Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)**

Кафедра автоматизации обработки информации (АОИ)

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Методические указания к лабораторным работам
и организации самостоятельной работы для студентов направления
«Бизнес-информатика» (уровень бакалавриата)

2022

Пекарская Светлана Станиславовна

Технологии программирования: Методические указания к лабораторным работам и организации самостоятельной работы для студентов направления «Бизнес-информатика» (уровень бакалавриата) / С.С. Пекарская. — Томск, 2022. — 97 с.

© Томский государственный университет систем управления и радиоэлектроники, 2022
© Пекарская С.С., 2022

Оглавление

1 ВВЕДЕНИЕ 4

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРОВЕДЕНИЮ ЛАБОРАТОРНЫХ РАБОТ 5

2.1 Лабораторная работа «Анализ идеи проекта» 5
2.2. Лабораторная работа «Карта пользовательских историй» 19
2.3. Лабораторная работа «Проектирование» 28
2.4. Лабораторная работа «Внедрение системы контроля версий» 50
2.5. Лабораторная работа «Реализация front-end приложения» ... 61
2.6. Лабораторная работа «Реализация back-end приложения» 77

3 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОРГАНИЗАЦИИ САМОСТОЯТЕЛЬНОЙ РАБОТЫ 95

3.1 Общие положения 95
3.2 Проработка лекционного материала 95
3.3 Подготовка к лабораторным работам 96

4. РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА 97

1 Введение

Целью лабораторных работ и самостоятельной работы по дисциплине «Технологии программирования» является закрепление теоретических и практических основ выполнения процессов жизненного цикла разработки программного обеспечения, а также формирование у студентов способности к самостоятельному или при помощи преподавателя анализу теоретического материала.

В результате проведения лабораторных работ и самостоятельной работы студенты должны: получить знания о методологиях разработки программного обеспечения, о процессах жизненного цикла программного обеспечения, о технологиях и инструментах, применяемых в процессах разработки программного обеспечения, овладеть навыками применения практик методологий разработки, овладеть навыками применения различных технологий и инструментов в процессах разработки программного обеспечения.

При изучении данной дисциплины необходимо знание студентами дисциплины «Базовые информационные технологии и процессы» в объёме третьего семестра, дисциплин «Объектно-ориентированное программирование» и «Организация баз данных» в объёме третьего и четвертого семестров.

Пособие предназначено для студентов, обучающихся по направлению «Бизнес-информатика» всех форм обучения.

2 Методические указания к проведению лабораторных работ

2.1 Лабораторная работа «Анализ идеи проекта»

Цель работы:

Изучить и научиться применять на практике одну из методик оценки ИТ-проекта с точки зрения его перспективности для разработки.

Форма проведения: Групповое выполнение задания.

Форма отчетности: Защита результата выполненного анализа преподавателю. Выполненный Lean Model Canvas (требуется прикрепить в эл. курсе индивидуально каждым участником подгруппы).

Теоретические основы

На сегодняшний день разработка программного обеспечения (ПО) носит промышленный характер, и, соотвественно, подход к разработке ПО не должен качественно отличаться от разработки промышленного продукта. Необходимы техническая компетентность и грамотное управление, а сама организация процесса разработки должна представлять собой строгий технологический процесс. Таким образом, для организации разработки необходима технология.

Технология разработки основывается на методологии разработки и инструментальных средствах разработки.

Методология позволяет ответить на вопросы «как?», «кто?» и «что?», то естьна какие этапы разбит процесс разработки, как организована команда разработчиков, какие роли имеются в команде, как распределена ответственность по этим ролям, а также какие рабочие продукты создаются в процессе разработки (различные документы, спецификации, диаграммы и т.д.).

Этапы определяются используемой в методологии моделью жизненного цикла разработки, такими как каскадная модель, инкрементная, итерационная, спиральная модели. В жизненном цикле (ЖЦ) ПО выделяют процессы, описывающие определённые виды деятельности. Классически процессы ЖЦ разделяют на основные, вспомогательные, организационные.

К основным процессам относят разработку, функционирование, сопровождение, а также покупку и поставку. Процесс разработки

включает в себя: анализ, проектирование, реализацию, тестирование и отладку, эксплуатационное документирование.

К вспомогательным управление конфигурацией, документирование по этапам, обеспечение качества, верификация и др.

К организационным относят такие как управление, создание инфраструктуры, усовершенствование, обучение. И соотвественно модели ЖЦ формализуют эти процессы определённым образом.

В зависимости от методологии разработки эти процессы могут делиться на более подробные, так проектирование может быть разделено на техническое и архитектурное, или же наоборот укрупнены — реализация, тестирование и отладка могут быть объединены в единый процесс конструирование. Но в детальном рассмотрении различия несущественны.

ПО представляет собой некий функционал для решения конкретных задач. Соответственно, потребность в ПО возникает, с появлением конкретной проблемы, которую требуемое ПО будет способно решить. Любой проект может быть охарактеризован такими параметрами как цель, бюджет, ЖЦ. Цели проекта должны быть конкретны (не должны иметь дополнительных смыслов), измеримы (чтобы их можно было оценить), согласованы (не должны противоречить друг другу, не должны быть взаимоисключающими), реалистичны (их возможно достичь) и достижимы в приемлемые сроки. Для формирования целей может быть использован подход SMART. Он может применяться к целям различных уровней как к глобальным целям, так и к менее крупным, в том числе и к целям по конкретным задачам.

Принципы SMART

Specific — точные и конкретные

Нет возможности интерпретировать цель разными способами.

Неверно: сделать верстку на сайт www.site.com кроссбраузерной

Верно: Сайт www.site.com должен одинаково отображаться в браузерах Opera 6+, Firefox 2+ , Chrome 4+

Measurable — измеримые

Неверно: повысить количество продаж

Верно: увеличить продажи бренда А на 25%

Achievable — достижимые

Реалистичность выполнения задачи влияет на мотивацию исполнителя, если цель не является достижимой — вероятность ее выполнения будет стремиться к 0. Также стоит отметить, каждый специалист обладает определённым набором знаний и навыков, поэтому цели нужно категоризировать, и для различных людей необходимо подбирать соответствующие задания.

Relevant/**r**ealistic — значимые/реалистичные

Значимость определяется на основе вклада решения конкретной задачи в достижение глобальных стратегических задач проекта. Цель должна быть реалистичной и уместной и не должна нарушать баланс с другими целями и приоритетами проекта.

Time bound/framed — ограниченные по времени

Любая задача должна иметь сроки выполнения. Как правило, задача без сроков вытесняется срочными задачами и её решение постоянно отодвигается отодвигается.

Неверно: сделать на сайте раздел "Контакты" для демонстрации клиенту до среды.

Верно: сделать на сайте раздел "Контакты" для демонстрации клиенту до 10:00 12.03.2021.

По возможности поставленная цель должна отвечать всем пяти принципам.

Для принятия решения о реализации проекта необходимо провести предварительные исследования, т.к. любой проект требует ресурсов финансовых, временных и человеческих, стоит провести подготовительную работу и определиться стоит ли затрачивать ресурсы на конкретный проект. На начальном этапе проводится анализ рынка. Необходимо определить, насколько высок спрос на планируемое ПО, какие на текущий момент есть предложения на рынке, кто является целевой аудиторией, является ли эта целевая аудитория платёжеспособной, можно ли на неё ориентироваться и т.д. При определении пользователей зачастую используют Метод персон — качественный метод исследования, целью которого является создание

нескольких персонажей с характеристиками потенциальных пользователей продукта. То есть каждый персонаж — собирательный образ одной из групп целевой аудитории. Как правило, учитывается возраст, доход, семейное положение, место проживания, сфера деятельности, какие потребности/мотивация, как будет использоваться программный продукт данным персонажем и т.д.

Как уже говорилось, в самом начале есть только идея продукта. Эту идею нужно осмыслить и презентовать другим.

PROBLEM
List your customer's top 3 problems
2

EXISTING ALTERNATIVES
List how these problems are solved today

SOLUTION
Outline a possible solution for each problem
4

KEY METRICS
List the key numbers that tell you how your business is doing
8

UNIQUE VALUE PROPOSITION
Single, clear, compelling message that turns an unaware visitor into an interested prospect
3

HIGH-LEVEL CONCEPT
List your X for Y analogy (e.g. YouTube = Flickr for videos)

UNFAIR ADVANTAGE
Something that can not be easily copied or bought
9

CHANNELS
List your path to customers
5

CUSTOMER SEGMENTS
List your target customers and users
1

EARLY ADOPTERS
List the characteristics of your ideal customers

COST STRUCTURE
List your fixed and variable costs
7

REVENUE STREAMS
List your sources of revenue
6

Lean Canvas is adapted from The Business Model Canvas (http://www.businessmodelgeneration.com) and is licensed under the Creative Commons Attribution-Share Alike 3.0 Un-ported License.

Рис. 1. — Lean Model Canvas

Для этого имеется достаточно удобный инструмент Lean Model Canvas — одностраничный шаблон. Данный инструмент был предложен Эшем Маурья, как некое переосмысление шаблона бизнес-модели Александра Остервальдера и Ива Пинье для IT-проектов. Также стоит отметить, что это Lean Model Canvas является одной из практик Lean — концепции бережливого производства, и позволяет учесть и показать все основные составляющие проекта компактно и достаточно доступно для

понимания на одном листе. Такая лаконичная форма анализа позволяет получить достаточно полное представление о планируемом проекте, и что немаловажно, позволяет достаточно просто представить свою идею другим, её основные аспекты. По сути это предварительный и в тоже время начальный этап разработки, который можно рассматривать как часть анализа требований.

Описание работы с шаблоном Lean Model Canvas

Существует несколько подходов к очерёдности заполнения пунктов шаблона. Так изначально Эш Маурья предлагал определять проблемы и затем определить сегменты заказчиков. На сегодняшний день им предложен другой порядок — определить сначала именно сегменты заказчиков и затем более конкретно определить проблемы именно данных заказчиков.

1. Целевая аудитория, кто клиент (Customer segments)

Изначально имеется некая идея проекта, идея некоего ПО, идея некоего функционала, то естьуже есть некое видение проблем, которые планируется решить. И пункты Problems и Customer segments (CS) тесно взаимосвязаны. Необходимо определить целевую аудиторию (CS) для кого будут определяться проблемы и будут предлагаться решения.

Если предполагается широкое распространение, необходимо выделить ту часть аудитории, с которой нужно начинать. И чем точнее будет выделен сегмент клиентов для продукта, тем лучше удастся понять и решить их проблемы. И тем успешнее будет итоговый продукт. Таким образом, можно начать с одного сегмента и потом расширить, чем сразу выделить большое число сегментов и попытаться решить проблемы их всех, продукт может в итоге получиться довольно усреднённым и поэтому неудобным для всех этих сегментов.

Необходимо изначально определить совпадают или нет две категории клиентов — кто пользуется продуктом (user), и кто за него платит (customer). Если приложение предназначено для детей, платят за него, как правило, родители. При несовпадении этих видов клиентов необходимо визуально разграничить в шаблоне что к какой категории относится, например, обозначить разным цветом.

Далее необходимо определить Early adopters или ранние участники — первые пользователи продукта, конкретные люди, те, кто даст первую обратную связь. По возможности следует включить их в обсуждение модели еще до создания продукта.

2. Какие проблемы целевой аудитории будут решаться (Problems)

Часто разработчик продукта исходит из того, что он может сделать, а не из того, что требуется клиенту. Важно определить проблемы клиента, и только после этого предложить решение.

Если проблемы разных сегментов клиентов отличаются, тогда для каждого сегмента предпочтительно создать свою модель. Так будет нагляднее, чем при попытке уместить все вместе на одной модели. Возможно, придется делать разные продукты для разных сегментов.

На следующем шаге необходимо рассмотреть имеющиеся альтернативы. Как правило, проблема клиента возникла не сейчас и существует уже какое-то время, и значит, как-то уже решается. И с имеющимися альтернативами потребуется конкурировать. Необходимо конкурентов выписать и затем тщательно проанализировать предлагаемые ими продукты.

3. Уникальное предложение (Unique value proposition)

Определив решения выделенных проблем, необходимо сформулировать в одном предложении, в чем будет заключаться уникальность продукта. Чем он будет отличаться от конкурентов — описать его ключевое отличие. Здесь же предлагается дать краткую и понятную аналогию — краткую ассоциацию, которую запомнят клиенты (в одной из статей Эш Маурья предлагает такой пример: “YouTube = Flickr for videos”).

4. Решение (Solution)

Пункт “Решение” — основная часть, что, собственно, предлагается. Здесь указываются ключевые возможности решения, но с учетом описанных ранее выделенных проблем, то есть указывается, как именно эти проблемы будут решаться. На каждую из указанных проблем предлагается решение (стоит помнить, что необходимо предложить вполне пригодное для практической реализации решение).

5. Каналы продаж (Channels)

Здесь определяется как клиенты узнают о новом продукте. Это могут быть социальные сети, блоги, различная реклама и т.д. Здесь стоит опираться на CS, необходимо выбрать те каналы, которые обычно используются данными CS.

6. Источники доходов (Revenue streams)

Продукт создается, как правило, не только ради реализации идеи, а чтобы еще и заработать денег. При анализе возможных источников доходов стоит обсудить предлагаемые идеи решений с ранними последователями, получить обратную связь, в частности готовы ли и они будут платить и в каком объёме за решение выделенных проблем.

7. Структура затрат (Cost structure)

После того, как определено, что создавать и как созданное продвигать для целевой аудитории (CS), можно определить структуру затрат. Согласно предложенного шаблона здесь необходимо определить постоянные и переменные статьи расходов. К постоянным относят те, что не зависят от объёма производства, например аренда помещения, коммунальные расходы, телефон, интернет, хостинг и т.д. К переменным можно отнести оплату бухгалтерских и юридических услуг, затраты на маркетинг, затраты на разработку (в частности, на различное ПО, лицензии и т.д.), зарплату сотрудников, оплату переработок и т.д.

