Unit Of Work და Repository დიზაინ პატერნები ტვირთის იმპორტის მაგალითზე

არჩილ მჭედლიშვილი
საქართველოს ტექნიკური უნივერსიტეტი

რეზიუმე

განხილულია Unit Of Work და Repository დიზაინ პატერნები, რაში გამოიყენება ისინი. რა იყო იმის საჭიროება რომ ეს დიზაინ პატერნები შექმნილიყო და რა მხრივ ამარტივებს დეველოპმენტს არსებულ სისტემებში ან ცვლილების შეტანისას. მოცემულია რეკომენდაციები Unit Of Work და Repository დიზაინ პატერნების გამოსაყენებლად დიდი და რთული ბიზნეს-ლოგიკის მქონე სისტემებში.

საკვანძო სიტყვები: სისტემების დაპროგრამება. პატერნი. Unit Of Work. Repository. ლოგისტიკა. პროგრამების ტესტირება.

1. შესავალი

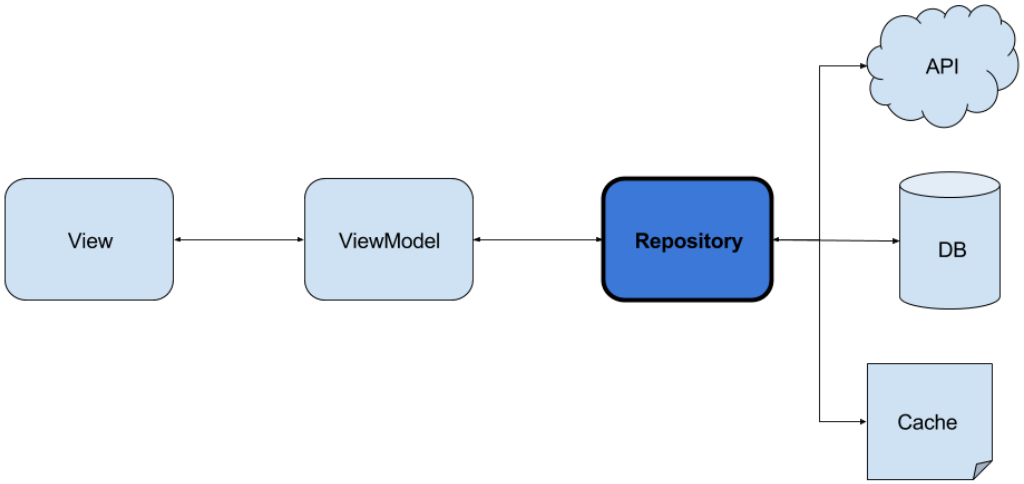
ნებისმიერი კომპიუტერული სისტემის ან პროგრამული აპლიკაციის დეველოპმენტის პროცესში, რომელსაც სჭირდება მონაცემთა ბაზები და/ან ვებ სერვისები მონაცემების დასამუშავებლად, დეველოპერები ხშირად აწყდებიან ინფორმაციის მიღების წყაროს მიერ შეცვლილი ობიექტების სტრუქტურის ცვლილების პრობლემას. ასევე შესაძლებელი იყო მონაცემთა შეცვლილი ტიპების მიღების შემთხვევები (მაგალითად, რიცხვითი ველი გადაკეთებულიყო ტექსტურ ველად), რაც დიდ პრობლემას იწვევს იმ მხრივ, რომ საჭირო ხდება, შესაბამისად, თავისი ბიზნეს ლოგიკების გარჩევა, ანალიზი და ცვლილება პრობლემის აღმოსაფხვრელად. ეს კი დიდ ძალისხმევას და დროის ფუჭ კარგვას ნიშნავს.

