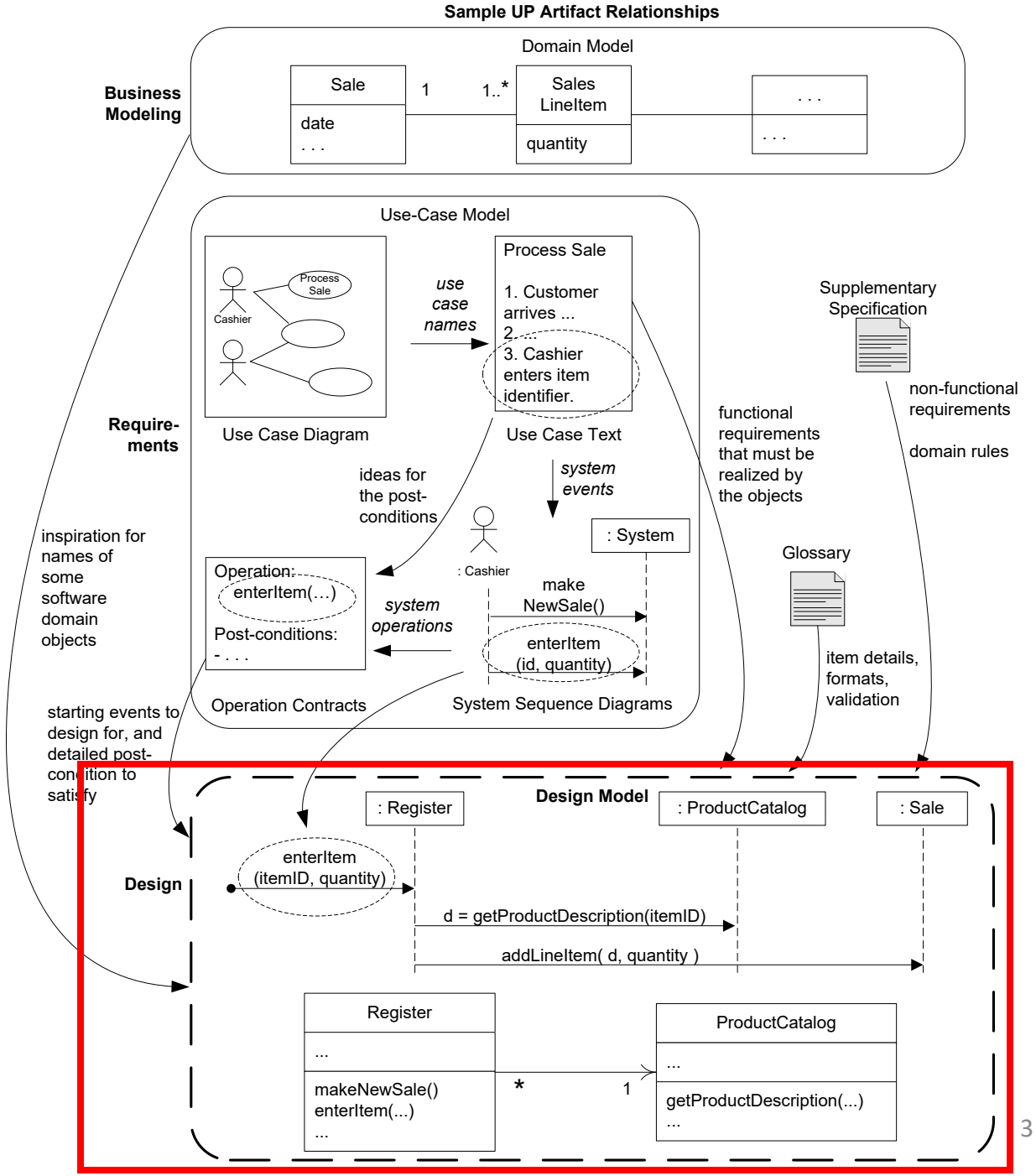
GoF Design Patterns

ISEP / LETI / ESOFT

Topics

- Gang of Four (GoF) Design Patterns
- Singleton Pattern
- Adapter Patter
- Repository Pattern
- Factory Method Pattern
- Abstract Factory Pattern
- Practical Open Issues in the DemoTasks Project
  - Using a Configuration File to set up the Repository Factory
  - Introducing the concept of Service to fully decoupling Business Logic from Persistence Logic

Artifacts Overview



Gang of Four (GoF) Design Patterns

- Each pattern describes a problem which occurs repeatedly in OOP SW development and describes a core solution to that problem
  - GRASP and SOLID principles are extended/applied/combined/detailed
  - Recipes for the common OO problems
- Helps:
  - Finding appropriate objects
  - Determining objects granularity
  - Specifying objects interfaces
  - Specifying object implementations
  - Programming to an interface and not to an implementation

GoF Design Patterns Catalog

- Creational Design Patterns
  - **Abstract Factory** *
  - Builder
  - **Factory Method** *
  - Prototype
  - **Singleton** *
- Structural Design Patterns
  - **Adapter** *
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavior Design Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

* Patterns addressed in this slide deck

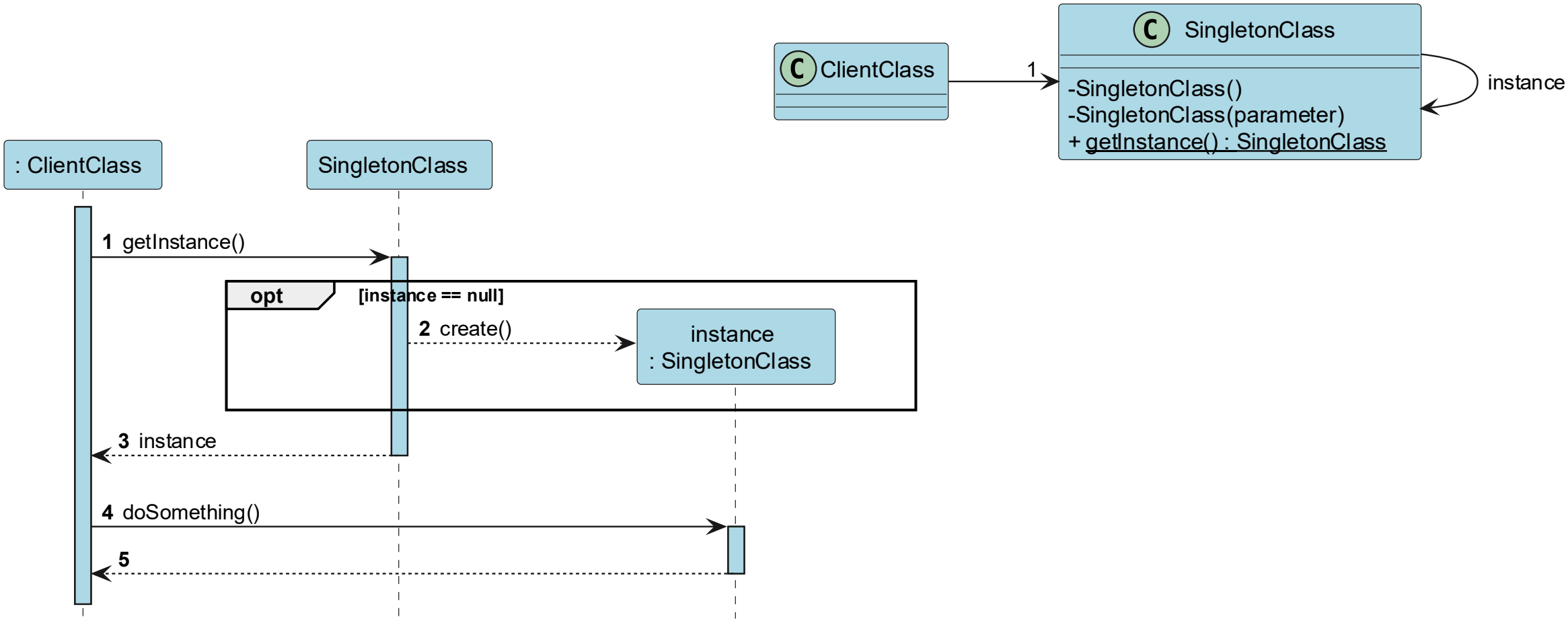
Singleton Pattern

Applied to the DemoTasks Project

Singleton Pattern

- **Problem**
  - How to ensure that there is **only one instance** of a given class; and
  - How to provide a global access point to such instance?
- **Solution**
  - In a very simple way, the *singleton* class needs to:
    - Have its **constructor(s) private** (or protected), to prevent other classes from creating instances
    - Have a **private static attribute to itself**
    - Have a **public static method** (usually called ***getInstance***) that is used by other classes to get the unique instance of the class
  - Instance creation
    - ***Lazy*** *initialization*: restricts the creation of the class instance until it is requested for the first time
    - ***Eager*** *initialization*: the class instance is created during the start-up time

