CLASS DIAGRAMS
LEVEL OF ABSTRACTION

UML Language Elements

Abstraction

- Domain
- System
- Implementation

UML LANGUAGE ELEMENTS

DEVELOPMENT = DESIGN + IMPLEMENTATION

development: converting the system specification into an executable system

Traditionally broken down into several stages:

- architectural design
- interface design
- abstract specification
- coding
- component design
- data structure design
- algorithm design
- debugging

- development is an iterative process with feedback between the stages
- design and implementation are typically interleaved

DESIGN VS. MODELING

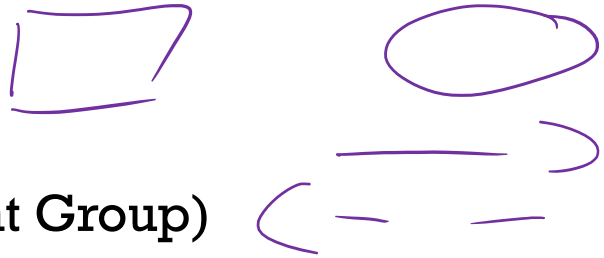
Design is the process of deciding how the requirements should be implemented.

- guided by ***design principles***
- part of ***development***

Modelling is the process of creating an abstract representation of the domain or the system.

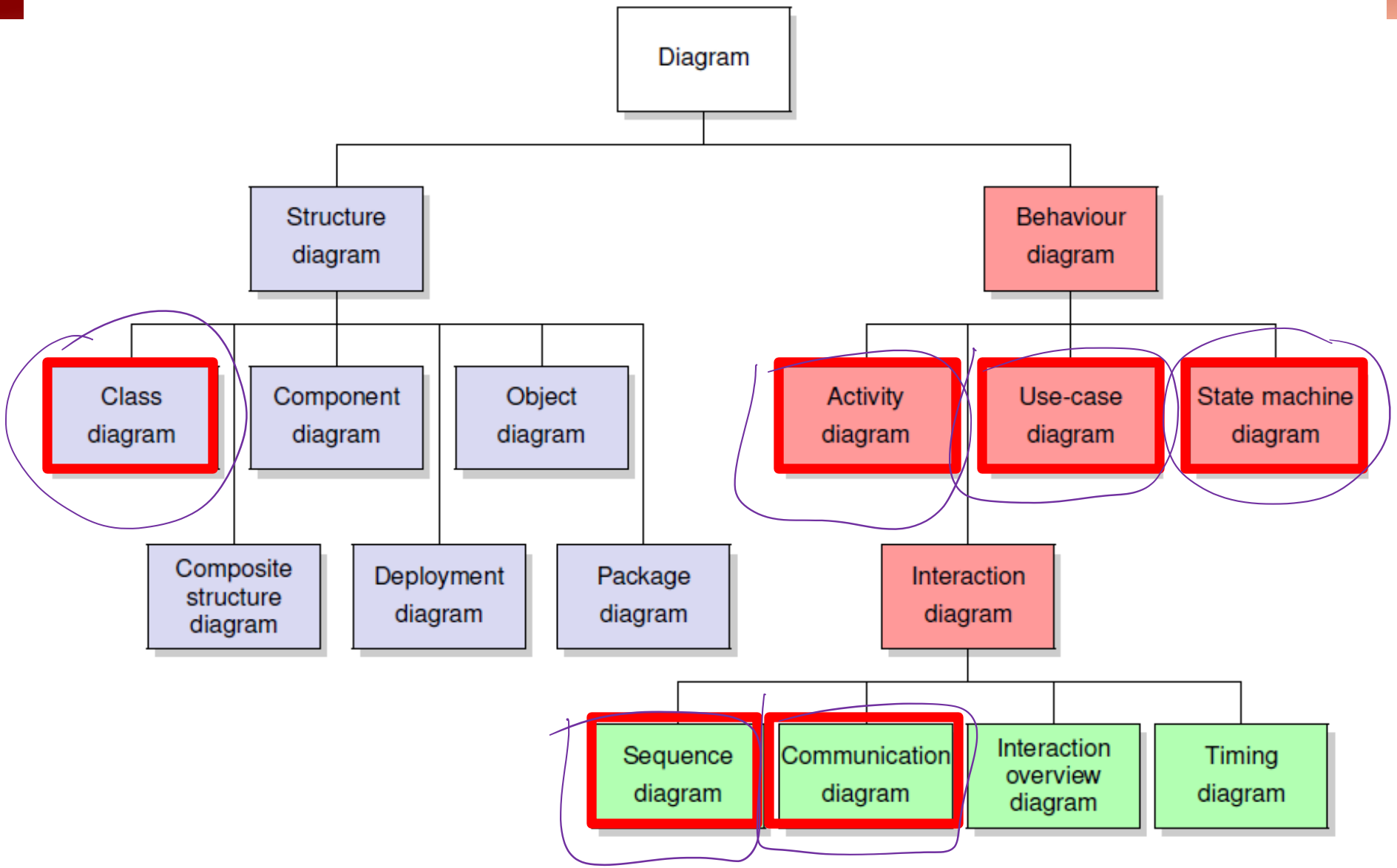
- uses ***modelling languages*** UML
- spans ***requirements*** and ***development***

UML (Unified Modeling Language)



- **graphical** modelling language
  - standardized by OMG (Object Management Group)
  - semi-formal
  - variety of different diagram types
- supports **object-oriented** designs
  - but no fixed methodology
- **unified**: each diagram gives a different view on the same system
- developed by Rumbaugh, Booch, Jacobson et al.
  - starting early-90's, unification mid-90's

UML DIAGRAM TYPES



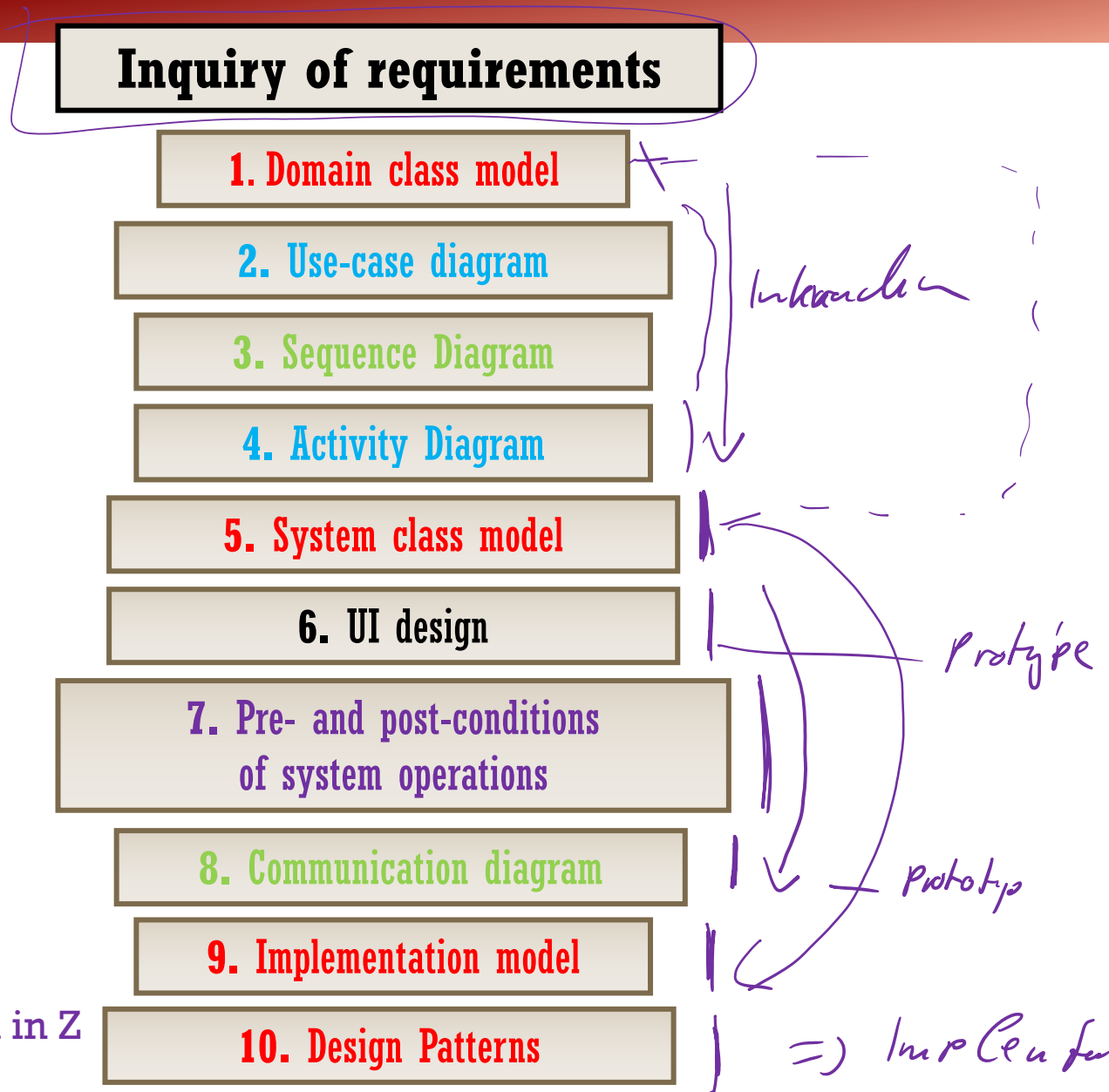
STRUCTURAL VS. BEHAVIORAL MODELING

- **System** = **structure** + **behavior**
- **structural models** show the system's organization in terms of its ***components*** and their ***relationships***
  - can be ***static*** (classes) or ***dynamic*** (threads)
- **behavioral models** show the system's ***dynamic*** as it is executing and responding to stimuli
  - can be ***events*** or ***data***

WHY UML DESIGN

- Design can have different levels of abstraction
- Allows talking with different domain experts (team, manager, client, etc.)
- Allows abstraction from programming language
- Allows to see the big picture

Overview of process we will use



Structural Models
Interaction Models
Behavioral Models
Formal Specification in Z

CLASS DIAGRAM ABSTRACTION

Domain Class model

System class model

Implementation model

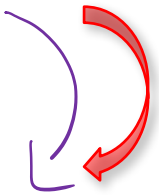
UML Conceptual Class Stereotypes

- UML classes don't have to be classes in a coding language
  - May abstract a potentially large portion of the system
  - Perspective required when interpreting diagram

- Different levels of abstraction
  - *Conceptual/Domain Classes* represent abstractions from the problem space
  - *Physical Classes* represent solution elements including technologies

DIFFERENT LEVELS OF DETAIL FOR CLASS DIAGRAMS

- **domain model**
  - developed during domain analysis to understand the domain
  - aspects of the domain that will not be implemented by the system are modeled, too
  - also called ***exploratory domain model***



shrink & refine

- **system class model**
  - only aspects of the domain that are implemented by the system are modeled

grow & refine

- **implementation model**
  - represents system model using programming language constructs

DOMAIN CLASS MODELS

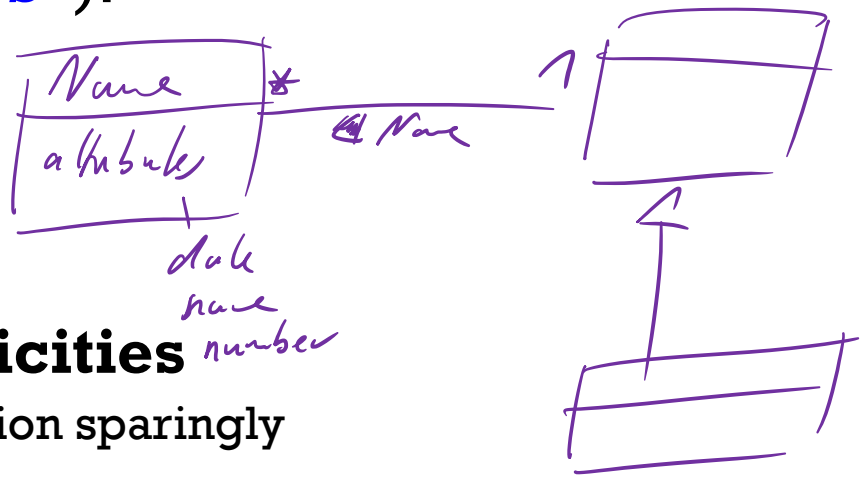
DOMAIN CLASS MODEL

The **domain class model** contains:

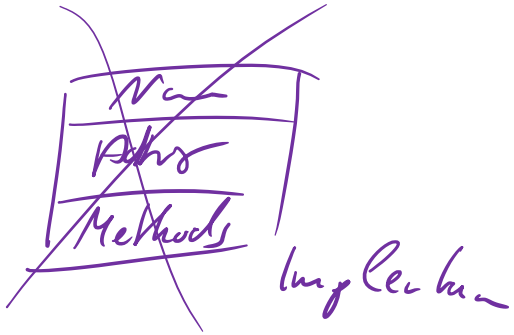
- relevant entities as **classes**:
  - physical objects
  - persons, organizations (**actors**)
  - events, processes, abstractions
- relationships between entities as **associations**:
  - relations
  - communications
  - part/whole relations (**aggregation**, **composition**)

DOMAIN CLASS MODEL

The domain class model should only use a limited set of notations ("***boxes and arrows***"):



- **classes**
  - use attributes sparingly
- **associations** with **multiplicities**
  - use aggregation and composition sparingly
- **generalization**



It should **not** contain:

- methods
- types
- roles

A RECIPE TO COOK DOMAIN CLASS MODELS

1. add classes (***without*** attributes)
   - identify relevant real system objects and represent them as classes
2. add generalizations
3. add associations (***with*** multiplicities)
   - identify relations between identifed objects
4. add aggregations and compositions
   - check whether or not they really necessary
5. add attributes (***no*** operations)
   - identify relevant core attributes and add them to the corresponding classes
6. stir until done

boxes

arrows

DISCOVERING DOMAIN CLASSES

Noun phrase analysis:

- analyse (textual) documents describing the system
  - requirements documents
- extract the **nouns** and **noun phrases**
- eliminate nouns that
  - are redundant, vague, or highly general
  - are too specific or represent specific instances
  - refer to the entire system or objects outside the application
- pay attention to nouns that describe different users
- types or other actors

EXTRACT: NOTES FROM MEETING

Notes from the first requirements elicitation meeting with the shop owners and employees

- system should help to keep track of currently available products in shop
- the system should support selling products to its customers
- system should support ordering articles from warehouse, e.g. if they are sold out
- system should be able to handle customer data, e.g. add/delete customers
- each employee has to login before working with the system
- the employees of the shop should handle the customers' purchases
- customers do not interact with the system directly but are served by employees
- each product has a unique ID and is available in a certain quantity in the shop
- customers can request products in a certain quantity through an employee
  - If the product is available in the requested quantity the product is sold and the available quantity of the product is decreased correspondingly
  - If the product is not available in the requested quantity a purchase order is created for the customer and sent to the warehouse. For each customer the number of purchase orders is restricted to five.

EXAMPLE: SHOP

Purchase order

Customer

Warehouse

Shop

Employee

Warehouse Administration

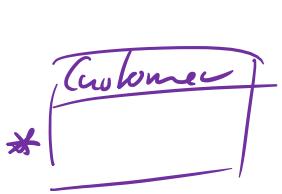
ProductType

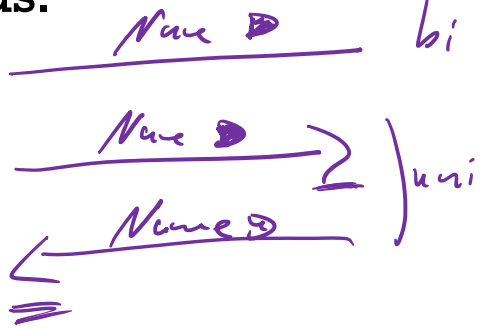
DISCOVERING ASSOCIATIONS

- Start with central and most important classes and
- Work outwards towards the less important classes.
- Add an association if one class
  - ***possesses*** or ***controls,***
  - ***is related to,***
  - ***communicates with,***
  - ***is a part of,***
  - ***or is a member of***

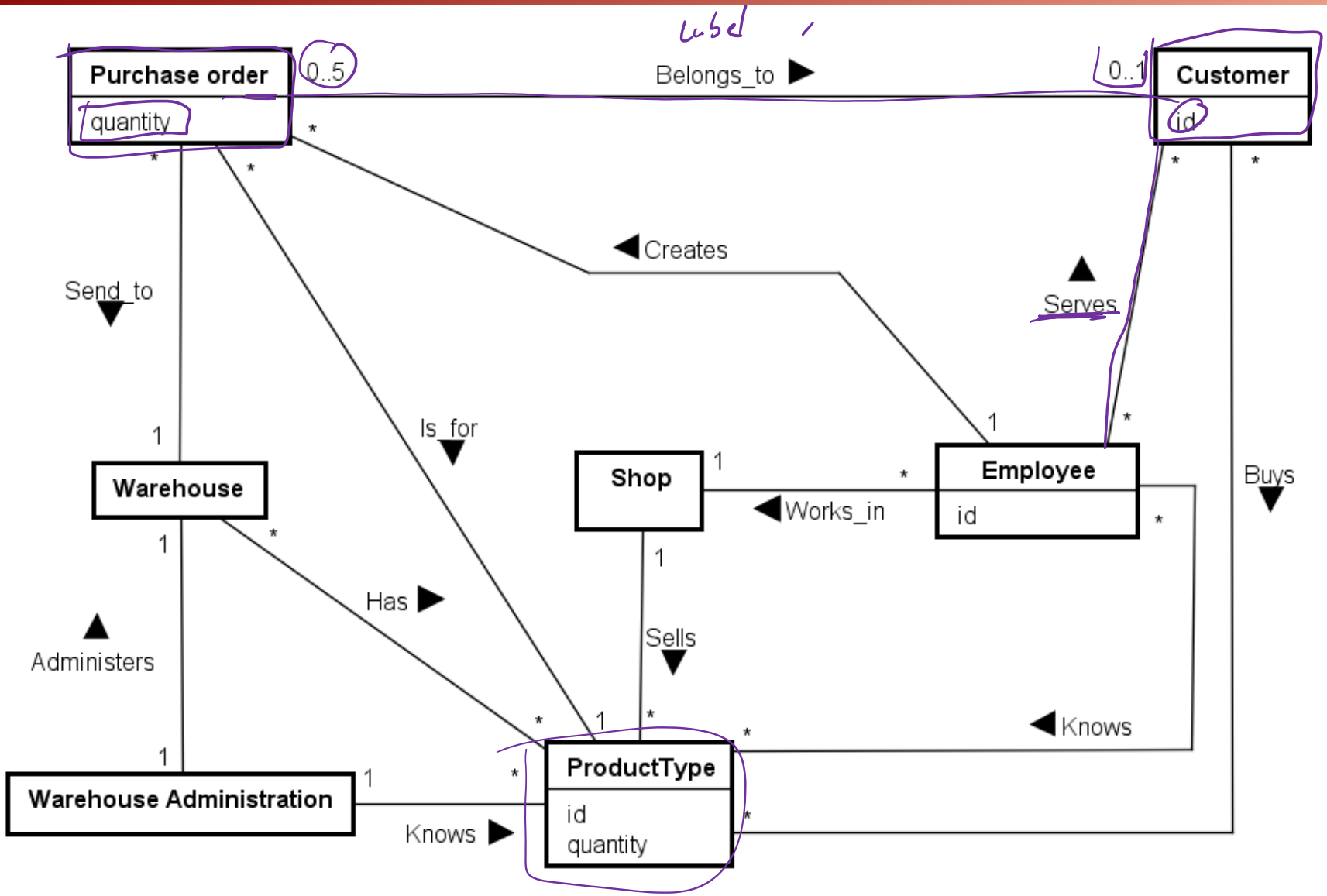
  some other class in the model.
- Label it clearly and specify the multiplicity at both ends.
- KISS: Keep it simple

Don't rely on verb phrases – associations are often left implicit!





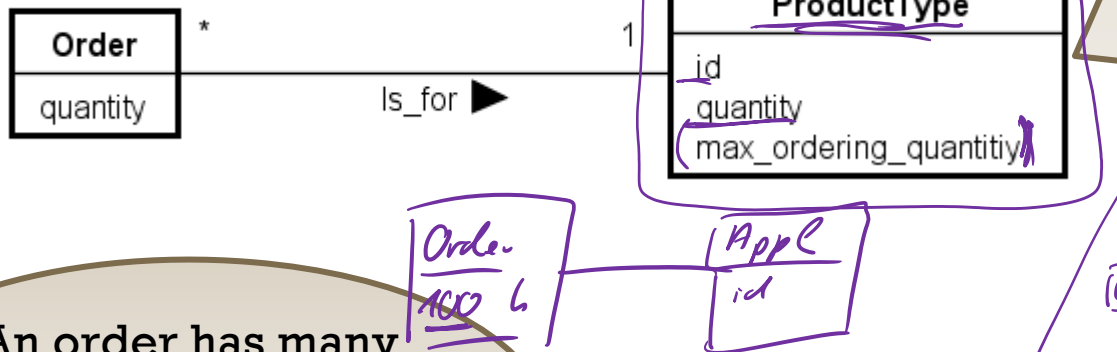
EXAMPLE: SHOP



PITFALLS - WRONG MULTIPLICITIES

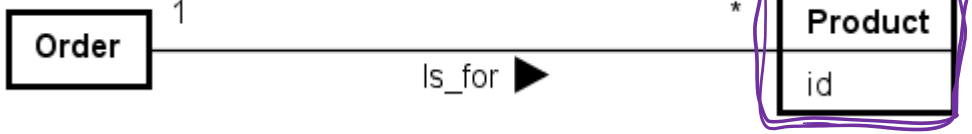
Sanity-check the multiplicities with a few questions:

- Do we model generic / indistinguishable
- or individual / distinguishable items?



Is a ProductType! An order has a quantity to model how many items of the ProductType are ordered

An order has many products, since each product in an individual product.

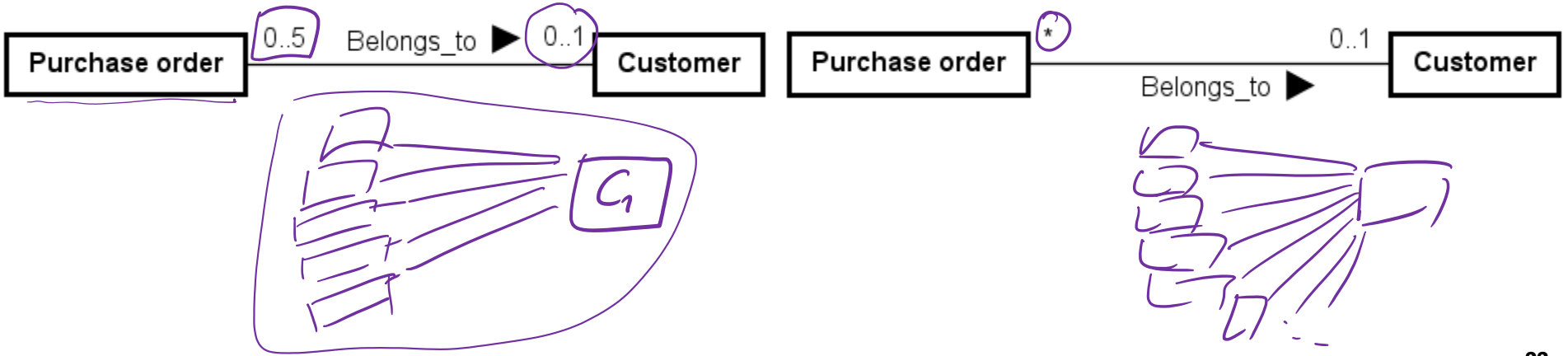


PITFALLS - WRONG MULTIPLICITIES

Sanity-check the multiplicities with a few questions:

- Do we model a static view (*snapshot*)
- or a dynamic view (*history*)?

