

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Инженерная школа информационных технологий и робототехники

Направление подготовки 09.03.04 Программная инженерия

Отделение информационных технологий

**КУРСОВОЙ ПРОЕКТ**

**по дисциплине «Проектирование и архитектура программных систем»**

|  |
| --- |
| **Тема проекта** |
| **Информационная система «Книжный магазин»** |

Студент

|  |  |  |  |
| --- | --- | --- | --- |
| Группа | ФИО | Подпись | Дата |
| 8К22 | Шаринский А.С. |  | 19.11.2024 |

Преподаватель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Должность | ФИО | Ученая степень, звание | Подпись | Дата |
| доцент ОИТ ИШИТР | Поляков А.Н. | к.т.н., доцент |  |  |

Томск – 2024 г.

Оглавление

[Оглавление 2](#_Toc182925781)

[Введение 4](#_Toc182925782)

[1. UML 5](#_Toc182925783)

[1.1. Функциональные и нефункциональные требования к системе 5](#_Toc182925784)

[1.2. Описание требований к системе: варианты использования 6](#_Toc182925785)

[1.3. Выявление классов. Построение и описание диаграммы классов анализа 12](#_Toc182925786)

[1.4. Построение и описание диаграмм состояний 16](#_Toc182925787)

[1.5. Построение и описание диаграммы проектных классов 19](#_Toc182925788)

[1.6. Построение и описание диаграмм последовательности для операций проектных классов 23](#_Toc182925789)

[1.7. Построение и описание диаграммы пакетов 25](#_Toc182925790)

[1.8. Модульное и интеграционное тестирование 29](#_Toc182925791)

[2. Паттерны 35](#_Toc182925792)

[2.1. Паттерны ООП: одиночка 35](#_Toc182925793)

[2.2. Паттерны ООП: медиатор 36](#_Toc182925794)

[2.3. Паттерны ООП: фабричный метод 38](#_Toc182925795)

[2.4. Паттерны ООП: абстрактная фабрика 40](#_Toc182925796)

[2.5. Паттерны ООП: прокси 43](#_Toc182925797)

[2.6. Паттерны ООП: состояние 44](#_Toc182925798)

[2.7. Паттерны ООП: стратегия 46](#_Toc182925799)

[2.8. Паттерны ООП: легковес 48](#_Toc182925800)

[Заключение 50](#_Toc182925801)

[Список литературы 51](#_Toc182925802)

[Приложение А 52](#_Toc182925803)

[Приложение Б 53](#_Toc182925804)

[Приложение В 65](#_Toc182925805)

[Приложение Г 68](#_Toc182925806)

[Приложение Д 81](#_Toc182925807)

[Приложение Е 86](#_Toc182925808)

[Приложение Ж 92](#_Toc182925809)

[Приложение К 98](#_Toc182925810)

[Приложение Л 103](#_Toc182925811)

[Приложение М 107](#_Toc182925812)

Введение

Информационная система в курсовой работе проектируется для книжного магазина.

Система предназначена для обслуживания клиентов книжного магазина и управления его товарами и заказами. В разрабатываемой системе предусмотрены функции регистрации и авторизации пользователей, поиск и сортировка товаров каталога по их характеристикам, формирование корзины клиентов, оформление заказов и изменения их состояния, управление списком избранных товаров и списком заказов пользователей.

Проектируемая система может быть использована не только клиентами книжного магазина, но и администраторами и операторами поддержки. Поэтому функции системы также включают добавления и удаления товаров из каталога магазина, управление учетными записями пользователей, просмотр списков заказов пользователей и удаление аккаунтов и определённых заказов клиентов.

При проектировании информационной системы были поставлены такие цели, как автоматизация процессов управления каталогом книжного магазина и списками заказов пользователей, повышение эффективности взаимодействия клиентов и сотрудников книжного магазина.

В ходе работы будут разработаны необходимые диаграммы UML, созданы тесты для выявленных классов системы и их методов, реализованы несколько паттернов проектирования.

1. UML
   1. Функциональные и нефункциональные требования к системе

Требования к программным продуктам или информационным системам можно разделить на две большие группы: функциональные и нефункциональные.

Функциональные требования описывают, что необходимо реализовать в продукте или системе, в том числе какие действия должны выполнять пользователи при взаимодействии с ними.

Функциональные требования включают в себя описание процессов, данных и взаимодействий между различными компонентами системы. Эти требования являются основой для разработки функциональности системы и часто описываются в виде пользовательских сценариев или историй.

Нефункциональные требования описывают, как должна работать система или программный продукт, и какими свойствами или характеристиками она должна обладать. Они связаны не с функциями, выполняемыми системой, а с такими интеграционными свойствами системы, как надёжность, время ответа или размер системы.

Нефункциональные требования основываются на бюджетных ограничениях, учитывают организационные возможности компании-разработчика, возможность взаимодействия разрабатываемой системы с другими программными и вычислительными системами, а также такие внешние факторы, как правила техники безопасности, законодательство о защите интеллектуальной собственности и т. п.

Функциональные требования:

1. Система должна отображать информацию о характеристиках товаров.
2. Система должна позволять клиенту фильтровать список товаров по их характеристикам: названию, автору, жанру, году издания и цене.
3. Система должна позволять пользователю авторизоваться в системе по логину и паролю.
4. Система должна позволять клиенту создать личный кабинет.
5. Система должна позволять клиенту добавлять товар в список избранных и удалять товар из него.
6. Система должна позволять клиенту добавить товар в корзину, изменить его количество в корзине и удалить товар из нее.
7. Система должна позволять клиенту оформить заказ с выбором способа получения: самовывоз или доставка курьером.
8. Система должна отображать историю заказов клиента.
9. Система должна отображать отзывы клиентов о товаре и позволять клиенту оставлять отзывы.
10. Система должна отображать актуальную информацию о наличии товаров на складе.

Нефункциональные требования:

1. Система должна отвечать на запрос клиента в течение не более 3 секунд.
2. Система должна быть легко расширяема для добавления новых функций и разделов.
3. Система должна обладать понятным и удобным пользовательским интерфейсом.
4. Система должна обеспечивать защиту данных пользователей.
5. Система должна быть написана на C#.
   1. Описание требований к системе: варианты использования

UML (англ. «Unified Modeling Language») – стандартизированный язык моделирования при проектировании программ. Его можно использовать для визуализации, спецификации, конструирования и документирования программных систем.

Словарь UML включает три вида строительных блоков:

* диаграммы;
* сущности;
* связи.

Диаграмма – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей).

Сущности – это абстракции, которые являются основными элементами модели, связи соединяют их между собой, а диаграммы группируют представляющие интерес наборы сущностей.

Диаграмма вариантов использования – диаграмма, описывающая, какой функционал разрабатываемой программной системы доступен каждой группе пользователей (иногда ее ещё называют диаграммой прецедентов).

Актёр обозначает любые сущности, использующие систему. Этими сущностями могут быть люди, технические устройства, время или даже другие системы. Вариант использования – набор действий, который может быть использован актёром для взаимодействия с системой. На диаграммах UML для связывания элементов используются различные соединительные линии, которые называются отношениями. Каждое такое отношение имеет собственное название и используется для достижения определённой цели.

Виды отношений:

* отношение ассоциации;
* отношение обобщения;
* отношение включения;
* отношение расширения.

Спецификация это – текстовое описание конкретных последовательностей действий (потока событий), которые выполняет пользователь при работе с системой. Спецификация должна включать:

* краткое описание варианта использования;
* имена главных и второстепенных актёров;
* предусловия;
* основной поток;
* постусловия;
* альтернативные потоки (при наличии).

Краткое описание – один абзац, в котором изложена цель прецедента (варианта использования).

С точки зрения отдельного прецедента существует два типа актёров:

* главные актёры – актёры, инициирующие прецедент;
* второстепенные актёры – актёры, взаимодействующие с прецедентом после его инициации.

Каждый прецедент всегда инициируется одним актёром. Однако один и тот же прецедент в разные моменты времени может инициироваться разными актёрами. Любой актёр, который может инициировать прецедент, является главным актёром. Все остальные актёры – второстепенные.

Предусловия и постусловия – это ограничения. Предусловия ограничивают состояние системы, необходимое для запуска прецедента. Постусловия ограничивают состояние системы после выполнения прецедента.

Основной поток описывает «идеальный» ход развития событий в прецеденте. У каждого прецедента есть один основной поток и может быть множество альтернативных потоков.

Диаграмма вариантов использования представлена на рис. 1.

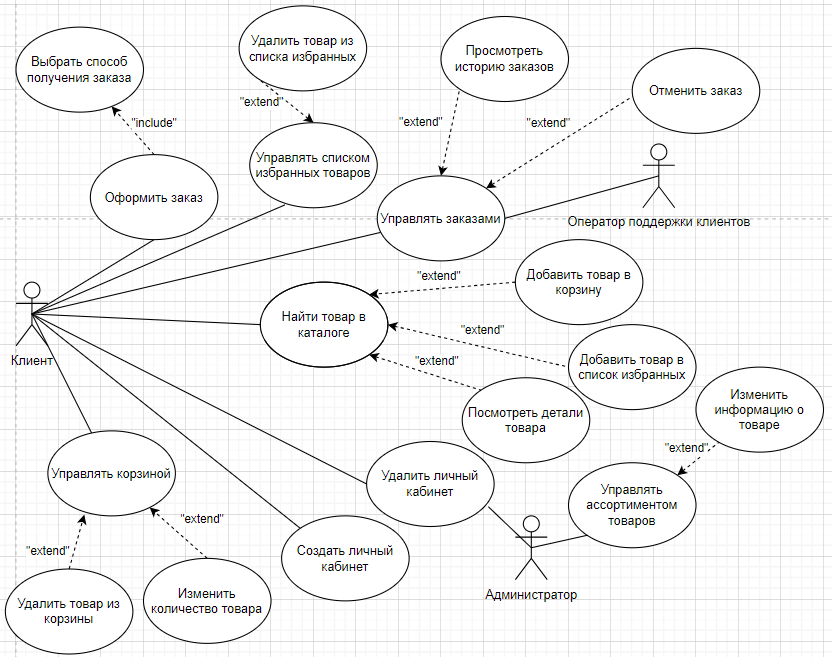


Рис. – Диаграмма вариантов использования

**Прецедент**: FindProduct

**ID**: 1

**Краткое описание**: Система ищет товары, которые удовлетворяют критериям поиска Клиента, выводит их на экран для Клиента и предоставляет Клиенту возможность добавления товара в список избранного или в корзину.

Главные актёры: Клиент.

Второстепенные актёры: нет.

Предусловия: Нет.

Основной поток:

1. Прецедент начинается, когда Клиент выбирает «найти товар в каталоге».
2. Клиент вводит в систему критерии поиска товара.
3. Система ищет товары, удовлетворяющие критериям Клиента.
4. Если система находит соответствующие товары.
   1. Для каждого найденного товара
      1. Система выводит на экран название товара.
      2. Система выводит на экран изображение товара.
      3. Система выводит на экран описание товара.
      4. Система выводит на экран цену товара.
      5. Если Клиент выбирает «добавить товар в корзину».
         1. Система добавляет один товар в корзину Клиента.
         2. Система выводит на экран уведомление о добавления товара в корзину.
      6. Если Клиент выбирает «добавить товар в избранное».
         1. Система добавляет товар в список избранного Клиента.
         2. Система выводит на экран уведомление о добавления товара в список избранного.
5. Иначе (Else)
   1. Система сообщает Клиенту о том, что нет товаров, соответствующих критериям поиска.

Постусловия: нет.

Альтернативные потоки: нет.

**Прецедент**: PlaceOrder

**ID**: 2

**Краткое описание**: Клиент оформляет заказ в системе.

Главные актёры: Клиент.

Второстепенные актёры: нет.

Предусловия:

1. Содержимое корзины для покупок является видимым.

Основной поток:

1. Прецедент начинается, когда Клиент выбирает «оформить заказ».
2. Если Клиент выбирает способ получения заказа «самовывоз».
   1. Клиент выбирает адрес получения из предложенного списка.
   2. Клиент производит онлайн-оплату заказа.
3. Если Клиент выбирает способ получения заказа «курьером».
   1. Клиент вводит адрес получения заказа.
   2. Клиент производит онлайн-оплату заказа.

Постусловия: нет.

Альтернативные потоки: нет.

**Прецедент**: OrderHistory

**ID**: 3

**Краткое описание**: Клиент или Оператор поддержки клиентов просматривает историю заказов Клиента и отменяет заказ.

**Главные актёры**: Клиент, Оператор поддержки клиентов.

Второстепенные актёры: нет.

Предусловия: нет.

Основной поток:

1. Прецедент начинается, когда Клиент или Оператор поддержки клиентов выбирает «открыть историю заказов».
2. Система выводит на экран список заказы Клиента в том порядке, в котором они были оформлены.
3. Если Клиент или Оператор поддержки клиентов выбирает «отменить заказ».
   1. Система возвращает средства Клиента на карту, с которой заказ был оплачен.
   2. Система отменяет заказ Клиента.
   3. Система сообщает Клиенту об успешной операции отмены заказа.

Постусловия: нет.

Альтернативные потоки: нет.

* 1. Выявление классов. Построение и описание диаграммы классов анализа

Диаграммы классов используются при моделировании программных систем наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования. Диаграмма классов не отображает динамическое поведение объектов, изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

* концептуальный уровень, на котором диаграммы классов отображают связи между основными понятиями предметной области;
* уровень спецификаций – это уже уровень, на котором диаграммы классов отображают связи объектов этих классов;
* уровень реализации – это уровень, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Концептуальная диаграмма классов используется для понимания и анализа проблемной области. На концептуальной диаграмме классов каждый класс представляет собой концепцию, обычно связанную с бизнес-областью, реальным миром.

Диаграммы классов оперируют тремя видами сущностей UML:

* структурные;
* поведенческие;
* аннотирующие.

Структурные сущности – это «имена существительные» в модели UML. В основном, статические части модели, представляющие либо концептуальные, либо физические элементы. Основным видом структурной сущности в диаграммах классов является класс.

Поведенческие сущности – динамические части моделей UML. Это «глаголы» моделей, представляющие поведение модели во времени и пространстве. Основной из них является взаимодействие – поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определенном контексте для достижения некоторой цели.

