OO Design

Starting with the basics

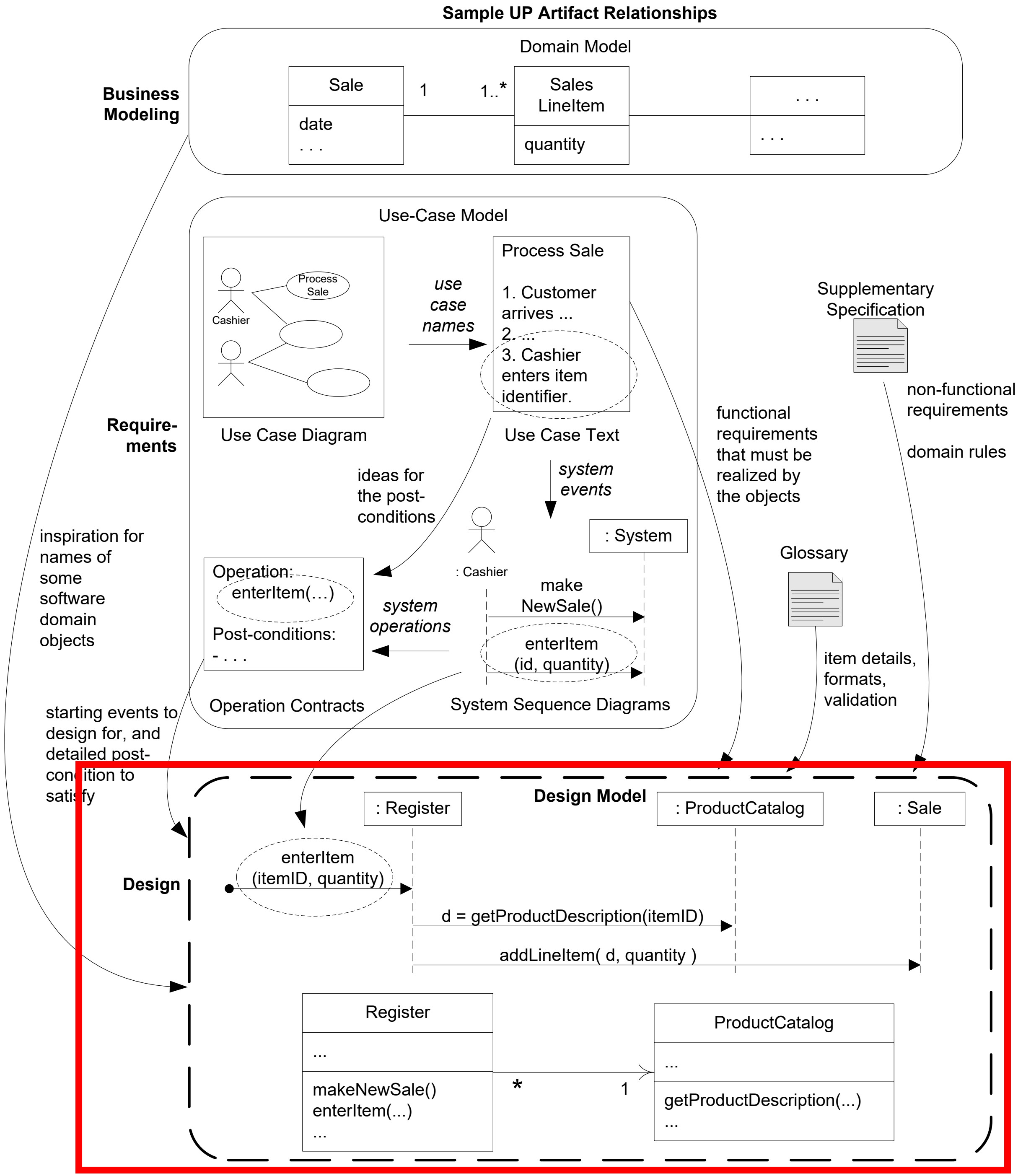
ISEP / LETI / ESOFT

Topics

- OO Design
- Responsibility-Driven Design (RDD)
- GRASP – General Responsibility Assignment Software Patterns
  - Pure Fabrication
  - Controller
  - Information Expert
  - Creator

OO Design

Artifacts Overview

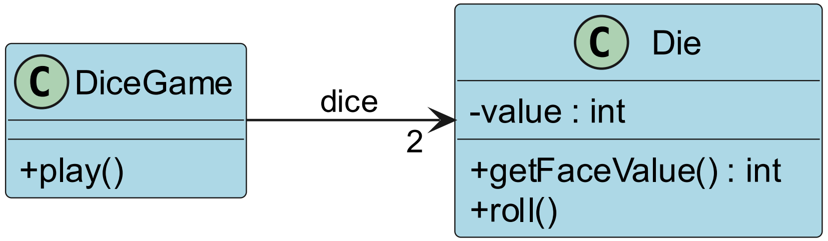


OO Design

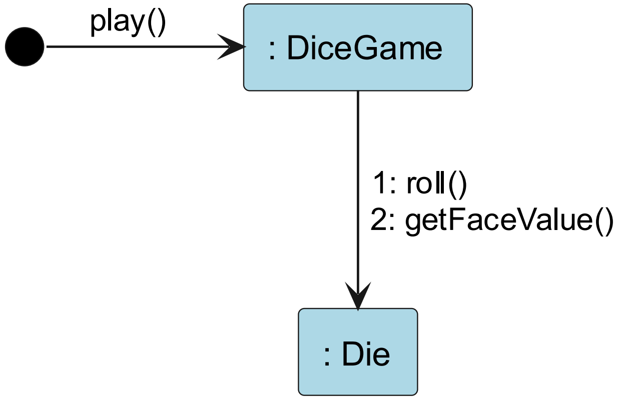
- Logical solution based on the Object-Oriented Paradigm (OOP)
  - in terms of collaborative objects and with responsibilities
  - described in a **Design Model**
- Design Model includes…
  - Static view: **Class Diagram (CD)**
  - Dynamic view (interaction diagrams):
    - **Sequence Diagram (SD)**: illustrates the interactions of objects in a "grid" format, in which objects are added successively to the right and in which the order of messages occurs from top to bottom.
    - Communication Diagram: illustrates interactions in a graph or network format, in which objects can be placed anywhere on the diagram, and where the order of messages is defined by a number.

Design Diagrams – Examples

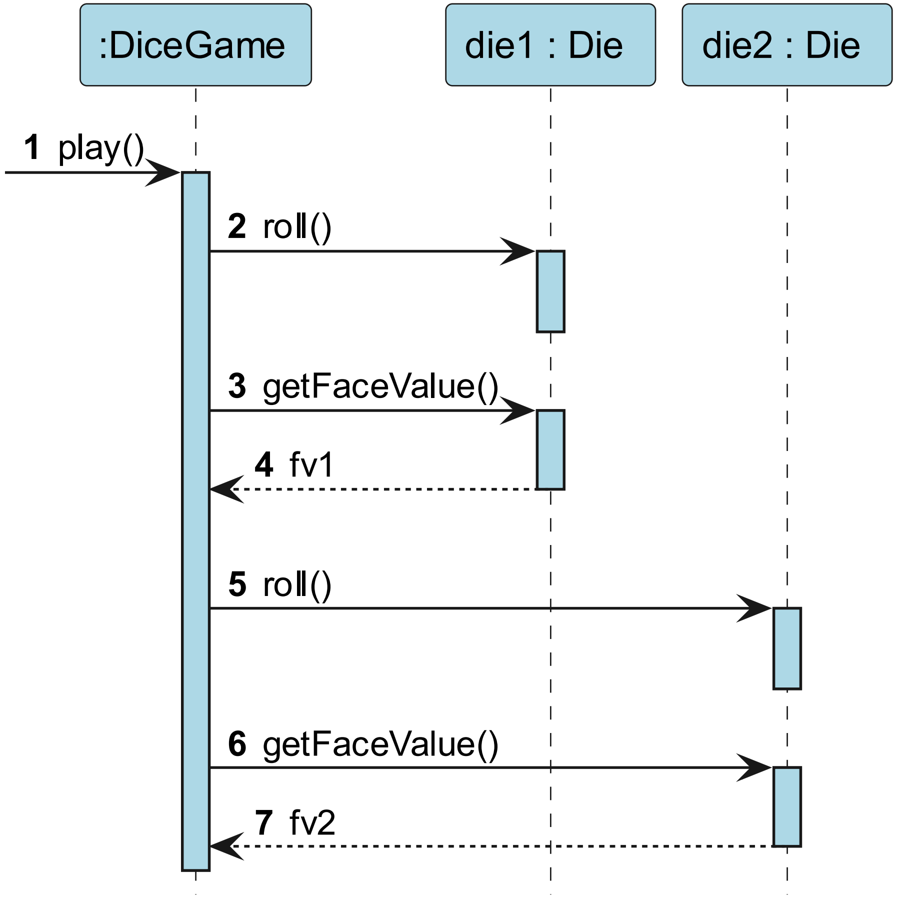
CLASS DIAGRAM



COMMUNICATION DIAGRAM



SEQUENCE DIAGRAM



Approach to Design

- After:
  - Identifying the requirements; and
  - Specifying the domain model
- Then, design begins by:
  1. Promoting conceptual classes to software classes
  2. Adding functions to software classes
  3. Defining messages between classes/objects
- To meet the requirements.