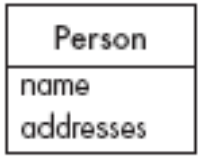
orders at a time vs. *all orders for a customer*



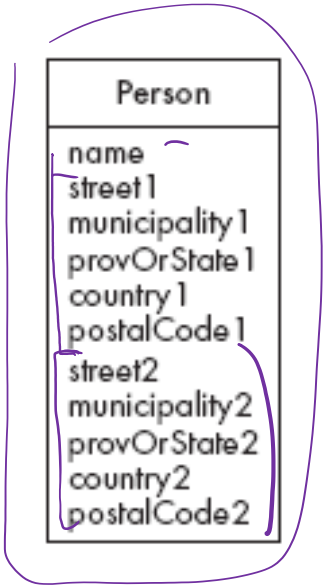
DISCOVERING ATTRIBUTES

- Information that must be maintained in each class
  - nouns which are rejected as classes may become attributes
- Attributes should generally contain a simple value
  - string, number, date, etc.
- If a subset of a class's attributes form a coherent group, then create a new class from these attributes

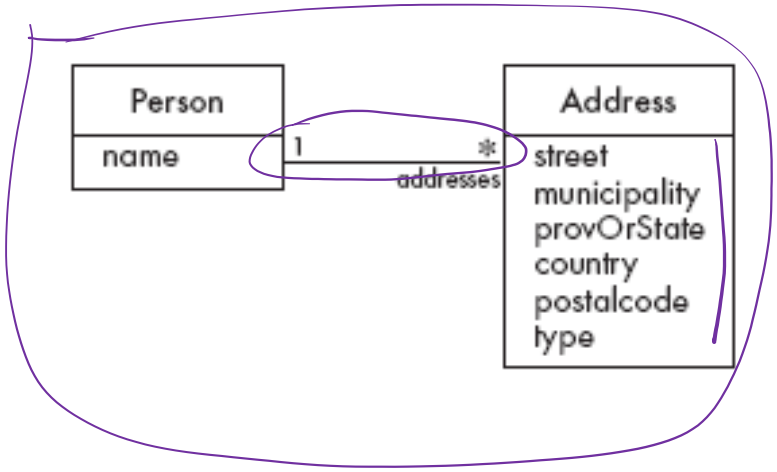
PITFALLS - REPEATED ATTRIBUTES



Bad, due to a plural attribute



Bad, due to too many attributes, and the inability to add more addresses



Good solution. The type indicates whether it is a home address, business address etc.

Source: Lethbridge/Laganiere, Object-Oriented Software Engineering

SUMMARY

What this diagram is for:

- This diagram is supposed to give an overview of all the real world objects that have something to do with the system and their relation. Everything important for the system should be included.
- Classes are still real world objects here (not classes that need to be implemented)
- All relevant things are classes in the diagram
- Relation between things is specified as associations (aggregation, composition as well)
- Generalization is also possible
- Model includes:
  - Association names with reading direction
  - Class names (without stereotypes)
  - Multiplicities
  - Basic Attributes (numbers, dates, names usually without types)
- A class can be a person, an object, a dataset etc.

SYSTEM CLASS MODELS

DEVELOPMENT OF THE SYSTEM CLASS MODEL

- determine the **system boundary**
  - **actors** are **outside** the system
- determine the **system layers**
  - **presentation layer**
  - **application layer** (middle layer, business logic)
  - **data layer**
- use **UML stereotypes** to denote which layer a class belongs to

Derive the system class model
systematically
from the **domain class model**.

<<BOUNDARY>> CLASSES

Boundary classes are the interface between system and environment:

- presentation layer (GUI)
- interfaces to other systems
- sensors and switches to control external devices

⇒ actors communicate with the system *only* via boundary classes

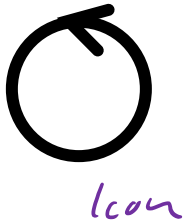
Design rule: one boundary class per actor.

<<CONTROL>> CLASSES

Control classes administrate the system operations:

- application layer
- encapsulate business processes and business logic

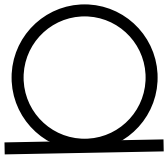
⇒ "glue" between boundary and entity classes



Design rule: one control class per use case.

<<ENTITY>> CLASSES

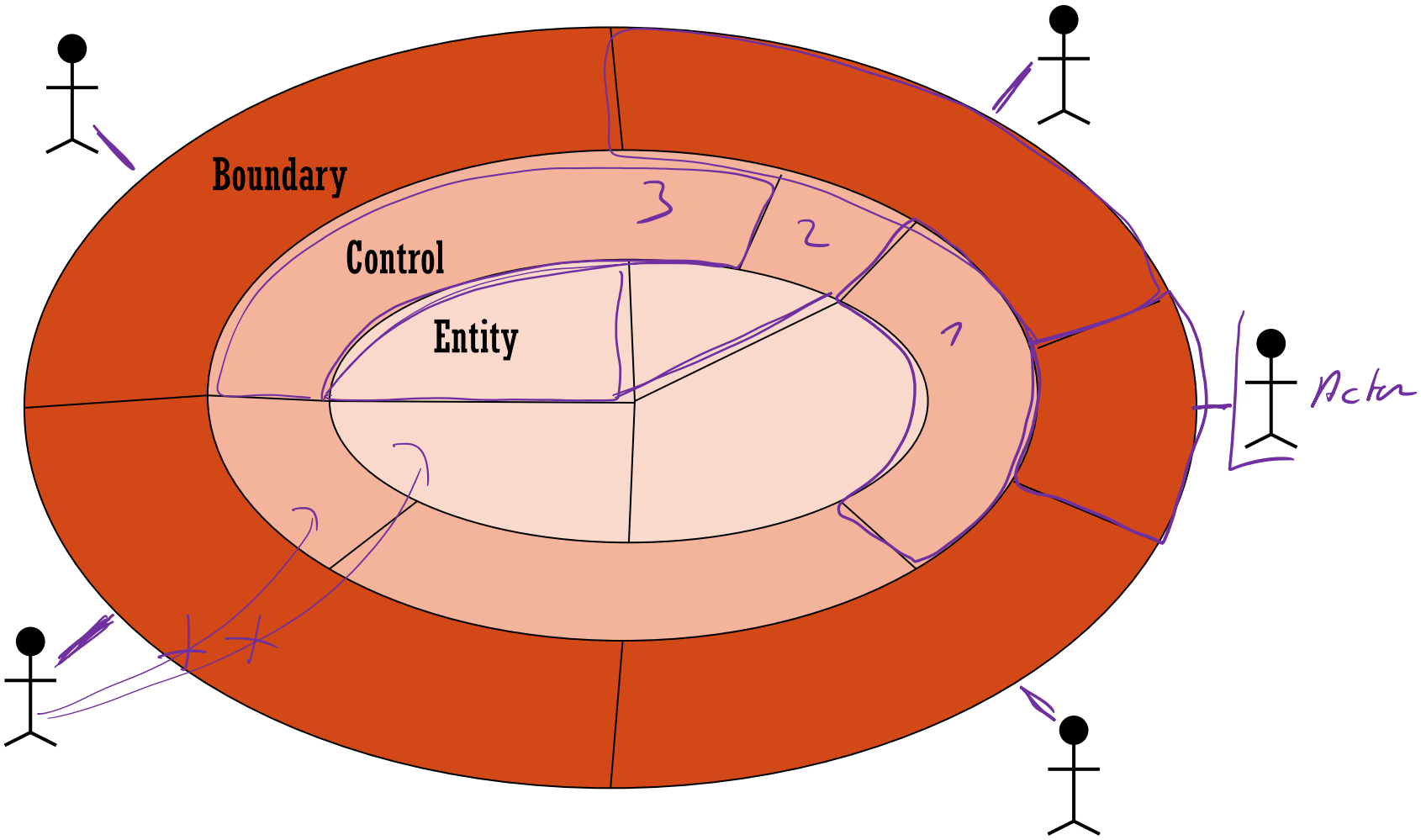
Entity classes operate the application data and manipulate the internal system state:



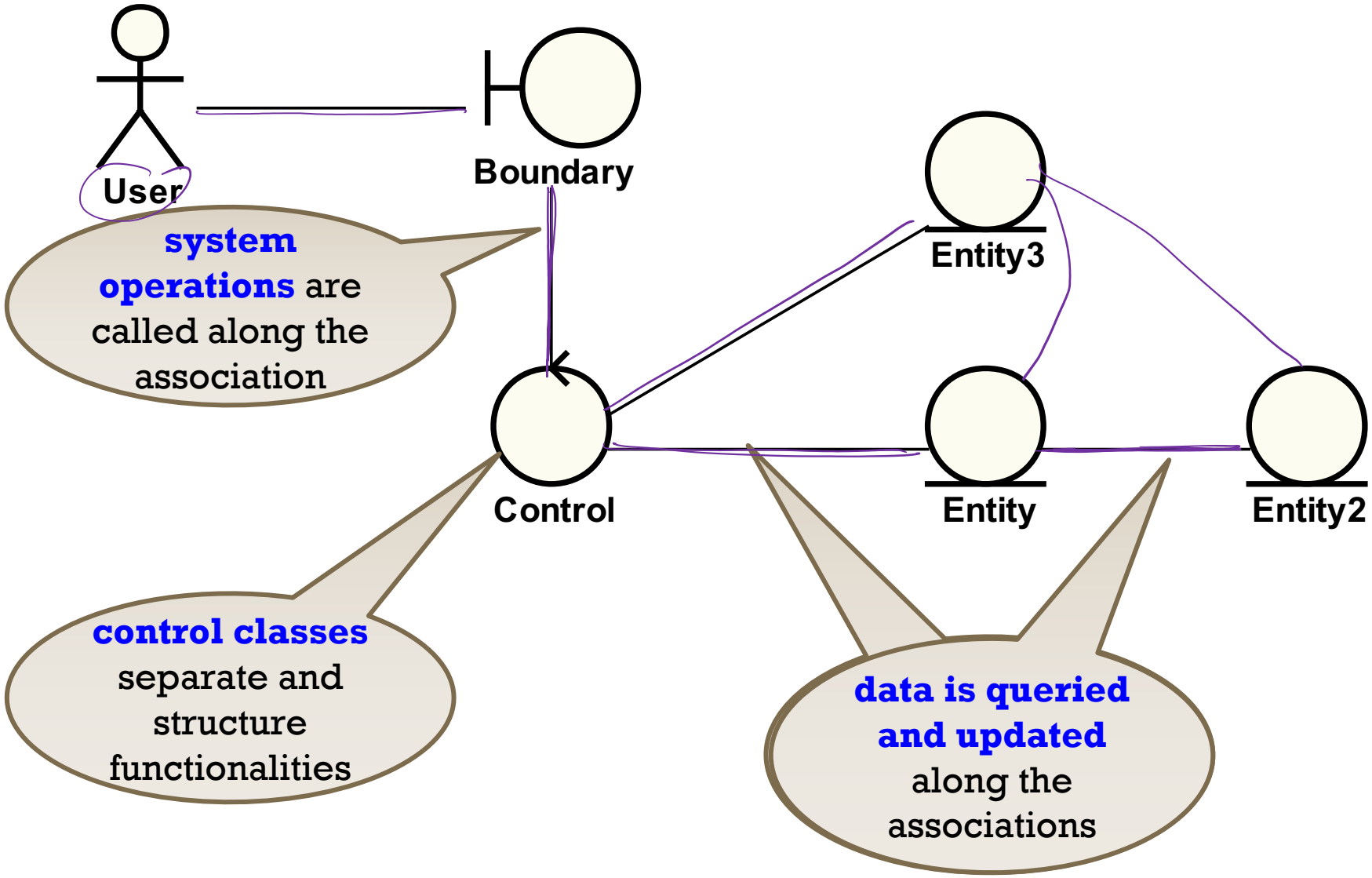
- data layer (persistence, database system)
- includes access methods
- If data for actors are needed, they must be mirrored within the system via additional entity classes

