МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ ДНР

ГВУЗ «ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

на тему:

«Проектирование системы регистрации и обслуживания пользователей Интернет»

по курсу

«Архитектура и проектирование программного обеспечения»

|  |  |
| --- | --- |
| Руководители:  Ищенко А. П. \_\_\_\_\_\_\_\_\_\_\_\_\_\_  Сереженко О. А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Выполнил:  ст. 3 курса, группы ПИ-16а  Михайлов Б.В. \_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

ДОНЕЦК – 2019

РЕФЕРАТ

Пояснительная записка к курсовому проекту содержит: 92 страницы, 35 рисунков, 2 приложения.

В курсовом проекте рассмотрен процесс проектирования программного обеспечения для системы регистрации и обслуживания пользователей Интернет, а именно разработан Интернет-магазин. Проведен анализ предметной области и сформированы требования к разрабатываемой программной системе. Исходя из требований (атрибутов качества), выбрана архитектура системы с применением системных паттернов. Сделан обзор методологий разработки программного обеспечения, языков описания архитектур программных систем и применяемых видов. Выполнены этапы проектирования системы, включая разработку в CASE-среде различных моделей на языке UML.

С учётом типа приложения выбраны инструментальные средства разработки программного обеспечения Microsoft Visual Studio 2017 C#. При разложении системы на объекты использованы паттерны проектирования. Создано программное обеспечение, обладающее следующими функциональными характеристиками: регистрация пользователей, авторизация пользователь, просмотр всех товаров, фильтр товаров по категориям, просмотр подробной информации о выбранном товаре, сортировка по цене в порядке убывания и возрастания, добавление товаров в корзину, просмотр корзины, удаление товаров из корзины, оформление заказа.

Область применения системы: (сфера торговли).

ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ «ИНТЕРНЕТ-МАГАЗИН», ПРОГРАММНАЯ СИСТЕМА, АРХИТЕКТУРА, МОДЕЛИ UML, ПАТЕРНЫ ПРОЕКТИРОВАНИЯ, ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5

1. АНАЛИЗ ОБЪЕКТА АВТОМАТИЗАЦИИ 6

1.1 Назначение системы и задачи автоматизации 6

1.2 Формирование и анализ требований 7

1.3 Задачи архитектуры программного обеспечения 9

2. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ СИСТЕМЫ 11

2.1 Анализ ключевых вопросов проектирования 11

2.2 Стратегии и методы проектирования ПО. 13

2.3 Определение целей архитектуры 14

2.4 Выявление основных сценариев 15

2.5 Тип приложения 17

2.6 Определение ограничений развёртывания 17

2.7 Выбор архитектурного стиля проектирования 19

2.8 Графическое представление архитектуры на языке UML 27

2.8.1 Диаграмма классов 27

2.8.2 Диаграмма компонентов 29

2.8.3 Диаграмма кооперации 30

2.8.4 Диаграмма деятельности 31

2.8.5 Диаграмма последовательности 32

2.8.6 Диаграмма развертывания 33

2.8.7 Диаграмма состояния 34

2.9 Детализация программ с помощью паттернов проектирования. 35

2.9.1 Паттерн Прототип 35

2.9.2 Паттерн Состояение 43

2.9.3 Паттерн Посетитель 53

2.10 Проектирование пользовательского интерфейса. 62

2.11 Выбор инструментальных средств разработки системы 64

2.12 Программная реализация базовых модулей системы 68

2.13 Анализ качества и оценка программного дизайна 72

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МОДУЛЯ ПРОЕКТА 76

ЗАКЛЮЧЕНИЕ ……………………………………………………………………… 77

СПИСОК ЛИТЕРАТУРЫ ………………...………………………………………… 78

ПРИЛОЖЕНИЕ А. Функциональная модель 79

ПРИЛОЖЕНИЕ Б. Фрагменты листинга 80

ВВЕДЕНИЕ

Проектирование играет важную роль в жизненном цикле создания программного обеспечения. Результат процесса проектирования - проект (дизайн), включающий определение архитектуры, компонентов, интерфейсов и других характеристик программной системы или её компонентов, которые в совокупности являются основой для последующего конструирования программного продукта. Современные программные продукты должны обладать необходимой производительностью, иметь хорошее качество и продаваться по доступной цене. На все эти параметры оказывает влияние архитектура программного обеспечения.

В настоящее время наиболее популярным видом виртуальной торговли является интернет-магазин. Интернет-магазин обычно содержит наглядный и красочный каталог предоставляемых товаров, с их достаточным описанием и указанием цены, что позволяет заинтересовать потенциального покупателя, помочь ему сделать выбор и, в итоге, совершить покупку.

Преимущества интернет-магазина:

1. Не требует затрат на аренду площади, найма продавцов и другого персонала;
2. Доступ к виртуальным витринам магазина может получить любой покупатель, не зависимо от места проживания;
3. Срок и стоимость создания интернет-магазина несоизмеримо ниже, чем обычного магазина;
4. Потенциальный покупатель получает исчерпывающую информацию об интересующем его товаре или услуге без помощи посредников (продавца, менеджера);

1. АНАЛИЗ ОБЪЕКТА АВТОМАТИЗАЦИИ
2. Назначение системы и задачи автоматизации

Интернет-магазин - это привычный каждому пользователю интернета web-сайт, представляющий информацию о товарах в удобном структурированном виде.

Интернет-магазины создаются с применением систем управления содержимым сайтов, оснащенных необходимыми модулями. Крупные интернет-магазины работают на специально разработанных или адаптированных типовых системах управления.

Система управления содержимым сайта интернет-магазина может быть коробочным товаром, самостоятельно устанавливаемым на хостинг-площадку, может быть частной разработкой web-студии, ей же обслуживаемой, или может быть программным сервисом, предоставляемым с помесячной оплатой.

С технической точки зрения интернет-магазин - это совокупность web-витрины и торговой системы - фронт-системы и бэк-офиса. Назначение web-витрины:

* представляет интерфейс к базе данных продаваемых товаров;
* работает с виртуальной торговой тележкой;
* оформляет заказы и регистрирует покупателей;
* предоставляет помощь покупателю в онлайн режиме;
* передает информацию в торговую систему;
* обеспечивает безопасность личной информации покупателя.

Далее торговая система осуществляет автоматическую обработку поступающих заказов – резервирует товар на складе, контролирует оплату и доставку товара.

В общем случае основные функции интернет-магазина - это информационное обслуживание покупателя, обработка заказов, проведение платежей, а также сбор и анализ различной статистической информации. Как было упомянуто выше, программный комплекс управления интернет-магазином позволяет формировать и интерфейс с покупателем, и функциональные возможности интернет-магазина, исходя из потребностей компании.

Целью курсового проекта является разработка интернет-магазина по продаже компьютерной техники, который позволит подробно ознакомится с различными товарами, оценить их характеристики, а при необходимости и приобрести. При разработке необходимо чтобы каждый покупатель, независимо от возраста, смог без проблем получить необходимую ему информацию о товаре, а также и при необходимости приобрести его.

На сайте необходимо реализовать систему дисконта, благодаря которой можно приобрести товар со скидкой. Скидка автоматически учитывается при предварительном расчете стоимости.

Также необходимо реализовать интерфейсные элементы для ввода регистрационной информации. Регистрация покупателя производится либо при оформлении заказа, либо при входе в магазин. После выбора товара от покупателя требуется заполнить форму, в которой указывается, каким образом будет осуществлена оплата и доставка. Для защиты персональной информации взаимодействие должно осуществляться по защищенному каналу. По завершению формирования заказа и регистрации вся собранная информация о покупателе поступает из электронной витрины в торговую систему интернет-магазина. В торговой системе осуществляется проверка наличия затребованного товара на складе, инициируется запрос к платежной системе. При отсутствии товара на складе направляется запрос поставщику, а покупателю сообщается о времени задержки.

## Формирование и анализ требований

1. Система должна поддерживать работу 24/7 без перебоев.
2. Система должна обеспечивать наивысшую производительность.
3. Система должна быть оснащена дружественным и понятным для пользователя интерфейсом.
4. Система должна осуществлять диалог с пользователем.
5. Система должна позволить снизить требования к квалификации пользователя системы.
6. Система должна осуществлять сбор статистики о самых просматриваемых товарах.
7. Система должна осуществлять передачу информации между пользователем и сервером.
8. В системе должен быть реализован личный кабинет, в котором пользователь может редактировать информацию о себе.
9. В системе должна быть возможность формирования статистики о Покупателях (критерии отбора в отчет: все, алфавит, период регистрации, платежи за период, платежи за товар, покупки за период, покупки за товар).
10. Необходимо наличие справки и подсказок по основным разделам системы.
11. При разработке системы должны быть использованы новейшие программные решения, которые не будут требовать слишком много энергоресурсов, и будут оптимальны.
12. Система должна иметь возможность сообщить об ошибке в своей системе или критической ситуации.
13. Система должна автоматически обновлять информацию о товарах в случае каких-либо изменений.
14. Система должна анализировать запросы пользователя (клиента) и на основании этих данных предлагать подобную продукцию.
15. В системе должен быть реализован поиск и фильтрация (сортировка) товаров по определенным свойствам, возможность управления этими свойствами
16. В системе должна быть реализована возможность работы с прайс-листами товаров (загрузка) в табличном формате xls, csv, txt
17. В системе должна быть реализована поддержка различных систем оплаты, как оффлайновых, так и электронных
18. В системе должна быть реализована система оповещения пользователей о событиях – подтверждение заказа, появление товара, проведение акции и т. д
19. В системе должна быть реализована удобная панель администратора, которая бы давала возможность владельцу Интернет-магазина осуществлять эффективное управление всем процессом работы сайта
20. В системе должна быть реализована возможность выбора различных способов и условий доставки товара, приобретенного покупателем;
21. Система должна иметь возможность принятия отзывов и предложений от клиентов интернет-магазина и отправлять их на сервер администраторам
22. Все пользовательские данные должны быть сериализованы и зашифрованы.
23. Система должна иметь интуитивно понятный интерфейс.
24. Система должна предоставлять пользователю возможность добавления товара в корзину.

## Задачи архитектуры программного обеспечения

1. Улучшение и повышение продуктивности процессов (если у заказчика есть обычный магазин по продаже компьютерной техники, он ожидает автоматизацию продажи и оформления покупок, а также увеличение объема продаж за счет расширения аудитории покупателей);
2. Уменьшение затрат (на обслуживание интернет-магазина необходимо меньше сотрудников, нет необходимости арендовать помещение);
3. Улучшение операционной деятельности (рабочее время сотрудников не будет тратиться на объяснение покупателям достоинств той или иной модели, объяснения, как настроить телефон, и пр.);
4. Повышение эффективности управления (так как обработка всей информации будет производиться в единой системе, упрощается получение отчетной информации, а также контроль за финансовыми потоками магазина);
5. Уменьшение рисков (например, исключается возможность кражи товара покупателями);
6. Повышение продуктивности работы пользователей — один сотрудник сможет совмещать различные функции за счет автоматизации процессов работы магазина;
7. Повышение возможности и прозрачности взаимодействия (интеграция с бухгалтерскими программами позволит автоматизировать расчет прибыли, налогов, учет товаров и пр.);
8. Улучшение характеристик безопасности (использование защищенных протоколов обмена данными позволит снизить риск кражи информации и взлома системы злоумышленниками).
9. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ СИСТЕМЫ
   1. Анализ ключевых вопросов проектирования

При проектировании ПО необходимо определиться со следующими ключевыми вопросами, которые оказывают существенное влияние на архитектуру ПО:

1. Параллелизм. Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (воспринимаемых пользователем как одновременные).

Основной сложностью при проектировании параллельных программ является обеспечение правильной последовательности взаимодействия между различными вычислительными процессами, а также координация ресурсов, распределяемых между процессами.

В разрабатываемой системе можно выделить следующие параллельные процессы:

* формирование и выдача страниц по запросам пользователей (обеспечивается на уровне архитектуры web-сервера);
* выдача запросов базой данных (обеспечивается на уровне архитектуры базы данных).

С другой стороны, ряд действий нельзя выполнять параллельно. Например, при регистрации, когда резервируется уникальный код доступа к учетной записи, нельзя одновременно давать регистрироваться двум пользователям, так как это может вызвать конфликт кодов. Функции обработки заказов также нельзя делать параллельно, так как товар на складе может закончиться при обработке одного из заказов.

1. Асинхронные процессы. Основным принципом для взаимодействия процессов является асинхронный обмен сообщениями. В случае с графическим интерфейсом данный подход является наиболее предпочтительным, так как работа графического интерфейса основана на сообщениях.

Для web-приложений выделить асинхронные события несколько трудней, так как обработка графической информации производится в браузере клиентов. Но для них также возможно создание асинхронных событий на основе использования специальных возможностей встроенного языка разметки.

Для разрабатываемой системы в качестве асинхронного события можно выделить автоматическое обновление статуса заказов на страницах без их перерисовки.

1. Обеспечение отказоустойчивости. Это одна из задач проектирования, связанная с обеспечением корректной обработки ошибок. В частности, для данной системы можно обеспечить корректную работу даже в случае сбоя путём вывода абоненту сообщения об ошибке и предложением вернуться к предыдущему пункту. Можно выделить следующие методы контроля ошибок:
   * + Исключения: в случае необработанного исключения или другой непредвиденной ситуации информация об этом автоматически отправляется в отдел техподдержки с описанием действий пользователя, которые привели к исключению.
     + Коды ошибок. Обработанные исключения имеют свои коды для увеличения скорости обработки информации.

## Стратегии и методы проектирования ПО.

Существуют различные общие стратегии, помогающие в проведении работ по проектированию. В отличие от общих стратегий, методы проектирования более специфичны и, в основном, предлагают и предоставляют нотации (или наборы нотаций) для использования совместно с этими методами, а также процессы, которым необходимо следовать в рамках используемого метода.

Для проектирования данной системы был выбран метод декомпозиции («разделяй-и-властвуй»). Этот метод, называемый также методом разбиения, возможно, является самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов. Он предполагает такую декомпозицию (разбиение) задачи размера *n* на более мелкие задачи, что на основе решений этих более мелких задач можно легко получить решение исходной задачи. В качестве примеров применений этого метода можно назвать сортировку слиянием или применение деревьев двоичного поиска.

Можно выделить следующие модули:

1. Модуль верификации.
2. Модуль информации по текущему пользователю.
3. Модуль общей информации.
4. Модуль обработки (изменение параметров товаров).
5. Модуль обратной связи (отправка отчёта об ошибках в отдел техподдержки).
6. Модуль связи с техподдержкой.

Стратегией проектирования было выбрано объектно-ориентированное проектирование ввиду того, что именно эта стратегия подходит к выбранному методу (методу разбиения).

## Определение целей архитектуры

Наличие чётких целей поможет сосредоточиться на архитектуре и правильном отборе проблем для решения. Цели архитектуры – это задачи и ограничения, очерчивающие архитектуру и процесс проектирования ПО, определяющие объём работ и помогающие понять, когда пора завершить процесс доработки архитектуры.

1. Начальное определение задач: задачей архитектуры будет являться разработка архитектуры нового приложения по продаже сотовых телефонов через Интернет.
2. Потребителями будут разработчики, тестировщики и другие 1Т-специалисты; архитектура будет представлена заказчику на одобрение, но ему может быть достаточно общей сметы затрат, чтобы утвердить или предложить переработать архитектуру, если предлагаемое решение получится слишком дорогим для него.
3. Основные ограничения, накладываемые на архитектуру: наличие только web-интерфейса; лимитирование бюджета на приобретение сервера для магазина, а также на покупку лицензий для сервера баз данных; определенные требования по пропускной способности сервера (число одновременных запросов, обрабатываемых в минуту, и др.).

## Выявление основных сценариев

Ключевые сценарии – это описание способа взаимодействия между разрабатываемой системой и одним или более действующим лицом (либо её пользователями, либо другими системами). Главной целью при продумывании архитектуры системы должно быть выявление нескольких возможных ключевых сценариев, что поможет в принятии решений по архитектуре, причём основная задача состоит в нахождении баланса между требованиями, предъявляемыми заказчиком.

Заинтересованными лицами проекта являются клиенты интернет-магазина(основные пользователи системы), не зарегистрированные пользователи(проходят регистрацию и становятся клиентами), а также администраторы (осуществляют управление интернет-магазином)ю

Клиенты могут оценивать работу интернет-магазина, отправлять отзывы о работе системы, предложения по улучшению системы. Также клиент может просматривать список всех товаров, которые есть на в наличии, добавлять товар в корзину, удалять товар из корзины, очищать корзину, оформлять заказ.

Администраторы осуществляют управление интернет магазином. Администратор может добавлять новые группы товаров, новые товары, редактировать информацию о товарах, назначать скидку на товар, назначать скидку определенному клиенту, просмотреть количество продаж за определенный период, просматривать список самых продаваемых товаров.

На рисунках 2.1 – 2.3 представлены диаграммы вариантов использования для различных пользователей системы.

