# MSO
# Analysis & UML 2

Hans Philippi
(based on the course slides of Wouter Swierstra)

August 30, 2018

# Recap & topics

- Last lecture: we have met with UML class diagrams
- Today: sequence diagrams & domain modeling

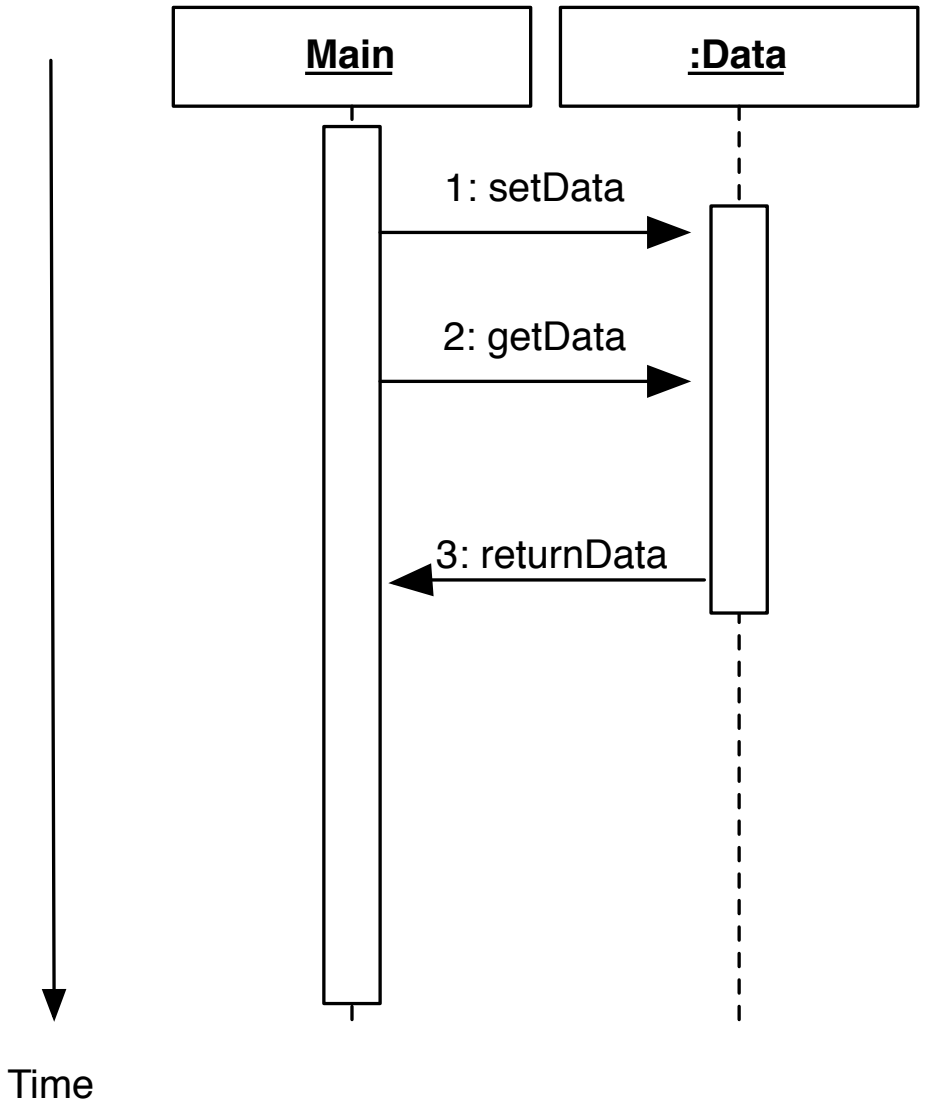
# Sequence diagrams

- A *class* diagram captures the static structure of your system (which classes are related how?) but does not say anything about the dynamic behaviour (what happens when the code is run?)
- A *sequence* diagram is another UML diagram that captures the dynamic control flow of your program