Singleton Pattern – *Lazy initialization*



Singleton Pattern – Overview

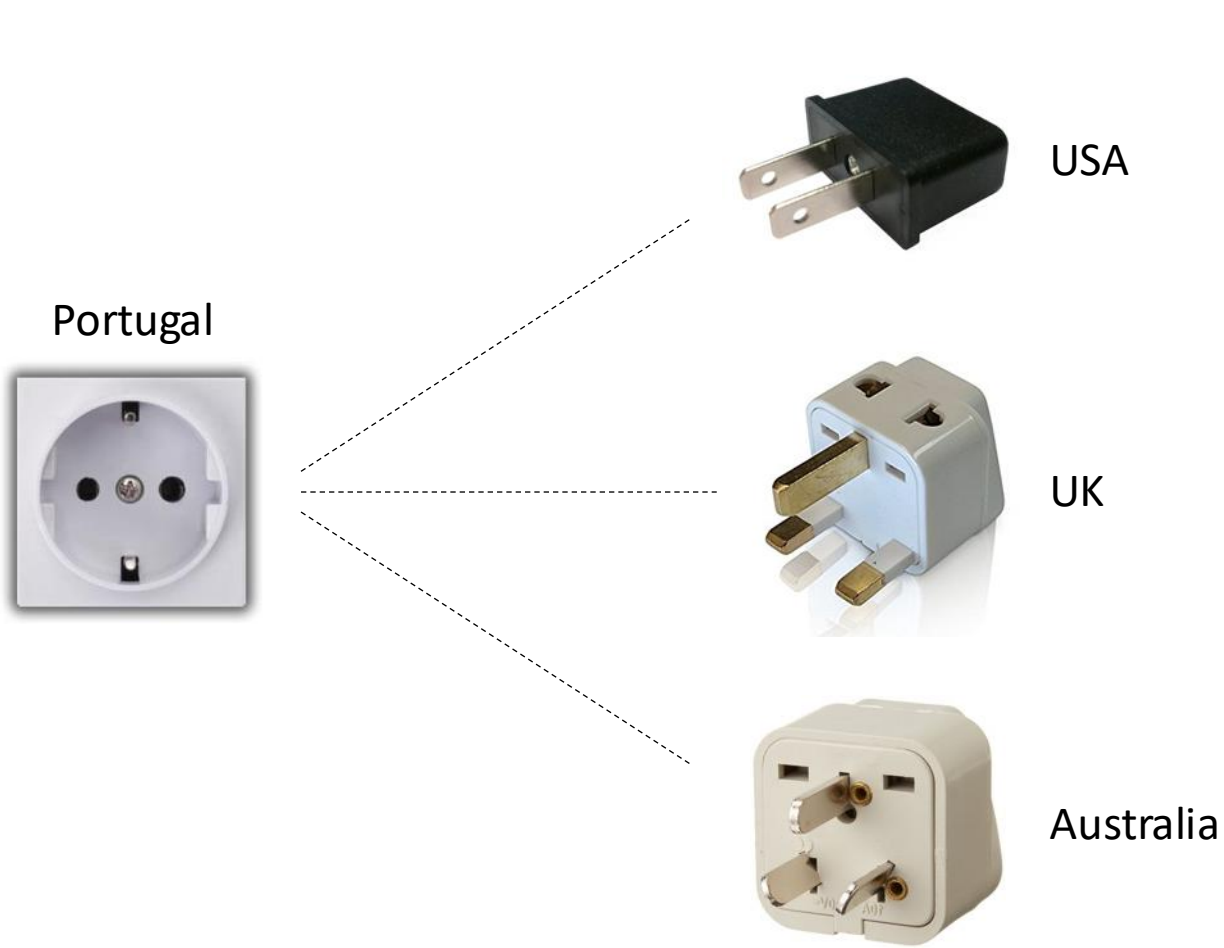
- The Singleton pattern is a well-known software design pattern
- **Singleton syndrome:** it is too often seen as the most appropriate pattern for your current use case → But it turns out it is not!
- Sometimes, it is considered an **Anti-Pattern**
  - Its usage should be **minimized**, as it promotes the existence of an application global state and testing activities become harder.
  - Although there are more effective techniques, they are beyond the scope of ESOFT.

App class

- In the DemoTasks Project, is there any singleton class in the domain?
  - **No!** Several persons might exist, and each one has/maintains his/her own set of categories and tasks.
- But is there any singleton class in the project?
  - **Yes!** The **App** class, used (exclusively) by the UI Controllers of the console application.
- What is the underlying rationale of **App** being a singleton?
  - The **App** class is somehow simulating a user authentication and thus, providing Controllers with a global access to the person instance, corresponding to such user
  - Please notice that:
    - **Only the Controller classes are aware of the App class/instance**
    - **The singleton is not being propagated to the domain layer**

Adapter Pattern

Supporting Multiple Electrical Connectors



Universal Adapter (1/2)

Travelers use an adapter, i.e. a device that allows using plugs of the 3 formats in European sockets.

Portugal

USA

UK

Australia

Universal Adapter (2/2)

Travelers use an adapter, i.e. a device that allows using plugs of the 3 formats in European sockets.

Portugal

USA

UK

Australia

It sounds like a good solution. But what if we want to support another format?

Israel

Specialized Adapter

To support a new format, simply develop a new adapter, without affecting the existing ones!

USA

Portugal

UK

Australia

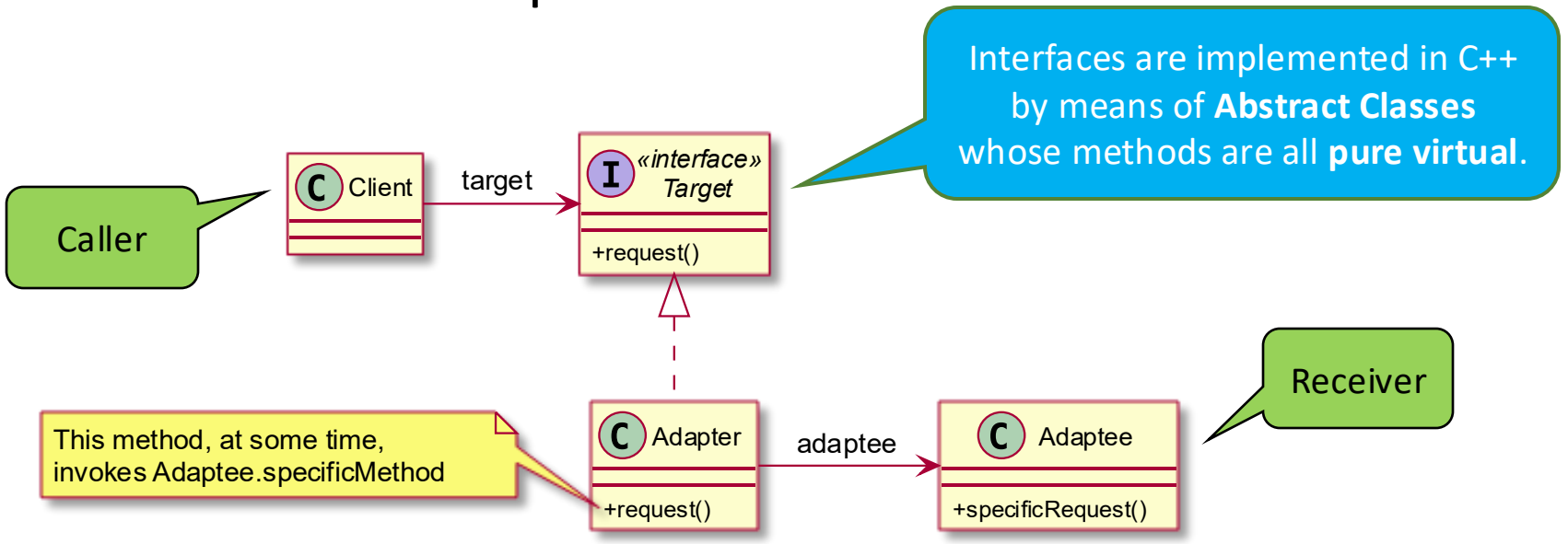
Adapter Pattern (1/2)

- **Problem**
  - How to provide a single stable interface to similar components having different interfaces?
    - How to allow objects with incompatible interfaces to collaborate with each other?
    - How to make two distinct interfaces compatible (each one of a different object)?

- **Solution**
  - Use, for each component, an intermediate **adapter object** to convert calls to the appropriate interface.

Adapter Pattern (2/2)

- The ***Adapter*** converts the interface of an ***Adaptee*** (i.e. the receiver object) to another interface (***Target***) that is expected by some ***Client*** (i.e. the caller object)
- Allows classes with incompatible interfaces to work together
- **An adapter must be implemented for each required adaptation**, allowing the easy addition of new adapted classes

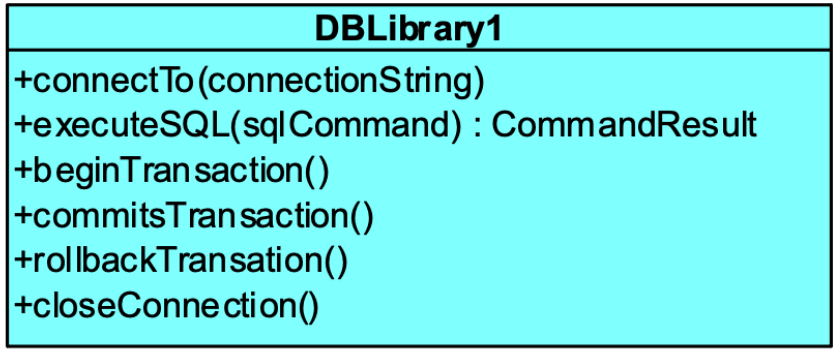


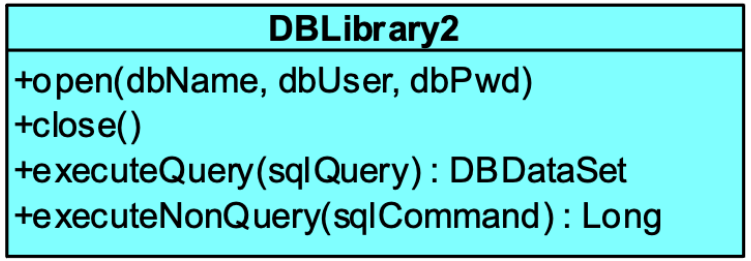
Applying the Adapter Pattern

Data Persistence Example

Data Persistence Example (1/6)

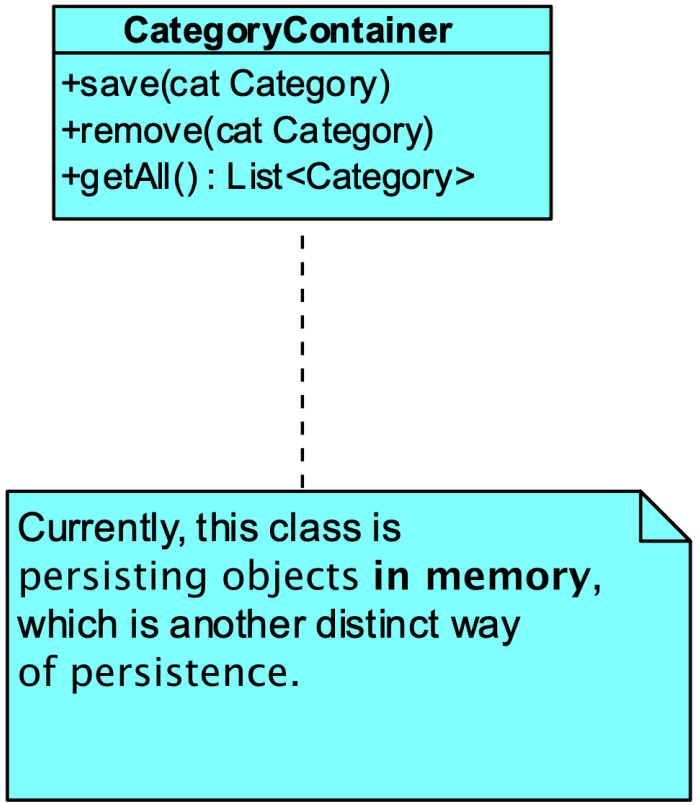
- How to support data persistence across multiple database providers, knowing that it must be used the provided native database library?
  - For DB1 Provider, the DBLibrary1 library must be used
  - For DB2 Provider, the DBLibrary2 library must be used
  - Several other database providers and libraries may be supported



