Cohesion and Coupling

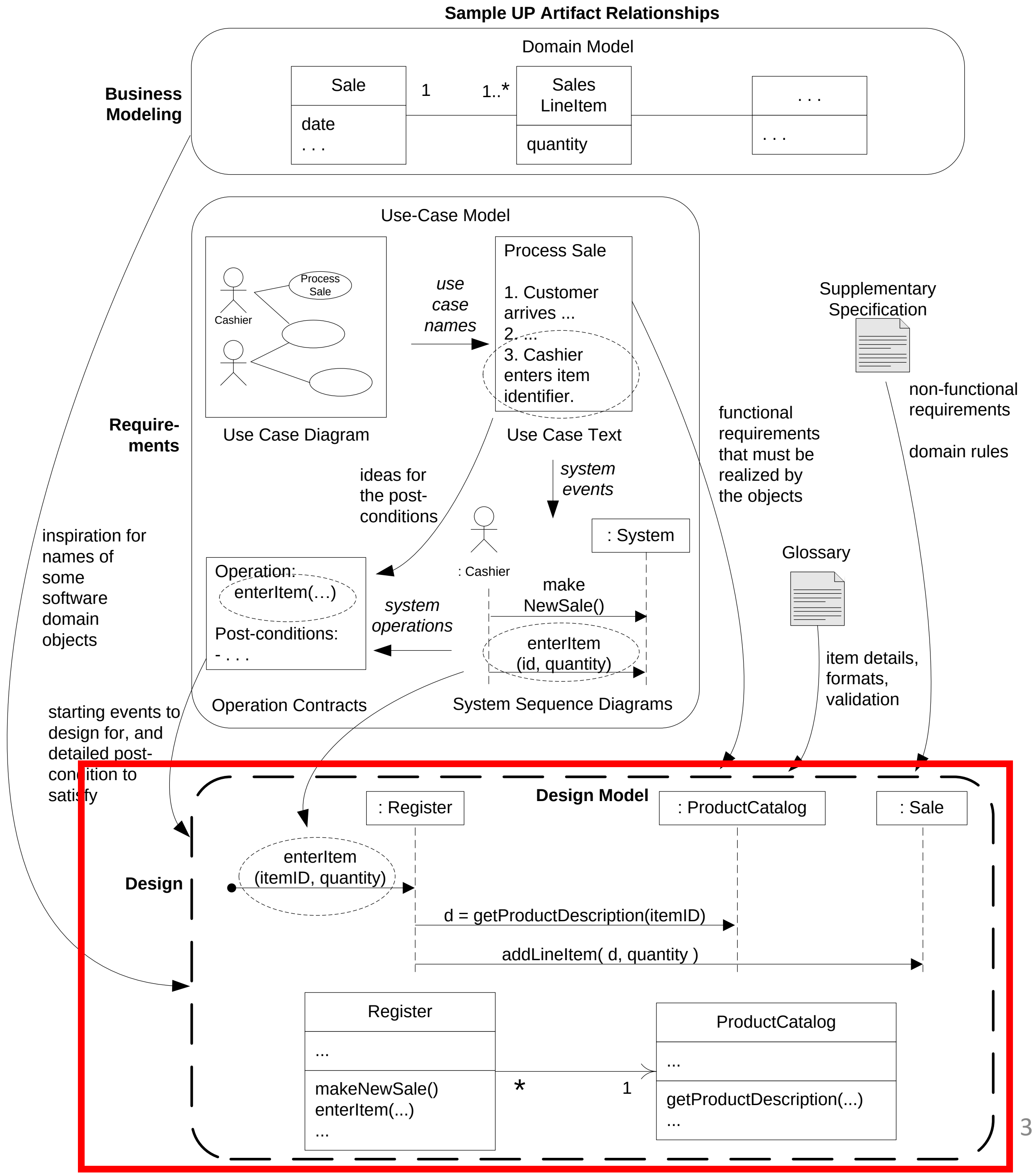
UPskill

Digital Skills & Jobs

Topics

- Modularity
- Cohesion
- GRASP: High Cohesion
- Coupling
- GRASP: Low Coupling
- Types of coupling

Artifacts Overview



GRASP - General Responsibility Assignment Software Patterns (or Principles)