Аннотирующие сущности – это поясняющие части UML-моделей, иными словами, комментарии, которые можно применить для описания, выделения и пояснения любого элемента модели.

Класс – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.

Атрибут (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса.

Операция (метод) – это реализация метода класса. Операцию можно описать более подробно, указав имена и типы параметров, их значения, принятые по умолчанию, а также тип возвращаемого значения.

Вершины диаграмм классов содержат классы, а дуги (ребра) – отношениями между ними. Существует шесть основных типов отношений между классами:

* Ассоциация (association). Показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.
* Наследование (inheritance). Специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя.
* Реализация (realization/implementation). Реализация определяет отношение, при котором класс-приемник обеспечивает свою собственную реализацию интерфейса другого класса-источника. Это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться.
* Зависимость (dependency). Изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого).
* Агрегация (aggregation). Разновидность ассоциации, представляющая структурную связь целого с его частями.
* Композиция (composition). Форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Множественность (многозначность) ассоциации представляет собой диапазон целых чисел, указывающий возможное количество связанных объектов. Он записывается в виде выражения с минимальным и максимальным значением; для их разделения используются две точки.

Концептуальная диаграмма классов представлена на рис. 2.

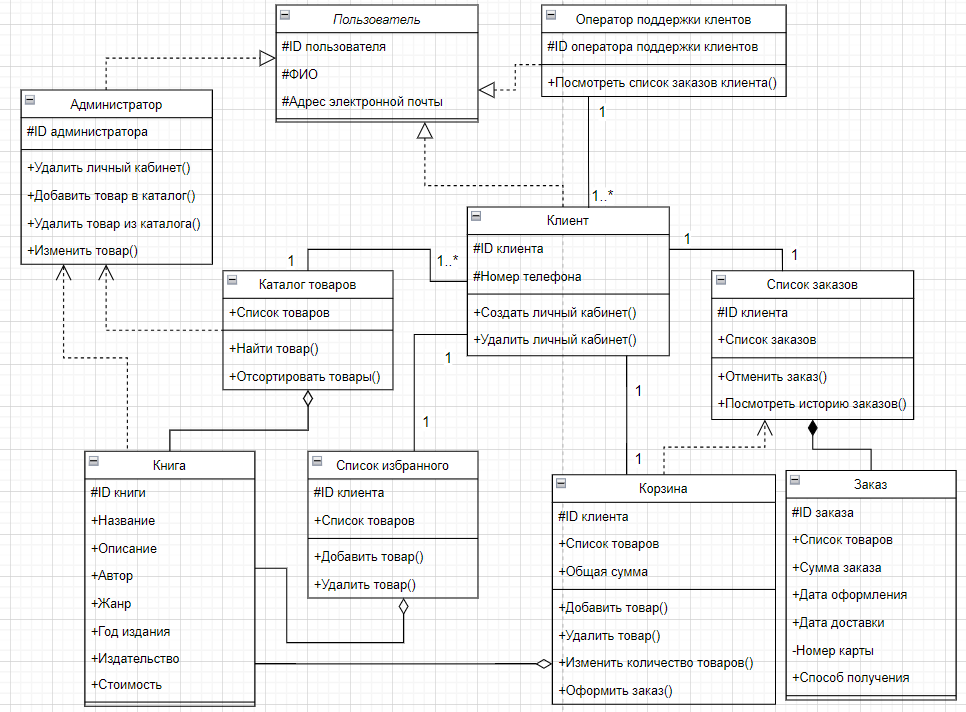


Рис. – Концептуальная диаграмма классов

1. Связи «Администратор – Пользователь», «Клиент – Пользователь», «Оператор поддержки клиентов – Пользователь» – реализация, так как каждый из дочерних классов имеет атрибуты ID пользователя, ФИО и адрес электронной почты и является пользователем системы («Пользователь» – абстрактный класс), но отдельно пользователем, не являющимся ни клиентом, ни оператором поддержки, ни администратором, быть никто не может.
2. Связи «Администратор – Каталог товаров», «Администратор – Книга» – зависимость, так как изменение, вносимое администратором методами «Изменить товар» или «Добавить товар в каталог», может привести к изменениям в атрибутах книги и каталога товаров.
3. Связь «Клиент – Каталог товаров» – двусторонняя ассоциация с кратностью один или больше, так как несколько клиентов могут посмотреть каталог товаров и найти нужную книгу, при этом каталог является единственным в магазине.
4. Связь «Клиент – Оператор поддержки клиентов» – двусторонняя ассоциация с кратностью один или больше, так как одним оператором поддержки могут обслуживаться неограниченное количество клиентов.
5. Связи «Клиент – Список избранного», «Клиент – Корзина», «Клиент – Список заказов» – двусторонняя ассоциация с кратностью единственный, так как у каждого клиента может быть только один список избранного, одна корзина и один список заказов, и наоборот.
6. Связи «Каталог товаров – Книга», «Список избранного – Книга», «Корзина – Книга» – агрегация, так как книга является частью каталога товаров, списка избранного и корзины, а уничтожение контейнера (каталога товаров, списка избранного или корзины) не приводит к уничтожению элементов (книг).
7. Связь «Список заказов – Заказ» – композиция, так как заказ является частью списка заказов, а уничтожение контейнера (списка заказов) приводит к уничтожению элементов (заказов).
8. Связь «Корзина – Список заказов» – зависимость, так как изменение, вносимое корзиной методом «Оформить заказ, приводит к изменениям в списке заказов (появление нового заказа).
   1. Построение и описание диаграмм состояний

Диаграмма состояний (Statechart Diagram) – методология объектно-ориентированного проектирования, предназначенная для представления жизненного цикла объектов в реальном или абстрактном мире.

Событие (Event) – сообщение, которое возникает в различных точках исполняемого кода при выполнении определенных условий.

Конечный автомат моделирует динамическое поведение реактивного объекта.

Три основных элемента автоматов – состояния, события и переходы:

* состояние (state) – «условие или ситуация в жизни объекта, при которых он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторого события»;
* событие (event) – «описание заслуживающего внимания происшествия, занимающего определенное положение во времени и пространстве»;
* переход (transition) – переход из одного состояния в другое в ответ на событие.

Реактивный объект – это объект, в широком смысле этого слова, который предоставляет автомату контекст. Реактивные объекты:

* отвечают на внешние события (т. е. события, происходящие вне контекста объекта);
* могут генерировать и отвечать на внутренние события;
* их жизненный цикл смоделирован как последовательность состояний, переходов и событий;
* их текущее поведение может зависеть от предыдущего поведения.

Диаграмма состояний позволяет описать поведение отдельно взятого объекта при определенных условиях. Также она покажет нам все возможные состояния, в которых может находиться объект, а также процесс смены состояний в результате внешнего влияния.

Каждое состояние поведенческого автомата может содержать нуль или более действий и деятельностей. У состояний протокольных автоматов нет действий или деятельностей. Действия считаются мгновенными и непрерываемыми, тогда как деятельности занимают конечное время и могут быть прерваны.

Два специальных действия – вход и выход – ассоциированы со специальными событиями entry и exit. У этих двух событий особая семантика. Событие entry происходит мгновенно и автоматически при входе в состояние. Это первое, что происходит, когда осуществляется вход в состояние. Это событие обусловливает выполнение ассоциированного с ним действия на входе. Событие exit – самое последнее, что происходит мгновенно и автоматически при выходе из состояния. Обусловливает выполнение ассоциированного действия на выходе.

Переходы показывают движение между состояниями. Переходы могут быть соединены переходными псевдосостояниями (junction pseudo-states). Это точки слияния или ветвления переходов.

В качестве редактора UML диаграмм был выбран draw.io. Результат создания диаграммы состояния представлен на рис. 3. На рисунке представлен процесс оформления и доставки заказа.

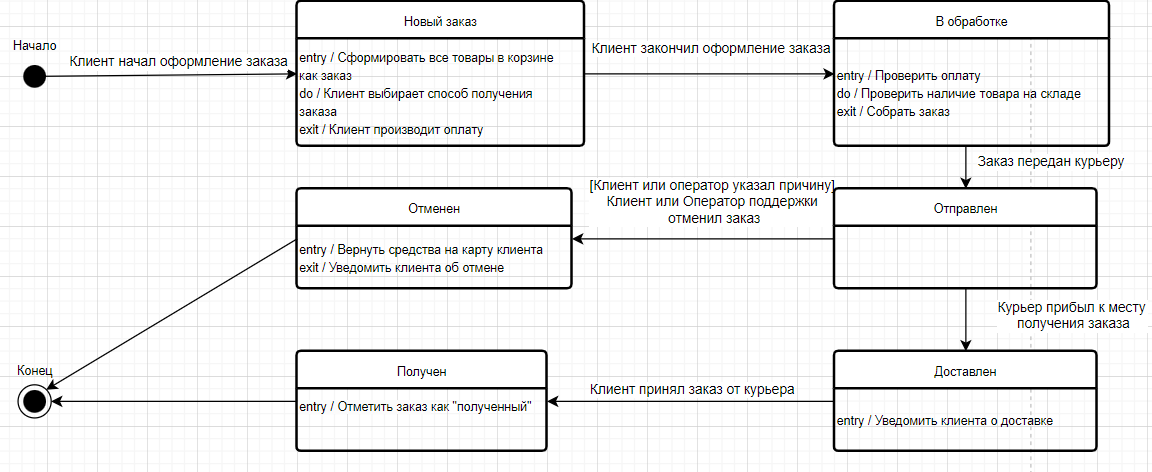


Рис. – Диаграмма состояний

1. Начало. Процесс перехода в состояние «Новый заказ»: клиент начал оформление заказа, выбрав опцию «Оформить заказ» в корзине.
2. Новый заказ. Специальное действие entry – все товары в корзине формируют заказ. Деятельность do – клиент выбирает способ получения заказа (курьером или самовывоз). Специальное действие exit – клиент производит оплату онлайн. Процесс перехода в состояние «В обработке»: Клиент закончил оформление заказа.
3. В обработке. Специальное действие entry – проверить, что оплата прошла. Деятельность do – проверить, что все товары заказа имеются в наличии. Специальное действие exit – заказ автоматически собирается из товаров, которые имеются на складе. Процесс перехода в состояние «Отправлен»: заказ передан курьеру.
4. Отправлен. Процесс перехода в состояние «Отменен»: предусловие – клиент или оператор поддержки клиентов указал причину отмены заказа, событие – клиент или оператор поддержки выбрал опцию «Отменить заказ» в списке заказов. Процесс перехода в состояние «Доставлен»: курьер прибыл к месту получения заказа.
5. Доставлен. Специальное действие entry – система уведомляет клиента о доставке. Процесс перехода в состояние «Получен»: заказ передан клиенту.
6. Отменен. Специальное действие entry – система возвращает клиенту средства за отмененный заказ. Специальное действие exit – система уведомляет клиента об успешной отмене заказа. Немедленный переход в конечное состояние.
7. Получен. Специальное действие entry – система отмечает заказ как «полученный». Немедленный переход в конечное состояние.
   1. Построение и описание диаграммы проектных классов

Диаграммы классов используются при моделировании программных систем наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования. Диаграмма классов не отображает динамическое поведение объектов, изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

* концептуальный уровень, на котором диаграммы классов отображают связи между основными понятиями предметной области;
* уровень спецификаций – это уже уровень, на котором диаграммы классов отображают связи объектов этих классов;
* уровень реализации – это уровень, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

В данной главе (построение диаграмм проектных классов), в отличие от диаграммы классов анализа необходимо будет показать все классы относящиеся к реализации системы, а не только к предметной области.

Диаграмма проектных классов программы представлена на рис. 4.

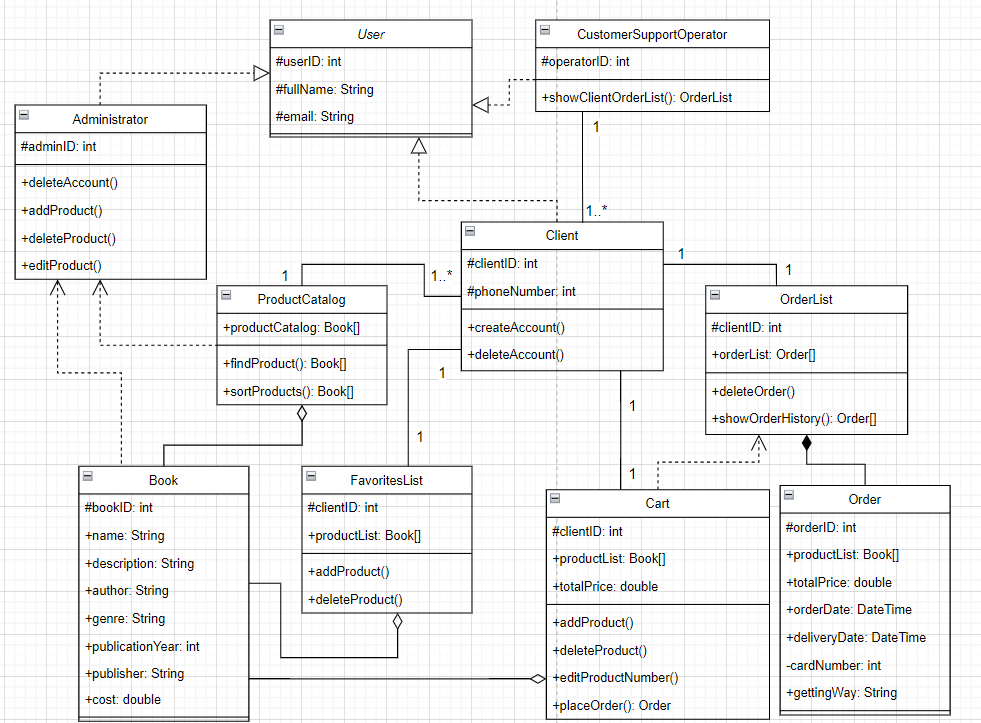


Рис. – Диаграмма проектных классов программы

Описание классов:

1. Класс «User» – абстрактный класс, имеющий следующие атрибуты с типом видимости «protected»: userID: int, fullName: String, email: String.
2. Класс «Administrator» – класс, реализующий абстрактный класс «User» и имеющий атрибут с типом видимости «protected» – adminID: int. В классе есть методы с типом видимости «public»: deleteAccount(), удаляющий личный кабинет клиента, addProduct(), добавляющий новый товар в каталог, deleteProduct(), удаляющий то-вар из каталога, editProduct(), изменяющий информацию о товаре.
3. Класс «Client» – класс, реализующий абстрактный класс «User» и имеющий атрибуты с типом видимости «protected» – clientID: int, phoneNumber: int. В классе есть методы с типом видимости «public»: createAccount(), создающий новый личный кабинет клиента, deleteAccount(), удаляющий личный кабинет клиента.
4. Класс «CustomerSupportOperator» – класс, реализующий абстрактный класс «User» и имеющий атрибут с типом видимости «protected» – operatorID: int. В классе есть метод с типом видимости «public»: showClientOrderList(), возвращающий объект класса «OrderList» для определенного клиента.
5. Класс «ProductCatalog» – класс, имеющий атрибут с типом видимости «public» – productCatalog: Book[]. В классе есть методы с типом видимости «public»: findProduct(), возвращающий коллекцию объектов класса Book, соответствующих какому-то заданному критерию, sortProduct(), возвращающий коллекцию объектов класса Book, от-сортированных по какой-то из характеристик.
6. Класс «Book» – класс, имеющий атрибут с типом видимости «protected» – bookID: int, и атрибуты с типом видимости «public»: name: String, description: String, author: String, genre: String, publicationYear: int, publisher: String, cost: double.
7. Класс «FavoritesList» – класс, имеющий атрибут с типом видимости «protected» – clientID: int, и атрибут с типом видимости «public»: productist: Book[]. В классе есть методы с типом видимости «public»: addProduct(), добавляющий объект типа Book в коллекцию productList, deleteProduct(), удаляющий объект типа Book из коллекции productList.
8. Класс «Cart» – класс, имеющий атрибут с типом видимости «protected» – clientID: int, и атрибуты с типом видимости «public»: productList: Book[], totalPrice: double. В классе есть методы с типом видимости «public»: addProduct(), добавляющий объект типа Book в коллекцию productList, deleteProduct(), удаляющий объект типа Book из коллекции productList, editProductNumber(), изменяющий количество товара в коллекции productList, placeOrder(), возвращающий объект класса Order и оформляющий новый заказ.
9. Класс «OrderList» – класс, имеющий атрибут с типом видимости «protected» – clientID: int, и атрибут с типом видимости «public»: orderList: Order[]. В классе есть методы с типом видимости «public»: deleteOrder(), удаляющий объект типа Order из коллекции orderList, showOrderHistory(), возвращающий коллекцию из объектов класса Order, отсортированную в хронологическом порядке.
10. Класс «Order» – класс, имеющий атрибут с типом видимости «pro-tected» – orderID: int, и атрибуты с типом видимости «public»: productList: Book[], totalPrice: double, orderDate: DateTime, delivery-Date: DateTime, cardNumber: int, gettingWay: String.

Описание связей и кратностей между классами:

1. Связи «Administrator – User», «Client – User», «CustomerSupportOperator – User» – реализация, так как каждый из дочерних классов имеет атрибуты ID пользователя, ФИО и адрес электронной почты и является пользователем системы («User» – абстрактный класс), но отдельно пользователем, не являющимся ни клиентом, ни оператором поддержки, ни администратором, быть никто не может.
2. Связи «Administrator – ProductCatalog», «Administrator – Book» – зависимость, так как изменение, вносимое администратором методами «editProduct» или «addProduct», может привести к изменениям в атрибутах книги и каталога товаров.
3. Связь «Client – ProductCatalog» – двусторонняя ассоциация с кратностью один или больше, так как несколько клиентов могут посмотреть каталог товаров и найти нужную книгу, при этом каталог является единственным в магазине.
4. Связь «Client – CustomerSupportOperator» – двусторонняя ассоциация с кратностью один или больше, так как одним оператором поддержки могут обслуживаться неограниченное количество клиентов.
5. Связи «Client – FavoritesList», «Client – Cart», «Client – OrderList» – двусторонняя ассоциация с кратностью единственный, так как у каждого клиента может быть только один список избранного, одна корзина и один список заказов, и наоборот.
6. Связи «ProductCatalog – Book», «FavoritesList – Book», «Cart – Book» – агрегация, так как книга является частью каталога товаров, списка избранного и корзины, а уничтожение контейнера (каталога товаров, списка избранного или корзины) не приводит к уничтожению элементов (книг).
7. Связь «OrderList – Order» – композиция, так как заказ является частью списка заказов, а уничтожение контейнера (списка заказов) приводит к уничтожению элементов (заказов).
8. Связь «Cart – OrderList» – зависимость, так как изменение, вносимое корзиной методом «placeOrder», приводит к изменениям в списке заказов (появление нового заказа).
   1. Построение и описание диаграмм последовательности для операций проектных классов

Актёр (Actor). Роль объекта вне системы, который прямо взаимодействует с ее частью – конкретным элементом.

Прецедент или вариант использования (Use Case) – описание поведения системы, когда она взаимодействует с кем-то (или чем-то) из внешней среды.

Диаграмма последовательности (Sequence Diagram) – диаграмма, на которой для некоторого набора объектов на единой временной оси показан жизненный цикл какого-либо определенного объекта и взаимодействие актеров ИС в рамках какого-либо определённого прецедента.

Диаграмма последовательности позволяет изобразить поведение нескольких объектов в рамках одного прецедента. Диаграмма последовательности удобна для представления взаимодействия объектов, но не для точного определения их поведения. Диаграмма показывает экземпляры объектов и сообщения, которыми обмениваются экземпляры в рамках одного прецедента.

Основные шаги построения диаграммы последовательности:

* добавление основных элементов;
* работа с сообщениями.

Объекты обычно подписываются в формате «объект:класс» и изображаются как в виде обычных прямоугольников, так и с использованием дополнительных обозначений.

Линия жизни (англ. lifeline) идет вертикально вниз от каждого объекта и упорядочивает сообщения на странице таким образом, чтобы они читались сверху вниз. Каждая линия жизни имеет полосу активности (зеленые вертикальные прямоугольники), показывающую интервал активности участника при взаимодействии.

Условия, как и циклы, изображаются с помощью фреймов взаимодействий (англ. interaction frames), позволяющих разметить диаграмму взаимодействия. Каждый фрейм представляет собой разделенную на несколько фрагментов область диаграммы, причем каждый фрейм имеет оператор, а каждый фрагмент может иметь защиту.

Диаграмма последовательности «Оформление заказа клиентом» представлена на рис. 5.

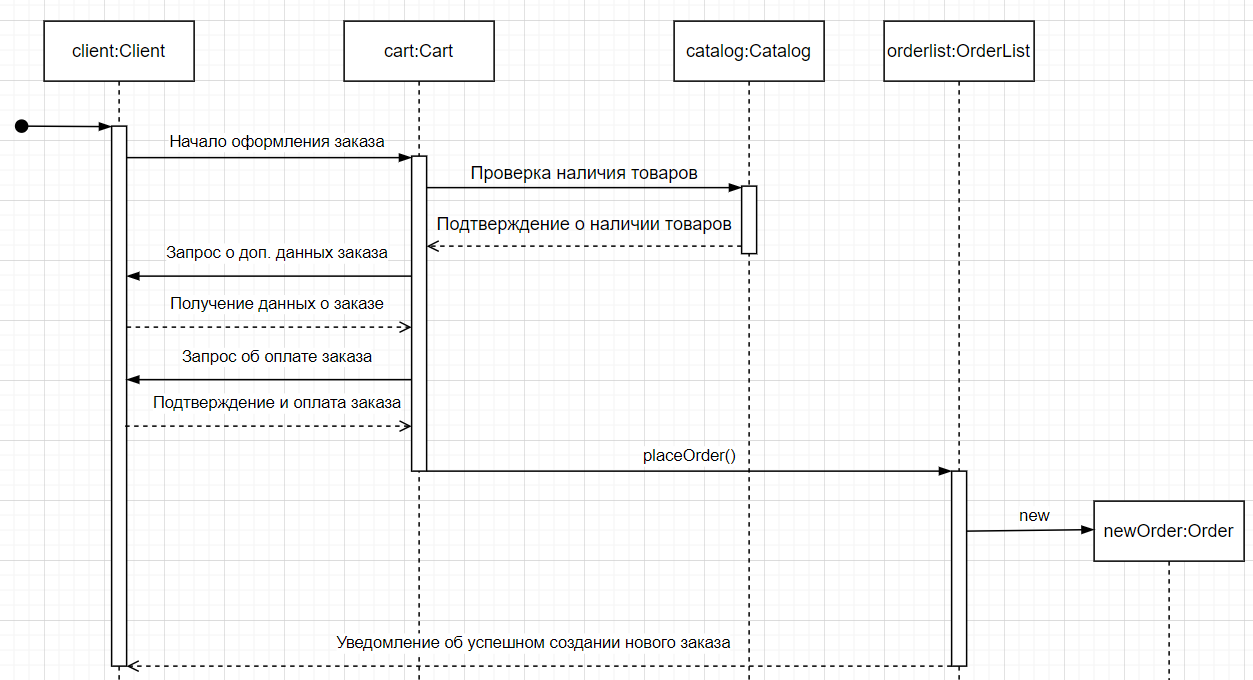


Рис. – Диаграмма последовательности «Оформление заказа клиентом»

Описание диаграммы:

Началом последовательности является объект класса Client, он начинает оформление заказа и передает сообщение об этом объекту класса Cart. Он, в свою очередь, передает сообщение объекту класса Catalog о проверке наличия товаров в магазине. Catalog подтверждает наличие товаров, находящихся в корзине. Cart отправляет запрос объекту класса Client о доп. данных заказа: способ получения заказа, проверка других свойств заказа. Cart получает эти данные и отправляет запрос об оплате заказа. После подтверждения оплаты клиентом Cart с помощью метода placeOrder() оформляет новый заказ в списке заказов OrderList. Для этого объект класса OrderList создает новый объект класса Order и помещает его в свою коллекцию. После этого клиенту отправляется сообщение об успешном создании нового заказа.

* 1. Построение и описание диаграммы пакетов

Пакет (package) – это конструкция UML, предназначенная для упорядочивания UML-моделей, а также для группировки классов.

Пакеты связываются друг с другом специальным отношением – зависимостью (dependence). Это направленное отношение, и идет оно от того пакета, который зависит, к пакету, который необходим тому, первому, зависимому.

Диаграмма пакетов служит, в первую очередь, для организации элементов в группы по какому-либо признаку с целью упрощения структуры и организации работы с моделью системы.

Классы составляют структурный костяк объектно-ориентированной системы. Хотя они исключительно полезны, но нужно нечто большее для структурирования больших систем, которые могут состоять из сотен классов. Пакет – это инструмент группирования, который позволяет взять любую конструкцию UML и объединить ее элементы в единицы высокого уровня. В основном пакеты служат для объединения классов в группы.

В модели UML каждый класс может включаться только в один пакет. Пакеты могут также входить в состав других пакетов. Пакет может содержать и подпакеты, и классы.

Каждый пакет представляет пространство имен (namespace). Это означает, что каждый класс внутри собственного пакета должен иметь уникальное имя. Имена пакетов на английском языке.

В распределении классов по пакетам могут помочь два правила: общий принцип замыкания (Common Closure Principle) и общий принцип повторного использования (Common Reuse Principle). Общий принцип замыкания гласит, что причины изменения классов пакета должны быть одинаковые. Общий принцип повторного использования утверждает, что классы должны использоваться повторно все вместе. Большинство причин, по которым классы должны объединяться в пакет, проистекают из зависимостей между классами.

Между пакетами может быть установлено отношение зависимости. Отношение зависимости показывает, что один пакет некоторым образом зависит от другого. Существует пять разных типов зависимостей между пакетами, каждый из которых имеет собственную семантику.

Зависимость «use» означает, что зависимости установлены скорее между элементами пакетов, а не между самими пакетами.

Зависимости «import» и «access» объединяют пространства имен клиента и поставщика. Это позволяет элементам клиента организовывать доступ к элементам поставщика с помощью неполных имен. Разница между этими двумя зависимостями в том, что «import» осуществляет открытое объединение, т. е. объединенные элементы поставщика становятся открытыми в клиенте. Тогда как «access» проводит закрытое объединение, т. е. объединенные элементы становятся в клиенте закрытыми.

Зависимость «trace» – «останется только один». Тогда как другие зависимости пакетов устанавливаются между сущностями одной модели, «trace» обычно представляет некоторое историческое развитие одного пакета в другой. Поэтому «trace» часто отражает отношения между разными моделями. Полная UML-модель может быть представлена в виде пакета с маленьким треугольником в верхнем правом углу.

Зависимость «merge» – сложное отношение, которое обозначает ряд трансформаций между элементами пакета поставщика и пакета клиента. Элементы пакета поставщика объединяются с элементами клиента для создания новых, расширенных клиентских элементов. Эта зависимость используется только при создании метамоделей (например, она широко применяется в метамодели UML) и не должна применяться в обычном ОО анализе и проектировании.

Диаграмма пакетов системы представлена на рис. 6.

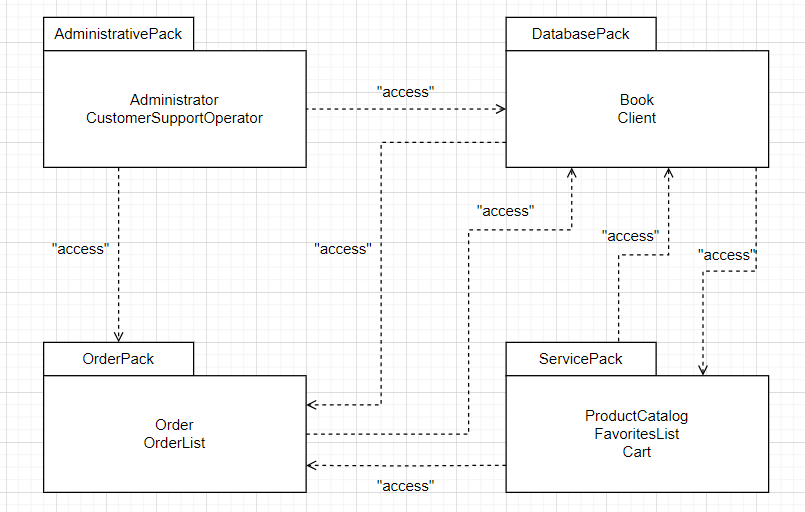


Рис. – Диаграмма пакетов системы «Книжный магазин»

Описание диаграммы:

AdministrativePack содержит классы «Administrator» «CustomerSupportOperator» и предназначен для управления системой с помощью классов пользователей Администратор и Оператор поддержки.

