Cohesion and Coupling

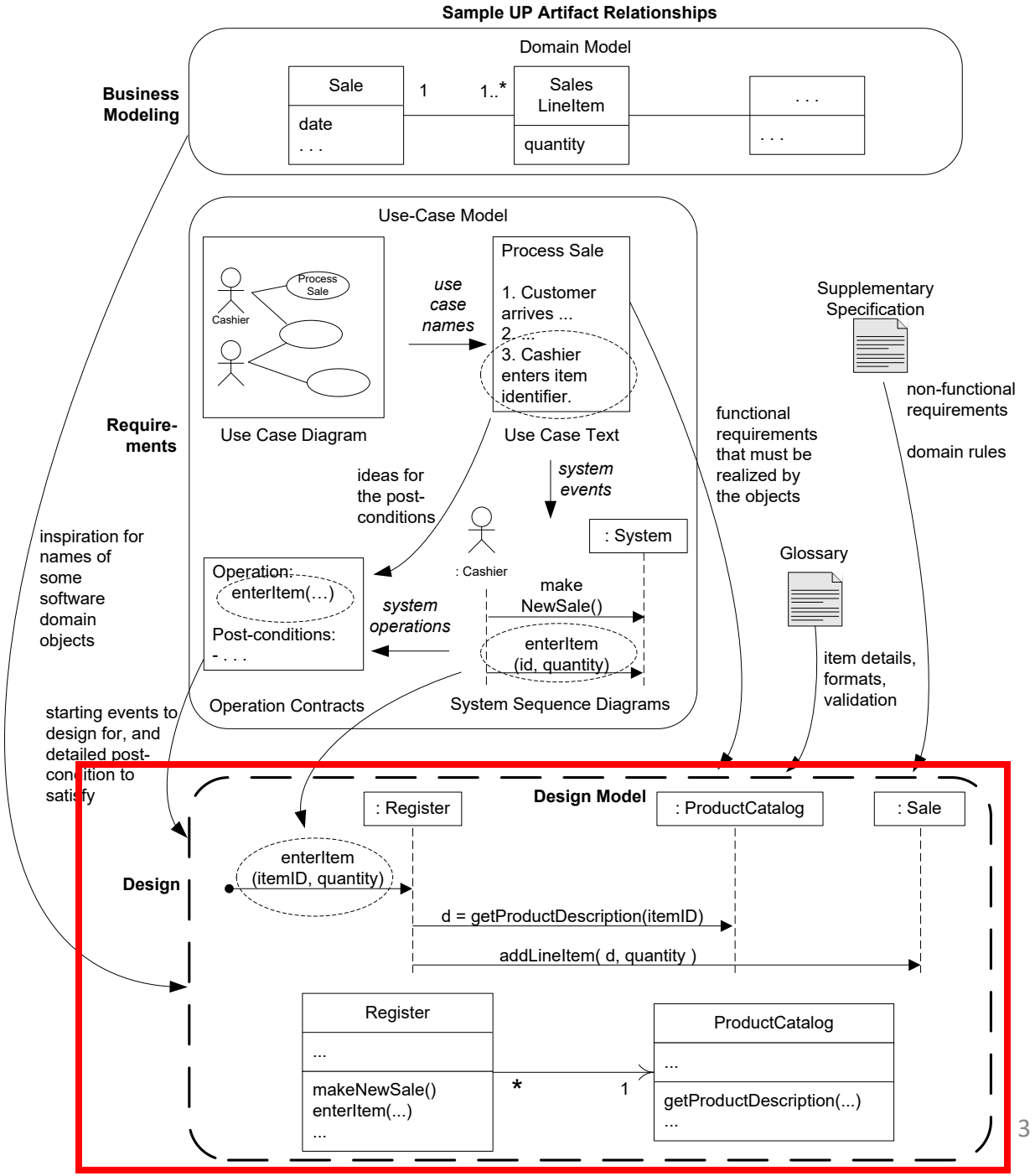
Promotion of Container Objects to Software Classes

ISEP / LETI / ESOFT

Topics

- Modularity
- Cohesion
- GRASP: High Cohesion
- Coupling
- GRASP: Low Coupling
- Types of coupling
- Promotion of Container Objects to Software Classes

Artifacts Overview



GRASP - General Responsibility Assignment Software Patterns (or Principles)

Recall previous lectures

- 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

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

  Slightly addressed in ESOFT

- 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* functions)

    } Tell, Don't Ask Principle
  - Delegate other responsibilities to other classes

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

  Slightly addressed in ESOFT

- 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

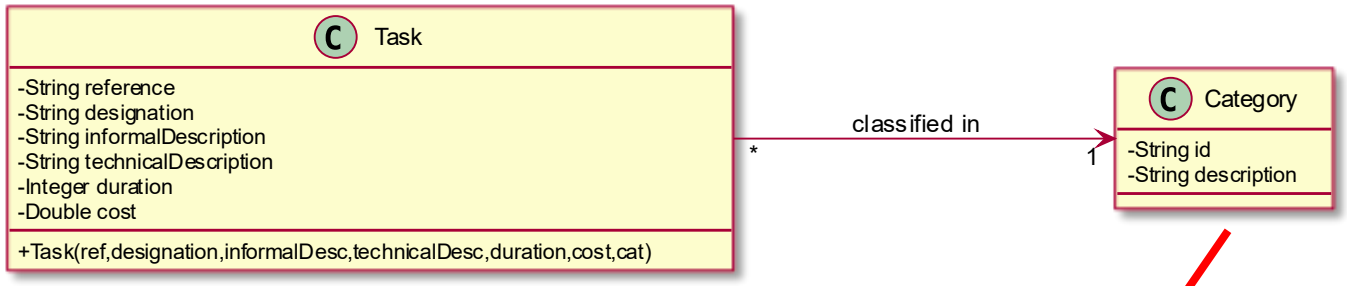
Types of Coupling

- 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 function that references a *TypeY* object ***(Knowledge)***
  - **Medium Coupling**
    3. *TypeX* calls functions of a *TypeY* object ***(Function)***
    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

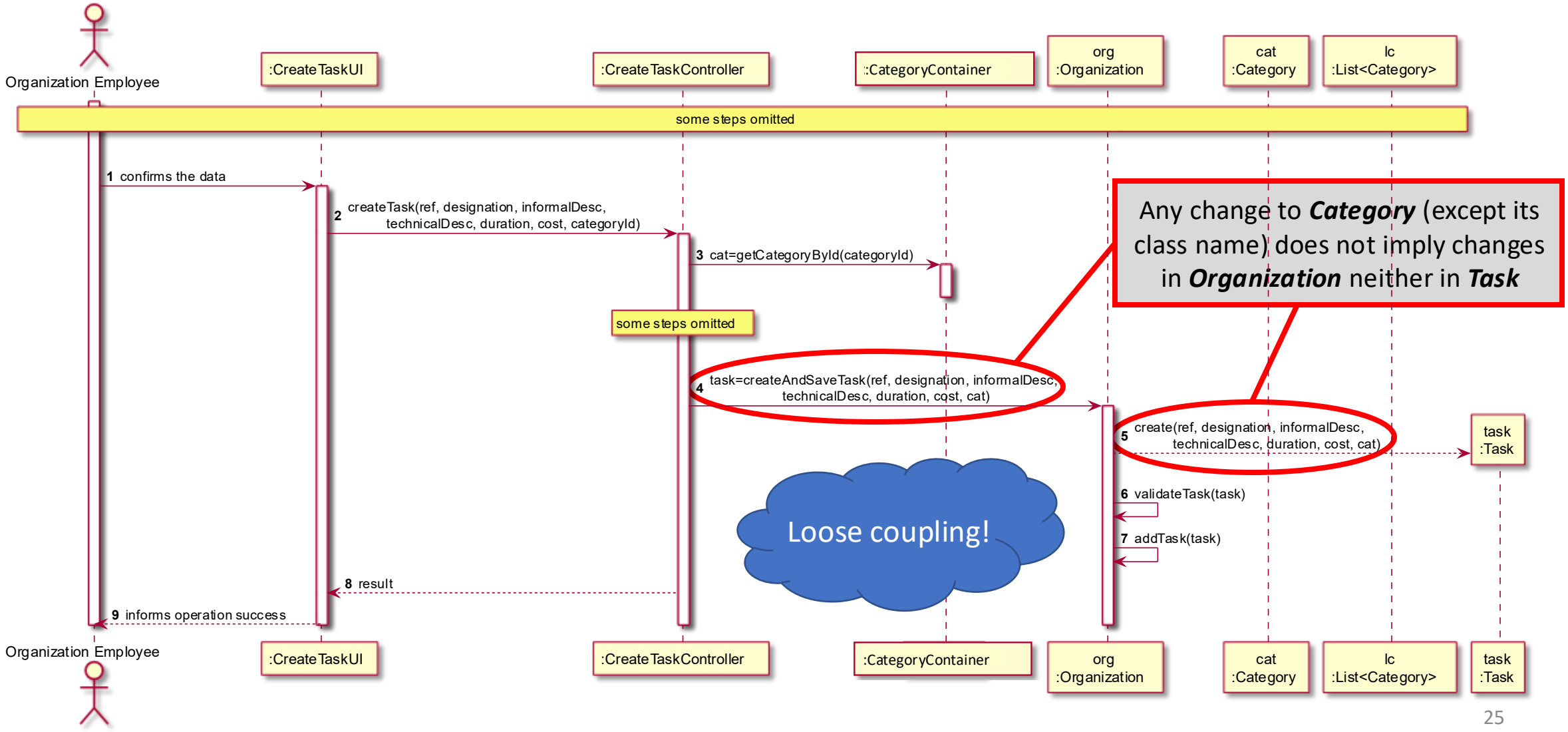
1. *TypeX* has an association to a *TypeY* object

Loose coupling!



