«Сейфуллин оқулары-18(2): «XXI ғасыр ғылымы – трансформация дәуірі» Халықаралық ғылыми-практикалық конференция материалдары = Материалы международной научно-практической конференции «Сейфуллинские чтения – 18(2): «Наука XXI века - эпоха трансформации» - 2022.- Т.I, Ч.III. - С.139-141.

# ОСОБЕННОСТИ РАЗРАБОТКИ АРХИТЕКТУРЫ ПРОГРАММНОГО ПРОДУКТА ДЛЯ МОНИТОРИНГА СТАТУСА ПРОЕКТОВ

*Беркамалов Б.С., магистрант 2-го курса*

*Казахский агротехнический университет им. С. Сейфуллина, г. Нур-Султан*

Введение. При выборе проектирования и построения архитектуры, необходимо учитывать текущие версии продуктов, удобство их сопровождения и расширение. В соответствии с основными принципами ООП, такими как: “Разделение задач”, “Инкапсуляция” и “Не повторяйся”, лучшим решением является применение многослойной архитектуры, где каждый слой содержит свой функционал [1].

1 Разделение задач в процессе формирования логики проекта

Один из важнейших принципов ООП. Он подразумевает собой разделение продукта на компоненты учитывая требуемый функционал. Важным советом является разделения пользовательского интерфейса и логики проекта (см. рисунок 1).

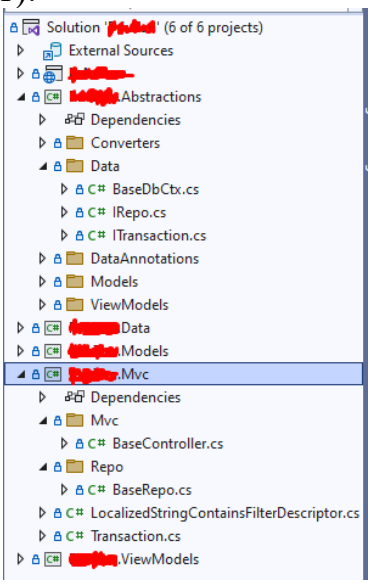


Рисунок 1 – Архитектура программного продукта

В данном варианте разработки проекта рассматривается реализация бизнес логики с использованием Базового интерфейсы для CRUD операций (IRepo<TEntity>), базового репозитория (BaseRepo<T>), наследующего все методы с базового интерфейса и базового абстрактного контроллера (BaseController<TEntity, TViewModel, TRepo>) (см. рисунок 2).

Это один из наиболее рациональных шаблонов проектирования архитекктуры, который позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает приложение, и является промежуточным звеном между классами, непосредственно взаимодействующими с используемыми данными и основным проектом [2].

2 Инкапсуляция частей приложения проекта

Инкапсуляция отдельных частей приложения позволяет изолировать их друг от друга. Правильно реализованная инкапсуляция способна получить низко связанную структура продукта [3].

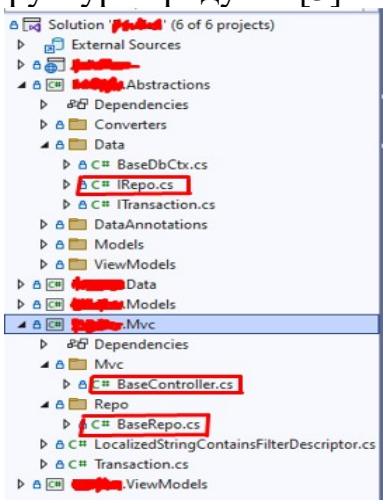
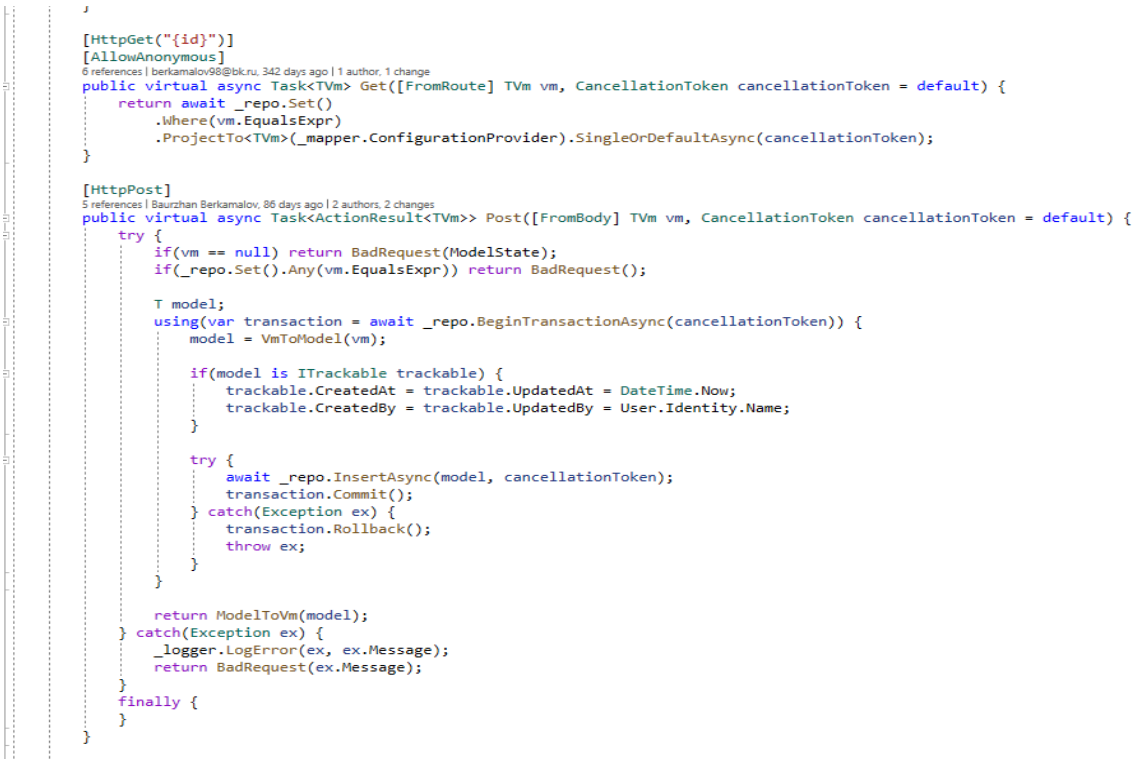