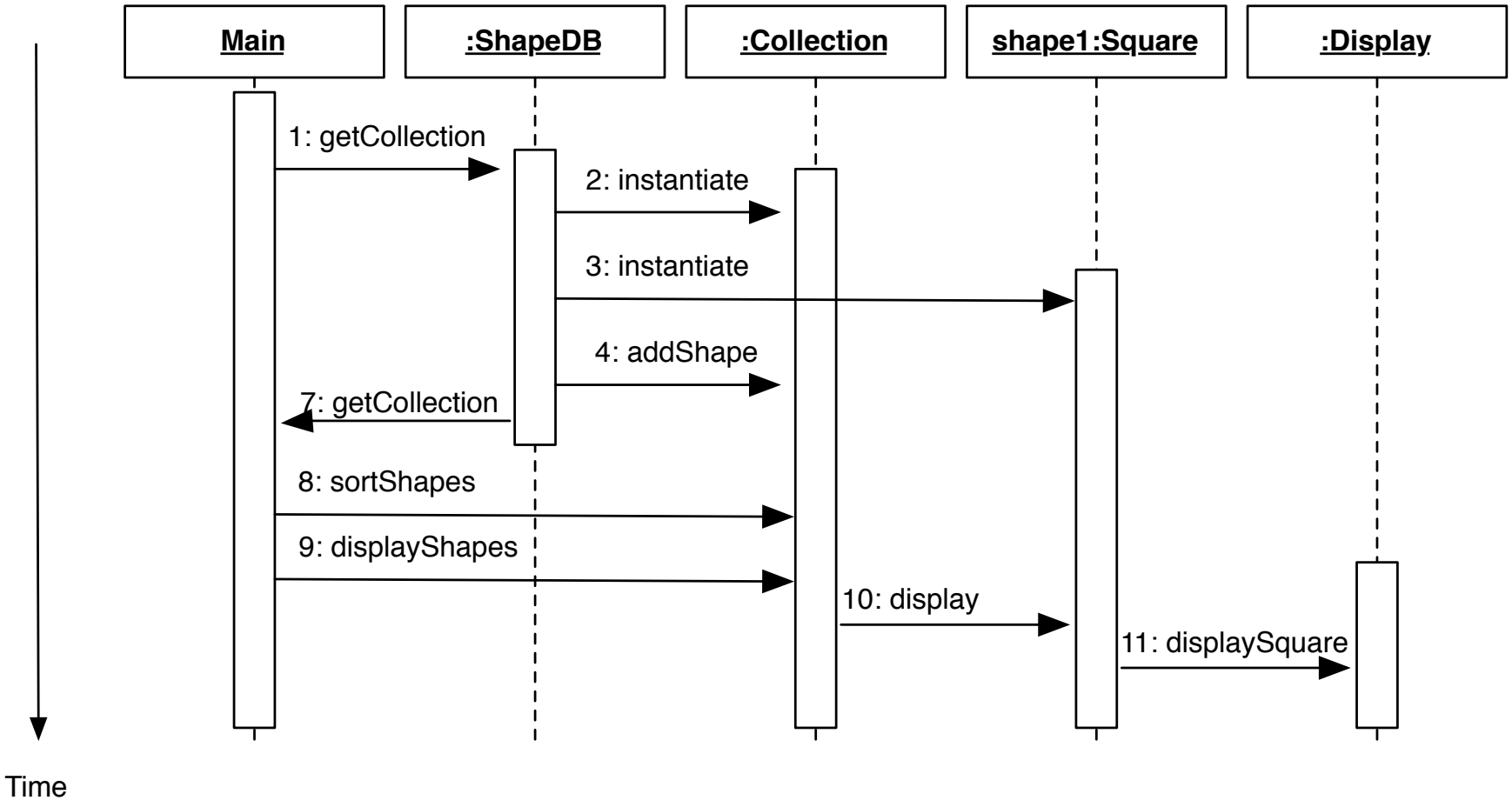
# Sequence – basics

- A sequence diagram consists of a number of columns, representing objects
- Horizontal arrows between objects represent method calls and returns
- Boxes indicate an objects lifetime – from creation to destruction
- Time flows downwards