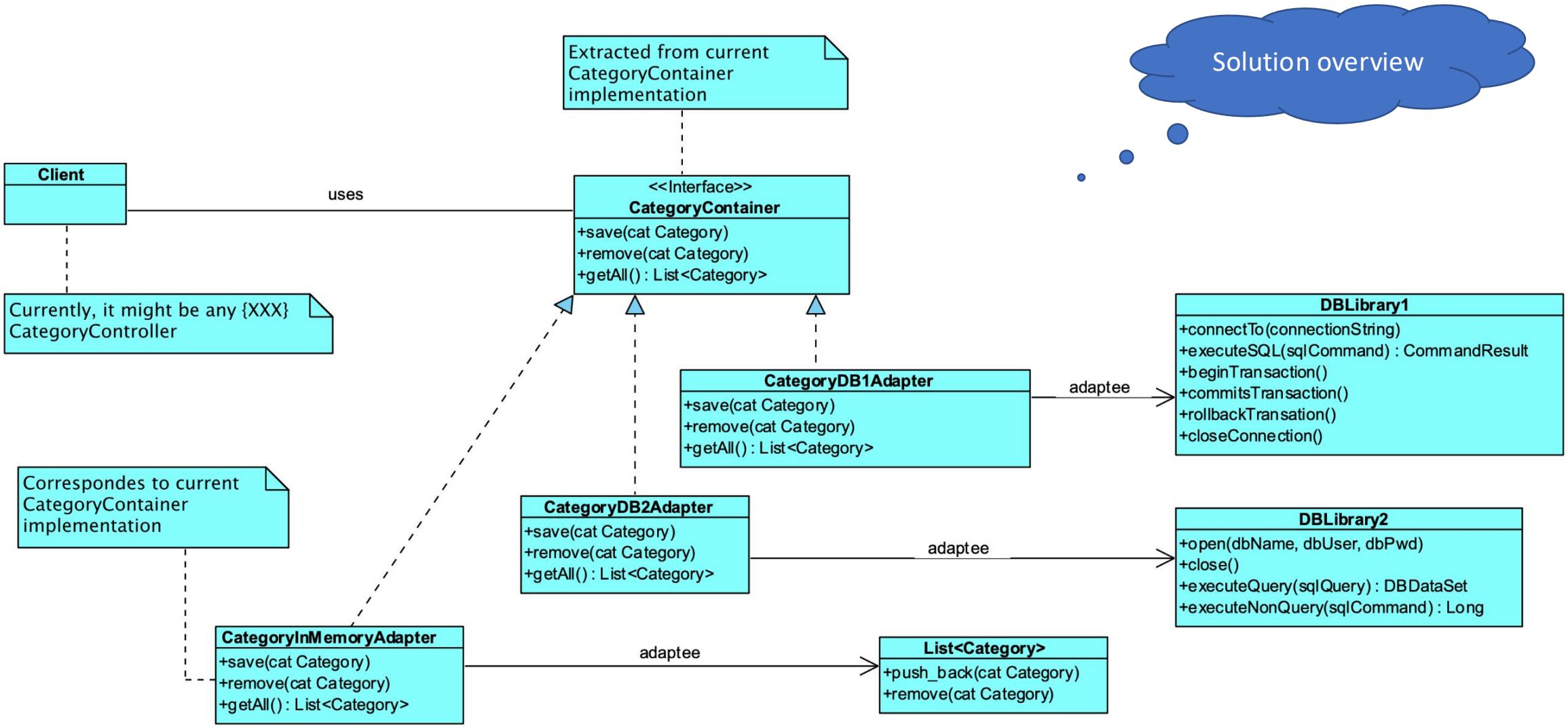


Data Persistence Example (2/6)

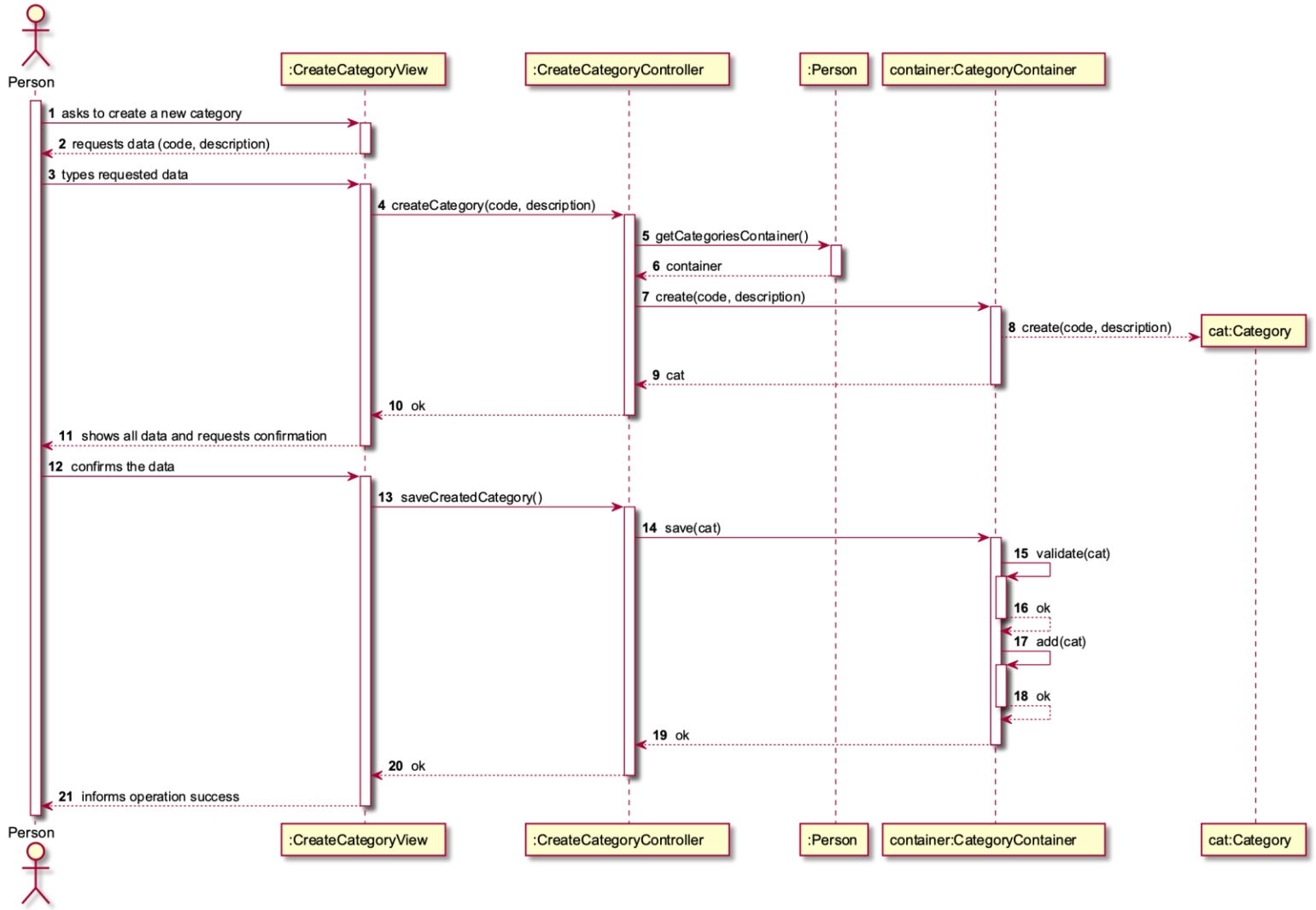
- How are we currently saving category objects in the DemoTasks project?
- By using the method:
  - `save(Category cat)`
    … to add and/or update a category
  - `remove(Category cat)`
    … to delete an existing category
  - `getAll():list<Category>`
    … to obtain all the existing categories



Data Persistence Example (3/6)



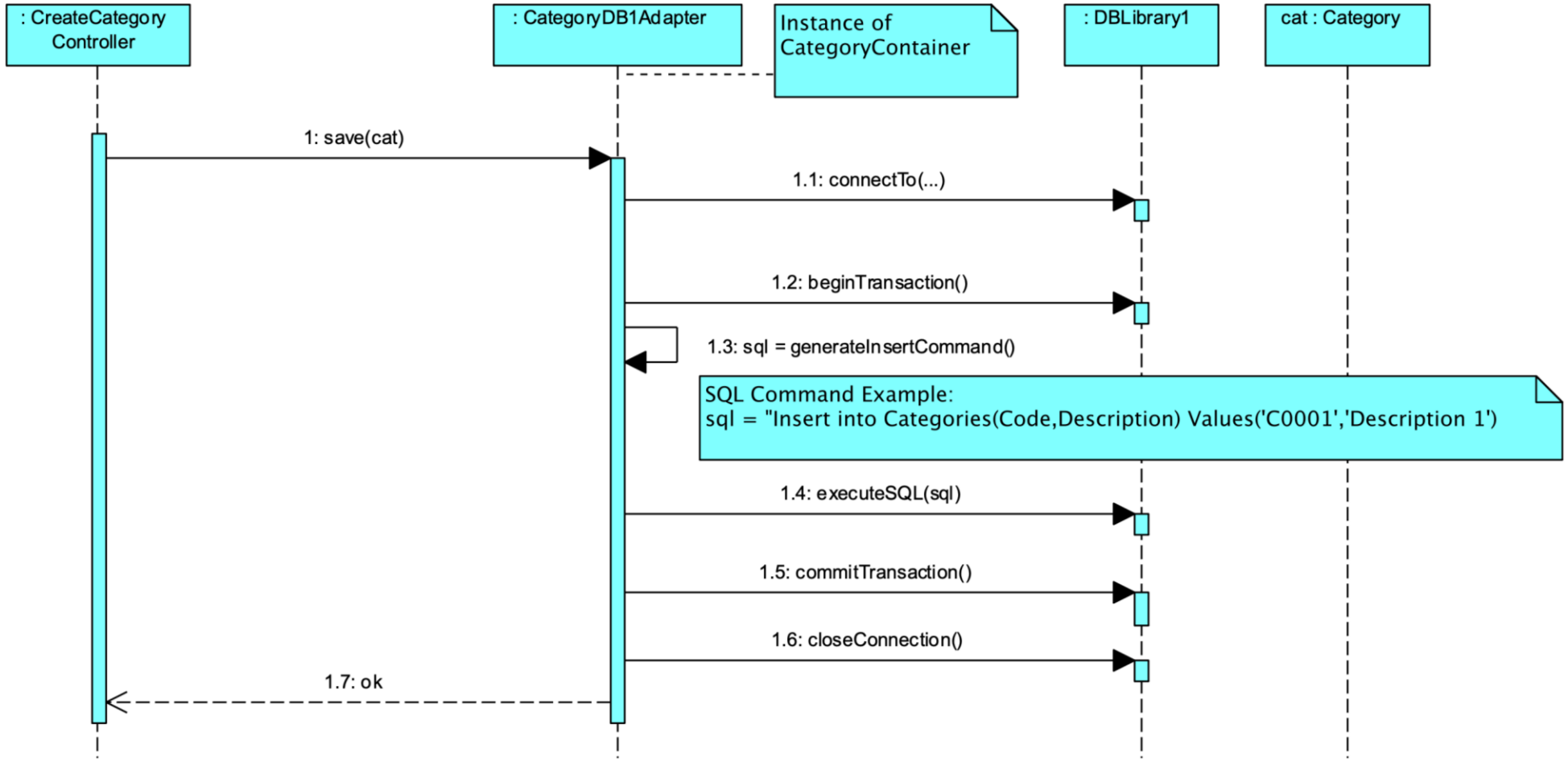
Data Persistence Example (4/6)