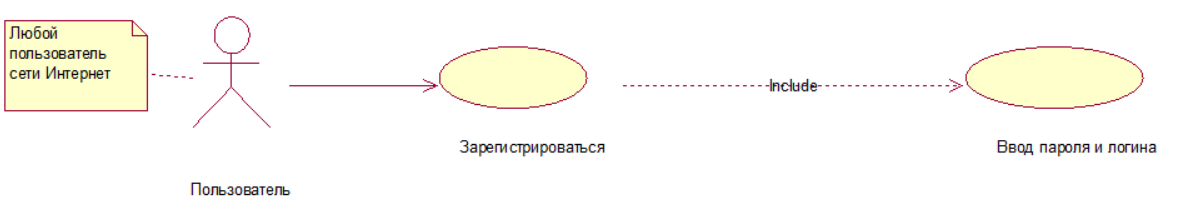


Рисунок 2.1 – Диаграмма вариантов использования для незарегистрированного пользователя

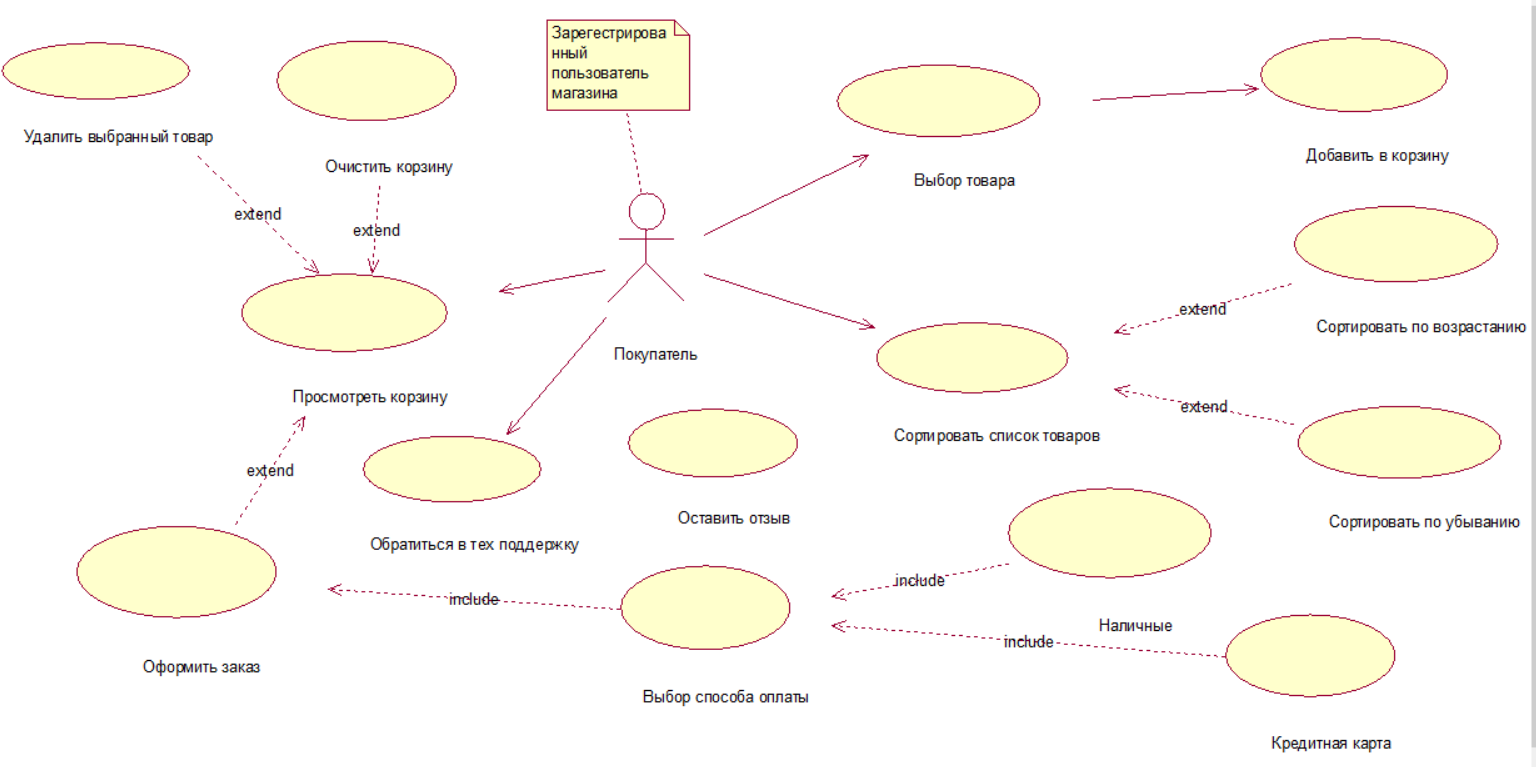


Рисунок 2.2 – Диаграмма вариантов использования для клиента интернет-магазина

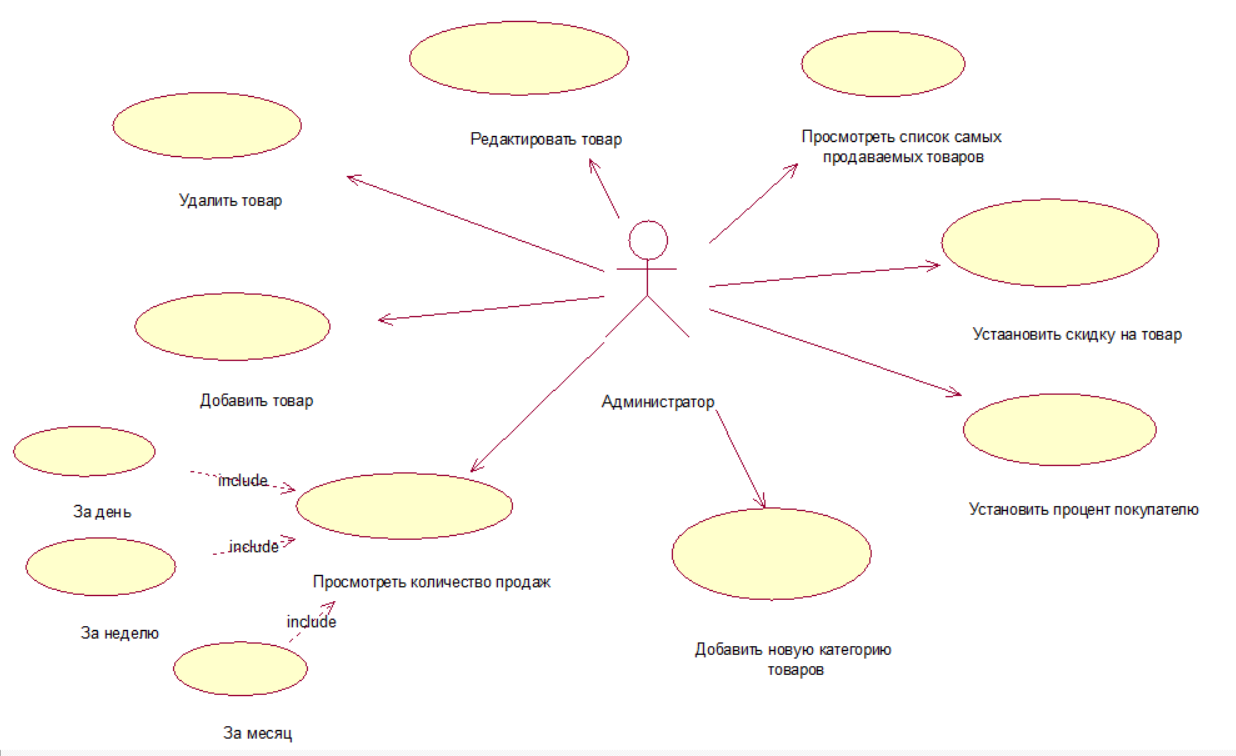


Рисунок 2.3 – Диаграмма вариантов использования для администратора интернет-магазина

## Тип приложения

Ввиду высокой сложности архитектуры и большого количества модулей, а также необходимости графической визуальной части, наиболее целесообразным будет выбор типа приложения Windows Form, так как формы подразумевает отображение не только текстовых данных, которые необходимы в данной системе, но и графической информации в виде графиков, диаграмм, таблиц выпадающих меню.

## Определение ограничений развёртывания

Ограничения по аппаратным требованиям:

1. Архитектура 32(86)bit/64bit.
2. Процессор 1 ГГц IA-32-processor.
3. Видеокарта - Видео адаптер с поддержкой DirectX 9 и WDDM версии 1.0 и старше.
4. Свободное место на жёстком диске: 1 Гб свободного места.
5. Оптический привод: DVD-ROM привод (Только для установки с DVD).

Ограничения по программным требованиям:

1. Требуемая операционная система: Windows 7/8/10.
2. СУБД PostgreSQL.
3. Языковой пакет Microsoft Visual C# 2017.

Ограничения по развёртыванию:

1. Установка отдельного сервера для приложения.
2. Оперативная память (RAM): 512Мб.
3. Web-сервер должен находиться на одном физическом сервере с БД.

Повышенные требования безопасности:

1. Хеширование и сериализация данных.
2. Облачное хранилище должно быть защищено паролем.
3. Вход в личный кабинет.
4. Доступ к пользовательской информации о товарах в корзине.

## Выбор архитектурного стиля проектирования

Архитектурный стиль (парадигма) может рассматриваться как обобщенный шаблон для проектирования архитектуры ПО, опирающийся на некий набор принципов и обеспечивающий абстрактную базу для определённого семейства систем. Каждый стиль определяет набор правил, которые задают типы компонентов, используемых для компоновки системы, и типы отношений, применяемых при компоновке, а также ограничения по способам компоновки и допущения о её семантике.

Архитектура программной системы практически никогда не ограничена лишь одним архитектурным стилем и, как правило, сочетает в себе несколько архитектурных стилей.

На выбор архитектурных стилей оказывает влияние множество факторов, например, способность организации к проектированию и реализации, возможности и опыт разработчиков, а также ограничения инфраструктуры и организации.

Клиент-серверная архитектура - это такая архитектура (способ организации структуры web-приложения), в которой задачи, поставленные перед системой условно делятся между двумя подсистемами:

* клиентом;
* сервер.

Свойства клиент серверной архитектуры:

* Сервер - является "центральной" частью, часто хранит данные и к нему обращаются множество клиентов. Можно сказать, что "сервер может жить без клиентов, а вот клиенты без сервера не могут".
* Клиент - программа, экземпляров которой обычно "больше", чем серверов. Например, к одному веб-серверу (хранящему сайт) может подключаться большое число веб-браузеров. Обычно именно с программой-клиентом работают пользователи (поэтому её так и называют), с другой стороны клиентом сервера может являться и робот.
* В очень нагруженных приложениях серверов тоже может быть очень много (но это значит часто всё равно значит, что клиентов в разы больше)

Клиент-серверная архитектура. Вычислительная (сетевая) клиент-серверная архитектура предполагает, что задания или сетевая нагрузка распределены между поставщиками услуг (сервисов) – серверами и заказчиками услуг – клиентами. Нередко клиенты и серверы взаимодействуют через компьютерную сеть и могут быть как различными физическими устройствами, так и различным ПО.

Данный архитектурный стиль обладает рядом преимуществ. В большинстве случаев становится возможным распределение функций вычислительной системы между несколькими независимыми компьютерами в сети. Это позволяет упростить обслуживание вычислительной системы. В частности, замена, ремонт, модернизация или перемещение сервера не затрагивают клиентов.

Все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов. На сервере проще обеспечить контроль полномочий и разрешать доступ к данным только клиентам с соответствующими правами доступа. Можно объединить различных клиентов. Ресурсы одного сервера могут использовать клиенты с разными аппаратными платформами, ОС и т.п.

Однако имеются недостатки. Неработоспособность сервера может сделать неработоспособной всю вычислительную сеть. Поддержка работы данной системы требует отдельного специалиста (системного администратора). Высока стоимость оборудования; впрочем, можно обойтись арендой места на сервере у специализированных поставщиков данных услуг.

Компонентная архитектура используется при проектировании и разработке систем, когда программные компоненты являются независимыми единицами, обладающими однозначно определенными интерфейсами и зависимостями (связями) и могут собираться и развертываться независимо друг от друга.

Данный подход призван решать задачи использования, разработки и интеграции таких компонентов в целях повторного использования активов (как архитектурных, так и в форме кода). Обычно в приложениях используются компоненты пользовательского интерфейса (их часто называют элементами управления), такие как таблицы и кнопки, а также вспомогательные или служебные компоненты, предоставляющие определённый набор функций, используемых в других компонентах

Данный архитектурный стиль обладает следующими преимуществами:

1. Простотой развертывания – существующие версии компонентов могут заменяться новыми совместимыми версиями, не оказывая влияния на другие компоненты или систему в целом.
2. Небольшой стоимостью – использование компонентов сторонних производителей позволяет уменьшить затраты на разработку и обслуживание.
3. Простотой разработки – для обеспечения заданной функциональности компоненты реализуют широко известные интерфейсы, что позволяет вести разработку различных частей системы без влияния их друг на друга.
4. Возможностью повторного использования компонентов и, как следствие, возможностью распределения затрат на разработку и обслуживание между несколькими приложениями или системами.
5. Возможностью упрощения технической реализации системы – достигается через использование контейнера компонентов и его сервисов; в качестве примера сервисов, предоставляемых контейнером, можно привести активацию компонентов, управление ЖЦ, организацию очереди вызовов методов, обработку событий и транзакции.

Одним из основных недостатков использования компонентной архитектуры является зависимость от поставщика компонентов. Если в компоненте обнаружена ошибка, то на ожидание её исправления может понадобиться время. Другим недостатком является зависимость приложения от структуры компонентов; например, если разработчик компонентов изменяет составляющие одной из частей, то это приводит к необходимости изменения кода в приложении, а, следовательно, к выпуску новой версии программного продукта.

В рамках многоуровневого представления вычислительных систем можно выделить три группы функций, ориентированных на решение различных подзадач:

1. Функции ввода и отображения данных (обеспечивают взаимодействие с пользователем).
2. Прикладные функции, характерные для данной предметной области.
3. Функции управления ресурсами (файловой системой, базой данных и т. д.).



Рисунок 2.4 – Компоненты сетевого приложения

Выполнение этих функций в основном обеспечивается программными средствами, которые можно представить в виде взаимосвязанных компонентов, где:

1. Компонент представления отвечает за пользовательский интерфейс.
2. Прикладной компонент реализует алгоритм решения конкретной задачи.
3. Компонент управления ресурсом обеспечивает доступ к необходимым ресурсам.

Автономная система (компьютер, не подключенный к сети) представляет все эти компоненты как на различных уровнях (ОС, служебное ПО и утилиты, прикладное ПО), так и на уровне приложений (не характерно для современных программ). Так же и сеть – она представляет все эти компоненты, но, в общем случае, распределенные между узлами. Задача сводится к обеспечению сетевого взаимодействия между этими компонентами.

Архитектура «клиент-сервер» определяет общие принципы организации взаимодействия в сети, где имеются серверы, узлы-поставщики некоторых специфичных функций (сервисов) и клиенты, потребители этих функций.

Практические реализации такой архитектуры называются клиент-серверными технологиями. Каждая технология определяет собственные или использует имеющиеся правила взаимодействия между клиентом и сервером, которые называются протоколом обмена (протоколом взаимодействия).

Процесс обработки данных происходит по схеме "клиент - сервер приложений - база данных". Поступивший запрос обрабатывается сервером приложений, который в свою очередь связывается с хранилищем данных и платежной системой, а при наличии подключения к бизнес процессу организации, производит обмен данными с соответствующими системами.

В любой сети (даже одноранговой), построенной на современных сетевых технологиях, присутствуют элементы клиент-серверного взаимодействия, чаще всего на основе двухзвенной архитектуры. Развитие тенденции в клиент-серверных технологиях связано со всё большим использованием распределённых вычислений. Они реализуются на основе модели сервера приложений, где сетевое приложение разделено на две и более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате. В этом случае двухзвенная клиент-серверная архитектура становится трёхзвенной.

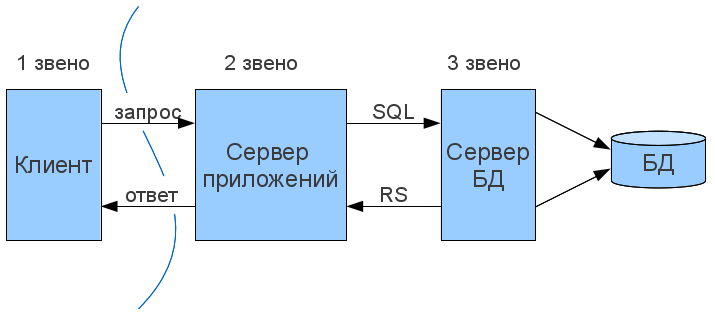


Рисунок 2.5 – Трёхзвенная клиент-серверная архитектура

Как правило, третьим звеном в трёхзвенной архитектуре становится сервер приложений, т. е. компоненты распределяются следующим образом:

1. Представление данных – на стороне клиента.
2. Прикладной компонент – на выделенном сервере приложений (как вариант, выполняющем функции [промежуточного ПО](http://www.4stud.info/networking/lecture6.html)).
3. Управление ресурсами – на сервере БД, который и представляет запрашиваемые данные.

Трёхзвенная архитектура может быть расширена до многозвенной путём выделения дополнительных серверов, каждый из которых будет представлять собственные сервисы и пользоваться услугами прочих серверов разного уровня. Абстрактный пример многозвенной модели приведён на рисунке 2.3.

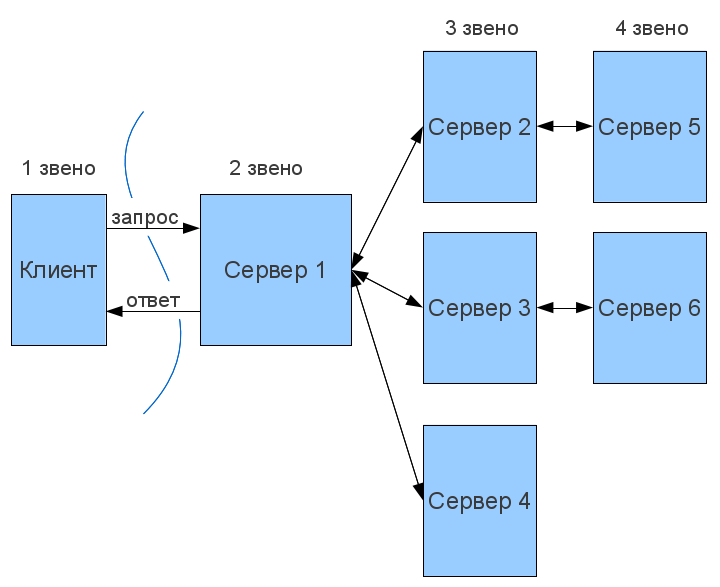


Рисунок 2.6 – Многозвенная клиент-серверная архитектура

Двухзвенная архитектура проще, так как все запросы обслуживаются одним сервером, но именно из-за этого она менее надежна и предъявляет повышенные требования к производительности сервера.

Трёхзвенная архитектура сложнее, но благодаря тому, что функции распределены между серверами второго и третьего уровня, эта архитектура представляет:

1. Высокую степень гибкости и масштабируемости.
2. Высокую безопасность (т. к. защиту можно определить для каждого сервиса или уровня).
3. Высокую производительность (т. к. задачи распределены между серверами).