Стоит отметить, что деление достаточно условно, так часть затрат в разных случаях может быть отнесена к постоянным или к переменным, например, зарплата сотрудников может быть как постоянными затратами, так и переменными (это может фиксированный оклад, почасовая оплата, сдельная оплата и т.д.). В этом пункте необходимо определить все требуемые затраты как постоянные, так и переменные. Стоит казать как первоначальные затраты на разработку, так и затраты на развитие и поддержку

8. Ключевые метрики проекта (Key metrics)

Любой прогресс необходимо измерять. И стоит сразу определить критерии отслеживания проекта. При достижении каких значений метрик проект будет считаться успешным. Здесь оценивается финансовый успех проекта, рост клиентской базы и т.д.

Существуют как базовые метрики, применимые в широком круге направлений и отраслей, так и более специфические, близкие к конкретному продукту. К базовым можно отнести такие метрики как:

Customer Acquisition Cost (CAC) — стоимость привлечения клиента. Показывает, какие затраты нужны для привлечения одного пользователя CAC = сумма потраченного рекламного бюджета на привлечение новых клиентов / количество привлечённых клиентов

Retention rate — коэффициент удержания, показывает какое количество пользователей продолжает использовать продукт через определённое время; данная метрика напрямую связана с метрикой “пожизненная ценность клиента” (LTV). Retention rate = количество активных пользователей на конец периода / количество активных пользователей на начало периода * 100%.

Churn rate — коэффициент оттока, показывает отток клиентов. Churn rate = (1 — количество пользователей на конец периода / количество пользователей на начало периода) * 100%.

Lifetime Value (LTV) — пожизненная ценность клиента, показывает чистый доход, который приносит клиент за все время использования продукта. Данная метрика должна всегда быть выше CAC (в противном случае проект убыточен). Методики расчёта варьируются в зависимости от сферы применения.

Net Promoter Score (NPS) — индекс потребительской лояльности, показывает, насколько охотно пользователи готовы поделиться впечатлениями от продукта. Для расчёта может быть использован следующий подход: среди пользователей проводится опрос с оценкой по шкале и градацией от “совсем не рекомендую” до “обязательно порекомендую”. Ответы разделить на 3 группы три группы клиентов — критично настроенные (большая часть интервала шкалы оценки), нейтрально настроенные и позитивно настроенные (обычно оставшаяся часть интервала в равных долях). NPS = общее число позитивно настроенные (%) — общее число критиков (%). Хорошим считается NPS более 50%.

Если говорить о конкретных направлениях, то здесь можно привести такие примеры:

- Для социальны сетей в качестве одной из метрик может выступать количество активных пользователей за некий длительный период, например, за месяц.
- Для видео/аудио сервисов — количество часов прослушивания/просмотра.
- Для мессенджеров — количество активных пользователей, например, за день (здесь предпочтителен короткий период).
- И др.

После определения 6, 7 и 8 пунктов их следует проанализировать вместе: получается ли при таких расходах, доходах выйти на некие показатели прибыли и какое для этого необходимо количество клиентов и т.д.

9. Скрытое преимущество/фора (Unfair advantage)

Последний и наиболее трудный пункт. На сегодняшний день технологии развиты, и любая вещь очень легко копируется, за достаточно непродолжительное время, в частности интересный и востребованный функционал программного сервиса может быть достаточно легко и за короткое время повторён конкурентом. Для заполнения данного пункта необходимо подумать есть ли возможность иметь преимущество перед конкурентами, в чём оно может заключаться. В качестве примеров форы можно назвать, например, значительную клиентскую базу, патенты, лицензии, бренд, высокую стоимость вхождения, новые технологии собственной разработки. Что-то, что не позволит конкурентам быстро повторить достижения.

Инструментарий для создания Lean Model Canvas

В базовом варианте может использоваться распечатанный на листе бумаги шаблон. При организации удалённой работы команд могут использоваться различные онлайн-сервисы. Один из таких сервисов — miro.com. Данный сервис имеет бесплатную версию, но требует регистрации. В бесплатной версии поддерживается множество шаблонов, в том числе и Lean Model Canvas.

Для работы с шаблоном Lean Model Canvas в сервисе miro.com требуется выполнить следующие шаги:

1. Зарегистрироваться/войти.
2. Выбрать шаблон: Create board — Show all templates — Lean Canvas.

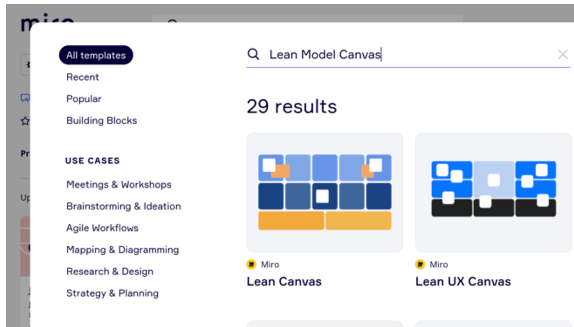

Рис. 2. — Шаблон Lean Canvas

3. Требуется выбрать пустой шаблон (Use blank template). В данном сервисе шаблон реализован как расчерченная доска со стикерами.

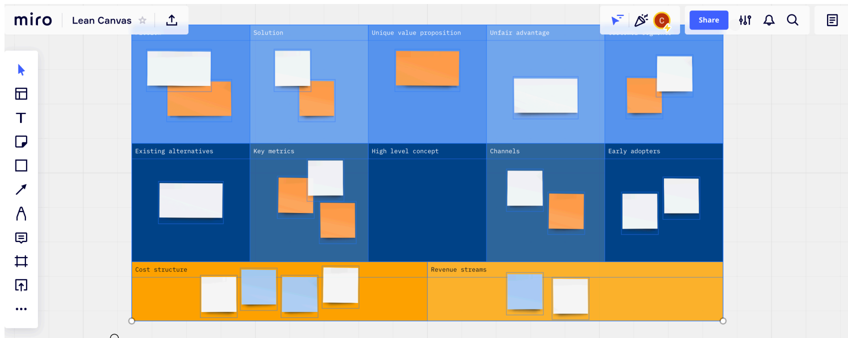

Рис. 3. — Blank template Lean Canvas

4. Для организации совместной работы требуется пригласить всех заинтересованных лиц (Share):

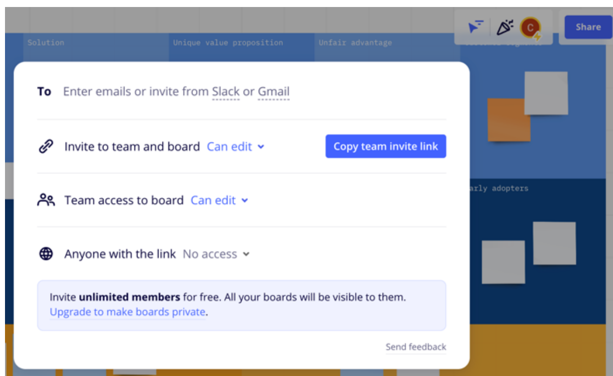

Рис. 4. — Добавление участников

- В графе “To” необходимо указать почту куда направить ссылку-приглашение. Этапы 1—5 выполняются одним из членов команды. Остальные должны получить ссылку и перейти по ней.
- Тем, кто перешёл по ссылке, будет предложено присоединиться и выбрать роль (в данном случае следует выбрать роль разработчика).
- После того как все присоединились к работе на поле шаблона будут видны курсоры всех участников с подписью и вносимые ими изменения.

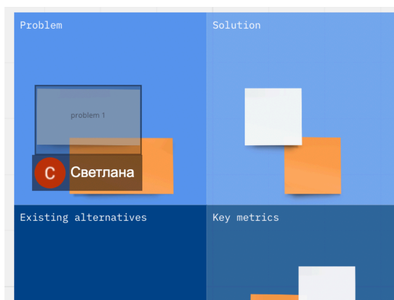

Рис. 5. — Совместная работа на доске miro

5. Для организации совместной удалённой работы, как правило, требуется дополнительно использовать некий видеочат, в бесплатной

версии эта опция не доступна, поэтому можно использовать любой удобный сторонний бесплатный сервис позволяющий организовать групповые видеозвонки.

6. Заполненный в результате проведённого анализа шаблон можно экспортировать:

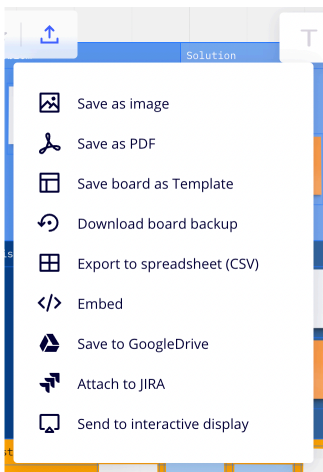

Рис. 6. — Экспорт заполненного шаблона

Задание

1. Разбиться на подгруппы по 4 человека (в данном составе вы будете работать над реализацией проекта в течение всего семестра).
2. Определиться с идеей проекта, идея проекта не должна повторяться, у каждой подгруппы должен быть уникальный проект.
3. Провести анализ идеи проекта с использованием Lean Model Canvas согласно рассмотренной методики.
4. Представить проведённый анализ (выполненный Lean Model Canvas и устная презентация идеи проекта на основе анализа).

Важно: все реализуемые в рамках дисциплины проекты должны иметь клиентскую и серверную часть, что позволит обращаться к реализованному сервису с различных устройств и получать актуальную информацию/получать доступ к функционалу.

Варианты:

1. Сервис для создания, хранения, редактирования и удаления заметок.
2. ToDo List с возможностью добавлять, сохранять и удалять текущие и выполненные задачи.
3. Календарь с возможностью добавлять, сохранять, и удалять события.
4. Kanban-доска.
5. Task manager.
6. Сервис расчёта (осуществление различных операций расчёта по заданным на клиенте параметрам на удалённом сервере и передача рассчитанного результата на клиентскую часть, например, сервис осуществляющий шифрование/дешифрование; решение уравнений: линейных, квадратных, систем уравнений и т.п.).
7. Агрегатор новостных сайтов для формирования новостной ленты.
8. Доска объявлений возможностью добавлять, сохранять, редактировать и удалять сохранённые объявления.
9. Ваша идея проекта.

Список источников

1. Вольфсон Б. И. Гибкие методологии программирования / Б. И. Вольфсон, 2012—112 с.
2. Гагарина, Л. Г. Технология разработки программного обеспечения: учебное пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Сидорова-Виснадул ; под ред. Л.Г. Гагариной. — Москва: ФОРУМ : ИНФРА-М, 2022. — 400 с. — URL: https://znanium.com/catalog/product/1699927 (дата обращения: 08.02.2022). — Режим доступа: по подписке.
3. Мирошниченко Е.А. Технологии программирования: учебное пособие / Е.А. Мирошниченко. — 2-е изд., испр. и доп.—Томск: Изд-во Томского политехнического университета, 2008. —124 с.
4. B Kayvan Kaseb “Using Lean in Software Development” URL: https://medium.com/kayvan-kaseb/using-lean-in-software-development-1b01bbb98d6e (дата обращения 08.02.2022).
5. Цикл статей “Advice and answers from the LEANSTACK Team” URL: http://ask.leanstack.com/en/ (дата обращения 08.02.2022).

6. Steve Mullen “An Introduction to Lean Canvas” URL: https://medium.com/@steve_mullen/an-introduction-to-lean-canvas-5c17c469d3e0 (дата обращения 08.02.2022).
7. Как использовать метод персон в продуктовом анализе URL: https://skillbox.ru/media/management/kak_ispolzovat_metod_person/ (дата обращения 08.02.2022).
8. SMART Goals Lead to Achievable Results URL: https://www.presentermedia.com/blog/smart-goals-and-objectives (дата обращения 08.02.2022).
9. Evgeny Lazarenko “Critical Metrics Every Product Manager Must Track” URL: https://productcoalition.com/critical-metrics-every-product-manager-must-track-c5f1e46e3423 (дата обращения 08.02.2022).
10. Как выбрать метрики для продукта: разбираем на примерах URL: https://skillbox.ru/media/management/kak_vybrat_metriki_dlya_produkta_razbiraem_na_primerakh/ (дата обращения 08.02.2022).

2.2. Лабораторная работа «Карта пользовательских историй»

Цель работы:

Изучить и научиться применять на практике одну из методик описания требований к разрабатываемому программному продукту.

Форма проведения: Групповое выполнение задания.

Форма отчетности: Защита результата выполненного анализа преподавателю. Сформированные пользовательские истории, карта пользовательских историй (требуется прикрепить в эл. курсе индивидуально каждым участником подгруппы).

Теоретические основы

Анализ требований

После того, как принято решение о реализации необходимо определить цели проекта. Так укрупнённо целью всего проекта является, например, интернет-магазин. Но в целом заказчик имеет к нему более конкретные требования. Они и должны быть определены на этапе анализа требований.

Существует, образно говоря, два полюса идея заказчика/пожелания пользователей и способы реализации этих идей/пожеланий конкретными разработчиками. Для старта проекта требуется достичь общего понимания/видения и создать некую спецификацию на реализуемый проект (например, ТЗ). Важно в процессе анализа требований понять, что именно хочет заказчик в действительности, заказчик не всегда точно представляет, что именно он хочет получить. Например, указывает задачу достаточно крупно: необходимо иметь раздел “Бухгалтерия”, в то время как заказчику нужно решать такие задачи как “Учёт заработной платы”, “Составление табеля” и т.д. И если на этапе анализа сформулировать требования недостаточно чётко проект может требовать значительной доработки, которая в ряде случаев не только повысит стоимость разработки, но и сделает проект дорогим и неудобным в поддержке, а некоторые уточнения в требованиях могут оказаться нереализуемы в силу особенностей спроектированной архитектуры и т.д. Таким образом, анализ требований очень важный этап разработки.

В рамках данного процесса проводится исследование предметной области, выявляются основные, наиболее важные требования к ПО со стороны заказчика/пользователя. На основании этого создаётся некая

спецификация требований на программный продукт, определяющая условия и результаты действия программного продукта, но не указывающая способов достижения этих результатов. В свою очередь полученная спецификация ложится в основу следующего этапа — проектирования.