- What would change here?
  - **Nothing!** The controller would still receive from Person, an instance of CategoryContainer, and would continue making the same calls
- The controller does not need to know which adapter object it is interacting with
- The Person object is responsible for providing the intended adapter object

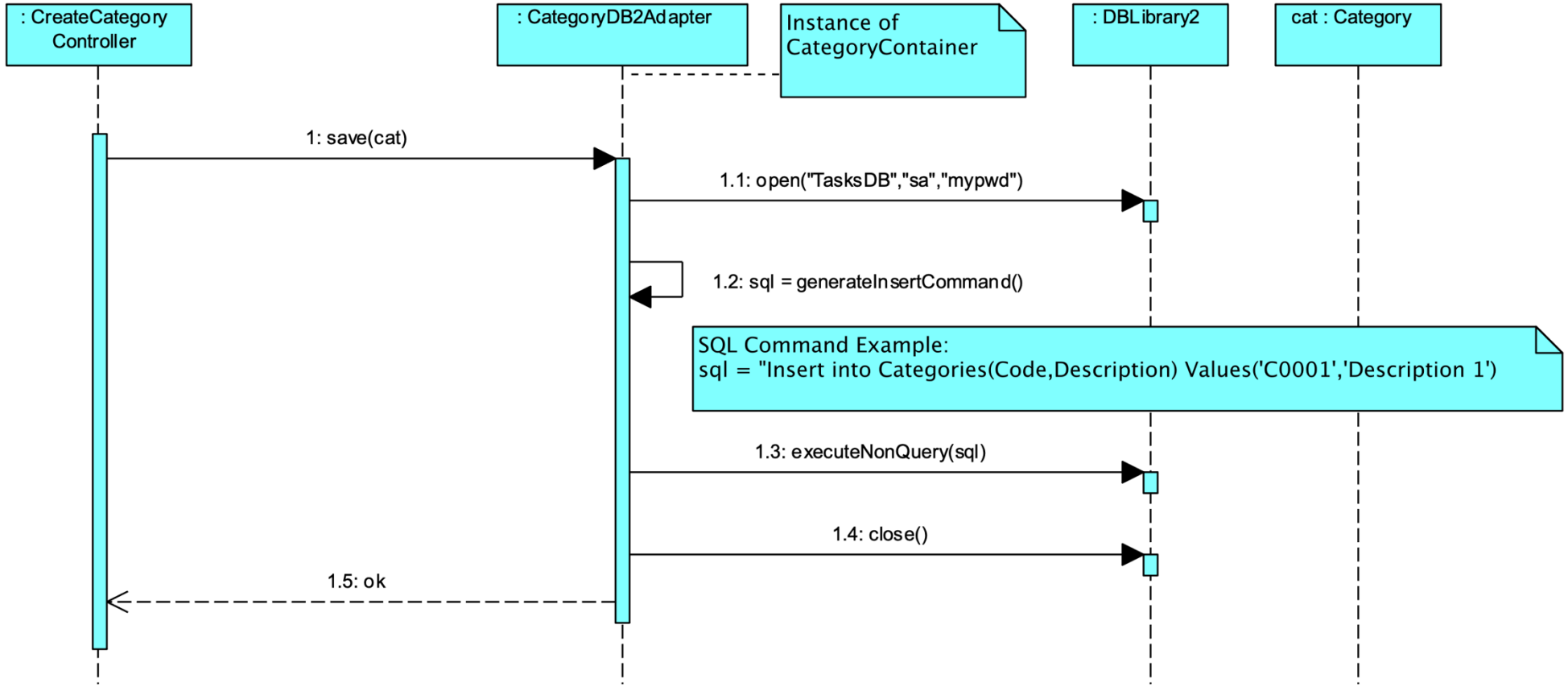
Data Persistence Example (5/6)

Detail of what would happen when using the **CategoryDB1Adapter**



Data Persistence Example (6/6)

Detail of what would happen when using the **CategoryDB2Adapter**



Adapter Pattern – Advantages

- Open/Closed Principle: new adapters can be easily created without having to change the client
- Single Responsibility Principle: separates business logic from details regarding specific interfaces and/or data conversions
- Promotes High Cohesion and Low Coupling
- Combines other GRASP principles such as:
  - Polymorphism
  - Protected Variation

When applied to data persistence, as we did in the previous example, the obtained solution might be seen as a **Repository Pattern** solution.

Repository Pattern

(this is not a GoF Pattern)

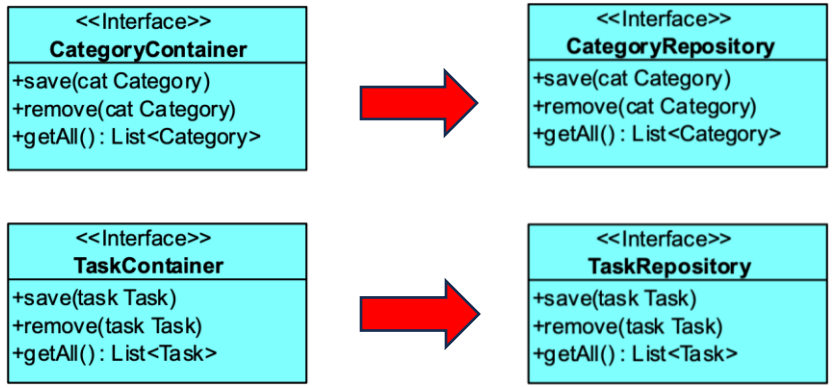
Repository Pattern

- **Problem**
  - How to separate the business logic from the data access logic while easily mapping from business objects to data and vice- versa?
    - How to persist business objects being agnostic of the data access logic?
    - How to recover business objects from persisted data?

- **Solution**
  - Use a repository to **intermediate the communication between the business logic and the data access logic**.
    - The repository should act like an in-memory domain object collection
    - (Business) Client objects use the repository to persist/recover domain objects
    - Each repository implementation encapsulates the mapping logic

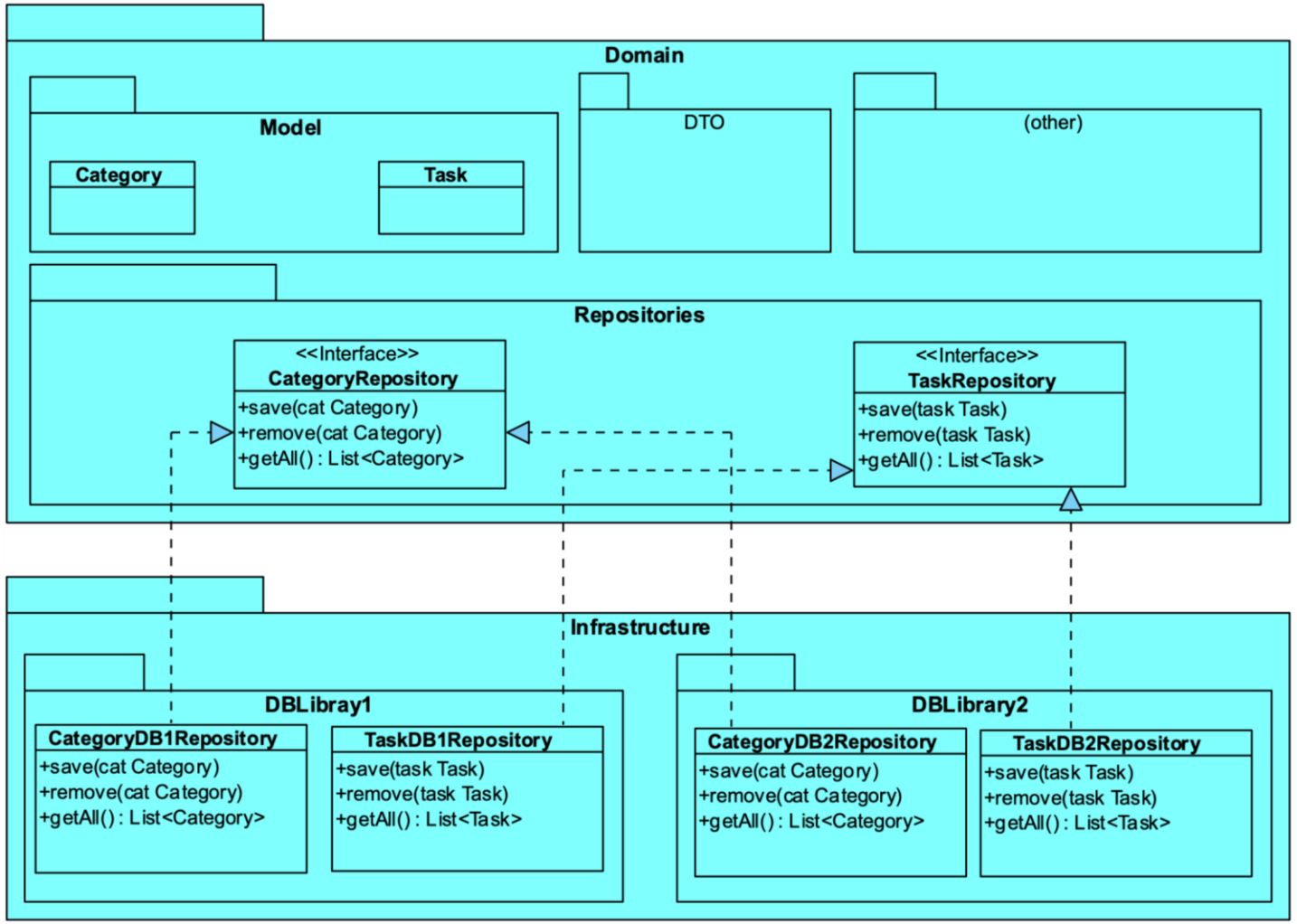
Repository Pattern – Example (1/2)

- In the DemoTasks project, the containers can be thought as repositories
  - CategoryContainer → CategoryRepository
  - TaskContainer → TaskRepository