- Create the diagrams (e.g. class and sequence diagram) **in parallel**

Recommended Method – User Scenario Realization

- Driven by the functional requirements, strongly supported by the user scenarios (either US or UC) and the Domain Model
- Therefore, for each US/UC the following artifacts are created:
    - **Rationale of responsibilities assignment** according to
        - **GRASP – General Responsibility Assignment Software Patterns (or Principles)**
        - SOLID
        - Other patterns (e.g. GoF) and best practices
    - **Sequence Diagram** highlighting interactions between classes/objects
    - Partial **Class Diagram**
- The complete Class Diagram results from the partial CD of each user scenario realization

User Scenario Realization

- A **User Scenario** is an instance of a Use Case, i.e. one path through the Use Case

- The realization of a user scenario highlights the link between requirements, expressed by a user scenario and the design of the objects that guarantee those requirements
    - *"A use-case realization describes how a particular use case is realized within the Design Model, in terms of collaborating objects" [RUP]*

- Usually, it only covers the **Main Success Scenario** (aka *happy path*) between the user and the system
    - If relevant, other possible flows (leading to success and/or handling errors) might also be considered and realized

Activities in OO Design

- OO Design is based on a metaphor named **Responsibility-Driven Design (RDD)**
  - It is used to think about how to assign responsibilities to objects

- Apply **GRASP**, **SOLID**, **GoF** and other principles and patterns during design and coding
  - These ones are seen as being the best practices for assigning responsibilities in well-established and characterized circumstances

Responsibility-Driven Design (RDD)

Responsibility-Driven Design (RDD)

- **Metaphor** to help with the OO software design process based on
  - the notion of **responsibility**; and
  - the idea of **collaboration**

- What are responsibilities?
  - A responsibility is an **abstraction** of what the object does
  - They are **obligations and behaviors** of an object depending on the role it performs in the system
  - A responsibility is not the same as a function, but **functions implement responsibilities**

Types of Responsibilities

- Responsibilities of **“doing”**
  - To do itself as, for instance, to create an object, to do calculations, etc.
  - To delegate, i.e. to initiate actions on other objects
  - To control and coordinate activities on other objects

- Responsibilities of **“knowing”**
  - To know its own private information
  - To know related objects
  - To know how to obtain or calculate new information

Idea of Collaboration

- RDD includes the idea of **collaboration** between objects

- Responsibilities are implemented through functions
  - which act alone; and/or
  - collaborate with other functions and objects

- Think about **software objects** just like one thinks about **people with responsibilities**, who collaborate with other people to do their work

Collaborating Objects – Example

- **Scenario**
  - A company sells several products. Each product is categorized in a single category.
- **Needs**
  1. An administrative wants to create a new category
  2. Someone wants to know the list of products for a given category (using an "id")
- Identified concepts/objects:
  - Company
  - Product
  - Category
- How could these objects collaborate to meet that needs?
  - (more on this in the next slides)

GRASP – General Responsibility Assignment Software Patterns (or Principles)

GRASP - General Responsibility Assignment Software Patterns (or Principles)

- GRASP is a methodical **approach to OO Design**
  - Based on principles/patterns for **responsibilities assignment**
  - Helps to understand the fundamentals of object design
  - Allows to apply design reasoning in a methodical, rational, and understandable way

- In UML, the design of Interaction Diagrams (e.g. class and sequence diagrams) is a means to consider and represent responsibilities
  - When designing, you decide which responsibilities to assign to each object

GRASP

- Pure Fabrication *
- Controller *
- Information Expert *
- Creator *

- High Cohesion
- Low Coupling
- Polymorphism
- Indirection
- Protected Variation

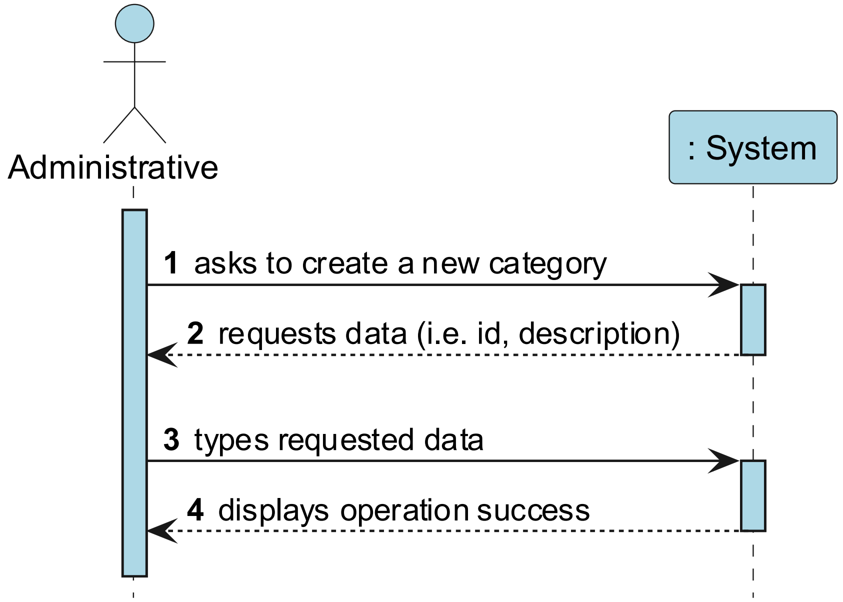
* Patterns addressed in this slide deck

GRASP

Pure Fabrication for the User Interface

Pure Fabrication Example (1/4)

- In the context of the example presented previously consider the need to **create a new category**, partially depicted in the following SSD


Administrative
: System
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 displays operation success

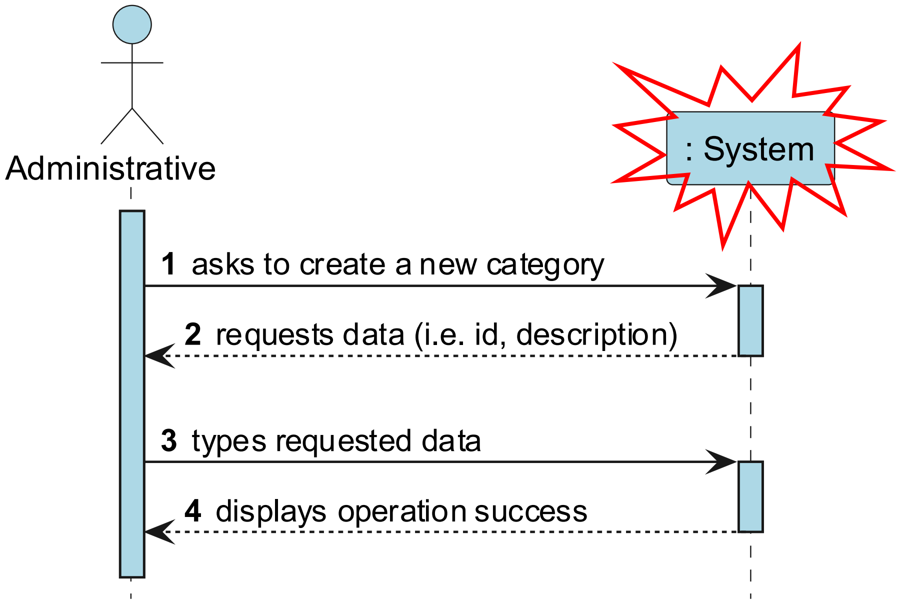
Pure Fabrication Example (2/4)

- **Problem**
  - Which class from the System should interact with the Actor?
  - Which class should have that responsibility?

- **Solution**
  - Assign a coherent set of responsibilities to an **artificial class** that does not represent a domain concept. Such class is made up to promote high cohesion, low coupling, and reuse.

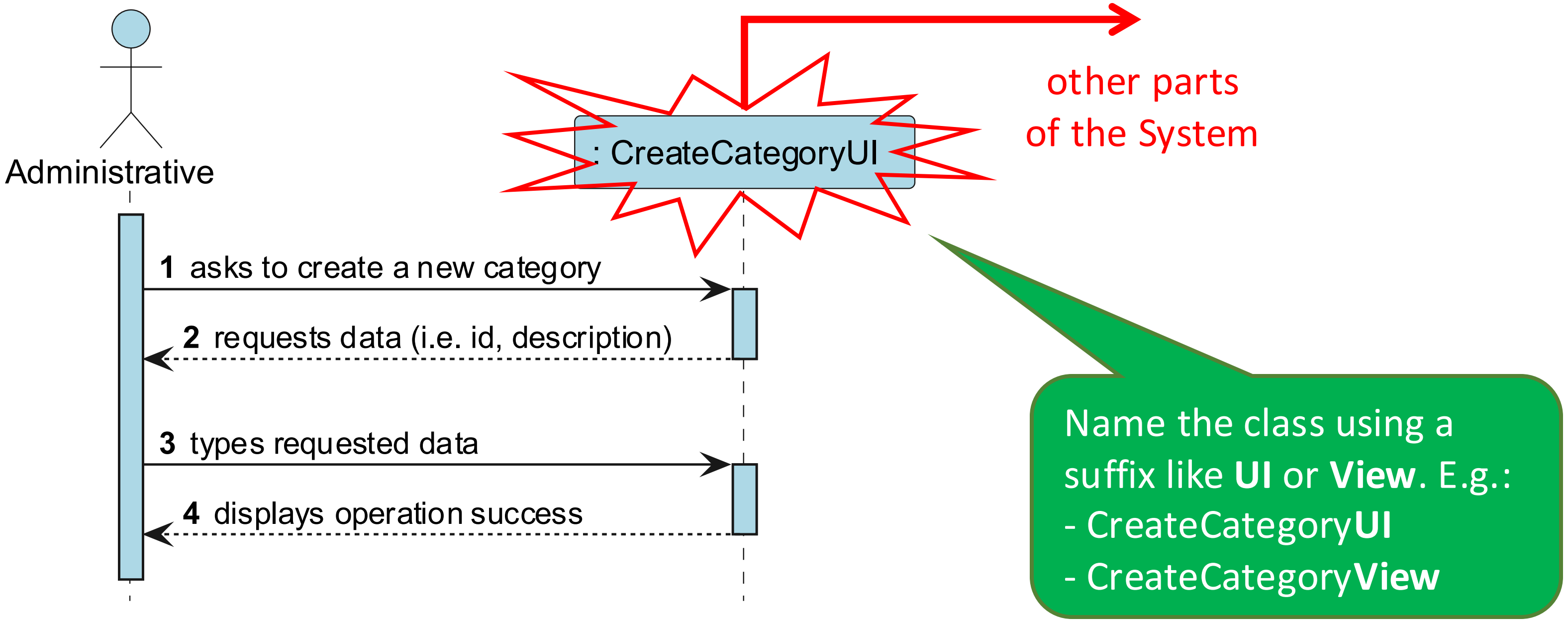
Pure Fabrication Example (3/4)

