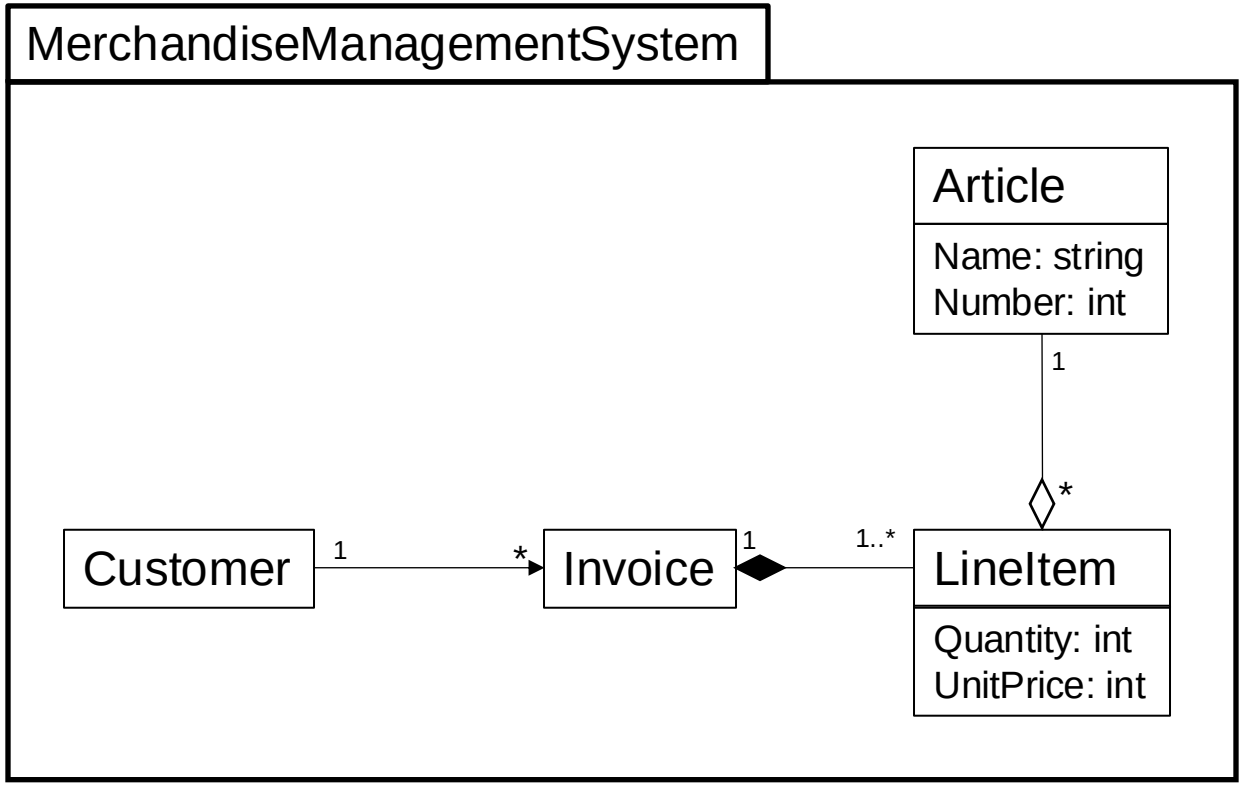
- change: transitive verb, an action (not a state like “to live in...“), therefore a method.
  change(module: Module, tool: Editor)
- probably belongs to class person or programmer

Exercise 1:

- Model the following situation as a UML class diagram:

- *A merchandise management system manages articles, customers, and invoices. An invoice consists of at least one invoice line item. Each invoice line item consists of an article, its quantity, and its unit price. An article has a name and a number. Each invoice is associated with a customer.*

Solution


MerchandiseManagementSystem
Article
Name: string
Number: int
1
*
Customer
1
*
Invoice
1
1..*
LineItem
Quantity: int
UnitPrice: int

Exercise 2:

- Model the following situation as a UML class diagram:

- *An issue tracking system for software stores so-called issues. An issue is either a bug report or a suggestion for an improvement. An issue consists of a descriptive text, a creation date, and a submitter. A bug report contains additionally a severity level as a digit from 1 to 5. Originally, an issue is in the state "submitted". It changes to the state "resolved", if it is linked with a commit into the configuration management system. A commit consists of one or several files that resolve the linked issue.*

Beware of nominalizations

- Seminar management:

  “Entry, change, and deletion of customers”

- entry, change and deletion are nouns, but not candidates for classes. The nouns are nominalizations of the transitive verbs enter, change, and delete, which are best modeled as methods.
- In nominalizations, subject and object are sometimes missing. In the above example: who is supposed to enter, change delete?
- Similar: *“... the function of entering“:* function is a noun, but not a candidate for a class. It indicates a method is following it in the text.