# Sequence – example

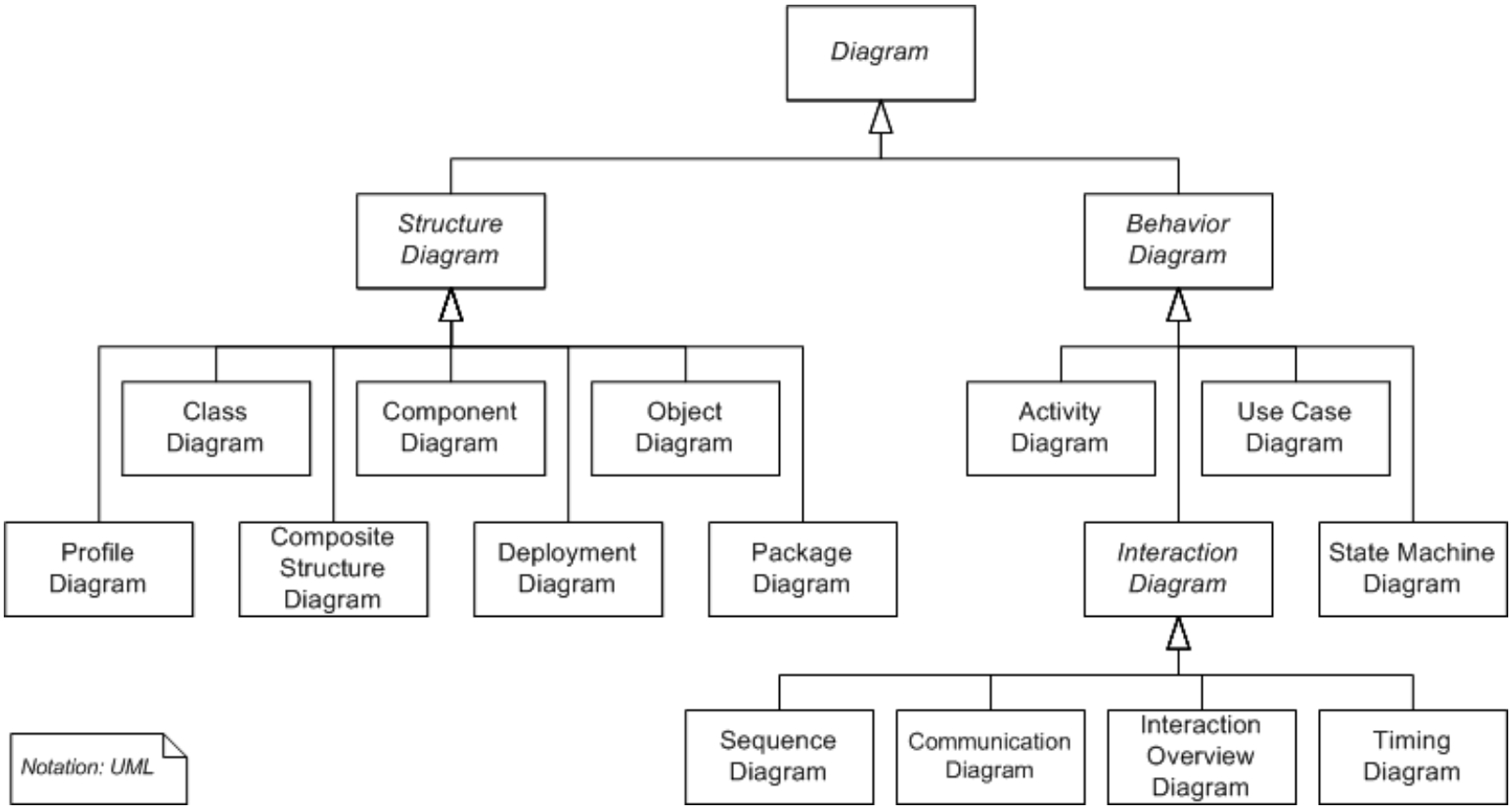


# Sequence – example



# Unified Modeling Language

There are many other flavours of the UML for modeling both static structure and dynamic behaviour of code.



