Hugbúnaðarverkefni 1 / Software Project 1

9. Domain Models

HBV501G – Fall 2018

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI- OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz Prep

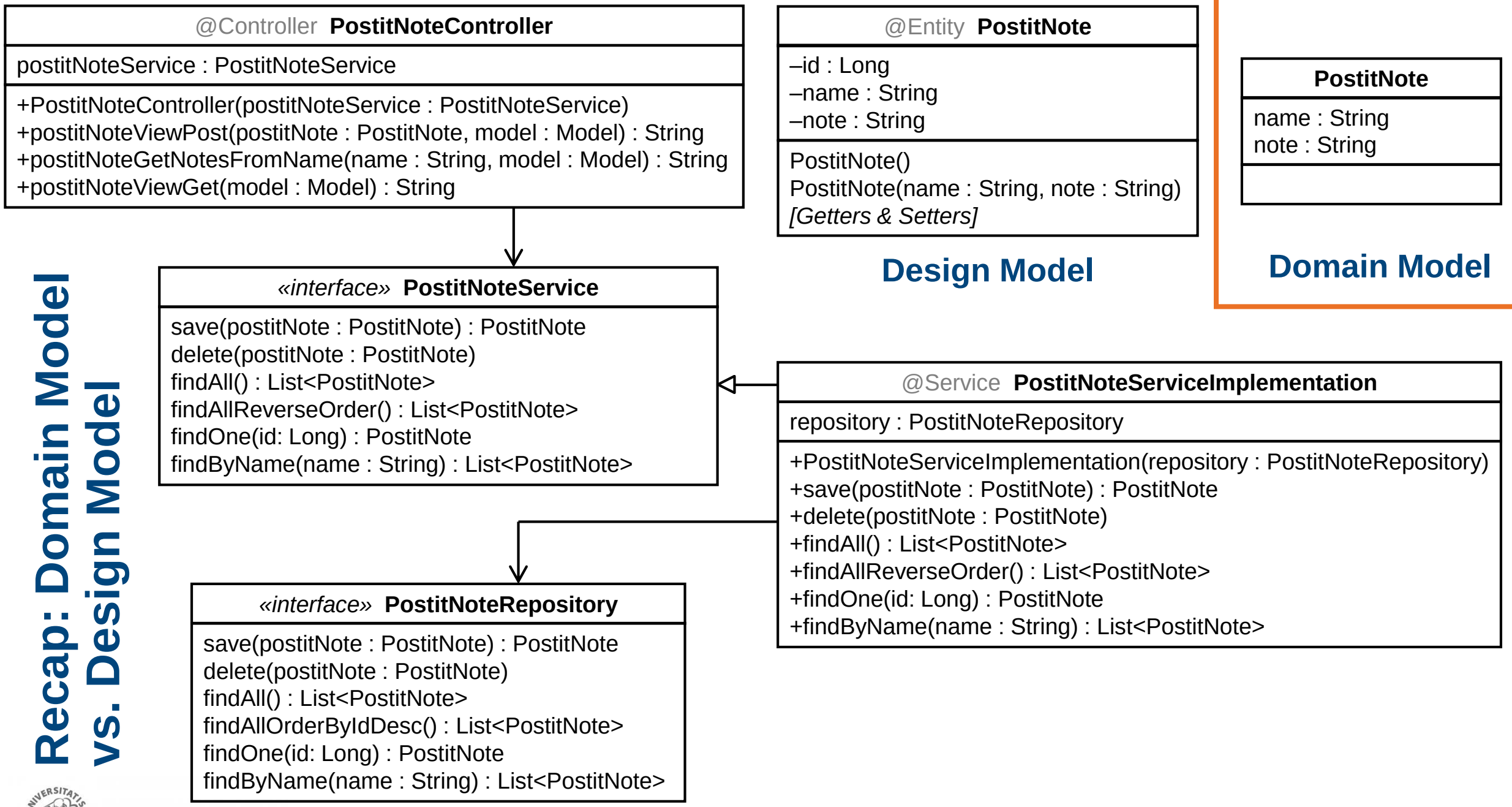
- Please prepare a scrap of paper with the following information:

- ID: _____@hi.is Date: _____
- a) _____ e) _____
- b) _____ f) _____
- c) _____ g) _____
- d) _____ h) _____

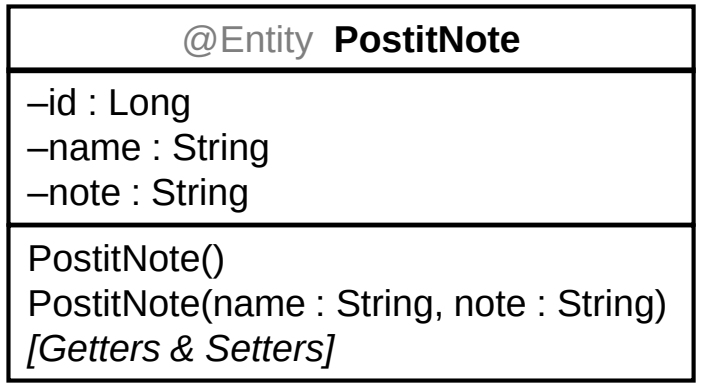
- During class, I'll show you questions that you can answer very briefly
  - No elaboration necessary
- Hand in your scrap at the end of class

- All questions in a quiz weigh same
- All quizzes (ca. 10 throughout semester) have the same weight
  - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam

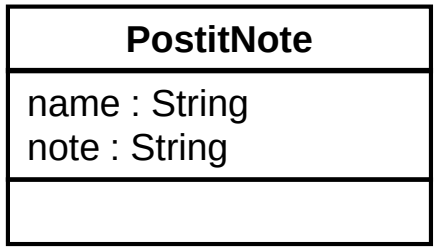
Recap: Domain Model vs. Design Model



From Domain Model to Design Model with the JPA



Design Model

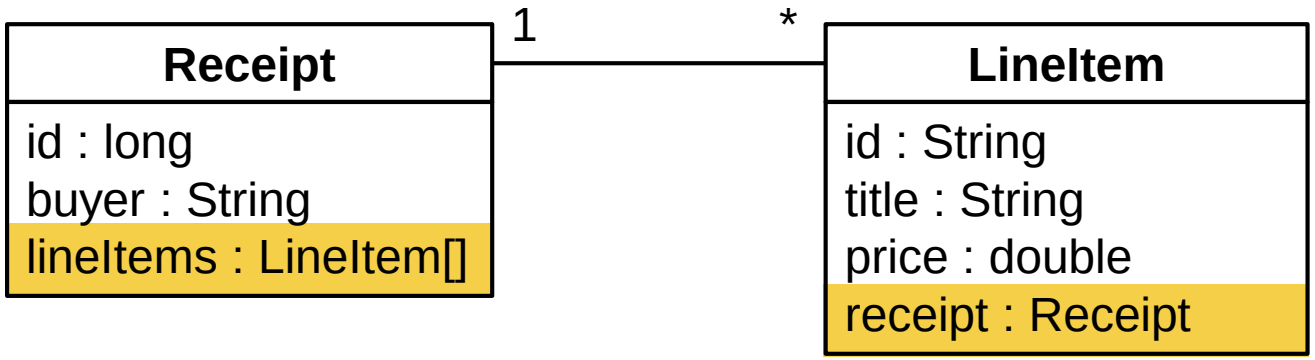


Domain Model

- Using the Java Persistence API (JPA) as shown in the previous lectures allows us to maintain an object-oriented perspective when moving from the domain model to the design model.
- Besides the addition of the constructors and primary key ID, we rarely need to think about how to map our entities to relational database tables, and how to retrieve them – the JPA is performing these tasks for us in the background!
- For example, in a one-to many relationship between two entities A and B, we typically get access to all instances of B associated with A automatically as object references, rather than having to make explicit database queries for them.

Recap: One-to-Many Entity Relationships in UML

- Example: Aggregation of line items in a sales receipt



Recap: One-to-Many Entity Relationship using JPA

```
@Entity
public class Receipt {
  private long id;
  private String buyer;
  private Set<LineItem> lineItems =
    new HashSet<>();

  @Id
  @ColumnName(name = "ReceiptId")
  @GeneratedValue(strategy =
    GenerationType.IDENTITY)
  public long getId() {...}
  public void setId(long id) {...}
  public String getBuyer() {...}
  public void setBuyer(String buyer) {...}

  @OneToMany(mappedBy = "receipt",
    fetch = FetchType.LAZY,
    cascade = CascadeType.ALL,
    orphanRemoval = true)
  public Set<LineItem> getLineItems()
    { return lineItems; }
  public void setLineItems(
    Set<LineItem> lineItems) {
      this.lineItems = lineItems;
    }
}
```

Recap: Many-to-One Entity Relationship using JPA

```
@Entity
@Table(name = "Receipt_LineItem")
public class LineItem {
  private long id;
  private String title;
  private double price;
  private Receipt receipt;
  @Id
  @ColumnName(name = "LineItemId")
  @GeneratedValue(strategy =
    GenerationType.IDENTITY)
  public long getId() {...}
  public void setId(long id) {...}
  public String getTitle() {...}
  public void setTitle(String title) {...}
  public double getPrice() {...}
  public void setPrice(double price) {...}

  @ManyToOne(fetch = FetchType.EAGER,
    optional = false)
  @JoinColumn(name = "ReceiptId")
  public Receipt getReceipt()
    { return receipt; }
  public void setReceipt(Receipt receipt)
    { this.receipt = receipt; }

}
```

Recap: Assignment 3

Review slides on sequence diagram syntax and structure in HBV401G!

- By **Sun 28 Oct**, submit as PDF document(s) in Ugla:
  - **A UML class diagram** showing the **design model** of your project:
    - Including the classes your final system shall consist of
      - with attributes, data types, methods, relations, multiplicities
  - A **UML sequence diagram** illustrating the control flow:
    - Showing the collaboration of JSPs, controllers, services, repository, and potential client-side components in one exemplary request-response cycle
      - Make sure the method calls in the sequence diagram reflect the methods specified in the class diagram
- On **Thu 1 Nov**, explain the diagrams of your design model to your tutor:
  - Why did you choose particular solutions?
  - Which aspects of the system's behavior are particularly complex?
  - Where do you see room for design improvement (if you had time after the semester)?
  - How are the MVC pattern and the design principles of encapsulation, separation of concerns etc. reflected in your design (especially if you have client-side code)?

Assignment 4 & 5 Schedule Preview

- **Assignment 4: Code Review**
  - By **Sun 11 Nov,** make the **project artifacts** you created available to another team:
    - Your project vision and design model from previous assignments (incl. fixes of identified issues)
    - A current snapshot of your source code
  - Take **one week** to **review** the other team's code and briefly **document** your findings:
    - Examine how they structured their system. Do vision and design model help to understand it?
    - Make suggestions for improvements in component design, technology use and coding style
  - By **Sun 18 Nov,** submit your **review report** to Ugla
    - What did you like? Which questions did you run into? What would you recommend to change?
  - On **Thu 22 Nov, discuss** your findings with the other team and your tutor.

- **Assignment 5: Final Product Presentation**
  - On **Thu 29 Nov, present** your **final product** to the other teams and your tutor.
  - By **Sun 2 Dec**, **submit** your final product's **source code**.

Course Schedule (updated)

Week	Thu: Team Consultations	Fri: Lectures	Sun: Assignments due
1		Introduction	
2	Phase 1: Inception	Rational Unified Process	Team Formation (9 Sep)
3	Phase 1: Inception	Requirements Engineering	
4	Phase 1: Inception	Requirements Engineering	#1: Vision & Scope Doc (23 Sep)
5	Vision & Scope Present. (27 Sep)	Spring Web Applications	#2: Use Case Document (30 Sep)
6	Use Cases Presentation (4 Oct)	Spring Web Persistence	
7	Phase 2: Elaboration	JavaServer Pages *(video)*	
8	Phase 2: Elaboration	Design Models	
9	Phase 2: Elaboration	Domain Models	#3: Design Model (28 Oct)
10	Design Model Present. (1 Nov)	Interaction Room	
11	Phase 3: Construction	Design Patterns	Exchange code w other team (11 Nov)
12	Phase 3: Construction	Design Patterns	#4: Code Review Report (18 Nov)
13	Code Review Present. (22 Nov)	Guest Lecture?	
14	Final Project Present. (29 Nov)	Final Exam Prep	#5: Final Project (2 Dec)