В разрабатываемом ПО всегда первична функциональность, пользователь ожидает от программы решения некоего перечня задач. Поэтому большое внимание уделяется формированию общего видения задач, в частности пользователь не всегда знает, что ему действительно нужно, разработчик не всегда может должным образом может интерпретировать конкретную потребность пользователя.

Пользовательские истории

Для описания потребностей пользователя существуют различные подходы. В agile-методологиях в качестве инструмента описания требований к программному продукту выступают Пользовательские истории (User story), они представляют собой неформальную короткую формулировку намерения, описывающую что-то, что система (ПО) должна делать для пользователя. Важно разграничивать требование и User story. Требование является формальным описанием функциональности.

User story создаётся при непосредственной работе с представителями заказчика. В целом они позволяют описать некие требования к продукту на понятном для всех заинтересованных лиц — стейкхолдеров языке, позволяет достичь общего видения проекта, а сам процесс их формирования позволяет вовлечь стейкхолдеров в процесс работы над проектом. Как уже отмечалось, сотрудничество с заказчиком является одной из основных ценностей для agile-методологий.

В целом, User story:

- Являются кратким описанием намерения (что-то что нужно сделать): короткое легко читаемое и понятное (понимаемое всеми одинаково).
- Представляет собой инкремент функциональности системы (ПО), которая может быть реализована в течение небольшого временного отрезка, например нескольких дней/недель.
- Может быть легко измерима (легко выделить критерий для оценки достигнута цель или нет).
- Не имеет детализации каким именно образом цель должна быть достигнута (на начальном этапе), что позволяет избежать задержек в

разработке, нагромождения требований и достаточно ограниченной формулировки решения.

- Требует минимальное документирование.

Для написания User story, как правило, используется шаблон:

Как, <роль/пользователь(actor)>, я <что-то хочу получить>, <с такой-то целью>

Таким образом, в User story один actor, одно действие, одна ценность. При этом истории должны быть максимально независимы друг от друга, для того чтобы не вызывать задержек работе, чтобы сложности реализации одной пользовательской истории не влияли на другие, не задерживали их реализацию.

Для оценки User story используется критерий INVEST:

Independent — независимая от других историй, то есть истории могут быть реализованы в любом порядке.

Negotiable — обсуждаемая, отражает суть, а не детали; не содержит конкретных шагов реализации.

Valuable — ценная для клиентов, бизнеса и стейкхолдеров.

Estimable — оцениваемая по сложности и трудозатратам.

Small — компактная, может быть сделана командой за одну итерацию.

Testable — тестируемая, имеет критерии приемки.

User story не являются конечными требованиями к системе и предназначены для ого, чтобы выяснить, что нужно заказчику/конечному пользователю. Следующим после написания пользовательских историй шагом является более детальное обсуждение с заказчиком каждой из них в отдельности. На данном этапе они детализируются, и в результате определяются верхнеуровневые требования к системе и критерии выполнения сформированного требования. При создании истории важно, что если при варьировании в шаблоне роли или ценности история не претерпевает изменений, то она не совсем корректна и должна быть доработана.

Стоит сказать, что помимо, пользовательских историй имеются и так называемые технические истории, например, необходимо перейти с одной библиотеки на другую, то естьВ качестве пользователя в данной истории

выступает разработчик. При этом реализация такой истории затрачивает ресурсы, а очевидной ценности для конечного пользователя системы не несёт, поэтому рекомендуется при написании таких историй исходить из того, какую ценность это изменение принесёт конечному пользователю, например, измеримо увеличится быстродействие, и соответственно писать такую историю для роли «Пользователь».

Для каждой пользовательской истории определяется важность и в соответствии с этим показателем будет определена очерёдность работы с ней. Чем важнее история, тем раньше она должна быть реализована.

Наиболее популярная на сегодняшний день agile-методология SCRUM и её комбинации с другими agile-методологиями. В SCRUM на начальном этапе формируют беклог продукта, состоящий из пользовательских историй (User story / US). Работа над проектом разбивается на так называемые спринты, длящиеся не более 1 месяца. Каждый спринт так же имеет свой беклог, в него переносятся пользовательские истории из беклога продукта, при планировании спринта они детализируются, то естьосновная задача истории разбивается на чёткие и понятные исполнителям подзадачи.

Для работы над проектом требуется обеспечить общее видение целей и задач у всех заинтересованных в успешной реализации лиц (стейкхолдеров). US, в общем случае, позволяют сформировать видение того, что должно быть реализовано, что будет делать ПП. Но также важно понимать и то, что за ПП это будет. Для комплексного проектирования продукта применяется User story mapping. основу данной методики ложится так называемый пользовательский путь/пользовательское путешествие (User Journey Map / UJM). Основное применение UJM — UX-дизайн продукта. В общем случае UJM отражает путь взаимодействия пользователя с приложением для решения некой задачи. UJM может быть построен для различных уровней детализации взаимодействия: может быть построен сквозной путь пользователя от входа/открытия приложения до завершения работы с ним или же путь пользователя для решения некой конкретной задачи. У программного продукта (ПП) может быть достаточно широкий круг пользователей, разные пользователи могут взаимодействовать с приложением также по-разному. Здесь так же, как и при определении целевой аудитории задействуется метод персон, определяются пользователи-персонажи и их характерное поведение и присущие им ожидания от ПП. Для каждого персонажа можно создать соответствующую карту пользовательского пути. Существует другое близкое понятие — карта пути/путешествия клиента Customer Journey Map / CJM — инструмент маркетинга, в общем

случае, для проектирования пути клиента от входа на сервис до совершения покупки.

Пользователи программного обеспечения как правило имеют определённые роли (в частности, обобщённая роль “Пользователь” или роль “Администратор”) и в зависимости от роли проходить определённый пользовательский путь. Есть различие между персонажем и ролью, персонаж — некий усреднённый портрет пользователя из определённого сегмента целевой аудитории, роль же, в общем случае, определяет основную цель использования, и также права доступа к функционалу программного продукта.

User story Mapping

Создание карты пользовательских историй (User story Mapping/ USM) позволяет комплексно представить основные аспекты программного продукта, определить и визуализировать план работы по реализации программного продукта. Данный инструмент предназначен для целостного проектирования программного продукта на основе пользовательского пути. По сути, USM представляет собой некую таблицу, в ней отражены роли пользователей, активности пользователей в приложении, задачи необходимые для обеспечения выполнения этих активностей, а задачи в свою очередь содержат свои подзадачи. Существуют различные подходы к составлению так называемого “скелета”, можно указывать роли и относительно них распределять активности, либо же роли можно не обозначать, а перечислить только активности (как правило, применяется в случае, если выделена одна роль).

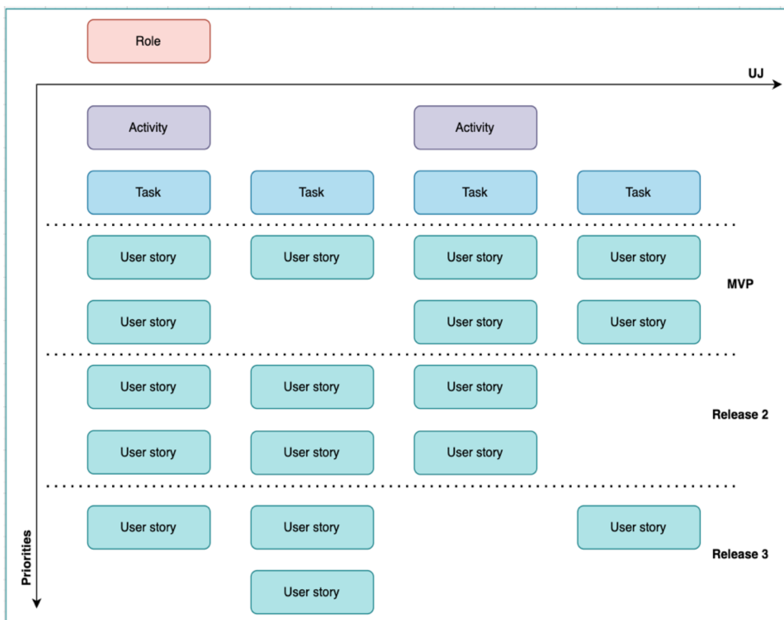

Рис. 7. — Карта пользовательских историй

Например, в системе работы с документами для редактирования некоего документа пользователь должен войти в систему, выбрать файл для редактирования, редактировать документ. Эти шаги, по сути, — активности доступные пользователю: “Вход в систему”, “Управление документами”, “Редактирование документа”. Каждая из активностей может включать одну или несколько задач, которые требуется решить для выполнения этой активности. В частности, для входа в систему, нужно перенаправить пользователя на форму ввода логина/пароля — и это перенаправление задача активности “Вход в систему”. Активность “Управление документами” включает в себя уже несколько задач, например, “Открыть документ”, “Найти документ”, “Отсортировать документы”, “Удалить документ” и т.д. И каждая задача в свою очередь имеет свои подзадачи, необходимые для выполнения задачи, например, “Удалить документ” может иметь такие подзадачи как: непосредственно “Удалить файл”, “Восстановить файл” (т.к. подразумевается работа с удалённым файлом, логически можно отнести эту подзадачу к задаче “Удаление файла”), “Безвозвратно удалить файл”. Подзадача — некое конкретное действие пользователя для достижения конкретной цели — пользовательская история.

Если рассматривать сервис доставки, здесь основной ролью является — клиент. Путь клиента можно описать как: найти товар, посмотреть фотографии, посмотреть описание, добавить товар в корзину, указать время и место доставки, ввести данные карты для оплаты, подтвердить покупку. Действия клиента могут быть сгруппированы в определённые этапы — задачи: так “посмотреть фотографии”, “посмотреть описание” могут быть сгруппированы в этап “ознакомиться с товаром”. Таким образом, Задача “ознакомиться с товаром” имеет две подзадачи: “посмотреть фотографии”, “посмотреть описание”. В описании/представлении товара могут быть также видео, отзывы покупателей и т.д. И они войдут в общий список подзадач, но по важности они будут иметь меньшее значение чем описание и фото. При этом подзадача представляет собой конкретную пользовательскую историю.

В первой версии продукта зачастую реализуются только те подзадачи, которые требуются для минимальной работоспособной версии продукта — MVP (minimum viable product/минимально жизнеспособный продукт). MVP позволяет создать прототип и получить обратную связь за достаточно короткий промежуток времени, внести требуемые коррективы добавить новые идеи и выпустить следующую версию, отвечающую запросам целевой аудитории. Важно отметить, MVP должен отражать изначальную идею программного продукта. Именно это позволит проверить гипотезу востребованности данного функционала/ данного программного продукта. Существуют и другие стратегии выбора задач для реализации в первой версии — MAP и RAT.

MAP (Minimum Awesome Product) — минимально привлекательный продукт, в данной стратегии предлагается некий уникальный функционал и собирается обратная связь, о том насколько востребован и интересен такой функционал целевой аудитории. Такая стратегия подходит в проектах B2B при наличии аналогов, но необходимо предлагать действительно уникальную и полезную бизнес-ценность.

RAT (Riskiest Assumption Test) — проверка рискованных гипотез, некий эксперимент, направленный на сбор обратной связи, не требует затрат на создание MVP. В частности, одним из методов применяемых при RAT используется анализ идеи по LMC.

Инструментарий для создания карты пользовательских историй

В базовом варианте может использоваться физическая расчерченная доска со стикерами. При организации удалённой работы команд могут использоваться различные онлайн-сервисы. Один из таких сервисов — miro.com. Данный сервис имеет бесплатную версию, но требует

регистрации. В бесплатной версии поддерживается множество шаблонов, в том числе и шаблоны для USM. Имеется как шаблон для карты содержащей роли пользователей, так и ориентированной только на активности.

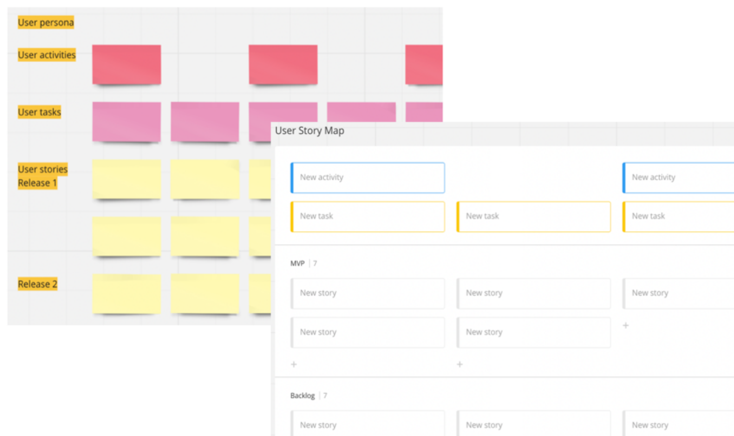

Рис. 8. — Шаблоны User Story Map в сервисе miro.com

Шаблон “User Story Map (Basic)” представляет собой электронный аналог доски со стикерами. “User Story Map Framework” имеет более широкий функционал, в данном шаблоне для каждой US можно указать сроки выполнения, исполнителей, теги и добавить ссылки при необходимости.

Для работы с шаблоном USM в сервисе miro.com требуется выполнить следующие шаги: войти, выбрать шаблон, пригласить участников команды, после чего можно перейти к совместному редактированию. Для организации совместной удалённой работы, как правило, требуется дополнительно использовать некий видеочат, в бесплатной версии эта опция не доступна, поэтому можно использовать любой удобный сторонний бесплатный сервис позволяющий организовать групповые видеозвонки.

Задание

1. Для планируемого программного продукта необходимо описать требования, используя US.
2. Сформировать UJM.
3. Провести USM, используя соотвествующий шаблон на платформе miro.com.
4. Защитить выполненную работу. Созданную USM экспортировать в формате PDF и прикрепить в эл. журнале.

Список источников

1. Jory MacKay «A Guide to User Story Mapping: Templates and Examples (How to Map User Stories)». — URL: https://plan.io/blog/user-story-mapping/ (дата обращения 01.02.2022).
2. The Ultimate Guide to Agile User Story Mapping. — URL: https://www.nimblework.com/agile/story-mapping/ (дата обращения 08.02.2022).
3. Mapping User Stories in Agile. — URL: https://www.nngroup.com/articles/user-story-mapping/ (дата обращения 08.02.2022).
4. Р. Баранов, Д. Кустов «User Story — инструкция по применению». — URL: https://scrumtrek.ru/blog/product-management/3364/user-story-instruktsiya-po-primeneniyu/ (дата обращения 08.02.2022).