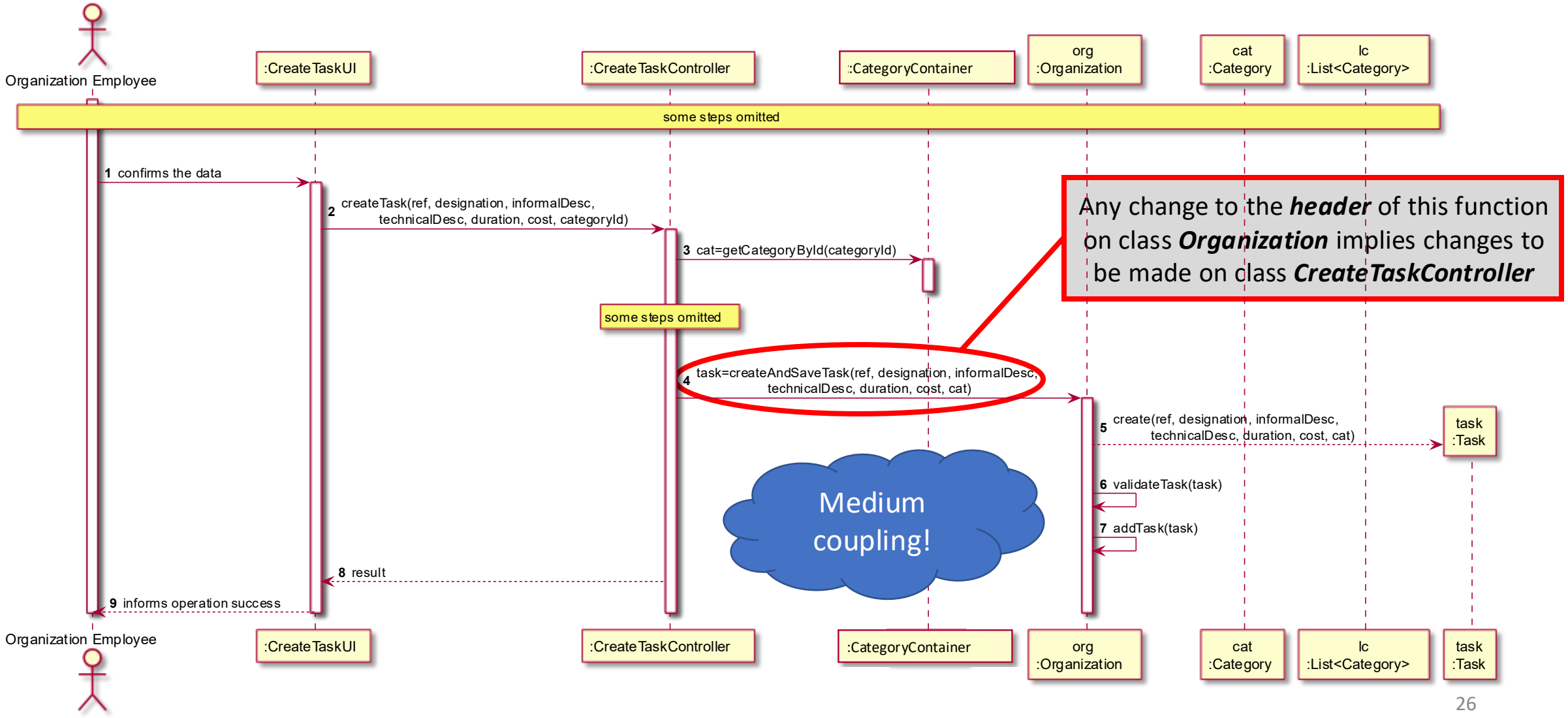
Any change to ***Category*** (except its class name) does not imply any changes in ***Task***

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



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

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



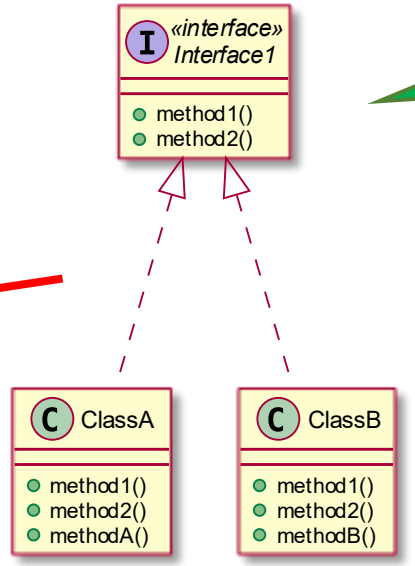
Any change to the ***header*** of this function on class ***Organization*** implies changes to be made on class ***CreateTaskController***

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

implements

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

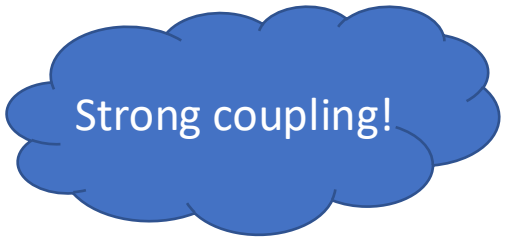


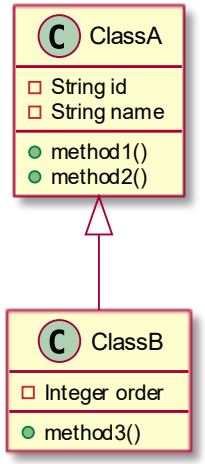
Interfaces are implemented in C++ by means of **Abstract Classes** whose functions are all **pure virtual**.

Medium coupling!

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





extends

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

Promotion of Container Objects to Software Classes

by applying High Cohesion, Low Coupling and Pure Fabrication

Motivating the Problem – Part I

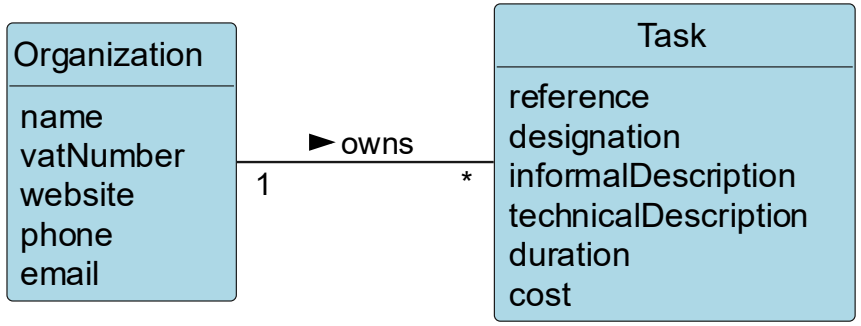
UC010 – List Organization Tasks

UC010 – List Organization Tasks

Platform for Outsourcing Tasks

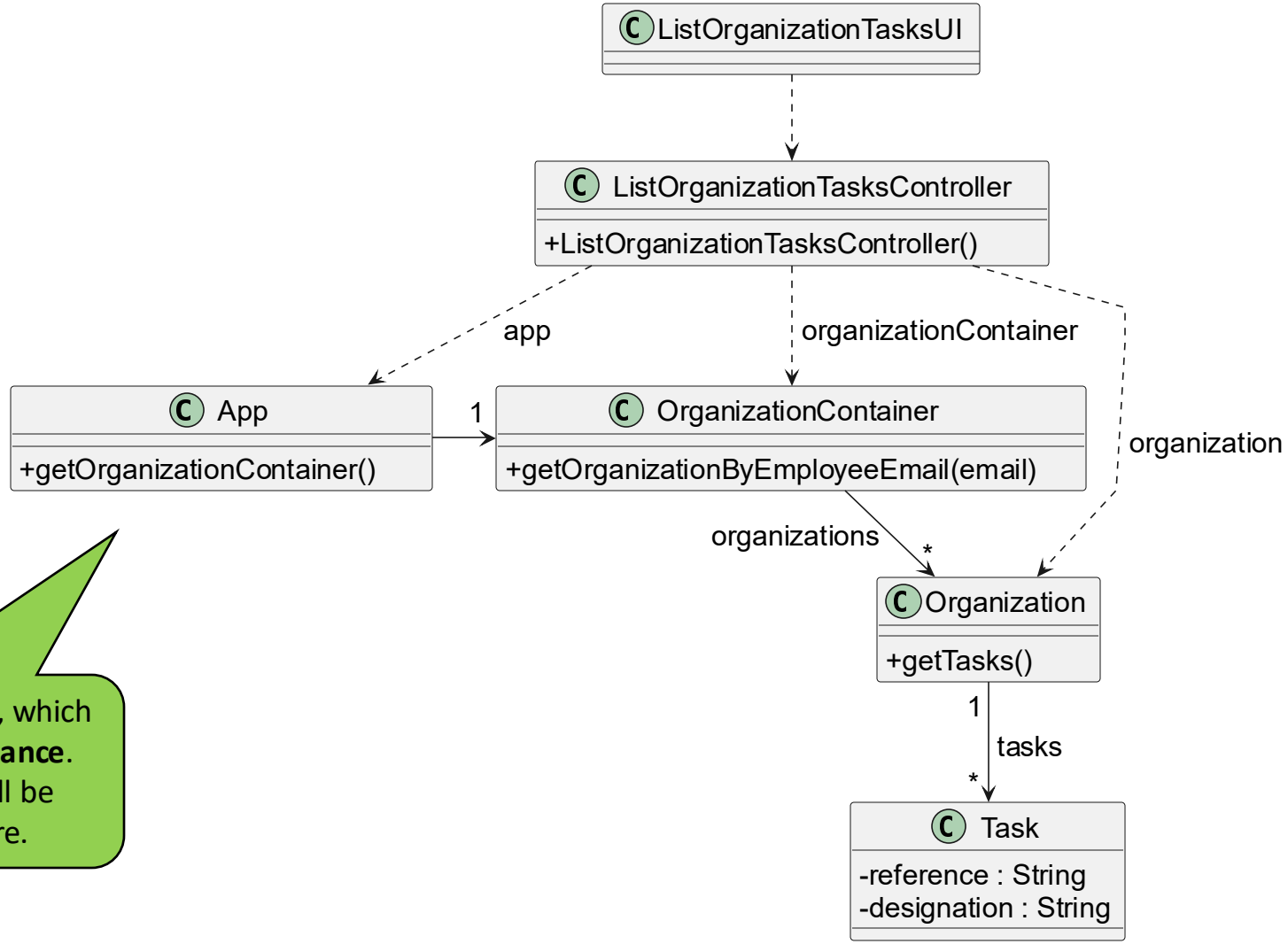
- As an Organization Employee, I want to list all tasks created by my organization.
  - AC1: The tasks must be sorted alphabetically by their designation.