Domain Models

see also:

- Larman: Applying UML and Patterns, Ch. 9 & 32

Recap: Domain Models vs. Design Models

Focus of this lecture

- We create **domain models** in order to **understand the application domain**
  - So we can talk to the business stakeholders about their requirements on their own terms
  - So we can introduce new team members to the purpose of the software before introducing them to its implementation

➢ Domain models illustrate business aspects / requirements / **problem domain**

- We create **design models** in order to express **how our solution shall work**
  - So we can talk to fellow developers
    - about what the best technical solutions to a given domain problem would be
    - about the concrete structures that we plan to implement
  - So we can introduce new team members to the internal structure/mechanics of the software after they have understood the application domain

➢ Design models illustrate technical aspects / implementation / **solution domain**

Focus of previous lecture and Assignment 3

Goals of Domain Models

- Illustrates noteworthy concepts / terms / objects in an application domain
  - i.e. an object-oriented model of the real world
    - more precisely: the *segment* of the real world that is *relevant* to your system
    - Note: Do not try to model the real world in all its detail – time-consuming and largely irrelevant!

- Source of inspiration for (later) object-oriented design of application logic
  - i.e. shows which concepts, relationships, dynamics the system must support
    - even if not all domain model elements may have exact counterparts in the implementation
    - Note: Do not model implementation-specific classes (i.e. your system's technical design) here!

➢ **Goal: Understand the application domain, so you can build software supporting it properly**

Creating a Domain Model

- A domain model is typically expressed as a UML class diagram showing
  - Conceptual classes (relevant things, concepts, ideas in the application domain)
  - Associations between conceptual classes
  - Attributes of conceptual classes (but no methods)

- UML sequence or (more commonly) activity diagrams can be added to understand the dynamics of the application domain

- **Approach:** For the requirements under design *in the current iteration*:
  1. Identify the conceptual classes
  2. Draw them as classes in a UML class diagram
  3. Add associations and attributes (but no methods)
  4. Draw UML sequence or activity diagrams to express dynamic aspects

Use a Pragmatic Modeling Approach

- Do not guess
  - Work together with domain experts, i.e. people who are regularly exposed to the concepts
- Do not spend more than a few hours on domain modelling in each iteration
  - Your goal is not to precisely document the whole world
  - But to understand what the relevant aspects for your system are, and how they work
- Do not use a modeling tool (unless the model is massive)
  - Just use a whiteboard and take a picture when you are done
    - Much more intuitive and straightforward, especially for domain experts working with you
    - Domain model will ultimately be replaced by your system model, which is worth more detailed modeling and preservation effort

Key Challenges in Domain Models vs. Design Models

- We create **design models** in order to express **how our solution shall work**
  - Key challenge is engineering a solution that enables efficient implementation, execution and maintenance, i.e. **creating an efficient design**
  - Supported by: Object-oriented design guidelines, design patterns

Quiz #6 Solution: Object-Oriented Design Heuristics

Which goals (a-h) do the principles (1-8) work toward?

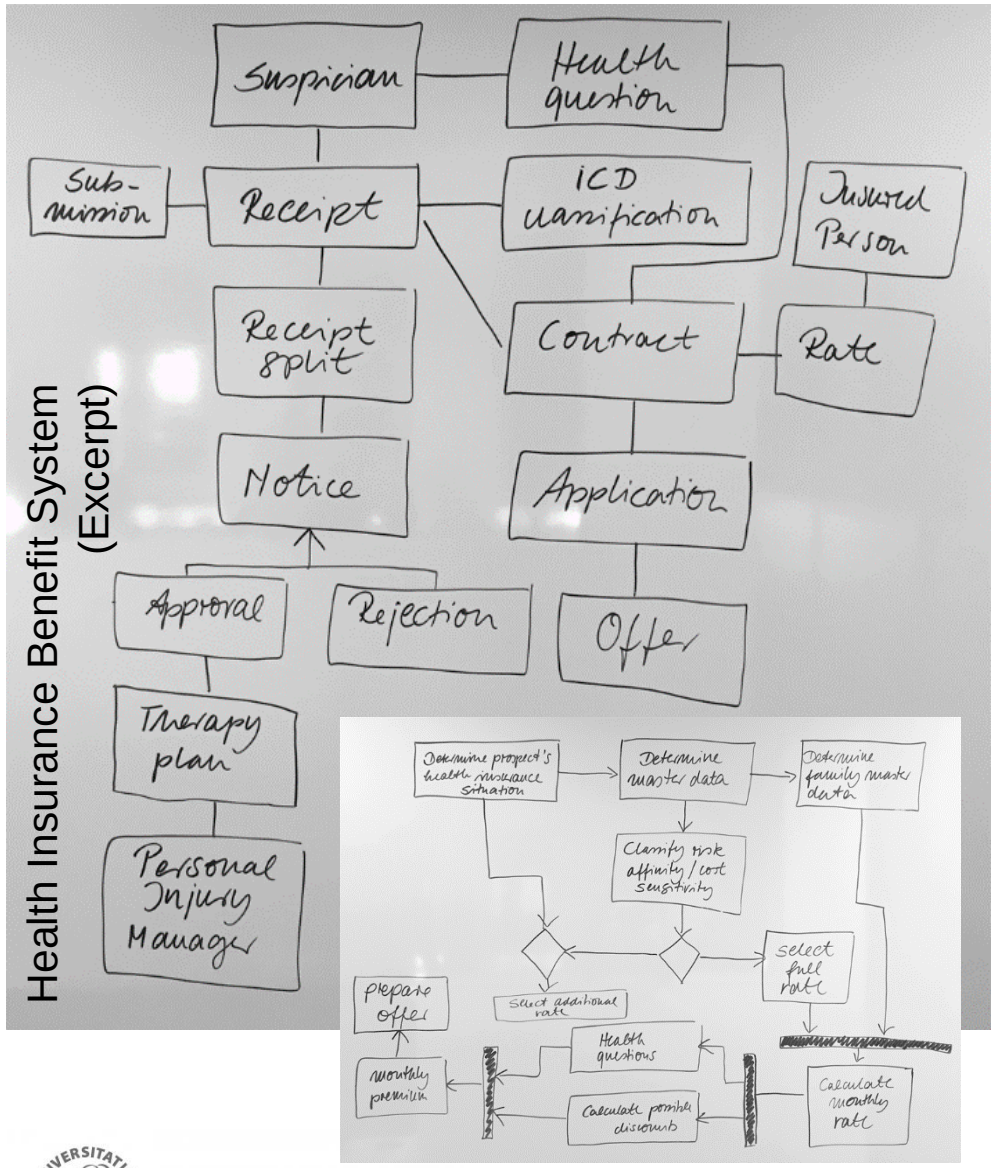
a) Assigning responsibility for reacting to user input	**2.** Controller
b) Decoupling objects even if they need to work closely together	**5.** Indirection
c) Expressing support concepts not present in the application domain	**3.** Fabrication
d) Expressing variations in object behavior	**8.** Polymorphism
e) Identifying the most suitable creator of objects of a particular class	**1.** Creator
f) Keeping objects independent from other objects	**6.** Low coupling
g) Keeping objects' responsibilities focused	**4.** High cohesion
h) Protecting other objects from variation/instability within an object	**7.** Protected variations