DatabasePack содержит классы «Client», «Book» и предназначен для сбора, хранения и работы с данными о пользователях и товарах.

ServicePack содержит классы «ProductCatalog», «FavoritesList», «Cart» и предназначен для работы с товарами (их просмотр в каталоге, добавление в список избранных и корзину клиента).

OrderPack содержит классы «Order», «OrderList» и предназначен для сбора, хранения и работы с данными о заказах клиентов.

Между пакетами AdministrativePack и DatabasePack одинарная связь, так как у классов Administrator и CustomerSupportOperator есть доступ к просмотру и изменению информации в базе данных книг и клиентов.

Аналогично, между пакетами AdministrativePack и OrderPack одинарная связь, так как у классов Administrator и CustomerSupportOperator есть доступ к просмотру и изменению информации в базе данных заказов.

Между пакетами ServicePack и OrderPack одинарная связь, так как данные для классов из пакета ServicePack есть доступ к изменению информации о заказах и списках заказов.

Между пакетами ServicePack и DatabasePack, OrderPack и DatabasePack двойная связь, так как для работы с товарами в каталоге, корзине, списке заказов и избранном нужна информация из базы данных, и в процессе этой работы данные о товарах в базе данных клиентов могут изменяться. Эти пакеты осуществляют двусторонний обмен данными.

* 1. Модульное и интеграционное тестирование

Модульные тесты (Unit tests). Модульные тесты тестируют только одну часть вашей реализации. Блок. Нет зависимостей или интеграций, нет специфики структуры. Они похожи на метод, который возвращает ссылку на определенном языке. Введение зависимости от внешних модулей или данных превращает модульные тесты в интеграционные. Возможность создания модульных тестов встроена в популярные среды разработки, такие как PyCharm и Eclipse.

Интеграционные тесты. В какой-то момент код связывается с базой данных, файловой системой или другой третьей стороной. Это может быть даже другой модуль в вашем приложении. Эта часть реализации должна быть проверена интеграционными тестами. Обычно они имеют более сложную настройку, которая включает подготовку сред тестирования, инициализацию зависимостей и так далее.

Функциональные тесты. Модульные и интеграционные тесты дают вам уверенность в том, что ваше приложение работает. Функциональные тесты смотрят на приложение с точки зрения пользователя и проверяют, что система работает должным образом. Модульные тесты (юнит-тесты) составляют большую базу набора тестов приложения. Как правило, они маленькие, их много, и они выполняются автоматически.

Тест состоит из трёх этапов:

• задание тестируемых данных (фикстур);

• использование тестируемого кода (вызов тестируемого метода);

• проверка результатов и сверка с ожидаемыми значениями.

Чтобы обеспечить модульность теста, нужно изолироваться от других слоёв приложения. Сделать это можно с помощью заглушек, моков и шпионов.

Мок (Mock) – объекты, которые настраиваются (например, специфично для каждого теста) и позволяют задать ожидания вызовы методов в виде ответов, которые мы планируем получить. Проверки соответствия ожиданиям проводятся через вызовы к Mock-объектам.

Заглушки (Stub) – обеспечивают жестко зашитый ответ на вызовы во время тестирования. Также они могут сохранять в себе информацию о вызове (например, параметры или количество этих вызовов). Такие иногда называют своим термином – шпион (Spy).

Разница этих терминов в том, что stub ничего не проверяет, а лишь имитирует заданное состояние. А mock – это объект, у которого есть ожидания. Например, что данный метод класса должен быть вызван определенное число раз. Иными словами, тест никогда не сломается из-за stub `а, а вот из-за mock`а может.

Современные инструментальные среды разработки приложений, такие как MS VisualStudio последних версий, предоставляют разработчикам программного обеспечения возможности создавать модульные и нагрузочные тесты, а также тесты для проверки пользовательского интерфейса. Для этого используются шаблоны тестовых проектов, к которым можно отнести следующие:

* макет модульного теста, который позволяет создавать тесты проверки модулей в процессе проектирования;
* макет с WEB-тестами производительности и нагрузочными тестами;
* макет с кодированными тестами пользовательского интерфейса.

Проверка метода validatePassword(String password):

* пароль должен содержать хотя бы одну цифру;
* пароль должен содержать хотя бы один специальный символ;
* пароль должен быть не менее 8 и не более 20 символов в длину;

Тестовые данные представлены в таблице 1.

Таблица 1 – Проверка метода validatePassword()

|  |  |  |
| --- | --- | --- |
| Требование | Пароль | Результат |
| Корректный пароль | ASQ102$$ | True |
| Количество символов (от 8 до 20) | AS2$ | False |
| AS32yreiwo1-9sesfbw29qnse9d8s0 | False |
| Наличие цифр | Dj7\*/RT23 | True |
| ASRTyreidRs$ | False |
| Наличие спецсимволов | @Urt45Er$ | True |
| AS32yreid8s0 | False |

Проверка метода validateEmail(String email):

* email должен заканчиваться верным доменом;
* email должен быть не более 30 символов в длину;
* email не содержит пробелов;

Тестовые данные представлены в таблице 2.

Таблица 2 – Проверка метода validateEmail()

|  |  |  |
| --- | --- | --- |
| Требование | Email | Результат |
| Корректный email | ivanov@gmail.com | True |
| Количество символов (до 30) | AS32yreiwo1r9sesfbw29qnse9d8s0@mail.ru | False |
| Оканчивается верным доменом | popova@mail.ru | True |
| smirnova@mail.com | False |
| Не содержит пробелов | smirnova@gmail.com | True |
| mironov 90@mail.ru | False |

Проверка методов класса ProductCatalog findProductByTitle (String Title), findProductByAuthor (String Author):

* При попытке обратиться к клиенту по несуществующему id, вызывается ArgumentNullException;

Тестовые данные представлены в таблице 3.

Таблица 3 – Проверка findProductByTitle(), findProductByAuthor()

|  |  |  |
| --- | --- | --- |
| Требование | Книги в каталоге | Результат |
| Корректный поиск по названию Title1 (2 искомые книги) | Book book1 = new Book("Title1", "Description1", "Author1", "Genre1", 2000, "Publisher1", 80.0);  Book book2 = new Book("Title1", "Description2", "Author1", "Genre1", 2004, "Publisher2", 90.0);  Book book3 = new Book("Title2", "Description3", "Author2", "Genre2", 2008, "Publisher3", 95.0); | List<Book> found = [book1, book2] |
| Корректный поиск по названию Title4 (нет искомых книг) | List<Book> found = [] |
| Корректный поиск по автору Author2 (1 искомая книга) | List<Book> found = [book3] |

Проверка void метода класса FavoritesList deleteProduct(Book book):

* Размер коллекции должен быть уменьшен, и он не должен содержать удаленный элемент;

Тестовые данные представлены в таблице 4.

Таблица 4 – Проверка метода deleteProduct()

|  |  |  |
| --- | --- | --- |
| Требование | Список избранного до удаления | Результат |
| Корректное удаление элемента из коллекции | [book1, book2, book3] | [book2, book3] |

Проверка виртуального метода класса Administrator getFullName() с использованием mock-объекта.

Тестовые данные представлены в таблице 5.

Таблица 5 – Проверка метода getFullName()

|  |  |  |
| --- | --- | --- |
| Требование | Экземпляр класса Mock<Administrator> | Результат |
| Корректное значение возвращаемой строки | Mock<Administrator> mock = new Mock<Administrator>("Попова Дарья", "popova@mail.ru", "popova$$12"); | True (actual = expected = «Попова Дарья») |

Скриншот проверки работы тестов представлен на рис. 7.

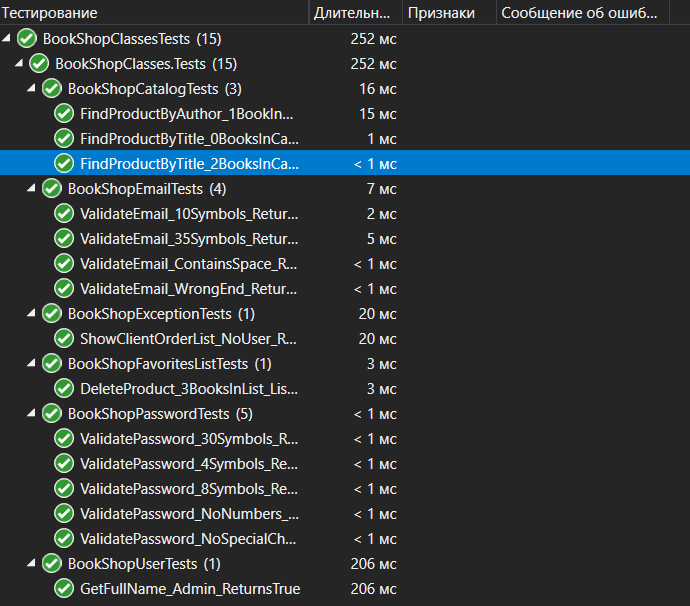
******

Рис. – Проверка работы тестов

Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода классов и методов тестов с комментариями представлен в приложении Б.

1. Паттерны
   1. Паттерны ООП: одиночка

В качестве среды разработки были выбраны Visual Studio и язык программирования C#.

**Имя** порождающего паттерна – Singleton (Одиночка).

**Задача** (условия применения паттерна). Этот паттерн гарантирует, что у класса есть только один объект (один экземпляр класса) и к этому объекту предоставляется глобальная точка доступа.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов). Паттерн Singleton в разрабатываемой системе может быть использован для создания класса ProductCatalog, поскольку экземпляр этого класса должен быть единственным.

Вся диаграмма состоит из одного единственного класса. Singleton определяет статический метод getInstance, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода getInstance должен стать единственным способом получить объект этого класса.

**Результат** (ожидаемые последствия применения паттерна)

В ходе работы был реализован паттерн Singleton для класса ProductCatalog разрабатываемой системы.

Плюсы:

* Гарантирует наличие единственного экземпляра класса.
* Предоставляет к нему глобальную точку доступа.
* Реализует отложенную инициализацию объекта-одиночки.

Минусы:

* Нарушает принцип единственной ответственности класса.
* Маскирует плохой дизайн.
* Проблемы мультипоточности.
* Требует постоянного создания Mock-объектов при юнит-тестировании.

Чтобы проверить паттерн, были добавлены 2 товара в каталог и проверено количество товаров в каталоге. По ссылке, созданной после обновления количества товаров в каталоге, был возвращен тот же единственный объект класса ProductCatalog. Скриншот результата проверки паттерна представлен на рис. 8. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении B.

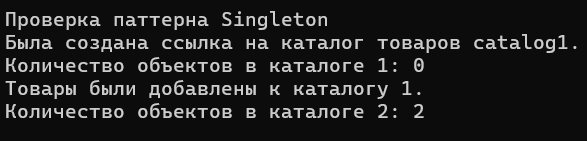


Рис. – Проверка паттерна Singleton

* 1. Паттерны ООП: медиатор

**Имя** паттерна поведения объектов – Mediator (Посредник).

**Задача** (условия применения паттерна). Паттерн «Посредник» используется, когда:

* имеются объекты, связи между которыми сложны и четко определены;
* нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
* поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов). Паттерн Mediator в разрабатываемой системе может быть использован для управления взаимодействием между классом Client и классами-коллегами Cart, OrderList, FavoritesList, поскольку между этими классами существует большое количество связей.

Участники паттерна:

*Mediator* – посредник;

* определяет интерфейс для обмена информацией с объектами Colleague;

*BookShopClientMediator* – конкретный посредник:

* реализует кооперативное поведение, координируя действия объектов *Colleague*;
* владеет информацией о коллегах и подсчитывает их;

*Colleague* – коллеги:

* каждый класс *Colleague* «знает» о своём объекте *Mediator*;
* все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

**Результат** (ожидаемые последствия применения паттерна). У паттерна «Посредник» есть следующие достоинства и недостатки:

* снижает число порождаемых подклассов;
* устраняет связанность между коллегами;
* упрощает протоколы взаимодействия объектов;
* абстрагирует способ кооперирования объектов;
* централизует управление.

Был реализован паттерн Mediator с классами Mediator, Colleague, BookShopClientMediator для разрабатываемой системы. Благодаря посреднику, все объекты-коллеги располагают информацией только о нем, поэтому количество взаимосвязей между Client и Cart, FavoritesList, OrderList сокращается.

Скриншот проверки корректности работы паттерна представлен на рис. 9. Так как объекты классов Cart, OrderList, FavoritesList связаны с объектом класса Client с помощью класса-посредника BookShopClientMediator, то при изменении полей в Cart, OrderList, FavoritesList, об этих изменениях будет уведомлен посредник. Соответствующие уведомления выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении Г.

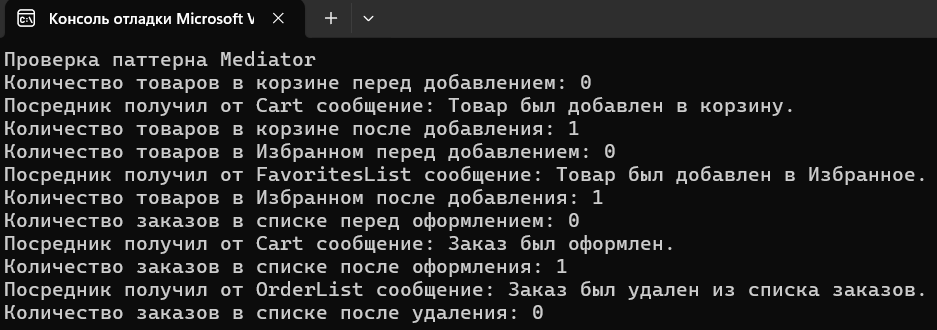


Рис. – Проверка паттерна Mediator

* 1. Паттерны ООП: фабричный метод

**Имя** порождающего паттерна – Factory Method (Фабричный метод).

**Задача** (условия применения паттерна). Паттерн предоставляет подклассам (дочерним классам, субклассам) интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Это позволяет использовать в коде программы не конкретные классы, а манипулировать абстрактными объектами на более высоком уровне.