Unit Of Work და Repository დიზაინ პატერნები სწორედ ასეთი პრობლემების მარტივად მოსაგვარებლად შეიქმნა [1]. ეს არის მიდგომა რომელიც დეველოპერს უმარტივებს საქმეს მომწოდებლის მხრიდან გარკვეული ცვლილებების შემთხვევაში, თავის კოდში უკვე მცირე ცვლილებით და მცირე დროის დანაკარგით სისტემა ისევ დაუბრუნოს მუშა მდგომარეობას. ასეთი პროცესში დეველოპერს უკვე საათები და დღეები აღარ დასჭირდება პრობლემის მოსაგვარებლად.

2. Repository დიზაინ პატერნი

დიზაინ პატერნები გამოიყენება აპლიკაციებში განმეორებადი პრობლემების გადასაწყვეტად. Repository მოდელი არის ერთ-ერთი ყველაზე ფართოდ გამოყენებული დიზაინ პატერნი. იგი არის ობიექტი, რომელიც ბაზის თითოეული ცხრილისათვის უნდა შეიქმნას Service ან Data დონეებში. მათი საშუალებით ხდება ბაზის ცხრილზე CRUD (Create Read Update Delete) ოპერაციების რეალიზება. Repository-ით ჩვენ ვიღებთ მონაცემების წყაროს, რომელსაც შემდგომ ბიზნეს შრეში გამოვიყენებთ. იმ შემთხვევაში, თუ მოგვიწევს მონაცემთა ბაზაში ცხრილის ან სერვისში მეთოდის მიერ დაბრუნებული ობიექტის სტრუქტურის ცვლილება,

მხოლოდ რეპოზიტორში არსებული მეპერის შეცვლით შევძლებთ დიდი ცვლილებების თავიდან არიდებას, როგორიცაა დიდი ბიზნეს ლოგიკების გადაკეთება და გადაწყობა ახალ, უკვე შეცვლილ მონაცემთა წყაროზე.



ნახ.1. Repository პატერნის დიზაინის მაგალითი

არსებითად, Repository დიზაინ პატერნი ხელს უწყობს ბიზნეს ლოგიკის დაწყვილებასა და გამოყენებულ მონაცემთა წვდომას იმ პირებთან, რომელთაც არ გააჩნიათ რაიმე ცოდნა იმის შესახებ, თუ საიდან იღებს სათავეს არსებული მონაცემები [2].

Repository დიზაინ პატერნის გამოყენებისას შეიძლება დაიმალოს მონაცემების საცავში განთავსების და მათი ამოღების შესახებ დეტალები. ეს მონაცემთა საცავი შეიძლება იყოს მონაცემთა ბაზა, xml ფაილი და ა.შ. შესაძლებელია ამ დიზაინ პატერნის გამოყენება იმ შემთხვევაშიც, როცა არ გვსურს გამოვაჩინოთ, თუ როგორ აწვდის მონაცემებს: ვებ-სერვისით თუ ORM-ს მიერ [1].

Repository განისაზღვრება, როგორც მეხსიერებაში არსებული დომენის ობიექტების კოლექცია [3]. Repository დიზაინ პატერნი გამოიყენება როგორც ლოგიკის გამყოფი, რომელიც ამოარჩევს მონაცემებს და შეუსაბამებს მათ ბიზნეს-ლოგიკის არსთა მოდელს. ბიზნეს ლოგიკა არ უნდა იყოს დამოკიდებული მონაცემთა ტიპებზე, რომლებიც ქმნის მონაცემების წყაროს შრეს. მაგალითად, მონაცემთა წყარო შეიძლება იყოს მონაცემთა ბაზა, SharePoint სია ან ვებ სერვისი [3].