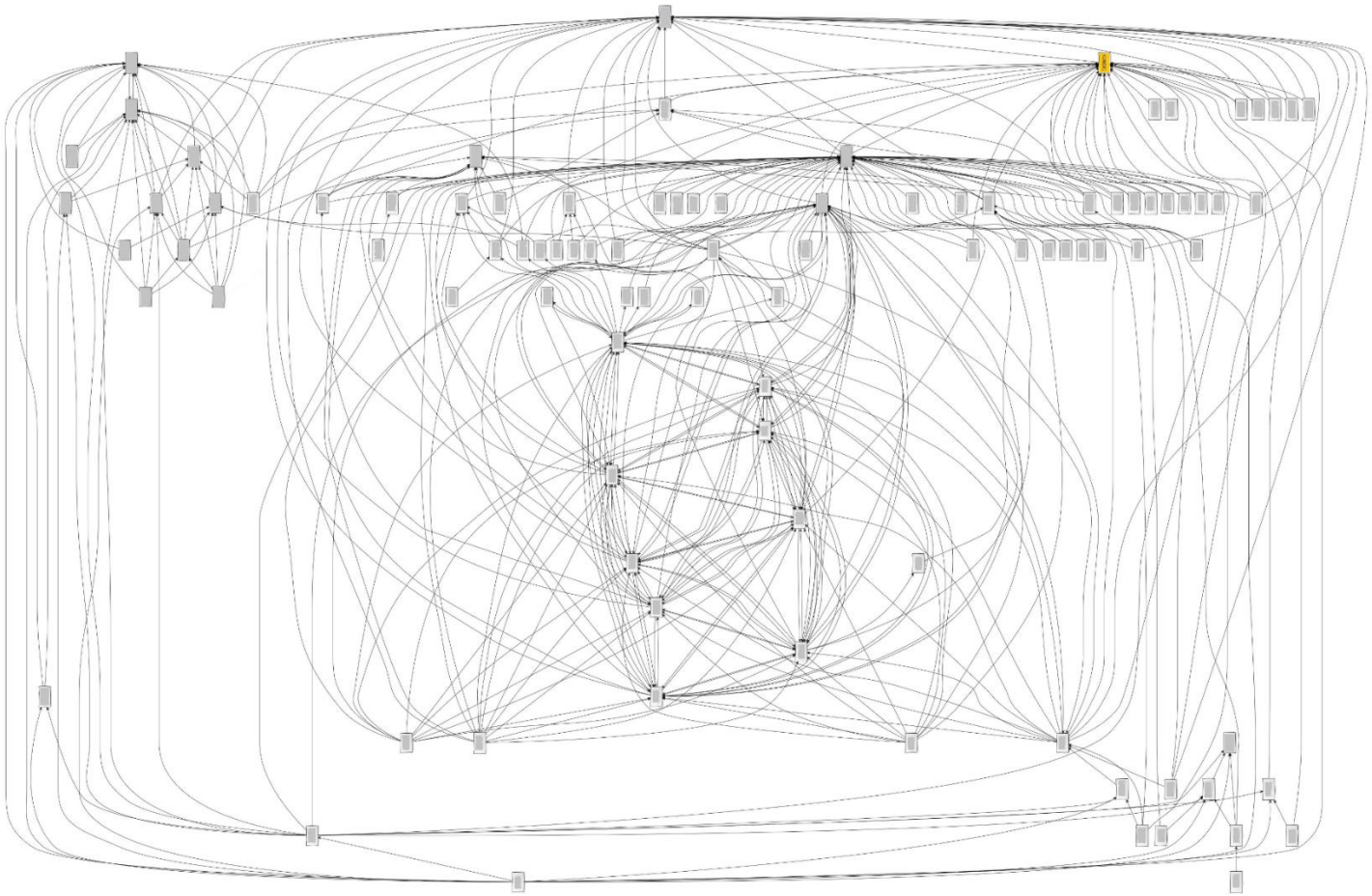
Key Challenges in Domain Models vs. Design Models

- We create **design models** in order to express **how our solution shall work**
  - Key challenge is engineering a solution that enables efficient implementation, execution and maintenance, i.e. **creating an efficient design**
  - Supported by: Object-oriented design guidelines, design patterns

- We create **domain models** in order to **understand the application domain**
  - Key challenge is understanding all of the domain's entities and their relationships, i.e. **not missing anything important**
  - Supported by: Strategies for identifying classes, guidelines for expressing relationships

Domain Model Examples from Actual Projects





Health Insurance Core System
(120 classes, 6 million customers with associated data objects)

Strategies for Identifying Conceptual Classes in Complex Domain Models

a) Extract noun phrases
   - e.g. in fully dressed use cases and the Vision and Scope document

b) Use a category list
   - especially helpful for business information systems

c) Reuse or modify existing models
   - something may be available from the client (but not necessarily up-to-date)

Identifying Conceptual Classes by Extracting Nouns

- Example: Extraction from a use case's main success scenario
  1. Customer arrives at POS checkout with goods and/or services to purchase
  2. Cashier starts new sale
  3. Cashier enters item identifier
  4. System records sale line item and presents item description, price, and running total. Price calculated from set of price rules.
  5. System presents total with taxes calculated.
  6. Cashier tells Customer the total, and asks for payment.