Recall previous presentations

- 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 presentation

Modularity

Modularity

- "Modularity is the property of a system that has been decomposed into a set of **cohesive and loosely coupled** modules" [*Booch, 1994*]

- It is one of the most classic principles of software development

- It consist of **decomposing a product into smaller parts** (or modules) with clear responsibilities
    - SW System → Applications → Layers → Components → Classes
    - Layer examples: Presentation/UI layer, Domain layer

Poor/Bad Design → Low Modularity

- Rigidity
  - It is difficult to change because each change affects too many parts of the system
- Fragility
  - When a change is made, failures are (very) hard to predict
- Immobility
  - Difficult to reuse in other applications because it is difficult to disconnect from the original application
- **High Cohesion and Low Coupling promote modularity**

Cohesion

Cohesion (1/2)

- It is a measure regarding the **coherence of the responsibilities** assigned to an element of the system. E.g.:
  - **Classes (of software)**
  - Components
  - Modules
  - Applications

Not addressed in this course

- Typically, it is measured in:
  - High Cohesion → to be achieved
  - Low Cohesion → to be avoided

Cohesion (2/2)

- A class with High Cohesion
  - Has a relatively small number of operations
  - The operations are closely related to each other
  - Delegate or collaborate with other classes to perform more complex tasks

- A class with Low Cohesion
  - Is difficult to understand
  - Is difficult to reuse
  - Is difficult to maintain

GRASP

High Cohesion (HC)

High Cohesion

- **Problem**
  - How to maintain classes/objects with coherent and easy-to-understand functionalities?

- **Solution**
  - Assign responsibilities so that cohesion remains high
  - Features should be strongly related with each other
  - Prevent the same class/object from doing many different things
  - Cooperate with other classes
    - Tell other classes to do something about data they know
    - Do not ask other classes for data (avoid *getX* methods)
  - Delegate other responsibilities to other classes

Tell, Don’t Ask Principle

Tell, Don’t Ask Principle

- Principle
  - **You should not ask** an object for its own data (*state*) and further act on that data to make some decisions
  - Instead, **you should tell** an object what to do, i.e. send commands to it
- Advantages
  - Promotes a clear separation of responsibilities
  - **Favors High Cohesion**
  - The solution becomes:
    - Easier to understand
    - Easier to maintain
    - Flexible enough to add new features
- Similar to Information Expert

Benefits of High Cohesion

- Greater **design clarity** and easier understanding
- **Maintenance** and improvements become simplified
- **Reuse** is facilitated because a class with high cohesion can be used for a clear specific purpose

- The higher the degree of cohesion, the better the quality of the software

Coupling

Coupling (1/2)

- It is a measure of **how strongly an element** is connected to, or has knowledge of, or **is dependent on other elements** of the system. E.g.:
  - **Classes (of software)**
  - Components
  - Modules
  - Applications

Not addressed in this course

- Typically, it is measured in:
  - Low Coupling → to be achieved
  - High Coupling → to be avoided

Coupling (2/2)

- A class with Low Coupling
  - Depends on few or no classes
  - Easy to understand
  - Easy to reuse
  - Easy to maintain

- A class with High Coupling
  - Depends on (many) other classes
  - Difficult to understand in isolation
  - Often needs to be changed by changes in related classes
  - More difficult to reuse

GRASP

Low Coupling (LC)

Low Coupling

- **Problem**
  - How to achieve low dependency, low impact on changes and increased reuse between classes/objects?

- **Solution**
  - Assign responsibilities to maintain a low coupling
  - Avoid unnecessary dependencies
  - Apply indirection mechanisms (Indirection Pattern) to assign the responsibility of mediation between two classes/objects to an intermediate class, thus ensuring decoupling (e.g. the controller classes play this role)

Benefits of Low Coupling

- It promotes **independence**, **modularity**, and **flexibility** of the code
- Classes are **simpler to understand** in isolation

- The lower the degree of coupling, the better the quality of the software.