Основная идея архитектуры «клиент-сервер» состоит в разделении сетевого приложения на несколько компонентов, каждый из которых реализует специфический набор сервисов. Компоненты такого приложения могут выполняться на разных компьютерах, выполняя серверные и/или клиентские функции. Это позволяет повысить надежность, безопасность и производительность сетевых приложений и сети в целом. Именно поэтому и был выбран данный архитектурный стиль проектирования.

## Графическое представление архитектуры на языке UML

* + 1. Диаграмма классов (рис. 2.7):

Вся программа разбита на 6 классов, которые соответствуют модулям:

Класс Control осуществляет управление интернет-магазином. Реализованы функции авторизации в системе, проверки логина и пароля, обновление списка товаров, сортировка товаров по убыванию и возрастанию. Также данный класс предоставляет возможность фильтра списка товаров по категориям. Реализован метод вызова поддержки, вывода списка товаров, вывода подробной информации о товарах и показ корзины пользователя. С некоторыми методами данного класса могут взаимодействовать клиенты, и со всеми – Администраторы.

Класс Product описывает товары, которыми есть в наличии в интернет-магазине. Также данный класс позволяет изменять, удалять, добавлять данные в базу данных интернет-магазина при условии, что пользователь, который производит данные манипуляции – администратор.

Класс Basket является контейнером для объектов класса Product. В этом классе хранится информация о товарах, которые пользователь добавил в корзину, а также количество добавленных товаров и их общая стоимость. Данные класс реализует такие методы: добавление товаров в корзину, очистка корзины, а также оформление заказов.

Класс Order описывает заказы. В этом классе хранится идентификатор пользователя, совершающий заказ, список товаров, тип оплаты, который выбрал клиент, а также время заказ. Данный класс реализует такие методы: показать список заказов за последний день, неделю, месяц, а также установить тип оплаты, отменить заказ и вывести сообщение об успешно совершенном заказе. С данным классом может взаимодействовать как пользователь, так и администратор.

Класс User описывает пользователей, которые есть в системе. В нём реализованы функции получения, удаления, редактирования, добавления данных о пользователях базы. Реализованы функции получения, удаления, редактирования, добавления товаров пользователя. С данным классом взаимодействует непосредственно пользователь, но имеет доступ и администратор. Администратор может только удалить пользователя.

В классе Administrator реализованы функции получения, удаления, редактирования, добавления данных о товарах и типах товаров. Реализованы функции установки дисконта на товар и конкретному клиенту. Реализована функция просмотра статистики по продажам за указанный период. С данным классом взаимодействует непосредственно администратор.

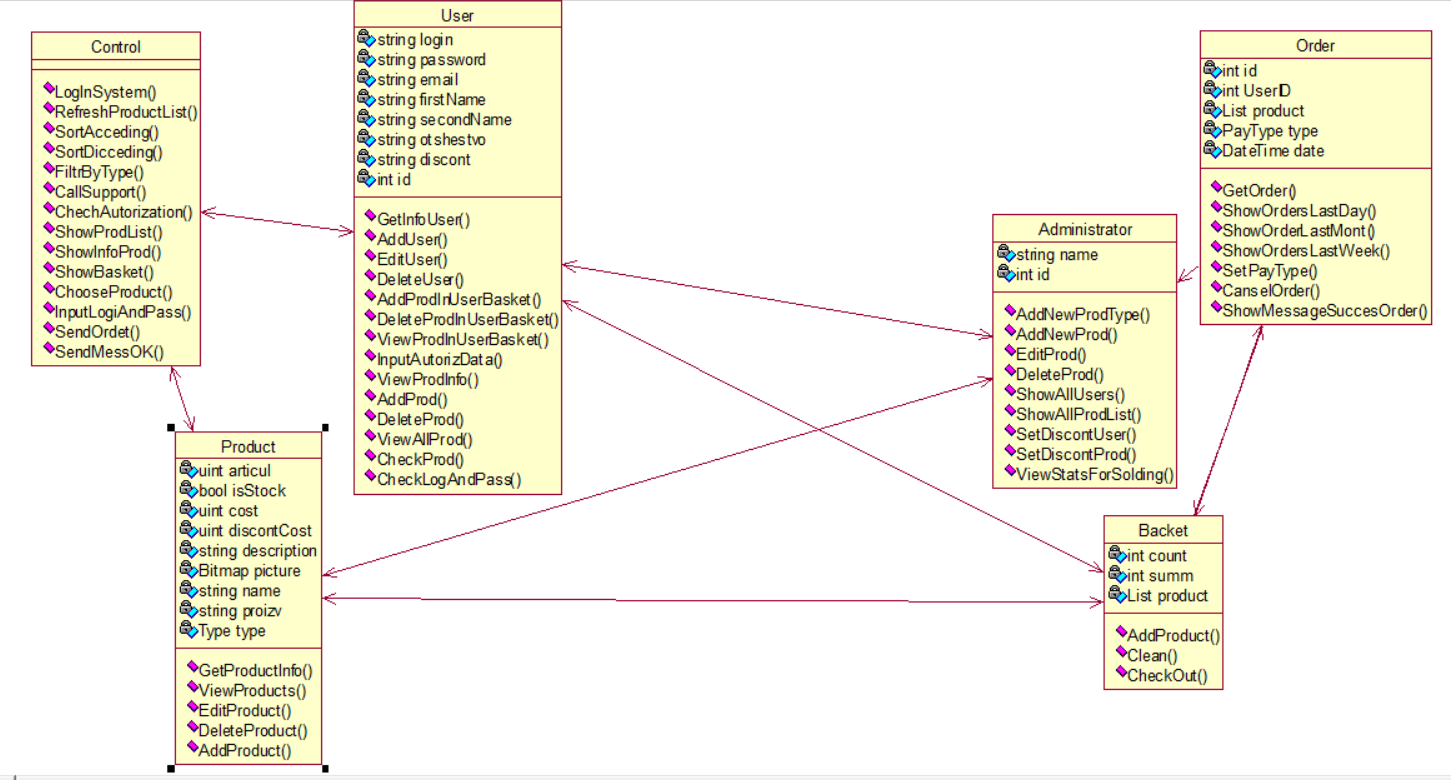


Рисунок 2.7 – Диаграмма классов

* + 1. Диаграмма компонентов (рис. 2.8):

Компонентами системы являются:

1. Web-сервер.
2. Сервер PostgreSQL.
3. Сервер БД.
4. Классы User.cs, Control.cs, Administrator.cs, Order.cs, Product.cs, Basket.cs.

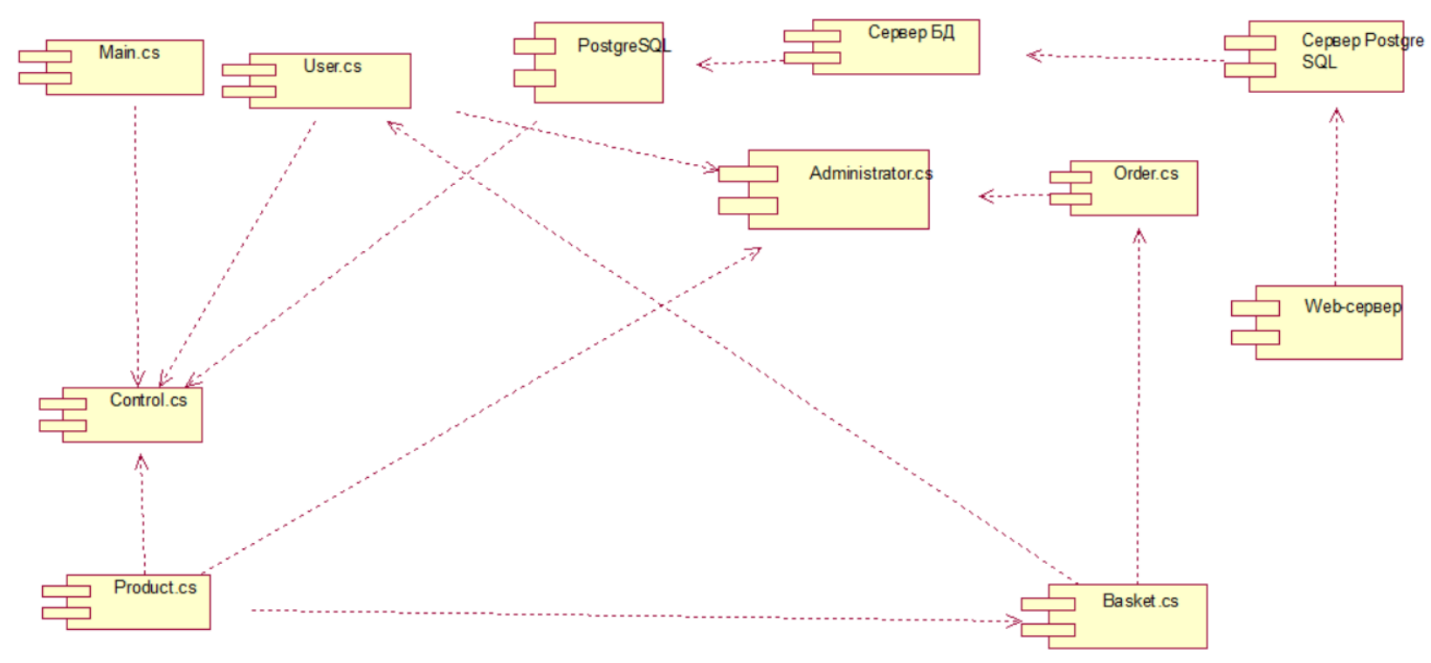
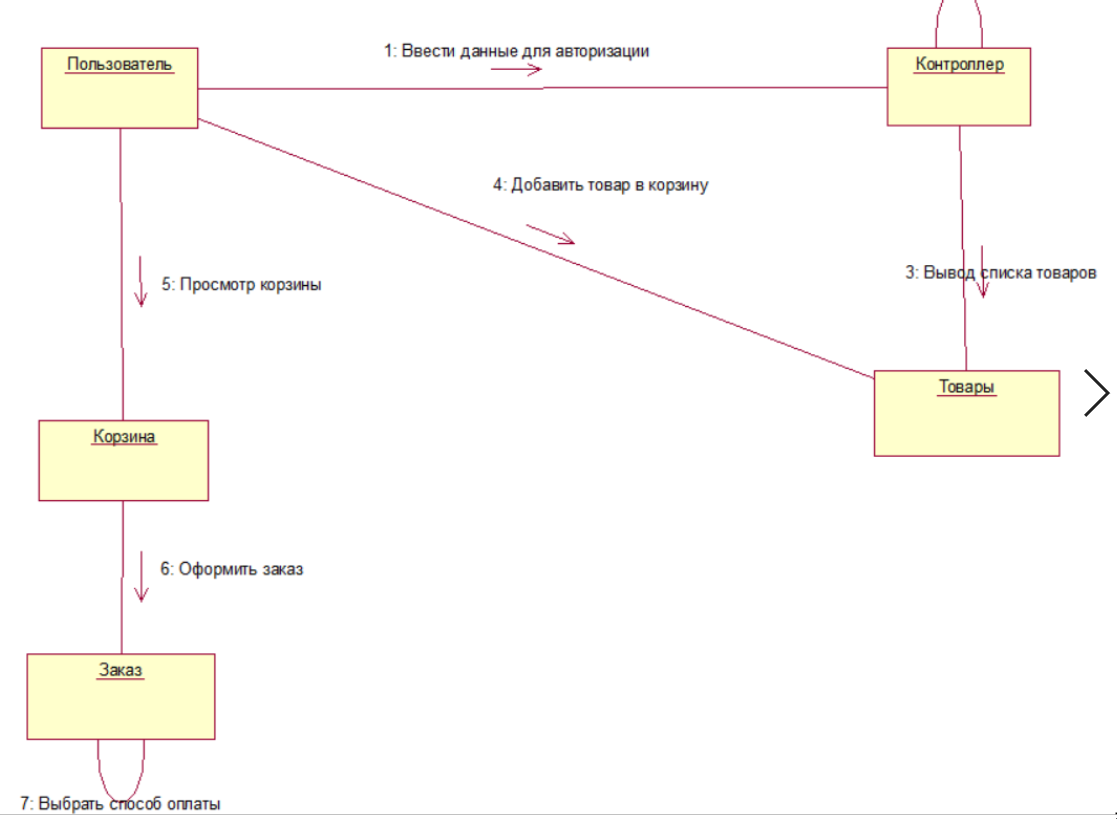


Рисунок 2.8 – Диаграмма компонентов

* + 1. Диаграмма кооперации (рис. 2.9):

Диаграмма кооперации предназначена для описания поведения системы на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый вариант использования. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных вариантов использования или наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

Рисунок 2.9 – Диаграмма кооперации

* + 1. Диаграмма деятельности (рис. 2.10):

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности. Диаграмма деятельности - это своеобразная блок-схема, которая описывает последовательность выполнения операций во времени. Их можно использовать для моделирования динамических аспектов поведения системы. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой операции в предыдущем состоянии.

На рисунке 2.10 представлена диаграмма событий для процесса оформления заказа.

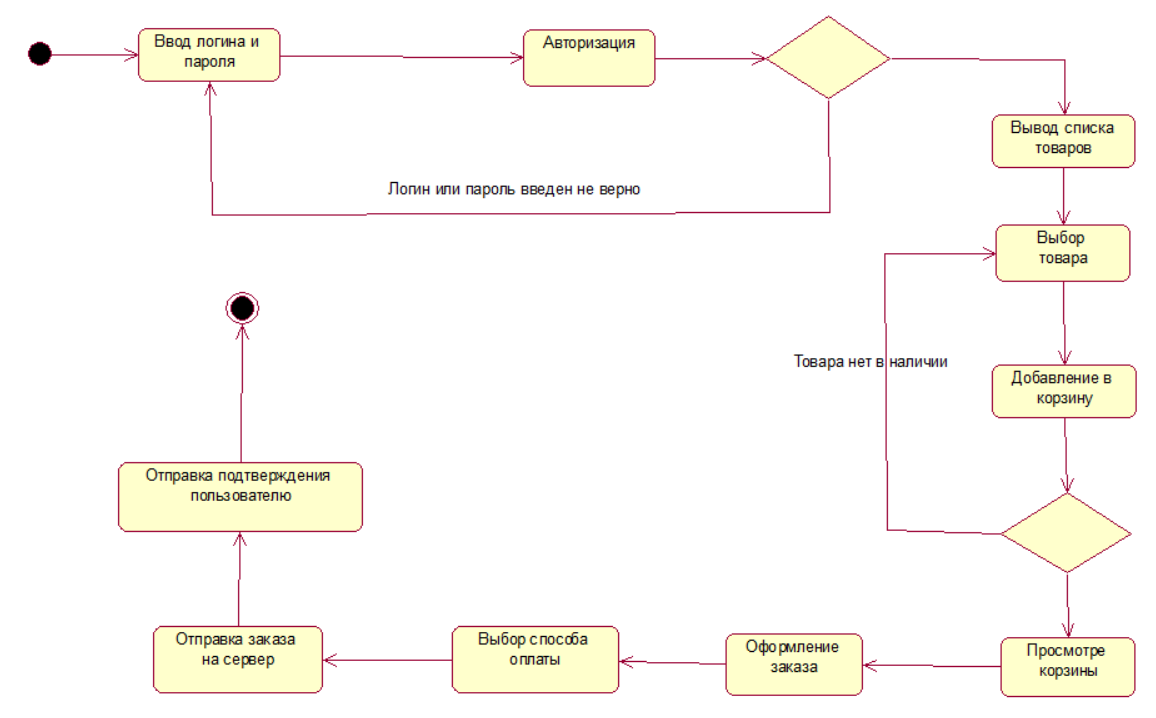


Рисунок 2.10 – Диаграмма деятельности процесса оформление заказа

* + 1. Диаграмма последовательности (рис. 2.11):

Диаграмма последовательности – способ описания поведения системы на основе указания последовательности передаваемых сообщений.

Диаграмма последовательности для прецедента авторизации представлена на рисунке 2.8

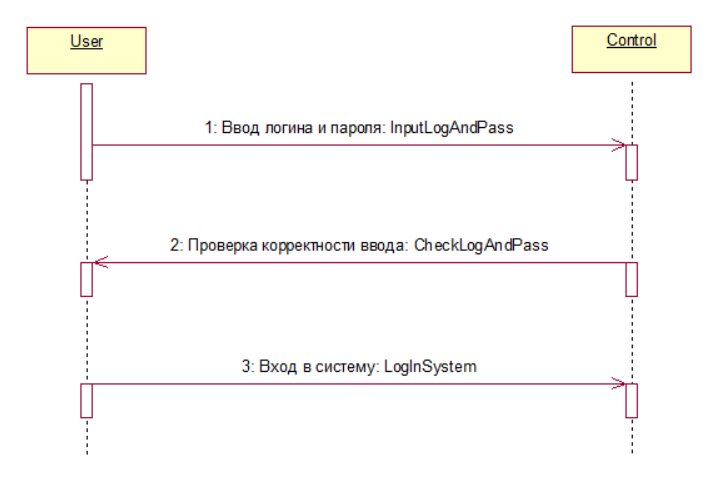


Рисунок 2.11 – Диаграмма последовательности для прецедента авторизации

* + 1. Диаграмма развёртывания (рис. 2.12):

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих лишь на этапе ее исполнения. При этом представляются только компоненты-экземпляры программы, являющиеся исполнимыми файлами или динамическими библиотеками. Те компоненты, которые не используются на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними.