- Relevant Domain Model excerpt



UC010 – Partial Class Diagram

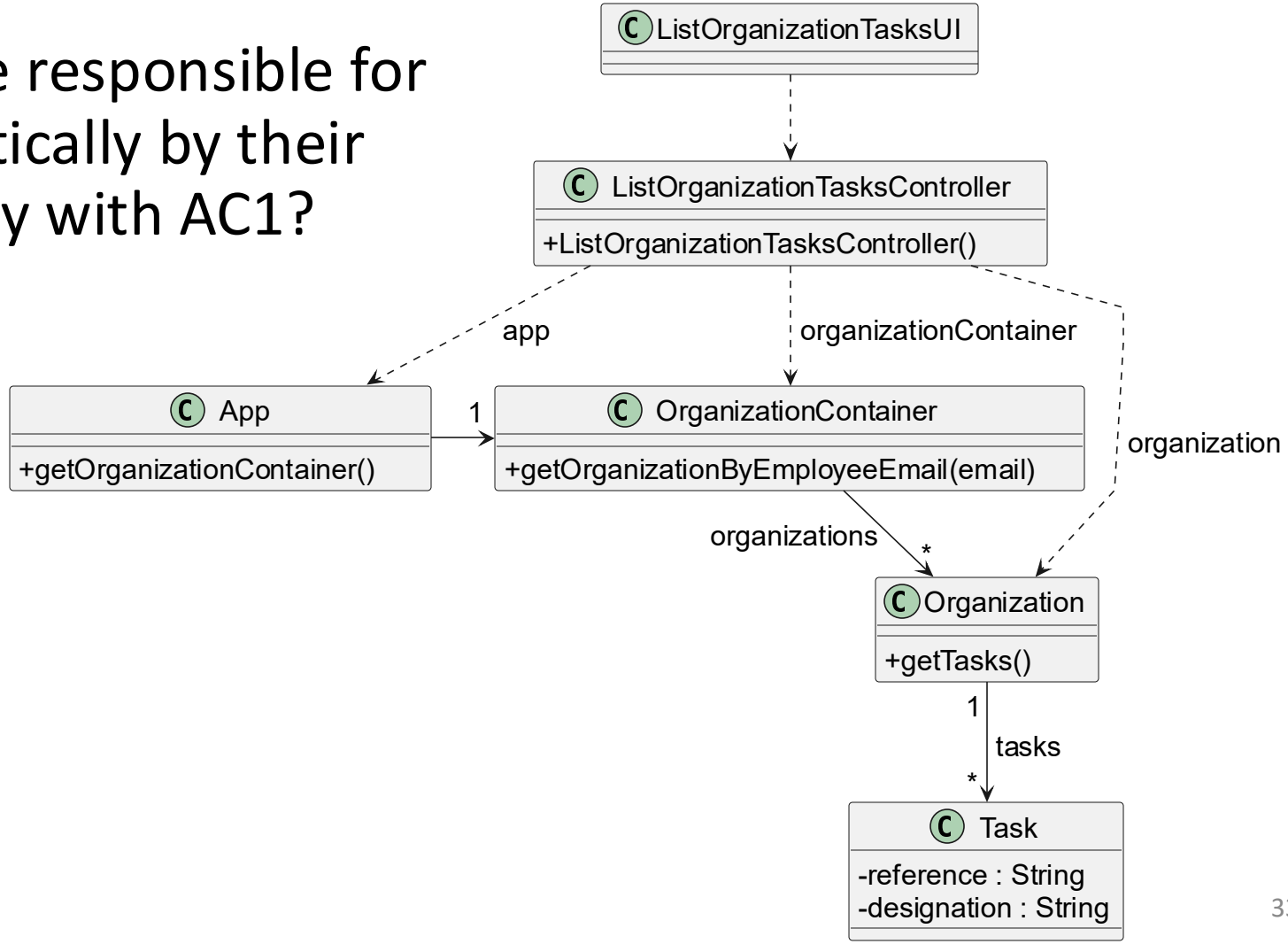
Platform for Outsourcing Tasks



The App class is a **singleton**, which means it has **only one instance**. The Singleton pattern will be address in a next lecture.

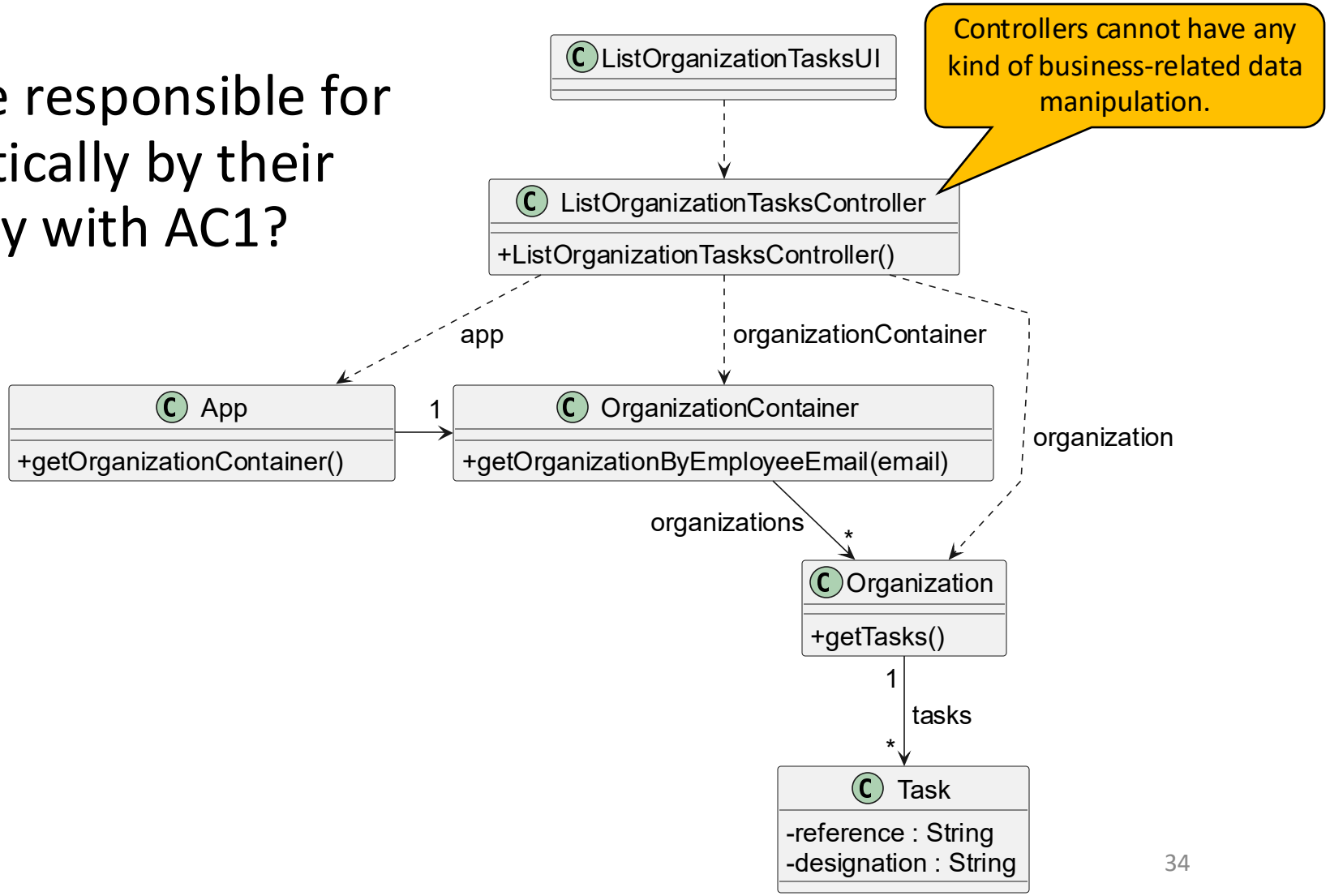
Complying with AC1: Sorting Tasks (1/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - UI?
  - Controller?
  - Organization?
  - Task?



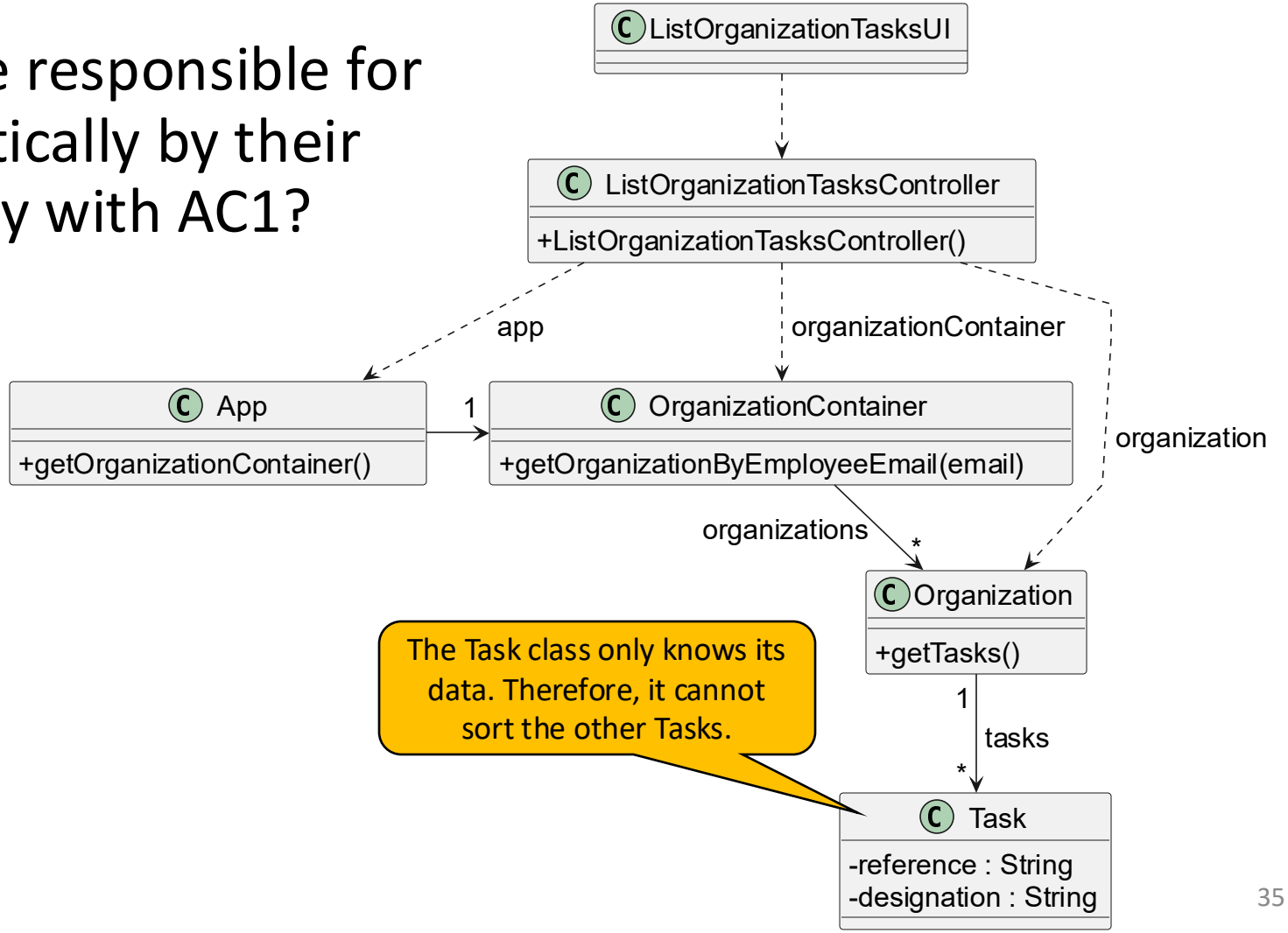
Complying with AC1: Sorting Tasks (2/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - UI?
  - ~~Controller?~~
  - Organization?
  - Task?



