УДК 004

Использование паттерна Abstract Factory языка программирования C# при создании графического интерфейса в среде Unity3D

Эрдман Александр Алексеевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Научный руководитель:
Глаголев Владимир Александрович
Приамурский государственный университет имени Шолом-Алейхема
К.г.н., доцент, доцент кафедры информационных систем, математики и правовой информатики

Аннотация
В статье описывается принцип работы паттерна Abstract Factory на примере создания UI в Unity3D. Реализация паттерна выполнена на языке программирования C# в среде программирования Visual Studio 2019. Результатом исследования является пример использования паттерна в программе графического интерфейса (UI).
Ключевые слова: C#, Unity3D, паттерны C#, графические интерфейсы

Using the Abstract Factory pattern of the C# programming language when creating a graphical interface in the Unity3D environment

Erdman Alexander Alekseevich
Sholom-Aleichem Priamursky State University
Student

Scientific supervisor:
Glagolev Vladimir Aleksandrovich
Sholom-Aleichem Priamursky State University
Ph.D, Associate Professor, Associate Professor of the Department of Information Systems, Mathematics and Legal Informatics

Abstract
The article describes how the Abstract Factory pattern works using the example of creating a UI in Unity3D. The pattern is implemented in the C# programming language in the Visual Studio 2019 programming environment. The result of the study is an example of using a pattern in a graphical interface (UI) program.
Keywords: C#, Unity3D, C# patterns, graphical interfaces

1 Введение

1.1 Актуальность

В современной сфере программирования, где требования к качеству и эффективности программных продуктов постоянно растут, использование передовых подходов и инструментов становится критически важным. Одним из таких инструментов являются паттерны проектирования, которые предоставляют проверенные и гибкие решения для часто встречающихся задач в разработке программного обеспечения. Из существующих паттернов проектирования выделятся Abstract Factory (Абстрактная фабрика). Данный паттерн востребован в программирование за счёт несложной структуры и предлагаемых возможностей. Одной из возможностей является создание нескольких видов объектов, похожих между собой. Это удобно использовать в программах, где необходимо создавать семейства взаимосвязанных или зависимых объектов без привязки к конкретным классам. Примером такой программы может выступать графический пользовательский интерфейс приложении (UI), в том числе и игровых. В статье примером использования паттерна Abstract Factor является программа интерфейса игры в среде Unity3D.

1.2 Обзор исследований

А.А. Пасынкова и О.Л. Викентьева в своей научной работе исследовали актуальность использования шаблонов проектирования при разработке архитектуры систем мониторинга [1]. В.В. Шишкин и В.М. Кандаулов рассмотрели основные принципы проектирования с использованием паттернов, преимущества такого подхода, а также структура, функции и реализация системы автоматизированного проектирования с помощью паттернов [2]. Н.Г. Талыбов в статье проанализировал основные принципы проектирования с использованием паттернов, преимущества такого подхода, а также структура, функции и реализация системы автоматизированного проектирования с помощью паттернов [3]. С.С. Валеев и Н.В. Кондратьева исследовали особенности проектировании систем безопасности на основе модели нулевого доверия и задачу разработки паттернов политики безопасности [4]. Н.А. Тоичкин в своей статье рассмотрел применение паттернов (шаблонов) проектирования для разработки информационной системы анализа и моделирования процессов, протекающих в сложных динамических системах [5].

1.3 Цель исследования

Целью исследования является описание принципа работы паттерна Abstract Factory и его применение в программе пользовательского интерфейса (UI), разработанного в среде Unity3D на языке программирования C#.

2 Материалы и методы

Для описания паттерна используется официальная документация языка программирования C#. Методологией описания структуры паттерна является

UML. В качестве IDE используется Visual Studio 2019. Разработка UI осуществляется по средствам Unity3D.

3 Результаты и обсуждения

Как упоминалось в актуальности исследования, Abstract Factory (Абстрактная фабрика) – это порождающий паттерн. Порождающие паттерны предназначены для создания и инициализации объектов.