⇒ connected to control and other entity classes via associations

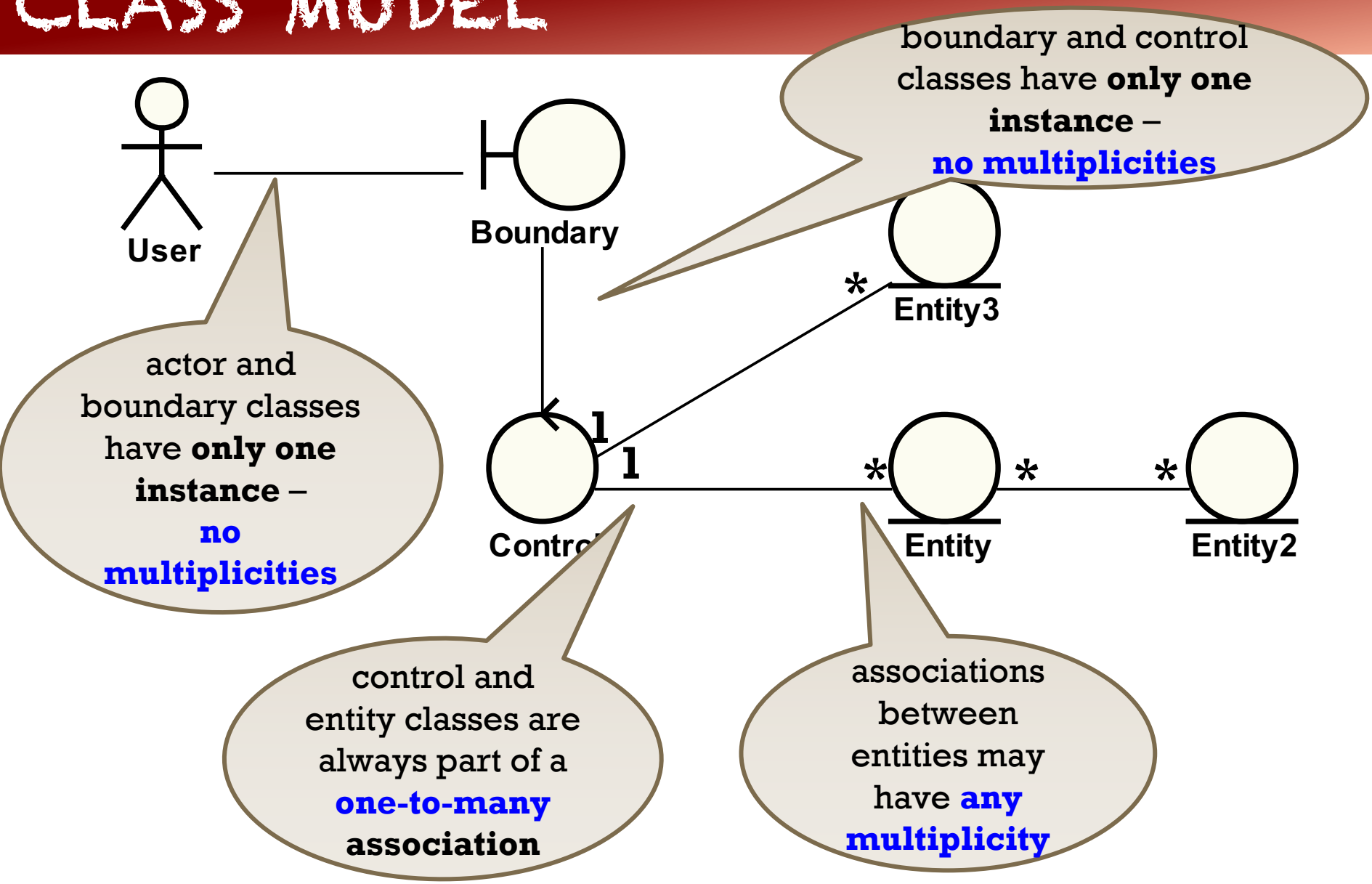
LAYERED ARCHITECTURE



ASSOCIATIONS IN THE SYSTEM CLASS MODEL



ASSOCIATIONS IN THE SYSTEM CLASS MODEL



A RECIPE TO COOK SYSTEM CLASS MODELS...

1. start with the domain class model
2. identify **actors**
   - check in the use case diagrams
3. identify **boundary classes** for **actors**
   - Identify and represent the user interface
4. identify **control classes** for **use cases**
   - between boundary and entity classes
   - typically one control class per use case
5. insert entity classes for actors (if required)
   - reflect necessary properties of the actors in the system
6. identify entity classes
   - Model known properties as attributes
   - ensure **1:1 associations** between actor/boundary and boundary/control classes
   - ensure that actors only talk to boundary classes
7. check model for **completeness**
   - insert new associations (if necessary)
   - model might differ structurally from domain class model
8. Add the known **attributes** in all classes
   - If necessary describe classes and their attributes separately

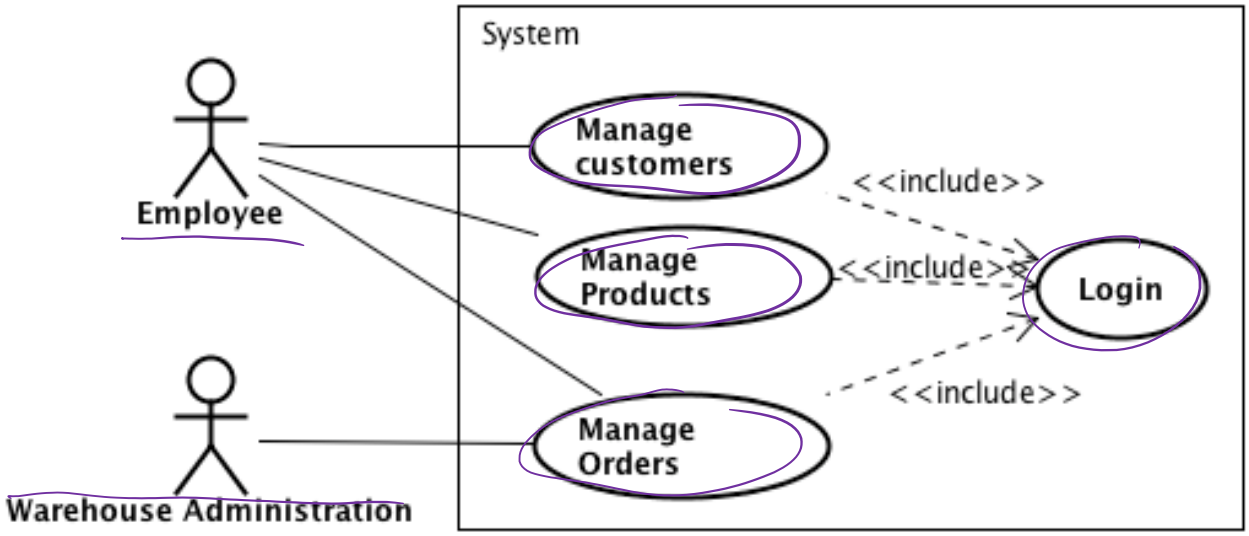
EXCURSE: USE CASE DIAGRAM

- First step to describe the complete behavior of the system (Use-Cases)
  - Actors
    - occur in the domain class model
    - are **not a part** of the system
    - interact with the system
    - represent **user roles**
  - Use-cases
    - describe the set of functions
    - describe interactions of actors with the system (scenarios)
      - are activated by an actor (system operation)
      - the system reacts with an observable event (system event)

SHOP: POSSIBLE SCENARIOS

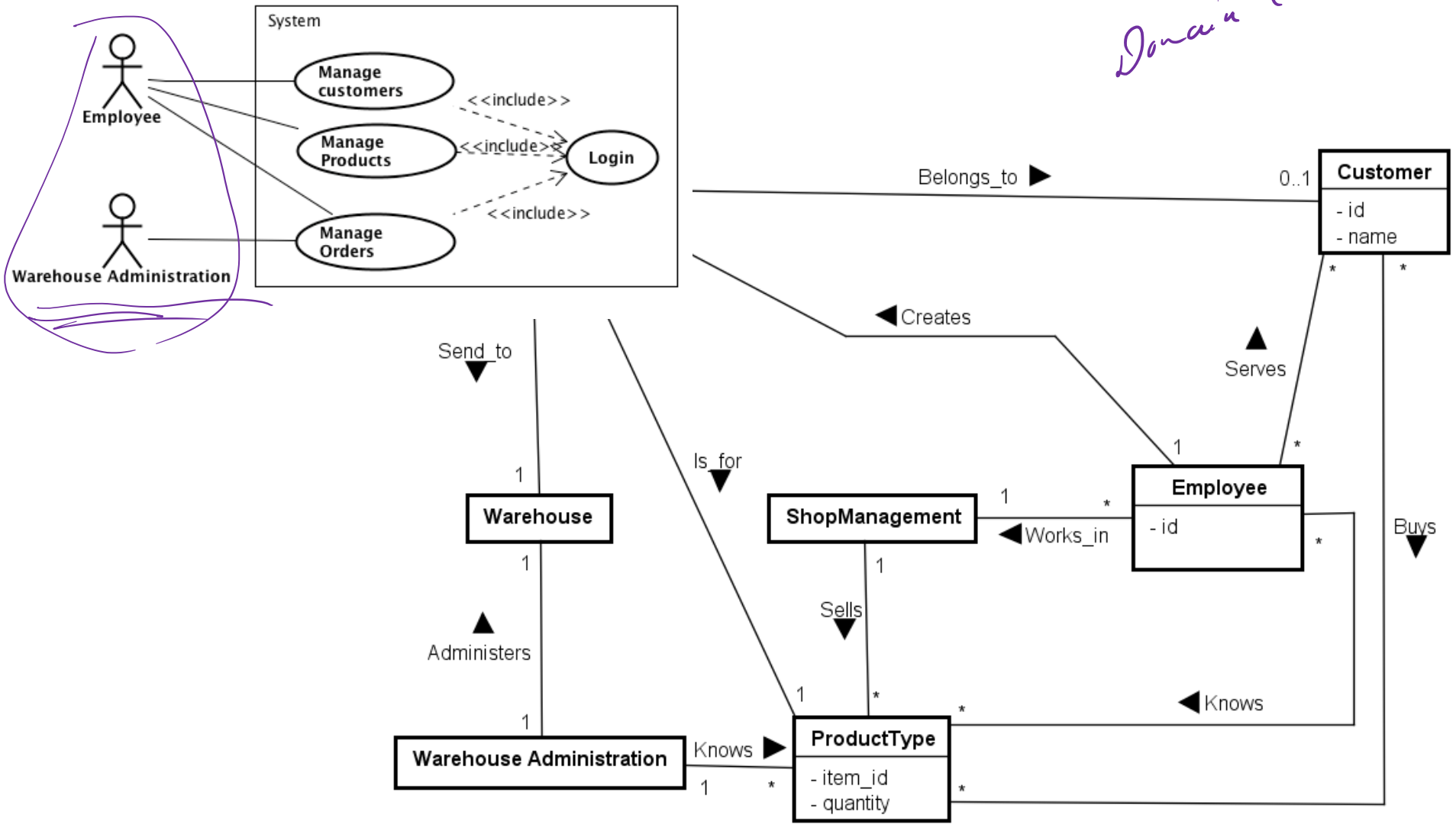
- Add customers
- Delete customer
- Sell products
- Create purchase order
- Login
- List products
- Delete order