- Caution: Easy for a quick start, but needs refinement with common sense
  - Not every noun is a conceptual class
  - Some conceptual classes may not be mentioned explicitly
  - Some nouns may refer to classes, some to attributes
  - Natural language can be ambiguous

Identifying Conceptual Classes From Categories

Conceptual Class Category	POS System Examples	Airline Examples	Game Examples
Business transactions	Sale, Payment	Reservation	
Transaction line items	Sales Line Item		
Products/services related to transactions/line items	Item	Flight, Seat, Meal	
Actors/roles of people related to transactions	Cashier, Customer	Passenger, Airline	Player
Place of transaction/service	Store	Airport, Plane, Seat	
Noteworthy events (often with assoc. time or place)	Sale, Payment	Flight	Game
Physical objects	Item, Register	Airplane	Board, Piece, Die
Descriptions of things (type information)	Product Description	Flight Description	
Catalogs	Product Catalog	Flight Catalog	
Containers of things	Store, Bin	Airplane	Board
Things in a container	Item	Passenger	Square, Piece
Collaborating systems	Credit Authorization System	Air Traffic Control	
Records of finance, work, transactions etc.	Receipt, Ledger	Maintenance Log	
Financial instruments	Cash, Check, Line of Credit	Ticket Credit	

For inspiration only – you typically won't have classes for all of these

Describing the Application Domain (“Real World”)

- **Use the existing terminology** of the application domain, don’t invent new terms
  - If there is a precise term for a concept, use it so users will know what you are talking about
  - Terms you choose will stick all the way to the implementation, user interface, documentation etc.
- **Exclude irrelevant or out-of-scope aspects** of the application domain
  - If your system’s implementation doesn’t need to “know” about it, don’t include it in the model
  - You can include concepts that will become relevant in later iterations (so you’re already aware of them), but you should not spend effort on describing them in detail yet
- **Don’t add concepts** to the model if they don’t exist in the application domain
  - Of course, you should model relevant non-physical concepts (e.g. Agreement, Route, Target…)
  - Don’t add concepts based on your assumptions or opinions on how things should work
  - Don’t add concepts that you need for technical reasons

➢ **How would you describe the structures and processes that your system supports**

- **to someone who just cares about what the system does, but not how it works?**
- **to someone who shall implement the system using a technology unknown to you?**

Example: POS System Domain Model with Identified Conceptual Classes

Ledger

ProductCatalog

ProductDescription

SalesLineItem

Item

Store

Sale

CashPayment

Customer

Register

Cashier

Adding Associations to Domain Models

- An association between two classes indicates the presence of a **relevant, memorable relationship** between instances of those classes.
  - i.e.: Between which objects do we need a long- or short-term memory of a relationship? E.g.:
    - A SalesLineItem must be permanently associated with a Sale
    - A Piece must be permanently associated with a Player, and temporarily with a Square on a Board
    - But while a Cashier can look up ProductDescriptions, there is no need to remember which Cashier looked up which ProductDescriptions (i.e. no association)

- Associations can also indicate other common relationships
  - see suggestions on next slide

- Beyond that, be reluctant about adding too many associations
  - Focus on the main structural bonds, not everything that somehow has to do with something

Common Relationships Expressed by Associations

Association Category	POS System Examples	Airline Examples	Game Examples
A is a transaction related to another transaction B	Cash Payment – Sale	Cancellation – Reservation	
A is a line item of a transaction B	Sales Line Item – Sale		
A is a product for a transaction (or line item) B	Item – Sales Line Item	Flight – Reservation	
A is a role related to a transaction B	Customer – Payment	Passenger – Ticket	
A is a physical or logical part of B	Drawer – Register	Seat – Airplane	Square – Board
A is physically or logically contained in B	Item – Shelf	Passenger – Airplane	Square – Board
A is a description for (or: the type of) B	Item Description – Item	Flight Description – Flight	
A is known/recorded/reported/captured in B	Sale – Register	Reservation – Flight Manifest	Piece – Square
A is a member of B	Cashier – Store	Pilot – Airline	Player – Game
A is an organizational subunit of B	Department – Store	Maintenance – Airline	
A uses/manages/owns B	Cashier – Register	Pilot – Airplane	Player – Piece
A is next to B	Sales Line Item – Sales Line Item	City – City	Square – Square

For inspiration only – you typically won't have associations for all of these

Using and Interpreting Associations in Domain Models

- Domain model associations are **bidirectional**
  - Arrow tips can be added for clarity, but don't prescribe references between software classes
- Associations should be **labeled** with verb phrases connecting the class names
  - e.g. Store stocks Item, Sale paid-by Cash Payment, Flight flies-to Airport, Piece is-on Square
- Ends of associations can be decorated with **multiplicities** indicating how many instances of a class can be associated with one instance of the other class at the same time
  - e.g. 0..1 (zero or one), * (zero or more), 1..* (one or more), 1 (exactly one), 1..42 (one to 42)
- **Multiple** associations can connect classes to symbolize different relationships
  - e.g. Flight flies-from Airport, Flight flies-to Airport
- An association can be **reflexive**, i.e. relating instances of the same class
  - e.g. Folder contains Folder

Associations in Domain Models vs. Design Models

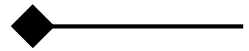
- **In an application domain model,** associations are NOT
  - Statements about data flows
  - Foreign key relations in a database
  - Instance variables
  - Object references in software

- Upon moving towards **technical design models** later…
  - Some domain model associations may survive in the technical model of the system and indicate object references, method calls, visibility, etc.
  - Some domain model associations may be implemented differently/indirectly
  - Some domain model associations may be eliminated or replaced with other constructs
  - Some associations may be added to facilitate convenient data access
- …but you shouldn't consider any of this yet when creating the domain model.