- What does the System consist of?
  - User Interface (UI)
  - Domain
  - …


Administrative
: System
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 displays operation success

Pure Fabrication Example (4/4)

- What does the System consist of?
  - User Interface (UI)
  - Domain
  - …



Administrative
: CreateCategoryUI
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 displays operation success
other parts of the System

Name the class using a suffix like **UI** or **View**. E.g.:
- CreateCategory**UI**
- CreateCategory**View**

GRASP

Controller

Controller

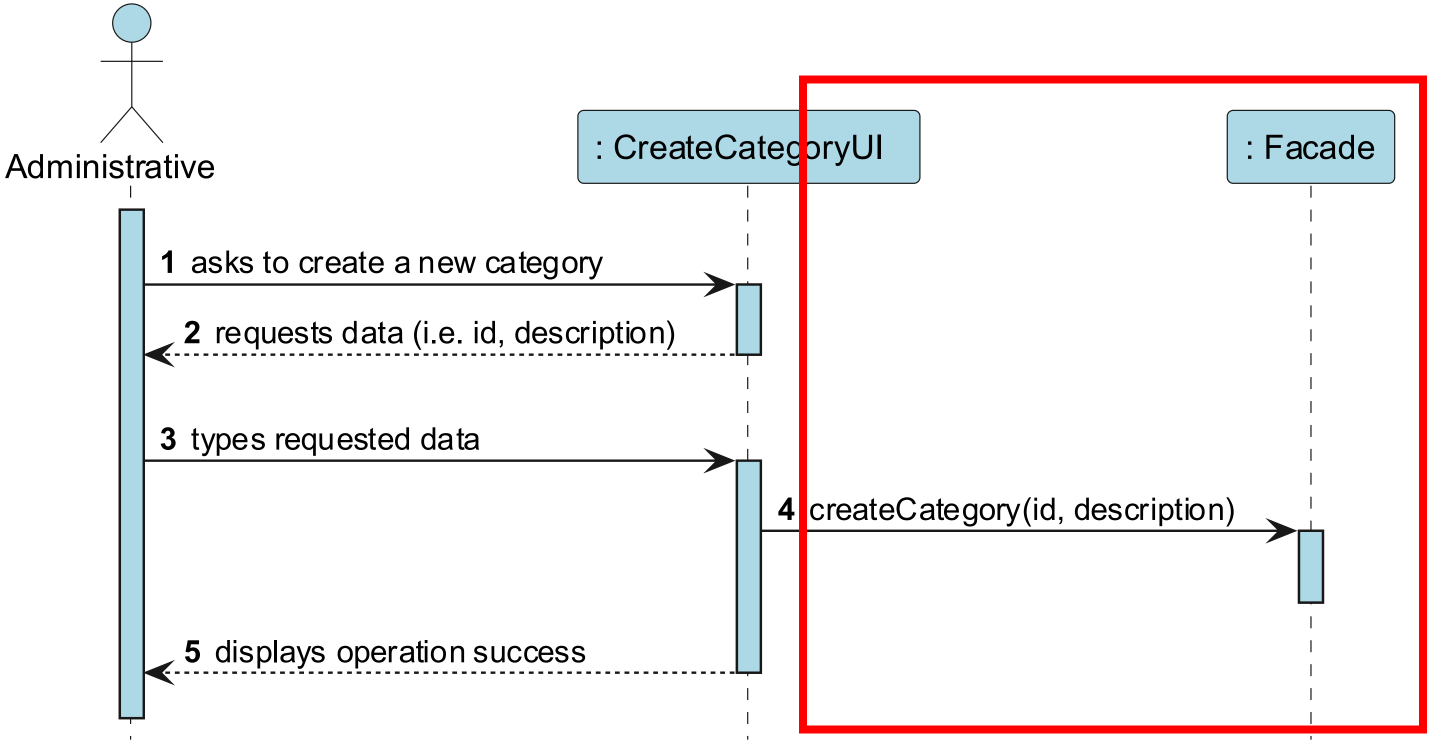
- **Problem**
  - Which class should be responsible for responding to an input event in the system generated by the User Interface (UI)?

- **Solution**
  - Assign the responsibility to **one** of the following classes
    - The one that globally represents the system, a device or a sub-system (***facade**** **controller**)
      - 1 class for the entire system
      - (not used in ESOFT)
    - The one that represents a user scenario (US/UC) in which the event occurs:
      - **1 Controller per US/UC**
      - Class name format: *<UCName>Controller*

* spelled *façade*

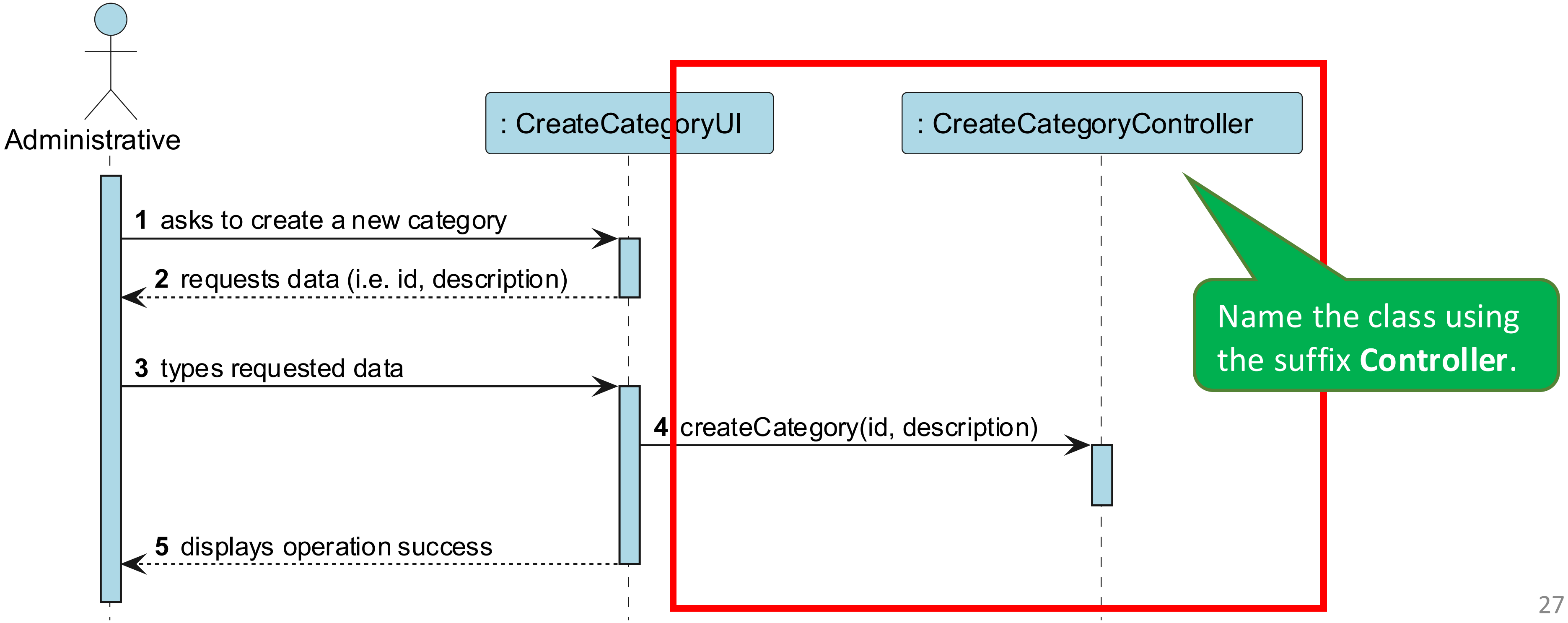
Controller: Facade vs. <*UCName*>Controller (1/2)

- ***facade* controller**
  - When there are few system events
  - Might represent the system or the device


Administrative
: CreateCategoryUI
: Facade
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 createCategory(id, description)
5 displays operation success

Controller: Facade vs. <*UCName*>Controller (2/2)

- **One Controller per US/UC (CreateCategoryController)**
  - When there are many events and the *facade* controller would be very extensive, with many responsibilities
  - Represents the user scenario in which the event occurs


Administrative
: CreateCategoryUI
: CreateCategoryController
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 createCategory(id, description)
5 displays operation success

Name the class using the suffix **Controller**.