# Analysis and design

- Suppose you have talked to all the stakeholders...
- and written various use cases...
- and agreed upon a set of requirements...
- How do I make a great *design*?

# Bridging the gap...

A requirements document:

- is written in English
- should be understood by the customer
- aims to establish the system requirements

A design:

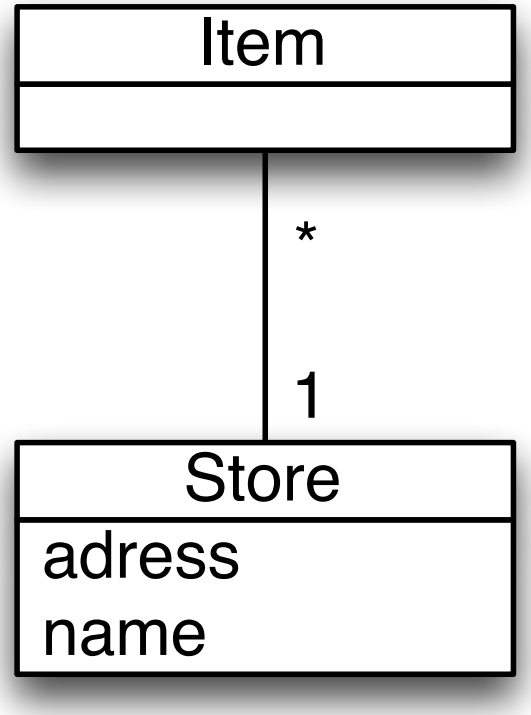
- is written using UML (class) diagrams
- should be understood by developers
- aims to communicate how to implement the system

# Analysis

Before fixing the design (or drawing UML class diagrams), it can be useful to define a domain model:

- specific to the problem domain
- captures users's point of view
- semi-formal notation – light-weight UML class diagrams without methods
- models the *real world* and not *software components*
- is very similar to *Entity-Relationship Modeling*

# Domain model – example



# Why write domain models?

- Domain models help you to understand the entities a customer works, and your software system will handle
- They help establish a common language with your customer
- They provide a *visual dictionary* describing the relevant concepts in your domain
- They are a first step towards defining the software system
- They are a first step towards designing a database scheme, if relevant

# Analysis – how to write a domain model?

- Talk to domain experts
- Identify existing solutions/components
- Identify *design patterns* (second half of this course)
- Textual analysis

# Textual analysis

- Once you start writing requirements and use cases, you quickly accumulate a lot of text
- Try to use this text to identify parts of the domain:
  - Nouns are good candidates for conceptual classes
  - Verbs are good candidates for associations

# Textual analysis taken (from Larman 2002)

1. Customer arrives at checkout with goods to purchase
2. Cashier starts a new sale
3. Cashier records item description, price, and running total for each item
4. System computes sales total, including tax
5. Cashier informs Customer of the total and processes payment
6. System logs completed sale, sends payment info to Accounting and Inventory departments
7. System prints a receipt
8. Happy Customer leaves with receipt and goods

# Textual analysis taken (from Larman 2002)

1. **Customer** arrives at **checkout** with **goods** to purchase
2. **Cashier** starts a new **sale**
3. **Cashier** records **item description**, **price**, and running **total** for each **item**
4. **System** computes sales **total**, including **tax**
5. **Cashier** informs **Customer** of the **total** and processes **payment**
6. **System** logs completed **sale**, sends payment info to **Accounting** and **Inventory departments**
7. **System** prints a **receipt**
8. Happy **Customer** leaves with **receipt** and **goods**