Scenarios



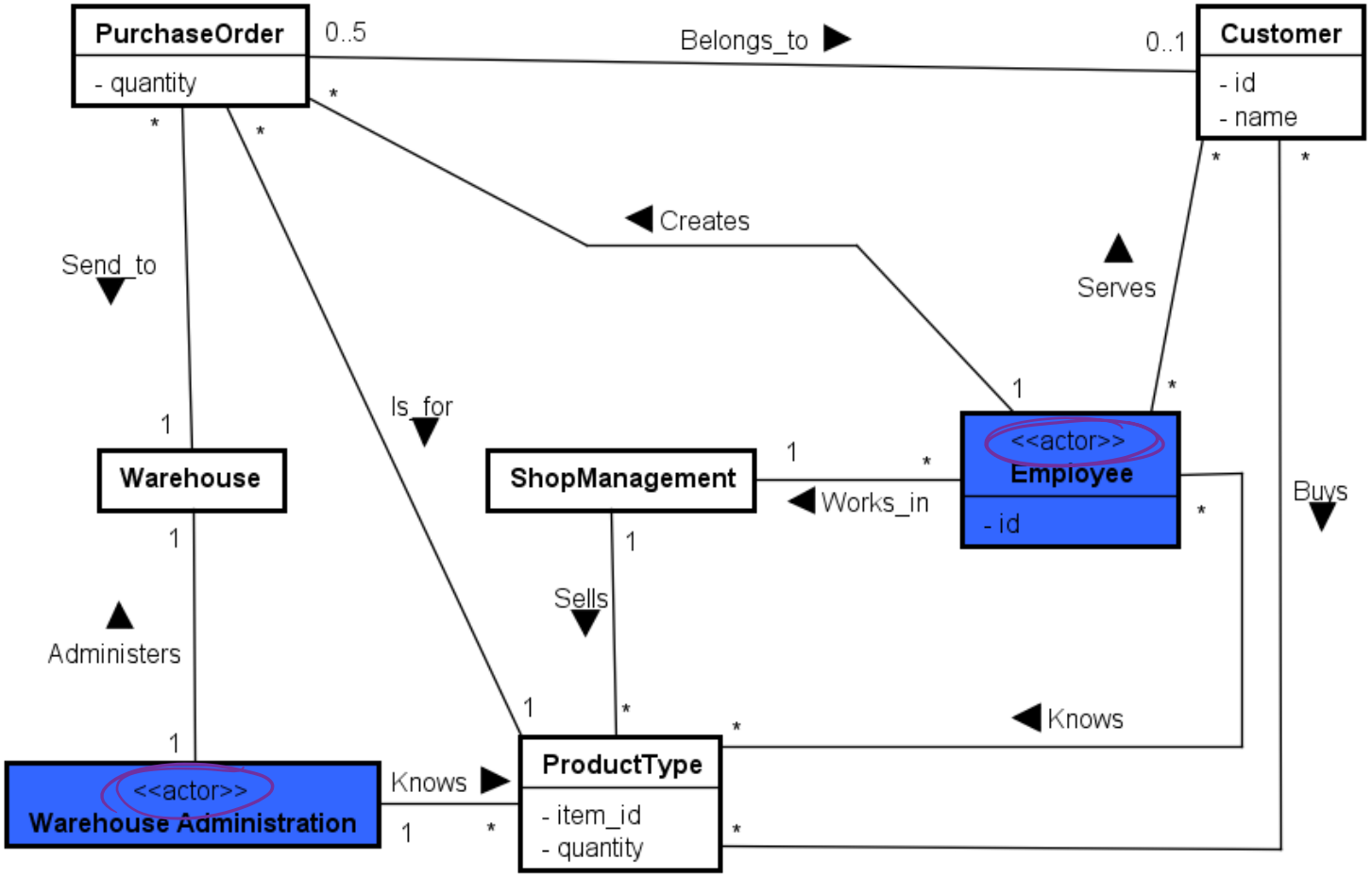
EXAMPLE: ACTORS

Check the use case diagrams for actors!

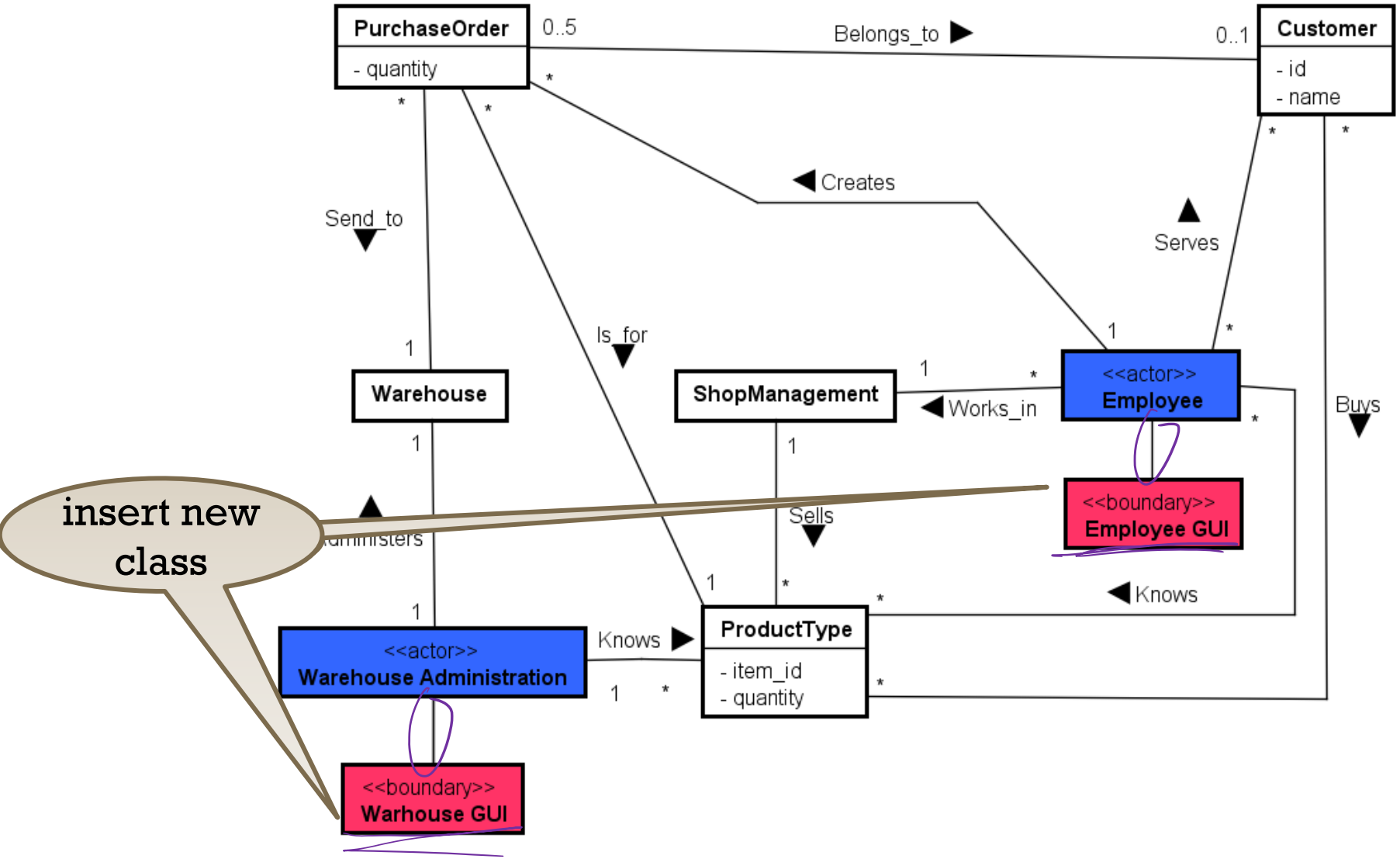


EXAMPLE: BOUNDARY

Insert a new boundary class for each actor, or change an existing class!



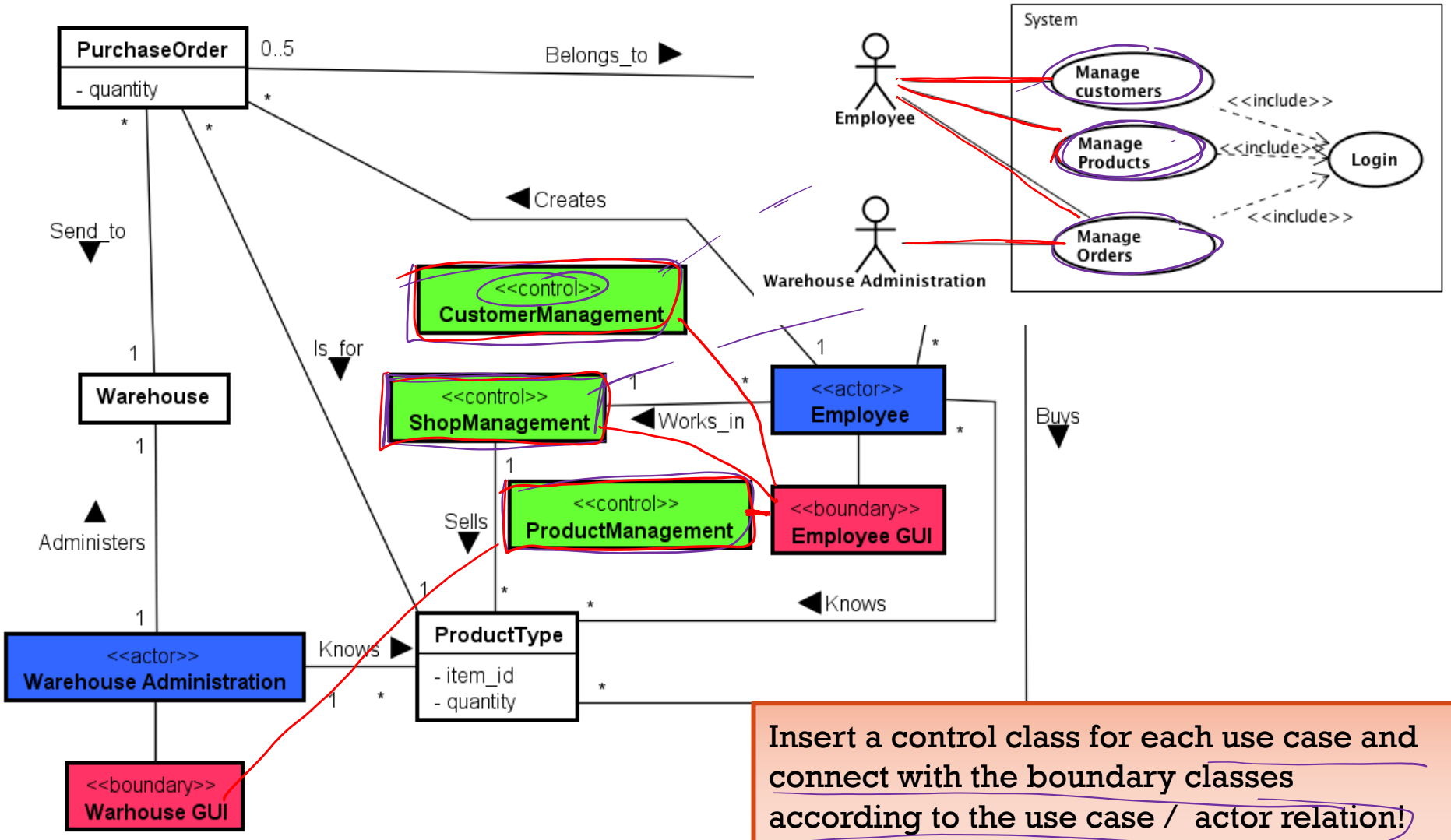
EXAMPLE: BOUNDARY CLASSES FOR ACTORS



A RECIPE TO COOK SYSTEM CLASS MODELS...