Types of Coupling

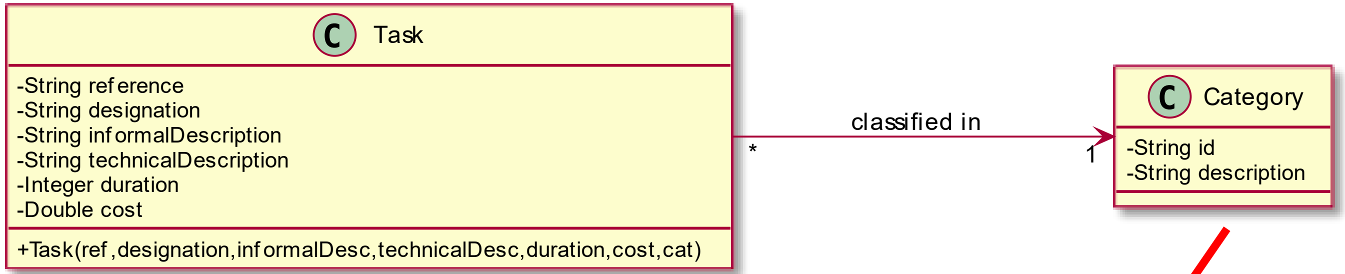
Types of Coupling

- An OO language, includes the following types of coupling:
  - **Loose Coupling**
    1. *TypeX* has an association to a *TypeY* object ***(Association: Aggregation / Composition)***
    2. *TypeX* has a method that references a *TypeY* object ***(Knowledge)***
  - **Medium Coupling**
    3. *TypeX* calls methods of a *TypeY* object ***(Method)***
    4. *TypeX* implements a *TypeY* interface ***(Implementation)***
  - **Strong Coupling**
    5. *TypeX* is (directly or indirectly) a *TypeY* subclass ***(Extension by Inheritance)***

- Each type of coupling has its own particularities and strengths
- Two classes can have several of these forms

1. *TypeX* has an association to a *TypeY* object

Loose coupling!



Task

-String reference
-String designation
-String informalDescription
-String technicalDescription
-Integer duration
-Double cost

+Task(ref,designation,informalDesc,technicalDesc,duration,cost,cat)