Абстрактная фабрика предоставляет интерфейс для создания семейств взаимосвязанных объектов, но не специфирует конкретных классов. Среди особенностей данного паттерна выделяется основная, которая касается всех остальных порождающих паттернов – инкапсуляция. Данное свойство позволяет выносить логику создания объектов в отельные классы. Второй особенностью является то, что паттерн предназначен не только для создания объектов, а для семейства объектов. Именно создание семейства объектов является ключевым моментом Абстрактной фабрики, которое отличает его от всех остальных паттернов рассматриваемой категории. Под семействами подразумевается группа взаимосвязанных или взаимозависимых объектов. Таким образом семейством объектов могут быть те объекты, которые либо работают сообща друг с другом либо эти объекты используются группой за раз. Третье свойство представляет из себя возможность работы как и с фабриками (интерфейсами) так и с порождаемыми объектами на уровне абстракций и интерфейса. За счёт этого отсутствует привязка к конкретной реализации. Упрощённая схема паттерна выглядит следующим образом (рис. 1).

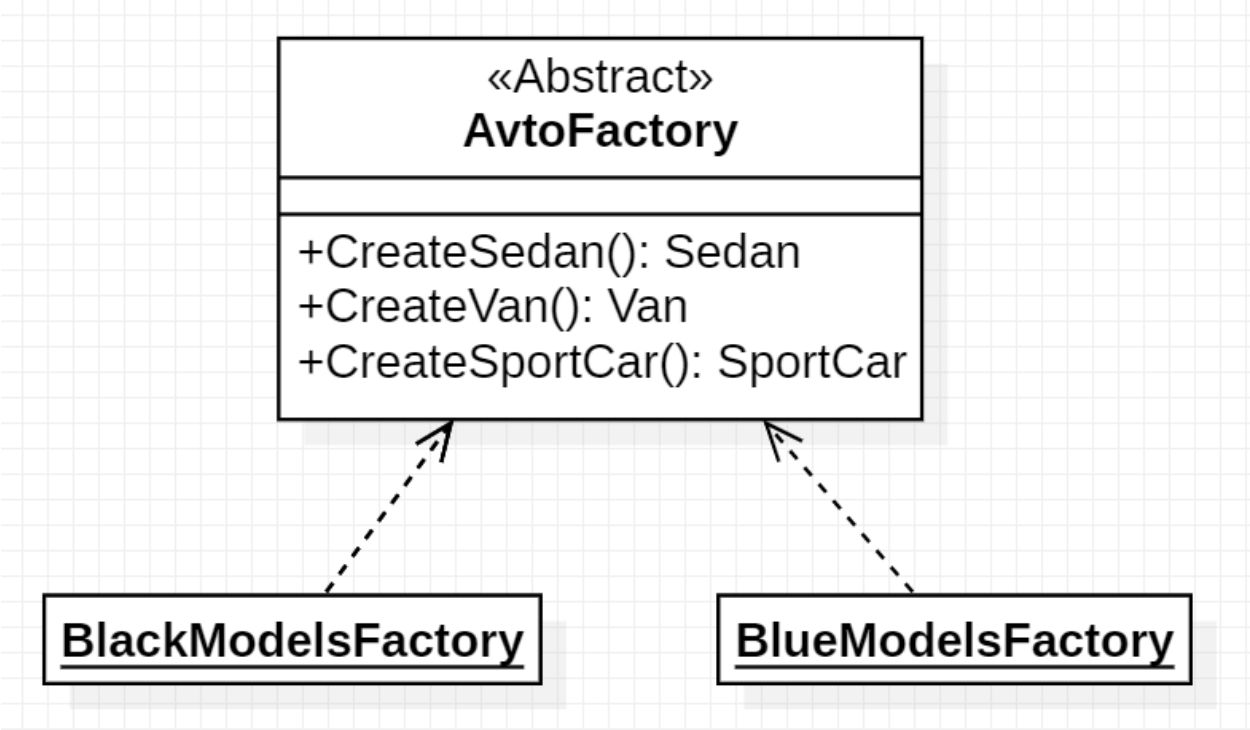

Рисунок 1. Упрощённая схематическое представление абстрактной фабрики

Главной частью схемы является абстрактная фабрика с тремя абстрактными методами: CreateSedan(): Sedan, CreateVan(): Van, CreateSportCar(0: SportCar. Данные методы создают соответственно автомобили классифицирующиеся по кузову – седан, фургон и купе. От вышеописанной фабрики наследуются ещё две фабрики, которые определяют (производят конкретные модели модели) цвет автомобилей – чёрные и синие. Семействами объектов здесь являются автомобили конкретного цвета.

Для полного понимания составляется UML-схема, в которой присутствуют две роли – интерфейс и объекты (рис. 2). В качестве темы примера сохранена была автомобильная тематика, в данном случае «Автомобильный завод».