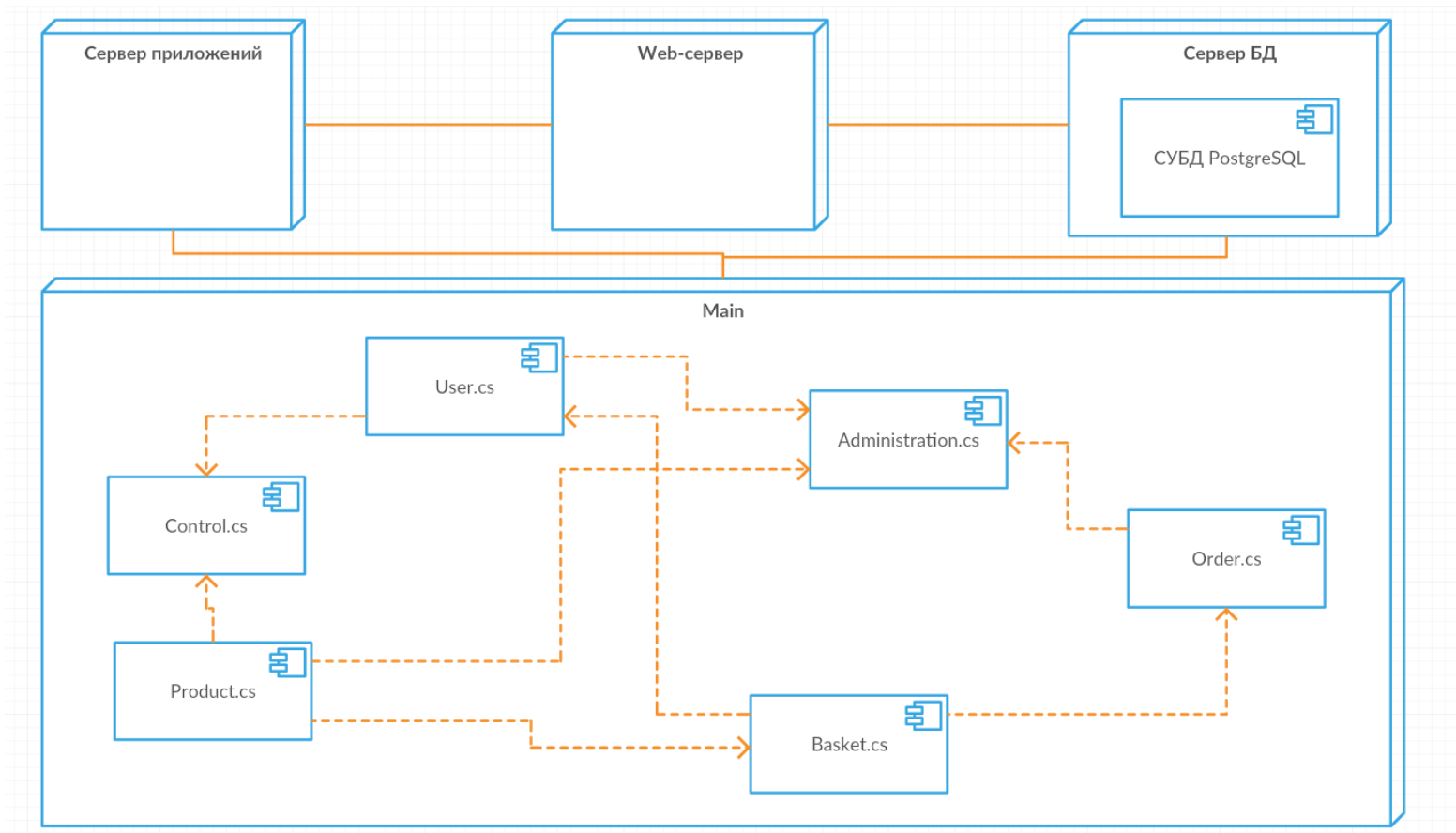


Рисунок 2.12 – Диаграмма развёртывания

* + 1. Диаграмма состояний (рис. 2.13):

Диаграмма состояний показывает, как объект переходит из одного состояния в другое. Очевидно, что диаграммы состояний служат для моделирования динамических аспектов системы.

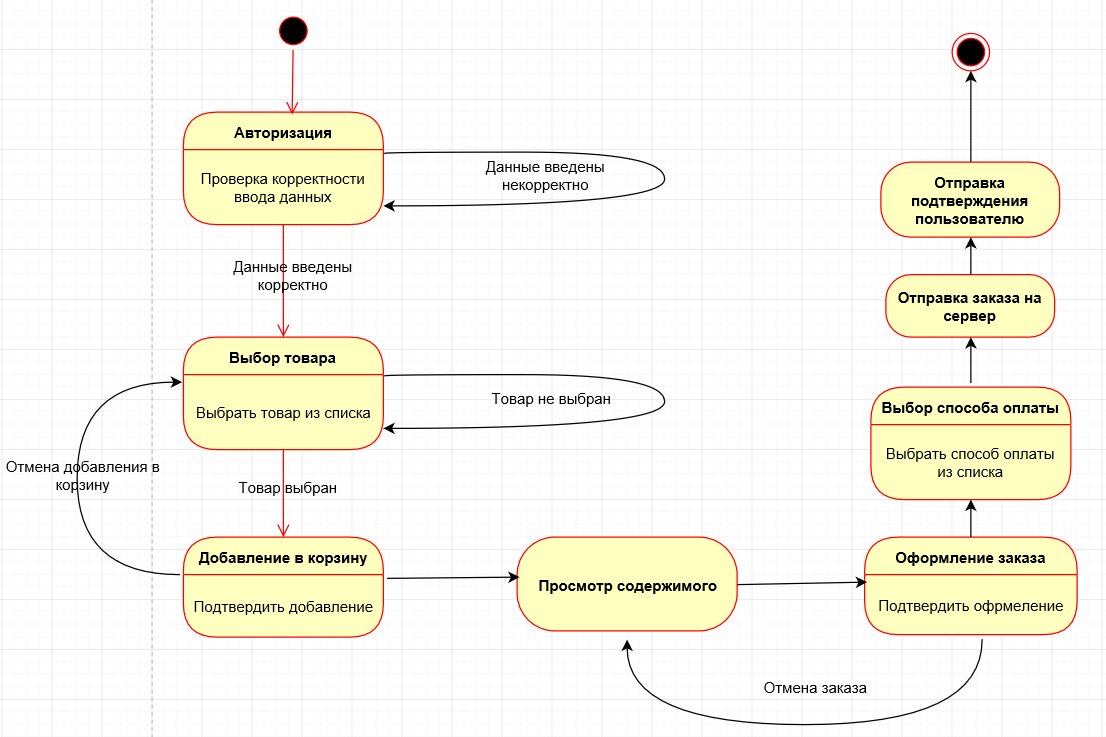


Рисунок 2.13 – Диаграмма состояний для процесса оформление заказа

## Детализация программ с помощью паттернов проектирования.

Паттерн в разработке программного обеспечения – повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Используемые паттерны в системе: прототип, состояние, посетитель.

2.9.1 Паттерн Прототип

Суть паттерна

Прототип – это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Проблема паттерна

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит — недоступной для остального кода программы.

Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь, чтобы перебрать все поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете копировать объекты, зная только их интерфейсы, а не конкретные классы.

Решение

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод clone.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям любого объекта текущего класса.

Объект, который копируют, называется прототипом (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.

В этом случае все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из приготовленного прототипа.

Аналогия из жизни

В промышленном производстве прототипы создаются перед основной партией продуктов для проведения всевозможных испытаний. При этом прототип не участвует в последующем производстве, отыгрывая пассивную роль.

Прототип на производстве не делает копию самого себя, поэтому более близкий пример паттерна — деление клеток. После митозного деления клеток образуются две совершенно идентичные клетки. Оригинальная клетка отыгрывает роль прототипа, принимая активное участие в создании нового объекта.

Структура паттерна Прототип представлена на рисунках 2.14 – 2.15.

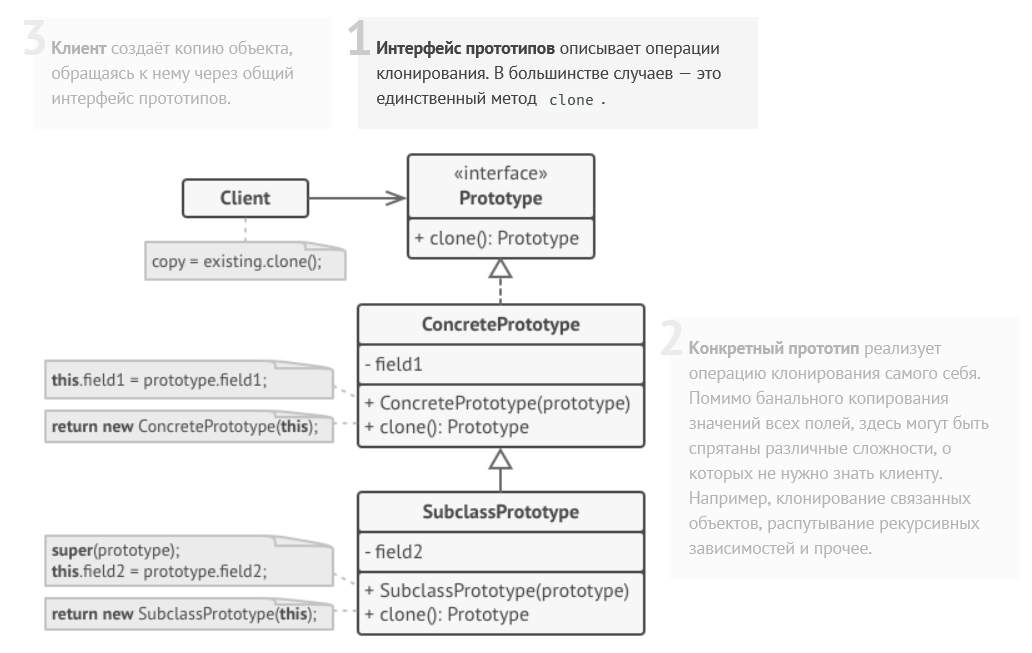


Рисунок 2.14 – Базовая реализация

1. Интерфейс прототипов описывает операции клонирования. В большинстве случаев — это единственный метод clone.
2. Конкретный прототип реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.
3. Клиент создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

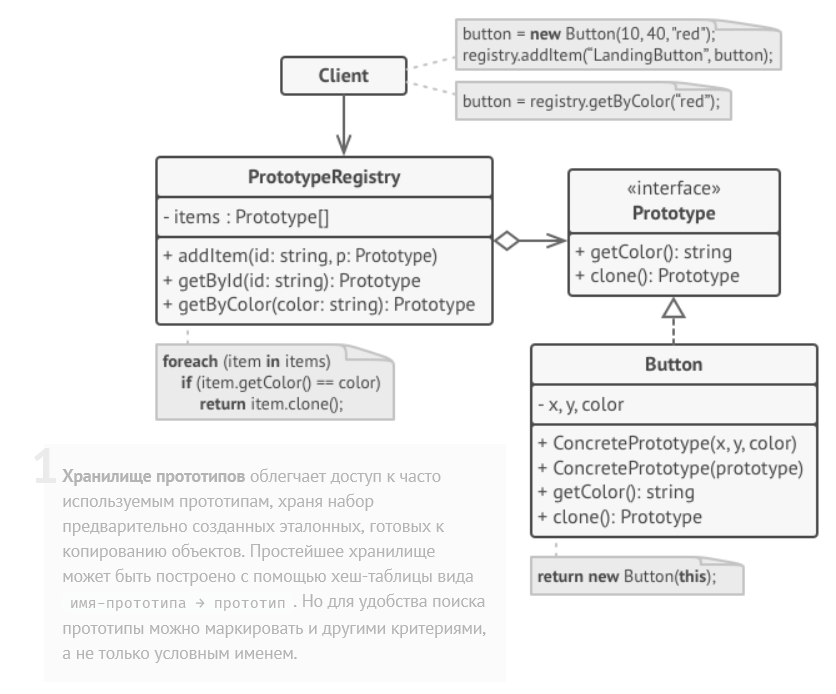


Рисунок 2.15 – Реализация с общим хранилищем прототипов

1. Хранилище прототипов облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида имя-прототипа → прототип. Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.

Псевдокод

В этом примере Прототип позволяет производить точные копии объектов геометрических фигур, не привязываясь к их классам.

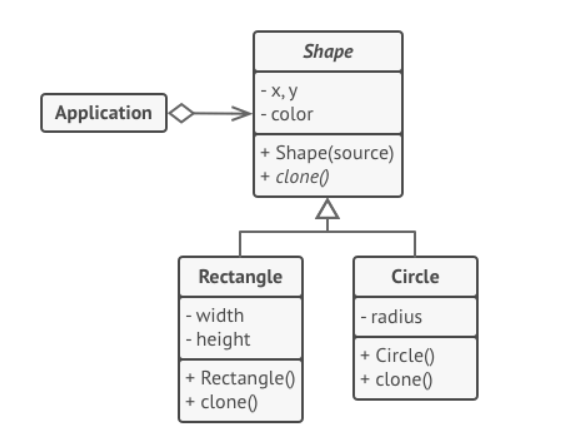


Рисунок 2.16 – Пример клонирования иерархии геометрических фигур

Все фигуры реализуют интерфейс клонирования и предоставляют метод для воспроизводства самой себя. Подклассы используют метод клонирования родителя, а затем копируют собственные поля в получившийся объект.

// Базовый прототип.

abstract class Shape is

field X: int

field Y: int

field color: string

// Обычный конструктор.

constructor Shape() is

// ...

// Конструктор прототипа.

constructor Shape(source: Shape) is

this()

this.X = source.X

this.Y = source.Y

this.color = source.color

// Результатом операции клонирования всегда будет объект из

// иерархии классов Shape.

abstract method clone():Shape

// Конкретный прототип. Метод клонирования создаёт новый объект

// текущего класса, передавая в его конструктор ссылку на

// собственный объект. Благодаря этому операция клонирования

// получается атомарной — пока не выполнится конструктор, нового

// объекта ещё не существует. Но как только конструктор завершит

// работу, мы получим полностью готовый объект-клон, а не пустой

// объект, который нужно ещё заполнить.

class Rectangle extends Shape is

field width: int

field height: int

constructor Rectangle(source: Rectangle) is

// Вызов родительского конструктора нужен, чтобы

// скопировать потенциальные приватные поля, объявленные

// в родительском классе.

super(source)

this.width = source.width

this.height = source.height

method clone():Shape is

return new Rectangle(this)

class Circle extends Shape is

field radius: int

constructor Circle(source: Circle) is

super(source)

this.radius = source.radius

method clone():Shape is

return new Circle(this)

// Где-то в клиентском коде.

class Application is

field shapes: array of Shape

constructor Application() is

Circle circle = new Circle()

circle.width = 10

circle.height = 10

circle.radius = 20

shapes.add(circle)

Circle anotherCircle = circle.clone()

shapes.add(anotherCircle)

// anotherCircle будет содержать точную копию circle.

Rectangle rectangle = new Rectangle()

rectangle.width = 10

rectangle.height = 20

shapes.add(rectangle)

method businessLogic() is

// Плюс Прототипа в том, что вы можете клонировать набор

// объектов, не зная их конкретные классы.

Array shapesCopy = new Array of Shapes.

// Например, мы не знаем, какие конкретно объекты

// находятся внутри массива shapes, так как он объявлен

// с типом Shape. Но благодаря полиморфизму, мы можем

// клонировать все объекты «вслепую». Будет выполнен

// метод clone того класса, которым является этот

// объект.

foreach (s in shapes) do

shapesCopy.add(s.clone())

// Переменная shapesCopy будет содержать точные копии

// элементов массива shapes.

Применимость

Когда ваш код не должен зависеть от классов копируемых объектов.

Такое часто бывает, если ваш код работает с объектами, поданными извне через какой-то общий интерфейс. Вы не можете привязаться к их классам, даже если бы хотели, поскольку их конкретные классы неизвестны.

Паттерн прототип предоставляет клиенту общий интерфейс для работы со всеми прототипами. Клиенту не нужно зависеть от всех классов копируемых объектов, а только от интерфейса клонирования.

Когда вы имеете уйму подклассов, которые отличаются начальными значениями полей. Кто-то мог создать все эти классы, чтобы иметь возможность легко порождать объекты с определённой конфигурацией.

Паттерн прототип предлагает использовать набор прототипов, вместо создания подклассов для описания популярных конфигураций объектов.

Таким образом, вместо порождения объектов из подклассов, вы будете копировать существующие объекты-прототипы, в которых уже настроено внутреннее состояние. Это позволит избежать взрывного роста количества классов в программе и уменьшить её сложность.

Шаги реализации

1. Создайте интерфейс прототипов с единственным методом clone. Если у вас уже есть иерархия продуктов, метод клонирования можно объявить непосредственно в каждом из её классов.
2. Добавьте в классы будущих прототипов альтернативный конструктор, принимающий в качестве аргумента объект текущего класса. Этот конструктор должен скопировать из поданного объекта значения всех полей, объявленных в рамках текущего класса, а затем передать выполнение родительскому конструктору, чтобы тот позаботился о полях, объявленных в суперклассе.

Если ваш язык программирования не поддерживает перегрузку методов, то вам не удастся создать несколько версий конструктора. В этом случае копирование значений можно проводить и в другом методе, специально созданном для этих целей. Конструктор удобнее тем, что позволяет клонировать объект за один вызов.

1. Метод клонирования обычно состоит всего из одной строки: вызова оператора new с конструктором прототипа. Все классы, поддерживающие клонирование, должны явно определить метод clone, чтобы использовать собственный класс с оператором new. В обратном случае результатом клонирования станет объект родительского класса.
2. Опционально, создайте центральное хранилище прототипов. В нём удобно хранить вариации объектов, возможно, даже одного класса, но по-разному настроенных.

Вы можете разместить это хранилище либо в новом фабричном классе, либо в фабричном методе базового класса прототипов. Такой фабричный метод должен на основании входящих аргументов искать в хранилище прототипов подходящий экземпляр, а затем вызывать его метод клонирования и возвращать полученный объект.

Наконец, нужно избавиться от прямых вызовов конструкторов объектов, заменив их вызовами фабричного метода хранилища прототипов.

Преимущества

* Позволяет клонировать объекты, не привязываясь к их конкретным классам.
* Меньше повторяющегося кода инициализации объектов.
* Ускоряет создание объектов.
* Альтернатива созданию подклассов для конструирования объектов.

Недостатки

* Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

2.9.2 Паттерн Состояние

Суть паттерна

Состояние — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

Проблема

Паттерн Состояние невозможно рассматривать в отрыве от концепции машины состояний, также известной как стейт-машина или конечный автомат.