2.3. Лабораторная работа «Проектирование»

Цель работы:

Выполнить описание объектной модели предметной области в UML-нотациях. На основании пользовательских историй выполнить макет интерфейса пользователя.

Форма проведения: Групповое выполнение задания.

Форма отчетности: Защита результата преподавателю: модель предметной области, макет и прототип интефейса программного продукта.

Теоретические основы

Декомпозиция и модель предметной области

Предметная область, в общем случае, — часть реального мира, подлежащая изучению и описанию по неким установленным заранее критериям.

В рамках анализа предметной области выделяют ее сущности, определяют первоначальные требования к функциональности и определяют границы проекта.

ПО создаётся для решения некой задачи. На уровне программной системы в целом задача всегда представлена достаточно крупно. Для решения задачу разбивают на более мелкие задачи, которые также могут быть разбиты на более мелкие. То есть каждая подзадача может быть достаточно сложна и в свою очередь требовать дальнейшего разбиения. То естьсложная задача может быть декомпозирована.

По типу элементов декомпозицию ПО можно разделить на:

- процедурную (алгоритмическую) основным элементом логической структуры являются процедура/функция/подпрограмма;
- объектно-ориентированную основным элементом логической структуры являются объекты конкретных классов.

Процедурная декомпозиция представляет ПО в виде процедур/функций и функциональных модулей, требуемых для решения поставленной задачи.

Выделяют три основных подхода процедурной декомпозиции:

- Нисходящее проектирование (метод «сверху-вниз») — выделяются элементы верхнего уровня, затем более мелкие и т.д.;
- Восходящее проектирование (метод «снизу-вверх») — на начальном этапе выделяются функции нижнего уровня, затем над ними “надстраивается” управление;
- Расширение ядра — на начальном этапе выбирается некоторое количество функций, которые формируют ядро, затем процесс идёт в двух направлениях: вниз, чтобы расширить функционал, и вверх, чтобы надстроить управление.

В рамках объектно-ориентированной декомпозиции создаётся модель предметной области. Как правило модель предметной области документируется и поддерживается в актуальном состоянии до этапа реализации. Одним из распространённых способов представления такой модели является использование UML-нотаций. UML (Unified Modeling Language — унифицированный язык моделирования) представляет собой язык объектного моделирования, использующий графические элементы для визуального проектирования и создания абстрактной модели разрабатываемой системы.

Построение логической модели системы в виде диаграммы классов является ключевым моментом в объектно-ориентированном подходе к проектированию ПО.

Диаграмма классов представляет собой статическую структурную модель проектируемой системы и является графическим представлением для взаимосвязей, независящих от времени. Для построения абстрактных моделей в UML используются сущности.

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

- концептуальный уровень (отображают связи между основными понятиями предметной области);

• уровень спецификаций (отображают связи объектов этих классов);

• уровень реализации (непосредственно показывают поля и операции конкретных классов).

Каждую из перечисленных моделей используют на конкретном этапе разработки программного обеспечения:

• концептуальную модель — на этапе анализа;

• диаграммы классов уровня спецификации — на этапе проектирования;

• диаграммы классов уровня реализации — на этапе реализации.

Диаграммы классов отражают различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывают их внутреннюю структуру и типы отношений. Выделяют следующие сущности: объекты, классы, интерфейсы, кооперации.

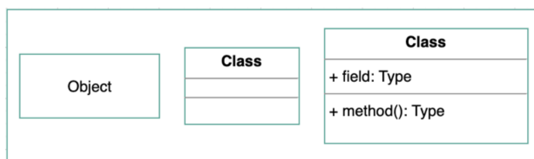

Рис. 9. — Диаграммы классов различных уровней абстракции

Взаимодействие между ними описывается через отношения зависимости, обобщения и ассоциации.

Объект представляет собой конкретную именованную сущность, обладающую свойствами и определенным образом проявляющую свое поведение. Объект является экземпляром класса. На диаграмме представляется в виде прямоугольника.

Класс (class) в UML служит для обозначения множества объектов, которые имеют одинаковую структуру, поведение и отношения с объектами из других классов. Как правило, на диаграмме класс изображают в виде прямоугольника, разделённого линиями на три секции горизонтальными линиями: имя класса, атрибуты класса, операции

класса. Имя класса является обязательным. В верхней секции отображается название класса, в средней секции отображается описание атрибутов, в нижней — отображается описание операций, выполняемых объектами данного класса.

Описание класса объектов для решения конкретной задачи производится в виде абстракции (выделены только значимые для задачи характеристики объекта).

Атрибуты класса представляют состав и структуру данных, которые хранят объекты класса. Для каждого атрибута указывается имя и тип содержащихся в нём данных. Имя атрибута — обязательный элемент обозначения атрибута, оно используется в качестве идентификатора соответствующего атрибута и является уникальной в пределах данного класса. Тип атрибута указывается текстовой строкой, имеющей осмысленное значение в пределах модели, к которой относится рассматриваемый класс. Тип атрибута указывается в зависимости от языка программирования, который будет использоваться для реализации данной модели.

Пример:

```
имя_студента String
```

имя_студента — имя атрибута,

String — тип атрибута.

Можно указать и начальное значение атрибута (на момент создания конкретного экземпляра класса): имя студента: String = Василий.

При программной реализации объекта выделяется память, которая необходима для хранения всех атрибутов и таким образом, каждый атрибут имеет конкретное значение в любой момент времени выполнения данной программы. У объектов одного класса (объект — экземпляр класса и в программе их может больше одного) одинаковый описанный в классе набор атрибутов, при этом значения атрибутов могут быть индивидуальными и изменяться в процессе выполнения программы.

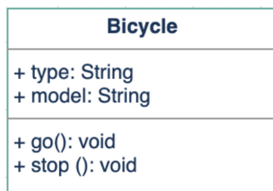

Рис. 10. — Класс «Bicycle»

Для ограничения доступа к определённым фрагментам кода используется **принцип инкапсуляции**, в частности используются модификаторы доступа (public, protected, private). Возможность скрыть внутренние данные и методы от внешнего кода позволяет изолировать детали реализации от пользователей класса. Так атрибутам класса можно присвоить видимость (visibility). Данная характеристика отображает степень доступности каждого атрибута для других классов. В языке UML определено несколько степеней видимости для атрибутов:

- (+) public — атрибут полностью открыт и доступен для других классов (объектов);
- (#) protected — атрибут открыт и доступен только для потомков данного класса;
- (-) private — атрибут недоступен для внешних классов (объектов).

Принцип наследования позволяет описать новый класс на основе существующего, и частично или полностью заимствовать его функциональность. Объекты могут быть связаны между собой между собой, например, объект класса Auto и объект класса Bicycle.

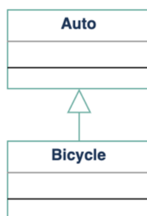

Рис. 11. — Наследование в UML нотации

Если представить классы объектов через множества таких объектов, то объекты класса Bicycle входят в множество объектов класса Auto, и

имеют схожее поведение, например, они могут передвигаться в пространстве. Класс-потомок полностью удовлетворяет спецификации родительского, полностью реализует интерфейс родительского класса. Но может иметь дополнительную функциональность, а также при необходимости в классе-потомке родительский метод может быть переопределён в зависимости от решаемой задачи. Методы класса-потомка имеют приоритет при выполнении (в случае если требуемый метод определён в классе, то будет использован соответствующий фрагмент кода, если нет, то требуемый метод будет вызван из класса-родителя). Так как наследование возможно только в направлении от родителя к потомку, то потомку доступны методы родителя, но родителю недоступны методы потомка, объект класса родителя не может к ним обратиться. Часть языков программирования поддерживает множественное наследование у одного класса-потомка несколько классов-родителей, такая организация может приводить к ошибкам при одинаковых именах методов у разных классов-родителей. Альтернативой является использование интерфейсов.

Полиморфизм в объектно-ориентированном программировании — возможность обработки разных типов данных, т. е. принадлежащих к разным классам, с помощью "одной и той же" функции, или метода. При этом идентично только имя, исходный код зависит от класса, соотвественно результаты работы одноименных методов могут существенно различаться.

Механизм “позднего связывания” позволяет определить принадлежность объекта конкретному классу, а также производить вызов метода, который имеет отношение к тому классу, объект которого был использован.

Принципы наследования и полиморфизма позволяют использовать объект подкласса может везде, где используется объект суперкласса. При вызове метода класса он ищется в самом классе, если данный метод существует в текущем классе, то вызывается он. Иначе выполняется обращение к родительскому классу и в нём производится поиск вызываемого метода. В случае, если в родительском классе поиск не дал результата, то продолжается обращение вверх по иерархии классов до самого верхнего уровня.

Помимо атрибутов в классе содержатся объявления операций, представляющие собой запросы, которые выполняются объектами данного класса. Таким образом, операция — абстракция того, что можно делать с объектом. Класс может содержать любое число операций или не

содержать ни одной операции. Набор операций, определяемых классом, является общим для всех объектов данного класса.

Для операции может быть указано имя операции (текстовая строка), возвращаемое значение и список параметров. Обязательным является имя операции, детальная спецификация операций может быть отложена на более поздние этапы моделирования. Для именования операций, как правило, используются глаголы, так как операции обозначают некое действие, соответствующие ожидаемому поведению объектов данного класса.

Для операции также задаётся видимость, она может быть задана как графически (+, #, -), так и соответствующим ключевым словом: public, protected, private. Операции, указанные как public, являются интерфейсной частью класса и представляют собой средство интеграции в программную систему.

При проектировании стоит учитывать, что каждый класс должен быть направлен на выполнение чего-то одного. Следует избегать слишком больших или слишком маленьких классов. В случае слишком больших классов увеличивается сложность модификации и изменения модели. В случае слишком маленьких классов увеличивается количество абстракций и, соответственно, усложняется логическое понимание и управление ими.

Модель представляет собой совокупность сущностей и взаимоотношений между ними. На диаграммах для описания логических связей между сущностями используются отношения. В UML существует несколько видов отношений: ассоциация, агрегация, композиция, обобщение, реализация, зависимость.

Ассоциация позволяет определить взаимосвязь объектов одной сущности (класса) с объектами другой сущности. Наиболее часто для связывания двух классов используются бинарные ассоциации. Для ассоциация может быть указанно название, отражающее суть данной связи. Так же ассоциация может обладать такой характеристикой, как множественность. Характеристика множественности ассоциации показывает количество объектов каждого класса имеющих возможность участвовать в ассоциации. Множественность отображается на концах ассоциации и представляется в виде конкретного числа или числового диапазона. Множественность может быть задана в виде звездочки, что означает любое количество объектов класса.

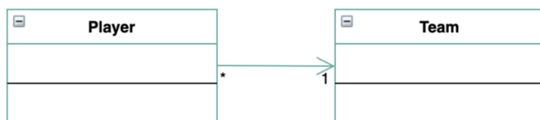

Рис. 12. — Ассоциация в UML нотации

Примеры:

- объект класса «Набор товаров» может быть связан с одним и более объектом класса «Товар»;
- объект класса «Букет» может быть связан с одним и более объектом класса «Цветок» и т.п.

Допускается связывание при помощи ассоциации объектов одного класса (например, для класса «Жилец дома» может быть задана ассоциация «Соседство», что позволит найти всех соседей конкретного жильца).

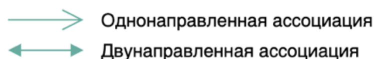

Рис. 13. — Типы ассоциаций

UML поддерживает несколько видов ассоциации. Наиболее распространены: однонаправленная («Рейс» — «Самолёт») и двунаправленная («Покупатель» — «Магазин»).

Однонаправленная ассоциация представляется в виде линии со стрелкой, указывающей направление. Двунаправленные ассоциации отображаются в виде линии с двумя концами, соединяющей два блока классов. Для ассоциации может быть указано имя и свойства (индикаторы, принадлежность, мультипликаторы и т.д.), отображаемые на на концах линии.

Агрегация является разновидностью ассоциации, определяющей отношение между целым и его частями. Агрегация не может определять отношение более двух классов (контейнер и содержимое).

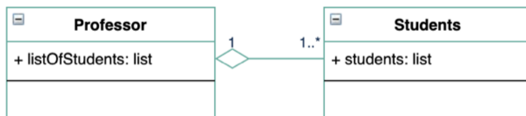

Рис. 14. — Агрегация в UML нотации

Для агрегации так же может быть указано имя. В UML агрегация изображается в виде пустого ромбика на блоке класса-контейнера и линией, связывающей этот ромбик с содержащимся классом.

Агрегация используется для описания отношения между классами в случае, когда один класс является коллекцией других классов. То есть один класс является контейнером, а другой является содержимым класса-контейнера. По умолчанию, агрегации присуще наименование “агрегация по ссылке”, при такой агрегации отсутствует зависимость времени существования содержащихся классов от содержащего их класса: если содержащий класс будет уничтожен, то его содержимое — нет.

Композиция более строгий вариант агрегации. Композиция изображается в виде закрашенного ромбика на блоке класса-контейнера и линией, связывающей этот ромбик с содержащимся классом. Композицию можно назвать “агрегацией по значению”, ей соответствует жесткая зависимость времени существования содержащихся классов от содержащего их класса: если содержащий класс будет уничтожен, то так же будет уничтожено и все его содержимое. Важным отличием от агрегации является использование мультипликаторов «0..1» или «1», так как необходимо отобразить, что часть является частью только одного целого. Примером агрегации может случить отношение «Дом» — «Квартира». Квартира является частью конкретного дома, и без дома квартира существовать не может. При этом имеется отношение «Квартира» — «Мебель», мебель находится в квартире, но не является её неотъемлемой частью, это отношение описывается уже агрегацией.