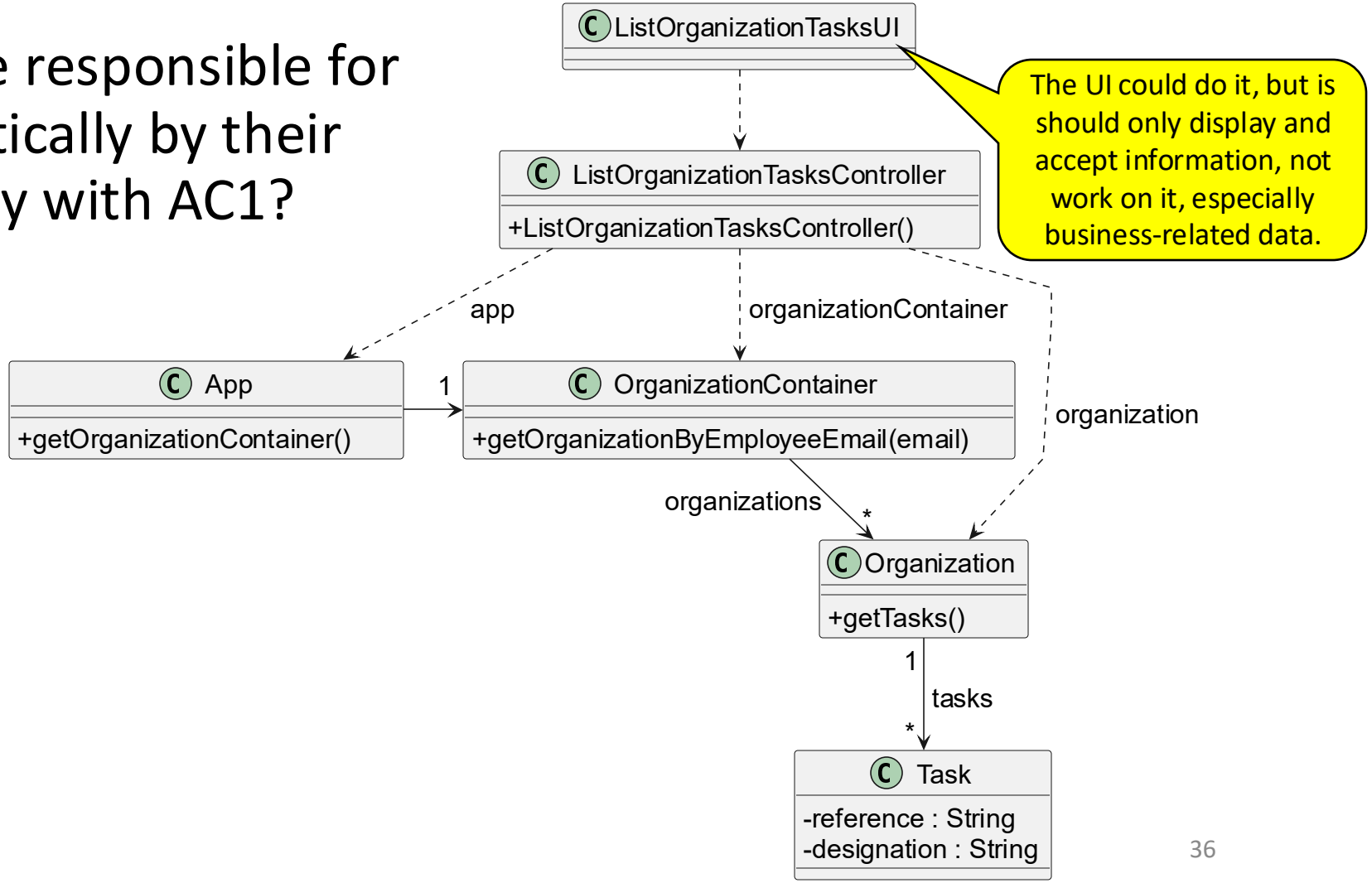
Complying with AC1: Sorting Tasks (3/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - UI?
  - ~~Controller?~~
  - Organization?
  - ~~Task?~~



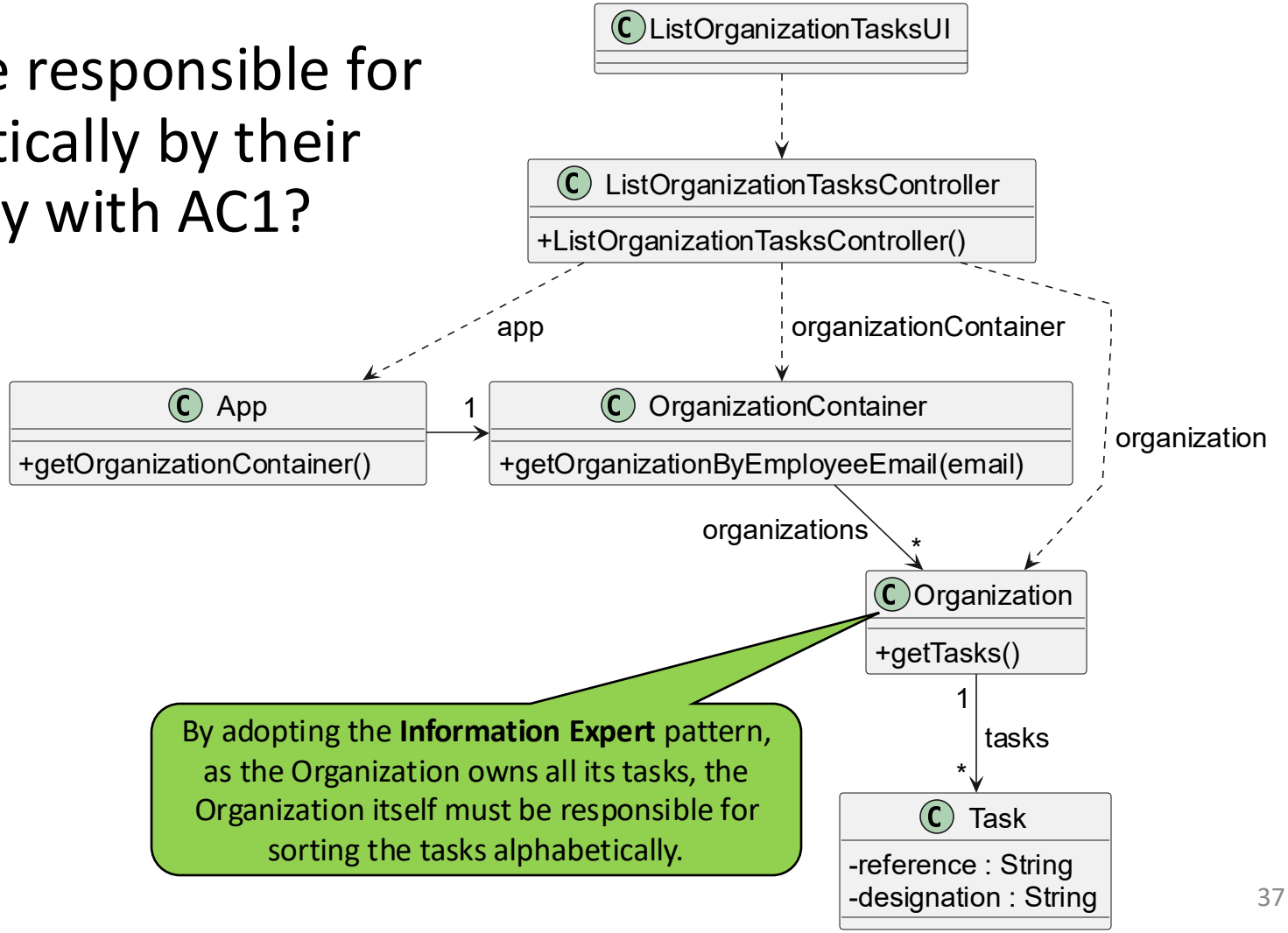
Complying with AC1: Sorting Tasks (4/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - ~~UI?~~
  - ~~Controller?~~
  - Organization?
  - ~~Task?~~



Complying with AC1: Sorting Tasks (5/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - ~~UI?~~
  - ~~Controller?~~
  - Organization? ✔
  - ~~Task?~~



UC010 – AC1 Rationale (1/2)

- To comply with AC1 and according to Information Expert (IE), **the Organization class should be responsible for sorting the Tasks** because it holds the Tasks for the Organization
- The Organization class should implement the **getSortedTasks()** function
- While this makes sense, as a side effect, this approach:
  - **Increases Coupling** between Organization and Task
    - Remember the **"Tell, Don't Ask"** principle – how can the Organization have access to the **designation** attribute for comparison without a **getDesignation()** function in the Task class?
  - **Reduces Cohesion** of the Organization class
    - The Organization must not only hold the created Tasks, but also have the responsibility for sorting those Tasks

UC010 – AC1 Rationale (2/2)

Is this the best approach?
Let's consider another
Use Case…

- To comply with AC1 and according to Information Expert (IE), **the Organization class should be responsible for sorting the Tasks** because it holds the Tasks for the Organization
- The Organization class should implement the **getSortedTasks()** function
- While this makes sense, as a side effect, this approach:
  - **Increases Coupling** between Organization and Task
    - Remember the **"Tell, Don't Ask"** principle – how can the Organization have access to the **designation** attribute for comparison without a **getDesignation()** function in the Task class?
  - **Reduces Cohesion** of the Organization class
    - The Organization must not only hold the created Tasks, but also have the responsibility for sorting those Tasks