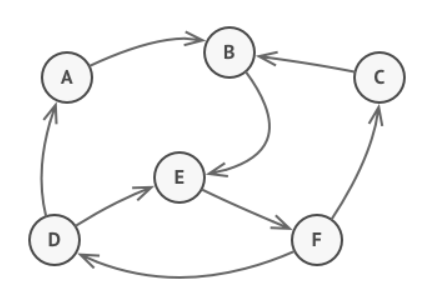


Рисунок 2.17 – Конечный автомат

Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

Такой подход можно применить и к отдельным объектам. Например, объект Документ может принимать три состояния: Черновик, Модерация или Опубликован. В каждом из этих состоянии метод опубликовать будет работать по-разному:

* Из черновика он отправит документ на модерацию.
* Из модерации — в публикацию, но при условии, что это сделал администратор.
* В опубликованном состоянии метод не будет делать ничего.

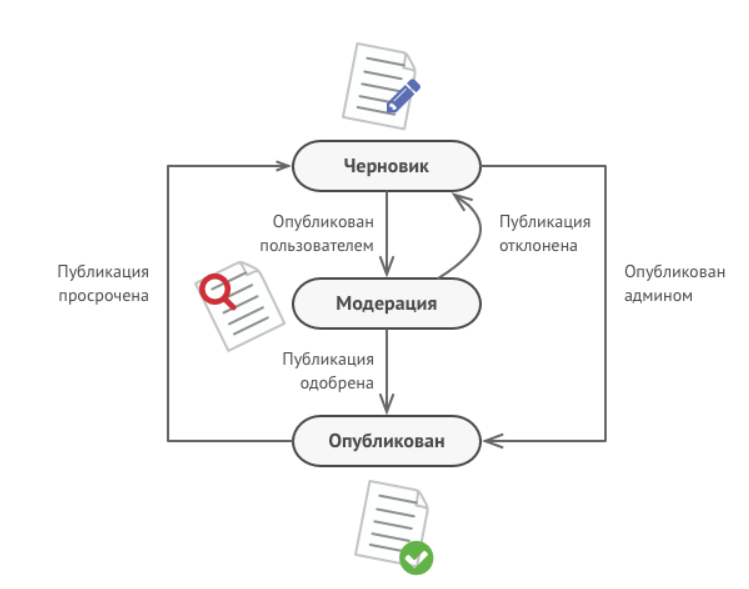


Рисунок 2.18 – Возможные состояния документа и переходы между ними

Машину состояний чаще всего реализуют с помощью множества условных операторов, if либо switch, которые проверяют текущее состояние объекта и выполняют соответствующее поведение.

**class** **Document**

**string** **state**;

*// ...*

**method** publish() {

switch (state) {

"draft":

state = "moderation";

break;

"moderation":

**if** (currentUser.role == 'admin')

state = "published"

break;

"published":

// Do nothing.

break;

}

}

*// ..*

Основная проблема такой машины состояний проявится в том случае, если в Документ добавить ещё десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит вас перепроверять работу всех методов, которые содержат условные операторы машины состояний.

Путаница и нагромождение условий особенно сильно проявляется в старых проектах. Набор возможных состояний бывает трудно предопределить заранее, поэтому они всё время добавляются в процессе эволюции программы. Из-за этого решение, которое выглядело простым и эффективным в самом начале разработки, может впоследствии стать проекцией большого макаронного монстра.

Решение

Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

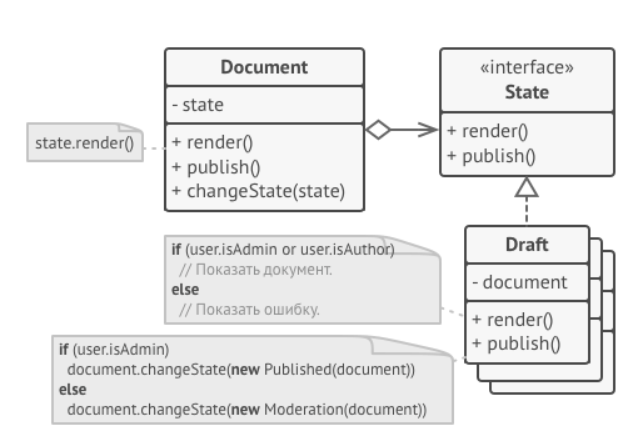


Рисунок 2.19 – Документ делегирует работу своему активному объекту-состоянию

Благодаря тому, что объекты состояний будут иметь общий интерфейс, контекст сможет делегировать работу состоянию, не привязываясь к его классу. Поведение контекста можно будет изменить в любой момент, подключив к нему другой объект-состояние.

Очень важным нюансом, отличающим этот паттерн от Стратегии, является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

Аналогия из жизни

Ваш смартфон ведёт себя по-разному, в зависимости от текущего состояния:

* Когда телефон разблокирован, нажатие кнопок телефона приводит к каким-то действиям.
* Когда телефон заблокирован, нажатие кнопок приводит к экрану разблокировки.
* Когда телефон разряжен, нажатие кнопок приводит к экрану зарядки.

Структура

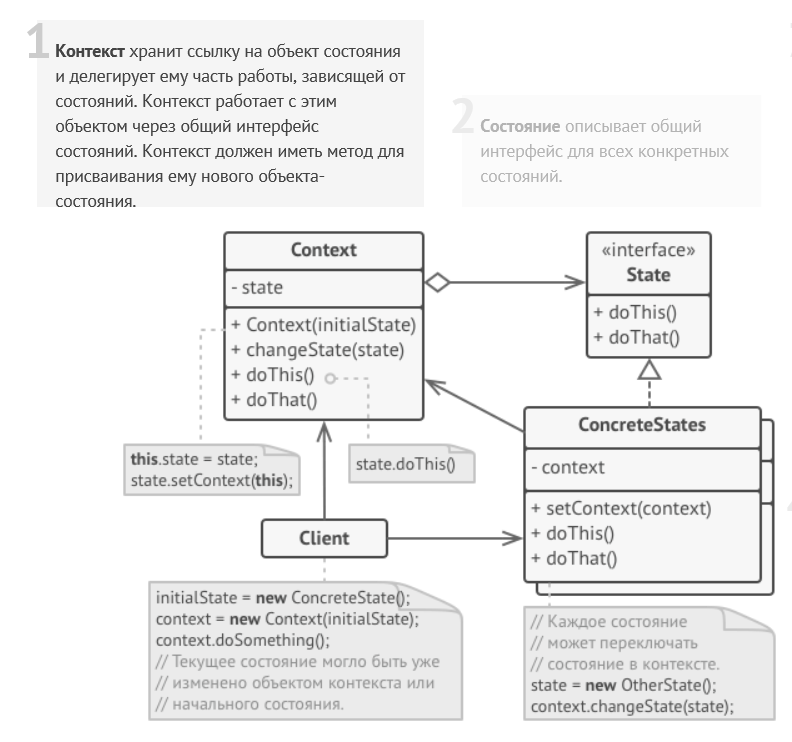


Рисунок 2.20 – Структура паттерна Состояние

1. Контекст хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний. Контекст работает с этим объектом через общий интерфейс состояний. Контекст должен иметь метод для присваивания ему нового объекта-состояния.
2. Состояние описывает общий интерфейс для всех конкретных состояний.
3. Конкретные состояния реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.

Состояние может иметь обратную ссылку на объект контекста. Через неё не только удобно получать из контекста нужную информацию, но и осуществлять смену его состояния.

1. И контекст, и объекты конкретных состояний могут решать, когда и какое следующее состояние будет выбрано. Чтобы переключить состояние, нужно подать другой объект-состояние в контекст.

Псевдокод

В этом примере паттерн Состояние изменяет функциональность одних и тех же элементов управления музыкальным проигрывателем, в зависимости от того, в каком состоянии находится сейчас проигрыватель.

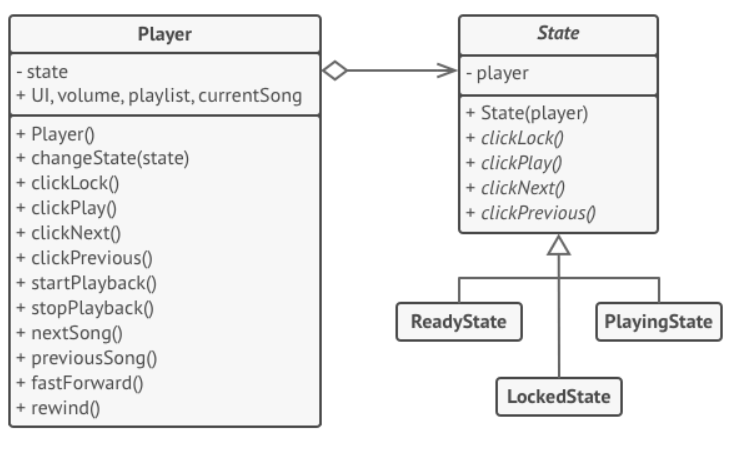


Рисунок 2.21 – Пример изменения поведения проигрывателя с помощью состояний

Объект проигрывателя содержит объект-состояние, которому и делегирует основную работу. Изменяя состояния, можно менять то, как ведут себя элементы управления проигрывателя.

// Общий интерфейс всех состояний.

abstract class State is

protected field player: AudioPlayer

// Контекст передаёт себя в конструктор состояния, чтобы

// состояние могло обращаться к его данным и методам в

// будущем, если потребуется.

constructor State(player) is

this.player = player

abstract method clickLock()

abstract method clickPlay()

abstract method clickNext()

abstract method clickPrevious()

// Конкретные состояния реализуют методы абстрактного состояния

// по-своему.

class LockedState extends State is

// При разблокировке проигрователя с заблокированными

// клавишами он может принять одно из двух состояний.

method clickLock() is

if (player.playing)

player.changeState(new PlayingState(player))

else

player.changeState(new ReadyState(player))

method clickPlay() is

// Ничего не делать.

method clickNext() is

// Ничего не делать.

method clickPrevious() is

// Ничего не делать.

// Конкретные состояния сами могут переводить контекст в другое

// состояние.

class ReadyState extends State is

method clickLock() is

player.changeState(new LockedState(player))

method clickPlay() is

player.startPlayback()

player.changeState(new PlayingState(player))

method clickNext() is

player.nextSong()

method clickPrevious() is

player.previousSong()

class PlayingState extends State is

method clickLock() is

player.changeState(new LockedState(player))

method clickPlay() is

player.stopPlayback()

player.changeState(new ReadyState(player))

method clickNext() is

if (event.doubleclick)

player.nextSong()

else

player.fastForward(5)

method clickPrevious() is

if (event.doubleclick)

player.previous()

else

player.rewind(5)

// Проигрыватель выступает в роли контекста.

class AudioPlayer is

field state: State

field UI, volume, playlist, currentSong

constructor AudioPlayer() is

this.state = new ReadyState(this)

// Контекст заставляет состояние реагировать на

// пользовательский ввод вместо себя. Реакция может быть

// разной, в зависимости от того, какое состояние сейчас

// активно.

UI = new UserInterface()

UI.lockButton.onClick(this.clickLock)

UI.playButton.onClick(this.clickPlay)

UI.nextButton.onClick(this.clickNext)

UI.prevButton.onClick(this.clickPrevious)

// Другие объекты тоже должны иметь возможность заменять

// состояние проигрывателя.

method changeState(state: State) is

this.state = state

// Методы UI будут делегировать работу активному состоянию.

method clickLock() is

state.clickLock()

method clickPlay() is

state.clickPlay()

method clickNext() is

state.clickNext()

method clickPrevious() is

state.clickPrevious()

// Сервисные методы контекста, вызываемые состояниями.

method startPlayback() is

// ...

method stopPlayback() is

// ...

method nextSong() is

// ...

method previousSong() is

// ...

method fastForward(time) is

// ...

method rewind(time) is

// ...

Применимость

Когда у вас есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется.

Паттерн предлагает выделить в собственные классы все поля и методы, связанные с определёнными состояниями. Первоначальный объект будет постоянно ссылаться на один из объектов-состояний, делегируя ему часть своей работы. Для изменения состояния в контекст достаточно будет подставить другой объект-состояние.

Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.

Паттерн предлагает переместить каждую ветку такого условного оператора в собственный класс. Тут же можно поселить и все поля, связанные с данным состоянием.

Когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов.

Паттерн Состояние позволяет реализовать иерархическую машину состояний, базирующуюся на наследовании. Вы можете отнаследовать похожие состояния от одного родительского класса и вынести туда весь дублирующий код.

Шаги реализации

1. Определитесь с классом, который будет играть роль контекста. Это может быть как существующий класс, в котором уже есть зависимость от состояния, так и новый класс, если код состояний размазан по нескольким классам.
2. Создайте общий интерфейс состояний. Он должен описывать методы, общие для всех состояний, обнаруженных в контексте. Заметьте, что не всё поведение контекста нужно переносить в состояние, а только то, которое зависит от состояний.
3. Для каждого фактического состояния создайте класс, реализующий интерфейс состояния. Переместите код, связанный с конкретными состояниями в нужные классы. В конце концов, все методы интерфейса состояния должны быть реализованы во всех классах состояний.

При переносе поведения из контекста вы можете столкнуться с тем, что это поведение зависит от приватных полей или методов контекста, к которым нет доступа из объекта состояния. Существует парочка способов обойти эту проблему.

Самый простой — оставить поведение внутри контекста, вызывая его из объекта состояния. С другой стороны, вы можете сделать классы состояний вложенными в класс контекста, и тогда они получат доступ ко всем приватным частям контекста. Но последний способ доступен только в некоторых языках программирования (например, Java, C#).

1. Создайте в контексте поле для хранения объектов-состояний, а также публичный метод для изменения значения этого поля.
2. Старые методы контекста, в которых находился зависимый от состояния код, замените на вызовы соответствующих методов объекта-состояния.
3. В зависимости от бизнес-логики, разместите код, который переключает состояние контекста либо внутри контекста, либо внутри классов конкретных состояний.

Преимущества

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

Недостатки

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

2.9.3 Паттерн Посетитель

Суть паттерна

Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Проблема

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.



Рисунок 2.22 – Экспорт геоузлов в XML

Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что код классов узлов сейчас очень стабилен, и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.

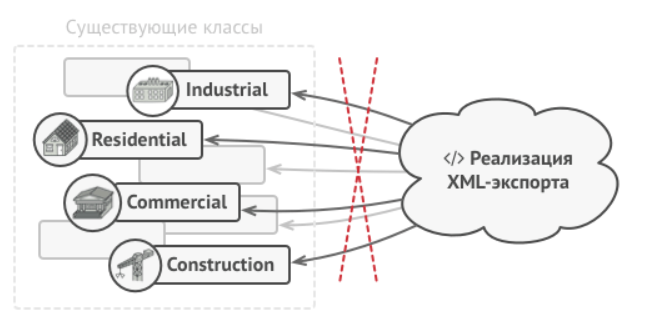


Рисунок 2.23 - Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно.

Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат данных, то эти классы снова пришлось бы менять.

Решение

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта, например:

**class** **ExportVisitor** **implements** Visitor **is**

**method** doForCity(City c) { ... }

**method** doForIndustry(Industry f) { ... }

**method** doForSightSeeing(SightSeeing ss) { ... }

*// ...*

Здесь возникает вопрос: как подавать узлы в объект-посетитель? Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

**foreach** (Node node in graph)

**if** (node instanceof City)

exportVisitor.doForCity((City) node)

**if** (node instanceof Industry)

exportVisitor.doForIndustry((Industry) node)

*// ...*

Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу Node, а не реального класса поданного узла.

Но паттерн Посетитель решает и эту проблему, используя механизм двойной диспетчеризации. Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

// Client code

**foreach** (Node node in graph)

node.accept(exportVisitor)

// City

**class** **City** **is**

**method** accept(Visitor v) **is**

v.doForCity(**this**)

*// ...*

// Industry

**class** **Industry** **is**

**method** accept(Visitor v) **is**

v.doForIndustry(**this**)

*// ...*

Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение позволит применять к объектам узлов и другие поведения, ведь классы узлов будут привязаны не к конкретному классу посетителей, а к их общему интерфейсу. Поэтому если придётся добавить в программу новое поведение, вы создадите новый класс посетителей и будете передавать его в методы узлов.

Структура

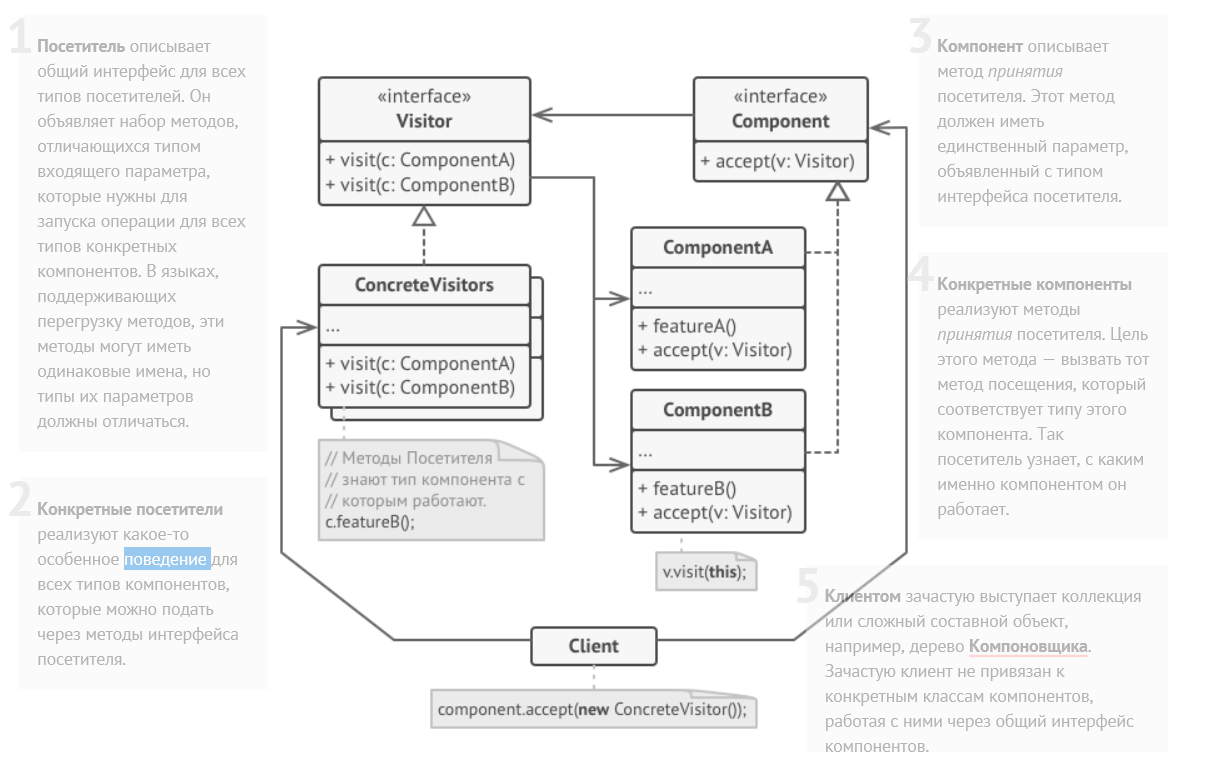


Рисунок 2.24 – Структура паттерна Посетитель

Псевдокод

В этом примере Посетитель добавляет в существующую иерархию классов геометрических фигур возможность экспорта в XML.