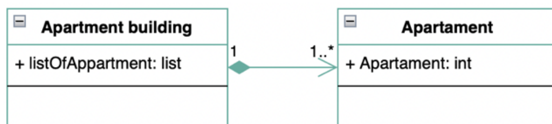

Рис. 15. — Композиция в UML нотации

Обобщение (наследование) используется для отображения связи между классом-родителем и классом-потомком. Данная связь используется в моделях, содержащих классы со сходным поведением. Обобщение заключается в том, что общие для классов элементы поведения выносятся на более высокий уровень, образуя класс-родитель. Обобщение отображается в виде линии с пустым треугольником у суперкласса (класса-родителя). Так класс «Фигура» является суперклассом для класса «Круг».

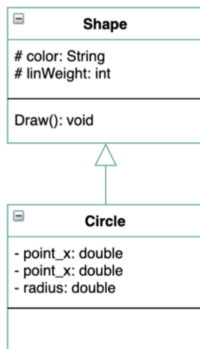

Рис. 16. — Наследование

Другой пример: класс «Животные» родительский класс для класса «Млекопитающие», который в свою очередь родительский класс для класса «Приматы». Взаимосвязь между такими классами можно описать фразой «А — это Б» (приматы являются млекопитающими, млекопитающие являются животными), также данное отношение называется взаимосвязь «is a» (от потомка к родителю).

В нотациях UML могут создаваться диаграммы различных уровней детализации, например, могут использоваться классы-ассоциации. Пример «Набор товаров» - «Товар» может быть детализирован следующим образом: обобщение показывает, что набор товаров также является товаром, который может быть предметом продажи, поставки, заказа и т.д. Класс «Опись» содержит список товаров, входящих в набор, класс-ассоциация «Включает» определяет количество каждого вида товаров в наборе.

Также используются отношения: реализация и зависимость. Отношение реализация описывает взаимосвязь класса-клиента, реализующего поведение, и классом-поставщиком, который поведение задаёт. В качестве класса-поставщика зачастую выступает интерфейс или

абстрактный класс. Реализация отображается стрелкой с пунктирной линией и пустым треугольником (по аналогии с обобщением).

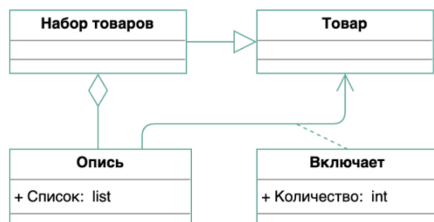

Рис. 17. — Отношения «реализация» и «Зависимость» в UML нотации

При помощи зависимости описывает отношение между двумя классами модели, при котором изменение одного класса приводит к изменению другого класса, обратная зависимость при этом не является обязательной. Зависимость отображается в виде пунктирной линии со стрелкой, связывающей зависимый элемент с тем, от которого он зависит. Зависимость может выражаться между экземплярами, классами или экземпляром и классом.

Как уже упоминалось в описании взаимоотношений классов используются **мультипликаторы**. Данная характеристика называется Мощность отношений или Кратность. Мощность отношений подразумевает число связей между каждым экземпляром класса в начале линии с экземпляром класса в ее конце.

Таблица 1. Мультипликаторы

Нотация	Описание	Пояснение (пример)
1	обязательно наличие одного экземпляра	у кошки одна мать
0..1	один или ни одного экземпляра	у кошки может быть хозяин или может не быть хозяина
1..*	один или более экземпляров	у кошки есть хотя бы одно место, где она ест
0..* или *	ноль или более экземпляров	у кошки могут быть котята, а может и не быть ни одного котёнка

Проектирование пользовательского интерфейса

Основная часть пользовательской истории — описание функциональности. Она должна отражать «Роль», «Действие», «Цель / ценность» (для одной роли, одно действие, одна ценность /ценность — определённая функциональность).

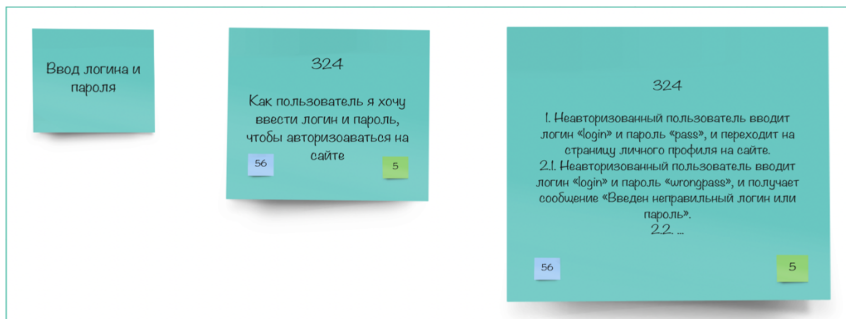

Рис. 18. — Представление пользовательской на различных этапах разработки

Помимо описания самой функциональности пользовательская история может содержать уникальный числовой идентификатор истории, он позволяет точно сказать, о какой конкретно пользовательской истории идет речь и обычно совпадает с идентификатором пользовательской истории в трекере задач.

Для пользовательской истории может быть задан приоритет, чем он выше, тем раньше история должна быть реализована.

Также истории добавляется оценка в стори поинтах, позволяющих оценить сложность её реализации. Стори поинты — условная величина, являющаяся весовым коэффициентом сложности реализации истории Чаще всего для оценки в Стори Поинтах (Story points) используются числа Фибоначчи 1, 2, 3, 5, 8, 13, как правило, числа больше 13 не задействуются.

В ряде случаев может применяться более подробное описание пользовательской истории (в частности, для большего удобства работы распределённых команд) — достаточно подробный сценарий, позволяющий провести демонстрацию пользовательской истории. Также может быть указана категория для повышения управляемости с помощью категоризации планируемых к выполнению задач.

Подробный сценарий, как правило, уже детализированная и готовая к реализации история, она показывает, как пользователь взаимодействует с системой. Сценарий по структуре напоминает вариант использования (Use case) — метод описания функционала, предложенный ещё в методологии RUP. Варианты использования сегодня уже не имеют массового применения, но многие компании по-прежнему работают с ними для детального описания.

Use case (Варианты использования/ВИ)

Вариант использования представляет собой некое формальное описание взаимодействия пользователя и системы при решении конкретной задачи. В качестве пользователя может выступать тот или иной вид пользователя системы, часть системы или другая система. Каждый ВИ направлен на достижение конкретной цели, на решение конкретной задачи и имеет соответствующее название, например, «Отправить заявку», «Сохранить файл» и т.д. Как правило, название ВИ имеет форму команды и начинается с глагола.

Проработанность и детальность ВИ зависит от стадии проекта. Выделяют следующие степени детальности ВИ:

Краткий ВИ (brief use case) содержит название, небольшое описание (как правило, пара предложений). Данный ВИ используется при планировании разработки, в том числе при планировании приоритетности, технической сложности функциональных требований к системе.

Бессистемный ВИ (casual use case) имеет более подробное и объёмное описание каким образом должна быть достигнута цель в сравнении кратким ВИ, не имеет строгой структуры изложения.

Детальный ВИ (detailed use case/fully dressed use case) представляет собой формальный документ строгой структуры (может иметь различную степень детализации, при этом, как и в какой последовательности приводится тот или иной аспект строго определено). Как правило, именно детальный ВИ подразумевается при упоминании варианта использования.

В рамках разработки в зависимости от тех или иных особенностей проекта схема написания ВИ может варьироваться, но типичной схемой можно назвать следующую:

1. Название — краткое понятное описание действия (как правило, содержит глагол).

2. Цель — краткая характеристика задачи решаемой в результате выполнения ВИ.
3. Начальное состояние — формально описывает условия необходимые для инициализации ВИ.
4. Основной сценарий — последовательность шагов, выполняемых в программе для достижения цели ВИ.
5. Альтернативный сценарий/сценарии, например сценарии ошибок последовательность шагов при возникновении ошибки на одном из шагов выполнения основного сценария.

Помимо этого, можно указать контекст использования, область действия, уровень, основное действующее лицо, участников и их интересы, предусловие, минимальные гарантии, триггер, расширения, список изменений в технологии и данных.

Формат описания варианта использования Fully dressed use case:

1. Имя — цель в виде краткой активной глагольной фразы.
2. Контекст использования — описание цели.
3. Область действия.
4. Уровень точности.
5. Основное действующее лицо.
6. Другие участники и их интересы.
7. Предусловие определяет, выполнение какого условия гарантирует система перед тем, как разрешить запуск варианта использования.
8. Минимальные гарантии — то, что система гарантирует участникам, в том числе, в случае если цель основного действующего лица не может быть достигнута.
9. Постусловие устанавливает, что интересы участников удовлетворяются по успешном завершении варианта использования.
10. Триггер — событие, запускающее выполнение варианта использования.
11. Основной сценарий или поток.
12. Расширения содержат последовательность шагов, описывающих, что происходит при возникновении определённого условия (расширения могут представлять собой отдельный ВИ).

13. Список изменений в технологии и данных.

14. Вспомогательная информация.

Варианты использования в формате fully dressed use case встречаются не часто, такой ВИ достаточно объёмный и не всегда удобен.

В процессе разработки ВИ требуемым образом детализируются от краткого до детального (в требуемом объёме). Так, например, при рассмотрении ВИ “Сохранить файл” для текстового редактора, на определённом моменте можно указать предусловие, начальное для выполнения данного ВИ состояние системы — “Введён текст/В сохранённый текст внесены изменения”.

При написании ВИ одним из главных критериев является лёгкость понимания ВИ, для её достижения можно варьировать подробность и ограничено точность ВИ. Трудно понимаемый ВИ считается плохим ВИ и должен быть переписан.

Пример ВИ:

Название

Создать учётную запись Пользователя

Цель/краткое описание

Система создаёт новую учётную запись для Пользователя

Начальное состояние (здесь можно обозначить тригер и предусловие):

Пользователем выбрана опция “Создать учётную запись”.

Основной сценарий:

1. Система просит ввести данные: адрес эл. почты, пароль, подтверждение пароля.

2. Система проверяет корректность введённых данных:

   2.1. Система проверяет формат ввдённого адреса эл. почты.

   2.2. Система проверяет совпадают ли пароль и подтверждение пароля.

3. Система создаёт учётную запись пользователя.

Альтернативный сценарий: Введён некорректный адрес эл. почты (возникает на шаге 3 п. 3.1)

Система сообщает Пользователю, что он ввел некорректный адрес эл. почты.

Альтернативный сценарий: Пароль и подтверждение пароля не совпадают (возникает на шаге 3 п. 3.2)

Система сообщает Пользователю, что пароль и подтверждение пароля не совпадают.

Альтернативный сценарий: Отмена (возникает на шагах 1-2)

Пользователь отменяет создание учётной записи.

Постусловие: новая учётная запись для Пользователя не создана.

Из приведённого примера видно, что любое действие для выполнения ВИ совершается кем-то, здесь Пользователем или Системой. Если использовать при написании только действие, например, как «сообщение отправляется Пользователю», появляется некая размытость, что ведёт к нечёткому пониманию, в частности, к появлению разного видения. При этом всякое действие не совершается само по себе, действие совершается кем-то. Поэтому при написании ВИ рекомендуется каждый шаг начинать с подлежащего, указывающего исполнителя — кто/что выполняет действие. При описании начального состояния или постусловия не требуется указывать исполнителя.

Данный ВИ может быть описан и менее формально (Fowler style, подход к описанию ВИ, предложенный Мартином Фаулером):

Создание учётной записи Пользователя

Основной сценарий

1. Пользователь выбирает опцию “Создать учётную запись”
2. Система просит ввести данные: адрес эл. почты, пароль, подтверждение пароля.
3. Система проверяет корректность введённых данных: проверяет корректность указанного адреса эл. почты, проверяет совпадают ли пароль и подтверждение пароля.

4. Система создаёт учётную запись пользователя.

Отклонение создания учётной записи

1. На шаге 3 Система определяет введённый адрес эл. почты как некорректный
2. Пользователь получает сообщение о том, что введён некорректный адрес эл. почты.

или

1. На шаге 3 Система определяет, что введённые пароль и подтверждение пароля не совпадают.
2. Пользователь получает сообщение о том, что введённый пароль и введённое подтверждение пароля не совпадают.

Отмена

Пользователь отменяет создание учётной записи.

В данном ВИ не описывается, как именно пользователю отправится сообщение. В данном ВИ этого и не требуется, “Отправить сообщение Пользователю” — отдельный ВИ.

Также на начальных этапах анализа ВИ не должны содержать привязки к деталям интерфейса, например, “Пользователь выбрал пункт меню Файл — Сохранить”. С развитием проекта ВИ могут детализироваться, включать уже технические детали, в частности, после того как спроектирован интерфейс пользователя, ВИ могут быть значительно уточнены.

ВИ могут быть использованы на всех этапах работы над проектом:

Анализ требований

ВИ используются для определения и фиксации требований к ПП, определения границ на этапе анализа (основное назначение).

Проектирование и реализация

ВИ служит источником информации о постановке задачи для выполнения процессов проектирования и реализации.

Тестирование и отладка

ВИ могут использоваться для тестирования, на основе ВИ могут быть написаны сценарии тестирования, в рамках которых проверяется, выполняет ли система то, что предусмотрено ВИ.

Документирование

ВИ могут быть применены при написании пользовательской документации (руководства пользователя). Основная часть документации посвящена тому как взаимодействовать с системой для решения тех или иных задач, поэтому в основу такого описания могут быть положены ВИ, так как они как раз и описывают взаимодействие с системой для решения конкретных задач.

Макет пользовательского интерфейса

Имея представление о том, как должно осуществляться взаимодействие пользователя с программы обеспечением, можно спроектировать интерфейс пользователя. Важным этапом является создание макета (wireframe), на котором будут отображены требуемые элементы интерфейса, wireframe создаётся с упором не на визуальную составляющую, а на структуру и содержание. Его создают на начальном этапе работы над проектом, чтобы составить общую картину будущего проекта.