Имеет смысл использовать данный паттерн, если:

* классу заранее неизвестно, объекты каких подклассов ему нужно создавать;
* класс спроектирован так, чтобы объекты, которые он создаёт, специфицировались подклассами;
* класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс принимает эти обязанности на себя.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов). Паттерн Factory Method в разрабатываемой системе может быть использован для создания объектом абстрактного класса Creator разных типов объектов класса Product: Book, Notebook, Pen. Классы BookCreator, NotebookCreator, PenCreator наследуются от абстрактного класса Creator. Участники паттерна:

*Product* – продукт:

* определяет интерфейс объектов, создаваемых абстрактным методом.

*Book, Notebook, Pen* – конкретный продукт:

* реализует интерфейс *Product*.

*Creator* — создатель:

* объявляет фабричный метод, который возвращает объект типа *Product*. Может также содержать реализацию этого метода «по умолчанию»;
* может вызывать фабричный метод для создания объекта типа *Product.*

*BookCreator, NotebookCreator, PenCreator* — конкретный создатель:

* переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса конкретного создателя.

**Результат** (ожидаемые последствия применения паттерна). Достоинства паттерна:

* Избавляет класс от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует принцип открытости/закрытости.

Недостаток паттерна:

* Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Был реализован паттерн Factory Method с классами Creator, BookCreator, NotebookCreator, PenCreator, Product, Book, Notebook, Pen для разрабатываемой системы. Паттерн выделяет код производства продуктов в одно место, упрощая поддержку кода и добавление новых продуктов в программу.

Скриншот проверки корректности работы паттерна представлен на рис. 10. Объекты классов, наследуемых от класса Creator, создают только экземпляры определённых классов. Соответствующие сообщения о созданиях объектов выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении Д.

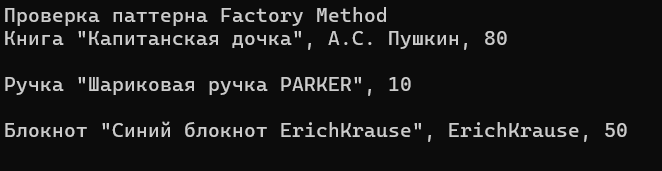


Рис. – Проверка паттерна Factory Method

* 1. Паттерны ООП: абстрактная фабрика

**Имя** порождающего паттерна – Abstract Factory (Абстрактная фабрика).

**Задача** (условия применения паттерна). Паттерн предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Имеет смысл использовать данный паттерн, если:

* программа должна быть независимой от процесса и типов создаваемых новых объектов;
* необходимо создать семейства или группы взаимосвязанных объектов исключая возможность одновременного использования объектов из разных этих наборов в одном контексте.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов). Паттерн Abstract Factory в разрабатываемой системе может быть использован для создания объектом клиентского класса ProductCatalog разных типов объектов класса Product: ElectronicBook, PrintedBook, Notebook, Pen. Классы ElectronicBookCreator, PrintedBookCreator, NotebookCreator, PenCreator наследуются от абстрактного класса ProductCreator. Участники паттерна:

* Абстрактные классы *Product, Book* определяют интерфейс для классов, объекты которых будут создаваться в программе;
* Конкретные классы *ElectronicBook, PrintedBook, Notebook, Pen* представляют конкретную реализацию абстрактных классов;
* Абстрактный класс фабрики *ProductCreator* определяет методы для создания объектов. Причем методы возвращают абстрактные продукты, а не их конкретные реализации;
* Конкретные классы фабрик *ElectronicBookCreator, PrintedBookCreator, NotebookCreator, PenCreator* реализуют абстрактные методы базового класса и непосредственно определяют какие конкретные продукты использовать;
* Класс клиента *Client* использует класс фабрики для создания объектов. При этом он использует исключительно абстрактный класс фабрики *ProductCreator* и абстрактные классы продуктов *Product, Book* и никак не зависит от их конкретных реализаций.

**Результат** (ожидаемые последствия применения паттерна).

Достоинства паттерна:

* Гарантирует сочетаемость создаваемых продуктов.
* Избавляет клиентский код от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует принцип открытости/закрытости.

Недостатки паттерна:

* Усложняет код программы из-за введения множества дополнительных классов.
* Требует наличия всех типов продуктов в каждой вариации.

Был реализован паттерн Abstract Factory с классами ProductCreator, PrintedBookCreator, ElectronicBookCreator, NotebookCreator, PenCreator, Product, ElectronicBook, PrintedBook, Notebook, Pen для разрабатываемой системы. Клиентский класс ProductCatalog создает объекты класса Product в зависимости от аргумента – объекта класса ProductCreator. Паттерн выделяет код производства продуктов в одно место, изолирует конкретные классы, упрощает замену семейств продуктов, гарантирует сочетаемость продуктов.

Скриншот проверки корректности работы паттерна представлен на рис. 11. Объекты классов, наследуемые от класса ProductCreator, создают только экземпляры определённых классов. Соответствующие сообщения о новых товарах в каталоге выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении Е.

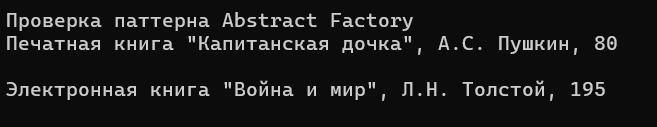


Рис. – Проверка паттерна Abstract Factory

* 1. Паттерны ООП: прокси

**Имя** структурирующего паттерна – Proxy (Прокси, Заместитель).

**Задача** (условия применения паттерна). Существует четыре ситуации, когда можно использовать паттерн Proxy:

* виртуальный Proxy. Он является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту, т. е. виртуальный заместитель создает «тяжелые» объекты по требованию;
* удаленный Proxy. Он предоставляет локального представителя для объекта, который находится в другом адресном пространстве («заглушки» в RPC и CORBA);
* защищающий Proxy контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа;
* интеллектуальный Proxy. Он выполняет дополнительные действия при доступе к объекту.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов). Паттерн Proxy в разрабатываемой системе может быть использован для создания виртуального заместителя для класса Client. Класс Client был переименован в RealClient, созданы интерфейс IClient (наследуется от интерфейса User) и класс заместителя ProxyClient.

Участники паттерна:

* *IClient* – субъект: определяет общий для *RealClient* и *ProxyClient* интерфейс, так что класс *ProxyClient* можно использовать везде, где ожидается *RealClient*;
* *RealClient* – реальный субъект: определяет реальный объект, представленный заместителем.
* *ProxyClient* при необходимости переадресует запросы объекту *RealClient*. Детали зависят от вида заместителя.

**Результат** (ожидаемые последствия применения паттерна). Достоинства паттерна:

* Позволяет контролировать сервисный объект незаметно для клиента.
* Может работать, даже если сервисный объект ещё не создан.
* Может контролировать жизненный цикл служебного объекта.
* Повышает быстродействие и безопасность кода.

Недостатки паттерна:

* Усложняет код программы из-за введения дополнительных классов.
* Увеличивает время отклика от сервиса.

Был реализован паттерн Proxy с классами RealClient и ProxyClient, а также интерфейсом IClient. Виртуальный заместитель используется для ленивой инициализации тяжелых объектов, как Client.

Скриншот проверки корректности работы паттерна представлен на рис. 12. Заместитель ProxyClient благодаря ленивой инициализации создает экземпляр RealClient только по требованию. Соответствующие сообщения о вызовах метода getFullName() выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении Ж.

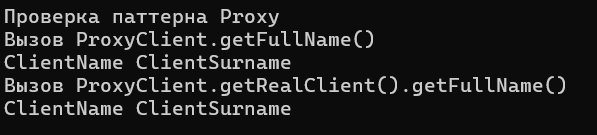


Рис. – Проверка паттерна Proxy

* 1. Паттерны ООП: состояние

**Имя** структурирующего паттерна – State (Состояние).

**Задача** (условия применения паттерна). State используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

Имеет смысл использовать данный паттерн, если:

* объект должен изменять свое поведение при изменении его внутреннего состояния;
* поведение, зависящее от состояния, должно определяться независимо. То есть добавление новых состояний не должно влиять на поведение существующих состояний.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов). Паттерн State в разрабатываемой системе может быть использован для описания состояний объектов класса Order: оформлен, доставлен и отменен. Участники паттерна:

* *OrderState*: определяет интерфейс состояния.
* Классы *PlacedOrder, CancelledOrder, DeliveredOrder* - конкретные реализации состояний.
* *Order*: представляет объект, поведение которого должно динамически изменяться в соответствии с состоянием. Выполнение же конкретных действий делегируется объекту состояния.

**Результат** (ожидаемые последствия применения паттерна).

Достоинства паттерна:

* Избавляет от множества больших условных операторов машины состояний.
* Концентрирует в одном месте код, связанный с определённым состоянием.
* Упрощает код контекста.

Недостаток паттерна:

* Может неоправданно усложнить код, если состояний мало и они редко меняются.

Был реализован паттерн State с классами OrderState, PlacedOrder, CancelledOrder, DeliveredOrder. Используя этот паттерн, можно сконцентрировать все изменения состояний класса Order в одном месте кода и без проблем добавить новые состояния для заказа.

Скриншот проверки корректности работы паттерна представлен на рис. 13. Благодаря абстрактному классу OrderState и наследуемым классам состояний заказа, можно отслеживать различные состояния заказа и их изменения. Сначала заказ был оформлен, затем отменен, после этого снова оформлен (восстановлен) и доставлен. Соответствующие сообщения об изменениях состояния заказа выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении К.

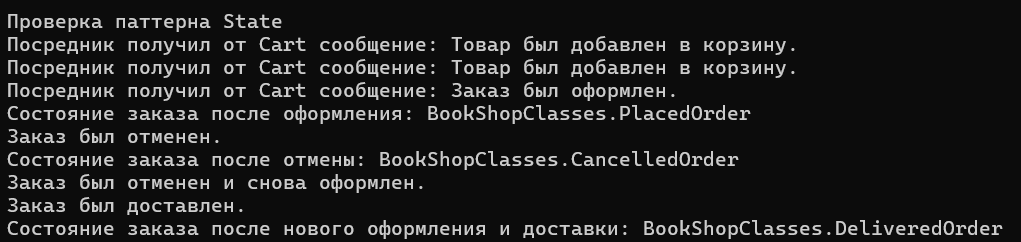


Рис. – Проверка паттерна State

* 1. Паттерны ООП: стратегия

**Имя** поведенческого паттерна – Strategy (Стратегия).

**Задача** (условия применения паттерна). Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

С помощью этого паттерна можно определить семейство алгоритмов, затем поместить каждый из них в собственный класс (их называют стратегиями) и сделать их объекты взаимозаменяемыми.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов).

Паттерн Strategy в разрабатываемой системе может быть использован для реализации поиска товаров в каталоге по разным ключам: автору и названию.

Участники паттерна:

* Интерфейс *IFindInCatalog*, который определяет метод *find*(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
* Классы *FindInCatalogByAuthor* и *FindInCatalogByTitle*, которые реализуют интерфейс *IFindInCatalog*, предоставляя свою версию метода *find*(). Подобных классов-реализаций может быть множество.
* Класс *ProductCatalog* хранит ссылку на объект *IFindInCatalog* и связан с интерфейсом *IFindInCatalog* отношением агрегации.

**Результат** (ожидаемые последствия применения паттерна).

Достоинства паттерна:

* Горячая замена алгоритмов на лету.
* Изолирует код и данные алгоритмов от остальных классов.
* Уход от наследования к делегированию.
* Реализует принцип открытости/закрытости.

Недостаток паттерна:

* Усложняет программу за счёт дополнительных классов.
* Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

Был реализован паттерн Strategy с интерфейсом IFindInCatalog и классами FindInCatalogByAuthor, FindInCatalogByTitle. Используя этот паттерн, можно реализовать поиск в каталоге по различным критериям и легко изменять каждый из способов.

Скриншот проверки корректности работы паттерна представлен на рис. 14. Интерфейс IFindInCatalog имеет метод find(). Реализации интерфейса производят поиск по каталогу в зависимости от ключа поиска. Соответствующие сообщения об найденных товарах выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении Л.

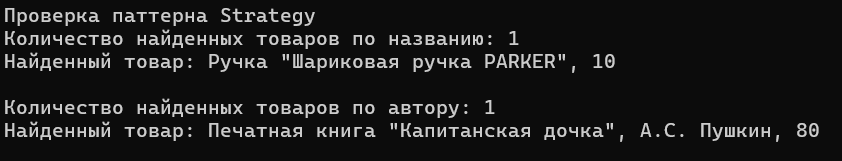


Рис. – Проверка паттерна Strategy

* 1. Паттерны ООП: легковес

**Имя** структурного паттерна – Flyweight (Легковес, Приспособленец).

**Задача** (условия применения паттерна). Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Применяется этот паттерн, когда выполнены все перечисленные условия:

* в приложении используется большое число объектов;
* из-за этого высоки расходы оперативной памяти;
* большую часть состояния объектов можно вынести за пределы их классов
* большие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено.

**Решение** (абстрактное описание решения задачи и модель решения в виде набора связанных классов).

Паттерн Flyweight в разрабатываемой системе может быть использован для создания типов продуктов ProductType, так как товаров может быть очень много, а такие поля, как автор и цена, могут часто совпадать.

**Результат** (ожидаемые последствия применения паттерна).

Достоинство паттерна:

* Экономит оперативную память.

Недостатки паттерна:

* Расходует процессорное время на поиск/вычисление контекста.
* Усложняет код программы из-за введения множества дополнительных классов.

Был реализован паттерн Flyweight с классами ProductType, ProductCreator (фабрика для Product). Используя этот паттерн, можно уменьшить объем используемой памяти за счет объединения некоторых полей Product в отдельный экземпляр класса ProductType.

Скриншот проверки корректности работы паттерна представлен на рис. 15. Создаются 5 товаров и помещаются в каталог. При этом были созданы только 3 разных типа товаров. При этом у каждого объекта класса Product есть ссылка на объект класса ProductType. Таким образом, решается проблема с большим количеством одинаковых полей у объектов. Соответствующая информация о типах товаров выведены в консоль. Код программы с комментариями находится в репозитории по ссылке в приложении А. Листинг кода с реализацией паттерна и комментариями представлен в приложении М.