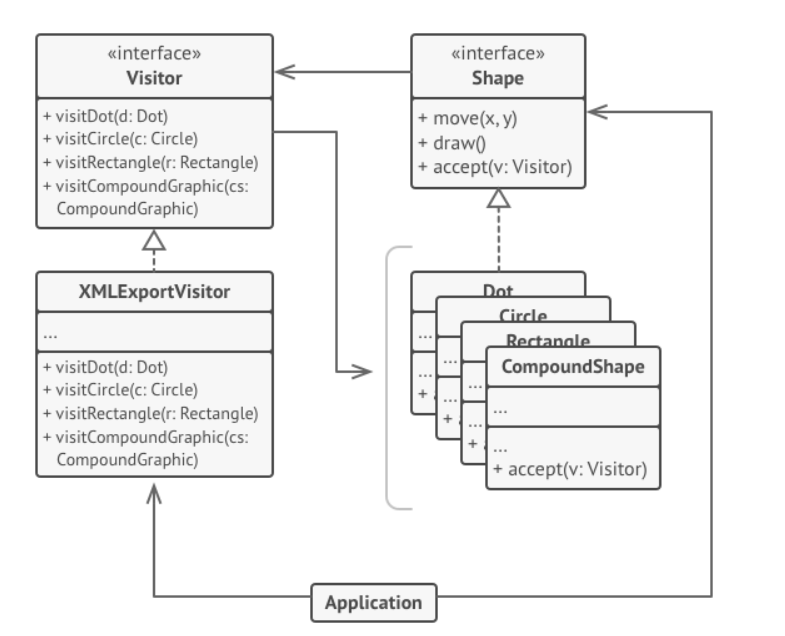


Рисунок 2.25 - Пример организации экспорта объектов в XML через отдельный класс-посетитель.

// Сложная иерархия компонентов.

interface Shape is

method move(x, y)

method draw()

method accept(v: Visitor)

// Метод принятия посетителя должен быть реализован в каждом

// компоненте, а не только в базовом классе. Это поможет

// программе определить, какой метод посетителя нужно вызвать,

// если вы не знаете тип компонента.

class Dot extends Shape is

// ...

method accept(v: Visitor) is

v.visitDot(this)

class Circle extends Dot is

// ...

method accept(v: Visitor) is

v.visitCircle(this)

class Rectangle extends Shape is

// ...

method accept(v: Visitor) is

v.visitRectangle(this)

class CompoundShape implements Shape is

// ...

method accept(v: Visitor) is

v.visitCompoundShape(this)

// Интерфейс посетителей должен содержать методы посещения

// каждого компонента. Важно, чтобы иерархия компонентов

// менялась редко, так как при добавлении нового компонента

// придётся менять всех существующих посетителей.

interface Visitor is

method visitDot(d: Dot)

method visitCircle(c: Circle)

method visitRectangle(r: Rectangle)

method visitCompoundShape(cs: CompoundShape)

// Конкретный посетитель реализует одну операцию для всей

// иерархии компонентов. Новая операция = новый посетитель.

// Посетитель выгодно применять, когда новые компоненты

// добавляются очень редко, а новые операции — часто.

class XMLExportVisitor is

method visitDot(d: Dot) is

// Экспорт id и координат центра точки.

method visitCircle(c: Circle) is

// Экспорт id, кординат центра и радиуса окружности.

method visitRectangle(r: Rectangle) is

// Экспорт id, кординат левого-верхнего угла, ширины и

// высоты прямоугольника.

method visitCompoundShape(cs: CompoundShape) is

// Экспорт id составной фигуры, а также списка id

// подфигур, из которых она состоит.

// Приложение может применять посетителя к любому набору

// объектов компонентов, даже не уточняя их типы. Нужный метод

// посетителя будет выбран благодаря проходу через метод accept.

class Application is

field allShapes: array of Shapes

method export() is

exportVisitor = new XMLExportVisitor()

foreach (shape in allShapes) do

shape.accept(exportVisitor)

Применимость

Когда вам нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.

Посетитель позволяет применять одну и ту же операцию к объектам различных классов.

Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но вы не хотите «засорять» классы такими операциями.

Посетитель позволяет извлечь родственные операции из классов, составляющих структуру объектов, поместив их в один класс-посетитель. Если структура объектов является общей для нескольких приложений, то паттерн позволит в каждое приложение включить только нужные операции.

Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

Посетитель позволяет определить поведение только для этих классов, оставив его пустым для всех остальных.

Шаги реализации

1. Создайте интерфейс посетителя и объявите в нём методы «посещения» для каждого класса компонента, который существует в программе.
2. Опишите интерфейс компонентов. Если вы работаете с уже существующими классами, то объявите абстрактный метод принятия посетителей в базовом классе иерархии компонентов.
3. Реализуйте методы принятия во всех конкретных компонентах. Они должны переадресовывать вызовы тому методу посетителя, в котором тип параметра совпадает с текущим классом компонента.
4. Иерархия компонентов должна знать только о базовом интерфейсе посетителей. С другой стороны, посетители будут знать обо всех классах компонентов.
5. Для каждого нового поведения создайте конкретный класс посетителя. Приспособьте это поведение для работы со всеми типами компонентов, реализовав все методы интерфейса посетителей.

Вы можете столкнуться с ситуацией, когда посетителю нужен будет доступ к приватным полям компонентов. В этом случае вы можете либо раскрыть доступ к этим полям, нарушив инкапсуляцию компонентов, либо сделать класс посетителя вложенным в класс компонента, если вам повезло писать на языке, который поддерживает вложенность классов.

1. Клиент будет создавать объекты посетителей, а затем передавать их компонентам, используя метод принятия.

Преимущества

* Упрощает добавление операций, работающих со сложными структурами объектов.
* Объединяет родственные операции в одном классе.
* Посетитель может накапливать состояние при обходе структуры компонентов.

Недостатки

* Паттерн не оправдан, если иерархия компонентов часто меняется.
* Может привести к нарушению инкапсуляции компонентов.

## Проектирование пользовательского интерфейса.

1. Структурная организация. Структура интерфейса интуитивно понятна абсолютному большинству пользователей, т.к. руководство к использованию и выбору функций содержится в самом интерфейсе.



Рисунок 2.26 – Структура пользовательского интерфейса

Простота. Один из путей поддержания простоты — представление на экране информации, минимально необходимой для выполнения пользователем очередно­го шага. В частности, следует избегать многословных командных имен или сооб­щений. Непродуманные или избыточные фразы затрудняют пользователю извле­чение существенной информации. Другой шаг к созданию простого, но эффективного интерфейса — размещение и представление элементов на экране с учетом их смыслового значения и логичес­кой взаимосвязи. Это позволяет использовать в процессе работы ассоциативное мышление пользователя.

Видимость (обратная связь). Необходимо всегда обеспечивать обратную связь для действий пользователя. Каждое дей­ствие пользователя должно получать визуальное подтверж­дение того, что программное обеспечение восприняло введенную команду; при этом вид реакции, по возможности, должен учитывать природу выполненного действия.

Толерантность. Пользователи обычно изучают особенности работы с новым программным про­дуктом методом проб и ошибок. Эффективный интерфейс должен принимать во внимание такой подход. На каждом этапе работы он должен разрешать только соот­ветствующий набор действий и предупреждать пользователей о тех ситуациях, где они могут повредить системе или данным; еще лучше, если у пользователя суще­ствует возможность отменить или исправить выполненные действия. Даже при наличии хорошо спроектированного интерфейса пользователи могут делать те или иные ошибки. Эти ошибки могут быть как «физического» типа (слу­чайный выбор неправильной команды или данных) так и «логического» (принятие неправильного решения на выбор команды или данных). Эффективный интерфейс должен позволять предотвращать ситуации, которые, вероятно закончатся ошибка­ми. Он также должен уметь адаптироваться к потенциальным ошибкам пользовате­ля и облегчать ему процесс устранения последствий таких ошибок. В проектируемой системе все важные моменты дополнены подтверждением действия. В случае выбора неправильного пункта меню система отреагирует подсказкой («Пожалуйста, повторите ввод»).

Повторное использование. Пользовательский интерфейс проектируемой системы разработан с нуля.

## Выбор инструментальных средств разработки системы

При проектировании программной системы используются различные инструментальные средства: инструменты компании Rational, поддерживающие все этапы жизненного цикла ПО, CASE-среда Rational Rose, Microsoft Visual Studio 2017, PostgreSQL.

Microsoft Visual Studio — линейка продуктов компании Майкрософт, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Данные продукты позволяют разрабатывать как консольные приложения, так и приложения с графическим интерфейсом.

Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и как отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения.

Включая (плагины) для расширения функциональности практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (например, Subversion и VisualSourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода).

Главным преимуществом Visual Studio 2017 является производительность. Обеспечивает возможность создания разнообразных приложений на основе одного набора навыков.

C# - объектно-ориентированный язык программирования.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

Основные достоинства С#:

1. C# создавался параллельно с каркасом Framework .Net и в полной мере учитывает все его возможности — как FCL, так и CLR.
2. C# является полностью объектно-ориентированным языком, где даже типы, встроенные в язык, представлены классами.
3. C# является мощным объектным языком с возможностями наследования и универсализации.
4. C# является наследником языков C/C++, сохраняя лучшие черты этих популярных языков программирования. Общий с этими языками синтаксис, знакомые операторы языка облегчают переход программистов от С++ к C#.
5. Сохранив основные черты своего великого родителя, язык стал проще и надежнее. Простота и надежность, главным образом, связаны с тем, что на C# хотя и допускаются, но не поощряются такие опасные свойства С++ как указатели, адресация, разыменование, адресная арифметика.
6. Благодаря каркасу Framework .Net, ставшему надстройкой над операционной системой, программисты C# получают те же преимущества работы с виртуальной машиной, что и программисты Java. Эффективность кода даже повышается, поскольку исполнительная среда CLR представляет собой компилятор промежуточного языка, в то время как виртуальная Java-машина является интерпретатором байт-кода.
7. Мощная библиотека каркаса поддерживает удобство построения различных типов приложений на C#, позволяя легко строить Web-службы, другие виды компонентов, достаточно просто сохранять и получать информацию из базы данных и других хранилищ данных.
8. Реализация, сочетающая построение надежного и эффективного кода, является немаловажным фактором, способствующим успеху C#.

Технология ASP.NET

Хотя ASP.NET берёт своё название от старой технологии Microsoft ASP, она значительно от неё отличается. Microsoft полностью перестроила ASP.NET, основываясь на Common Language Runtime (CLR), которая является основой всех приложений Microsoft .NET. Разработчики могут писать код для ASP.NET, используя практически любые языки программирования, входящие в комплект .NET Framework (C#, Visual Basic.NET и JScript .NET). ASP.NET имеет преимущество в скорости по сравнению со скриптовыми технологиями, так как при первом обращении код компилируется и помещается в специальный кэш, и впоследствии только исполняется, не требуя затрат времени на парсинг, оптимизацию, и т. д.

Преимущества asp.net:

* Компилируемый код выполняется быстрее, большинство ошибок отлавливается ещё на стадии разработки
* Значительно улучшенная обработка ошибок во время выполнения запущенной готовой программы, с использованием блоков try..catch
* Пользовательские элементы управления (controls) позволяют выделять часто используемые шаблоны, такие как меню сайта
* Использование метафор, уже применяющихся в Windows-приложениях, например, таких как элементы управления и события
* Расширяемый набор элементов управления и библиотек классов позволяет быстрее разрабатывать приложения
* ASP.NET опирается на многоязыковые возможности .NET, что позволяет писать код страниц на VB.NET, Delphi.NET, Visual C#, J# и т. д.
* Возможность кэширования всей страницы или её части для увеличения производительности
* Возможность кэширования данных, используемых на странице
* Возможность разделения визуальной части и бизнес-логики по разным файлам («code behind»)
* Расширяемая модель обработки запросов
* Расширенная событийная модель
* Расширяемая модель серверных элементов управления
* Расширяемая модель серверных элементов управления
* Встроенная поддержка AJAX
* ASP.NET имеет преимущество в скорости по сравнению с другими технологиями, основанными на скриптах.

PostgreSQL (произносится «Пост-Грэс-Кью-Эл») — свободная объектно-реляционная система управления базами данных (СУБД).

Функции являются блоками кода, исполняемыми на сервере, а не на клиенте БД. Хотя они могут писаться на чистом SQL, реализация дополнительной логики, например, условных переходов и циклов, выходит за рамки SQL и требует использования некоторых языковых расширений. Функции могут писаться с использованием одного из следующих языков:

1. Встроенный процедурный язык PL/pgSQL, во многом аналогичный языку PL/SQL, используемому в СУБД Oracle.
2. Скриптовые языки — PL/Lua, PL/LOLCODE, PL/Perl, PL/PHP, PL/Python, PL/Ruby, PL/sh, PL/Tcl, PL/Scheme.
3. Классические языки — C, C++, Java (через модуль PL/Java).
4. Статистический язык R (через модуль PL/R).

PostgreSQL допускает использование функций, возвращающих набор записей, который далее можно использовать так же, как и результат выполнения обычного запроса.

## Программная реализации базовых модулей системы

Для программной реализации был выбран модули регистрации и авторизации пользователей, а также просмотра списка товаров, личный кабинет и просмотр корзины. Эти модули использует такие паттерны проектирования, как «Прототип» - объекты класса копируются в главный исполнимый файл для последующей их обработки и «Состояние» - в зависимости от действий пользователя и состояния программы поведение объектов может меняться. Листинг модуля указан в Приложении Б. Примеры экрана на рис. 2.27 – 2.33.