Motivating the Problem - Part II

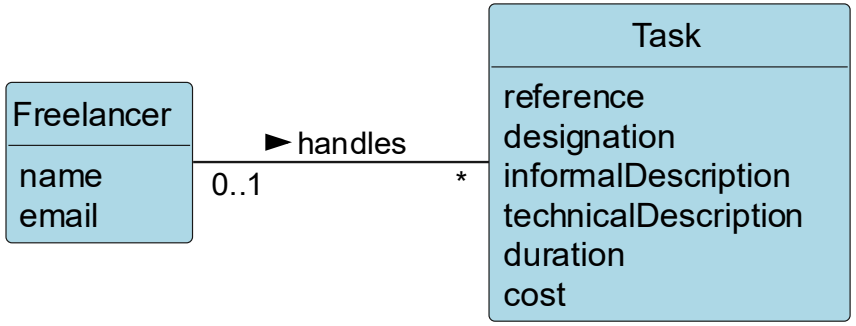
UC011 – List Freelancer Tasks

UC011 – List Freelancer Tasks

Platform for Outsourcing Tasks

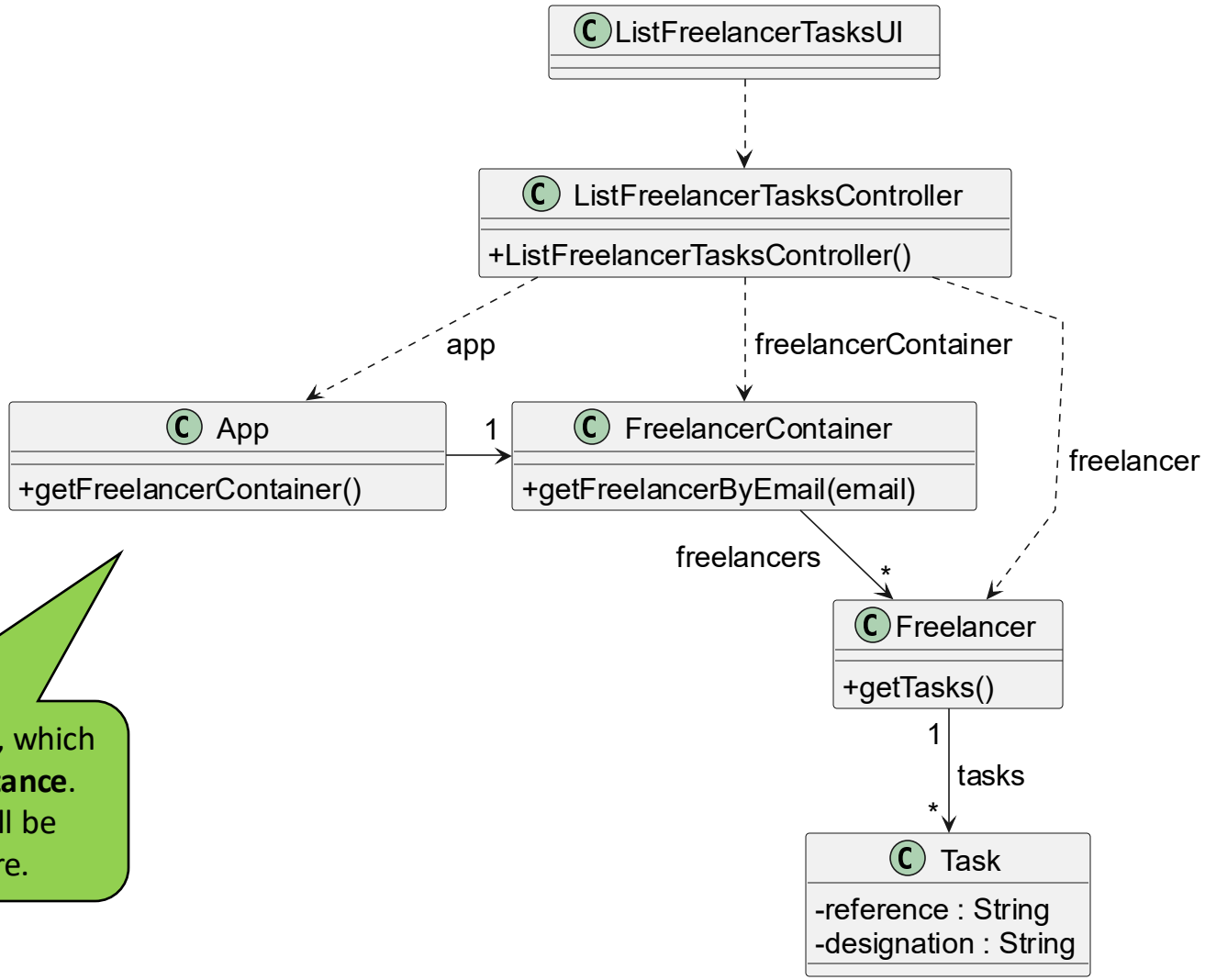
- As a Freelancer, I want to list all my assigned tasks.
  - AC1: The tasks must be sorted alphabetically by their designation.

- Relevant Domain Model excerpt



UC011 – Partial Class Diagram

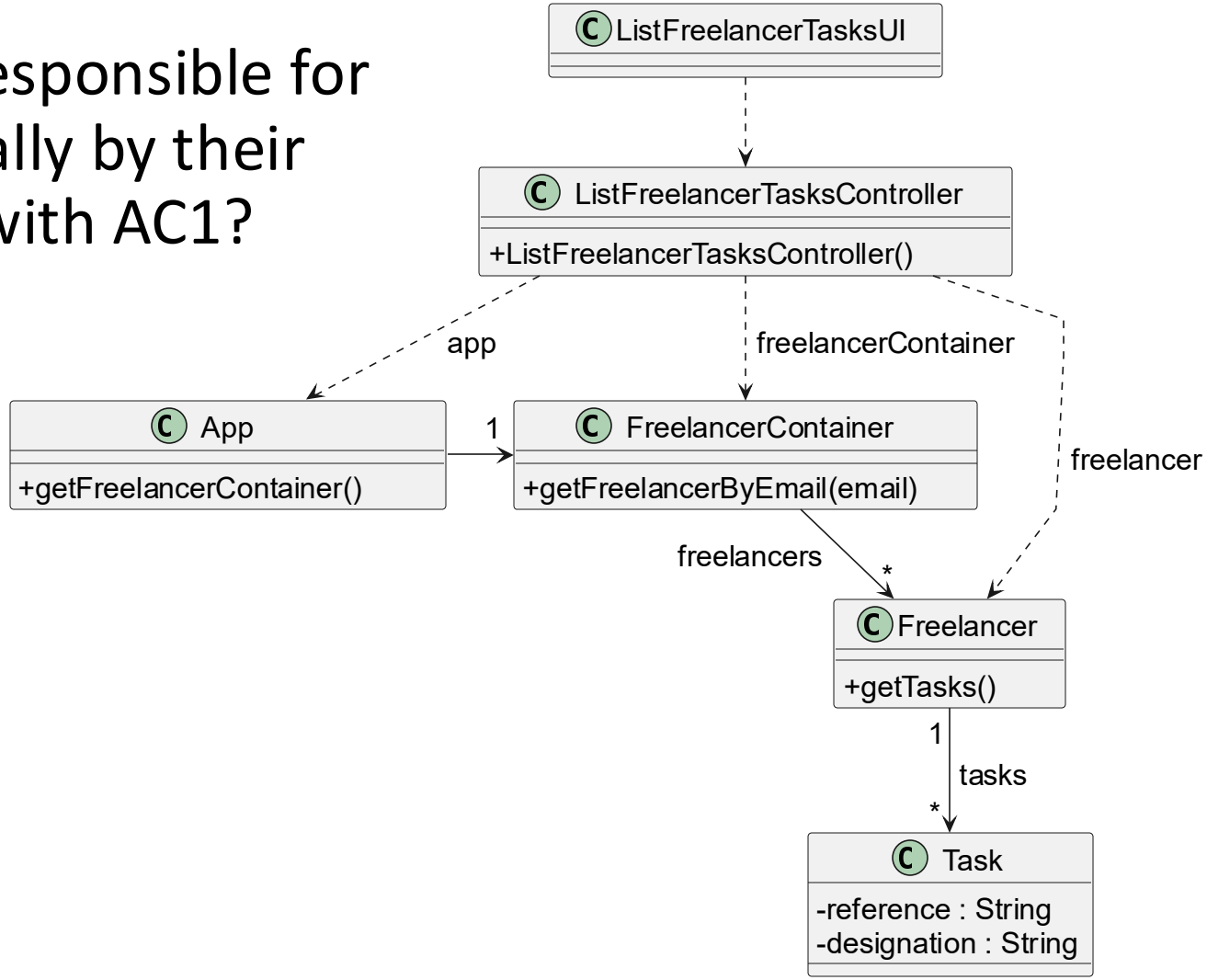
Platform for Outsourcing Tasks



The App class is a **singleton**, which means it has **only one instance**. The Singleton pattern will be address in a next lecture.

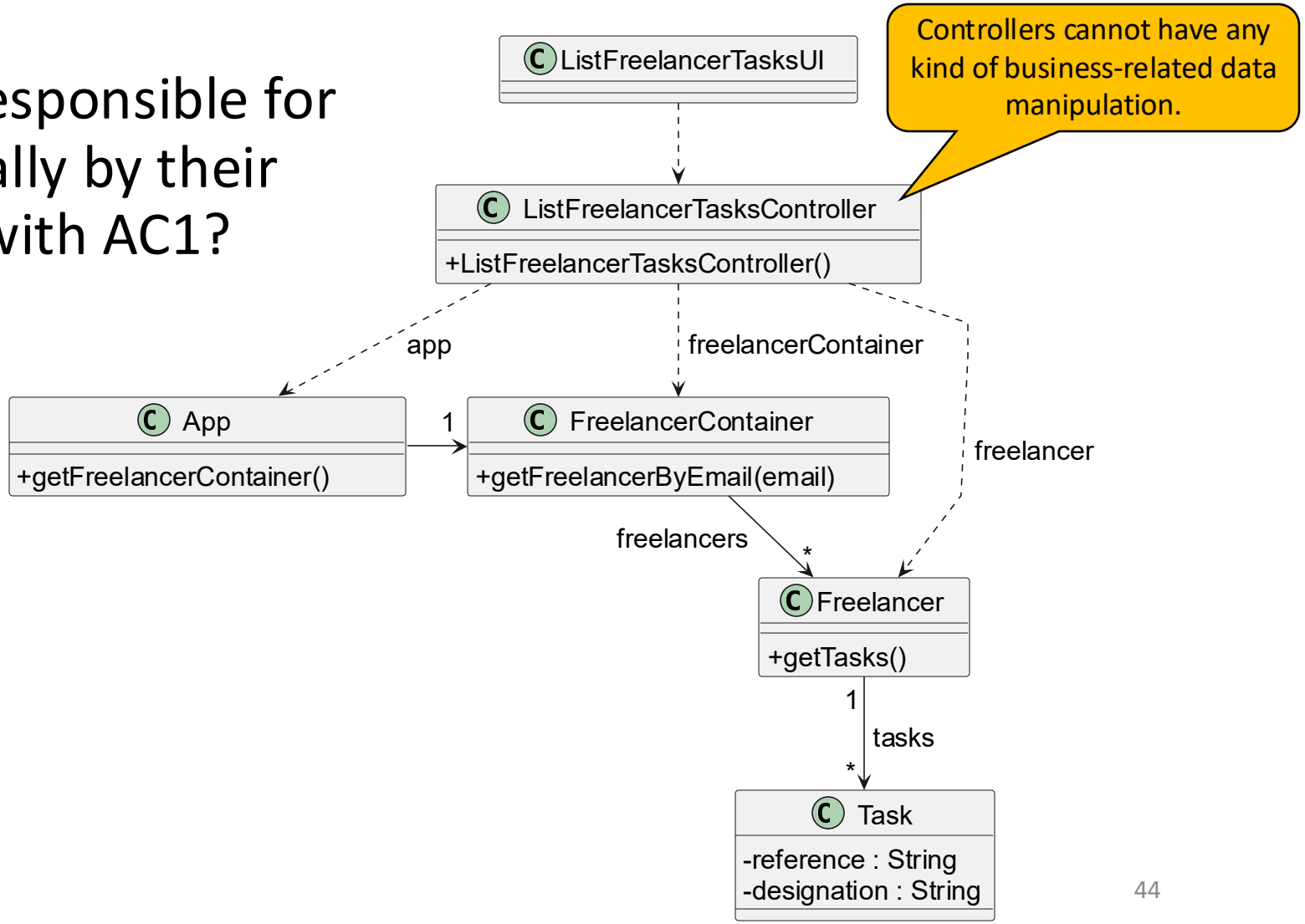
Complying with AC1: Sorting Tasks (1/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - UI?
  - Controller?
  - Organization?
  - Task?