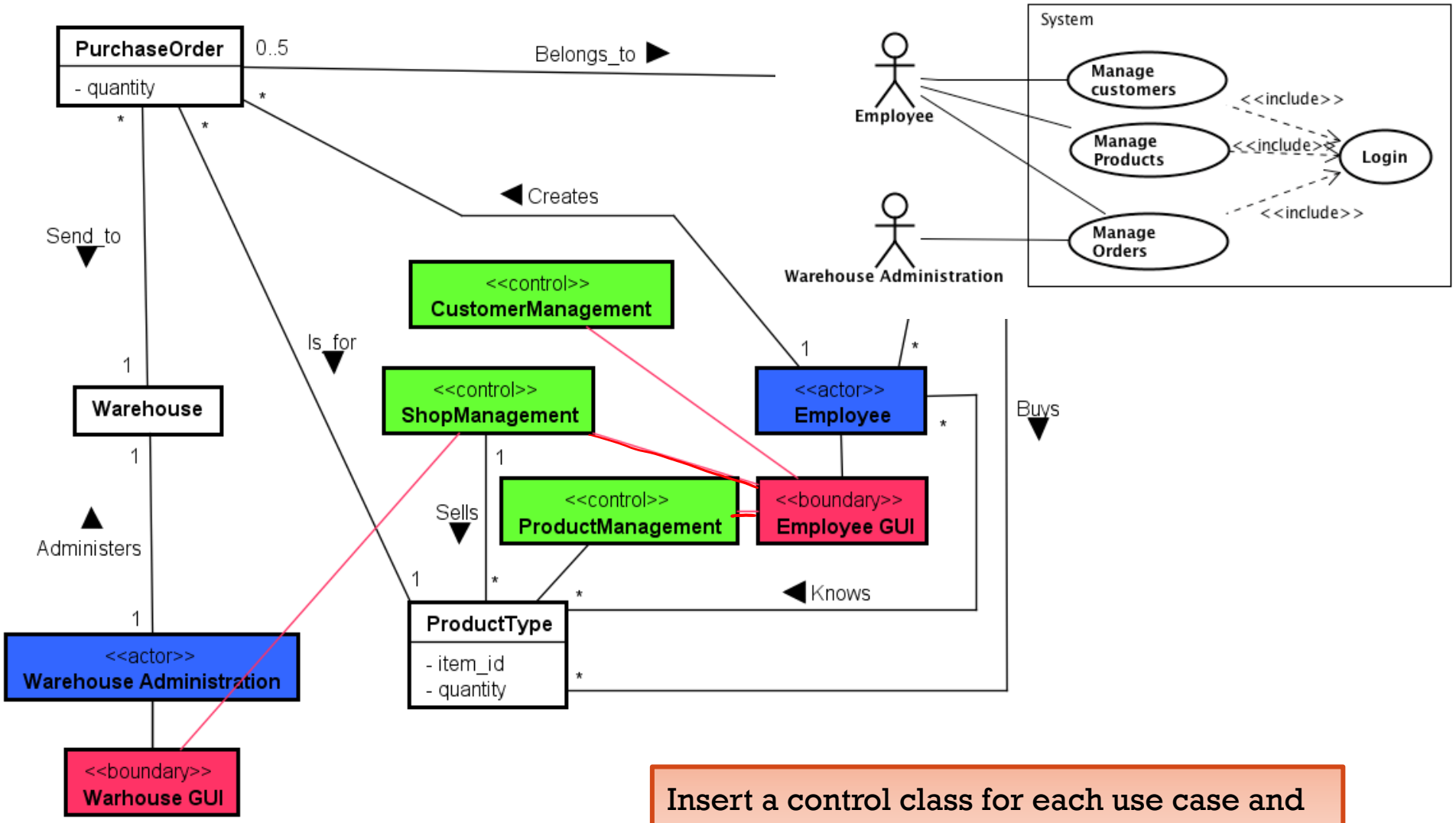
1. start with the domain class model
2. identify **actors**
   - check in the use case diagrams
3. identify **boundary classes** for **actors**
   - Identify and represent the user interface
4. identify **control classes** for **use cases**
   - between boundary and entity classes
   - typically one control class per use case
5. insert entity classes for actors (if required)
   - reflect necessary properties of the actors in the system
6. identify entity classes
   - Model known properties as attributes
   - ensure **1:1 associations** between actor/boundary and boundary/control classes
   - ensure that actors only talk to boundary classes
7. check model for **completeness**
   - insert new associations (if necessary)
   - model might differ structurally from domain class model
8. Add the known **attributes** in all classes

EXAMPLE: CONTROL CLASSES FOR USE CASES



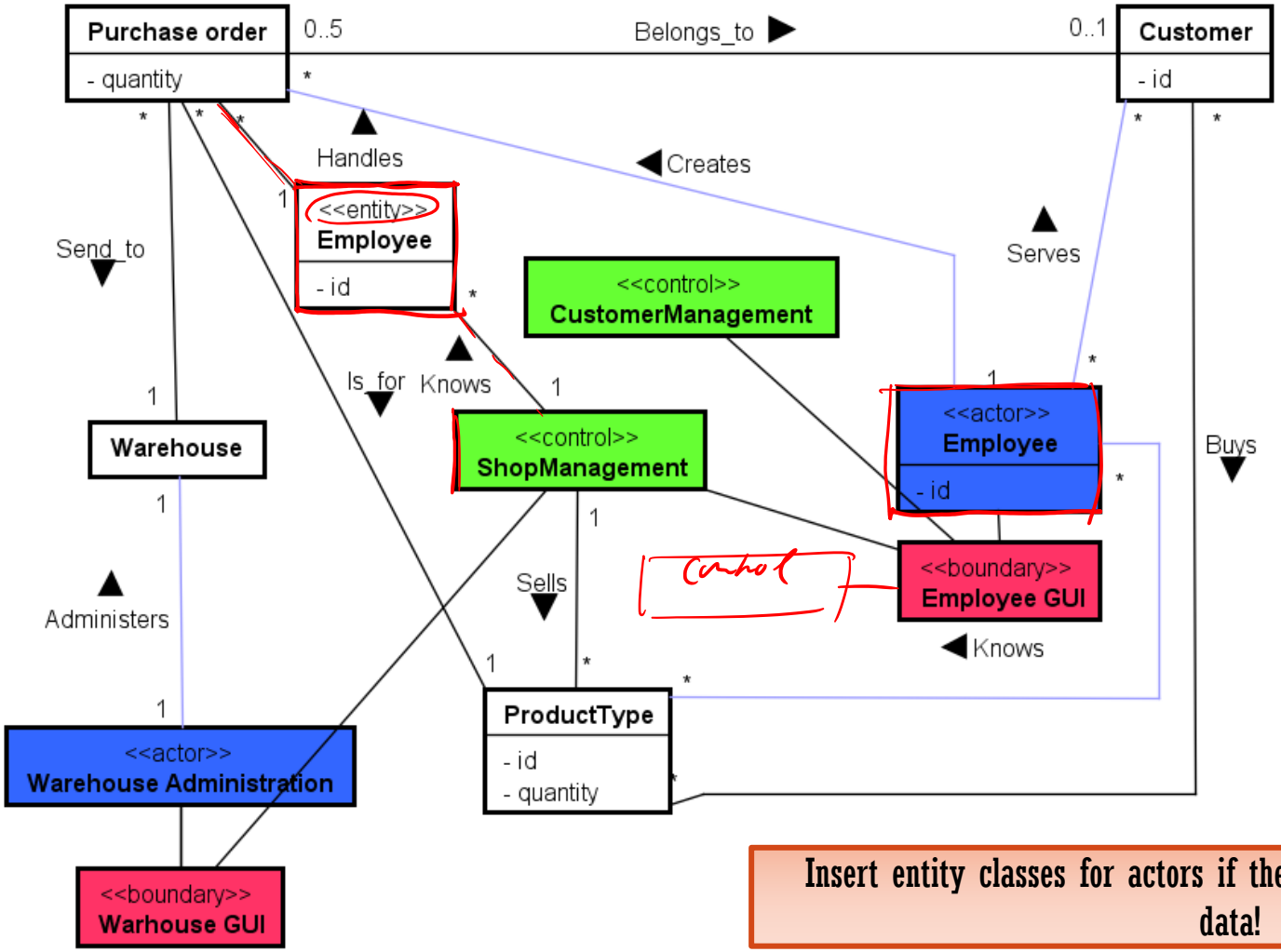
Insert a control class for each use case and connect with the boundary classes according to the use case / actor relation!

EXAMPLE: CONTROL CLASSES FOR USE CASES?



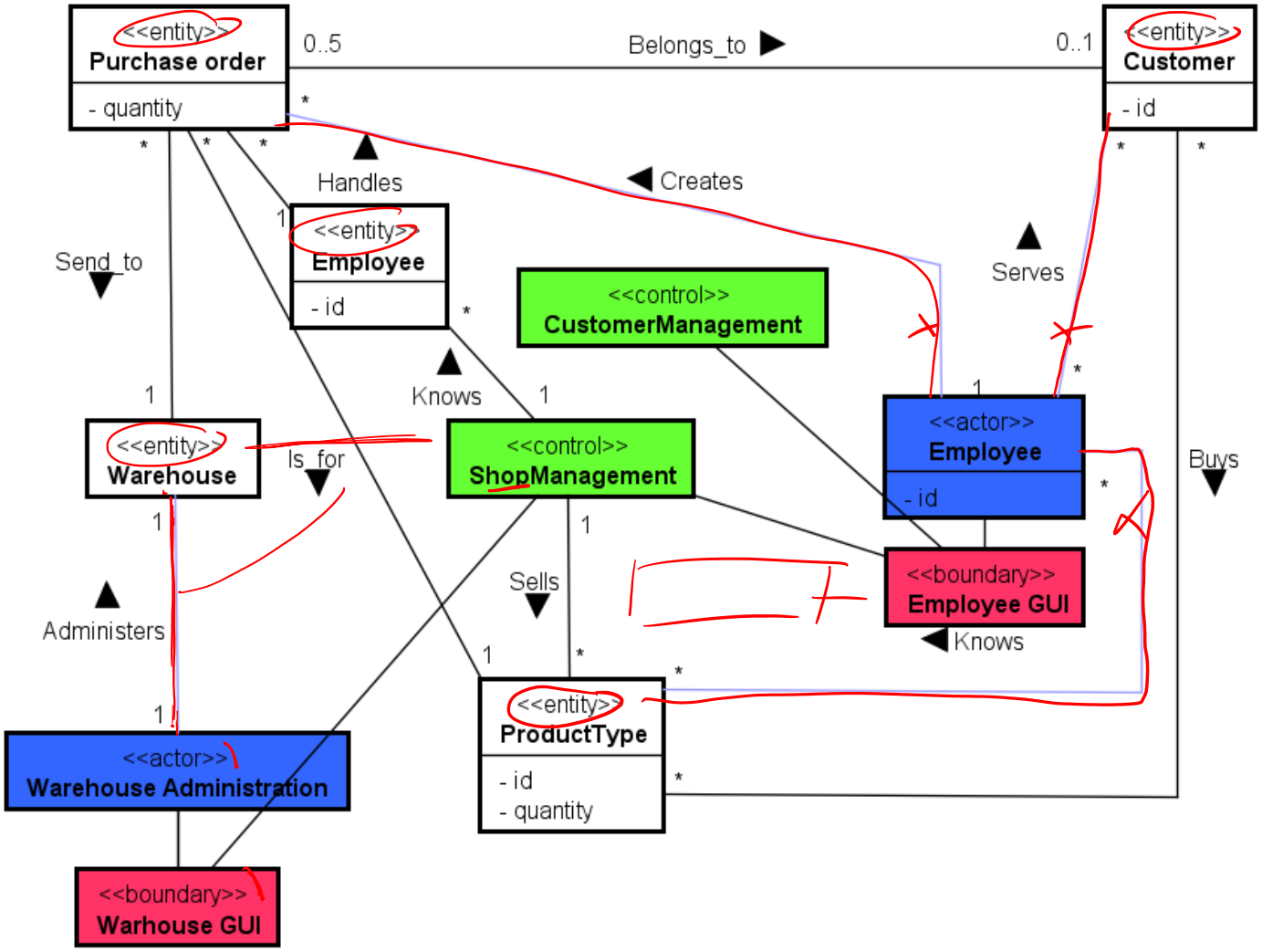
Insert a control class for each use case and connect with the boundary classes according to the use case / actor relation!

EXAMPLE: ENTITY CLASSES FOR ACTORS?