Controller as a Frontier Between Layers

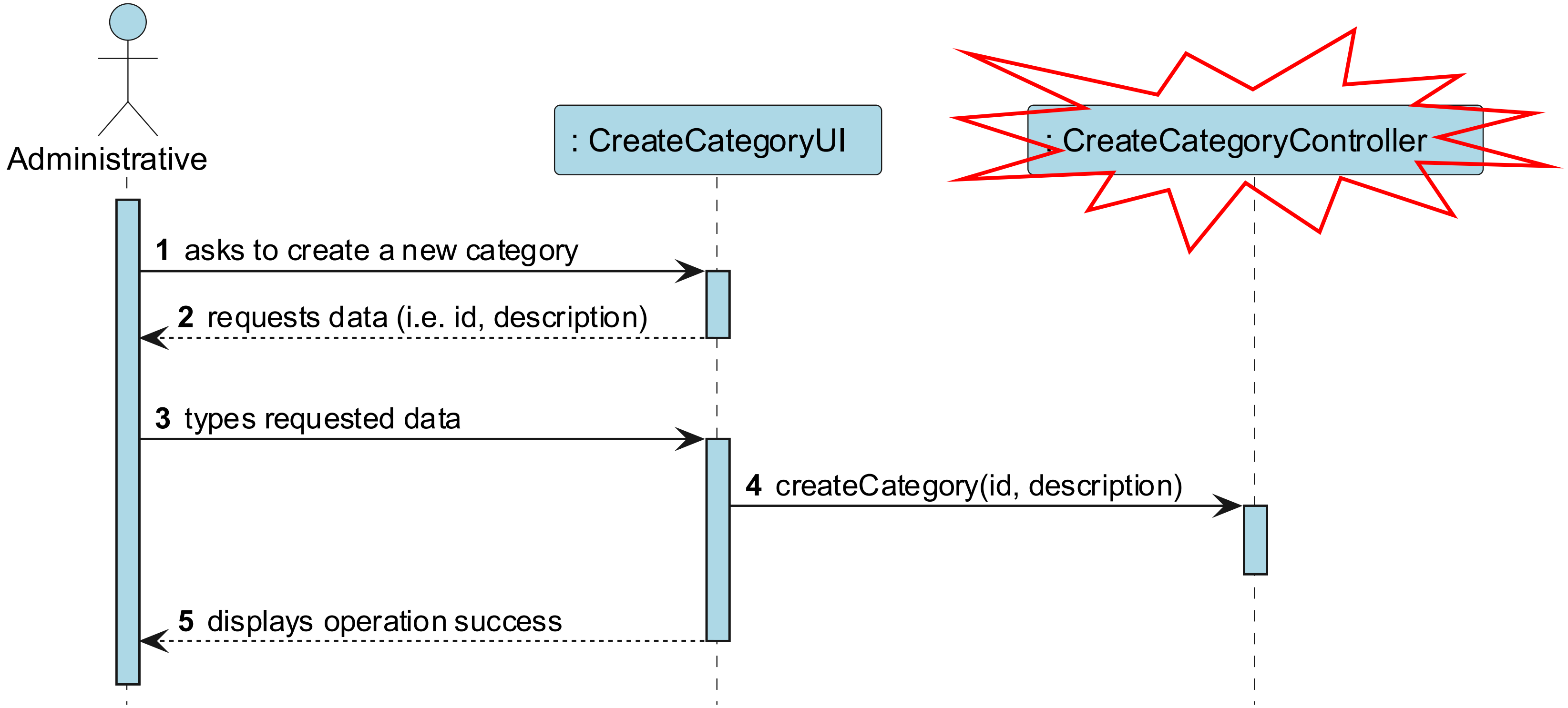
- UI Layer
  - Set of all classes whose responsibilities are somehow related to the user interface, i.e. concerned with presenting and collecting data to/from the user
  - E.g.: data forms, menus and menu options, reports and dashboards

- Domain Layer
  - Set of classes whose responsibilities are somehow related to business logic
  - Captures/represents business entities/concepts and ensures business rules

- **Controllers act as intermediaries/frontiers between the two layers**
  - Ensuring the decouple between the UI Layer and the Domain Layer

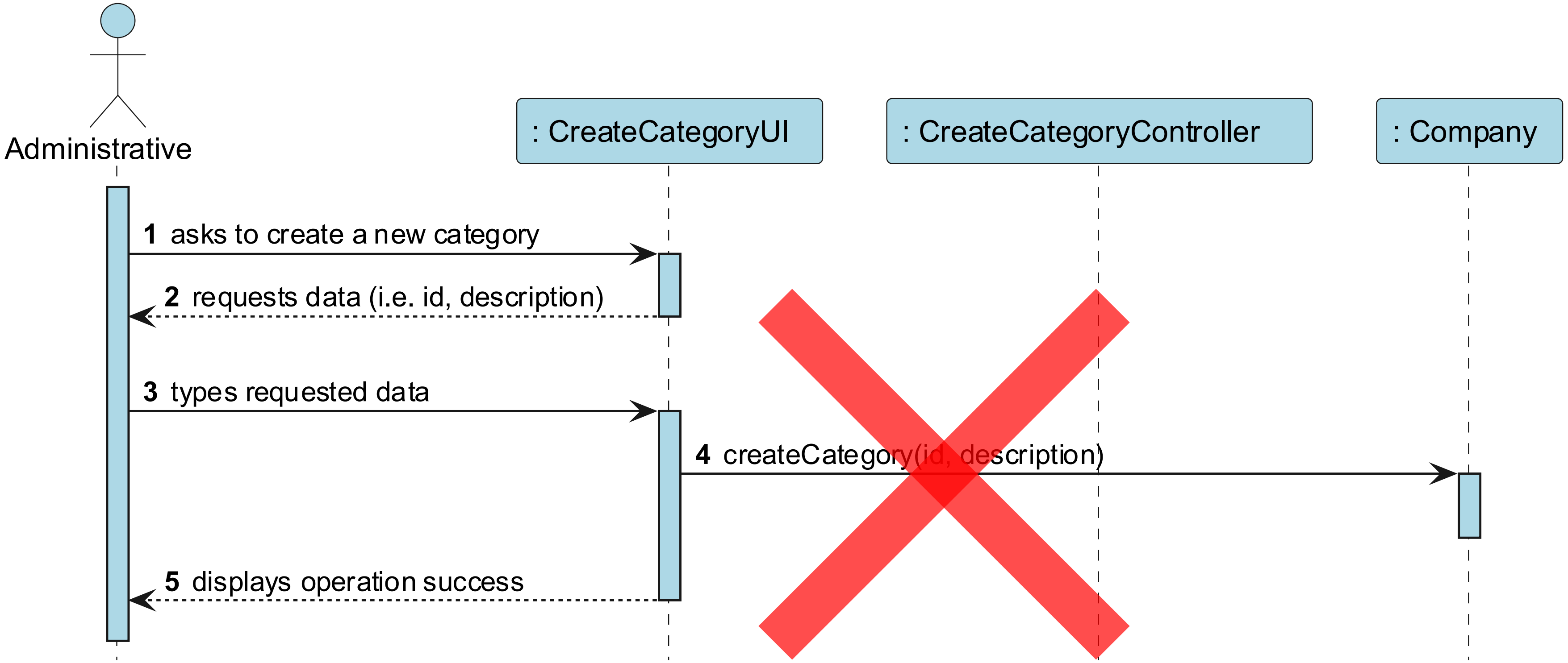
Controller

- The Controller is the first class/object after the UI Layer that is responsible for receiving or handling a system operation


Administrative
: CreateCategoryUI
: CreateCategoryController
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 createCategory(id, description)
5 displays operation success

Controller

- The direct communication of UI classes with the domain classes must be avoided
- The Controller enforces this


Administrative
: CreateCategoryUI
: CreateCategoryController
: Company
1 asks to create a new category
2 requests data (i.e. id, description)
3 types requested data
4 createCategory(id, description)
5 displays operation success

Controller as a Delegator

- If all the events in a US/UC are handled in the same class (i.e. the controller class), then:
    - It is possible to maintain information about the state of the US/UC
    - It is possible to identify errors in the sequence of events
- Usually, a **Controller should**:
    - **Coordinate/Control the flow/activity of a US/UC**
    - **Not be processing data/information – it should delegate (processing) to other classes/objects**
- System operations should be handled at the Domain Layer by the controllers and not at the UI Layer by (G)UI objects

Benefits of Using Controllers

- Provides guidance on which class/object should handle external events (e.g. from the UI)
- **Increases the possibility of reuse**
- **Allows** the domain layer to be used with **different user interfaces**
- Controls the sequence of events

GRASP

Information Expert

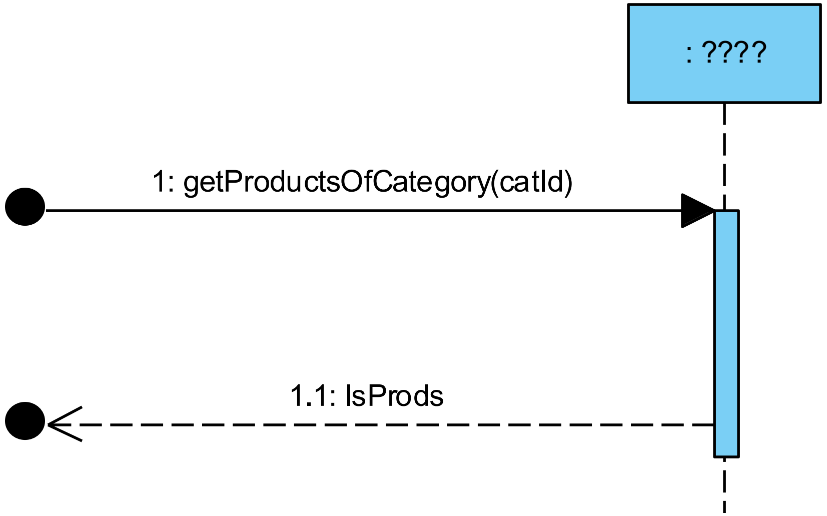
Information Expert

- **Problem**
  - What is the **general principle** for assigning responsibilities to objects?

- **Solution**
  - Assign the responsibility to the **“information expert”**
    - I.e., assign to the class that contains the information needed to fulfill that responsibility
    - Which class?
      - Get inspired by the **Domain Model**

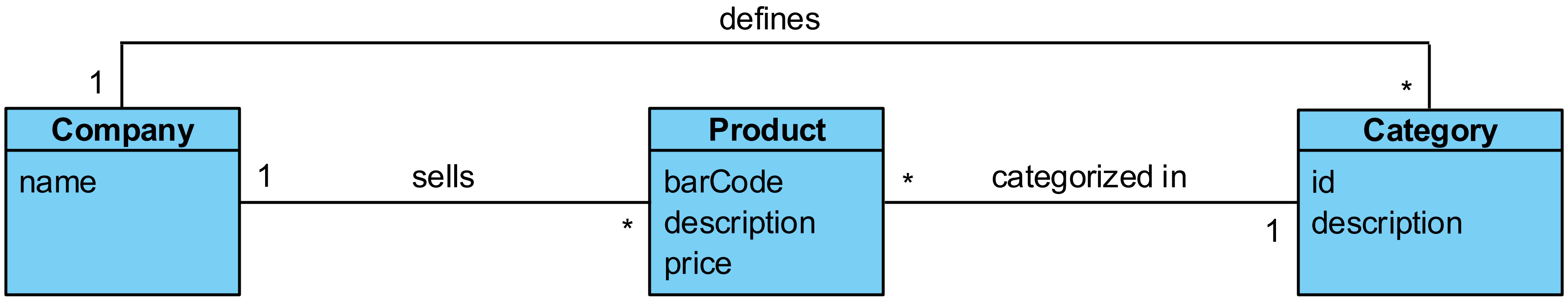
Information Expert – Example (1/7)

- In the context of the example presented previously, consider the need for **someone to obtain a list of products in a certain category** (using its “id”), partially depicted below
- Which class can provide the list of products?