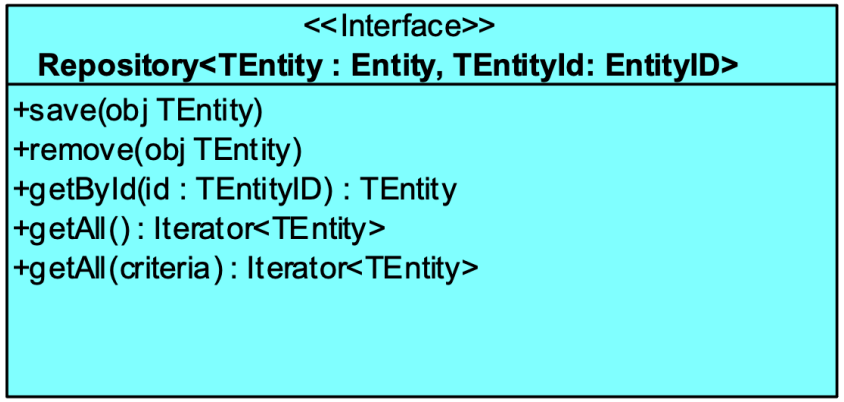
- Each Repository might have several implementations
  - One using an in-memory approach (e.g.: based on the *list* standard class)
  - Other using the DBLibrary1 to persist on a given DB Provider (e.g.: MS SQL Server)
  - Another using the DBLibrary2 to persist on another DB Provider (e.g.: Oracle)

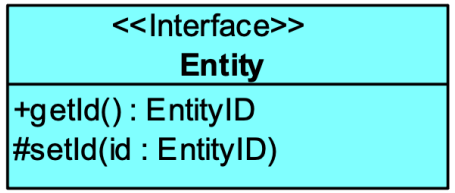
Repository Pattern – Example (2/2)

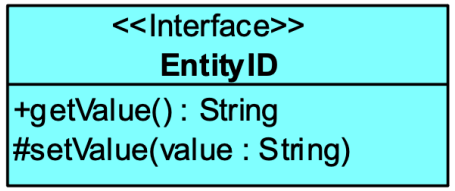


Repository – Common Operations

- Save (create or update) a domain object (aka entity object)
- Remove a domain object
- Get (retrieve) a domain object by its identity (aka primary key)
- Get all domain objects
- Get domain objects that meet some criteria







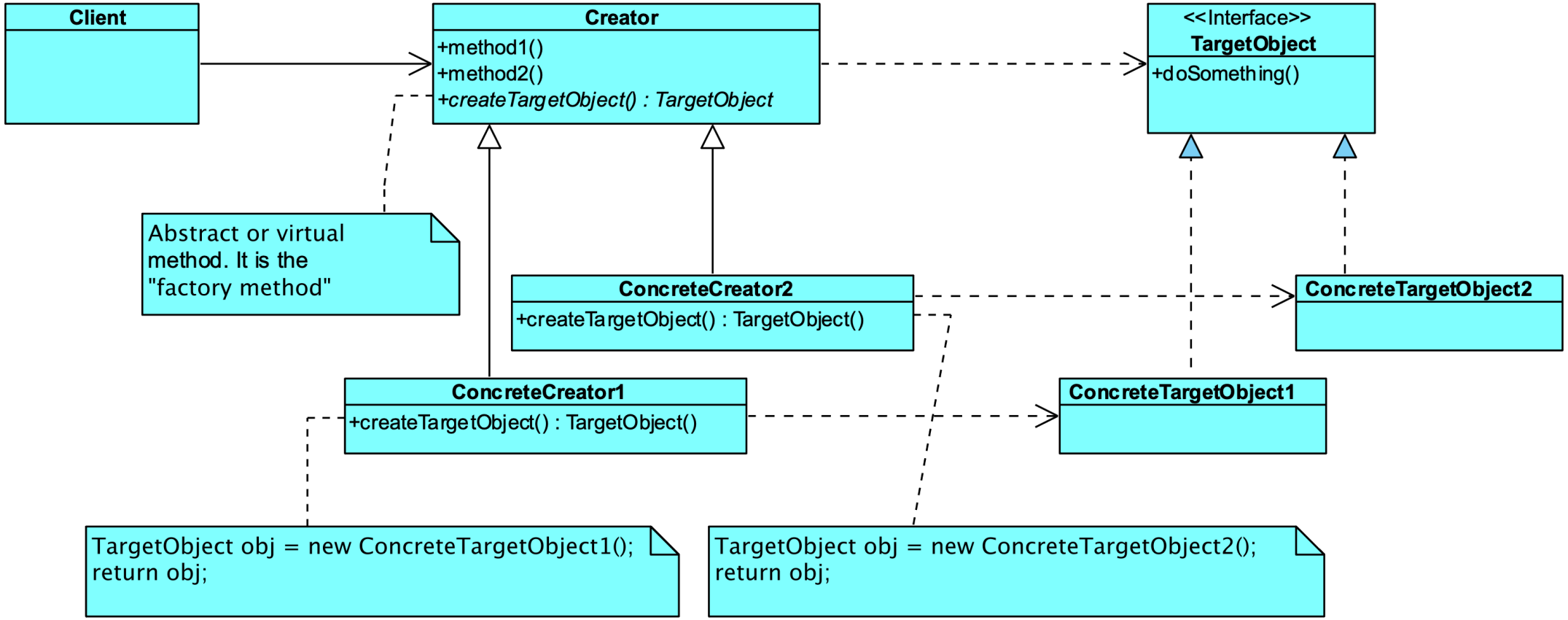
Factory Method Pattern

Factory Method Pattern

- **Problem**
  - How to hide and decouple client classes from the complexity of creating a given concrete object for which multiple subclasses (i.e. implementations) may exist?
    - E.g.: In the DemoTasks Project, how to hide and decouple the Person class from concrete CategoryRepository implementations?

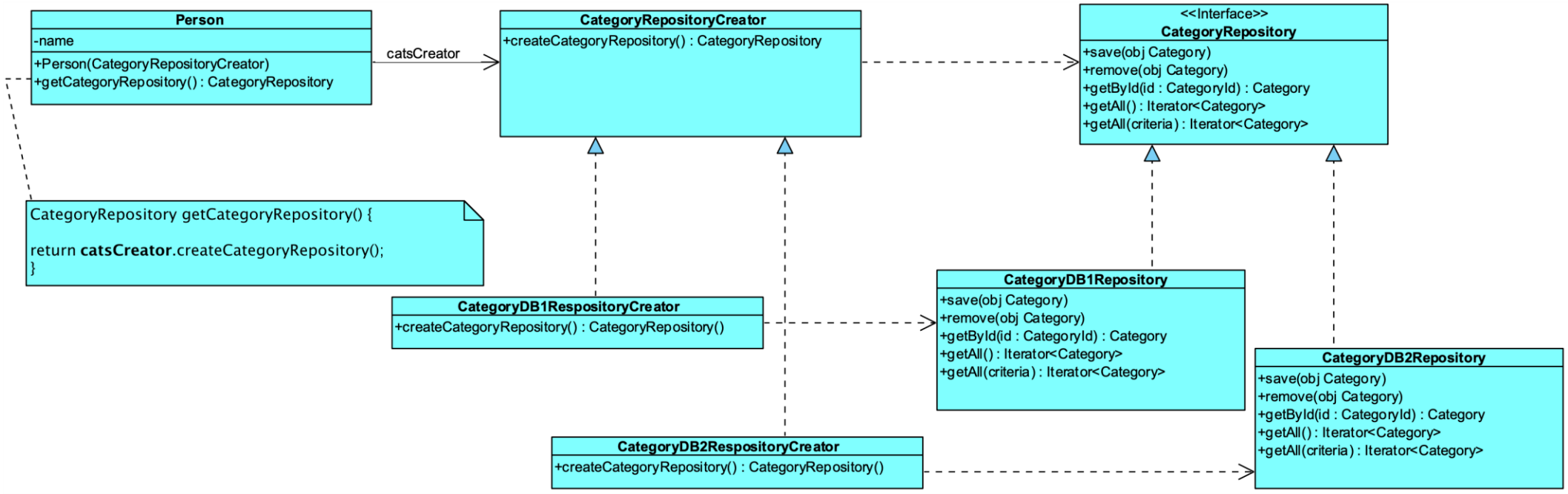
- **Solution**
  - Make sure all concrete objects share a common interface (or abstract class), then specify a creator (factory) method that returns any object as an instance of the common interface and, finally, ensure that client classes request concrete objects using the creator method for which multiple implementations might also exist.

Factory Method Pattern – Generic Overview



Factory Method Pattern – Example

- In the DemoTasks Project, how to hide and decouple the Person class from concrete CategoryRepository implementations?



Factory Method Pattern – Advantages and Usages

- Advantages
  - Allows subclasses to choose the type of objects to be created
  - Promotes High Cohesion and Low Coupling since the client code only interacts with the resulting interfaces (or abstract classes)
  - Open/Closed Principle and Single Responsibility Principle are also promoted

- Usages
  - When a client class does not know what type of objects can be created
  - When a client class wants its subclasses to decide the type of objects to be created
  - When creating concrete objects is (very) complex

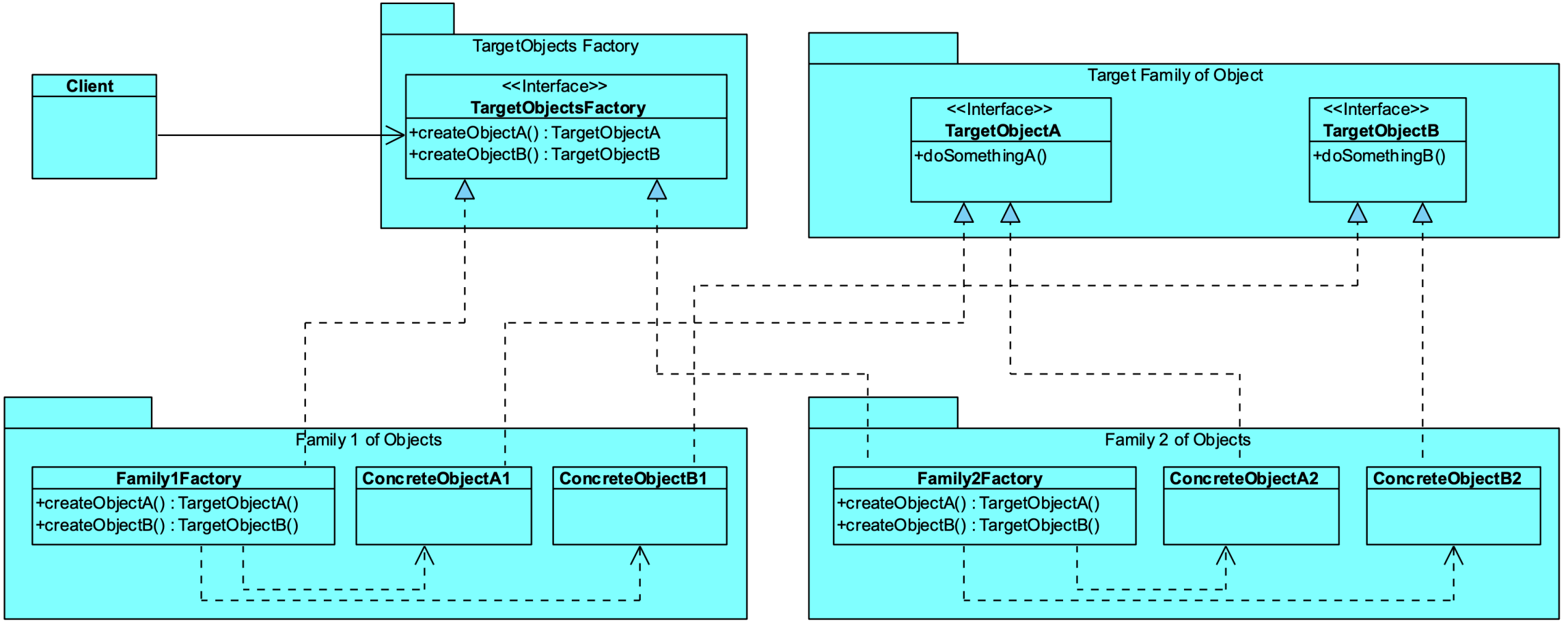
Abstract Factory Pattern

Abstract Factory Pattern

- **Problem**
  - How to hide and decouple client classes from the complexity of creating families of objects, such that the objects of a given family were designed to work together?
    - E.g.: In the DemoTasks Project one might considered that all domain objects should be persisted in the same place/database. As so, per each database provider, one family of repositories (for Category and Task) will exist. How to hide and decouple the Person class from this complexity?