Finding classes:
Checking the candidates

When is a class probably redundant?

- the class has no attributes, no methods, and no associations.
- the class has the same attributes, methods, and association/aggregations as another class.
- the class contains only methods that could be placed in another class
- the class models implementation details
- the class has only one or two attributes. Could they be added to another class?

Naming conventions for classes

- Name should be expressive
- Name should be a noun or noun phrase in singular, capital first letter
  - as concrete as possible
  - stand for the entire set of attributes and methods in the class
  - must not be a role name in an association

Finding classes

- General note:
  - Do not yet construct inheritance relations, except for those already spelled out in the requirements document
  - It is OK to have a class with only one object
  - If in doubt about how much to bundle into a class, use smaller classes

Finding associations

- Is there a relationship between two or more objects?
  - Does it exist for a longer period?
  - Is the relationship relevant for the problem?
  - Does the relation exist independently of other, unrelated classes?

- Examples:
  - “to live in Paris“ (longer period: association)
  - „to serve someone“ (short period, transitive verb, action: method with parameter)
  - „painting the wall with a brush“ (nominalized „paint“, transitive verb, action, short period: method w. 2 parameters)
  - „company employs workers“ (transitive verb, long term: association)

When to use role names in associations

- The more general the class name, the more important is the role name.
- Add role names when:
  - the association is with the same class (such as for list elements)
  - there are several associations between the same classes
  - a class takes on different roles in associations with several other classes
  - (roles have no defined semantics in UML, but can help clarify what is being modeled)
- If there are several associations between the same classes, check whether they all add something, otherwise delete redundant ones.

Cardinalities in associations

- Is there a „must have“ relation?
  - As soon as an object is generated, the link to the other object must also be established.
- Is there a „may have“ relation?
  - The relationship can be established at an arbitrary point in time after the object is generated
- Is the upper limit fixed or variable?
  - Is a fixed upper limit dictated by the problem?
  - If in doubt, use a flexible upper limit (*)

Finding attributes

- Attributes can be found in requirements texts:
  - adjectives can indicate an attribute (“connection closed” means that there is an attribute “connectionStatus”)
  - phrases that express „has“, „consists of“, etc., see Abbott´s method.
    - primitive data types are modeled as attributes
    - objects can be modeled as attributes, with associations, or as part of aggregations/compositions
- Attributes can be found in screens and forms, as subordinate elements in a group (see “customer” in the barbershop example)
- Attributes can reflect real properties as found in “technical data“, “technical specifications“, “ingredients“) (size, weight, capacity, voltage, horsepower, etc.)
- Is an attribute ever visible at some user or external interface? If not, then it is probably an implementation detail and can be omitted.
- Check whether the attribute belongs to a class or to an association.

Attribute conventions

- Attribute name should be
  - expressive
  - a noun or adjective-noun combination
  - begin with lower case
  - not repeat the class name
  - unique and understandable within class

Class attributes

Use class attributes if:

- all objects of a class have an attribute with the same value.
  - Example: the number of corners of all rectangles is 4.
- information about the objects of a class as a whole
  - Example: number of objects of a class
- information that is needed by the constructor
  - Example: default values that may change from time to time.