Рисунок 2 - Абстрактные классы

В данном случат слой “Data” зависит от “Models”, но слой “Models” полностью независим и внесение корректировок никак не носят рисков. Для предотвращения дублирования логики приложения, ее следует инкапсулировать в конструкции программирования. [4]. Не следует связывать поведение, которое повторяется нерегулярно. В рассматриваемом проекте созданы базовые классы и интерфейсы, позволяющие задать стандарт кодов для всего решения. Класс “BaseController” позволяет предотвратить написание идентичного кода для CRUD операций (см рисунок 3).



```
    }

    [HttpGet("{id}")]
    [AllowAnonymous]
    6 references | berkamalov98@bk.ru, 342 days ago | 1 author, 1 change
    public virtual async Task<TVm> Get([FromRoute] TVm vm, CancellationToken cancellationToken = default) {
        return await _repo.Set()
            .Where(vm.EqualsExpr)
            .ProjectTo<TVm>(_mapper.ConfigurationProvider).SingleOrDefaultAsync(cancellationToken);
    }

    [HttpPost]
    5 references | Baurzhan Berkamalov, 86 days ago | 2 authors, 2 changes
    public virtual async Task<ActionResult<TVm>> Post([FromBody] TVm vm, CancellationToken cancellationToken = default) {
        try {
            if(vm == null) return BadRequest(ModelState);
            if(_repo.Set().Any(vm.EqualsExpr)) return BadRequest();

            T model;
            using(var transaction = await _repo.BeginTransactionAsync(cancellationToken)) {
                model = VmToModel(vm);

                if(model is ITrackable trackable) {
                    trackable.CreatedAt = trackable.UpdatedAt = DateTime.Now;
                    trackable.CreatedBy = trackable.UpdatedBy = User.Identity.Name;
                }

                try {
                    await _repo.InsertAsync(model, cancellationToken);
                    transaction.Commit();
                } catch(Exception ex) {
                    transaction.Rollback();
                    throw ex;
                }
            }

            return ModelToVm(model);
        } catch(Exception ex) {
            _logger.LogError(ex, ex.Message);
            return BadRequest(ex.Message);
        }
        finally {
        }
    }
```

Рисунок 3 - Абстрактный базовый контроллер

Методы класса “BaseRepo” наследуются от базового интерфейса “IRepo” (см. рисунок 4).

Данные методы принимают любую сущность и производят транзакции, что позволяет отправить нужный класс сущности в наследуемых от “BaseController” контроллерах без повторной реализации под определенную ситуацию.

```
public interface IRepo<T>
    : IDisposable
    where T : class {

    IQueryable<T> Set();
    11 references | 0 changes | 0 authors, 0 changes
    Task<T> InsertAsync(T entity, CancellationToken cancellationToken = default);
    8 references | 0 changes | 0 authors, 0 changes
    Task<T> UpdateAsync(T entity, CancellationToken cancellationToken = default);
    2 references | 0 changes | 0 authors, 0 changes
    Task<bool> DeleteAsync(T entity, CancellationToken cancellationToken = default);
    10 references | 0 changes | 0 authors, 0 changes
    Task<ITransaction> BeginTransactionAsync(CancellationToken cancellationToken = default);
    4 references | 0 changes | 0 authors, 0 changes
    Task<bool> SaveAsync(CancellationToken cancellationToken = default);
}
```

Рисунок 4 – Базовый интерфейс

В данной статье был произведен обзор архитектурных принципов, паттернов и методов их применения в примерах реализованных решениях. В ходе исследования сделан вывод, что при выборе архитектуры

приложения следует, прежде всего, осуществлять правильное формулирование задач исследования и всегда учитывать возможное расширение приложения.

## Список использованной литературы


1 Jon Skeet C# in Depth. 2022 – 497 с.

2 Клири Стивен Конкурентность в c#. асинхронное параллельное и многопоточное программирование. 2019 – 304 с.

3 Steve S. Architect Modern Web Applications with ASP.NET Core and Azure. 2022.– 119 с.

4 Роберт Мартин, Мика Мартин Принципы, паттерны и методики гибкой разработки на языке C#. 2011. – 757 с.