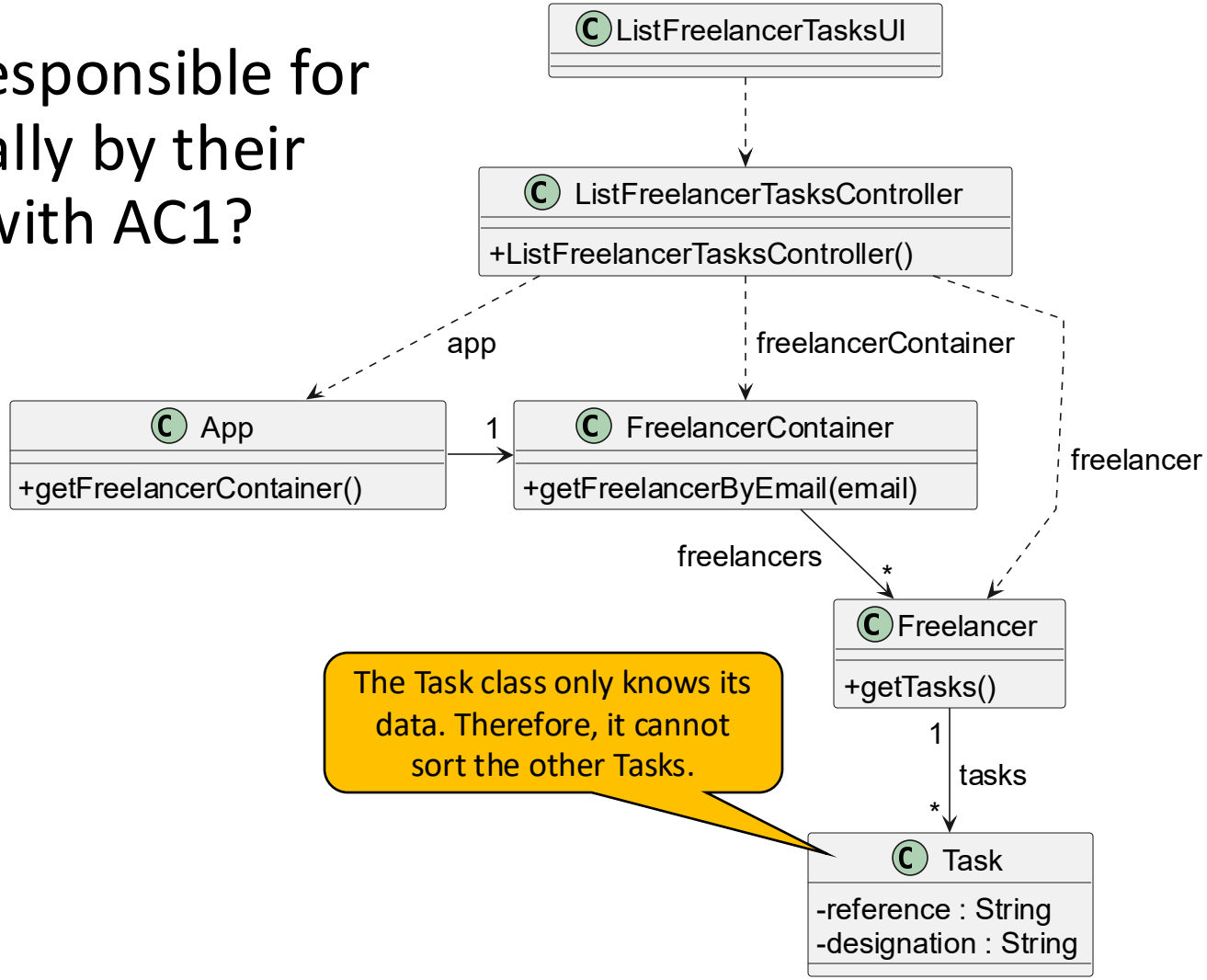
Complying with AC1: Sorting Tasks (2/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - UI?
  - ~~Controller?~~
  - Organization?
  - Task?



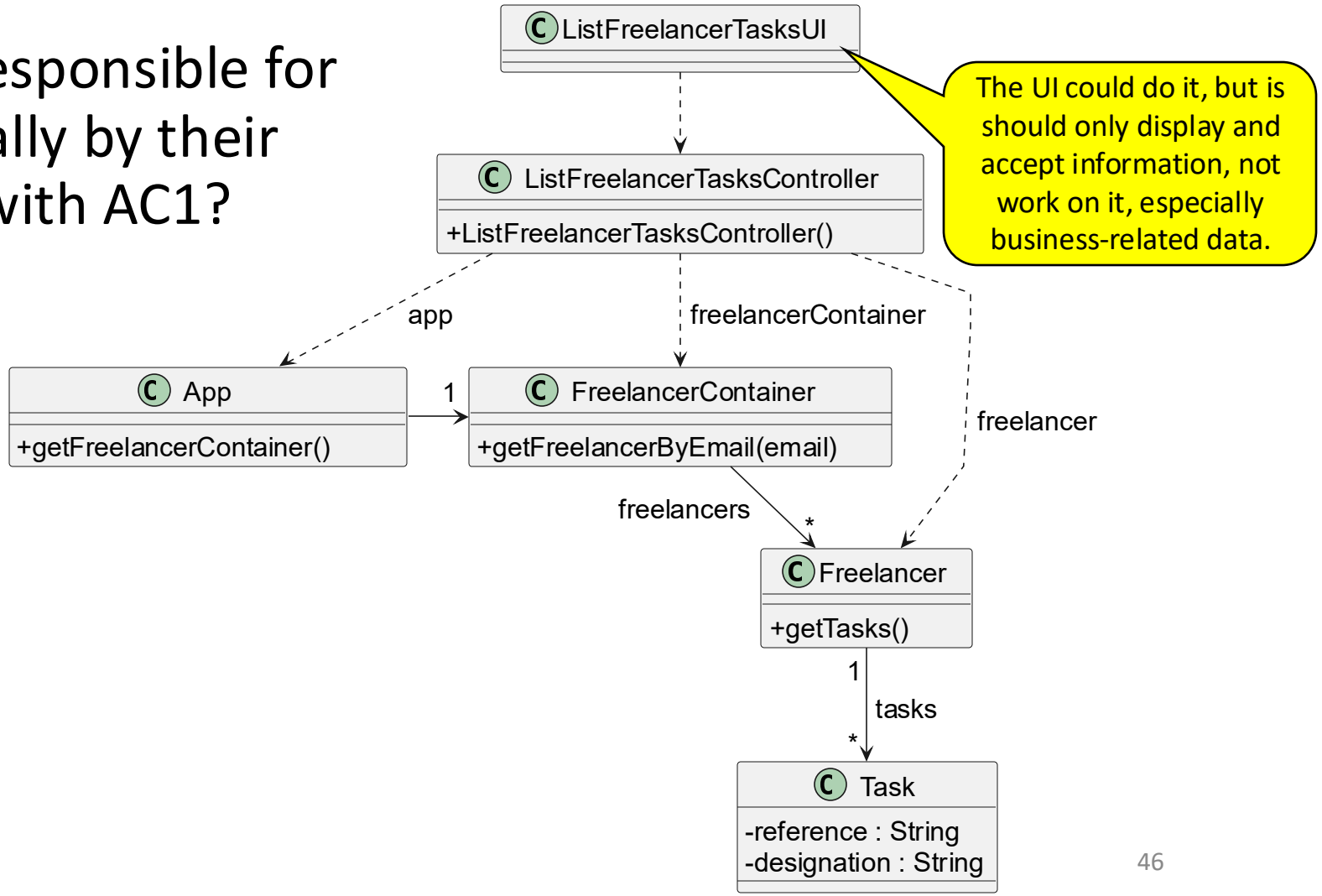
Complying with AC1: Sorting Tasks (3/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - UI?
  - ~~Controller?~~
  - Organization?
  - ~~Task?~~



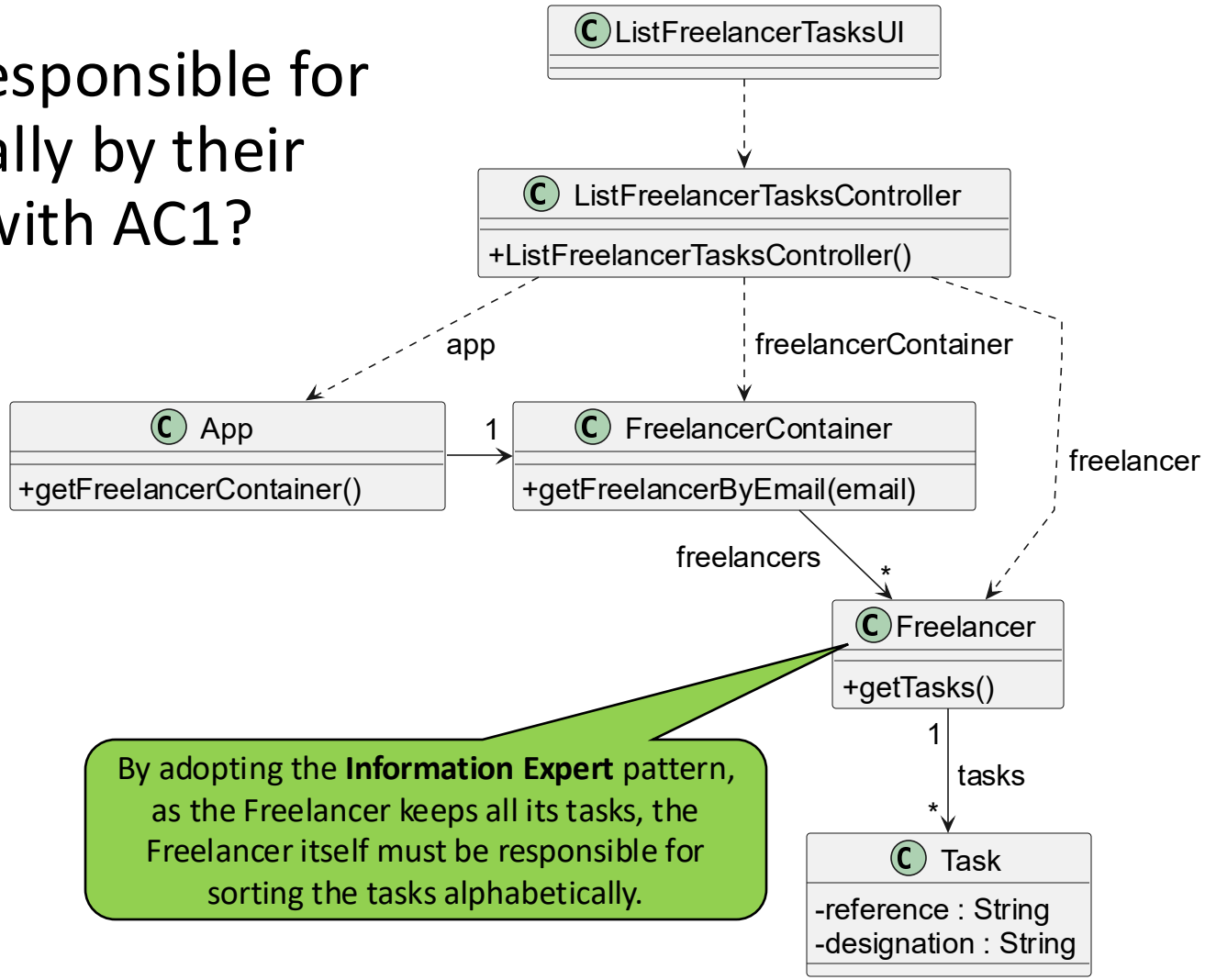
Complying with AC1: Sorting Tasks (4/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - ~~UI?~~
  - ~~Controller?~~
  - Organization?
  - ~~Task?~~