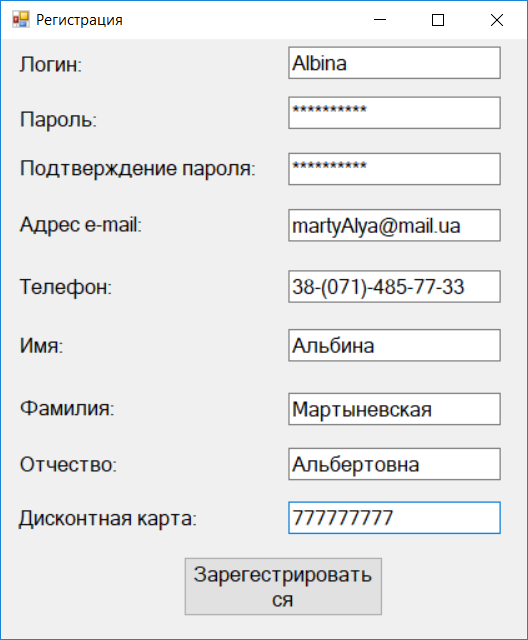


Рисунок 2.27 – Пример формы регистрации пользователей

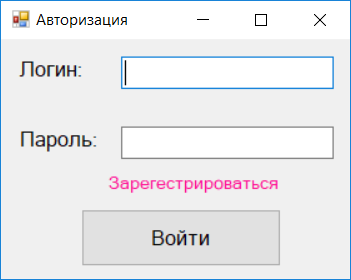


Рисунок 2.28 – Пример формы авторизации

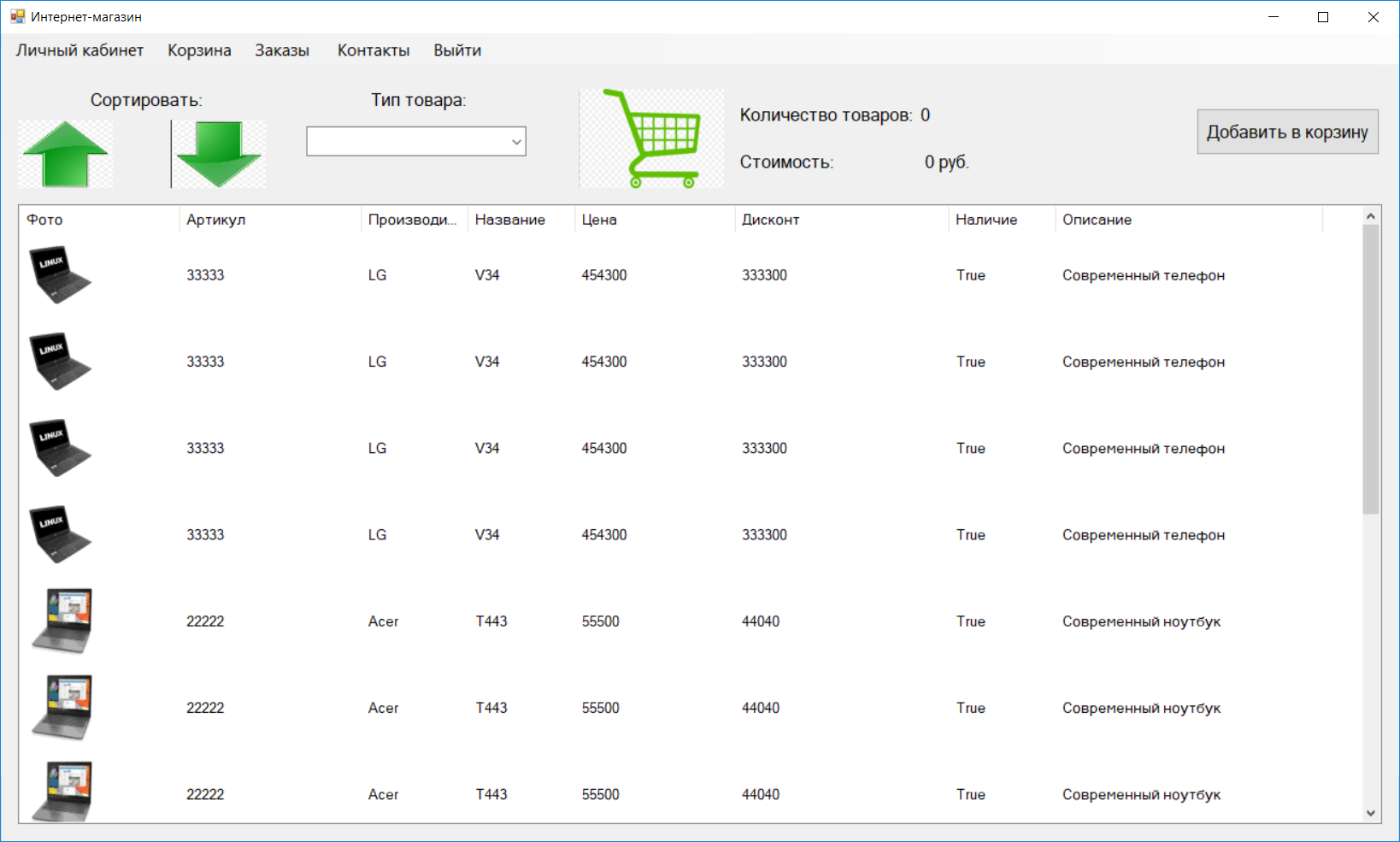


Рисунок 2.29 – Пример главной формы

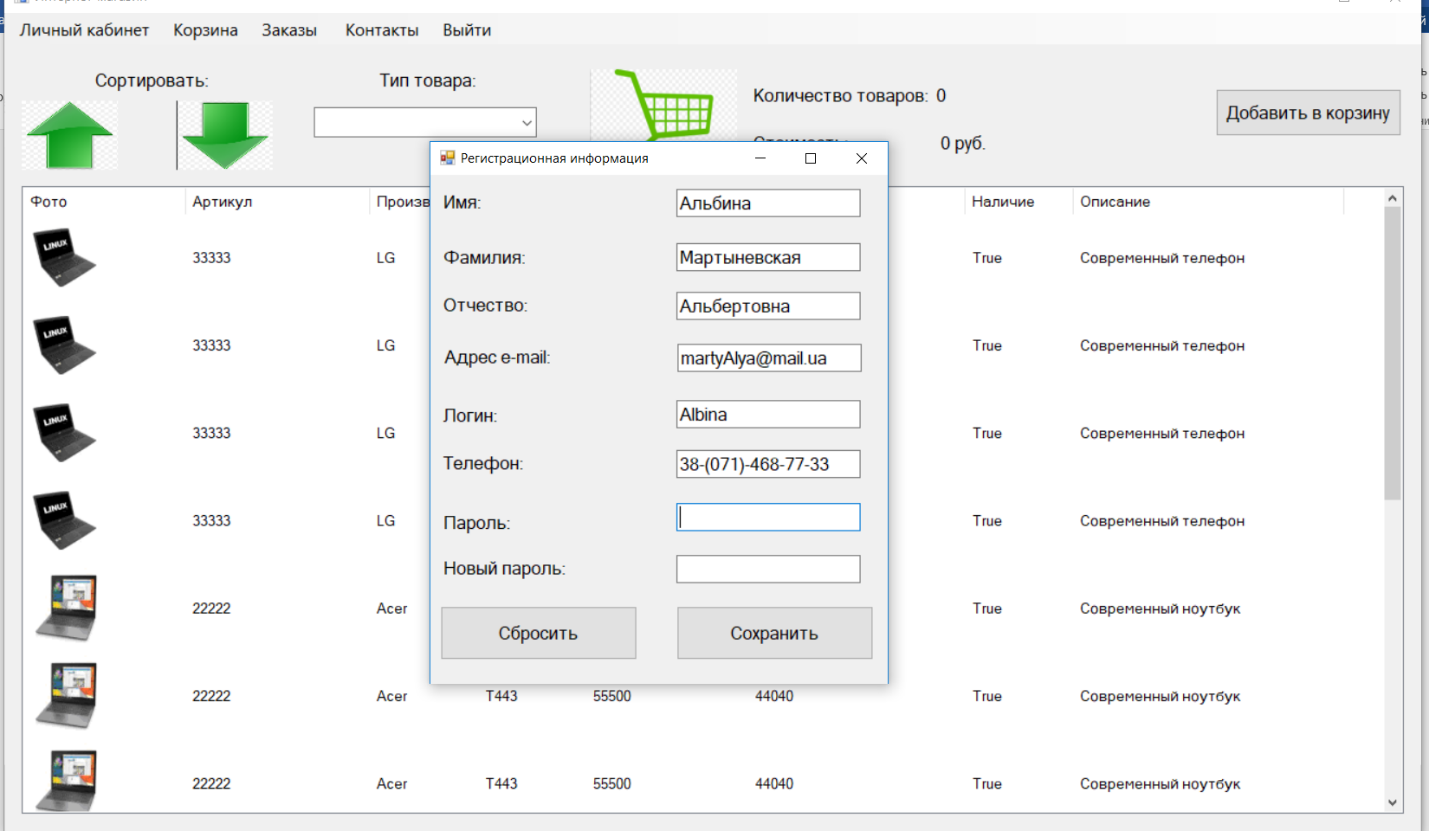


Рисунок 2.30 – Пример формы личного кабинета

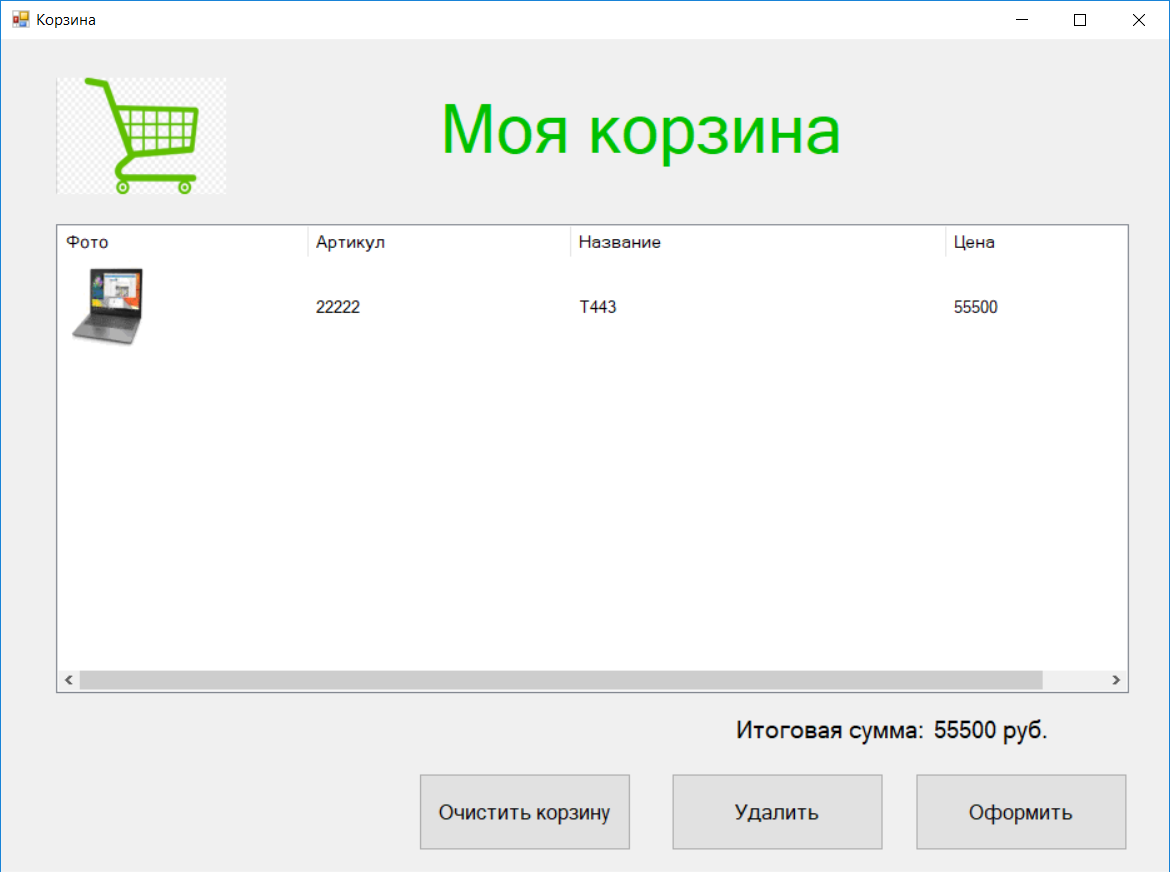


Рисунок 2.31 – Пример формы корзина

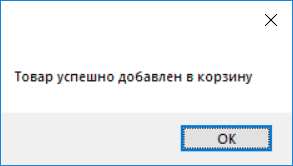


Рисунок 2.32 – Сообщение об успешном добавлении товара в корзину

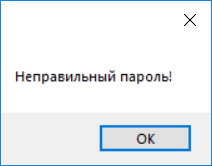


Рисунок 2.33 – Сообщение при ошибке авторизации

## Анализ качества и оценка программного дизайна

Существует целый спектр различных атрибутов, помогающих оценить разработку и добиться качественного дизайна:

1. Применимые к run-time, т.е. ко времени выполнения системы. Среднее время отклика системы – 0.16мс, что позволяет оценить качество дизайна с точки зрения производительности как высокое;
2. Ориентированные на design-time, т.е. позволяющие оценивать качество получаемого дизайна ещё на этапе проектирования; например, средняя нагруженность классов методами – 4 метода в классе;
3. Повторное использование компонентов - общие компоненты используются несколькими модулями разрабатываемой системы;
4. Расширяемость  приложения или системы в связи с появлением новых функциональных требований может быть реализована ввиду относительной простоты приложений;
5. Переносимость. Система не предназначена для использования на других платформах, кроме Windows XP – Windows 10;
6. Поддержка системы может быть легко обеспечена ввиду дешевизны и скорости разработки, прозрачности поведения приложения, простоты анализа ошибок и проблем в работе.
7. Модульность системы – проектируемая система имеет разделение на модули;
8. Возможность тестирования присутствует в данной системе;
9. Возможность и простота локализации. На данный момент система имеет лишь русскую локализацию, однако впоследствии можно добавить любой другой язык и осуществлять переключение между локализациями простым триггером;
10. Совместимость между версиями приложений, между различными приложениями и внешними подсистемами. При желании пользователь может не обновлять приложение в случае выхода новой, более совершенной, версии, однако дополнительный функционал при этом не будет предоставлен;
11. Атрибуты качества архитектурного дизайна как такового. Концептуальная целостность — это фундаментальная идея, или представление, которое объединяет проектное решение системы на всех его уровнях. Одни и те же задачи проектируемая система выполняет одними и теми же способами;
12. Полнота — система содержит всю необходимую информацию для её реализации. В неё включается вся информация об описываемом параметре, известная на момент описания. Система также не содержит не выявленных и не определенных моментов;
13. Однозначность — проектируемая система должна быть внутренне не противоречива, и все работающие с ней должны понимать её одинаково. Требования к системе выражены просто, кратко и точно, используя известные термины. Обычно базовые знания пользователей ПО различаются. Поэтому в его состав нужно включить раздел с определением понятий прикладной области.

Метрики измерения качества ПО:

1. Обзор дизайна (software design review), например, неформальный обзор архитектуры проектировщиком проекта: архитектура довольно проста, что позволит расширить круг возможных пользователей системы. Каждое действие детально описано, поэтому шанс пользователя «заблудиться» в системе минимален.
2. Статический анализ кода это процесс выявления ошибок и недочетов в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода. Задачи, решаемые программами статического анализа кода можно разделить на 3 категории:
3. Выявление ошибок в программах. После тестирования готового модуля, ошибок в коде не осталось;
4. Рекомендации по оформлению кода. Некоторые статические анализаторы позволяют проверять, соответствует ли исходный код, принятому в компании стандарту оформления кода. Имеется в виду контроль количества отступов в различных конструкциях, использование пробелов/символов табуляции и так далее. В Microsoft Visual Studio 2017 код оформляется автоматически (например, автоматическое добавление закрывающей скобки, либо операторов инкапсуляции при создании нового класса;
5. Подсчет метрик. Метрика программного обеспечения - это мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций. Существует большое количество разнообразных метрик, которые можно подсчитать, используя те ли иные инструменты. Примеры метрик качества ПО были представлены и детально разобраны выше.
6. Симуляция и прототипирование. Перед представлением отдельного модуля системы, были рассмотрены несколько его прототипов, в частности, без очистки экрана и с отдельной функцией показа номера. Однако, после тестирования (симуляции действий пользователя системы) было решено создать отдельный экран для каждого пункта подменю.

# ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МОДУЛЯ ПРОЕКТА

Рассмотрим модуль фильтра по типу товаров. Этот модуль использует такой паттерн проектирования, как Состояние. Данный паттерн реализован в классе Control, с целью повышения быстродействия программы. Быстродействие достигается благодаря тому, что паттерна Состояние уменьшает количество методов в классе и объединяет несколько действий в одном методе, исполнение которых выбирается в соответствии с действиями пользователя.

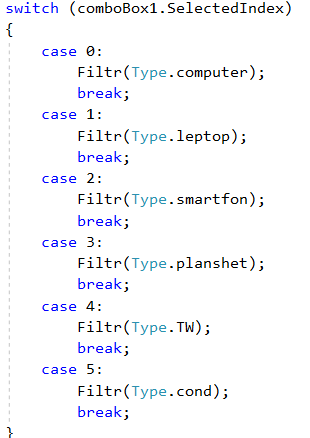


Рисунок 3.1 – Пример использования паттерна «Состояние»

ЗАКЛЮЧЕНИЕ

Результатом курсового проекта является рассмотрение процесса проектирования программного обеспечения для системы автоматизации деятельности Республиканского центра занятости.

После проведённого тестирования работы отдельных модулей можно сделать вывод о том, что программа всегда выполняется верно, в независимости от того, какие данные в неё поступили.

В процессе работы над курсовым проектом были получены знания в методологии программной инженерии, а также умения и навыки проектирования программных систем с использованием различных моделей архитектурных решений.

СПИСОК ЛИТЕРАТУРЫ

1. Макконнелл C. Совершенный код. Мастер-класс: пер. с англ. / С. Макконнелл. СПб.: Питер, 2007. - 896 с.
2. Буч Г. UML. Классика CS: пер. с англ. / Г. Буч, А. Якобсон. 2-е изд. перераб. и доп. СПб.: Питер, 2005. - 736 с.
3. Ларман К. Применение UML 2.0 и шаблонов проектирования: пер. с англ. / К. Ларман. М.: Вильямс, 2007. - 736 с.
4. Антипов В.А. Введение в программную инженерию: Учебник / В.А.Антипов, А.А.Бубнов, А.Н.Пылькин, В.К.Столчнев. – М.: КУРС:ИНФРА-М, 2017. – 336 с.
5. Брауде Э. Технология разработки программного обеспечения: пер. с англ. / Э. Брауде. СПб.: Питер, 2004. - 655 с.
6. Орлов С.А., Цилькер Б.Я. Технологии разработки программного обеспечения: Учебник для вузов. 4-е изд. Стандарт третьего поколения. – СПб.: Питер, 2012. – 608 с.

ПРИЛОЖЕНИЕ А. Функциональная модель

Описание системы с помощью IDEF0 называется функциональной моделью. Функциональная модель предназначена для описания существу ющих бизнес-процессов, в котором используются как естественный, так и графический языки. Для передачи информации о конкретной системе источником графического языка является сама методология IDEF0.