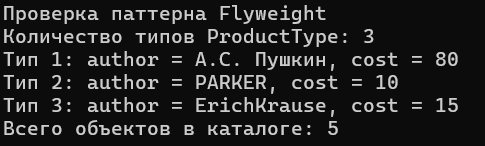


Рис. – Проверка паттерна Flyweight

Заключение

В ходе выполнения курсовой работы была спроектирована информационная система «Книжный магазин».

Система выполняет поставленные функциональные и нефункциональные требования: были автоматизированы процессы взаимодействия клиентов магазина и его сотрудников, управления каталогом товаров и списками заказов пользователей.

При проектировании системы были построены и описаны необходимые диаграммы: диаграмма вариантов использования, концептуальная диаграмма классов, диаграмма состояний, диаграмма проектных классов, диаграмма последовательности для операций проектных классов, диаграмма пакетов. Благодаря этим диаграммам, разработка системы становится существенно легче.

В ходе проектирования системы также были реализованы несколько паттернов проектирования на языке программирования C# в среде Visual Studio: Singleton, Mediator, Factory Method, Abstract Factory, Proxy, State, Strategy, Flyweight. Все использованные паттерны были протестированы на корректность работы.

Работая над этой системой в будущем, можно расширить функционал, создать удобный пользовательский интерфейс.

Список литературы

1. **UML Diagram Templates UML // Lucidchart. – [Б. м.], 2024. – URL: https://www.lucidchart.com/blog/uml-diagram-templates#usecase (дата обращения: 05.09.2024).**
2. **Полное руководство по 14 типам диаграмм UML // Кибермедиана. – [Б. м.], 2022. – URL: https://www.cybermedian.com/ru/a-comprehensive-guide-to-14-types-of-uml-diagram/ (дата обращения: 05.09.2024).**
3. Приемы объектно-ориентированного проектирования. Паттерны проектирования : справочник / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. — Москва : ДМК Пресс, 2007. — 368 с. — ISBN 5-93700-023-4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/1220 (дата обращения: 15.10.2024). — Режим доступа: для авториз. пользователей.
4. Unit Testing Doesn’t Affect Codebases the Way You Would Think // NDepend Blog. – [Б. м.], 2004-2023. – URL: https://blog.ndepend.com/unit-testing-affect-codebases/ (дата обращения: 08.10.2024).
5. Фримен Э., Робсон Э., Сьерра К., Бейтс Б. Head First. Паттерны проектирования. Обновленное юбилейное издание. – СПб.: Питер, 2018. – 656 с.: ил. — (Серия «Head First O’Reilly»).
6. Полное руководство по языку программирования C# 12 и платформе .NET 8 // METANIT.COM: Сайт о программировании. – [Б. м.], 2024. – URL: https://metanit.com/sharp/tutorial/ (дата обращения: 15.10.2024).

Приложение А

Репозиторий разрабатываемой системы. URL: <https://codelab.tpu.ru/ass233/bookshop>

Приложение Б

Листинг кода. Модульное и интеграционное тестирование. Классы и методы тестов с комментариями.

namespace BookShopClasses.Tests

{

[TestClass()]

public class BookShopPasswordTests

{

//простая проверка метода, возвращающего значение

//корректный пароль => возвращает True

[TestMethod()]

public void ValidatePassword\_8Symbols\_ReturnsTrue()

{

//Arrange

string password = "ASQ102$$";

bool expected = true;

//Act

bool actual = BookShopValidate.validatePassword(password);

//Assert

Assert.AreEqual(expected, actual);

}

//простая проверка метода, возвращающего значение

//в пароле меньше 8 символов => возвращает False

[TestMethod()]

public void ValidatePassword\_4Symbols\_ReturnsFalse()

{

//Arrange

string password = "AS2$";

//Act

bool actual = BookShopValidate.validatePassword(password);

//Assert

Assert.IsFalse(actual);

}

//простая проверка метода, возвращающего значение

//в пароле больше 20 символов => возвращает False

[TestMethod()]

public void ValidatePassword\_30Symbols\_ReturnsFalse()

{

//Arrange

string password = "AS32yreiwo1-9sesfbw29qnse9d8s0";

//Act

bool actual = BookShopValidate.validatePassword(password);

//Assert

Assert.IsFalse(actual);

}

//простая проверка метода, возвращающего значение

//в пароле нет специальных символов => возвращает False

[TestMethod()]

public void ValidatePassword\_NoSpecialChars\_ReturnsFalse()

{

//Arrange

string password = "AS32yreid8s0";

//Act

bool actual = BookShopValidate.validatePassword(password);

//Assert

Assert.IsFalse(actual);

}

//простая проверка метода, возвращающего значение

//в пароле нет цифр => возвращает False

[TestMethod()]

public void ValidatePassword\_NoNumbers\_ReturnsFalse()

{

//Arrange

string password = "ASRTyreidRs$";

//Act

bool actual = BookShopValidate.validatePassword(password);

//Assert

Assert.IsFalse(actual);

}

}

[TestClass()]

public class BookShopEmailTests

{

//простая проверка метода, возвращающего значение

//корректный email => возвращает True

[TestMethod()]

public void ValidateEmail\_10Symbols\_ReturnsTrue()

{

//Arrange

string email = "ivanov@gmail.com";

bool expected = true;

//Act

bool actual = BookShopValidate.validateEmail(email);

//Assert

Assert.AreEqual(expected, actual);

}

//простая проверка метода, возвращающего значение

//неверный домен => возвращает False

[TestMethod()]

public void ValidateEmail\_WrongEnd\_ReturnsFalse()

{

//Arrange

string email = "smirnova@mail.com";

//Act

bool actual = BookShopValidate.validateEmail(email);

//Assert

Assert.IsFalse(actual);

}

//простая проверка метода, возвращающего значение

//email имеет более 30 символов в длину => возвращает False

[TestMethod()]

public void ValidateEmail\_35Symbols\_ReturnsFalse()

{

//Arrange

string email = "AS32yreiwo1r9sesfbw29qnse9d8s0@mail.ru";

//Act

bool actual = BookShopValidate.validateEmail(email);

//Assert

Assert.IsFalse(actual);

}

//простая проверка метода, возвращающего значение

//email содержит пробелы

[TestMethod()]

public void ValidateEmail\_ContainsSpace\_ReturnsFalse()

{

//Arrange

string email = "mironov 90@mail.ru";

//Act

bool actual = BookShopValidate.validateEmail(email);

//Assert

Assert.IsFalse(actual);

}

}

[TestClass()]

public class BookShopExceptionTests

{

//тест, проверяющий, что метод действительно возбуждает исключение определённого типа при возникновении исключительной ситуации

//попытка обратиться к клиенту по несуществующему id => ArgumentNullException

[TestMethod()]

public void ShowClientOrderList\_NoUser\_ReturnsTrue()

{

//Arrange

BookShopClientMediator mediator = new BookShopClientMediator();

CustomerSupportOperator support = new CustomerSupportOperator(mediator, "Смирнова Ольга", "smirnova@gmail.com", "smirnovaOlga10&");

IClient client = new ProxyClient(mediator, "Иванов Петр", "ivanov@mail.ru", "ivanov10+", "+7-912-319-29-10");

int clientID = client.getID();

//Act

bool actual = false;

try

{

support.showClientOrderList(clientID-1);

}

catch (ArgumentNullException)

{

actual = true;

}

//Assert

Assert.IsTrue(actual);

}

}

[TestClass()]

public class BookShopCatalogTests

{

//тест, проверяющий возвращаемое значение, являющееся коллекцией

//по названию Title1 найдены 2 книги => Коллекция с двумя элементами [book1, book2]

[TestMethod()]

public void FindProductByTitle\_2BooksInCatalog\_ReturnsTrue()

{

//Arrange

Globals.getInstance().getProductCatalog().getCatalog().Clear();

BookShopClientMediator mediator = new BookShopClientMediator();

Administrator admin = new Administrator(mediator, "AdminName AdminSurname", "admin@gmail.com", "admin10%");

Book book1 = new PrintedBook("Title1", "Description1", "Author1", "Genre1", 2000, "Publisher1", 80.0, 100);

Book book2 = new PrintedBook("Title1", "Description2", "Author1", "Genre1", 2004, "Publisher2", 90.0, 120);

Book book3 = new PrintedBook("Title2", "Description3", "Author2", "Genre2", 2008, "Publisher3", 95.0, 150);

admin.addProduct((book1));

admin.addProduct((book2));

admin.addProduct((book3));

//Act

bool actual = false;

Globals.getInstance().getProductCatalog().findInCatalog = new FindInCatalogByTitle();

List<Product> found = Globals.getInstance().getProductCatalog().find("Title1");

if (found.Count == 2)

{

actual = true;

}

//Assert

Assert.IsTrue(actual);

}

//тест, проверяющий возвращаемое значение, являющееся коллекцией

//по названию Title4 найдено 0 книг (пустая коллекция) => Пустая коллекция []

[TestMethod()]

public void FindProductByTitle\_0BooksInCatalog\_ReturnsTrue()

{

//Arrange

BookShopClientMediator mediator = new BookShopClientMediator();

Administrator admin = new Administrator(mediator, "AdminName AdminSurname", "admin@gmail.com", "admin10%");

Book book1 = new PrintedBook("Title1", "Description1", "Author1", "Genre1", 2000, "Publisher1", 80.0, 100);

Book book2 = new PrintedBook("Title1", "Description2", "Author1", "Genre1", 2004, "Publisher2", 90.0, 120);

Book book3 = new PrintedBook("Title2", "Description3", "Author2", "Genre2", 2008, "Publisher3", 95.0, 150);

admin.addProduct((book1));

admin.addProduct((book2));

admin.addProduct((book3));

//Act

bool actual = false;

Globals.getInstance().getProductCatalog().findInCatalog = new FindInCatalogByTitle();

List<Product> found = Globals.getInstance().getProductCatalog().find("Title4");

if (found.Count == 0)

{

actual = true;

}

//Assert

Assert.IsTrue(actual);

}

//тест, проверяющий возвращаемое значение, являющееся коллекцией

//по автору Author2 найдена 1 книга => Коллекция с одним элементом [book3]

[TestMethod()]

public void FindProductByAuthor\_1BookInCatalog\_Returns1Book()

{

//Arrange

BookShopClientMediator mediator = new BookShopClientMediator();

Administrator admin = new Administrator(mediator, "AdminName AdminSurname", "admin@gmail.com", "admin10%");

Book book1 = new PrintedBook("Title1", "Description1", "Author1", "Genre1", 2000, "Publisher1", 80.0, 100);

Book book2 = new PrintedBook("Title1", "Description2", "Author1", "Genre1", 2004, "Publisher2", 90.0, 120);

Book book3 = new PrintedBook("Title2", "Description3", "Author2", "Genre2", 2008, "Publisher3", 95.0, 150);

admin.addProduct((book1));

admin.addProduct((book2));

admin.addProduct((book3));

List<Product> expected = new List<Product>(new Book[1] { new PrintedBook("Title2", "Description3", "Author2",

"Genre2", 2008, "Publisher3", 95.0, 150) });

//Act

Globals.getInstance().getProductCatalog().findInCatalog = new FindInCatalogByAuthor();

List<Product> actual = Globals.getInstance().getProductCatalog().find("Author2");

//Assert

Assert.IsNotNull(actual);

Assert.AreEqual(actual.Count, expected.Count);

Assert.AreEqual(actual[0].title, expected[0].title);

Assert.AreEqual(actual[0].type.author, expected[0].type.author);

}

}

[TestClass()]

public class BookShopFavoritesListTests

{

//тест для метода, не возвращающего значение (void)

//проверка удаления одного товара из списка избранного с 3 элементами => список избранного с 2 элементами

[TestMethod()]

public void DeleteProduct\_3BooksInList\_ListCountIs2()

{

//Arrange

BookShopClientMediator mediator = new BookShopClientMediator();

IClient client = new ProxyClient(mediator, "ClientName ClientSurname", "client@gamil.com", "client10%", "+7-123-456-78-90");

Book book1 = new PrintedBook("Title1", "Description1", "Author1", "Genre1", 2000, "Publisher1", 80.0, 100);

Book book2 = new PrintedBook("Title1", "Description2", "Author1", "Genre1", 2004, "Publisher2", 90.0, 120);

Book book3 = new PrintedBook("Title2", "Description3", "Author2", "Genre2", 2008, "Publisher3", 95.0, 150);

client.addProductToFavorites(book1);

client.addProductToFavorites(book2);

client.addProductToFavorites(book3);

//Act

bool actual = false;

client.deleteProductFromFavorites(book1);

if (client.getFavoritesList().productList.Count == 2)

{

actual = true;

}

//Assert

Assert.IsTrue(actual);

}

}

[TestClass()]

public class BookShopUserTests

{

//тест, использующий mock-объект

//проверка виртуального метода getFullName() => True (aclual и expected совпадают)

[TestMethod()]

public void GetFullName\_Admin\_ReturnsTrue()

{

BookShopClientMediator mediator = new BookShopClientMediator();

//Arrange

Mock<Administrator> mock = new Mock<Administrator>(mediator, "Попова Дарья", "popova@mail.ru", "popova$$12");

String expected = "Попова Дарья";

//Setup

mock.Setup(s => s.getFullName()).Returns(expected);

//Act

String actual = mock.Object.getFullName();

//Assert

Assert.AreEqual(actual, expected);

}

}

}

Приложение В

Листинг кода (класс ProductCatalog с применением паттерна Singleton)

//класс ProductCatalog (паттерн Singleton)

public class ProductCatalog

{

//приватный статичный экземпляр класса

private static ProductCatalog instance;

//приватный конструктор класса

private ProductCatalog()

{

productCatalog = new List<Book>();

}

//метод getInstance

public static ProductCatalog getInstance()

{

if (instance == null)

{

instance = new ProductCatalog();

}

return instance;

}

//список книг в каталоге

private List<Book> productCatalog;

public List<Book> getCatalog()

{

return productCatalog;

}

//метод поиска товара по названию

public List<Book> findProductByTitle(String title)

{

List<Book> result = new List<Book>();

for (int i = 0; i < this.productCatalog.Count; i++)

{

if (this.productCatalog[i].title.Equals(title))

{

result.Add(this.productCatalog[i]);

}

}

return result;

}

//метод поиска товара по автору

public List<Book> findProductByAuthor(String author)

{

List<Book> result = new List<Book>();

for (int i = 0; i < this.productCatalog.Count; i++)

{

if (this.productCatalog[i].author.Equals(author))

{

result.Add(this.productCatalog[i]);

}

}

return result;

}

//метод сортировки товаров по цене

public List<Book> sortProductsByCost()

{

List<Book> result = new List<Book>(productCatalog);

for (int i = 0; i < result.Count - 1; i++)

{

for (int j = i + 1; j < result.Count; j++)

{

Book book1 = result[i];

Book book2 = result[j];

if (book1.cost > book2.cost)

{

Book temp = result[i];

result[i] = result[j];

result[j] = temp;

}

}

}

return result;

}

}

Приложение Г

Листинг кода. Реализация паттерна Mediator.