Complying with AC1: Sorting Tasks (5/5)

- Which class should be responsible for sorting tasks alphabetically by their designation, to comply with AC1?
  - ~~UI?~~
  - ~~Controller?~~
  - Organization? ✔
  - ~~Task?~~



UC011 – AC1 Rationale (1/5)

- To comply with AC1 and according to Information Expert (IE), **the Freelancer class should be responsible for sorting the Tasks** because it holds the Tasks assigned to the Freelancer
- The Freelancer class should implement the **getSortedTasks()** function
- While this makes sense, as a side effect, this approach:
  - **Increases Coupling** between Freelancer and Task
    - Remember the **"Tell, Don't Ask"** principle – how can the Freelancer have access to the **designation** attribute for comparison without a **getDesignation()** function in the Task class?
  - **Reduces Cohesion** of the Freelancer class
    - The Freelancer must not only hold the created Tasks, but also have the responsibility for sorting those Tasks

UC011 – AC1 Rationale (2/5)

Doesn't this smell fishy and look like a repetition of what happened before?

- To comply with AC1 and according to Information Expert (IE), **the Freelancer class should be responsible for sorting the Tasks** because it holds the Tasks assigned to the Freelancer
- The Freelancer class should implement the **getSortedTasks()** function
- While this makes sense, as a side effect, this approach:
  - **Increases Coupling** between Freelancer and Task
    - Remember the **"Tell, Don't Ask"** principle – how can the Freelancer have access to the **designation** attribute for comparison without a **getDesignation()** function in the Task class?
  - **Reduces Cohesion** of the Freelancer class
    - The Freelancer must not only hold the created Tasks, but also have the responsibility for sorting those Tasks

UC011 – AC1 Rationale (3/5)

Doesn't this smell fishy and look like a repetition of what happened before?

Haven't we increased, once again, the coupling of the Task class?

- To comply with AC1 and according to Information Expe **the Freelancer class should be responsible for sorti** holds the Tasks assigned to the Freelancer
- The Freelancer class should implement the **getSortedTasks()** function
- While this makes sense, as a side effect, this approach:
  - **Increases Coupling** between Freelancer and Task
    - Remember the **"Tell, Don't Ask"** principle – how can the Freelancer have access to the **designation** attribute for comparison without a **getDesignation()** function in the Task class?
  - **Reduces Cohesion** of the Freelancer class
    - The Freelancer must not only hold the created Tasks, but also have the responsibility for sorting those Tasks

UC011 – AC1 Rationale (4/5)

Doesn't this smell fishy and look like a repetition of what happened before?

- To comply with AC1 and according to Information Expe
  the Freelancer class should be responsible for sorti
  holds the Tasks assigned to the Freelancer

Haven't we increased, once again, the coupling of the Task class?

- The Freelancer class should implement the **getSortedTasks()** function

How to avoid code repetition on different classes?

- While this makes sense, as a side effect, this approach:
  - **Increases Coupling** between Freelancer and Task
    - Remember the **"Tell, Don't Ask"** principle – how can the Freelancer have access to the **designation** attribute for comparison without a **getDesignation()** function in the Task class?
  - **Reduces Cohesion** of the Freelancer class
    - The Freelancer must not only hold the created Tasks, but also have the responsibility for sorting those Tasks

UC011 – AC1 Rationale (5/5)

Doesn't this smell fishy and look like a repetition of what happened before?

Haven't we increased, once again, the coupling of the Task class?

How to avoid code repetition on different classes?

- To comply with AC1 and according to Information Expe
  the Freelancer class should be responsible for sorti
  holds the Tasks assigned to the Freelance
- The Freelancer class sh imp en **rtedTasks()** function
- While this makes sens h:
  - **Increases Coupling** bet
    - Remember the **"Tell, D** ncer have access to the **designation** attrib **gnation()** function in the Task class?
  - **Reduces Cohesion** of the Fr lanc lass
    - The Freelancer must not only hold create Tasks, but also have the responsibility for sorting those Tasks

HOW TO SOLVE THIS PROBLEM?

How to solve this problem? (1/2)

- Why not have both Organization and Freelancer classes **delegate the sorting responsibility to another class**?

- What responsibilities should that class have?
  - That class should be responsible for handling operations over a particular container object
  - In this case, the class would be handling responsibilities related to the tasks container

How to solve this problem? (2/2)

- **Keep using a Container object** (e.g. list) **if the association requires typical container functions only** (e.g. add, get, delete, iterator)

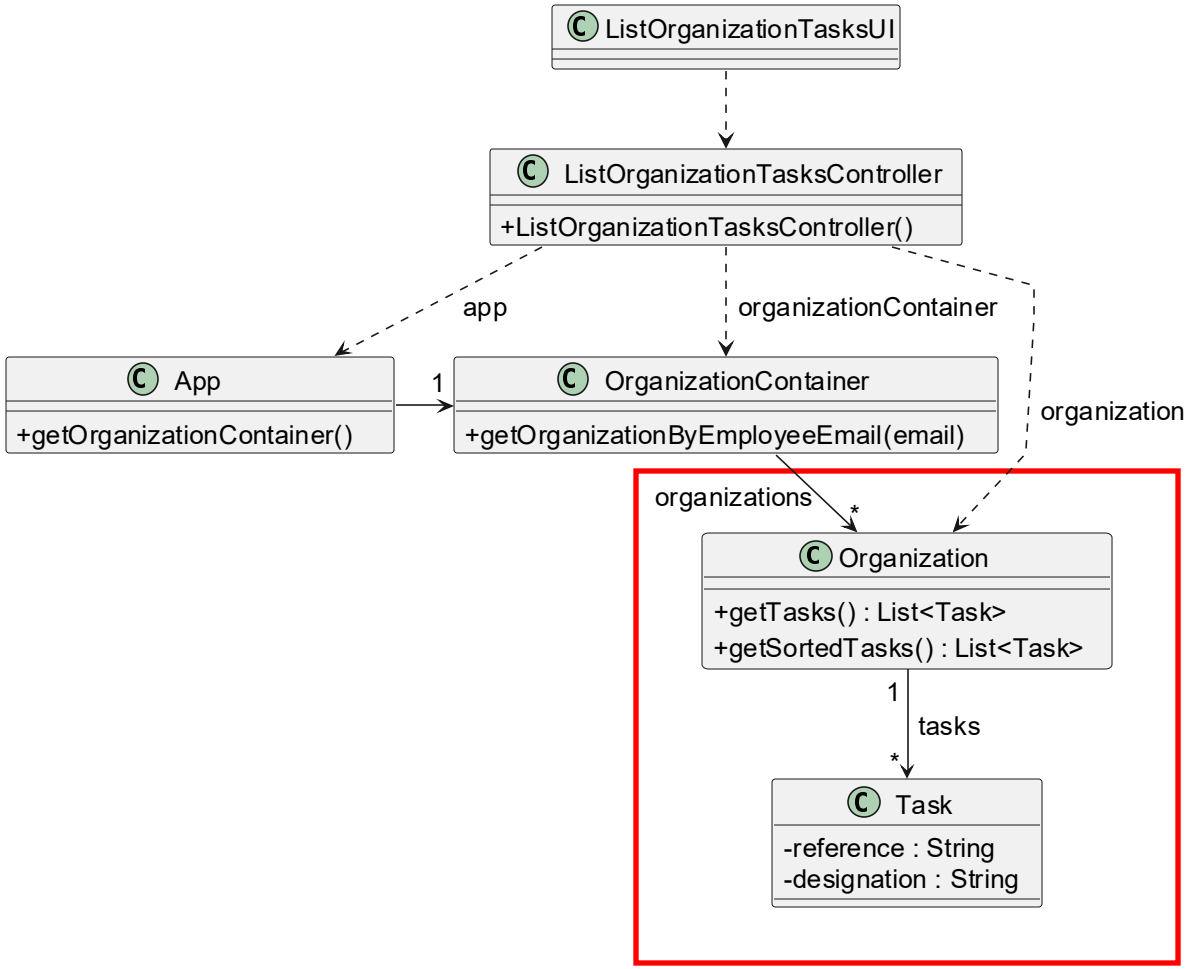
- **Promote the Container object to a New Class if**, in addition to the container functions, **specific functions are required**
  - E.g.: a sorting function like getSortedTasks()

How to name the new software class?