classified in

*

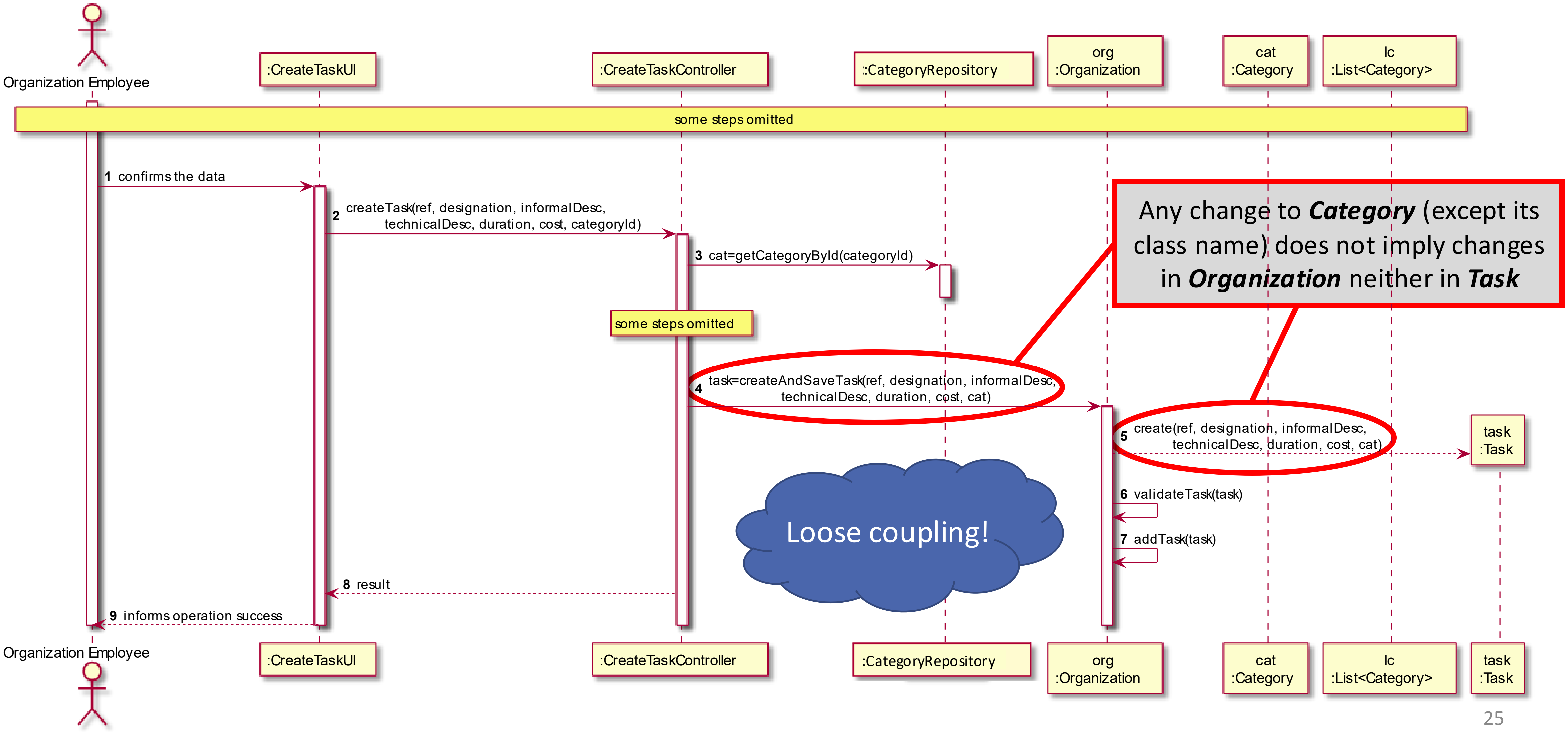
1

Category

-String id
-String description

Any change to ***Category*** (except its class name) does not imply any changes in ***Task***

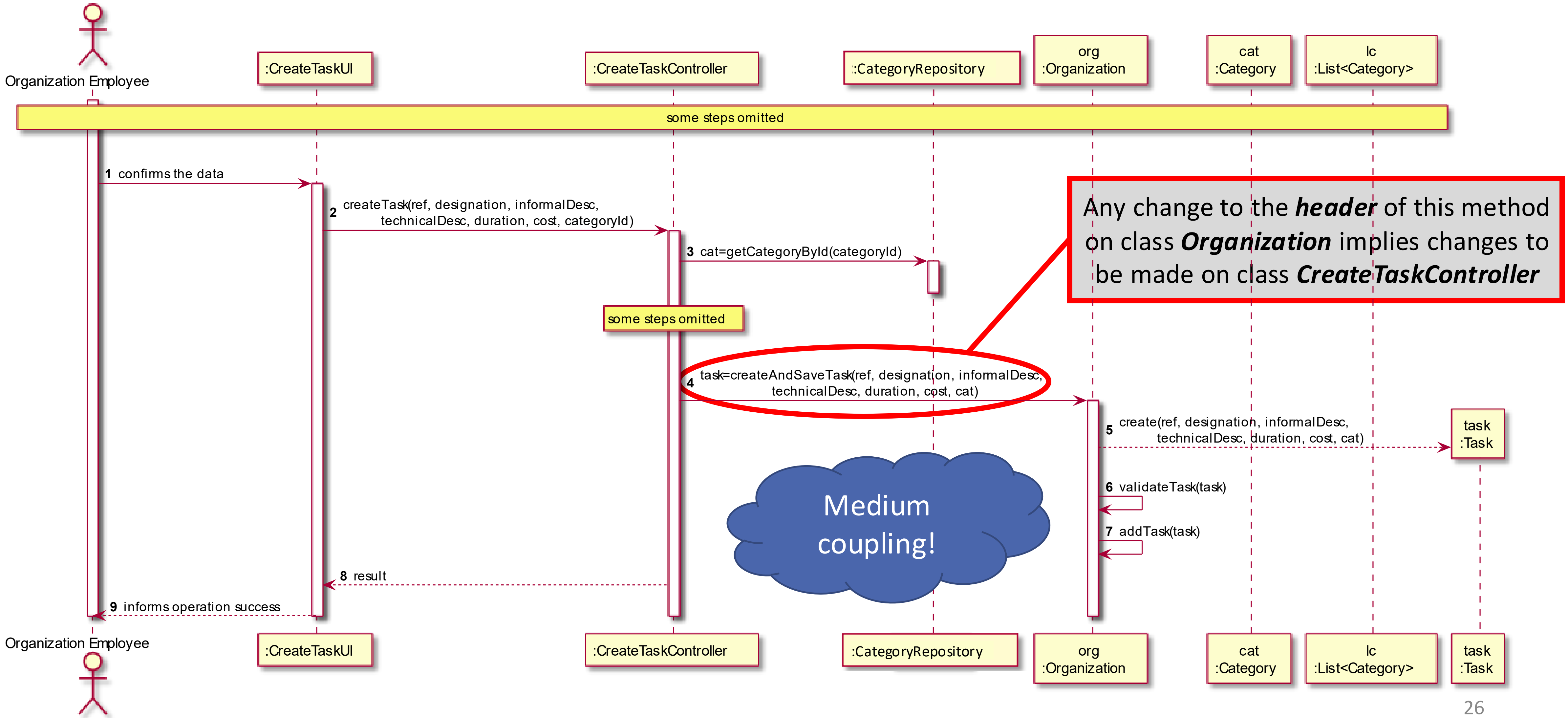
2. *TypeX* has a method that references a *TypeY* object