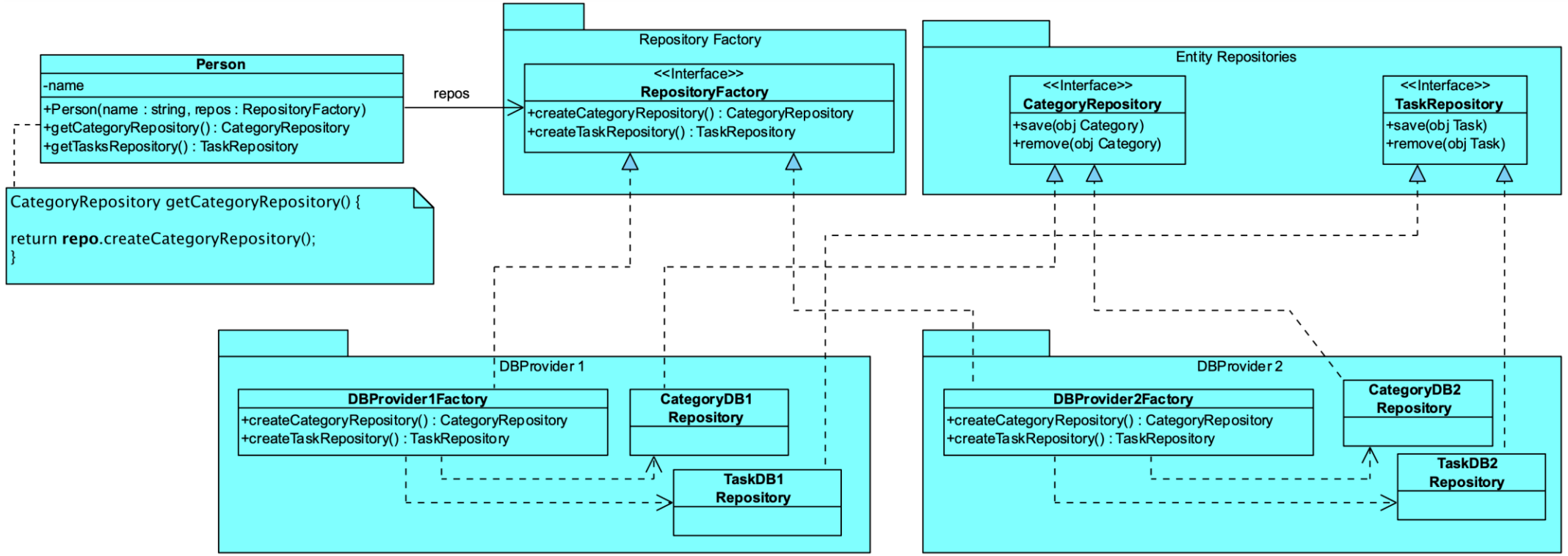
- **Solution**
  - For each family create a factory (FamilyFactory). All family factories should implement a common shared interface providing method to create the concrete objects of such family. Concrete objects are returned as instances of another shared interface.

Abstract Factory Pattern – Generic Overview



Abstract Factory Pattern – Example

- In the DemoTasks Project, each database provider represents a family of Repository objects



Abstract Factory Pattern – Advantages and Usages

- Advantages
  - Ensures that the objects being used belong to the same family
  - Promotes High Cohesion and Low Coupling since the client code only interacts with the resulting interfaces (or abstract classes)
  - Open/Closed Principle and Single Responsibility Principle are also promoted

- Usages
  - When a client class does not know which family of objects will be created
  - When a set of objects has been designed to work together
  - When creating concrete objects is (very) complex

Factory Method vs. Abstract Factory

- Methods of an Abstract Factory can be seen as Factory Methods
  - Both decouple the client classes from the concrete implementation classes through interface (or abstract classes)
  - Usually, Factory Methods create objects through inheritance while Abstract Factories create object through composition
- Factory Method: suitable to create objects that derive from a particular base class
- Abstract Factory: suitable to have a factory that is used to create other factories which, in turn, create objects derived from a base class. The intent is to create a collection of related objects.

Practical Open Issues in the DemoTasks Project

Using a Configuration File to set up the Repository Factory

Q1: Person vs. Data Persistence

- As seen, multiple database providers for data persistence can be supported by applying the Repository and Abstract Factory patterns
  - For each database provider (or distinct database library) there will be a distinct set of *{XXX}Repository* implementations, as well as a *RepositoryFactory* implementation to ensure that all repositories belong to the same “family”.
- Why are the repositories to be used dependent on each Person object?
  - It is a decision grounded on some business logic. Since all manipulated data (i.e. categories and tasks) belong to a certain person, he/she may select his/her preferred persistence mechanism from those available/supported.
  - Such selection might be made, for instance, when registering to use the application.
- As the DemoTasks application simulates a single user, his/her data (e.g. its name) as well as his/her choice regarding data persistence can be saved, for example, in an application configuration file (called, for exemple, “config.ini”) which is read every time the application starts-up.

Q1: Configuration File Example

- Two parameters are being defined: PERSON_NAME and PERSISTENCE_MODE

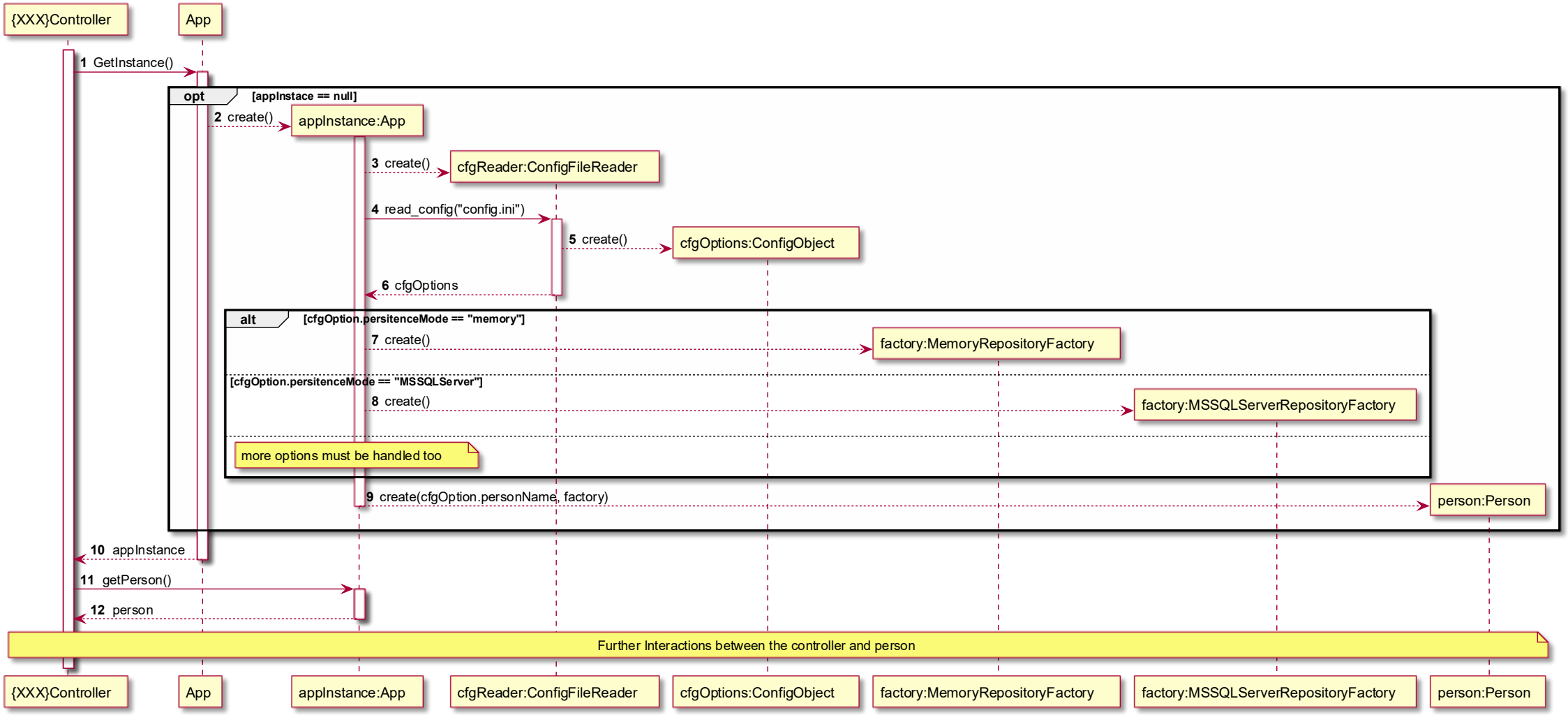
```
# This is a configuration file in standard configuration file format
# Lines beginning with a hash or a semicolon are ignored by the application program.
# Blank lines are also ignored by the application program.

# Set the name of the person using the application
PERSON_NAME Micky

# Persistence mode can be one of the following values: memory, MSSQLServer, Oracle.
# Default value is "memory"
PERSISTENCE_MODE memory

# Other options
```