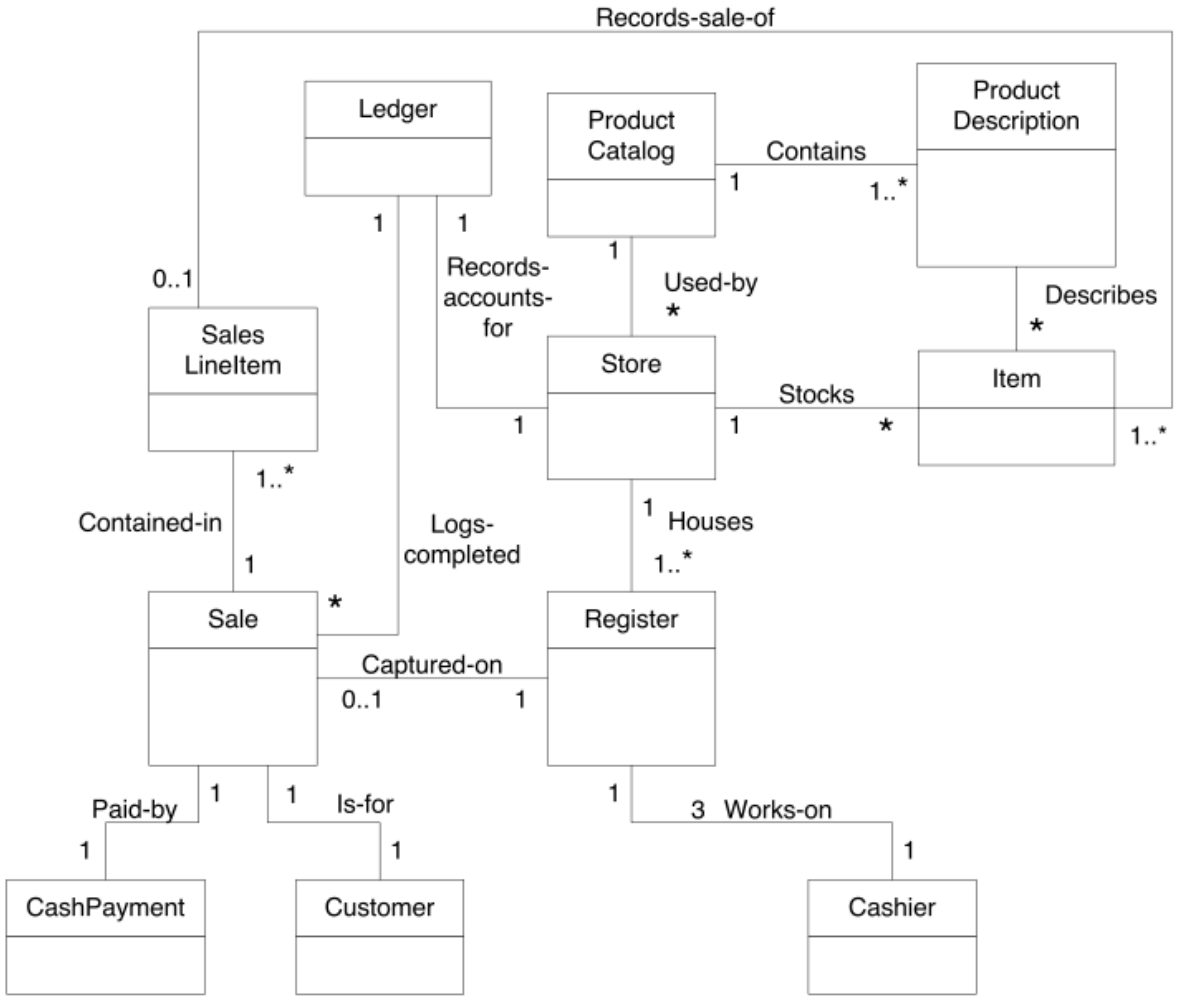
# Selecting conceptual classes

- Customer, Checkout, System, Receipt: should these be objects or not?
- What about Price, Total, Description? Are these objects or attributes?
- Are Items and Goods two words for the same concept?