Any change to ***Category*** (except its class name) does not imply changes in ***Organization*** neither in ***Task***

Loose coupling!

3. *TypeX* calls methods of a *TypeY* object



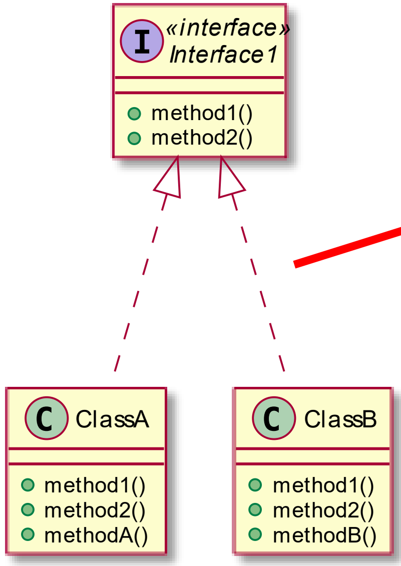
Any change to the ***header*** of this method on class ***Organization*** implies changes to be made on class ***CreateTaskController***

Medium coupling!

4. *TypeX* implements a *TypeY* interface

- The implementation mechanism establishes a contract between a class and the code that uses it
  - The interface describes what any class implementing the interface must do
  - E.g.: ClassA and ClassB must implement both `method1()` and `method2()`
- Form of polymorphism with a weaker coupling than with classes
- Verifiable at compile time

Medium coupling!



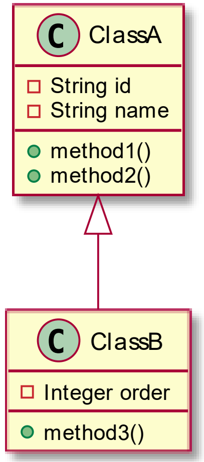
implements

```
ClassA implements Interface1
ClassB implements Interface1
```

5. *TypeX* is (directly or indirectly) a *TypeY* subclass

- The subclass inherits all the public and protected members (attributes, operations and relations) from its superclass
  - New members can be added to the subclass
  - Existing members can be specialized by the subclass
- All instances of the subclass are also instances of the superclass
- Not all instances of the superclass are instances of the subclass

Strong coupling!



extends

`ClassB` is a subclass of **`ClassA`**
`ClassA` is a superclass of **`ClassB`**

Summary (1/2)

- A subclass is strongly coupled to its superclass
  - The generalization/specialization between classes must be carefully analyzed
  - Favor *implements* over *extends*

- “You should code to interfaces, not implementations.”

- Generic classes, which are highly reusable, have an even lower coupling

- Usually, high coupling with stable and widely used elements is not a problem (e.g. Java Libraries)

Summary (2/2)

- Combine High Cohesion and Low Coupling with other GRASP patterns to assign responsibilities to objects

- Evaluate design alternatives using High Cohesion and Low Coupling

- Adopt design alternatives favoring
  - Modularity
  - Reusability
  - Maintainability

Coupling
Cohesion

References & Bibliography

- Larman, Craig; Applying UML and Patterns; Prentice Hall (3rd ed.); ISBN 978-0131489066
- Booch, G. 1994. Object-Oriented Analysis and Design. Redwood City, CA.: Benjamin/Cummings.
- Fowler, Martin; Patterns of Enterprise Application Architecture; Addison Wesley; ISBN-13: 978-0321127426