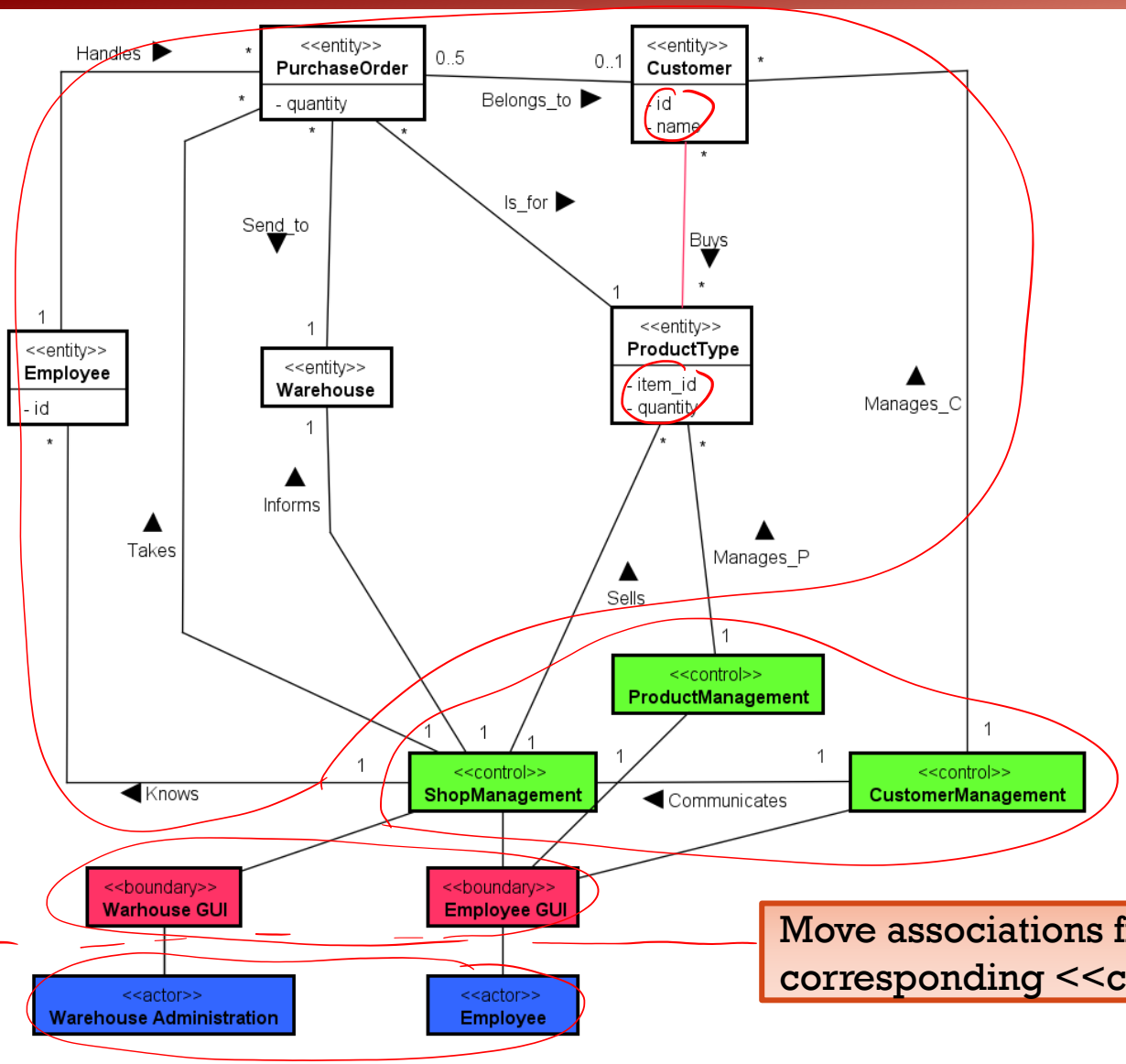


Insert entity classes for actors if the system needs to keep actor data!

EXAMPLE: ADD STEREOTYPES FOR ENTITIES



EXAMPLE: ENFORCE 1:1 RELATIONS

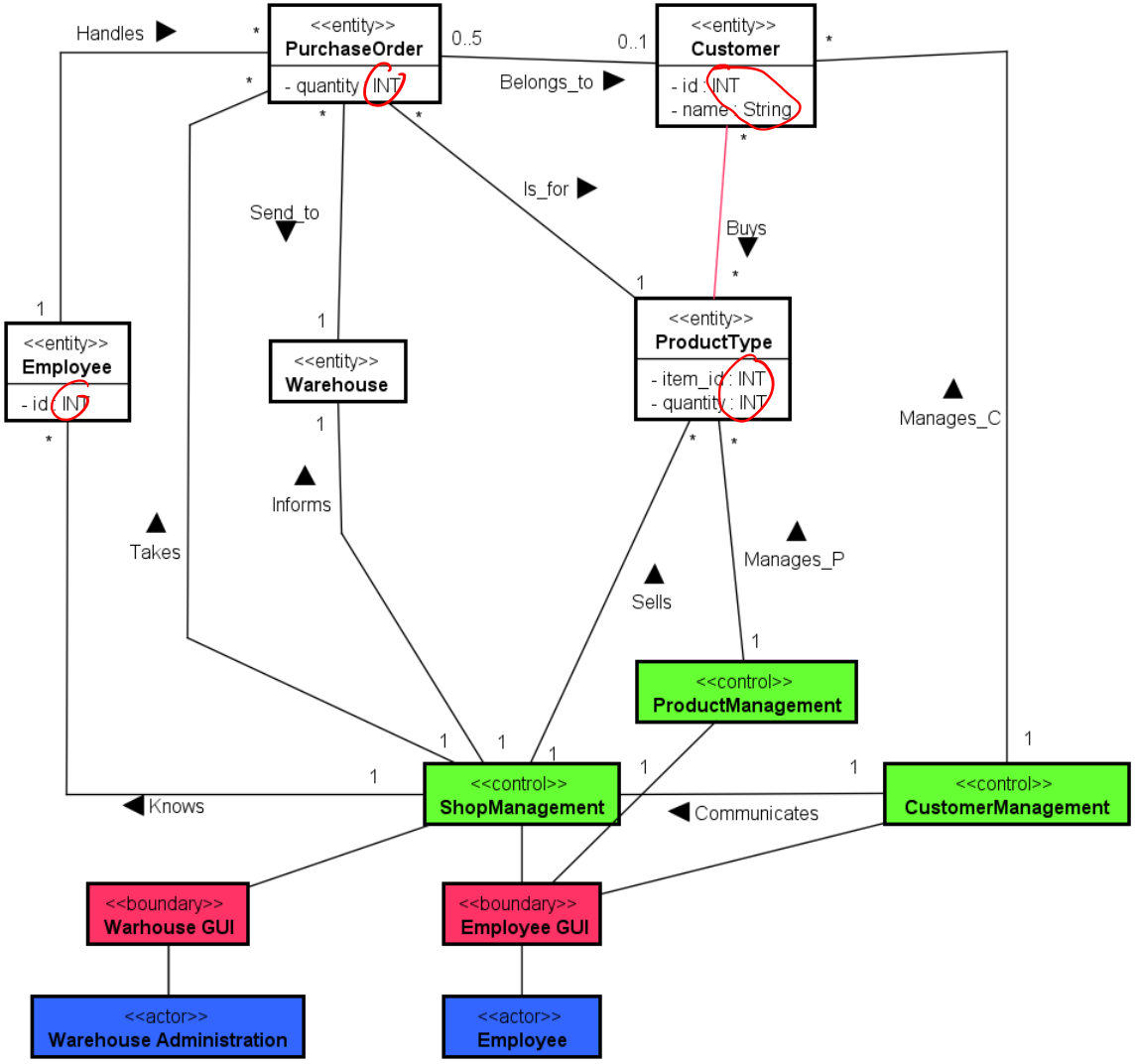


Move associations from <<actor>> to corresponding <<control>>!

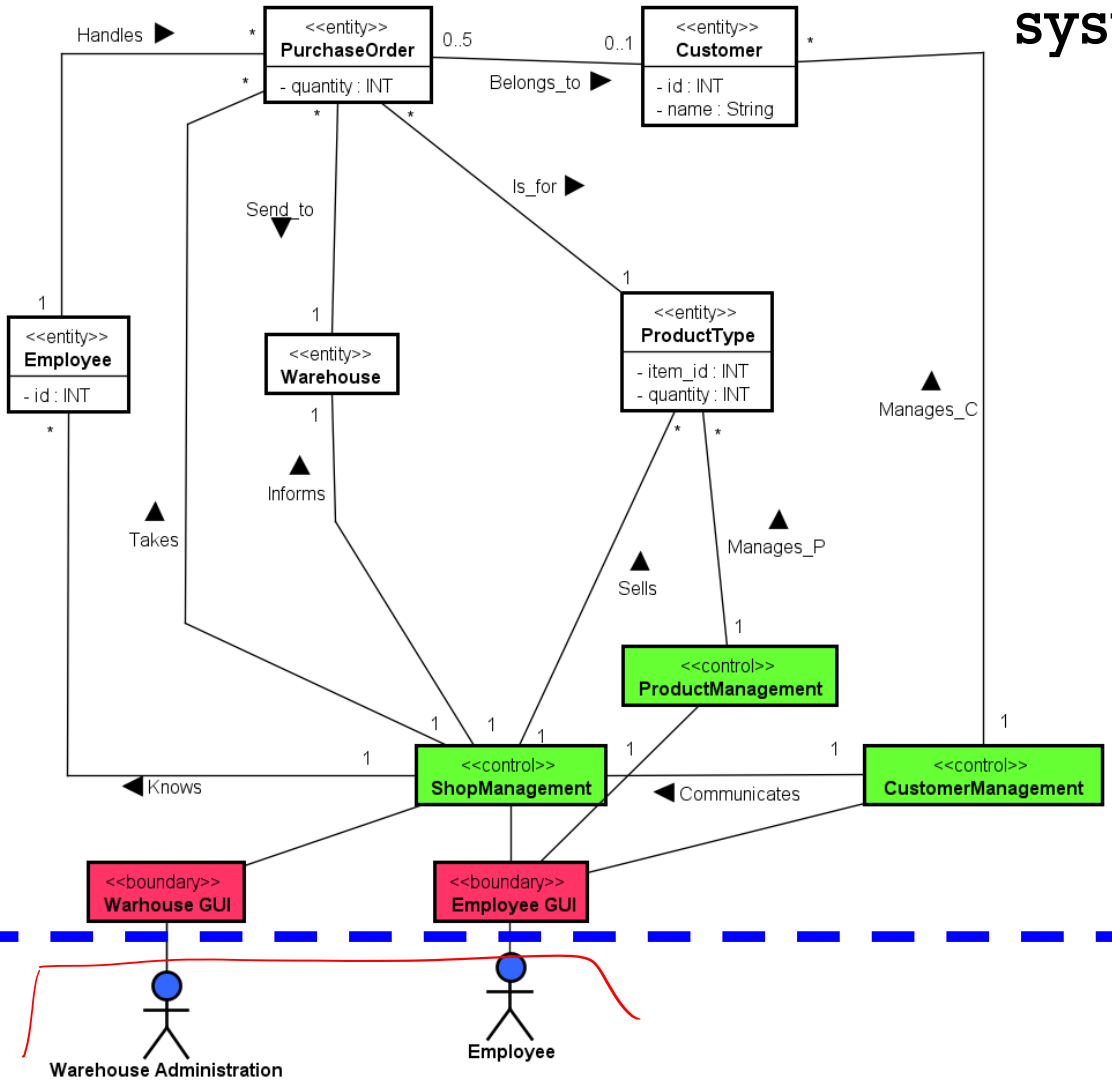
A RECIPE TO COOK SYSTEM CLASS MODELS...

1. start with the domain class model
2. identify **actors**
   - check in the use case diagrams
3. identify **boundary classes** for **actors**
   - Identify and represent the user interface
4. identify **control classes** for **use cases**
   - between boundary and entity classes
   - typically one control class per use case
5. insert entity classes for actors (if required)
   - reflect necessary properties of the actors in the system
6. identify entity classes
   - Model known properties as attributes
   - ensure **1:1 associations** between actor/boundary and boundary/control classes
   - ensure that actors only talk to boundary classes
7. check model for **completeness**
   - insert new associations (if necessary)
   - model might differ structurally from domain class model
8. Add the known **attributes** in all classes

EXAMPLE: ADD TYPES TO ATTRIBUTES



RESULT: SYSTEM CLASS MODEL



SUMMARY (1)

What is this diagram for:

- This diagram is more specific than the Domain Class model, it now considers what is included in the system and what is outside of the systems scope. It is based on the Domain Class model but adds detail to it and does not show things that are outside of our systems scope.