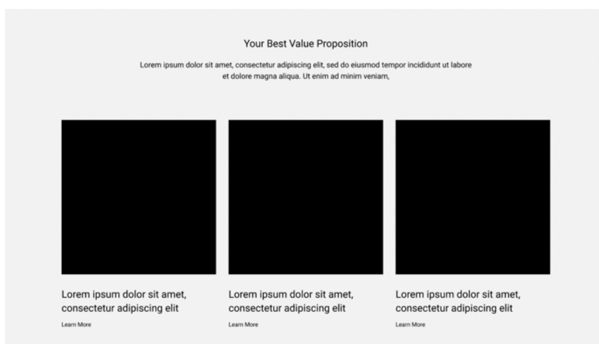

Рис. 19. — Figma Wireframe Kit

Wireframe — скелет дизайна, и поэтому должен изображать все важные детали финального программного продукта. Он является неким

аналогом технического задания на создание дизайн-макета или прототипа пользовательского интерфейса. Как правило, при создании wireframe применяются всего несколько цветов, например, черный, серый, голубой и белый цвета. Внешне он выглядит как набор различных прямоугольных блоков и линий. В этих блоках и стрелочках заложена структура продукта и порядок взаимодействия пользователя с ним. По сути wireframe — образ дизайн-макета низкой точности, его основной задачей является показать:

- что будет расположено на сайте/в приложении, какой контент;
- как будет структурирована информация и где будет отображаться;
- базовую визуализацию взаимодействия между интерфейсом и пользователем (без интерактива).

Wireframe является довольно быстрым инструментом, позволяющим относительно быстро спроектировать интерфейс и быстро вносить правки на различных этапах обсуждения. При создании wireframe, если что-либо занимает слишком много времени на подготовку (например, выбор иконок, загрузка изображений), этот элемент должен быть представлен в упрощенном виде, например, может быть заменён на «заполнитель» (placeholder).

После того как макет готов можно переходить к разработке дизайн-макета —визуального образ продукта, отображающего расположение и характеристики всех элементов, блоков и структуры в целом. В дизайн-макете отсутствуют интерактивные элементы и анимация, что упрощает его разработку. В то же время позволяет наглядно представить дизайн будущих страниц в отличие от блочной схемы исходного прототипа. Дизайн-макет отражает размеры, фон и цветовое оформление, количество и расположение блоков, границы и отступы между блоками и элементами, дизайн элементов интерфейса. Если в wireframe отражена суть взаимодействия, то в дизайн-макете интерфейс представлен визуально, таким каким его увидит пользователь.

Задание

1. При проектировании объектной модели предметной области необходимо определить классы их атрибуты и методы. Для их определения необходимо проанализировать описание предметной области. Как правило, сущность в описании предметной области соответствует существительному, являющемуся основным в неком рассматриваемом контексте. Глаголы, характеризующие действия в

рамках контекста, могут быть преобразованы в методы класса. Атрибутами класса могут выступать существительные, упомянутые в контексте, но не являющиеся в нём основными.

Например:

> Сформировать **заказ**, рассчитать его вес, стоимость и стоимость доставки.

В данном контексте основным является «заказ», при этом «вес», «стоимость», «стоимость доставки» — характеристики заказа и, соответственно, в объектной модели они станут атрибутами класса «Заказ». Глагол «рассчитать», относящийся к характеристикам заказа, может быть преобразован в соответствующие методы класса: «Рассчитать вес», «Рассчитать стоимость», «Рассчитать стоимость доставки».

Для создания UML-диаграммы в рамках данной лабораторной работы будет использоваться сервис draw.io.

Необходимо выбрать, куда Вы будете сохранять полученный документ:

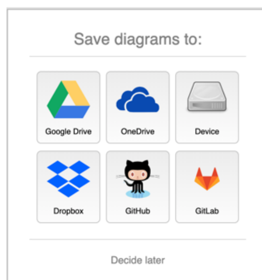

Рис. 20. — Сохранение документа draw.io.

a. Для создания UML-диаграммы необходимо выбрать и развернуть раздел «UML», содержащий нотации UML:

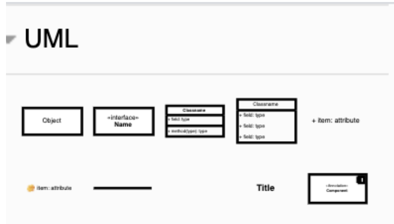

Рис. 21. — UML нотации в draw.io

b. Используя требуемые элементы (класс, ассоциация, агрегация, композиция, обобщение), создать UML-диаграмму предметной области.

2. Для создания макетов в данной лабораторной работе будет использоваться сервис figma.com.

a. Необходимо зарегистрироваться на сервисе.

b. Создать команду (Create new team +) и добавить участников.

Name your team

TeamName

After creating a team, you can invite others to join.

Create team

Рис. 22. — Назначение имени команде в Figma

Add your collaborators

You can update user permissions on the team page after setting up.

Email

Email

Email

Add another

Рис. 23. — Приглашение участников команды в Figma

c. Создать макет пользовательского интерфейса (wireframe) программного продукта (см. Figma Wireframe Kits, а также Figma Community).

d. На основании wireframe создать дизайн-макет.

Список источников

1. Блинов И.М., Романчик В.С. Java. Промышленное программирование: практ. пособие — Минск: УниверсалПресс, 2007. — 704 с.

2. Вольфсон Б. И. Гибкие методологии разработки / Б. И. Вольфсон — СПб.: Питер, 2012 — 112 с.

3. Гагарина Л.Г. Технология разработки программного обеспечения. Учеб. пос. / Л.Г.Гагарина, Е.В.Кокорева, Б.Д.Виснадул; Под ред. проф. Л.Г.Гагариной — М. ИД ФОРУМ НИЦ Инфра-М, 2013. — 400 с.

4. Гради Буч Язык UML. Руководство пользователя : пер. с англ. / Гради Буч, Джеймс Рамбо, Ивар Якобсон — ДМК Пресс, 2007 — 496 с.

5. Диаграммы классов UML. Логическое моделирование. — URL: 08.02.2022, свободный.

6. Иванова Г.С. Технология программирования М.: Изд-во МГТУ имени Баумана, 2002

2.4. Лабораторная работа «Внедрение системы контроля версий»

Цель работы:

Внедрение в проект разработки программного обеспечения системы контроля версий.

Форма проведения: Индивидуальное выполнение (регистрация аккаунта на GitHub и создание локального репозитория) и групповое выполнение задания (настройка удалённого репозитория).

Форма отчетности: Защита результата преподавателю: обоснование выбора рабочего процесса в git демонстрация работы с репозиториями, устный опрос по базовым командам git (ссылку на репозиторий GitHub прикрепить в эл. курсе индивидуально каждым участником подгруппы).

Теоретические основы

Существует несколько подходов к работе с системой контроля версий и организации ведения разработки, такие как использование функциональных веток, Gitflow, GitHub flow.

Выполнение задания

Часть 1

Рассмотреть предложенные подходы к организации работы в системе контроля версий git и выбрать один из них (выбор необходимо обосновать).

Часть 2

Создать локальный и удалённый репозиторий на GitHub. Создать требуемые процессом ветки. Создать и разрешить конфликт при слиянии изменений.

1. Создать аккаунт на GitHub
   a. login/password
   b. подтвердить эл. почту
   c. создать новый репозиторий

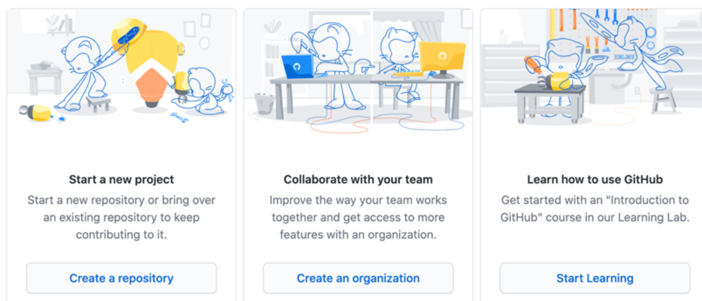

Рис. 24. — Создание репозитория на GitHub

Для проекта требуется один репозиторий, соответственно, создаёт репозиторий один из участников команды (требуется создать публичный репозиторий).

Owner * Repository name *
spekarskaya / proj-repo
Great repository names are proj-repo is available. Need inspiration? How about fantastic-umbrella?
Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.
Add a README file
This is where you can write a long description for your project. Learn more.
Add .gitignore
Choose which files not to track from a list of templates. Learn more.
Choose a license
A license tells others what they can and can't do with your code. Learn more.

Рис. 25. — Настройки создаваемого репозитория на GitHub

После того как репозиторий создан GitHub предлагает выполнить следующие шаги:

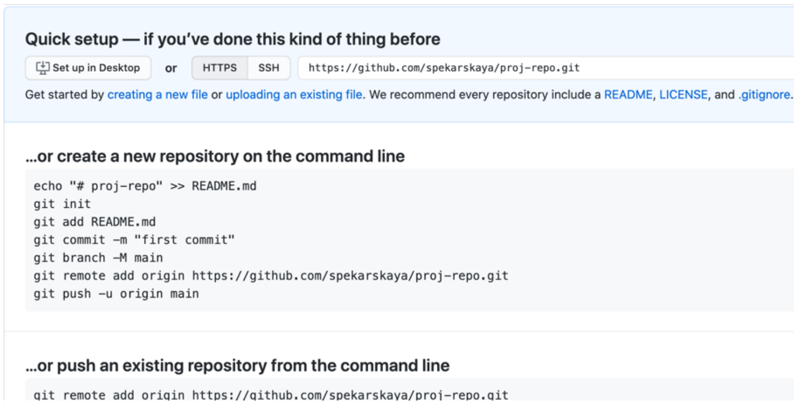

Рис. 26. — Инструкция по созданию репозитория на GitHub

1. До выполнения данных шагов необходимо создать локальную директорию проекта: mkdir proj-repo и перейти в неё cd proj-repo. Далее можно переходить к предлагаемым шагам.

2. echo «# proj-repo» >> README.md данная команда запишет строку «# proj-repo» в конец файла README.md (если использовать «>», то файл будет перезаписан). После можно выполнить команду ls, чтобы просмотреть содержимое директории:

   ```
   mo-2:proj-repo svetlana$ ls
   README.md
   ```

   Для просмотра внесённых изменений используется команда git diff.

3. git init инициализирует данную директорию как репозиторий

```
    Initialized empty Git repository in /Users/User/Documents/proj-repo/.git/
```

При работе с git следует задать определённые настройки, в частности указать имя и эл. почту. Для этого необходимо выполнить команду *git config user.email «user@company.com».* Эта информация будет использоваться для подписи коммитов и отправки изменений в удаленный репозиторий. Имеется три уровня хранения настроек: ну уровне системы, на уровне пользователя, на уровне проекта/репозитория. Настройки в данной работе стоит указать локально для конкретного репозитория.

```
Config file location
```

```
--global        use global config file
--system        use system config file
--local         use repository config file
```

4. *git add* README.md данная команда добавляет файл в staging area.

   Staging area содержит файлы проекта, которые находятся под версионным контролем. Только добавленные файлы будут входить в коммит.

Проект как правило содержит разнообразные файлы и не все они требуют версионного контроля. Для того чтобы определённые файлы или директории были исключены из верстанного контроля (не вносились в коммиты) используется файл. gitignore с перечислением шаблонов, соответствующих таким файлам. В общем случае к такого рода файлам относят:

- Каталоги зависимостей (# /vendor, /node_modules и т.д);
- Build каталоги, такие как (# /public или /dist и т.д);
- Runtime файлы (# log, cache, временные файлы (tmp) и т.д);
- Файлы, содержащие конфиденциальную информацию (# ключи API, пароли и т.д);
- Системные файлы (# .DS_Store или Thumbs.db);
- Различные конфигурационные файлы (# для IDE, текстового редактора и т.д).

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются.
- Можно использовать стандартные glob шаблоны.
- Можно заканчивать шаблон символом / для указания каталога.
- Для того, чтобы инвертировать шаблон используют восклицательный знак ! в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ * соответствует 0 или более символам; последовательность [abc] — любому

символу из указанных в скобках (в данном примере a, b или c); знак вопроса (?) соответствует одному символу; [0-9] соответствует любому символу из интервала (в данном случае от 0 до 9).

5. После того как файл был добавлен в staging area можно посмотреть статус командой *git status*:

```
On branch master

No commits yet

Changes to be committed:

  (use "git rm --cached <file>..." to unstage)

      new file:   README.md
```

Из вывода видно, что на данный момент не было выполнено ни одного коммита, а файл README.md добавлен в Changes to be committed и приведена команда для исключения файла. Так же указано, что это new file, т.е те изменения, которые произошли в репозитории.

6. Следующим шагом необходимо выполнить *git commit -m "first commit"*:

```
Committer: User <email@.local>

 1 file changed, 1 insertion(+)

 create mode 100644 README.md
```

-m позволяет написать сообщение к коммиту сразу, если указать git commit без неё, будет открыт редактор для ввода сообщения (по умолчанию vim), где в первой строке необходимо ввести сообщение к коммиту (для того, чтобы перейти в режим редактирования необходимо выполнить команду i). Для сохранения сообщения в vim используется команда w, для выхода их редактора q, соотвественно, команда wq позволяет сохранить изменения и выйти из редактора.

Также можно использовать *git commit -am*, -am позволяет выполнить комбинацию команд *git add* и *git commit -m*.

7. Историю коммитов для текущей ветки можно просмотреть при помощи *git log*:

```
commit 16a9819fa5a1584f59dd3144dd21ac87eab13a10 (HEAD -> main)
```

```
Author: User <email@.local>
Date:   Mon Mar 15 21:14:32 2021 +0700

    first commit
```

По умолчанию, без аргументов, git log выводит список коммитов созданных в данном репозитории в обратном хронологическом порядке: от последнего к первому, то есть самые последние коммиты показываются первыми.

8. Команда *git branch -M main* создаёт основную ветку с именем *main.* При помощи команды git branch -l можно просмотреть, какие ветки

```
mo-2:proj-repo svetlana$ git branch -l
* main
```

9. Далее указывается ссылка на удалённый репозиторий git remote add origin https://github.com/spekarskaya/proj-repo.git, то есть добавляется удалённый репозиторий.

```
main
remotes/origin/HEAD -> origin/main
```

10. Командой git push -u origin main изменения в локальном репозитории вносятся в удалённый репозиторий.