Recap: Aggregation and Composition

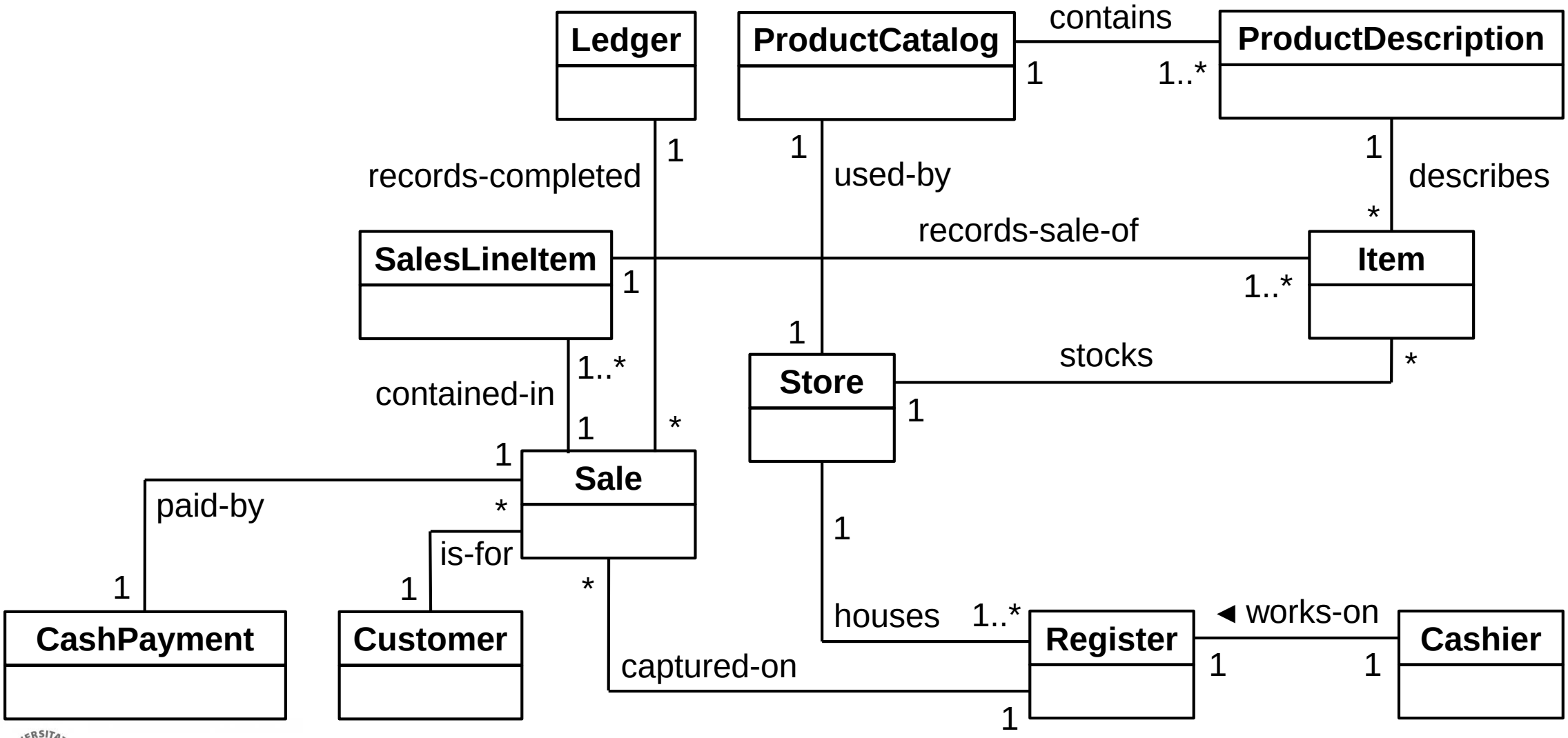
Base, not tip of “arrows”!

- **Aggregation** describes a “contains” relationship
  - e.g. a Container holds Goods
  - But is only weakly defined in the UML, has no clear implications for the technical design, and is just as well expressed by regular association with 1..* multiplicity and “contains” label
  - ➢ Guideline: Don’t bother modeling it explicitly



- **Composition** describes a “consists of” relationship implying that
  - A part instance (e.g. Square) belongs to only one composite instance (e.g. Board) at a time
  - Any part must belong to a composite at any time (“no loose Fingers”)
  - [Weaker semantics: Composite should be responsible for creation and deletion of parts, i.e.
    - upon its creation, a composite creates or requires its constituent parts
    - upon its deletion, a composite deletes its parts or attaches them to another composite]
  - Do not rely on people understanding and implementing these semantics, but make them explicit in design model through comments
  - ➢ Guideline: Can be defined in the domain model, but is not crucial – if in doubt, leave it out

Example: POS Domain Model with Associations



Adding Attributes to Domain Models

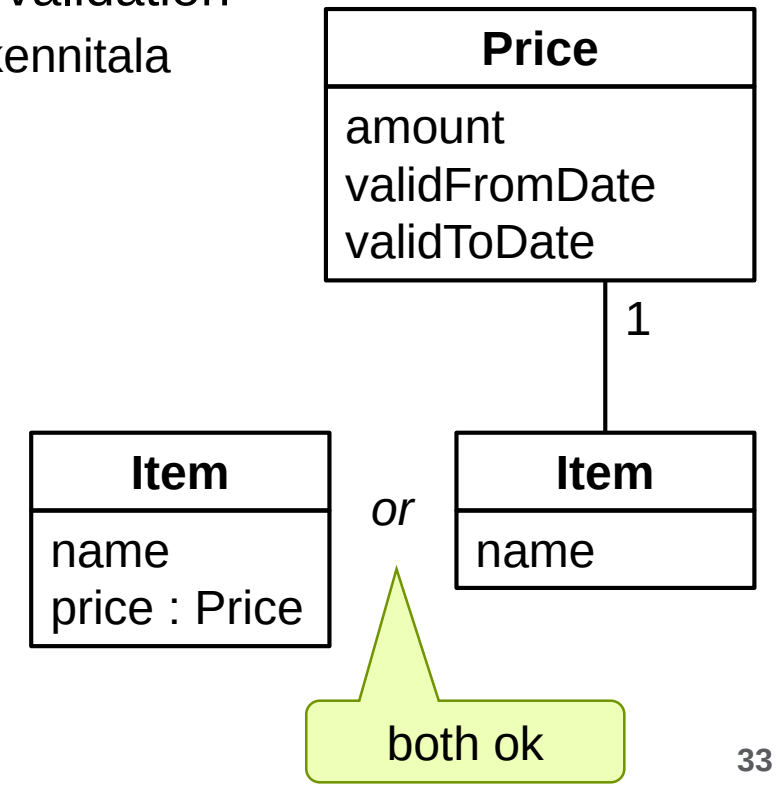
- An attribute is a logical data value of an object.
  - e.g. a Sale needs Date and Time attributes, a Product needs a Price attribute, etc.
- Guideline: Include attributes in the domain model that…
  - are needed to satisfy information requirements expressed in use cases
  - characterize individual instances of a class in a relevant way
- Attributes are typically primitive data types or trivial data structures
  - i.e. they are Boolean values, dates, numbers, characters, text, or addresses, phone numbers, colors, shapes, product codes, ISBNs, enumerated types (Size = {S, M , L}) etc.
  - But their data type is typically not specified in the model (unless it is non-obvious), and does not prescribe particular types to be used in the technical design
- Do not include attributes that are merely technical IDs or foreign keys
  - An ISBN is fine (part of the real world), a database ID is not (implementation detail)
- Do not include derived attributes, unless their omission would be confusing
  - e.g. no need for “total price” attribute of a Sale that can be derived from LineItems’ prices