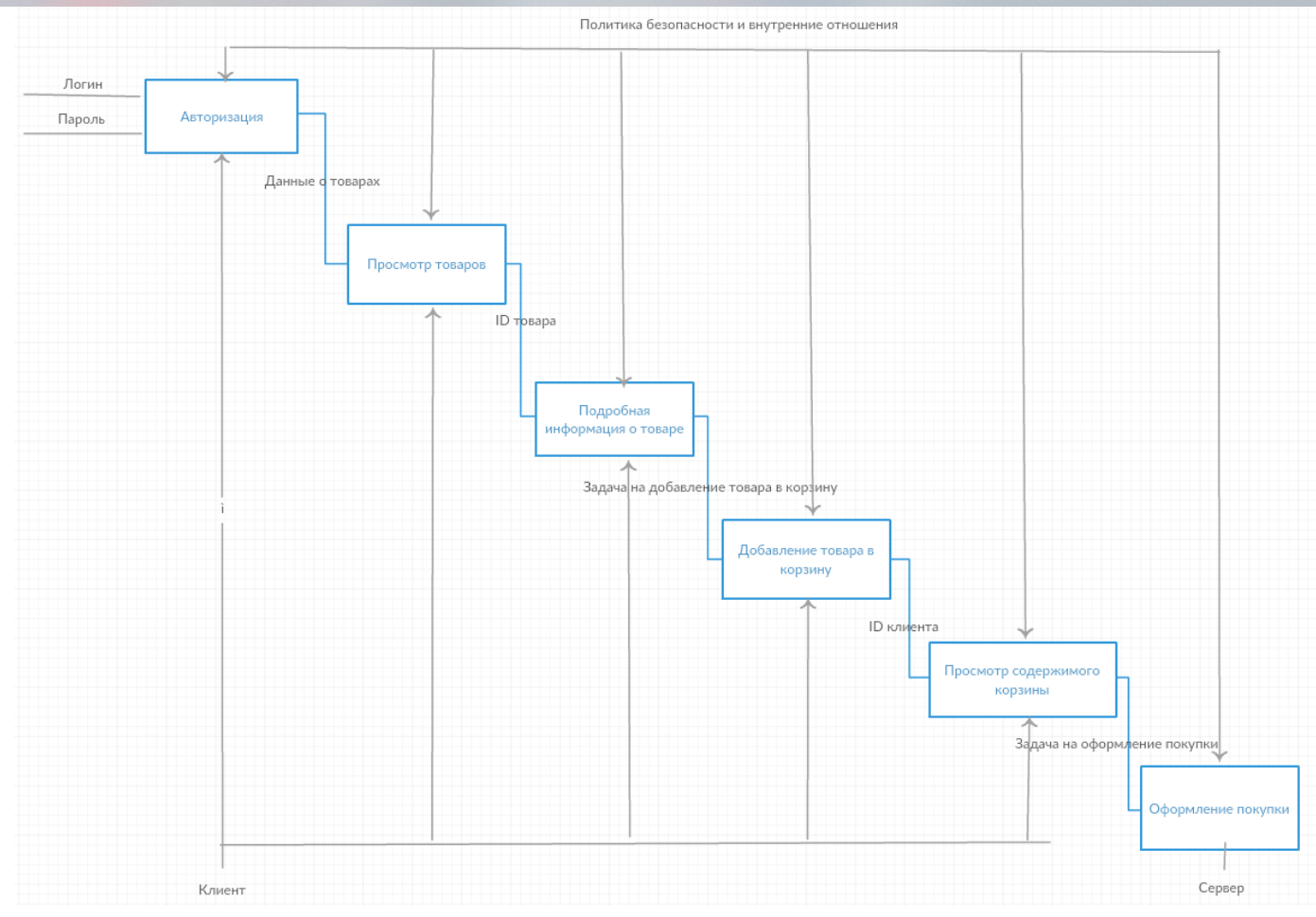


Рисунок ПР А.1 – Функциональная модель по методологии IDEF0

ПРИЛОЖЕНИЕ Б. Фрагменты листинга

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.IO;

namespace Kursovaya

{

public partial class registration : Form

{

public registration()

{

InitializeComponent();

}

private void registration\_Load(object sender, EventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

if (textBox1.Text.Trim(' ') == "" || textBox2.Text.Trim(' ') == "" || textBox3.Text.Trim(' ') == "" || textBox4.Text.Trim(' ') == "" || textBox5.Text.Trim(' ') == "" || textBox6.Text.Trim(' ') == "" || textBox7.Text.Trim(' ') == "" || textBox8.Text.Trim(' ') == "")

{

MessageBox.Show("Не все поля обязательные поля заполнены!");

return;

}

if (textBox2.Text.Trim(' ') != textBox3.Text.Trim(' '))

{

MessageBox.Show("Введенные пароли не совпадают!");

return;

}

string logins = "";

using (StreamReader sr = new StreamReader("logins.txt"))

{

logins = sr.ReadToEnd();

}

string[] massOfLogins = logins.Split("\n\r".ToString().ToCharArray(),StringSplitOptions.RemoveEmptyEntries);

for(int i = 0; i < massOfLogins.Length; i++)

{

string[] oneUser = massOfLogins[i].Split('|');

if (oneUser[0] == textBox1.Text.Trim(' '))

{

MessageBox.Show("Пользователь с таким логином уже существует!");

return;

}

}

File.AppendAllText("logins.txt", Environment.NewLine + textBox1.Text.Trim(' ') + "|" + textBox2.Text.Trim(' ') + "|" + textBox4.Text.Trim(' ') + "|" + textBox5.Text.Trim(' ') + "|" + textBox6.Text.Trim(' ') + "|" + textBox7.Text.Trim(' ') + "|" + textBox8.Text.Trim(' ') + "|" + textBox9.Text.Trim(' '));

MessageBox.Show("Регистрация прошла успешно");

Close();

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.IO;

namespace Kursovaya

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

}

private void textBox2\_TextChanged(object sender, EventArgs e)

{

}

private void textBox1\_TextChanged(object sender, EventArgs e)

{

}

private void label2\_Click(object sender, EventArgs e)

{

}

private void label1\_Click(object sender, EventArgs e)

{

}

private void label3\_Click(object sender, EventArgs e)

{

registration regForm = new registration();

regForm.Show();

}

private void button1\_Click(object sender, EventArgs e)

{

if (textBox1.Text.Trim(' ') == "" || textBox2.Text.Trim(' ') == "")

{

MessageBox.Show("Поле логин или пароль не заполнено!");

return;

}

string logins = "";

using (StreamReader sr = new StreamReader("logins.txt"))

{

logins = sr.ReadToEnd();

}

string[] massOfLogins = logins.Split("\n\r".ToString().ToCharArray(), StringSplitOptions.RemoveEmptyEntries);

for (int i = 0; i < massOfLogins.Length; i++)

{

string[] oneUser = massOfLogins[i].Split('|');

if (oneUser[0] == textBox1.Text.Trim(' '))

{

if(oneUser[1] == textBox2.Text.Trim(' '))

{

main mainForm = new main();

Hide();

main.currentUser = new User(oneUser[0],oneUser[1],oneUser[2],oneUser[3], oneUser[4], oneUser[5], oneUser[6], oneUser[7]);

mainForm.Show();

}

else

{

MessageBox.Show("Неправильный пароль!");

}

return;

}

}

MessageBox.Show("Не существует пользователя с таким логином!");

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.IO;

namespace Kursovaya

{

public partial class Form2 : Form

{

public Form2()

{

InitializeComponent();

Init();

}

public void Init()

{

textBox6.Text = main.currentUser.firstName;

textBox7.Text = main.currentUser.secondName;

textBox8.Text = main.currentUser.otshestvo;

textBox4.Text = main.currentUser.email;

textBox1.Text = main.currentUser.login;

textBox5.Text = main.currentUser.phone;

}

private void Form2\_Load(object sender, EventArgs e)

{

}

private void label4\_Click(object sender, EventArgs e)

{

}

private void textBox3\_TextChanged(object sender, EventArgs e)

{

}

private void button2\_Click(object sender, EventArgs e)

{

if (textBox1.Text.Trim(' ') == "" || textBox2.Text.Trim(' ') == "" || textBox3.Text.Trim(' ') == "" || textBox4.Text.Trim(' ') == "" || textBox5.Text.Trim(' ') == "" || textBox6.Text.Trim(' ') == "" || textBox7.Text.Trim(' ') == "" || textBox8.Text.Trim(' ') == "")

{

MessageBox.Show("Не все поля заполнены!");

return;

}

if (textBox2.Text.Trim(' ') != main.currentUser.password)

{

MessageBox.Show("Старый пароль введен не верно");

return;

}

string logins = "";

using (StreamReader sr = new StreamReader("logins.txt"))

{

logins = sr.ReadToEnd();

}

string[] massOfLogins = logins.Split("\n\r".ToString().ToCharArray(), StringSplitOptions.RemoveEmptyEntries);

for (int i = 0; i < massOfLogins.Length; i++)

{

string[] oneUser = massOfLogins[i].Split('|');

if (oneUser[0] == textBox1.Text.Trim(' ') && textBox1.Text.Trim(' ')!= main.currentUser.login)

{

MessageBox.Show("Пользователь с таким логином уже существует!");

return;

}

}

var path = "logins.txt";

using (var fileStream = File.Open(path, FileMode.Create, FileAccess.Write))

{

using (var writer = new StreamWriter(fileStream))

{

}

}

for (int i = 0; i < massOfLogins.Length; i++)

{

string[] oneUser = massOfLogins[i].Split('|');

if (oneUser[0] != main.currentUser.login)

{

File.AppendAllText("logins.txt",massOfLogins[i]+Environment.NewLine);

}

else

{

File.AppendAllText("logins.txt", textBox1.Text.Trim(' ') + "|" + textBox3.Text.Trim(' ') + "|" + textBox4.Text.Trim(' ') + "|" + textBox5.Text.Trim(' ') + "|" + textBox6.Text.Trim(' ') + "|" + textBox7.Text.Trim(' ') + "|" + textBox8.Text.Trim(' ') + "|" + main.currentUser.discont + Environment.NewLine);

main.currentUser = new User(textBox1.Text.Trim(' '), textBox3.Text.Trim(' '), textBox4.Text.Trim(' '), textBox5.Text.Trim(' '), textBox6.Text.Trim(' '), textBox7.Text.Trim(' '), textBox8.Text.Trim(' '),main.currentUser.discont);

MessageBox.Show("Данные успешно сохранены");

}

}

}

private void button1\_Click(object sender, EventArgs e)

{

textBox1.Text = main.currentUser.login;

textBox2.Text = "";

textBox3.Text = "";

textBox4.Text = main.currentUser.email;

textBox5.Text = main.currentUser.phone;

textBox6.Text = main.currentUser.firstName;

textBox7.Text = main.currentUser.secondName;

textBox8.Text = main.currentUser.otshestvo;

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Kursovaya

{

public partial class Form3 : Form

{

public Form3()

{

InitializeComponent();

Init();

}

private void button1\_Click(object sender, EventArgs e)

{

listView1.Items.Clear();

main.corzina.Clear();

label3.Text = CountTotalSumm().ToString();

MessageBox.Show("Корзина очищена!");

}

public void Init()

{

ImageList imageList = new ImageList();

imageList.ImageSize = new Size(80, 80);

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

{

imageList.Images.Add(main.corzina[i].picture);

}

listView1.SmallImageList = imageList;

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

{

ListViewItem listViewItem = new ListViewItem(new string[] { "", main.corzina[i].articul.ToString(), main.corzina[i].name.ToString(), main.corzina[i].cost.ToString() });

listViewItem.ImageIndex = i;

listView1.Items.Add(listViewItem);

}

label3.Text = CountTotalSumm().ToString() + " руб.";

}

public int CountTotalSumm()

{

int summ = 0;

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

{

summ += (int)main.corzina[i].cost;

}

return summ;

}

private void button3\_Click(object sender, EventArgs e)

{

int index = listView1.SelectedIndices[0];

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

{

if (Convert.ToInt32(listView1.Items[index].SubItems[1].Text) == Convert.ToInt32(main.corzina[i].articul))

{

listView1.Items.Clear();

main.corzina.RemoveAt(i);

Init();

break;

}

}

}

private void button2\_Click(object sender, EventArgs e)

{

if(main.corzina.Count == 0)

{

MessageBox.Show("Корзина пуста!");

return;

}

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Collections;

namespace Kursovaya

{

public partial class main : Form

{

public main()

{

InitializeComponent();

label5.Text = "0";

label6.Text = " 0 руб.";

}

public static User currentUser;

public static List<product> corzina;

public List<product> products;

private void контактыToolStripMenuItem\_Click(object sender, EventArgs e)

{

}

private void main\_Load(object sender, EventArgs e)

{

products = new List<product>();

corzina = new List<product>();

products.Add(new product(1309108,"ATX/AMD", "A4-6300",true, 12800, 12550, "3.7 Chz/RAM 4 GB/HDD 500 GB/noODD/без ОС", new Bitmap("image\\4.png"), Type.computer));

products.Add(new product(1306729, "ATX/AMD", "A6-6400K", true, 13800, 13550, "3.9 Chz/RAM 4 GB/HDD 1 TB/noODD/без ОС", new Bitmap("image\\5.png"), Type.computer));

products.Add(new product(1309205, "ATX/AMD", "A6-7400K", true, 13500, 13250, "3.5 Chz/RAM 4 GB/HDD 1 TB/noODD/без ОС", new Bitmap("image\\6.png"), Type.computer));

products.Add(new product(1309971, "ATX/AMD", "Athlon 200GE", true, 23400, 22950, "3.2 Chz/RAM 4 GB/GPU 2GB GT1030/HDD 1 TB/noODD/без ОС", new Bitmap("image\\7.png"), Type.computer));

products.Add(new product(1310134, "ATX/AMD", "Athlon 200GE", true, 26500, 26000, "3.2 Chz/RAM 8 GB/GPU 2GB GT1030/HDD 1 TB/noODD/без ОС", new Bitmap("image\\8.png"), Type.computer));

products.Add(new product(1309970, "ATX/AMD", "Athlon 200GE", true, 32000, 31400, "3.2 Chz/RAM 8 GB/GPU 4GB RX560/HDD 1 TB/noODD/без ОС", new Bitmap("image\\8.png"), Type.computer));

products.Add(new product(1300367, "ATX/AMD", "Ryzen 3 2200G", true, 40500, 39700, "3.5 Chz/RAM 8 GB/GPU 4GB RX470/HDD 1 TB/noODD/без ОС", new Bitmap("image\\9.png"), Type.computer));

products.Add(new product(1309968, "ATX/AMD", "Ryzen 5 2600", true, 40500, 39700, "3.5 Chz/RAM 8 GB/GPU 4GB RX470/HDD 1 TB/noODD/без ОС", new Bitmap("image\\9.png"), Type.computer));

ImageList imageList = new ImageList();

imageList.ImageSize = new Size(80, 80);

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

{

imageList.Images.Add(products[i].picture);

}

listView1.SmallImageList = imageList;

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

{

ListViewItem listViewItem = new ListViewItem(new string[] {"", products[i].articul.ToString(), products[i].proizv, products[i].name,products[i].cost.ToString(),products[i].discontCost.ToString(),products[i].isStock.ToString(),products[i].description });

listViewItem.ImageIndex = i;

listView1.Items.Add(listViewItem);

}

}

private void личныйКабинетToolStripMenuItem\_Click(object sender, EventArgs e)

{

Form2 form2 = new Form2();

form2.Show();

}

private void listView1\_SelectedIndexChanged(object sender, EventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

// listView1.Clear();

// Sorting(0,1,true);

listView1.ListViewItemSorter = new ListViewItemComparer(4);

}

private void button1\_Click\_1(object sender, EventArgs e)

{

}

private void pictureBox1\_Click(object sender, EventArgs e)

{

IsAccedind.isAcced = true ;

listView1.ListViewItemSorter = new ListViewItemComparer(4);

}

private void pictureBox2\_Click(object sender, EventArgs e)

{

IsAccedind.isAcced = false;

listView1.ListViewItemSorter = new ListViewItemComparer(4);

}

private void button1\_Click\_2(object sender, EventArgs e)

{

listView1.Items.Clear();

}

private void comboBox1\_SelectedIndexChanged(object sender, EventArgs e)

{

listView1.Items.Clear();

switch (comboBox1.SelectedIndex)

{

case 0:

Filtr(Type.computer);

break;

case 1:

Filtr(Type.leptop);

break;

case 2:

Filtr(Type.smartfon);

break;

case 3:

Filtr(Type.planshet);

break;

case 4:

Filtr(Type.TW);

break;

case 5:

Filtr(Type.cond);

break;

}

}

public void Filtr(Type type)

{

ImageList imageList = new ImageList();

imageList.ImageSize = new Size(80, 80);

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

{

imageList.Images.Add(products[i].picture);

}

listView1.SmallImageList = imageList;

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

{

if (products[i].type == type)

{

ListViewItem listViewItem = new ListViewItem(new string[] { "", products[i].articul.ToString(), products[i].proizv, products[i].name, products[i].cost.ToString(), products[i].discontCost.ToString(), products[i].isStock.ToString(), products[i].description });

listViewItem.ImageIndex = i;

listView1.Items.Add(listViewItem);

}

}

}

private void button1\_Click\_3(object sender, EventArgs e)

{

if(listView1.SelectedIndices.Count == 0)

{

MessageBox.Show("Товар не выбран!");

return;

}

int index = listView1.SelectedIndices[0];

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

{

if (Convert.ToInt32(listView1.Items[index].SubItems[1].Text) == Convert.ToInt32(products[i].articul))

{

corzina.Add(products[i]);

MessageBox.Show("Товар успешно добавлен в корзину");

label5.Text = corzina.Count.ToString();

int totalPrice = 0;

for (int j = 0; j < corzina.Count; j++)

{

totalPrice += (int)corzina[j].cost;

}

label6.Text = totalPrice.ToString() + " руб.";

break;

}

}

}

private void корзинаToolStripMenuItem\_Click(object sender, EventArgs e)

{

Form3 form3 = new Form3();

form3.Show();

}

private void main\_FormClosed(object sender, FormClosedEventArgs e)

{

Environment.Exit(0);

}

}

class ListViewItemComparer : IComparer

{

private int col;

public ListViewItemComparer()

{

col = 0;

}

public ListViewItemComparer(int column)

{

col = column;

}

public int Compare(object x, object y)

{

ListViewItem lviX = x as ListViewItem;

ListViewItem lviY = y as ListViewItem;

int result;

if (lviX == null && lviY == null)

{

result = 0;

}

else if (lviX == null)

{

result = -1;

}

else if (lviY == null)

{

result = 1;

}

string Xstr = lviX.SubItems[col].Text;

string Ystr =lviY.SubItems[col].Text;

result = Convert.ToInt32(Xstr).CompareTo(Convert.ToInt32( Ystr));

if (IsAccedind.isAcced == false)

{

return -result;

}

else

{

return result;

}

}

}

public static class IsAccedind

{

public static bool isAcced;

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

namespace Kursovaya

{

public enum Type { computer, leptop,smartfon,planshet, TW, cond }

public class product

{

public uint articul;

public bool isStock;

public uint cost;

public uint discontCost;

public string description;

public Bitmap picture;

public string name;

public string proizv;

public Type type;

public product(uint articul, string proizv, string name, bool isStock,uint cost,uint discontCost,string description,Bitmap picture,Type type)

{

this.articul = articul;

this.proizv = proizv;

this.name = name;

this.isStock = isStock;

this.cost = cost;

this.discontCost = discontCost;

this.description = description;

this.picture = picture;

this.type = type;

}

public product(product product)

{

this.articul = product.articul;

this.proizv = product.proizv;

this.name = product.name;

this.isStock = product.isStock;

this.cost =product.cost;

this.discontCost = product.discontCost;

this.description = product.description;

this.picture =product.picture;

this.type = product.type;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Kursovaya

{

public class User

{

public string login;

public string password;

public string email;

public string phone;

public string firstName;

public string secondName;

public string otshestvo;

public string discont;

public User(string login,string password,string email,string phone,string firstName,string secondName,string otshestvo,string discont)

{

this.login = login;

this.password = password;

this.email = email;

this.phone = phone;

this.firstName = firstName;

this.secondName = secondName;

this.otshestvo = otshestvo;

this.discont = discont;

}

}

}