Design of inheritance hierarchy

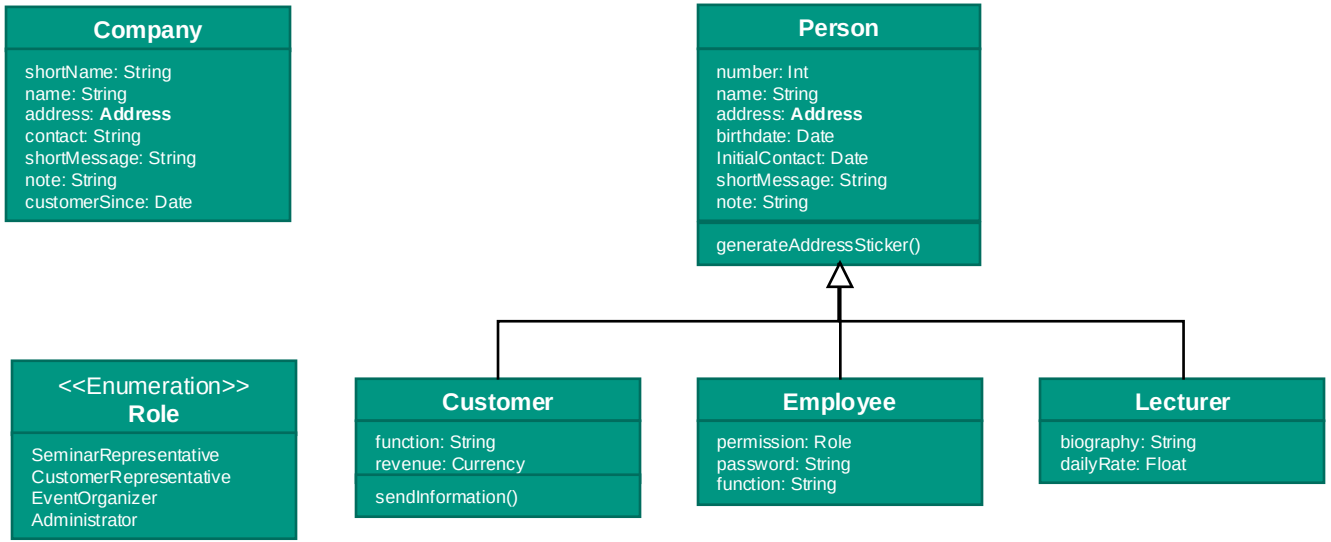
- Follow the substitution principle: Make U a subclass of O only when it is sure that every U can also be seen as an O.
- Pull up attributes and methods that are common to several classes into a common superclass (bottom-up method).
  - Is the superclass abstract?
- Choose a superclass: is there a class that already has several attributes and methods that also occur in another class? (top-down method)
- Avoid “God classes“: Split large classes into coherent, smaller ones.
  - can you split attributes and methods into sets, such that attributes and methods of one set use each other, but nothing of another set? Then split!
- In practice, there are many associations, and relatively few inheritance relations.

Design of inheritance hierarchy

- The classes `Sub1` and `Sub2` are subclasses of class `Super` **`if:`**
  - It is necessary to distinguish `Sub1` and `Sub2`
  - `Sub1` und `Sub2` inherit the attributes/methods/associations of `Super`, but they also have additional (different!) ones or overwrite methods of `Super` differently.
  - all **`exemplars of`** `Sub1` and `Sub2` can be viewed as exemplars of the superclass `Super`.

Example: Seminar organisation

Company and Person have many things in common, but they also have important differences (for example, companies have no birthdate), so a firm is not a subclass of person. But address has been pulled out in a separate class.



Determine life cycle of objects

- Design state diagrams for classes where necessary.
- Is there a non-trivial life cycle for an object?
  - The life cycle is trivial, if there is only one state between allocation and deallocation of an object. The object always reacts in the same way in a given context. No state diagram is needed in that case.
  - non-trivial, if methods can only be applied in certain situations, or when the same event can cause different actions (depending on the state). Embedded systems or control systems often have many states and therefore need a state diagram.

*Developing Object-Oriented Software – An Experience-Based Approach, Prentice Hall, 1997

Finding methods

- Take methods from sequence, activity, or state diagrams or from verbs in the textual requirements
- Choose an expressive name
  - beginning with a verb
  - name should describe the computation
- split large methods into several (private helper methods)

Subsystems or packages

- Combine classes with a common purpose into packages or subsystems.
  - Strong cohesion within a subsystem
  - weak coupling between subsystems (only few associations should cross subsystem boundaries)

Literature

- Brügge, Dutoit: Object-Oriented Software Engineering, Kap. 5 (**read it!**)
- Meyer: Object-oriented software construction, 2. edition, Prentice Hall, 1997