Classes vs. Attributes

- When should a thing be modeled as a conceptual class, when as an attribute?
- Guidelines:
  - If we do not think of something as a number or text in the real world, it is probably a conceptual class, not an attribute.
    - Example: A Flight class' destination airport should not just be a string attribute ("KEF"), but an associated Airport class which may have attributes of its own
  - If equality is based on identity instead of value, it should be a conceptual class
    - Example: 1000 ISK and 1000 ISK are the same price (equality based on value → attribute), but Jón Jónsson and Jón Jónsson may be two distinct people (equality based on identity → class).
- Conceptual classes should always be related with an association, not expressed as an attribute in the domain model *(see next slide for exception)*
  - To facilitate richer description of the conceptual classes and associations
  - This does not prescribe a certain technical implementation.

Classes vs. Attributes

- Sometimes, an aspect that is originally considered to be of a primitive type (i.e. an attribute) may turn out to be better represented as a conceptual class, e.g. if:
  - It is composed of separate **sections** (that are relevant for the system to distinguish)
    - e.g. name → first name, middle name, last name, prefix, suffix, gender…
  - There are **operations** associated with it, such as parsing or validation
    - e.g. validate a credit card number, extract a birth date from a kennitala
  - It has other relevant **attributes**
    - e.g. a price may have a start and end date, be valid for a certain amount of items only, etc.
  - It is a quantity with a **unit that is not fixed**
    - e.g. payments in different currencies
- These kinds of conceptual classes may be specified as an attribute or an association in a domain model
  - Choose whichever way seems more natural



Modeling Type Information

- How can type information (i.e. general characteristics of a group of objects) be explicitly modeled?
- Example:
  - A store sells items with some generic, some individual characteristics (e.g. custom stamps).
    - Generic characteristics of all stamps: Manufacturer, material, price…
    - Specific characteristics added upon ordering: Individual lettering
  - Is it sufficient to have just one Stamp class, or should we distinguish between GeneralStamp and OrderedStamp classes?
- Guiding questions:
  - What would happen if all individual stamp objects were deleted?
    - Would your system miss crucial information to sell new stamps?
  - What should happen if some general characteristic of stamps had to be changed?
    - Would you need to change information in all individual stamp objects as well?
    - Or would you want to prevent the individual stamp objects from being affected by changes?

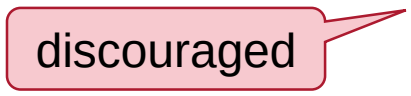
Modeling Type Information

- Guidelines: Add a type class (e.g. ItemDescription for an Item) if
  - there needs to be a description of an item,
    independent of the current existence of any instances of that type of item
  - deleting instances of items would result in loss of information
    that needs to be maintained about general characteristics of that type of item
  - it avoids the storage of redundant or duplicated information in each instance of the item

- Example: Modeling customizable stamps

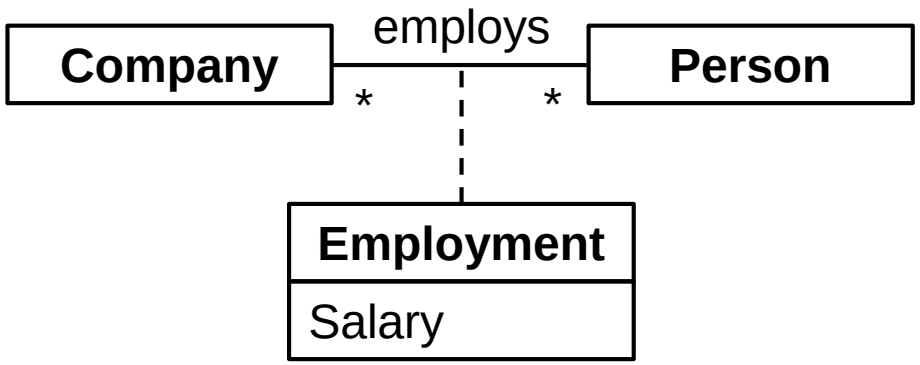
preferred

discouraged



Association Classes

- Associations may have attributes just like classes do
  - Example: A person may have employment contracts with several companies, but the contract details are neither attributes of the person nor of the company
    - They are attributes of the association
  - ➢ Model such attributes in **association classes**