//абстрактный класс Mediator (Посредник)

public abstract class Mediator

{

//абстрактный метод уведомления

public abstract string notify(Colleague colleague, String message);

public abstract void addProductToCart(RealClient client, Product product);

public abstract void deleteProductFromCart(RealClient client, Product product);

public abstract void editProductNumberInCart(RealClient client, Product product, int k);

public abstract void addProductToFavorites(RealClient client, Product product);

public abstract void deleteProductFromFavorites(RealClient client, Product product);

public abstract void placeOrder(RealClient client, int card, string way);

public abstract void deleteOrderFromOrderList(RealClient client, Order order);

}

//абстрактный класс Colleague (Паттерн Mediator)

public abstract class Colleague

{

//ссылка на посредника

protected Mediator mediator;

//конструктор класса

public Colleague(Mediator mediator)

{

this.mediator = mediator;

}

}

//реализация посредника BookShopClientMediator

public class BookShopClientMediator : Mediator

{

private Cart cart;

private FavoritesList favorites;

private OrderList orders;

public void SetCart(Cart cart) => this.cart = cart;

public void SetFavorites(FavoritesList favs) => this.favorites = favs;

public void SetOrders(OrderList ords) => this.orders = ords;

public override void addProductToCart(RealClient client, Product product)

{

client.addProductToCart(product);

}

public override string notify(Colleague colleague, string message)

{

if (colleague.GetType().Equals(typeof(Cart)))

{

return "Посредник получил от Cart сообщение: " + message;

}

else if (colleague.GetType().Equals(typeof(FavoritesList)))

{

return "Посредник получил от FavoritesList сообщение: " + message;

}

else if (colleague.GetType().Equals(typeof(OrderList)))

{

return "Посредник получил от OrderList сообщение: " + message;

}

else

{

return "";

}

}

public override void addProductToFavorites(RealClient client, Product product)

{

client.favoritesList.addProduct(product);

}

public override void deleteOrderFromOrderList(RealClient client, Order order)

{

client.orderList.deleteOrder(order);

}

public override void deleteProductFromCart(RealClient client, Product product)

{

client.cart.deleteProduct(product);

}

public override void deleteProductFromFavorites(RealClient client, Product product)

{

client.favoritesList.deleteProduct(product);

}

public override void editProductNumberInCart(RealClient client, Product product, int k)

{

client.cart.editProductNumber(product, k);

}

public override void placeOrder(RealClient client, int card, string way)

{

client.cart.placeOrder(card, way);

}

}

//класс FavoritesList (наследуется от класса Colleague)

public class FavoritesList : Colleague

{

public List<Product> productList;

//конструктор класса

public FavoritesList(Mediator mediator) : base(mediator)

{

this.productList = new List<Product>();

}

//метод добавления товара в список избранного

public void addProduct(Product product)

{

productList.Add(product);

Console.WriteLine(mediator.notify(this, "Товар был добавлен в Избранное."));

}

//метод удаления товара из списка избранного

public void deleteProduct(Product product)

{

for (int i = 0; i < productList.Count; i++)

{

if (product.Equals(productList[i]))

{

productList.Remove(productList[i]);

i -= 1;

}

}

Console.WriteLine(mediator.notify(this, "Товар был удален из Избранного."));

}

}

//класс Cart (наследуется от класса Colleague)

public class Cart : Colleague

{

public List<Product> productList;

private double totalPrice;

//конструктор класса

public Cart(Mediator mediator) : base(mediator)

{

this.productList = new List<Product>();

this.totalPrice = 0;

}

//метод, возвращающий полную стоимость всех товаров в корзине

public double getPrice()

{

double result = 0;

for (int i = 0; i < productList.Count; i++)

{

result += productList[i].type.cost;

}

totalPrice = result;

return result;

}

//метод добавления товара в корзину

public void addProduct(Product product)

{

productList.Add(product);

Console.WriteLine(mediator.notify(this, "Товар был добавлен в корзину."));

}

//метод изменения числа товаров в корзине на k

public void editProductNumber(Product product, int k)

{

for (int i = 0; i < k; i++)

{

productList.Add(product);

}

Console.WriteLine(mediator.notify(this, "Количество товара в корзине было изменено."));

}

//метод удаления товара из корзины

public void deleteProduct(Product product)

{

for (int i = 0; i < productList.Count; i++)

{

if (product.Equals(productList[i]))

{

productList.Remove(productList[i]);

i -= 1;

}

}

Console.WriteLine(mediator.notify(this, "Товар был удален из корзины."));

}

//метод оформления нового заказа

public Order placeOrder(int cardNumber, String gettingWay)

{

RealClient client = null;

for (int i = 0; i < Globals.getInstance().getUsersList().Count; i++)

{

User user = Globals.getInstance().getUsersList()[i];

if (user.GetType().Equals(typeof(RealClient)))

{

RealClient userClient = (RealClient)user;

if (userClient.cart.Equals(this))

{

client = (RealClient)Globals.getInstance().getUsersList()[i];

}

}

}

Order order = new Order(client.getCart().productList, cardNumber, gettingWay);

client.getOrderList().orderList.Add(order);

Console.WriteLine(mediator.notify(this, "Заказ был оформлен."));

return order;

}

}

//класс OrderList (наследуется от класса Colleague)

public class OrderList : Colleague

{

public List<Order> orderList;

//конструктор класса

public OrderList(Mediator mediator) : base(mediator)

{

this.orderList = new List<Order>();

}

//метод удаления заказа из списка заказов

public void deleteOrder(Order order)

{

for (int i = 0; i < orderList.Count; i++)

{

if (order.Equals(orderList[i]))

{

orderList[i].cancel();

orderList.Remove(orderList[i]);

}

}

Console.WriteLine(mediator.notify(this, "Заказ был удален из списка заказов."));

}

//метод, возвращающий список заказов в хронологическом порядке

public List<Order> showOrderHistory()

{

List<Order> result = new List<Order>(orderList);

for (int i = 0; i < result.Count - 1; i++)

{

for (int j = i + 1; j < result.Count; j++)

{

Order order1 = result[i];

Order order2 = result[j];

if (order1.orderTime > order2.orderTime)

{

Order temp = result[i];

result[i] = result[j];

result[j] = temp;

}

}

}

return result;

}

//переопределённый метод ToString()

public override string ToString()

{

StringBuilder result = new StringBuilder("Список заказов клиента \n");

for (int i = 0; i < orderList.Count; i++)

{

result.Append(orderList[i].ToString()).Append("\n");

}

return result.ToString();

}

}

//интерфейс User

public interface User

{

public int getID();

public String getEmail();

public String getPassword();

public String getFullName();

}

//класс Client (паттерн Mediator)

public class Client : User

{

protected int userID;

protected String fullName;

protected String email;

protected String password;

protected Mediator mediator;

protected String phoneNumber;

//конструктор класса

public Client(Mediator mediator, String fullname, String email, String password, String phone)

{

this.mediator = mediator;

this.userID = Globals.getInstance().getUserID();

this.fullName = fullname;

if (BookShopValidate.validateEmail(email))

{

this.email = email;

}

else throw new Exception();

if (BookShopValidate.validatePassword(password))

{

this.password = password;

}

else throw new Exception();

Globals.getInstance().getUsersList().Add(this);

this.phoneNumber = phone;

this.cart = new Cart(mediator);

this.favoritesList = new FavoritesList(mediator);

this.orderList = new OrderList(mediator);

}

//у каждого Client свои корзина, список избранного и список заказов

public Cart cart { get; private set; }

public OrderList orderList { get; private set; }

public FavoritesList favoritesList { get; private set; }

public void addProductToCart(Product product)

{

cart.addProduct(product);

}

public void deleteProductFromCart(Product product)

{

cart.deleteProduct(product);

}

public void editProductNumberInCart(Product product, int k)

{

cart.editProductNumber(product, k);

}

public void addProductToFavorites(Product product)

{

favoritesList.addProduct(product);

}

public void deleteProductFromFavorites(Product product)

{

favoritesList.deleteProduct(product);

}

public void placeOrder(int card, String way)

{

cart.placeOrder(card, way);

}

public void deleteOrderFromOrderList(Order order)

{

orderList.deleteOrder(order);

}

//удаление собственного аккаунта

public void deleteAccount()

{

Globals.getInstance().getUsersList().Remove(this);

}

public Cart getCart()

{

return cart;

}

public FavoritesList getFavoritesList()

{

return favoritesList;

}

public OrderList getOrderList()

{

return orderList;

}

public int getID()

{

return userID;

}

public String getEmail()

{

return email;

}

public String getPassword()

{

return password;

}

public virtual String getFullName()

{

return fullName;

}

}

Приложение Д

Листинг кода. Реализация паттерна Factory Method.

//абстрактный класс Creator (паттерны Factory Method, Abstract Factory, Flyweight)

public abstract class ProductCreator

{

public static List<ProductType> types = new List<ProductType>();

public static ProductType getProductType(string author, double cost)

{

ProductType type = null;

for (int i = 0; i < types.Count; i++)

{

if (types[i].author.Equals(author) && types[i].cost.Equals(cost))

{

type = types[i];

break;

}

}

if (type == null)

{

type = new ProductType(author, cost);

types.Add(type);

}

return type;

}

public ProductCreator() { }

// фабричный метод

abstract public Product create(params object[] args);

}

//класс BookCreator (паттерны Factory Method и Abstract Factory)

public abstract class BookCreator : ProductCreator

{

public BookCreator() : base() { }

// переопределённый фабричный метод

public abstract override Product create(params object[] args);

}

//класс NotebookCreator (паттерн Factory Method)

public class NotebookCreator : ProductCreator

{

public NotebookCreator() : base() { }

// переопределённый фабричный метод

public override Product create(params object[] args)

{

return new Notebook((string) args[0], (string)args[1], (string)args[2], (string)args[3],

(int)args[4], (double)args[5]);

}

}

//класс PenCreator (паттерн Factory Method)

public class PenCreator : ProductCreator

{

public PenCreator() : base() { }

// переопределённый фабричный метод

public override Product create(params object[] args)

{

return new Pen((string)args[0], (string)args[1], (string)args[2], (string)args[3],

(string)args[4], (double)args[5]);

}

}

//абстрактный класс Book, наследуется от Product

public abstract class Book : Product

{

public String genre;

public int publicationYear;

public String publisher;

//конструктор класса

public Book(string title, string description, string author, string genre, int publicationYear, string publisher, double cost)

: base(title, description, ProductCreator.getProductType(author, cost))

{

this.genre = genre;

this.publicationYear = publicationYear;

this.publisher = publisher;

}

//абстрактный метод ToString()

public abstract override string ToString();

}

//класс Notebook, наследуется от Product

public class Notebook : Product

{

public String sheet\_type;

public int pages;

//конструктор класса

public Notebook(string title, string description, string author, string sheet\_type, int pages, double cost)

: base(title, description, ProductCreator.getProductType(author, cost))

{

this.pages = pages;

if (sheet\_type.Equals("В клетку") || sheet\_type.Equals("В линейку") || sheet\_type.Equals("Без разметки")

|| sheet\_type.Equals("Другое"))

{

this.sheet\_type = sheet\_type;

}

else throw new ArgumentException();

}

//переопределённый метод ToString()

public override string ToString()

{

String result = "Блокнот \"" + this.title + "\", " + this.type.author + ", " + this.type.cost + "\n";

return result;

}

}

//класс Pen, наследуется от Product

public class Pen : Product

{

public String color;

public String refill;

//конструктор класса

public Pen(string title, string description, string author, string color, string refill, double cost)

: base(title, description, ProductCreator.getProductType(author, cost))

{

this.color = color;

if (refill.Equals("Шариковая") || refill.Equals("Гелевая") || refill.Equals("Другая"))

{

this.refill = refill;

}

else throw new ArgumentException();

}

//переопределённый метод ToString()

public override string ToString()

{

String result = "Pучка \"" + this.title + "\", " + this.type.cost + "\n";

return result;

}

}

Приложение Е

Листинг кода. Реализация паттерна Abstract Factory.