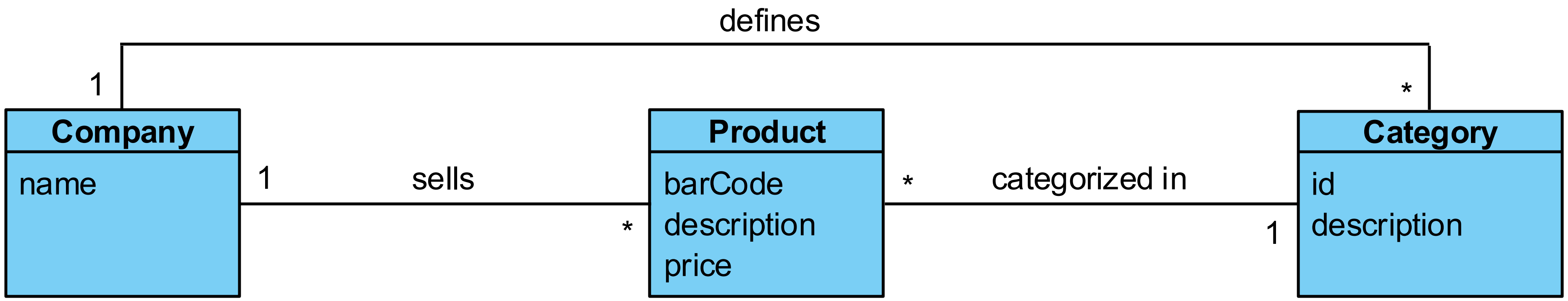
Information Expert – Example (2/7)

- Consider the following Domain Model underlying the problem



Information Expert – Example (3/7)

- Which information do we need?
  - Which class **knows** all its products?
  - Which class **knows** the category a product is categorized in?
  - Which class **knows** the information (e.g. "id") of a category?



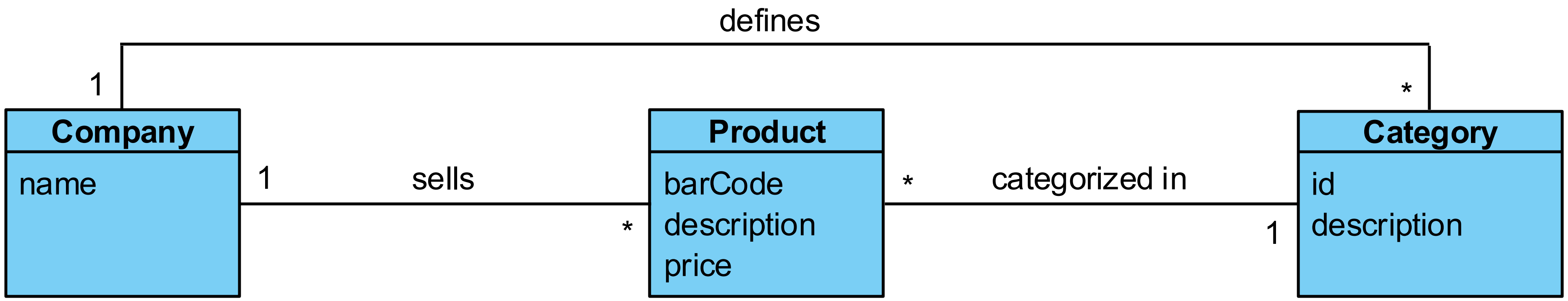
Information Expert – Example (4/7)

- Which information do we need?
  - Which class **knows** all its products?

    → **Company**
  - Which class **knows** the category a product is categorized in?

    → **Product**
  - Which class **knows** the information (e.g. "id") of a category?

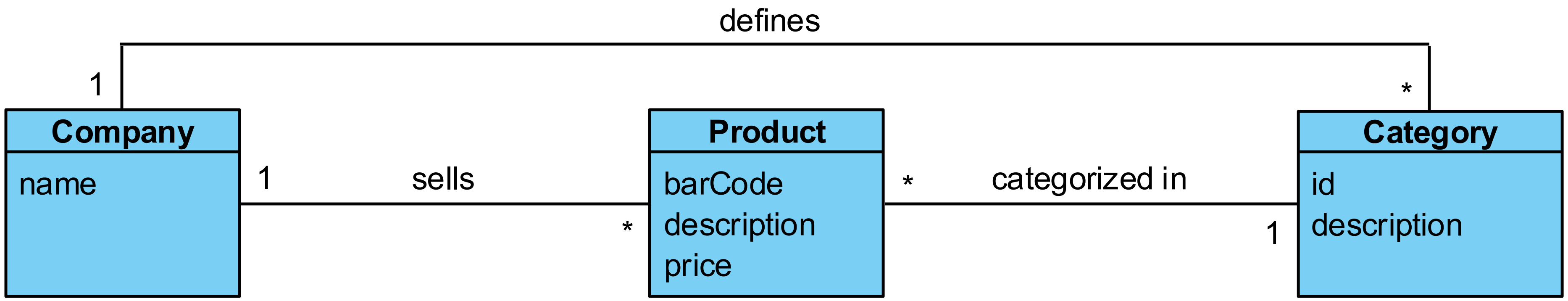
    → **Category**



Information Expert – Example (5/7)

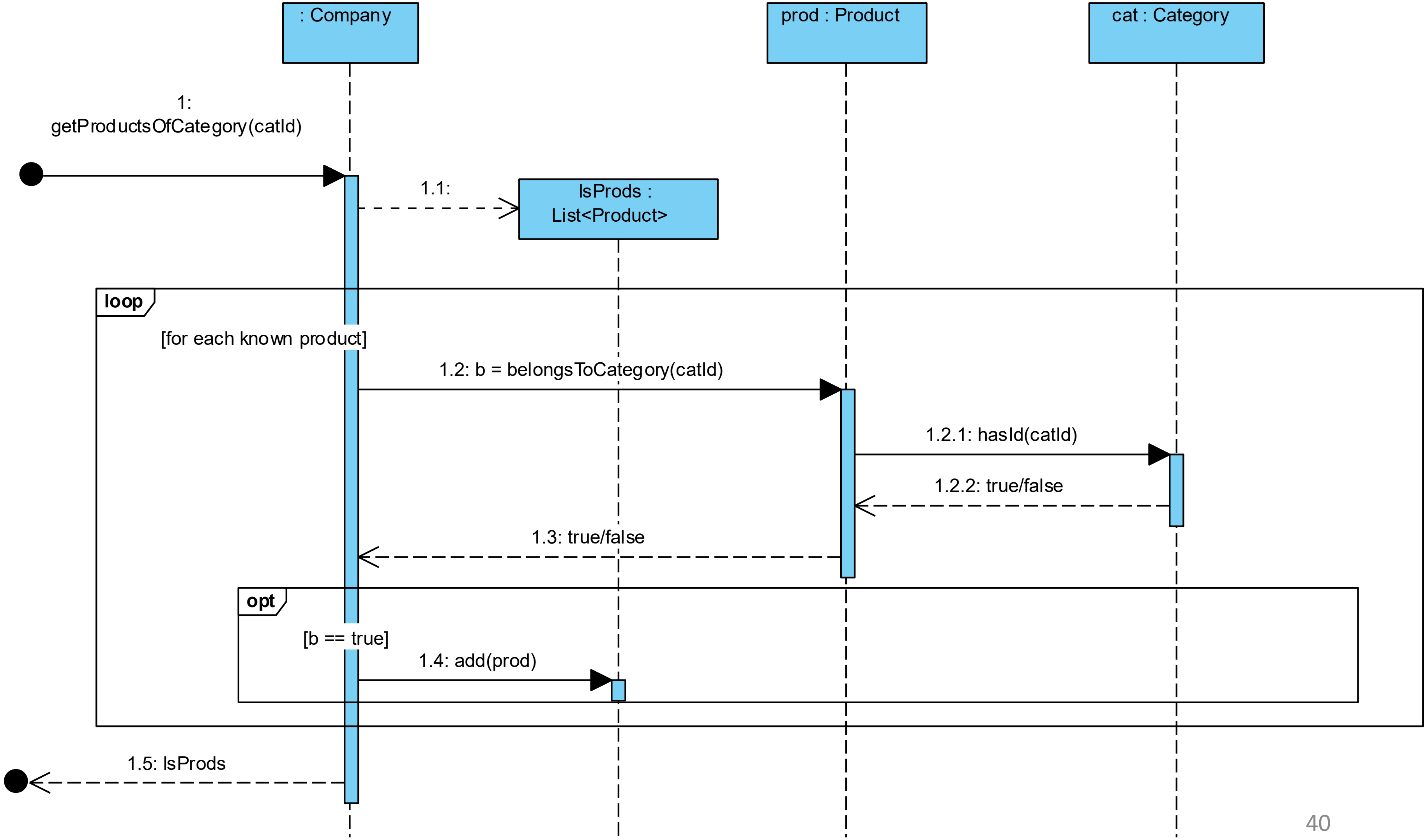
- Assigning responsibilities

Design Class	Responsibility
Company	Knows all its products
Product	Knows which category it is categorized in
Category	Knows its own “id”