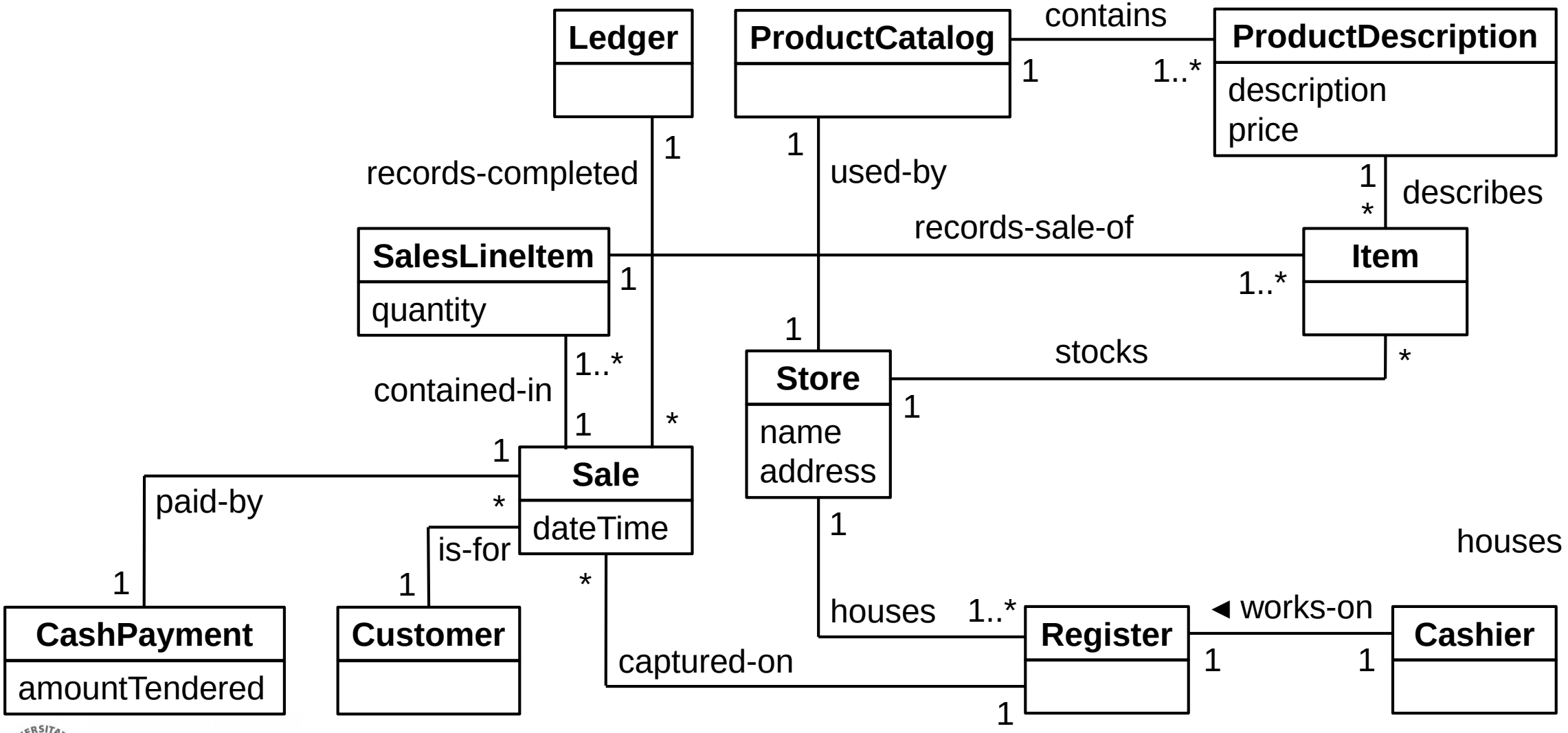
- Guidelines for association classes:
  - Use whenever a attribute is related to an association, but not the classes it connects
  - There is a many-to-many association between two concepts, and information associated with the association itself
  - The lifetime of instances of the association class depend on the lifetime of the association

Recap: Generalization and Specialization

- Inheritance between super- and sub-classes can be used in domain models
  - Expressed in UML with inheritance generalization arrows (“extends”)
- Define a subclass of a superclass if
  - The subclass has additional attributes and/or associations of interest
  - The subclass behaves/is operated on/handled/reacted to/manipulated differently than the superclass or other subclasses
- Define a superclass for subclasses if
  - Subclasses represent variations of a similar concept embodied in the superclass
  - All subclasses have the same attribute/association that can be factored out to the superclass
- Abstract classes can be used in domain models as well
  - For generic concepts, where concrete instances can exist only of subclasses

- Note: These domain model constructs do not prescribe implementation choices.

Example: POS Domain Model with Attributes



The System as Part of the Real World

- **Q:** How am I supposed to model just the “real world”,
  when my system is influencing / shaping / enabling that world?

- **A:** A domain model is not supposed to model the world *without your system*,
  but just *without knowledge of* your system’s *implementation details*.

The System as Part of the Real World

- No matter if you are
  - developing a system supporting existing processes,
  - building a system that will enable a completely new business model,
  - designing a game that has no real-life equivalent:
- **Model how things are supposed to work once your system is in place**
  - Only if you need to understand a very complex domain, it may help to model how things are currently done first – but don't spend too much effort before moving on to model future vision
- This may mean that
  - your system is a key actor (e.g. matching customers and advertisers)
  - your system is introducing new conceptual classes (e.g. Timeline, Post, Like)
- That's fine – just stay on conceptual level, but don't go to technical level yet
  - No software classes, web pages, database tables etc. in a domain model
  - For the domain model, it shouldn't matter if you'll build a Windows, Web or Android app

Iterative-Incremental Development of Domain Model

- Don’t try to create “the one” correct and complete domain model early on
  - It will never be either correct or complete
  - It will cost more than you will gain from it

- Model only what you need to understand
  - To shape the overall architecture and interfaces of the system
  - To design and implement the features of the current iteration

➢ Have a relatively broad (but not too broad) domain model first, then refine details as your project progresses

- Work closely together with domain experts.
  - It’s your responsibility to understand them, not theirs to educate you!

In-Class Quiz #7: Domain Models vs. Design Models

- Complete the following sentences with **(1)** "…in domain models", **(2)** "…in design models", or **(3)** "…in domain and design models":

a) Classes such as `EventListener` and `ArrayList` can be found…
b) Failing to include an important aspect is more likely a risk…
c) Methods are usually not defined for classes…
d) Abstract classes can be used…
e) Discussions among developers revolve around aspects expressed…
f) Classes such as `Customer` and `Contract` can be found…
g) Modeling something in an inefficient way is more likely a risk…
h) Associations are usually unidirectional…