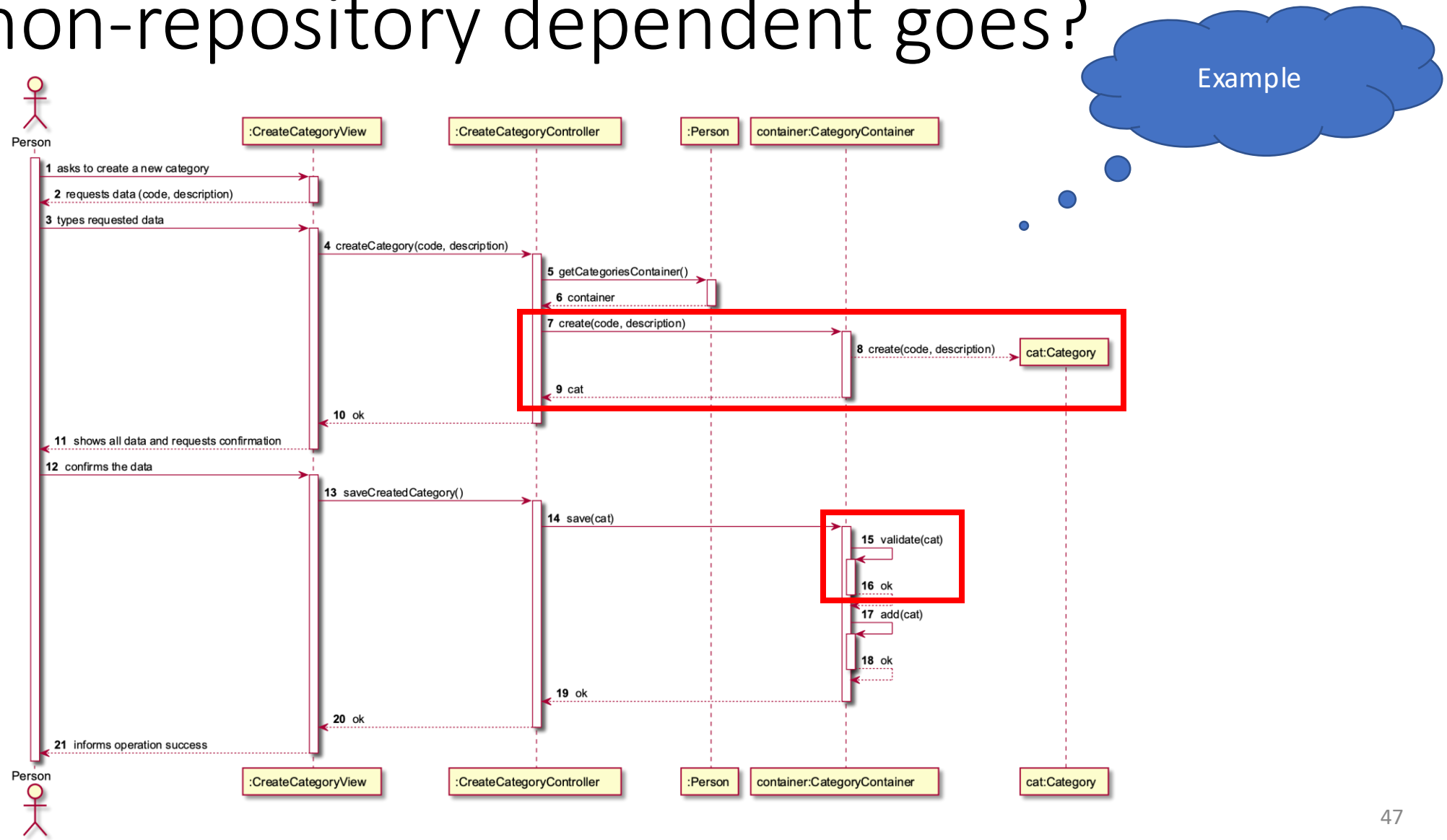
Q1: Initialization Flux



Practical Open Issues in the DemoTasks Project

Fully decoupling Business Logic from Persistence Logic through Services

Q2: Where does the common logic that is non-repository dependent goes?



Q2: What were the Container's responsibilities?

- First set of responsibilities (**business responsibilities**):
  - Ensure business logic regarding the related business entities (e.g. category). E.g.:
    - Creating an instance either from primitive data types or from a known DTO
    - Checking/Avoiding adding duplicated entities
- Second set of responsibilities (**persistence responsibilities**):
  - Persisting/Retrieving the related business entities (e.g. using an in-memory list)
  - Strongly coupled with a concrete persistence method

It violates the Single Responsibility Principle !!

Q2: Splitting Container responsibilities

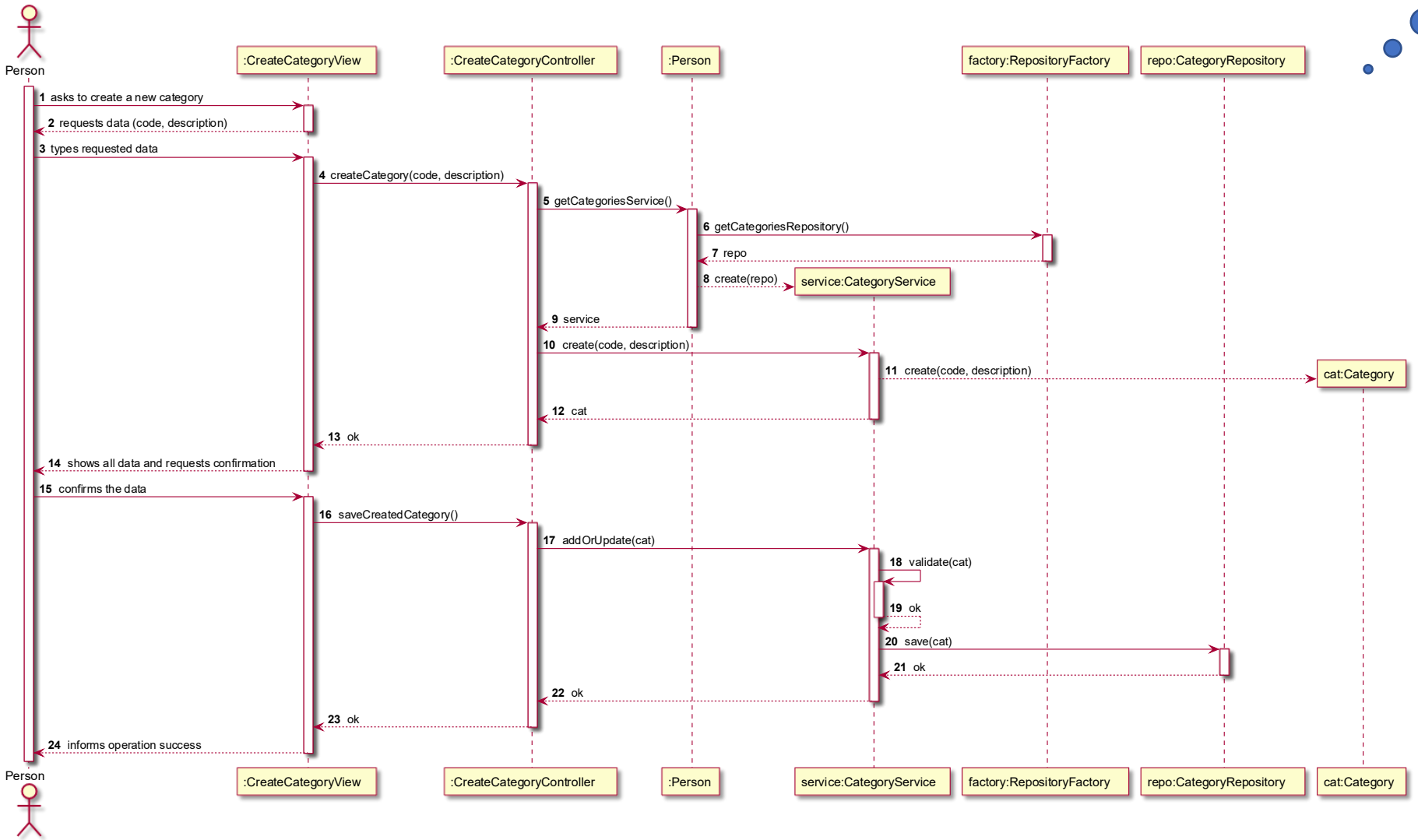
- **Persistence responsibilities**
  - Are delegated to any object implementing a given repository interface (e.g. *CategoryRepository*)
  - Are no longer coupled to any concrete persistence method

- **Business responsibilities**
  - Cover all common logic that is non-repository dependent
  - Suggestion: rename “Container” classes to “Service” classes
    - It is more coherent with well know best practices

This way, the SRP is no longer violated !!