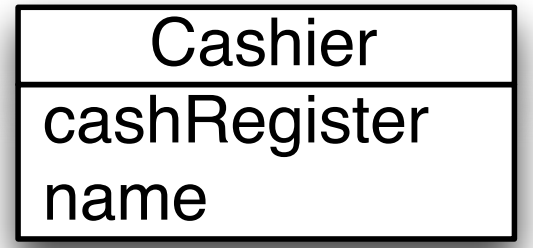
# Textual analysis – identify verbs

1. Customer *arrives* at checkout with goods to purchase
2. Cashier *starts* a new sale
3. Cashier *records* item description, price, and running total for each item
4. System *computes* sales total, including tax
5. Cashier *informs* Customer of the total and processes payment
6. System *logs* completed sale, *sends* payment info to Accounting and Inventory departments
7. System *prints* a receipt
8. Happy Customer *leaves* with receipt and goods

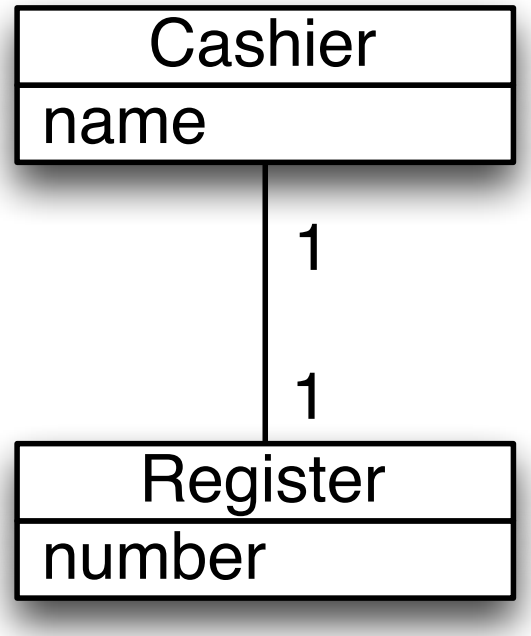
# Example – (from Larman, Ch 11)



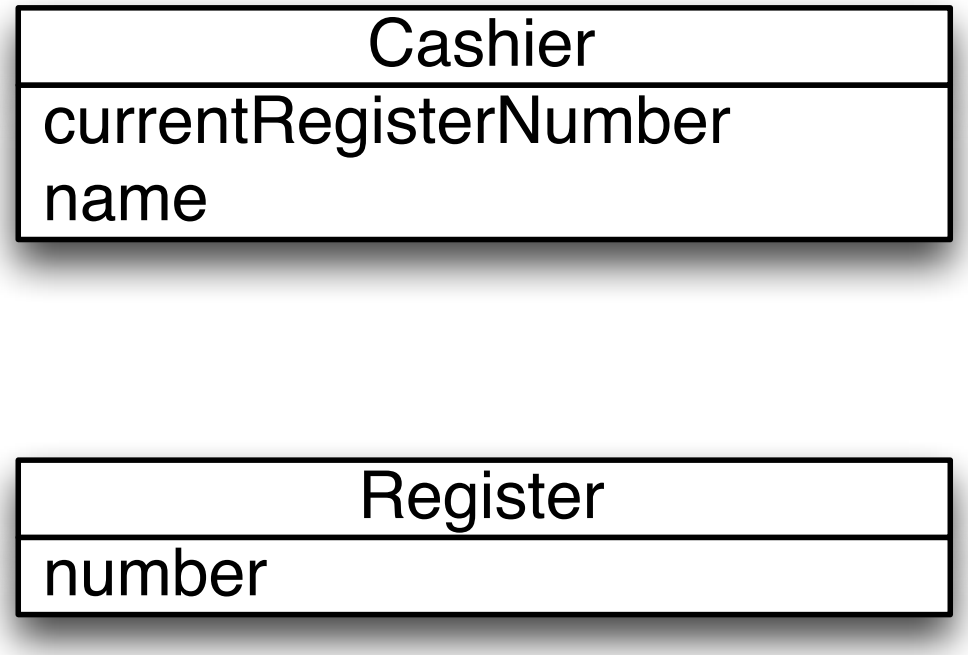
# A few common pitfalls in domain modeling

