|  |
| --- |
| FAVV – AFSCA |
| favv.gif |
| **[Technical Fiche : MediatR]** |

Document Information

|  |  |
| --- | --- |
| Information | Description |
| Document owner | Emmanuel Nuyttens |
| Creation date | 14/02/2020 |
| Last update date |  |
| Document version | 1.0 |

Document history

|  |  |  |
| --- | --- | --- |
| Version | Date | Modification |
| 1.0 | 14/02/2020 | Initial Created |
|  |  |  |
|  |  |  |
|  |  |  |

Document validation

|  |  |  |  |
| --- | --- | --- | --- |
| Nom | Role | Department | Date |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Document lecture references

|  |  |
| --- | --- |
| Title | Authors |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Contents

Aucune entrée de table des matières n'a été trouvée.

1. Introduction

In software development, the **mediator pattern** defines an **object that encapsulates how a set of related object interact**. This pattern **is considered to be a behavioral pattern** due to the way it can **alter the applications running behavior.**

With the mediator pattern**, communication between objects is encapsulated within a mediator object**. Objects no longer communicate directly with each others, but instead **communicate through the mediator**. This reduces the dependencies between communicating objects, thereby reducing tight-coupling.

1. Overview

What kind of problems can the Mediator pattern solve ?

1. Tight coupling between a set of interacting objects.
2. Change interaction between a set of objects independently.

Defining a set of interacting objects by accessing and updating each other directly is inflexible because it introduces tight-coupling to each other and makes it almost impossible to change the interaction independently from each others. This tight-coupling also make the concerned objects hard to be reusable and also makes them hard to be testable.

What kind of solution does the Mediator design pattern describe ?

* Define a separate (mediator) object that encapsulates the interaction between a set of objects.
* Objects delegate their interaction with other Objects to a mediator object instead of interacting with each other directly.

1. Definition

The essence of the Mediator Pattern is to “define an object that encapsulates how a set of objects interact”. It promotes loose-coupling by keep objects from referring to each other explicitly, and it allows their interactions to be varied independently. Client classes use the Mediator to send messages to other clients, and can receive messages from other Clients though an event on the Mediator class.

|  |  |
| --- | --- |
| mediator-design-pattern-in-c-uml-diagram | On the left is an UML diagram representing a typical “mediator” implementation. A Mediator will typically have reference to what they call “Colleague” object instances which would, by non-existence of the Mediator call each other directly. We want to avoid this by introducing the Mediator in between. Mediator will have reference to the Colleague objects and execute actions on their behalf when they have to communicate between each other. |

1. Creating a Custom Mediator

In this part we will create a Custom Mediator, and in next part we will replace this Custom MediatR with an implementation of the MediatR library create by Jimmy Bogard (<https://github.com/jbogard/MediatR>).

We will implement the same scenario we started from when we implemented the **BPT-CQRS persistence** for the .NET Full framework 4.8, so we will start from the student registration domain model as shown below:

|  |  |
| --- | --- |
|  | Core object is the **Student** class, where each student is uniquely defined by its **email**, a student has a name and a possible collection of **Enrollments** and **Disenrollment’s** to/from a C**ourse.**  The number of enrollments that a **Student** can have is limited to 2. Each **Enrollment** has a **Student, Course** and **Grade.**  The **Course** class has a unique name and a number of **Credits** that a **Student** can receive for the Course.  The **Disenrollement** class shows the **Courses** which where discontinued for a certain **Student**.  **Comment** and **Grade** are mandatory fields. So when a **Student** enrolls for a course, a Grade should be specified. And when a **Student** discontinues a course, a **Comment** of discontinuation should be added. |

But because we want to test the Mediator pattern, implementation of the Command and Event handling methods will be a bit different. For our playground, we will start to implement a single scenario:

* RegisterStudentCommand:
  + When a user is registered in the student registration system, he has to provide his name and email and enter 2 courses he will enroll in with appropriate Grade.
  + We will have a class called RegisterCommand and RegisterCommandHandler.
* StudentRegisteredEvent:
  + This event will be published after a student has been registered. We will have 3 parties who are interested in the registration of a student :