Q2: Proposed solution — Example

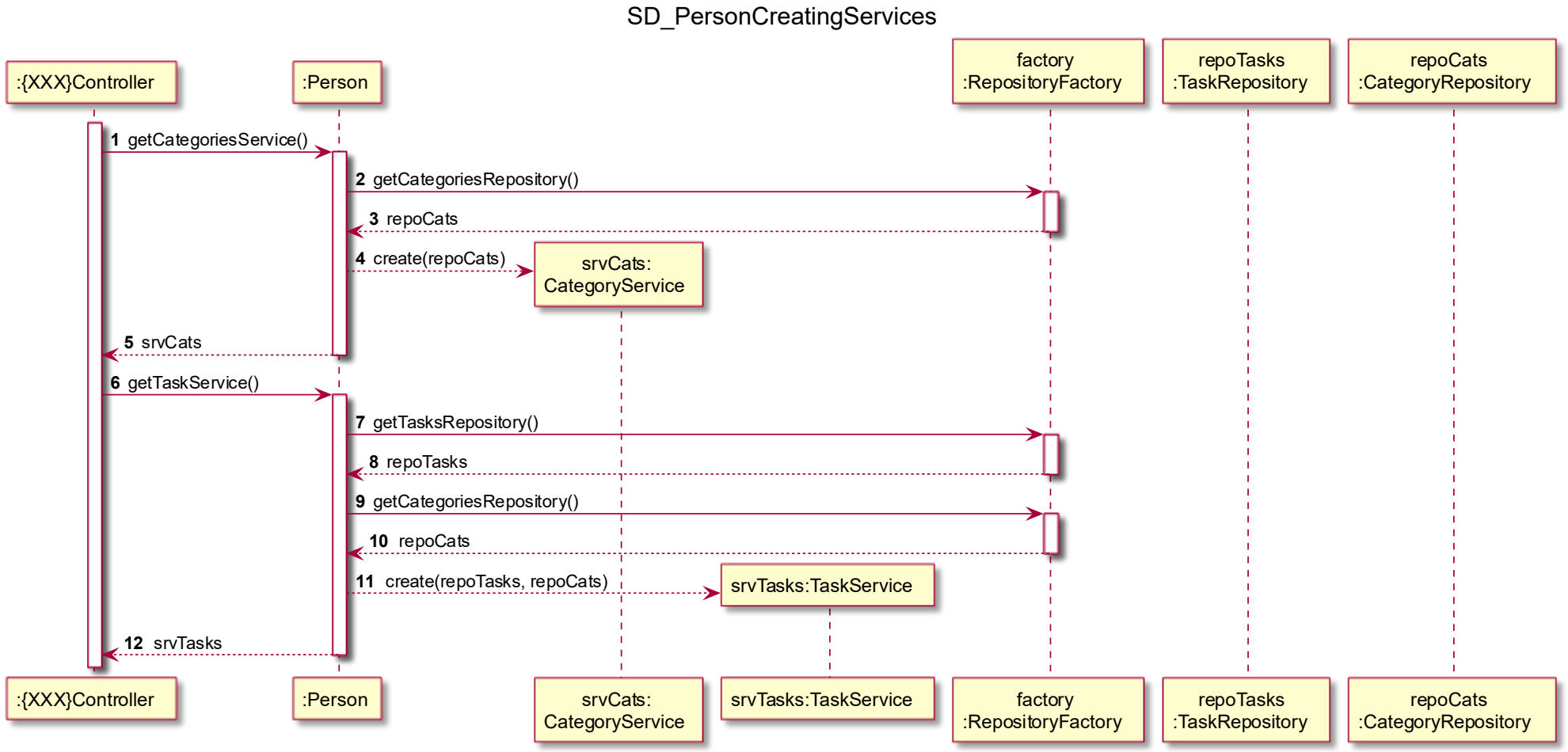
DemoTasks
Project **US01**



Service classes

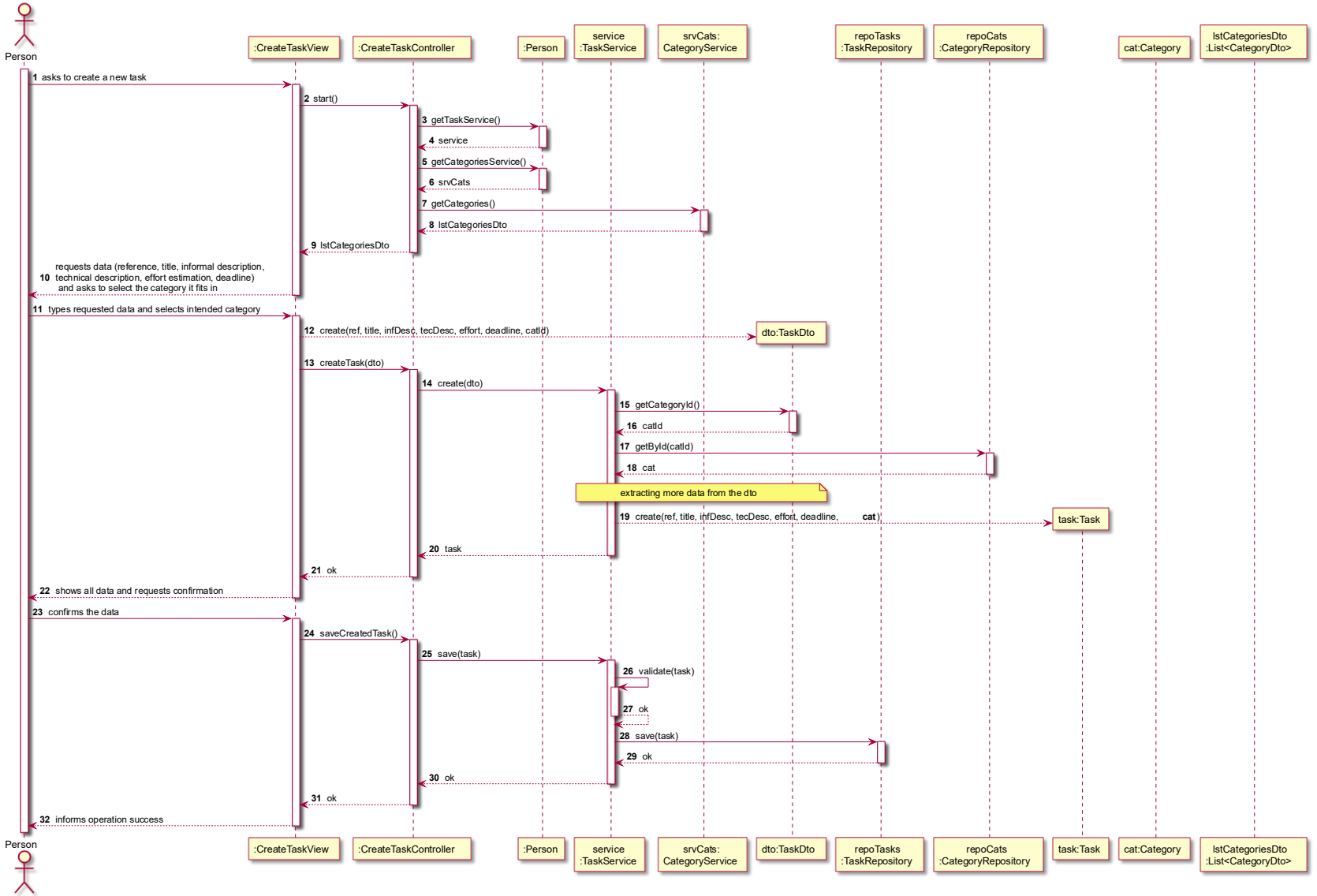
- Responsible for implementing the business logic that, by the application of other principles (e.g. Information Expert, Tell Don't Ask, SRP), should not be assigned to any domain class (e.g. Category, Task)
- Each service class supports the realization of one (or more) use cases related to a single business entity. E.g.:
  - *CategoryService*: supports the realization of UC related with the Category entity
  - *TaskService*: supports the realization of UC related with the Task entity
- Usually, these classes are stateless
  - There is no need to have (or keep) an (internal) state
  - They can be created when needed, used to complete a UC, and then destroyed without any loss of information

Q2: Creating Services (on any US)



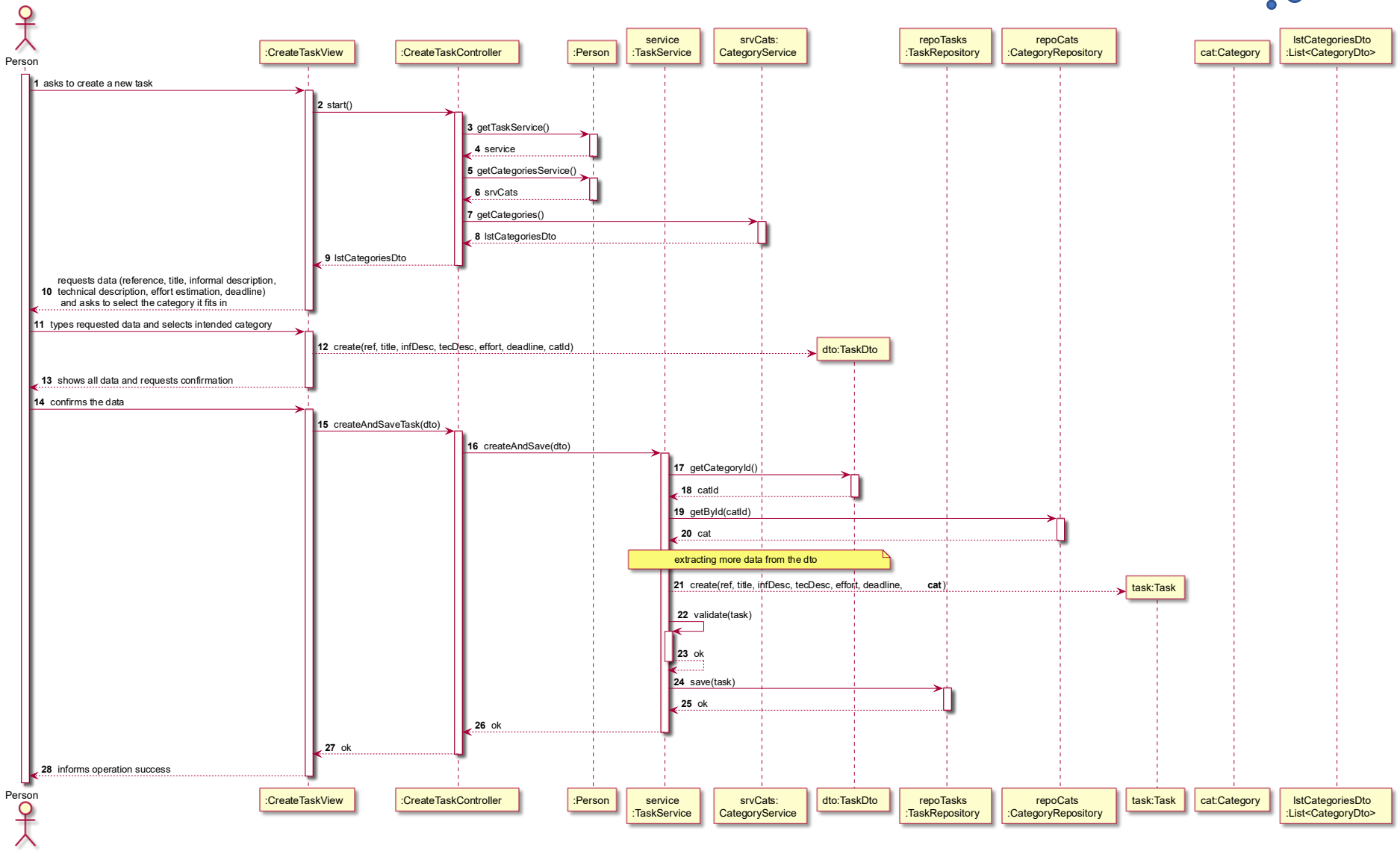
Q2: Creating a Task – Alternative 1

DemoTasks
Project **US05**



Q2: Creating a Task – Alternative 2

DemoTasks
Project **US05**



Revising *DemoTasks* Core Architecture

- **Controllers:** seen as pure intermediaries between the UI Layer and the Domain Layer
- **Domain Layer**, comprising:
  - **Model:** as the set of classes that capture domain concepts, their relationships and related business logic
  - **Services:** as a set of pure fabrication classes that are responsible for the business logic related to use cases, which could hardly be assigned to (domain) model classes
  - **Repositories:** as a set of interfaces that specify the API used by the domain layer to save/retrieve its model entities to/from any concrete data storage mechanism
  - **DTO and Mappers:** as a set of classes whose concern is to avoid exposing model entities directly to other layers
- **Infrastructure:** as set of at least one concrete implementation of each interface specified at the repositories level of the domain layer

Summary

- GoF Design Patterns:
  - Are focused on more specific problems than GRASP and SOLID principles
  - Present generic, but robust, solutions to repeatedly occurring problems in OO SW development
- The **Singleton pattern** should be carefully adopted → use it wisely!
- The **Adapter pattern** promotes Open/Closed and Single Responsibility principles, as well as High Cohesion and Low Coupling
- The **Factory Method and Abstract Factory patterns** are suitable to decouple client classes from creating instances of concrete implementation classes
- Knowing and Adopting more Design Patterns help us evolving the core architecture of the DemoTask Project:
  - To satisfy new functional and non-functional requirements
  - To improve some SW quality metrics such as maintainability, extensibility, and testability

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
- Gang of Four Design Patterns. Available on: https://springframework.guru/gang-of-four-design-patterns
- The Catalog of Design Patterns. Available on: https://refactoring.guru/design-patterns/catalog