На схеме изображен абстрактный класс интерфейса (AbstractFactory), который создаёт объекты (CreateModelA(), CreateModelB() и CreateModelC()). Также размещены абстрактные классы данных объектов – ModelA, ModelB и ModelC. У данных классов присутствуют наследники – конкретные объекты: RealModelA1, RealModelB1, RealModelC1, RealModelA2, RealModelB2 и RealModelC2. То есть данные объекты представляют из себя уже уникальные модели автомобилей разных производителей.

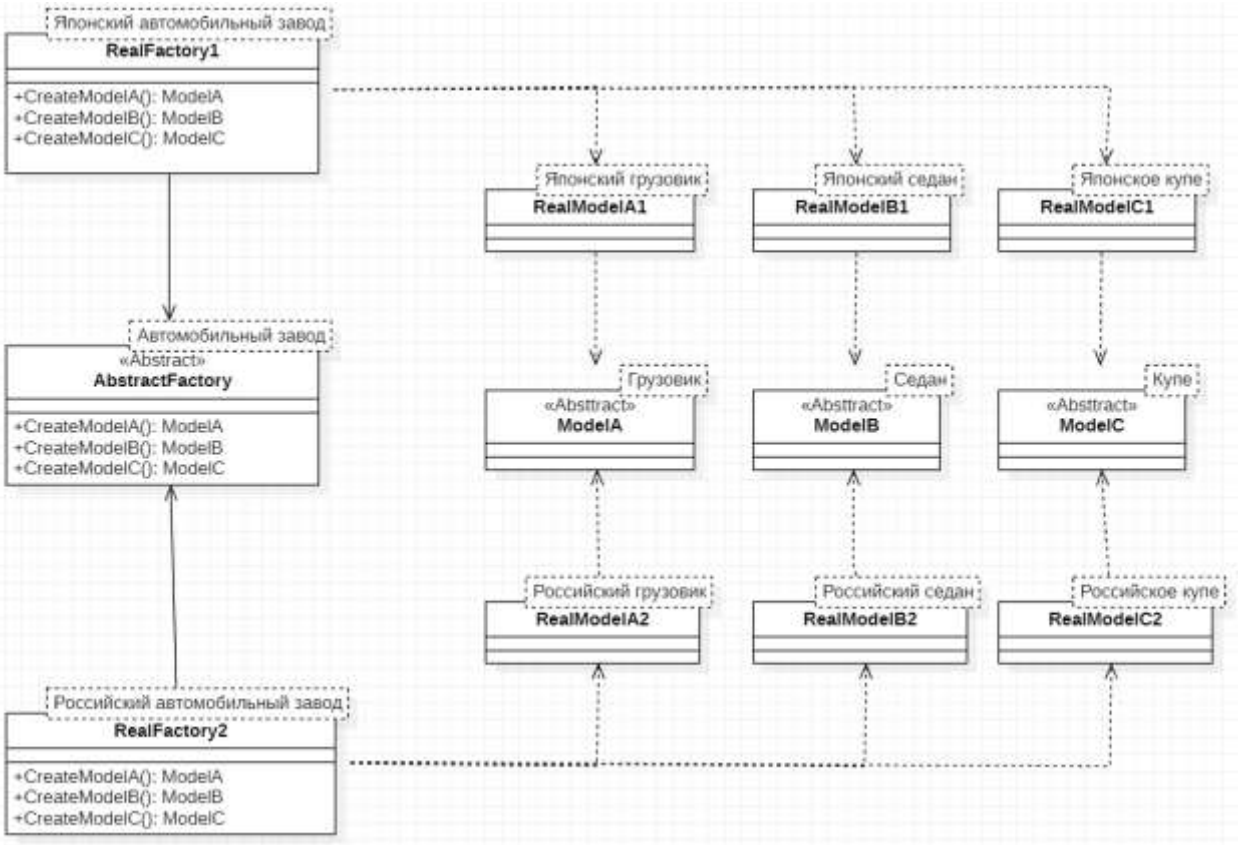

Рисунок 2. UML-схема паттерна Abstract Factory

Вышеописанные объекты создаются с помощью определённых интерфейсов, порождаемых абстрактной фабрикой (RealFactory1 и RealFactory2).

Схематически паттерн представлен. Далее необходимо обозначить положительные стороны абстрактной фабрики.