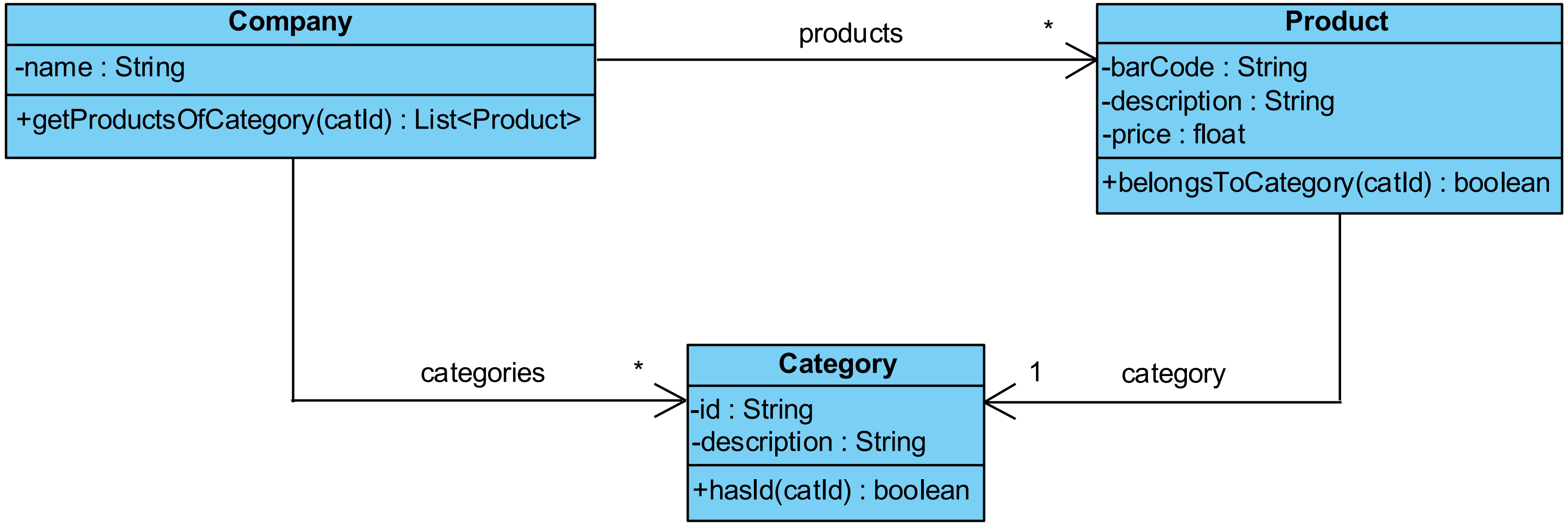
Information Expert – Example (6/7)

- Sequence Diagram



Information Expert – Example (7/7)

- Class Diagram



Information Expert – Contraindications

- The use of Information Expert can sometimes cause
  - Cohesion problems
  - Coupling problems
- … which can be overcome by applying GRASP patterns
  - High Cohesion
  - Low Coupling

- These problems and patterns will be addressed in the next lecture

GRASP

Creator

Creator (1/2)

- **Problem**
  - Who should be responsible for **creating a new object** of a given class?

- **Solution**
  - Assign to class B the responsibility for creating instances of class A under the following conditions (in order of preference):
    1) B contains or aggregates instances of A
    2) B records instances of A
    3) B closely uses A
    4) B has the data for initializing A

Creator (2/2)

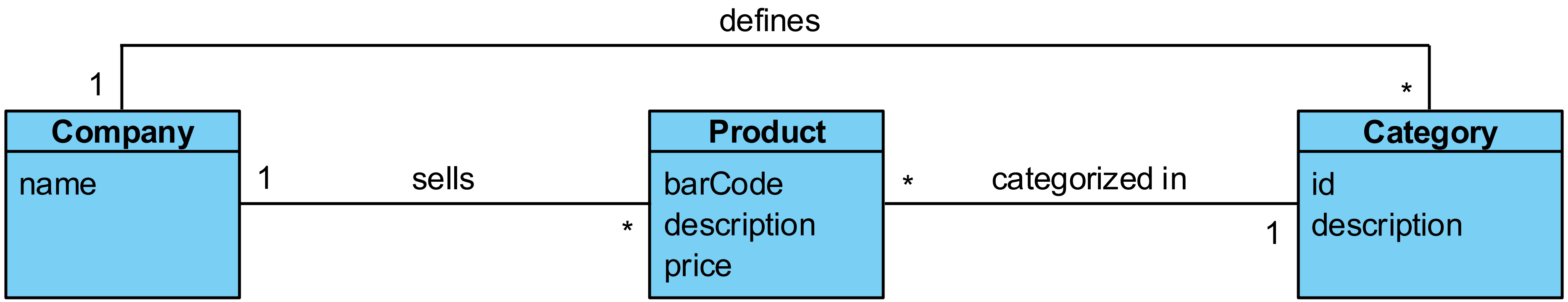
- Guides the assignment of responsibility for creating objects

- Based on finding relationships of (e.g.):
  - Aggregation
  - Composition
  - Registration
  - Using

- Assigning this responsibility to a class establishes a link (coupling) between the two classes

Creator – Example (1/8)

- Which class should be responsible for creating a new (instance of) Category?

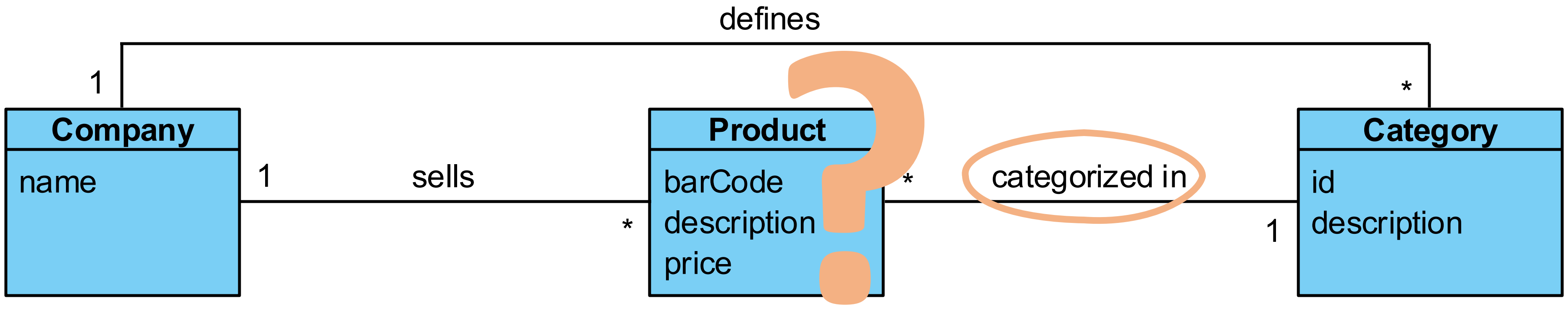


[Creator Rationale] B is responsible for creating A if:

1) B contains or aggregates instances of A
2) B records instances of A
3) B closely uses A
4) B has the data for initializing A

Creator – Example (2/8)

- Which class should be responsible for creating a new (instance of) Category?

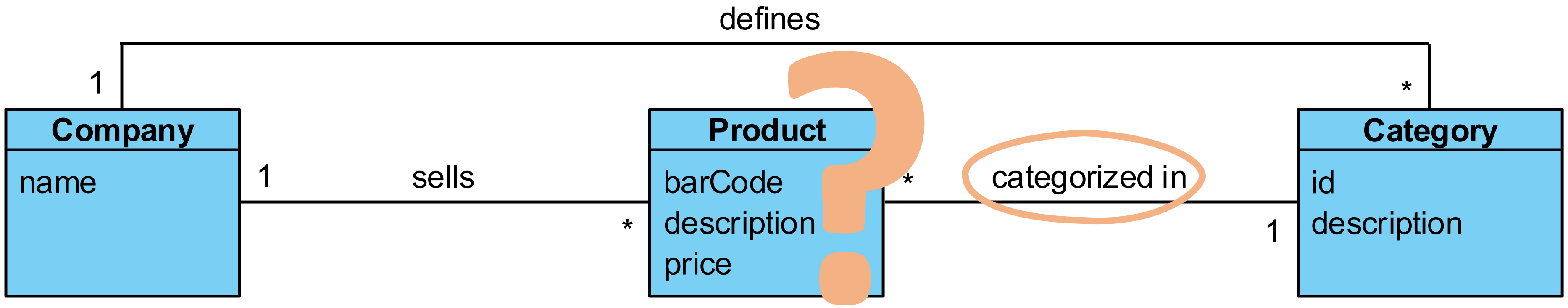


[Creator Rationale] *Product* is responsible for creating *Category* if:

1) *Product* contains or aggregates instances of *Category*
2) *Product* records instances of *Category*
3) *Product* closely uses *Category*
4) *Product* has the data for initializing *Category*

Creator – Example (3/8)

- Which class should be responsible for creating a new (instance of) Category?

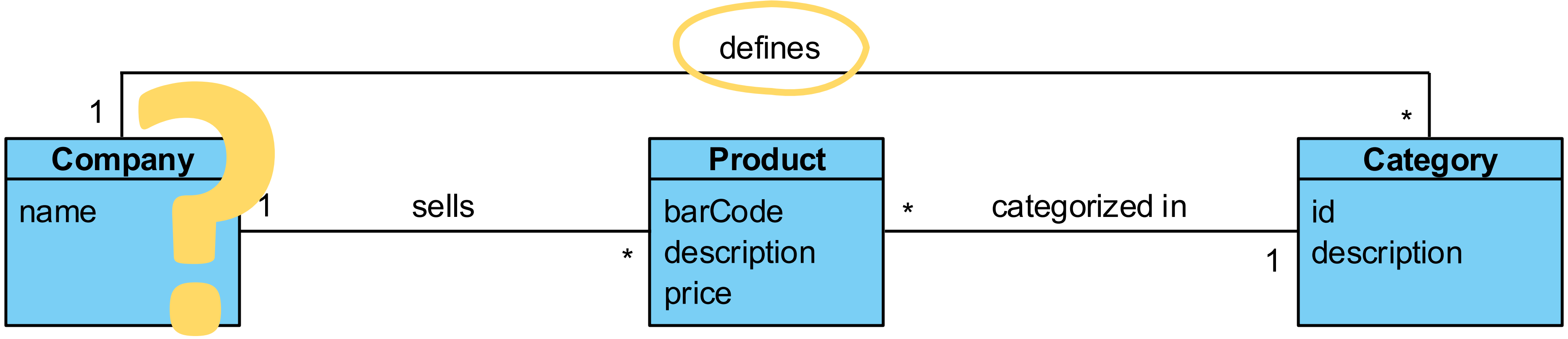


[Creator Rationale] *Product* is responsible for creating *Category* if:

1) ~~*Product* contains or aggregates instances of *Category*~~
2) *Product* records instances of *Category*
3) *Product* closely uses *Category*
4) ~~*Product* has the data for initializing *Category*~~

Creator – Example (4/8)

- Which class should be responsible for creating a new (instance of) Category?