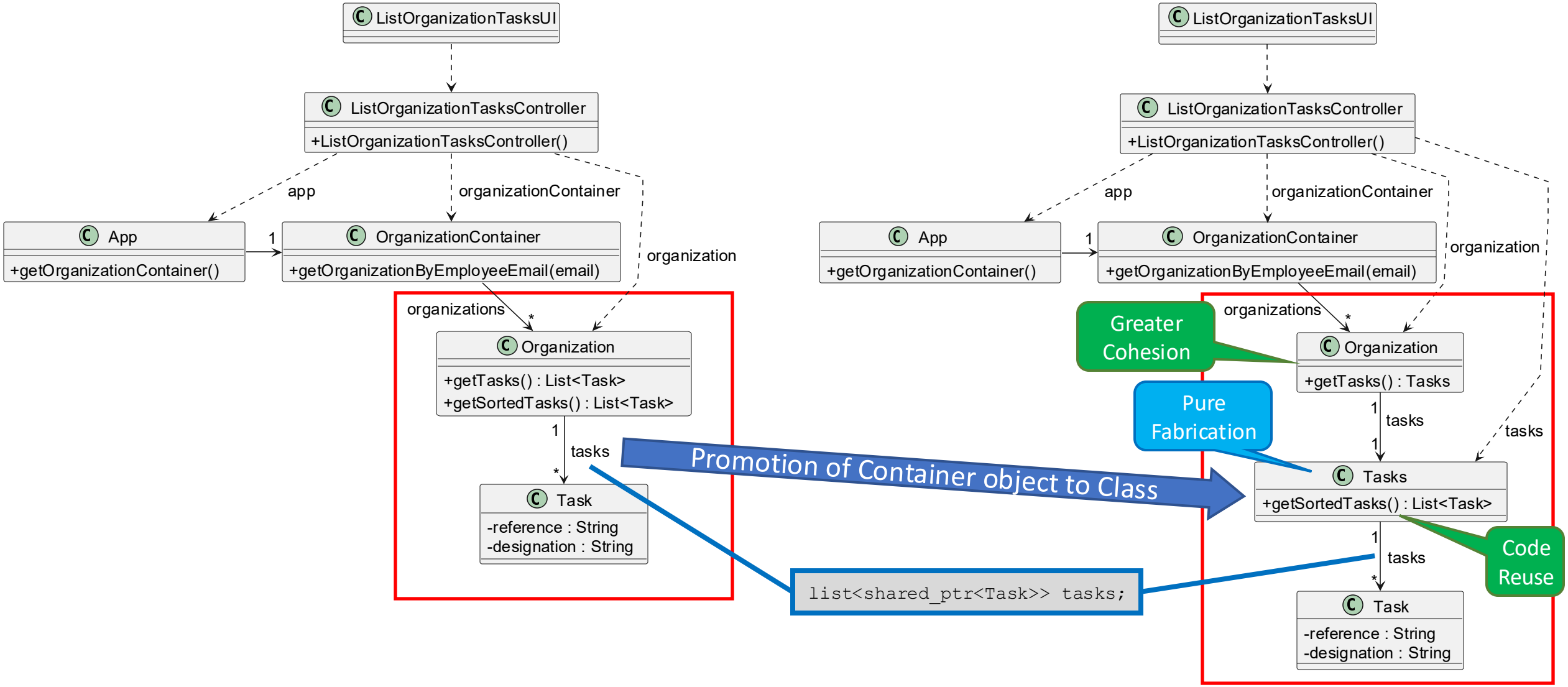
- When the New Class has a more **“global” scope** in the system, it might be named using as a suffix like, for example: Store, **Container** or Repository
  - E.g.: OrganizationStore, **OrganizationContainer**, OrganizationRepository

- When the New Class has a more **“local” scope** (i.e. restricted to a given instance), it might be named using a suffix like, for example: List (when of type List) or simply the name of the class **in plural**
  - E.g.: TaskList, **Tasks**

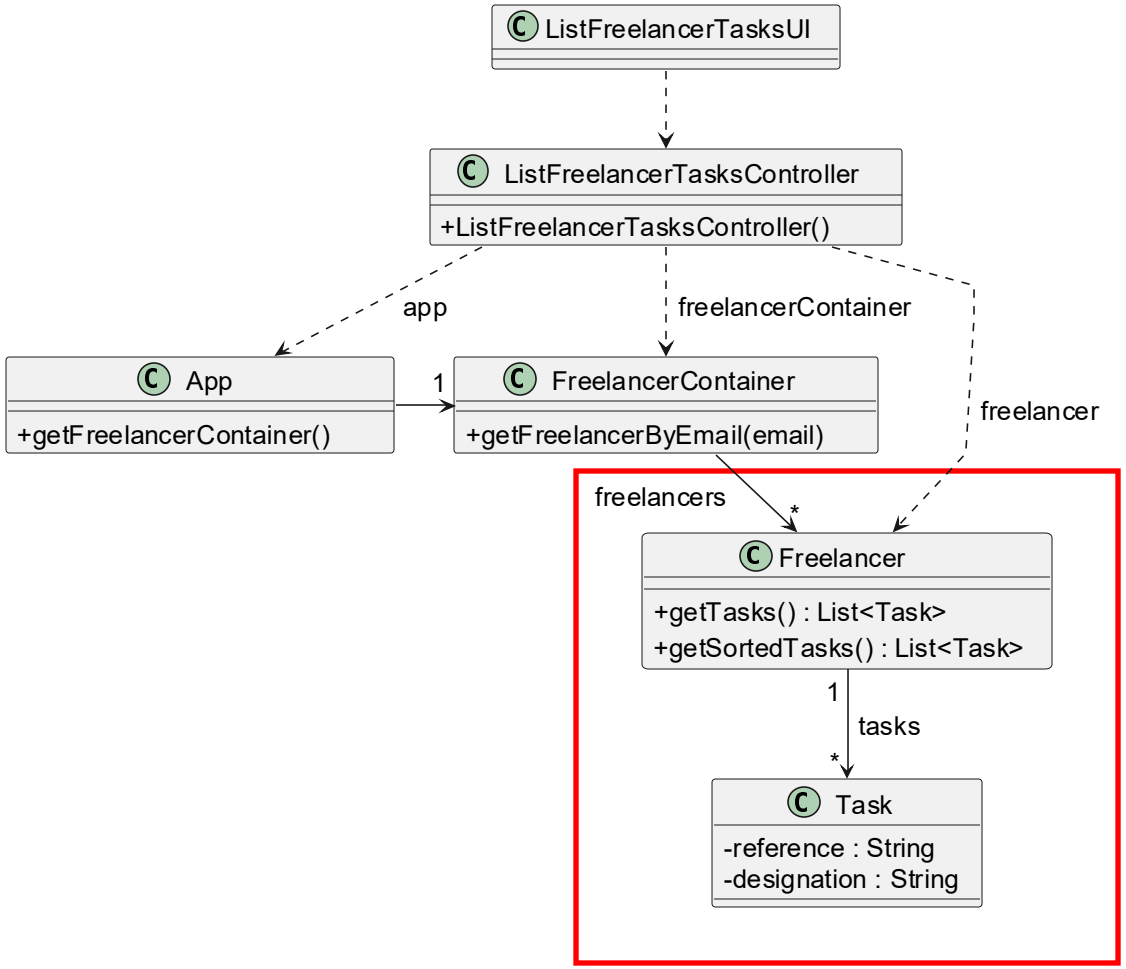
Changes to the UC010 design (1/2)



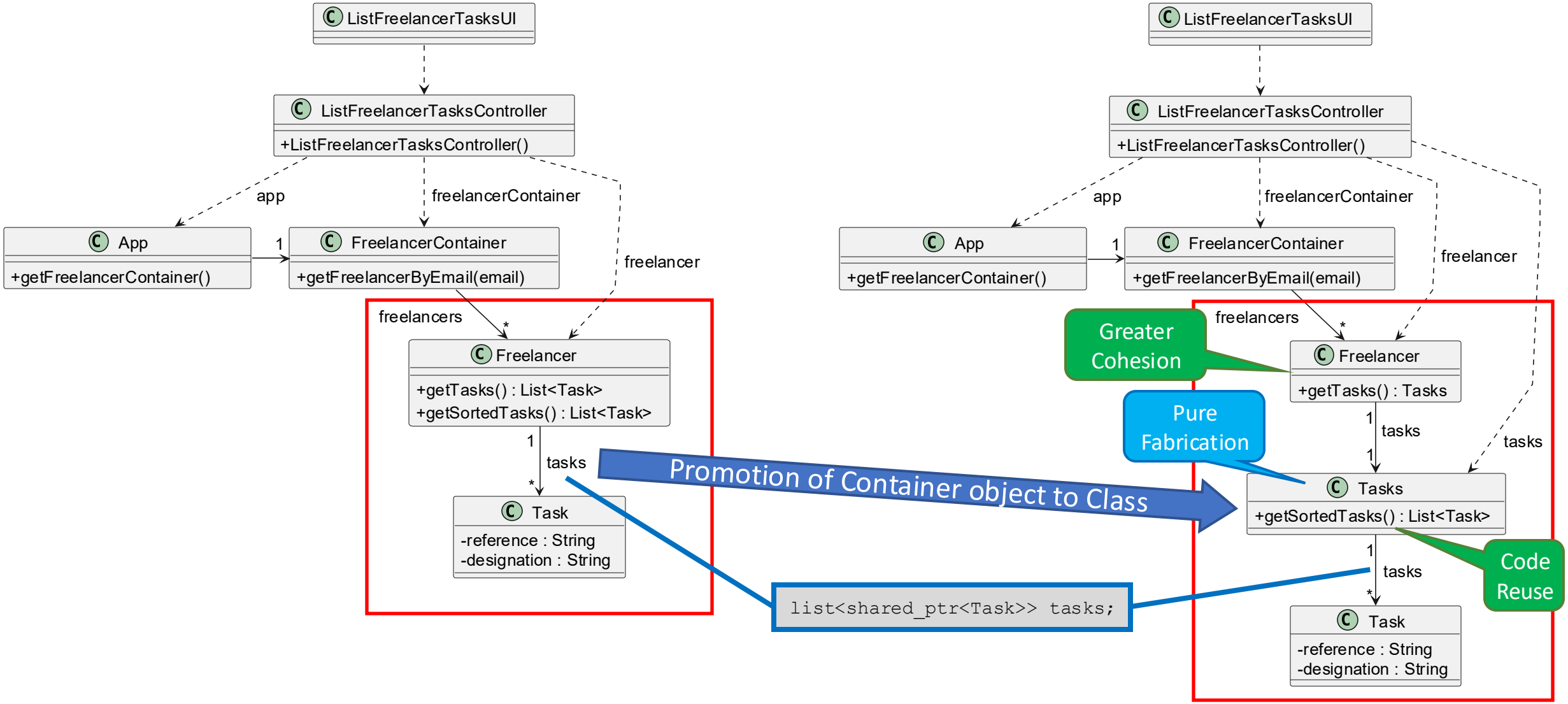
Changes to the UC010 design (2/2)



Changes to the UC011 design (1/2)



Changes to the UC011 design (2/2)



Summary (1/2)

- 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

Summary (2/2)

- High Cohesion and Low Coupling must be considered while designing
  - Not only to promote container objects to software classes
    - Pure Fabrication
  - But also, on other scenarios (e.g.: filter/sort a container object by some criteria) to evaluate plausible alternatives
    - Tell, Don’t Ask
    - Information Expert

- The Domain Model is used to inspire the creation of software classes (the Design Model), but the opposite is not true

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