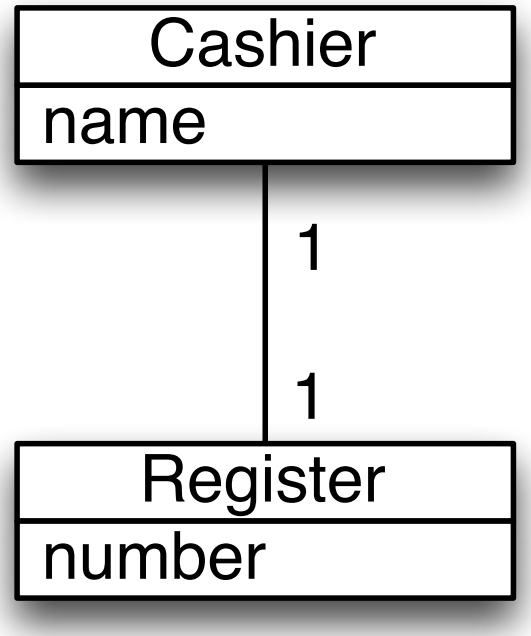



But a cash register might be related to other objects. . .




*Tip:* keep the types of attributes simple


Cashier
currentRegisterNumber
name
Register
number




*Tip:* favour associations over keys

# The Noun-Verb Analysis

The idea is very simple and easy to execute, but:

- Natural language is very imprecise
- Many more nouns than relevant classes (more than one name for the same concept)
- The design is determined by your use cases – but these two documents serve very different purposes

Don't be afraid to rephrase your use cases!

# Case study: elevators

- The elevator has one button for each floor
  - Light up when pressed
  - Requests that the elevator stops at that floor
  - The button's light is turned off once the elevator has stopped at the corresponding floor
- Each floor has two buttons to request the elevator to go up or down (except the first floor and top floor, which only have one):
  - Light up when pressed
  - Requests that the elevator stops at that floor
  - The button's light is turned off is once the elevator has arrived at the floor.

(Taken from Peter Müller's course on Software Engineering)

# Use case: FetchElevator

Main success scenario:

1. Passenger pushes button in the hall
2. System lights up button
3. System closes elevator doors
4. System moves elevator to requested floor
5. System turns off button's light
6. System opens elevator doors

# Use case: RideElevator

Main success scenario:

1. Passenger pushes elevator button
2. System lights up button
3. System closes elevator doors
4. System moves elevator to requested floor
5. System turns off button's light
6. System opens elevator doors

# Homework Exercise

Can you come up with a domain model?

What sequence diagrams correspond to these use cases?

# Validation

After developing your domain model, you still need to *validate* it:

- *Correctness:* Does it accurately model reality?
- *Completeness:* Is every scenario (including the exceptional cases) described?
- *Consistency:* Does the model contradict itself?
- *Unambiguous:* Does the model describe one reality (and not many)?
- *Realistic:* Can the model be implemented?

# How to validate?

You may use diagrams with different purposes (use cases, classes, sequence diagrams, etc.) to describe the same system.

- Do all these diagrams agree?
- Take care of homonyms and synonyms
- Is every class described exactly once?
- No dangling associations?

# Material covered

- Design Patterns explained: chapter 2
- Applying UML and patterns: chapter 10–13 and chapter 15