Среди положительных сторон паттерна является следование принципам ООП таким, как SRP (принцип единственной ответственности) и DIP (принцип инверсии зависимостей). Благодаря SRP код выносится в отдельный модуль из-за чего исключается дублирование кода, так как код создания семейства находится в одном месте. Принцип DIP обеспечивает отсутствие зависимостей от конкретных реализаций, что в свою очередь делает код гибким (устойчивым к изменениям).

После описания паттерна следует его реализация на практическом примере. В начале статьи упоминались пользовательские интерфейсы (UI), которые базируются на паттерне абстрактная фабрика.

Исходя из UML-схемы (рис. 2) и её описание можно заметить, почему именно в UI чаще используется паттерн.

В современных приложениях начиная от профессиональных IDE и заканчивая обычными блокнотами можно заметить, что интерфейс данных программ практически всегда меняется. Из этих изменений можно выделить: изменение цветовой палитры, изменение дизайна кнопок, окон и фона. В UI видеоигр ярко выражены постоянные изменения и реакция интерфейса на действия игрока и игрового мира – изменения анимаций, появление различных подсказок, сообщений и тому подобное. Помимо уже готовых и работающих программа, паттерн абстрактной фабрики даёт преимущество программистам на этапе проектирования и разработки ПО, когда происходят постоянные производственные изменения интерфейса с сохранением предыдущих версий и быстрая возможность на них переключаться.

В практической части статьи поставлена задача на реализацию создание персонажей на графическом интерфейсе.

Для реализации задачи используется подготовленный проект, созданный в Unity3D. Для полной реализации паттерны были созданы три группы персонажей, отличающихся по цвету – красные, синие и зелёные. Каждая группа имеет три разных типа персонажей – маг, лучник и рыцарь.

На графическом интерфейсе размещены кнопки переключения цвета групп, кнопки создания определённых юнитов и кнопка удаления персонажа с экрана.

Начала реализации паттерна начинается с персонажей. Для большинства сущностей код будет схож, так что демонстрация реализации паттерна будет показывать на программах одной группы персонажей, а именно на красной.

В начале создаются абстрактные классы: Knight, Mage, Archer (рис. 3).

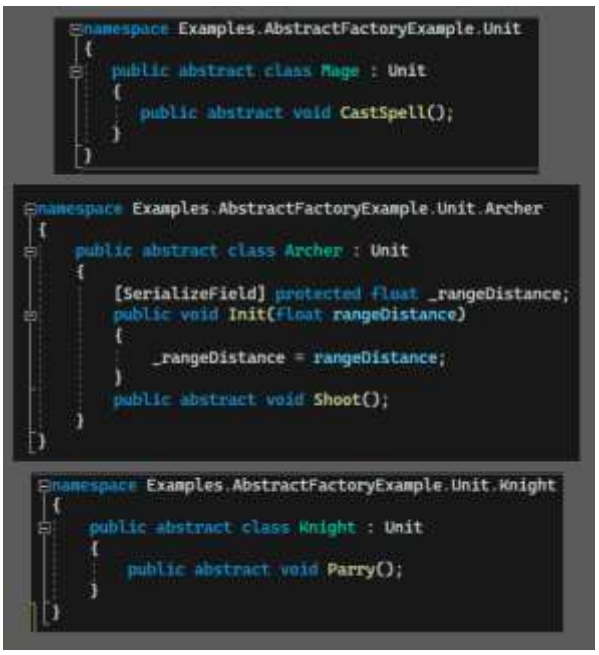

Рисунок 3. Абстрактные классы Mage, Archer, Knight

У всех трёх персонажей объявлены свои собственные абстрактные методы для заклинания, стрельбы и парирования. Все описанные классы наследуют класс Unit, который в себе содержит общие параметры здоровья и урона (рис. 4).

```
using UnityEngine;

namespace Examples.AbstractFactoryExample.Unit
{
    public class Unit : MonoBehaviour
    {
        [SerializeField] protected int _health;
        [SerializeField] protected float _attackValue;
    }
}
```