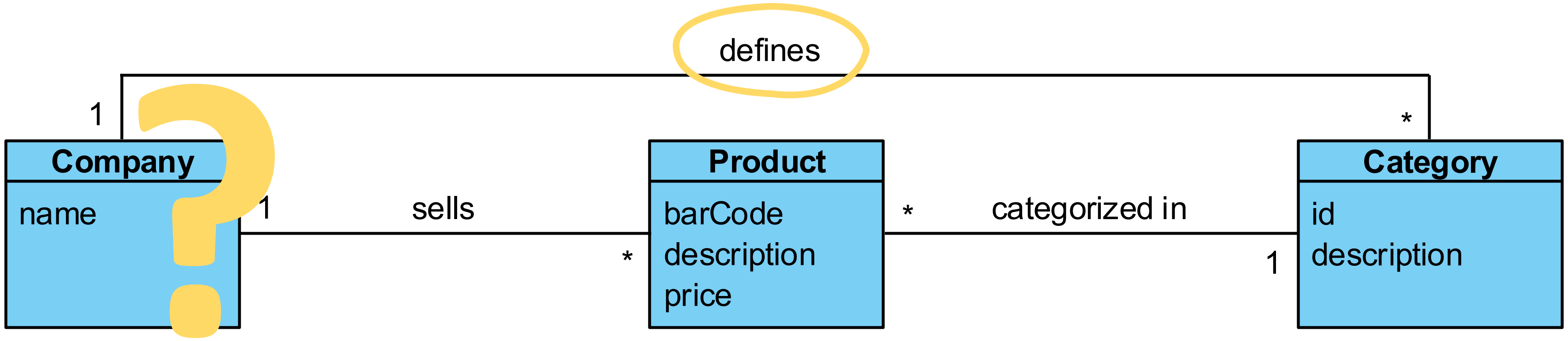


[Creator Rationale] *Company* is responsible for creating *Category* if:

1) *Company* contains or aggregates instances of *Category*
2) *Company* records instances of *Category*
3) *Company* closely uses *Category*
4) *Company* has the data for initializing *Category*

Creator – Example (5/8)

- Which class should be responsible for creating a new (instance of) Category?

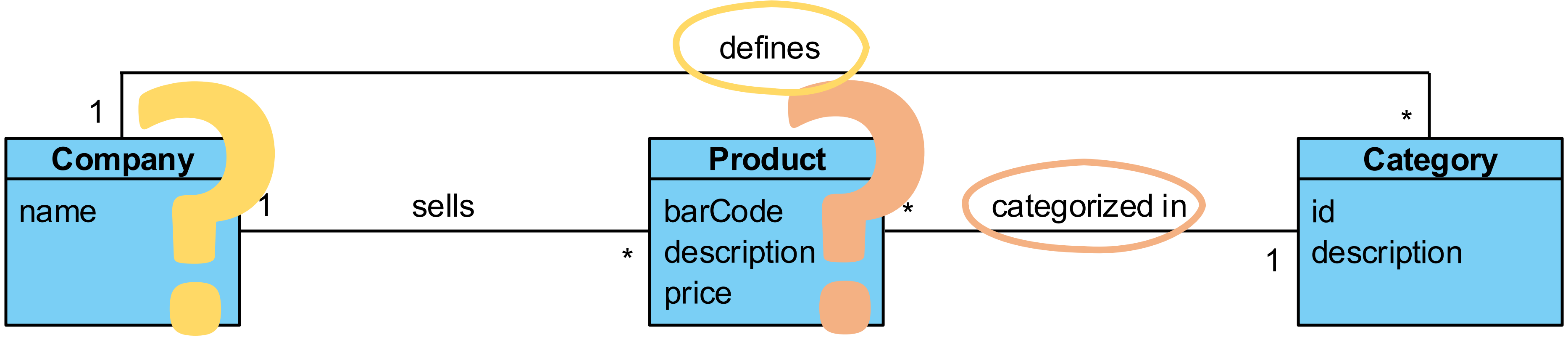


[Creator Rationale] *Company* is responsible for creating *Category* if:

1) ***Company* contains or aggregates instances of *Category***
2) *Company* records instances of *Category*
3) *Company* closely uses *Category*
4) *Company* has the data for initializing *Category*

Creator – Example (6/8)

- Which class should be responsible for creating a new (instance of) Category?



[Creator Rationale] *Company* is responsible for creating *Category* if:

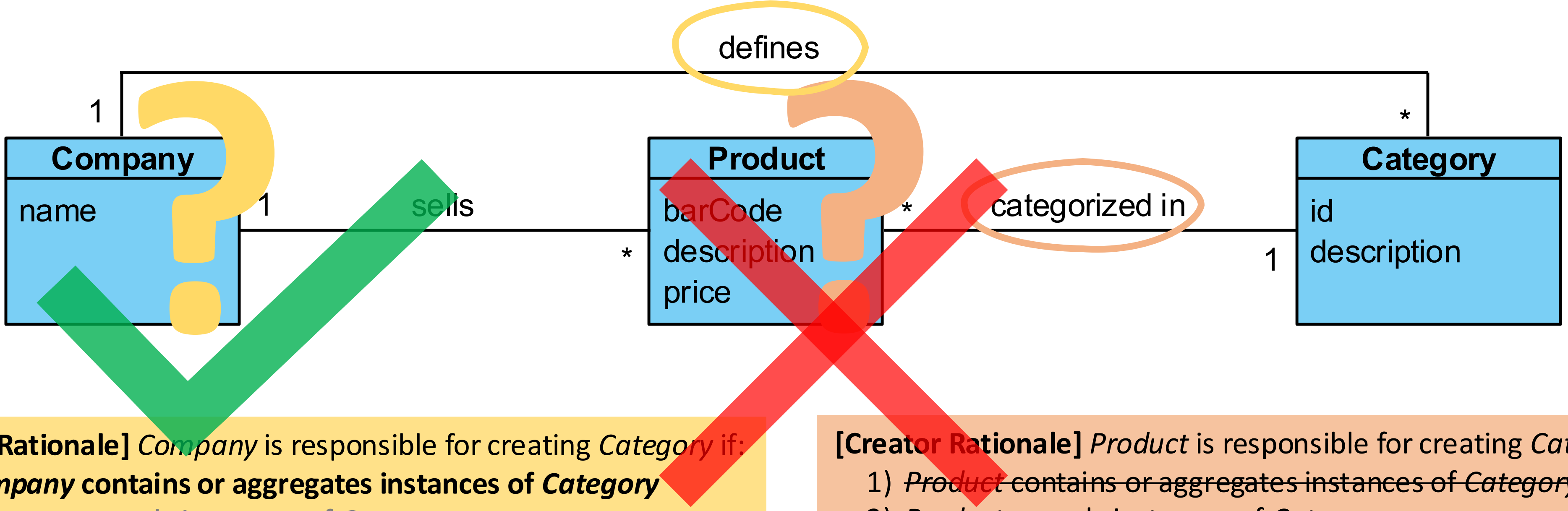
1) ***Company* contains or aggregates instances of *Category***
2) *Company* records instances of *Category*
3) *Company* closely uses *Category*
4) *Company* has the data for initializing *Category*

[Creator Rationale] *Product* is responsible for creating *Category* if:

1) ~~*Product* contains or aggregates instances of *Category*~~
2) *Product* records instances of *Category*
3) *Product* closely uses *Category*
4) ~~*Product* has the data for initializing *Category*~~

Creator – Example (7/8)

- Which class should be responsible for creating a new (instance of) Category?



[Creator Rationale] *Company* is responsible for creating *Category* if:

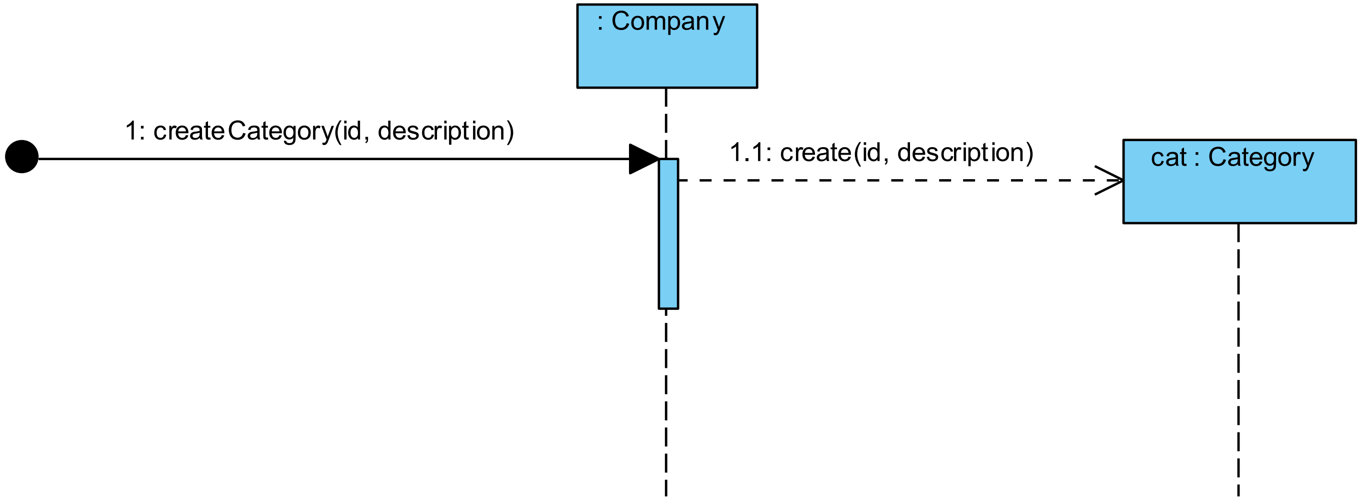
1) ***Company* contains or aggregates instances of *Category***
2) *Company* records instances of *Category*
3) *Company* closely uses *Category*
4) *Company* has the data for initializing *Category*

