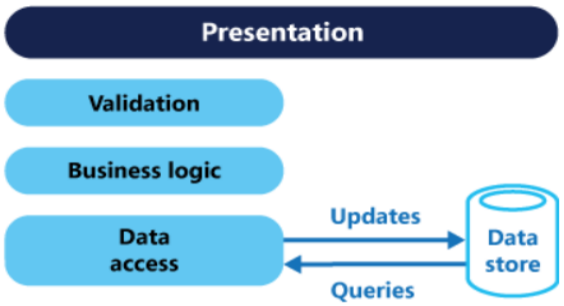
Nume: Tamas Horvath
An: 4 IS
Grupa: 3.1
Tema 2 PSSC: CQRS (Command Query Responsibility Segregation)

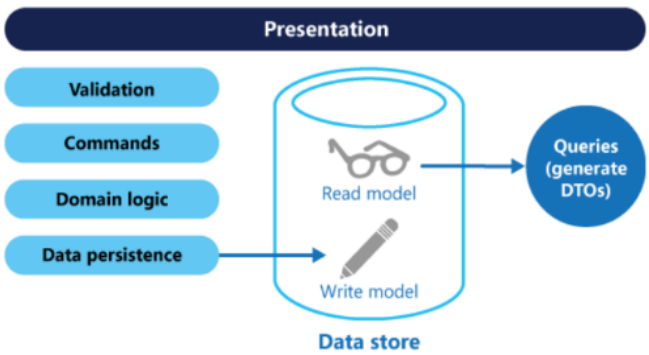
În sistemele tradiționale de gestionare a datelor, comenzile care pot fi executate pe o baza de date sunt query-urile(cereri de date) sau modificarile(update-uri, delete-uri, etc). Aceste comenzi se execută pe acelasi entity set dintr-un singur repository. In mod normal, in aceste sisteme, toate operatiile CRUD(create, read , update, delete) sunt aplicate pe aceeasi reprezentare a entitatii. Problema principala este ca aceasta abordare functioneaza bine cu o logica de business limitata si aduce in discutie diferite dezavantaje, limitand capacitatile sistemului. Rezolvarea problemelor aduse de limitarile sistemelor descrise mai sus este folosirea pattern-ului CQRS.

Folosirea unei abordari traditionale de tip CRUD intr-un astfel de sistem complex poate sa para abordarea mai simpla si mai accesibila deoarece anumite tool-uri pot sa genereze acces la date foarte rapid. In plus, acestea pot fi adaptate foarte usor pentru nevoile din cadrul proiectului. Totusi, abordarea clasica poate sa aduca dezavantaje semnificative, acestea fiind:



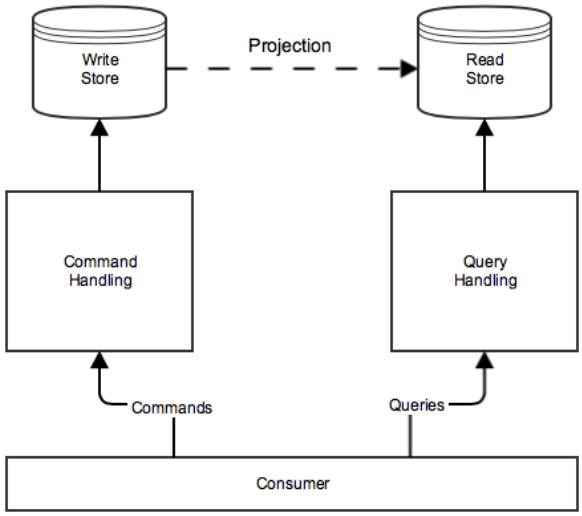
- Pot aparea inconsistente intre reprezentarea datelor citite si scrise, cum ar fi coloane in plus sau propietati care trebuie schimbate chiar daca nu sunt relevante operatiei executate.
- Exista riscul de insconsistente atunci cand se lucreaza in paralel cu acelasi set de date si au loc modificari pe tabele. Cu cat complexitatea sistemului si numarul actorilor creste, apare riscul de conflicte din cauza modificarilor concurente si riscul de a pierde din performanta aplicatiei.
- Gestionarea securității și permisiunilor risca să devina mai complexă; fiecare entitate este supusă atât operațiilor de citire cât și de scriere.

CQRS este un pattern destul de restrans, care poate fi implementat relativ usor, chiar daca nu exista tool-uri care sa genereze codul automat ca in cazul design-urilor de tip CRUD. Acest pattern este inspirat din CQS(Command Query Separation), al carui idee principala este ca o metoda trebuie fie sa schimbe starea unui obiect fie sa returneze un rezultat, niciodata ambele. Prin acest principiu, putem sa impartim tipul metodelor in 2 categorii:



- Comenzi: schimba starea unui obiect sau a sistemului
- Query-uri: returneaza rezultate fara a modifica starea obiectelor

In practica, separarea celor 2 categorii este facila si intuitiva. Query-urile sunt cele care returneaza ceva si comenzile sunt cele de tip void. Totusi, CQRS este aplicabil doar in cazuri specifice, deoarece si acest pattern vine cu probleme care trebuie luate in considerare. Separarea data store-ului in 2 entitati pentru citire si modificare poate creste performanta si securitatea, dar in acelasi timp creste complexitatea si consistenta sistemului. Problema principala este ca store-ul de unde are loc citirea, trebuie sa fie mereu la zi cu store-ul unde au loc modificarile. Astfel, trebuie luate in considerare momentele in care un utilizator a solicitat date, dar cele 2 store-uri nu sunt inca sincronizate. Aceste probleme pot fi tratate, intr-o anumita masura, prin folosirea event sourcing-ului in paralel cu CQRS.



Un avantaj foarte clar al acestui pattern este ca, in cazul aplicatiilor foarte mari, cele 2 parti ale sistemului pot fi implementate de echipe separate, fara ca ele sa trebuiasca sa interactrioneze. Totusi, cel mai mare avantaj adus de CQRS este faptul ca metodele care schimba starile obiectelor sunt separate de cele care nu. In acest fel, este mult mai usor sa se imbunatateasca partea de performanta, deoarece metodele de citire pot fi alterate fara a influenta partea de comenzi in vreun fel.

Urmatorul exemplu este o implementare destul de simpla a arhitecturii CQRS pentru o universitate. Aici este modelul de citire a datelor:

```
namespace Readmodel
{
    public interface StudentsDao
    {
        StudentDisplay FindById(int nrmatricolstudent);
        ICollection<StudentDisplay> FindByName(string name);
        ICollection<StudentsGrades> FindStudentsGrades(int
nrmatricolstudent);
        ICollection<StudentDisplay> FindCompScienceStudent(string faculty);
    }

    public class StudentDisplay
    {
        public int nrmatricolstudent { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
```

```
        public string Faculty { get; set; }
        public float Average { get; set; }
        public IList<string> PassedExams { get; set; }
        public IList<int> Grades { get; set; }
    }

    public class StudentsGrades
    {
        public int nrmatricolstudent { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
        public IList<int> Grades { get; set; }
    }
}
```

Si aici este implementarea modelului de comanda:

```
    public interface ICommand
    {
        Guid Grade { get; }
    }

    public class ChangeGrade : ICommand
    {
        public ChangeGrade()
        {
            this.Grade = Guid.NewGuid();
        }
        public Guid Grade { get; set; }
        public int nrmatricolstudent { get; set; }
        public int Name { get; set; }
        public int Surname { get; set; }
    }
```