Рисунок 4. Класс Unit

Затем создаются конкретные классы RedKnight, RedMage, RedArcher от абстрактных классов (рис. 5).

```
namespace Examples.AbstractFactoryExample.Unit.Knight
{
    public class RedKnight : Knight
    {
        [SerializeField] private float _redKnightKoef;
        public void Init(float redKnightKoef)
        {
            _redKnightKoef = redKnightKoef;
        }
        public override void Parry()
        {

        }
    }
}
```

```
namespace Examples.AbstractFactoryExample.Unit
{
    public class RedMage : Mage
    {
        [SerializeField] private float _fireBallRadius;
        [SerializeField] private float _fireBallDamage;
        public void Init(float fireBallRadius, float fireBallDamage)
        {
            _fireBallRadius = fireBallRadius;
            _fireBallDamage = fireBallDamage;
        }
        public override void CastSpell()
        {

        }
    }
}
```

```
namespace Examples.AbstractFactoryExample.Unit.Archer
{
    public class RedArcher : Archer
    {
        [SerializeField] private float _fireArrowDamage;
        public void Init(float fireArrowDamage, float rangeDistance)
        {
            _fireArrowDamage = fireArrowDamage;
            base.Init(rangeDistance);
        }
        public override void Shoot()
        {

        }
    }
}
```

Рисунок 5. Классы RedKnight, RedMage, RedArcher

Данные классы содержат свои уникальные параметры персонажей – силу, зону и шанс срабатывания способностей.

После классов персонажей нужно реализовать создание этих персонажей в UI. Для этого будет использовать абстрактная фабрика персонажей. Но сначала приведётся пример создание персонажей без помощи паттерна, а через конструкцию switch case (рис. 6).

```
_knightButton.onClick.AddListener(() =>
{
    Knight knight;

    switch (_currentType)
    {
        case UnitType.Red:
            var redPrefab = Resources.Load<GameObject>("Prefabs/knight_red");
            var redGo = Instantiate(redPrefab);
            var redKnight = redGo.GetComponent<RedKnight>();
            redKnight.Init(redKnightKoef: 3.3f);
            knight = redKnight;
            break;

        case UnitType.Blue:
            var bluePrefab = Resources.Load<GameObject>("Prefabs/knight_blue");
            var blueGo = Instantiate(bluePrefab);
            var blueKnight = blueGo.GetComponent<BlueKnight>();
            blueKnight.Init(4.0f, 2.0f);
            knight = blueKnight;
            break;

        case UnitType.Green:
            var greenPrefab = Resources.Load<GameObject>("Prefabs/knight_green");
            var greenGo = Instantiate(greenPrefab);
            var greenKnight = greenGo.GetComponent<GreenKnight>();
            knight = greenKnight;
            break;

        default:
            knight = new RedKnight();
            Debug.LogError("Something wrong");
            break;
    }

    knight.transform.SetParent(_unitGrid.transform);
});
```

Рисунок 6. Создание персонажей с помощью конструктора switch case

Данный пример демонстрирует не оптимальный код, так как он является громоздким, прописывается в одном методе onClick, из-за чего нарушается принцип ООП SRP, а именно возникает высокий риск дублирования кода. Например, в процессе разработки может возникнуть потребность создавать персонажа не по кнопке, а по совершению какого либо события, то в новый скрипт придётся переносить всю конструкцию switch case. Также проблемы возникают в случае того, если количество групп персонажей увеличивается, что влекёт за собой ощутимый рост и громоздкость кода.

Чтобы избежать проблем и оптимизировать свой код следует использовать абстрактную фабрику персонажей. Для этого создаётся абстрактный класс UnitsFactory (рис. 7).

```
namespace Examples.AbstractFactoryExample
{
    public abstract class UnitsFactory
    {
        public abstract Knight CreateKnight();
        public abstract Mage CreateMage();
        public abstract Archer CreateArcher();
    }
}
```

Рисунок 7. Класс UnitsFactory

В данном классе присутствуют три метода создание каждого типа персонажей. Для красных персонажей создаётся RedUnitsFactory, который содержит методы на создание персонажа из prefab (рис. 8). Данный класс наследует предшествующий UnitsFactory.

```
namespace Examples.AbstractFactoryExample
{
    public class RedUnitsFactory : UnitsFactory
    {
        public override Knight CreateKnight()
        {
            var prefab = Resources.Load<GameObject>("Prefabs/knight_red");
            var go = GameObject.Instantiate(prefab);
            var knight = go.GetComponent<RedKnight>();
            knight.Init(3.3f);
            return knight;
        }
        public override Mage CreateMage()
        {
            var prefab = Resources.Load<GameObject>("Prefabs/mage_red");
            var go = GameObject.Instantiate(prefab);
            var mage = go.GetComponent<RedMage>();
            mage.Init(1.0f, 5.0f);
            return mage;
        }
        public override Archer CreateArcher()
        {
            var prefab = Resources.Load<GameObject>("Prefabs/archer_red");
            var go = GameObject.Instantiate(prefab);
            var archer = go.GetComponent<RedArcher>();
            archer.Init(1.0f, 1.0f);
            return archer;
        }
    }
}
```