Repository დიზაინ პატერნი მოიცავს შემდეგს [1]:

1) IRepository ინტერფეისს - ეს ინტერფეისი არის ყველა Repository კლასის მშობელი ინტერფეისი.

```
public interface IRepository<T> where T : class
{
    IQueryable<T> GetAll();
    IQueryable<T> Where(Expression<Func<T, bool>> predicate);
    int Count();
    int Count(Expression<Func<T, bool>> predicate);
    bool Any();
    bool Any(Expression<Func<T, bool>> predicate);
```

```
        T FirstOrDefault();
        T FirstOrDefault(Expression<Func<T, bool>> predicate);
        void Add(T entity);
        void AddRange(List<T> entities);
        void Remove(int Id);
        void Remove(T entity);
    }
```

2) Repository class - ეს არის ზოგადი Repository კლასი;

3) ერთ ან მეტ Repository კლასებს, რომლებიც აიმპლემენტებენ IRepository ინტერფეისს.

```
public class Repository<T> : IRepository<T> where T : class
    {
        protected DbContext DbContext { get; set; }

        protected DbSet<T> DbSet { get; set; }

      public Repository(DbContext dbContext)
      {
         DbContext = dbContext ?? throw new NullReferenceException("DbContext is
                null");
         DbSet = DbContext.Set<T>();
      }
      public void Add(T entity)
        {
            DbSet.Add(entity);
        }
      public void AddRange(List<T> entities)
        {
            DbSet.AddRange(entities);
        }
      public bool Any(Expression<Func<T, bool>> predicate)
        {
            return DbSet.Any(predicate);
        }
      public int Count(Expression<Func<T, bool>> predicate)
        {
            return DbSet.Count(predicate);
        }
      public T FirstOrDefault()
        {
            return DbSet.FirstOrDefault();
        }
      public T FirstOrDefault(Expression<Func<T, bool>> predicate)
        {
            return DbSet.FirstOrDefault(predicate);
        }
      public IQueryable<T> GetAll()
        {
         return typeof(T).GetProperty("IsDeleted") != null ? DbSet.Where
                      ("IsDeleted == @0", false) : DbSet;
        }
      public void Remove(int Id)
        {
            var entity = DbSet.Find(Id);
```

```
            if (entity == null)
              throw new NullReferenceException("Cannot find Record in Database");

              DbSet.Remove(entity);
          }
        public void Remove(T entity)
          {
              DbSet.Remove(entity);
          }
        public IQueryable<T> Where(Expression<Func<T, bool>> predicate)
          {
              return DbSet.Where(predicate);
          }
      }
```

3. UnitOfWork დიზაინ პატერნი

UnitOfWork და Repository დიზაინ პატერნები მიზნად ისახავს იყოს აბსტრაქციის შრე ბიზნეს ლოგიკასა (BL) და მონაცემთა წვდომის შრეს (Data Access Layer) შორის [4]. იგი ხელს უწყობს აპლიკაციას თავიდან აიცილოს მონაცემთა საცავში განსახორციელებელი ცვლილებები და შეუძლია ხელი შეუწყოს ავტომატიზებულ unit-ტესტირებას (ტესტირებაზე ორიენტირებულ დეველოპ-მენტს EntityFramework-ის გამოყენების საფუძველზე.

UnitOfWork დიზაინ პატერნი ახდენს თითოეული რეპოზიტორის თავის თავში რეალიზებას. შემდგომში, როდესაც მათი გამოყენება იქნება საჭირო, თითოეული რეპოზიტორის შემოტანა არ უნდა გახდეს საჭირო. მხოლოდ ერთი UnitOfWork-ის მოცემა საკმარისია ყველა რეპოზიტორის მისაღებად.

UnitOfWork-ის ინტერფეისი გამოიყურება შემდეგნაირად:

```
public interface IUnitOfWork
  {
    void Commit();
    string ConnectionString { get; }
    IRepository<PersonalDocument> PersonalDocument { get; }
    IRepository<Address> Address { get; }
    IRepository<Contact> Contact { get; }
    IQueryable<SP_ApproximatelySearchResult_Result>
   SP_ApproximatelySearchResult(string FirstName, string LastName, string Address,
            int NamePercent, int AddressPercent, int PageSize, int CurrentPage,
            int TotalResult);
}
```