//класс BookCreator (паттерны Factory Method и Abstract Factory)

public abstract class BookCreator : ProductCreator

{

public BookCreator() : base() { }

// переопределённый фабричный метод

public abstract override Product create(params object[] args);

}

//класс PrintedBookCreator (паттерн Abstract Factory)

public class PrintedBookCreator : BookCreator

{

public PrintedBookCreator() : base() { }

// переопределённый фабричный метод

public override Product create(params object[] args)

{

return new PrintedBook((string)args[0], (string)args[1], (string)args[2], (string)args[3],

(int)args[4], (string)args[5], (double)args[6], (int)args[7]);

}

}

//класс EbookCreator (паттерн Abstract Factory)

public class ElectronicBookCreator : BookCreator

{

public ElectronicBookCreator() : base() { }

// переопределённый фабричный метод

public override Product create(params object[] args)

{

return new ElectronicBook((string)args[0], (string)args[1], (string)args[2], (string)args[3],

(int)args[4], (string)args[5], (double)args[6], (double)args[7]);

}

}

//класс NotebookCreator (паттерн Factory Method)

public class NotebookCreator : ProductCreator

{

public NotebookCreator() : base() { }

// переопределённый фабричный метод

public override Product create(params object[] args)

{

return new Notebook((string) args[0], (string)args[1], (string)args[2], (string)args[3],

(int)args[4], (double)args[5]);

}

}

//класс PenCreator (паттерн Factory Method)

public class PenCreator : ProductCreator

{

public PenCreator() : base() { }

// переопределённый фабричный метод

public override Product create(params object[] args)

{

return new Pen((string)args[0], (string)args[1], (string)args[2], (string)args[3],

(string)args[4], (double)args[5]);

}

}

//абстрактный класс Book, наследуется от Product

public abstract class Book : Product

{

public String genre;

public int publicationYear;

public String publisher;

//конструктор класса

public Book(string title, string description, string author, string genre, int publicationYear, string publisher, double cost)

: base(title, description, ProductCreator.getProductType(author, cost))

{

this.genre = genre;

this.publicationYear = publicationYear;

this.publisher = publisher;

}

//абстрактный метод ToString()

public abstract override string ToString();

}

//реализация абстрактного класса Book

public class ElectronicBook : Book

{

public double fileSize;

//конктруктор класса

public ElectronicBook(string title, string description, string author, string genre, int publicationYear,

string publisher, double cost, double fileSize) : base(title, description, author, genre, publicationYear, publisher, cost)

{

this.fileSize = fileSize;

}

//переопределённый метод ToString()

public override string ToString()

{

String result = "Электронная книга \"" + this.title + "\", " + this.type.author + ", " + this.type.cost + "\n";

return result;

}

}

//реализация абстрактного класса Book

public class PrintedBook : Book

{

public int pages;

//конктруктор класса

public PrintedBook(string title, string description, string author, string genre, int publicationYear,

string publisher, double cost, int pages) : base(title, description, author, genre, publicationYear, publisher, cost)

{

this.pages = pages;

}

//переопределённый метод ToString()

public override string ToString()

{

String result = "Печатная книга \"" + this.title + "\", " + this.type.author + ", " + this.type.cost + "\n";

return result;

}

}

//класс Notebook, наследуется от Product

public class Notebook : Product

{

public String sheet\_type;

public int pages;

//конструктор класса

public Notebook(string title, string description, string author, string sheet\_type, int pages, double cost)

: base(title, description, ProductCreator.getProductType(author, cost))

{

this.pages = pages;

if (sheet\_type.Equals("В клетку") || sheet\_type.Equals("В линейку") || sheet\_type.Equals("Без разметки")

|| sheet\_type.Equals("Другое"))

{

this.sheet\_type = sheet\_type;

}

else throw new ArgumentException();

}

//переопределённый метод ToString()

public override string ToString()

{

String result = "Блокнот \"" + this.title + "\", " + this.type.author + ", " + this.type.cost + "\n";

return result;

}

}

//класс Pen, наследуется от Product

public class Pen : Product

{

public String color;

public String refill;

//конструктор класса

public Pen(string title, string description, string author, string color, string refill, double cost)

: base(title, description, ProductCreator.getProductType(author, cost))

{

this.color = color;

if (refill.Equals("Шариковая") || refill.Equals("Гелевая") || refill.Equals("Другая"))

{

this.refill = refill;

}

else throw new ArgumentException();

}

//переопределённый метод ToString()

public override string ToString()

{

String result = "Pучка \"" + this.title + "\", " + this.type.cost + "\n";

return result;

}

}

Приложение Ж

Листинг кода. Реализация паттерна Proxy.

//интерфейс IClient (паттерн Proxy)

public interface IClient : User

{

public void addProductToCart(Product product);

public void deleteProductFromCart(Product product);

public void editProductNumberInCart(Product product, int k);

public void addProductToFavorites(Product product);

public void deleteProductFromFavorites(Product product);

public void placeOrder(int card, String way);

public void deleteOrderFromOrderList(Order order);

public void deleteAccount();

public Cart getCart();

public FavoritesList getFavoritesList();

public OrderList getOrderList();

public RealClient getRealClient();

}

//класс RealClient (паттерн Proxy)

public class RealClient : IClient

{

protected int userID;

protected String fullName;

protected String email;

protected String password;

protected Mediator mediator;

protected String phoneNumber;

//конструктор класса

public RealClient(Mediator mediator, String fullname, String email, String password, String phone)

{

this.mediator = mediator;

this.userID = Globals.getInstance().getUserID();

this.fullName = fullname;

if (BookShopValidate.validateEmail(email))

{

this.email = email;

}

else throw new Exception();

if (BookShopValidate.validatePassword(password))

{

this.password = password;

}

else throw new Exception();

Globals.getInstance().getUsersList().Add(this);

this.phoneNumber = phone;

this.cart = new Cart(mediator);

this.favoritesList = new FavoritesList(mediator);

this.orderList = new OrderList(mediator);

}

//у каждого Client свои корзина, список избранного и список заказов

public Cart cart { get; private set; }

public OrderList orderList { get; private set; }

public FavoritesList favoritesList { get; private set; }

public void addProductToCart(Product product)

{

cart.addProduct(product);

}

public void deleteProductFromCart(Product product)

{

cart.deleteProduct(product);

}

public void editProductNumberInCart(Product product, int k)

{

cart.editProductNumber(product, k);

}

public void addProductToFavorites(Product product)

{

favoritesList.addProduct(product);

}

public void deleteProductFromFavorites(Product product)

{

favoritesList.deleteProduct(product);

}

public void placeOrder(int card, String way)

{

cart.placeOrder(card, way);

}

public void deleteOrderFromOrderList(Order order)

{

orderList.deleteOrder(order);

}

//удаление собственного аккаунта

public void deleteAccount()

{

Globals.getInstance().getUsersList().Remove(this);

}

public Cart getCart()

{

return cart;

}

public FavoritesList getFavoritesList()

{

return favoritesList;

}

public OrderList getOrderList()

{

return orderList;

}

public int getID()

{

return userID;

}

public String getEmail()

{

return email;

}

public String getPassword()

{

return password;

}

public virtual String getFullName()

{

return fullName;

}

public RealClient getRealClient()

{

return this;

}

}

//класс ProxyClient (аналогичен классу RealClient, за исключением дополнительного поля realCli-ent и метода getRealClient())

public class ProxyClient : IClient

{

//ссылка на реальный объект

private RealClient realClient;

protected int userID;

protected String fullName;

protected String email;

protected String password;

protected Mediator mediator;

protected String phoneNumber;

//конструктор класса

public ProxyClient(Mediator mediator, String fullname, String email, String password, String phone)

{

this.mediator = mediator;

this.userID = Globals.getInstance().getUserID();

this.fullName = fullname;

if (BookShopValidate.validateEmail(email))

{

this.email = email;

}

else throw new Exception();

if (BookShopValidate.validatePassword(password))

{

this.password = password;

}

else throw new Exception();

this.phoneNumber = phone;

this.cart = new Cart(mediator);

this.favoritesList = new FavoritesList(mediator);

this.orderList = new OrderList(mediator);

}

//метод, возвращающий реальный объект

public RealClient getRealClient()

{

if (realClient == null)

{

realClient = new RealClient(mediator, fullName, email, password, phoneNumber);

}

return realClient;

}

}

Приложение К

Листинг кода. Реализация паттерна State.

//абстрактный класс OrderState (паттерн State)

public abstract class OrderState

{

public abstract void cancel(Order order);

public abstract void deliver(Order order);

public abstract void place(Order order);

}

//класс PlacedOrder (паттерн State)

public class PlacedOrder : OrderState

{

//изменение состояния заказа на "отменен"

public override void cancel(Order order)

{

order.state = new CancelledOrder();

Console.WriteLine("Заказ был отменен.");

}

//изменение состояния заказа на "доставлен"

public override void deliver(Order order)

{

order.state = new DeliveredOrder();

Console.WriteLine("Заказ был доставлен.");

}

//изменение состояния заказа на "оформлен"

public override void place(Order order)

{

Console.WriteLine("Заказ уже оформлен.");

}

}

//класс CancelledOrder (паттерн State)

public class CancelledOrder : OrderState

{

//изменение состояния заказа на "отменен"

public override void cancel(Order order)

{

Console.WriteLine("Заказ уже был отменен.");

}

//изменение состояния заказа на "доставлен"

public override void deliver(Order order)

{

Console.WriteLine("Заказ был отменен, поэтому его невозможно доставить.");

}

//изменение состояния заказа на "оформлен"

public override void place(Order order)

{

order.state = new PlacedOrder();

Console.WriteLine("Заказ был отменен и снова оформлен.");

}

}

//класс DeliveredOrder (паттерн State)

public class DeliveredOrder : OrderState

{

//изменение состояния заказа на "отменен"

public override void cancel(Order order)

{

Console.WriteLine("Заказ уже доставлен и его нельзя отменить.");

}

//изменение состояния заказа на "доставлен"

public override void deliver(Order order)

{

Console.WriteLine("Заказ уже доставлен.");

}

//изменение состояния заказа на "оформлен"

public override void place(Order order)

{

Console.WriteLine("Заказ уже доставлен и его нельзя оформить снова.");

}

}

//класс Order

public class Order

{

//состояние заказа

public OrderState state { get; set; }

protected int orderID;

public List<Product> productList;

public double totalPrice;

public DateTime orderTime;

public DateTime deliveryTime;

public int cardNumber;

public String gettingWay;

//конструктор класса

public Order(List<Product> productList, int card, String way)

{

this.state = new PlacedOrder();

this.orderID = Globals.getInstance().getOrderID();

this.productList = productList;

this.cardNumber = card;

//дата заказа равна текущей дате

orderTime = DateTime.Now.Date;

//дата доставки заказа равна дате через 3 дня

deliveryTime = DateTime.Today.AddDays(3).Date;

this.gettingWay = way;

//вычисление полной стоимости заказа

totalPrice = 0;

for (int i = 0; i < productList.Count; i++)

{

this.totalPrice += productList[i].type.cost;

}

}

//паттерн State, изменение состояния на "отменен"

public void cancel()

{

this.state.cancel(this);

}

//паттерн State, изменение состояния на "доставлен"

public void deliver()

{

this.state.deliver(this);

}

//паттерн State, изменение состояния на "оформлен"

public void place()

{

this.state.place(this);

}

//переопределённый метод ToString()

override public String ToString()

{

String result = "Заказ " + orderID.ToString() + "\n";

for (int i = 0; i < productList.Count; i++)

{

result += productList[i].ToString() + " \n";

}

return result;

}

}

Приложение Л

Листинг кода. Реализация паттерна Strategy.

//интерфейс IFindInCatalog (паттерн Strategy). Есть единственное поле со списком найденных товаров.

public interface IFindInCatalog

{

public List<Product> findInCatalog(String key);

}

//класс findByAuthor (поиск по автору, паттерн Strategy)

public class FindInCatalogByAuthor : IFindInCatalog

{

public List<Product> findInCatalog(String author)

{

List<Product> result = new List<Product>();

for (int i = 0; i < Globals.getInstance().getProductCatalog().getCatalog().Count; i++)

{

if (Globals.getInstance().getProductCatalog().getCatalog()[i].type.author.Equals(author))

{

result.Add(Globals.getInstance().getProductCatalog().getCatalog()[i]);

}

}

return result;

}

}

//класс findByAuthor (поиск по названию, паттерн Strategy)

public class FindInCatalogByTitle : IFindInCatalog

{

public List<Product> findInCatalog(String title)

{

List<Product> result = new List<Product>();

for (int i = 0; i < Globals.getInstance().getProductCatalog().getCatalog().Count; i++)

{

if (Globals.getInstance().getProductCatalog().getCatalog()[i].title.Equals(title))

{

result.Add(Globals.getInstance().getProductCatalog().getCatalog()[i]);

}

}

return result;

}

}

//класс ProductCatalog (паттерны Singleton и Strategy). Есть поле findInCatalog, которое возвращает один из классов реализации стра-тегии поиска по каталогу. Есть метод find(), которое реализует поиск товара по ключу – названию или автору.

public class ProductCatalog

{

//приватный статичный экземпляр класса

private static ProductCatalog instance;

//ссылка на реализацию интерфейса Стратегии

public IFindInCatalog findInCatalog { private get; set; }

//приватный конструктор класса

private ProductCatalog()

{

productCatalog = new List<Product>();

}

//метод создания объектов класса Product (паттерн Abstract Factory)

public Product createProduct(ProductCreator productCreator, params object[] args)

{

Product product = productCreator.create(args);

productCatalog.Add(product);

return product;

}

//метод getInstance

public static ProductCatalog getInstance()

{

if (instance == null) //если объект еще не создан

{

instance = new ProductCatalog(); //создать новый объект

}

return instance; // вернуть ранее созданный объект

}

//список книг в каталоге

private List<Product> productCatalog;

public List<Product> getCatalog()

{

return productCatalog;

}

//метод поиска продуктов по ключу (паттерн Strategy)

public List<Product> find(String key)

{

return findInCatalog.findInCatalog(key);

}

//метод сортировки товаров по цене

public List<Product> sortProductsByCost()

{

List<Product> result = new List<Product>(productCatalog);

for (int i = 0; i < result.Count - 1; i++)

{

for (int j = i + 1; j < result.Count; j++)

{

Product product1 = result[i];

Product product2 = result[j];

if (product1.type.cost > product2.type.cost)

{

Product temp = result[i];

result[i] = result[j];

result[j] = temp;

}

}

}

return result;

}

}

Приложение М

Листинг кода. Реализация паттерна Flyweight.

//абстрактный класс Creator (паттерны Factory Method, Abstract Factory, Flyweight)

public abstract class ProductCreator

{

public static List<ProductType> types = new List<ProductType>();

public static ProductType getProductType(string author, double cost)

{

ProductType type = null;

for (int i = 0; i < types.Count; i++)

{

if (types[i].author.Equals(author) && types[i].cost.Equals(cost))

{

type = types[i];

break;

}

}

if (type == null)

{

type = new ProductType(author, cost);

types.Add(type);

}

return type;

}

public ProductCreator() { }

// фабричный метод

abstract public Product create(params object[] args);

}

//класс ProductType (паттерн Flyweight)

public class ProductType

{

public string author;

public double cost;

public ProductType(string author, double cost)

{

this.author = author;

this.cost = cost;

}

}

//абстрактный класс Product (паттерны Factory Method, Flyweight)

public abstract class Product

{

protected int productID;

public String title;

public String description;

public ProductType type;

public Product(string title, string description, ProductType type)

{

this.productID = Globals.getInstance().getProductID();

this.title = title;

this.description = description;

this.type = type;

}

//переопределённый метод ToString()

public override abstract string ToString();

}