Рисунок 8. Класс создания красных персонажей

Таким образом получается непосредственно работа с абстрактными классами, которые имеют взаимосвязь между собой и наследование, что придаёт всему коду программы гибкость, лёгкую расширяемость, избавляет от дублирование кода, а также убирает зависимость от конкретных реализаций.

В скрипте UI подключается класс UnitsFactory, с помощью которого при нажатии на кнопку вызываем соответствующие методы на создание персонажей (рис. 9).

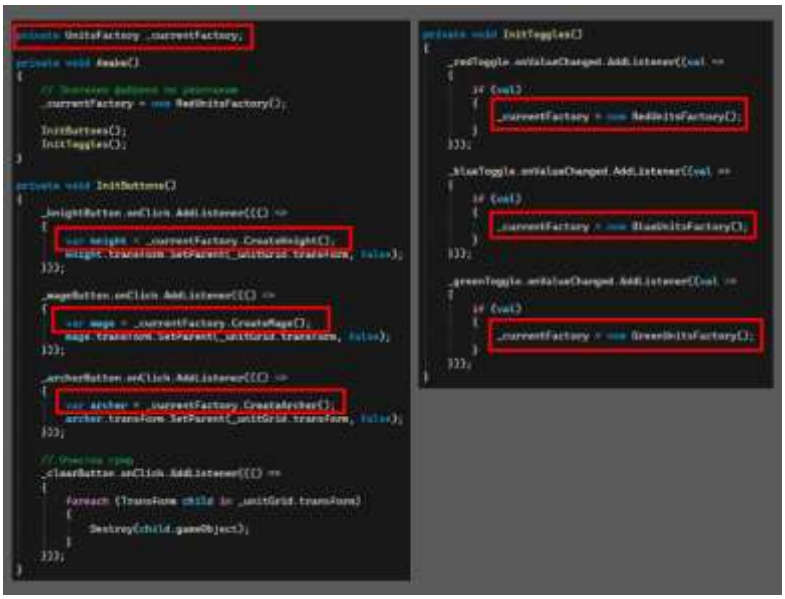

Рисунок 9. Методы Awake и InitToggles

За счёт того, что нет зависимости между какой-то конкретной реализации, то нет данных о том, что хранит в себе переменная currentFactory, содержащая UnitsFactory. Для возможности выбора используется переключатели toggle, которые задают значение переменной _currentFactory.

В конце запускается проект и проверяется корректность реализации паттерна абстрактной фабрики (рис. 10).

Рисунок 10. Результат реализации паттерна Abstract Factory

Таким образом в ходе написания статьи был описан и реализован популярный паттерн поведения Abstract Factory. Данный паттерн является оптимальным подходом в создании программ, где присутствуют семейства объектов, что было показано на примере создания механики появления персонажей на графическом интерфейсе.

Библиографический список

1. Пасынкова А.А., Викентьева О.Л. Проектирование архитектуры системы мониторинга на основе паттернов проектирования // Proceedings of the Institute for System Programming of the RAS. 2023. Т. 35. № 3. С. 137-150.
2. Шишкин В.В., Кандаулов В.М. Система автоматизированного проектирования сложных машиностроительных изделий на базе паттернов проектирования Автоматизация процессов управления. // 2011. № 3. С. 56-62.
3. Талыбов Н.Г. Система автоматизированного проектирования на базе паттернов проектирования // В сборнике: Информационные системы и технологии: достижения и перспективы. II международная научная конференция. Сумгаит, 2020. С. 232-233.
4. Валеев С.С., Кондратьева Н.В. Паттерны проектирования архитектуры нулевого доверия // Инженерный вестник Дона. 2023. № 9 (105). С. 76-83.
5. Тоичкин Н.А. Систематизация процесса проектирования информационных систем с использованием паттернов на примере ИС "анализа и моделирования" // В сборнике: Теория и практика системной динамики. Апатиты, 2007. С. 83-85.