UnitOfWork-ის კლასი შემდეგნაირია:

```
public class UnitOfWork : IUnitOfWork
  {
    private BaseDbContext BaseDbContext { get; set; }
    public UnitOfWork(string connectionString = null, string host = null)
    {
      CreateDbContext(connectionString, host);
```

```
    }
    public void Commit()
    {
        BaseDbContext.SaveChanges();
    }
    public string ConnectionString
    {
        get { return BaseDbContext.Database.Connection.ConnectionString; }
    }
    protected void CreateDbContext(string connectionString = null, string host = null)
    {
        BaseDbContext = BaseDbContext.CreateInstance<BaseDbContext>(connectionString, host);
    }
    public Contracts.IRepository<PersonalDocument> PersonalDocument => new
                Repository<PersonalDocument>(BaseDbContext);
    public Contracts.IRepository<BlackList> BlackList => new Repository<BlackList>(BaseDbContext);
    public Contracts.IRepository<Contact> Contact => new Repository<Contact>(BaseDbContext);
    public Contracts.IRepository<Address> Address => new Repository<Address>(BaseDbContext);
}
```

4. Unit Of Work და Repository დიზაინ პატერნები ტვირთის იმპორტში

Unit of Work და Repository დიზაინ პატერნები საქართველოში იმპორტირებული ტვირთის მართვის სისტემაში გამოყენებულია ყველა კომპონენტში, სადაც ხდება მონაცემებთან ურთიერთობა. დიზაინ პატერნების გამოყენების მიზეზი კი არის სწორედ ის, რაც ზემოთ, მათი შექმნის დროს ჩამოვაყალიბეთ - ესაა ცვლილებებზე სისტემის მდგრადობა.

5. დასკვნა

თუ დეველოპერს უწევს დიდ სისტემებთან მუშაობა და შესაბამისად დიდი და რთული ბიზნეს ლოგიკები უწერია თავის სისტემაში, კოდის ოპტიმიზაციის და დროის უკეთესი მენეჯმენტისთვის უმჯობესია, რომ აქ მონაცემებთან წვდომისას გამოიყენოს Unit Of Work და Repository დიზაინ პატერნები.

ლიტერატურა – References – Литература:

1. Joydip Kanjilal. (2016). How to implement the Repository design pattern in C#Info world. https://www.infoworld.com/article/3107186/application-development/how-to-implement-the-repository-design-pattern-in-c.html

2. Repository Pattern A data persistence abstraction. DevIQ. (2018). https://deviq-.com/repository-pattern/

3. The Repository Pattern. (2010). https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649690(v=pandp.10)

3. Karlsson K. (2017). C# - UnitOfWork And Repository Pattern for Data Store Insulation and Test Ddriven Development. https://medium.com/@utterbbq/c-unitofwork-and-repository-pattern-305cd8ecfa7a

4. Karlsson K. (2010). Entity Framework Tips. https://www.entityframeworktips.com/blog/c-unitofwork-and-repository-pattern/#download-section

UNIT OF WORK AND REPOSITORY DESIGN PATERNS ON THE EXAMPLE OF IMPORT OF GOODS

Mchedlishvili Archil

Georgian Technical University

Summary

The paper describes the Unit of Work and Repository design patterns where they are used. What was the need to create these design patterns, and in what way simplify the development of the existing systems or changes. Recommendations on using the Unit Of Work and Repository design patterns in systems with large and complex business logics is given.

UNIT OF WORK И REPOSITORY ШАБЛОНЫ ДИЗАЙНА НА ПРИМЕРЕ ИМПОРТА ТОВАРОВ

Мчедлишвили А.

Грузинский Технический Университет

Резюме

Рассмотрены Unit Of Work и Repository дизайн паттерны и сферы их применениячины создания дизайн паттернов и как они упрощают девелопмент существующих систем и внесение в них изменений. Приведены рекомендации по применению Unit Of Work и Repository дизайн паттернов в системах, которые характеризуются сложной бизнес-логикой.

**** <u>რედაქციის შენიშვნა:</u> სტატია იბეჭდება ავტორის მიერ წარმოდგენილი სახით.*