Main things in this diagram:

- Actors are only connected to Boundaries
- Boundaries are only connected to Actors or Controls
- Controls are only connected to Boundaries or Entities
- Each association has multiplicities and a name with reading direction (except Actor-Boundary and Boundary-Control)
- Each class has a stereotype
- Classes have attributes and types of attributes
- Relations between classes are still represented with associations (not as attributes in classes)
- Names of classes and associations help reading and understanding the diagram
- You can create new classes for Boundaries and Controls or reuse classes from your Domain class model, when you think they represent a Boundary or Control

SUMMARY (2)

Consistency

- Consistency with Domain Class diagram:
  - Actor and Entity classes can also be found in the Domain class model (just in the domain class model they were not specified as such)
  - If classes in Domain class model had attributes, they have to be in the System class model as well
  - If a class or classes from the domain class model are neither an Entity, Control, Boundary or Actor you might need to delete it from the Domain class model
  - If a class from the Domain class model is an Actor and the System also needs to store their data in the System you need to mirror the class (have two classes with basically the same name in your System class model - one as Actor one as Entity.)
- Consistency with Use Case diagram:
  - Actors are the same as in Use Case diagram
  - Boundary for each actor
  - Control class for each Use Case (rule of thumb for the beginning, this might change later in your design if you think it makes sense)

IMPLEMENTATION CLASS MODEL

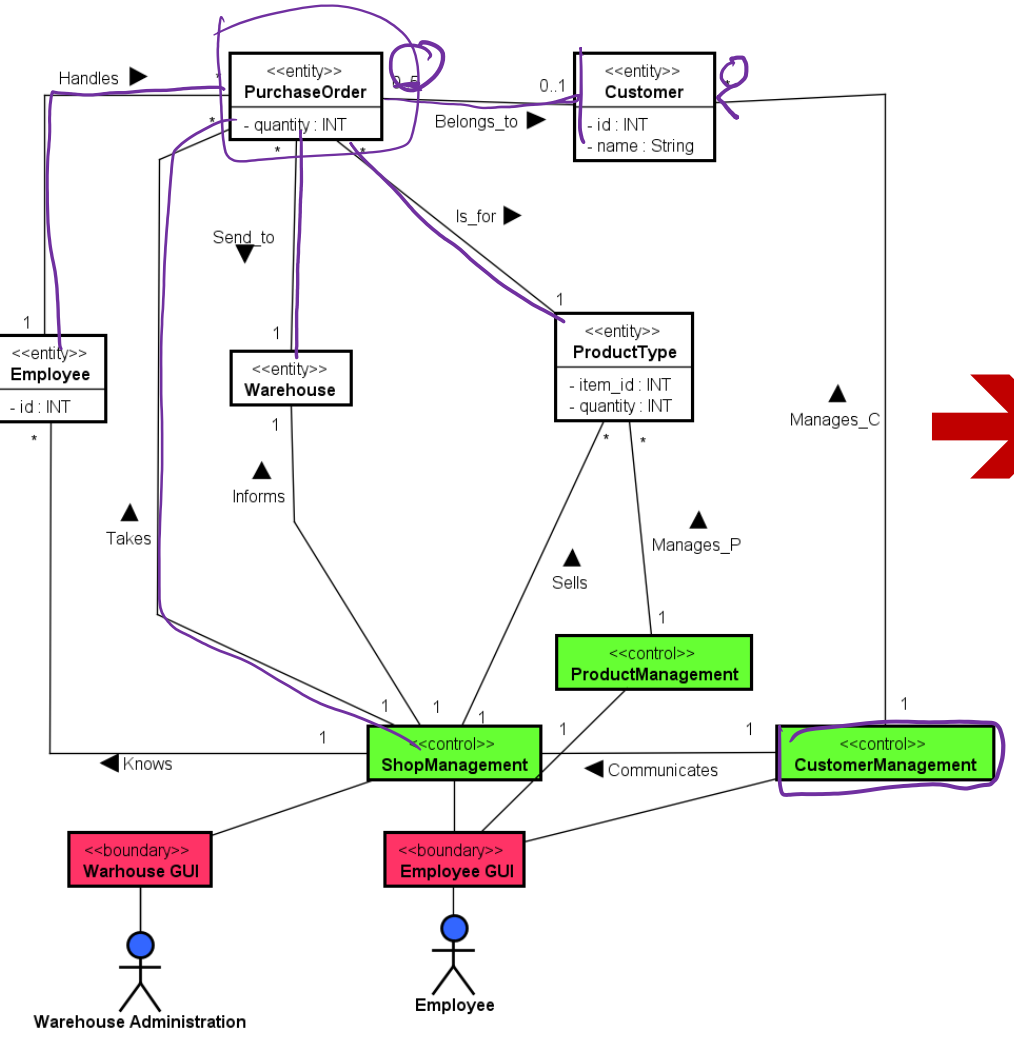
IMPLEMENTATION CLASS MODEL

- This one is very close to the implementation
- Goal: Systematically or automatically create code

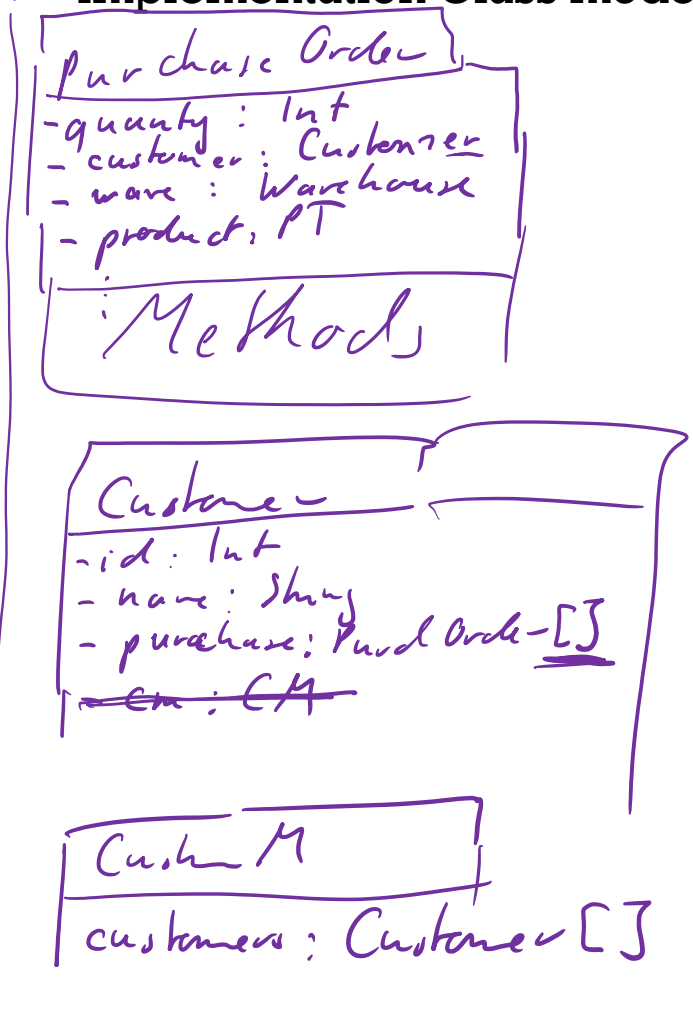
- Includes all class elements (names, members, methods): types, access specifier
- Includes programming language specific features (the others where basically programming language independent)
- Does not use Associations anymore

EXAMPLE:

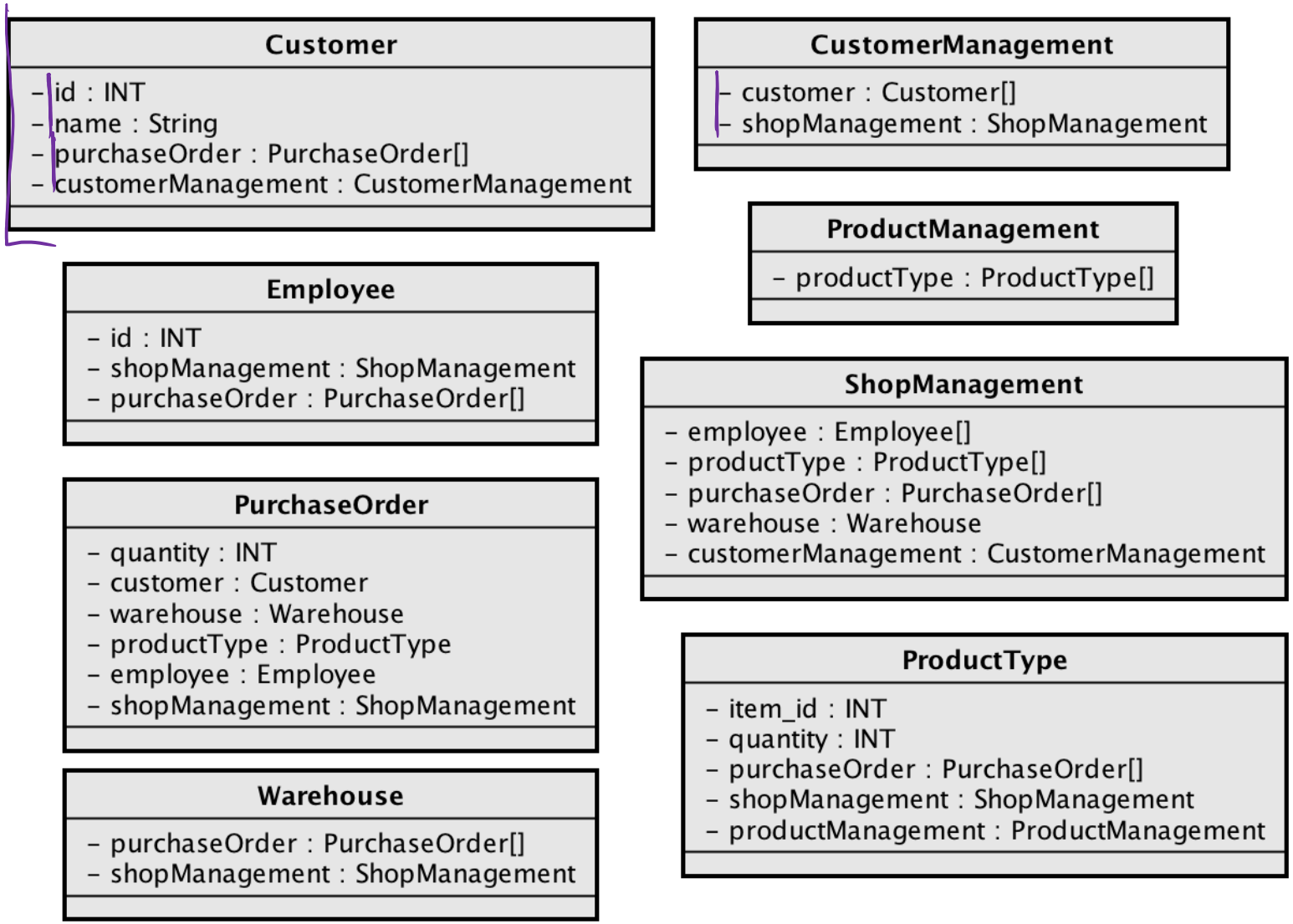
System Domain Class model



Implementation Class model



EXAMPLE: IMPLEMENTATION CLASS MODEL



SUMMARY

- Close to implementation
- Has not associations
- Is consistent to System Class model
- We will revisit this later on!

OVERALL SUMMARY

SUMMARY

- Class diagrams can be used for different level of abstraction
  - It is necessary to decide at beginning what level of detail you need
  - Each level of abstraction has its uses do not mix the abstraction levels

- Class diagrams have a specific syntax – it is a language
  - Make sure to use the correct syntax for what you want to convey
  - Do not make up your own style – nobody will understand

IN YOUR PROJECT

- For now I want you to start with the Domain Class Model
- The System Class model will be used later in the process
- The Implementation Class model will also be used later and we will revisit how to derive it
  - Then we will also include methods
  - Types
  - Constructors
  - Etc.

Overview of process we will use