[Creator Rationale] *Product* is responsible for creating *Category* if:

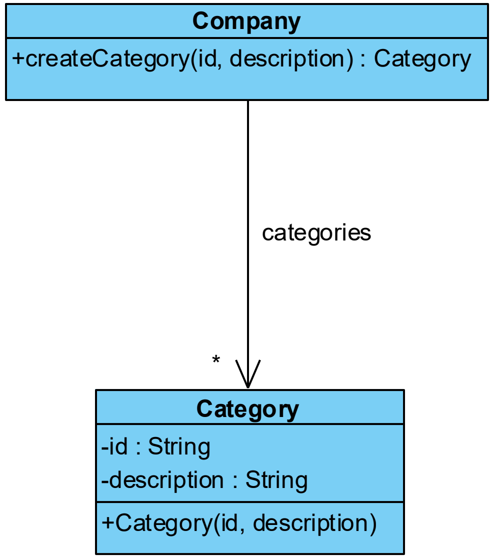
1) ~~*Product* contains or aggregates instances of *Category*~~
2) *Product* records instances of *Category*
3) *Product* closely uses *Category*
4) ~~*Product* has the data for initializing *Category*~~

Creator – Example (8/8)

- Company contains all its categories
- Company must have a function to create categories
- Function: ***createCategory(id, description)***



: Company
1: createCategory(id, description)
1.1: create(id, description)
cat : Category



Company
+createCategory(id, description) : Category
categories
*
Category
-id : String
-description : String
+Category(id, description)

Creator – Contraindications

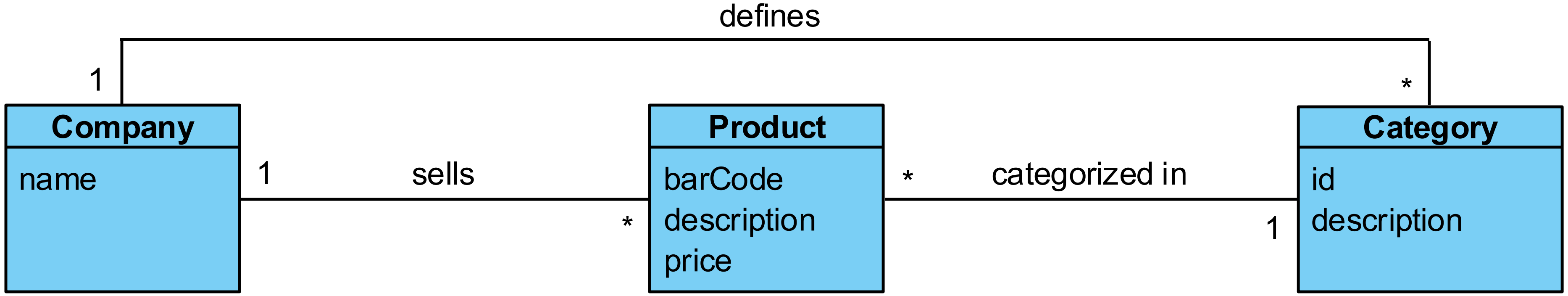
- Object creation can be **complex**
- In such situations, it may be advantageous to **delegate** the instantiation to other classes
- This can be done by applying patterns (not addressed in ESOFT)
  - Abstract Factory
  - Factory Method
  - Builder

Responsibilities

More responsibilities to decide

More responsibilities to decide (1/4)

- Who is responsible for **creating** new instances of Category?
- Who is responsible for **saving** the input data (id and description)?



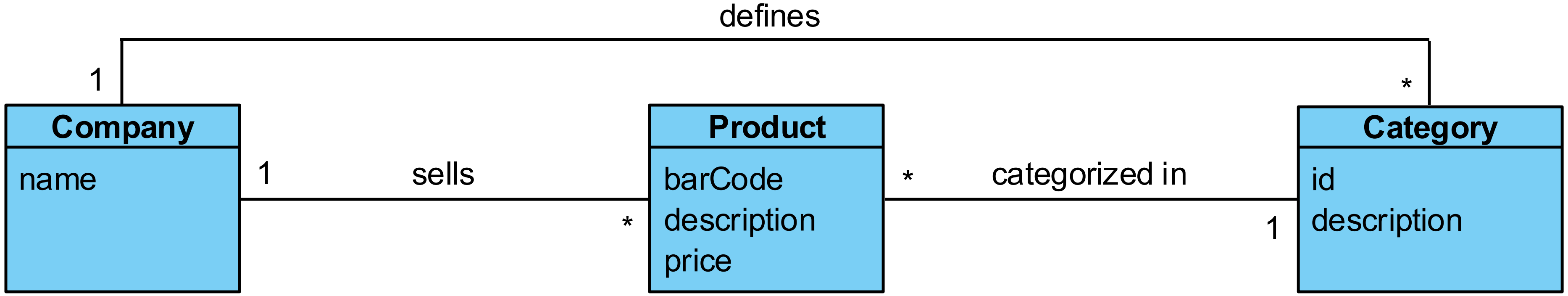
More responsibilities to decide (2/4)

- Who is responsible for **creating** new instances of Category?

  → **Company**, by applying the **Creator** pattern (cf. previous slides)

- Who is responsible for **saving** the input data (id and description)?

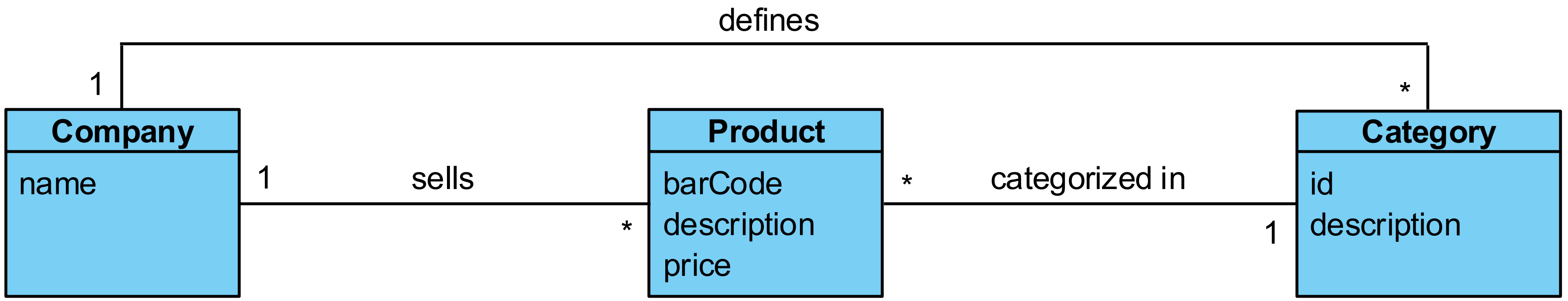
  → **Category** (the object being created), by applying the **Information Expert (IE)** pattern since such data are the attribute values of the object



More responsibilities to decide (3/4)

- Who is responsible for **validating** the object being created?

- Who is responsible for **saving** the newly created object?



More responsibilities to decide (4/4)

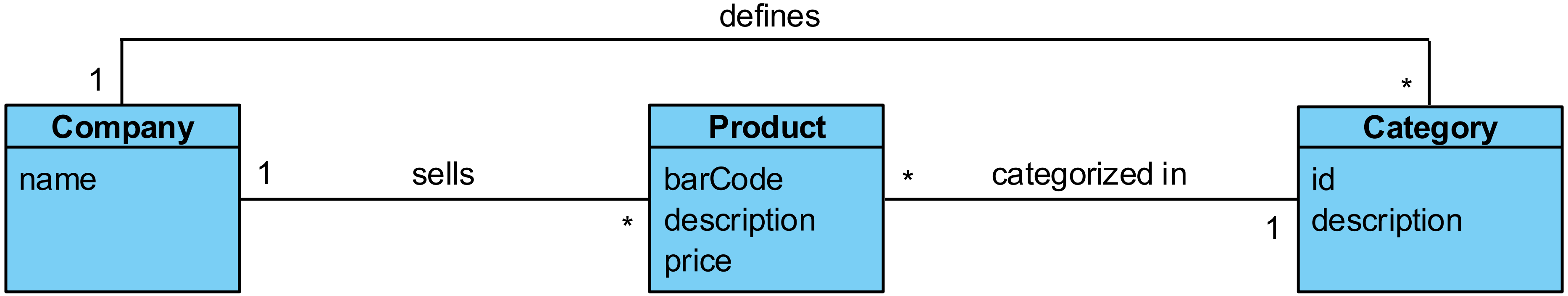
- Who is responsible for **validating** the object being created?

  → **Category** (the object itself), by applying **IE** since the object knows its own data and, therefore, can ensure the applicable local business rules (e.g. mandatory attributes, invalid characters) **— LOCAL VALIDATION**

  → **Company**, by applying **IE** since it knows all existing category objects and, therefore, can ensure applicable global business rules (e.g. avoiding duplicate objects) **— GLOBAL VALIDATION**

- Who is responsible for **saving** the newly created object?

  → **Company**, by applying **IE** since company contains all the (defined) categories (check the domain model)



Summary

- In the context of OO Design, assignment of responsibilities must comply with consolidated principles, patterns and best practices
- GRASP is an OO responsibility assignment guide, which promotes:
  - Modularity
  - Reusability
  - Maintainability
- GRASP principles are combined with each other
- Many other patterns are based on GRASP

Bibliography

- Fowler, M. (2003). UML Distilled (3rd ed.). Addison-Wesley. ISBN: 978-0-321-19368-1
- Freeman, E., & Robson., E. (2021). Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software (2nd ed.). O'Reilly. ISBN: 978-1-492-07800-5
- Larman, C. (2004). Applying UML and Patterns (3rd ed.). Prentice Hall. ISBN: 978-0-131-48906-6
- GRASP Explained. Available on: https://www.kamilgrzybek.com/blog/posts/grasp-explained