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 228 bytes | 228.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/spekarskaya/proj-repo.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

11. В удалённом репозитории можно перейти в раздел “Code” в нём будут показаны, в частности, произведённые комиты.

12. Добавить пользователей в проект можно Settings - Manage access - Invite a collaborator.

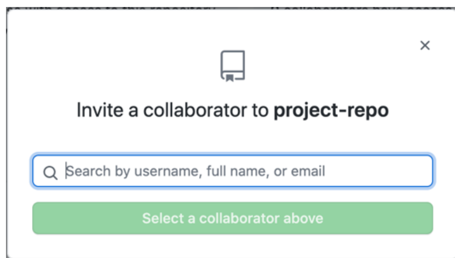

Рис. 27. — Добавление пользователей в проект

13. Создать локальную копию удалённого репозитория можно при помощи git clone:

```
Cloning into 'proj-repo'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

По умолчанию будет создана соответствующая директория в текущей.

13. Для работы над изменениями изолированно используется механизм ветвления. Ветки можно создавать локально при помощи *git branch*, можно создавать в удалённом репозитории.

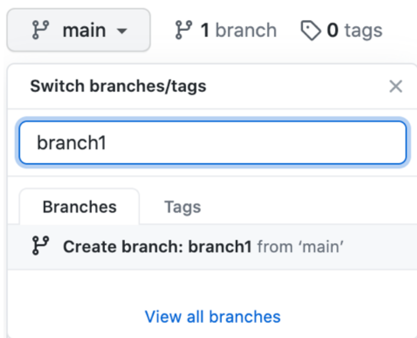

Рис. 28. — Создание ветки

14. Для того, чтобы получить изменения из удалённого репозитория для отслеживаемой ветки и слить их с локальной текущей используется команда *git pull*.

15. Для получения данных проекта, которых в локальном репозитории ещё нет используется *git fetch*.

```
From https://github.com/spekarskaya/proj-repo
 * [new branch]        branch1      -> origin/branch1
```

   Если после выполнения git fetch выполнить команду git branch -a (с опцией -a данная команда выводит все имеющиеся ветки: локальные и удалённые).

```
* local1
  main
  origin/main
  remotes/origin-1/main
  remotes/origin/branch1
  remotes/origin/main
```

16. В случае необходимости перейти с одной ветки на другую используется команда *git checkout <branch name>*:

```
Branch 'branch1' set up to track remote branch 'branch1' from 'origin'.
Switched to a new branch 'branch1'
```

17. После того как работа над конкретными изменениями закончена локально закончена эти изменения необходимо и влить в основную в требуемую ветку. При отправке изменений в удалённый репозиторий, как упоминалось раньше, используется git push.

   Для внесения изменений в ветку в удалённом репозитории требуется создать pull request для этого зайти в «Code», выбрать branches и для требуемой ветки нажать «New pull request».

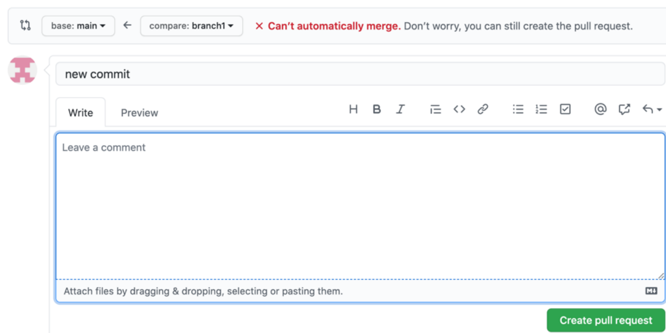

Рис. 29. — Создание ветки pull request

При этом могут возникнуть конфликты, в случае если внесены изменения в один и тот же фрагмент в разных ветках и изменения одной из них уже слиты с основной, а другая ветка о внесённых в основную ветку изменениях «не знает».

Сообщение о наличии конфликта выглядит следующим образом.

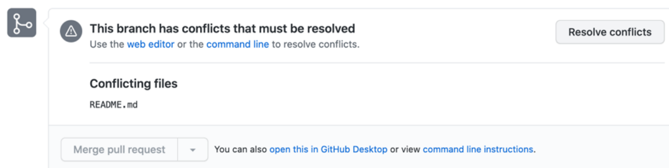

Рис. 30. — Сообщение о конфликте при слиянии веток

Для разрешения конфликта можно использовать web editor или command line. По ссылке command line откроется инструкция по разрешению конфликта через командную строку.

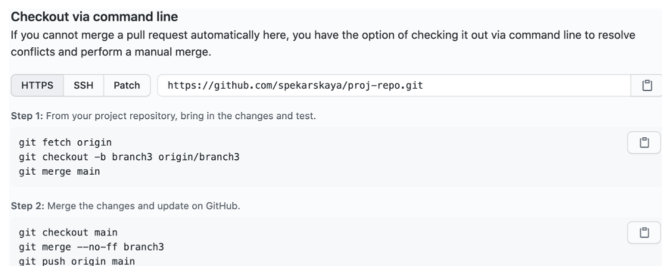

Рис. 31. — Инструкция по разрешению конфликта через командную строку

При выборе web editor соответственно откроется редактор, в котором отображён конфликт. В редакторе содержимое файла необходимо привести к требуемому, финальному виду. После чего данный конфликт необходимо отметить, как разрешённый (Mark as resolved) и выполнить Merge pull request.

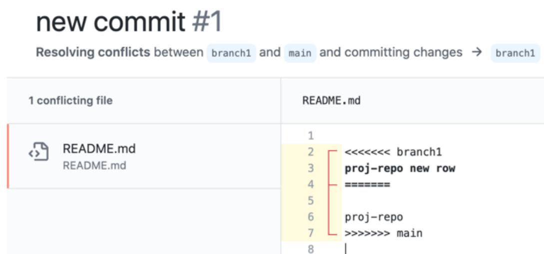

Рис. 32. — Разрешение конфликта в web editor

Часть 3

Самостоятельно изучить команды:

- Команда git add
- Команда git commit
- Команда git log
- Команда git diff
- Команда git push
- Команда git branch
- Команда git checkout
- Команда git pull
- Команда git fetch
- Команда git reset / git revert
- Команда git merge
- Команда git rebase (-i)
- Команда git rebase squash
- Отличие git merge и git rebase
- Команда git cherry-pick
- Команда git tag
- Что такое remote? (git remote/ git remote add origin)
- Команда git clone
- Команда git stash (pop, apply, drop)

Список источников

1. git. — URL: https://git-scm.com/ (дата обращения 08.02.2022).
2. Git Feature Branch Workflow. — URL: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow (дата обращения 08.02.2022).
3. Gitflow Workflow. — URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (дата обращения 08.02.2022).
4. GitHub flow. — URL: https://docs.github.com/en/get-started/quickstart/github-flow (дата обращения 08.02.2022).
5. How to Select a Git Branch Mode? — URL: https://www.alibabacloud.com/blog/how-to-select-a-git-branch-mode_597255 (дата обращения 08.02.2022).

2.5. Лабораторная работа «Реализация front-end приложения»

Цель работы:

Разработка клиентской части приложения.

Форма проведения: Групповое выполнение задания (реализация распределённых в команде задач).

Форма отчетности: Защита результата преподавателю: демонстрация клиентской части приложения, свёрстанной согласно разработанному ранее дизайн-проекту и содержащей всю необходимую функциональность для выполнения пользовательских историй (со стороны клиентской части приложения), демонстрация работы в репозитории на GitHub.

Теоретические основы

Вёрстка веб-страниц

В разработке сайтов и веб-приложений вёрсткой называется перевод дизайн-макетов в интерактивный, читаемый браузерами вид. То есть, при вёрстке создаётся код, который формирует из предоставленного графического шаблона веб-страницу, с элементами которой может работать пользователь. Минимально вёрстка требует применения языка разметки, например, HTML (HyperText Markup Language), и описания стилей элементов при помощи CSS (Cascading Style Sheets).

HTML относится к формальным языкам, его теги (см. # https://www.w3schools.com/tags/default.asp) и их иерархическая структура жестко описаны в спецификации. С определённого момента стал набирать популярность XML (см. # https://www.w3schools.com/xml/default.asp) — расширяемый язык разметки, позволяющий создавать собственные теги и формировать их структуру. Браузер при работе с HTML пропускает разные мелкие ошибки, такие как недочеты в структуре, не корректные указания атрибутов и т.д. XML имеет более строгий синтаксис, что повышает качество кода. Так как HTML был основным, для получения преимуществ XML и сохранения стандарта HTML в целом, был создан промежуточный вариант XHTML (см. # https://www.w3schools.com/html/html_xhtml.asp), по сути, это HTML, соблюдающий синтаксис XML. Применение XHTML позволило снизить вероятность возникновения ошибок в коде, что повысило общее качество гипертекстового документа и уровень вебразработок в целом. Отмечается, что за счёт строгого

синтаксиса проще создавать кроссбраузерный код, а также увеличить быстродействие и простоту обработки документа, что позволяет корректно работать на устройствах с малыми вычислительными мощностями.

Первым широко распространившимся видом вёрстки стала табличная вёрстка. Суть этого метода верстки в том, что весь сайт представляет собой одну большую таблицу. То есть все содержимое должно быть заключено в один парный тег — table. В одну таблицу можно вложить неограниченное число других элементов, в том числе и других таблиц. Информация в таблицах выводится только в ее ячейках. Они, в свою очередь, располагаются в строках. Ячеек в одной строке может быть негараниченное количество. К плюсам табличной верстки можно отнести возможность создания так называемого резинового макета посредством задания ширины в процентах и регулирования высоты ячеек при помощи различных настроек. И также кроссбраузерность, т.к. теги для табличных данных появились очень давно и поддерживаются большинстве даже совсем старых версий браузеров, таким образом сайт на таблицах одинаково выглядит в разных браузерах. Среди минусов можно назвать, конечно же, большой объём кода, что в свою очередь ведёт к высокой сложности редактирования такой разметки. Тяжёлые таблицы снижают скорость загрузки, в случае медленного соединения это будет заметно. Из-за большой разметки и обилия вложенных тегов текст на странице хуже воспринимается поисковыми системами. Обилие вложенных тегов также снижает эффективность применения стилей, для их применения может понадобиться значительно увеличить код. Когда-то данный подход был наиболее подходящим, но с развитием технологий появились более удобные способы организации страницы. И таблицы теперь в основном используют для представления именно табличных данных.

На смену пришла блочная вёрстка, достаточно широко и сегодня. При таком подходе базовый шаблон страницы создаётся при помощи так называемых блоков, обозначаемых тегом <div>. Данный тег изначально универсален, и в нём могут быть расположены различные элементы. Суть блочной вёрстки состоит в следующем, в графическом редакторе создаётся макет сайта: размечается, где какая область страницы будет находиться и сколько места занимать, подготавливаются фоны и изображения. Каждая часть страницы помещается в свой блок <div>: верх сайта — в первый, меню — во второй, контент — в третий и т. д. Каждый блок наполняется содержимым средствами HTML, а также позиционируется и оформляется с помощью CSS-разметки. В большинстве случаев блоки являются независимыми друг от друга, за счет чего отдельные блоки могут добавляться или удаляться в макете веб-

страницы. Конечный HTML-документ представляет собой набор блоков <div> с контентом внутри. В данном подходе регламентируется чёткое разделение содержимого и оформления (физически в разных файлах), таблицы используются в основном по прямому назначению. Поисковыми системами данный вид вёрстки проще и лучше индексируется лучше, её код менее объёмный.

С HTML5 появилась возможность семантической вёрстки. HTML5 является единым языком разметки, который сочетает в себе синтаксические нормы HTML и XHTML. Благодаря новым синтаксическим особенностям и элементам он расширяет и рационализирует разметку документов, а также обогащает семантическое содержимое документа. В данном подходе для структурирования html-документов используются теги, которые разделяют код на логические блоки (явно показывающие их роль и содержание в структуре web-страницы). Выделяют четыре группы тегов: теги структуры документа, текстовые теги, медиа теги (описывают тип медиа-контента на странице: видео, аудио, изображение), корреляционные теги (используются для связи между элементами, например нумерованные и маркированные списки, где требуется определять определённый порядок).

Данный тип вёрстки даёт большее преимущество, так как семантический элемент четко описывает свое значение, как для браузеров, так и для разработчиков, делает страницы более удобными для правильной индексации в поисковых системах, повышает доступность для разнообразных устройств.

Каскадные таблицы стилей (Cascading Style Sheets)

Каждый элемент каким-либо образом оформлен для представления на странице, идентичные элементы могу иметь идентичное оформление. И, соответственно, оформление каждого элемента по отдельности ведёт к увеличению издержек, уходит больше времени, создаётся больший объём кода. Решением стало применение каскадных таблиц стилей CSS (Cascading Style Sheets — каскадные таблицы стилей) — технология описания внешнего вида документа, оформленного языком разметки, позволяющая разделить содержимое (HTML или другом языке разметки) и оформление документа (собственно CSS), что может увеличить доступность документа, предоставить большую гибкость и возможность управления его представлением, а также уменьшить сложность и повторяемость в структурном содержимом (см. # https://www.w3schools.com/css/default.asp, https://developer.mozilla.org/en-US/docs/Web/CSS). Использование CSS предполагает, что веб-страница будет содержать только теги логического форматирования, а внешний вид

элементов будет определяться через стили. Такой подход позволяет вести работу над дизайном и вёрсткой параллельно.

CSS позволяет представлять один и тот же документ в различных стилях или методах вывода, таких как экранное представление, печать или при выводе устройствами, использующими шрифт Брайля, например.

Стиль — совокупность правил, применяемых к элементу гипертекста и определяющих способ его отображения, включает в себя все типы элементов дизайна: шрифт, фон, текст, цвета ссылок, поля и расположение объектов на странице.

Таблица стилей — совокупность стилей, применяемых к веб-странице.

Каскадирование — порядок применения различных стилей к веб-странице. Поддерживающий таблицы стилей браузер будет последовательно применять их в соответствии с приоритетом: связанные, внедренные, встроенные стили.

Метод встраивания (Inline) позволяет применять стиль к заданному тегу HTML (встроенный стиль применяется к любому тегу HTML с помощью атрибута style. Такие таблицы являются внутренними). Внедрение (Embedded) позволяет управлять стилями страницы целиком (используется тег <style>, который обычно размещают в заголовке HTML-документа <head>...</head>). Связывание (Linked или External) позволяет вынести описание стилей во внешний файл.

Каскадирование поддерживает наследование, т.е. конкретный стиль будет применен ко всем дочерним элементами гипертекстового документа, если не указано иное. Например, при применении определённого цвет текста в теге <div>, все теги внутри этого блока будут отображаться этим же цветом.

Использование стилей достаточно удобный и эффективный инструмент для работы при вёрстке веб-страниц и оформления текста, ссылок, изображений и других элементов. Разделение разметки и оформления позволяет применять единое оформление к различным документам, просто обращаясь к таблицам стилей. Централизованное хранение стилей позволяет достаточно просто вносить требуемые изменения, внесённые изменения будут отображены везде, где используются соответсвующие CSS, т.е. достаточно отредактировать стили и оформление нужных документов сразу же поменяется требуемым образом.

Стоит отметить, что хранящиеся в отдельном файле стили, как правило, кэшируются и при повторном обращении к нему извлекается из кэша браузера, при этом код самой страницы меньше, что позволяет увеличить скорость загрузки.

Вариативность устройств

Для корректной работы на различных устройствах страницы должны быть адаптивны, т.е страница должна выглядеть одинаково хорошо на всех устройствах (# ноутбук, принтер, планшет, смартфон и т.д.). Адаптивность может быть достигнута различными способами, среди них можно выделить использование медиа-запросов и flexbox-контейнеров, CSS Grid Layout.

В CSS3 была введена поддержка аппаратно-зависимых таблиц стилей, позволившая создавать стили и таблицы стилей для определенных типов устройств. Были определены возможные медиа типы для разных типов устройств и дополнительно ключевое слово all, чтобы указать, что таблица стилей применяется ко всем типам носителей. В общем случае медиа-запрос состоит из ключевого слова (тип устройства) и выражения, проверяющего характеристики устройства. Данные запросы могут использоваться при необходимости применять разные CSS-стили в зависимости от типа отображения (# для принтера, монитора и т.д.), а также конкретных характеристик устройства (# ширина окна просмотра в браузере), или внешней среды (# внешнее освещение). Медиа-запросы могут быть комбинированными, для этого используются соотвествующие логические операторы and, or, not, only. Стоит отметить, что оператор not применяется ко всему медиа-запросу сразу, т.е. применятся соответствующая таблица будет, если характеристики противоположны указанным в запросе.

CSS flexbox — модуль макета гибкого контейнера — представляет собой способ компоновки элементов. Основной целью flexbox является предоставление возможности изменения своих элементов по ширине и высоте для того, чтобы они максимально эффективно умещались в доступном месте родительского контейнера, в частности — это удобно в тех случаях, когда нужно соответствовать всем типам дисплеев устройств и размерам экранов. Flex-контейнер расширяет вложенные элементы для того, чтобы заполнить доступное пространство или же урезает их, чтобы избежать переполнения. Flexbox состоит из гибкого контейнера (flex container) и гибких элементов (flex items). Гибкие элементы могут выстраиваться в строку или столбец, а оставшееся свободное пространство распределяется между ними различными способами. В основе лежит идея расположения элементов по осям в одном из 4-

направлений слева направо, справа налево, сверху вниз или снизу вверх. Flexbox позволяет переопределять порядок отображения элементов, автоматически определять размеры элементов таким образом, чтобы они вписывались в доступное пространство, позволяет переносить элементы внутри контейнера, не допуская его переполнения, создавать колонки одинаковой высоты и многие другие.

CSS Grid Layout — двухмерная система компоновки на основе сетки. Grid представляет собой наборы горизонтальных и вертикальных линий, пересекающихся между собой — один набор определяет столбцы, а другой строки. Т.е. элементы (grid items) помещаются в сетку, соответственно, по строкам и столбцам и располагаются вдоль главной/основной (main) и поперечной (cross) осей. Grid содержит функции выравнивания, чтобы мы могли контролировать, как элементы выравниваются после размещения в области сетки, и как выравнивается весь Grid. Строки и столбцы Grid Layout могут объединяться между собой как строки или столбцы электронных таблиц. При помощи различных свойств ими можно манипулировать для создания требуемых макетов (все свойства в Grid делятся на родительские (свойства Grid Container) и дочерние (свойства Grid Item)). В ячейку сетки может быть помещено более одного элемента, или области могут частично перекрывать друг друга. Grid Layout может применяться в комбинации CSS flexbox, что позволяет создавать хорошо адаптивные макеты.

Стоит отметить, что в третьем поколении каскадных таблиц (CSS3) стилей для указания размеров объектов используются не пиксели, а проценты. Это в свою очередь упрощает адаптивную вёрстку.

Кроссбраузерность

Корректное отображение зависит от многих факторов, с частью из них можно справиться с помощью рассмотренных адаптивных подходов. Но помимо различных типов и параметров устройств, есть еще и браузеры, представляющие основную часть ПО для отображения. Исторически сложилось так, что они имеют значимые отличия, в частности в обработке представления. Все браузеры соблюдают общие правила и стандарты, но алгоритмы обработки html-кодов и каскадных таблиц CSS у них могут разниться. И поэтому не всегда один и тот же элемент выглядит одинаково в разных браузерах. Свойство сайта идентично работать и отображаться различных браузерах, во всех требуемых браузерах получило название кроссбраузерности. Под идентичностью в данном случае понимается отсутствие расхождений в верстке и способность отображать материал с одинаковой степенью читабельности.

Для корректного отображения в разных браузерах используют разные подходы. Существует подход, основанный на использовании универсальных элементов, тех, что отображаются в большинстве браузеров одинаково. Такой подход делает код более простым. Другим довольно распространённым подходом является использование CSS hacks - наборов специальных селекторов или правил, понимаемых только каким-то определенным браузером. При таком подходе необходимо для всех требуемых браузеров прописать соотвествующие фрагменты кода. При малом числе браузеров (версий браузеров) этот подход приемлем, при большом их числе это может быть неудобно. Использование CSS hacks делает код не эстетичным и не валидным при этом код является рабочим.

Также можно использовать специальные вендорные префиксы, в зависимости от движка лежащего в основе браузера. Префикс в данном случае представляет собой приставку к названиям CSS-свойств, обеспечивающую поддержку браузерами, в которых определенная функция ещё не внедрена на постоянной основе, например, находится в стадии разработки или тестирования, свойство написано для конкретного браузера, и оно не содержится в стандартном списке свойств или свойство реализует частичный функционал. Они имеют смысловую нагрузку только для тех браузеров, к которым они относятся, и дают возможность браузеру воспринимать не стандартные свойства, а также не воспринимать те стили, которые предназначены для других пользовательских клиентов. Отмечается, что благодаря появлению вендорных префиксов большинство компаний производящих свои браузеры стали внедрять собственные свойства CSS. Работа с префиксами выглядит следующим образом: для элемента прописывается CSS свойство для браузеров, которые его понимают, далее через точку с запятой перечисляется то же самое свойство, но с разными вендорными префиксами для разных браузеров. Браузер из такого кода интерпретирует только то свойства, которое написано под него, а написанные для других браузеров игнорирует.

Методологии вёрстки

С определённого момента появилась концепциями что всё есть компонент: кнопка- компонент, заголовок - компонент, страница в целом тоже компонент. Т.е. под компонентом в общем случае понимается самостоятельный и независимый участок разметки со своей логикой и стилями. У компонента есть свое текущее состояние. В рамках этой концепции совершенствовалась и CSS - технология, как уже говорилось в основе применения каждой технологии лежит некая методология, определяющая её применение. В данном случае методологии

регламентируют подходы и правила создания кода разметки. Выделяют следующие методологии БЭМ, OOCSS, SMACSS, Atomic CSS, MCSS, AMCSS и т.д.

Одной из наиболее популярных является БЭМ, название представляет собой аббревиатуру «Блок, элемент, модификатор». Блок — логически и функционально независимый компонент страницы, доступный для использования в нескольких местах сайта; элементы части блока, не имеют функционального смысла вне блока, элементы могут быть обязательными и опциональными; модификаторы представляют собой свойства блока или элемента, которые меняют его внешний вид, состояние или поведение. Использование модификаторов опционально. У блока/элемента может быть несколько разных модификаторов одновременно.

OOCSS означает объектно-ориентированный CSS (Object-Oriented CSS). В этот подход заложены две основные идеи: разделение структуры и оформления и разделение контейнера и содержимого. С помощью такой структуры разработчик получает общие классы, которые можно использовать неоднократно использовать (принцип DRY), но возникают сложности поддержки, так при изменении стиля конкретного элемента скорее всего придется менять не только CSS (т.к. большинство классов общие), но и добавлять классы в разметку. Развитием этой методологии стали другие среди них, например, Atomic CSS, где регламентируется, что всё что имеет повторное использование должно быть оформлено в класс.

Язык программирования

На стороне клиента приложение обеспечивает интерфейс и определённую функциональность. Для создания интерфейса используется язык разметки гипертекста, например, HTML, и каскадные таблицы стилей CSS. Для обеспечения функциональности на стороне клиента, в частности для динамического обновления контента, управления мультимедиа и анимацией, используются языки сценариев, такие как JavaScript.

JavaScript — прототипно-ориентированный, интерпретируемый язык с динамической типизацие. Данный язык является также мультипарадигменным, он подходит как для императивного (объектно-ориентированное программирование), так и декларативного (функциональное программирование) стилей программирования. JavaScript реализует стандарт для скриптовых языков — ECMAScript. На сегодняшний день в веб-разработке JavaScript используется как на стороне клиента, так и на стороне сервера (# Node.js).

Клиентский JavaScript-код может встраиваться в HTML-документы различными способами:

- встроенные сценарии в теге <script></script>;
- из внешнего файла, заданного атрибутом src тега <script>;
- в обработчик события, заданный в качестве значения HTML-атрибута (# onclick, onmouseover и т.д.);
- как тело URL-адреса, использующего специальный спецификатор псевдопротокола JavaScript.

На сегодняшний день для клиентской часть довольно часто используются и други языки скриптовые программирования, в частности TypeScript.

Application Programm interface (API) браузера

В целом, на стороне клиента существует множество различных API. В первую очередь это API браузера. Среди наиболее востребованных можно назвать:

- API для работы с документами, например, DOM (Document Object Model) API. Данный API позволяет динамически создавать, изменять, удалять различные элементы документа.
- API, принимающие данные от сервера, зачастую применяются для обновления частей веб-страницы, для того чтобы сделать страницу более отзывчивой и не перезагружать страницу целиком. К такого рода API относят XMLHttpRequest и Fetch API.

Document Object Model (DOM)

Объектная Модель Документа (Document Object Model) описывает стандарт (W3C DOM и WHATWG) для осуществления доступа к документу. DOM описывает структуру документа, позволяет организовать доступ элементам документа из программы, как к частям этой структуры, что в свою очередь позволяет программно изменять содержимое документа, стили его элементов и его структуру. DOM не зависит от конкретного языка программирования и обеспечивает структурное представление документа согласно единому API, и, следовательно, реализация DOM может быть создана для любого языка.

DOM поддерживает объектно-ориентированное представление веб-страницы, и позволяет различным языкам сценариев, таким как JavaScript, изменять страницу путём манипуляции объектами документа. DOM может применяться при работе с:

HTML (# см. JavaScript HTML DOM),

XML (# см. XML DOM),

SVG (# см. SVG Document Object Model).

Всё описанное стандартом поддерживается браузерами, но браузеры предоставляют и дополнительные расширения, неподдерживаемые стандартами, и поддержку того или иного дополнительного расширения необходимо уточнять для каждого конкретного браузера.

DOM HTML

Веб-страница — документ, который может быть представлен в HTML-коде и в интерпретированном представлении в браузере. DOM, по сути, предоставляет ещё один способ представления, хранения и управления документа.

Объекты организованы в виде DOM-дерева, каждый объект представляется узлом DOM-дерева. Для работы с различными элементами страницы в DOM поддерживаются соотвествующие интерфейсы (# см. DOM Living Standard, MDN Web Docs "Document Object Model (DOM)").

HTML

```
<!DOCTYPE html>
<html lang="en">
<body>

<header>
    <h1>
        Pallet "Green colors"
    </h1>
    <!--comment-->>
    <p>
        Color name: RGB / HEX
    </p>
</header>

    </body>
</html>
```

DOM

```
├DOCTYPE: html
└HTML lang="en"
  ├HEAD
  └BODY
    ├#text:
    ├HEADER
    │ ├#text:
    │ ├H1
    │ │ └#text: Pallet "Green colors"
    │ ├#text:
    │ ├#comment: comment
    │ ├#text: >
    │ ├P
    │ │ └#text: Color name: RGB / HEX
    │ └#text:
    └#text:
```

Рис. 33. — Представления веб-документа в HTML (слева) и в виде DOM (справа) / *DOM создана при помощи Live DOM Viewer*

Каждый элемент в документе — весь документ в целом, заголовок, блоки внутри документа, таблицы, текст внутри элементов и т.д. представляют собой части объектной модели документа. Она организуется в виде дерева, каждый уже которого представляет собой объект. На основании такой модели можно получить доступ к элементам страница и взаимодействовать с ними при помощи языка программирования.

В полученной DOM отражены все элементы страницы, указанные в HTML-коде, помимо основных отражены также перенос каретки, пробелы и комментарии. В DOM-дереве до элемента body перенос каретки и пробелы игнорируются, после представляются пустыми текстовыми узлами, для комментариев используется отдельный тип узлов — #comment.

Просмотр DOM возможен, в частности, при помощи инструментов разработчика в браузере.

Навигация по DOM-дереву

Основной объект, образно говоря «точка входа в приложение» — document. Он представляет собой веб-страницу в целом и отвечает за глобальную функциональность, в частность используется для создания и удаления элементов в документе, для получения URL-адреса страницы и т.д. Также используется объект window, он представляет собой окно, содержащее DOM-документ.

Навигация по DOM-дереву осуществляется следующим образом:

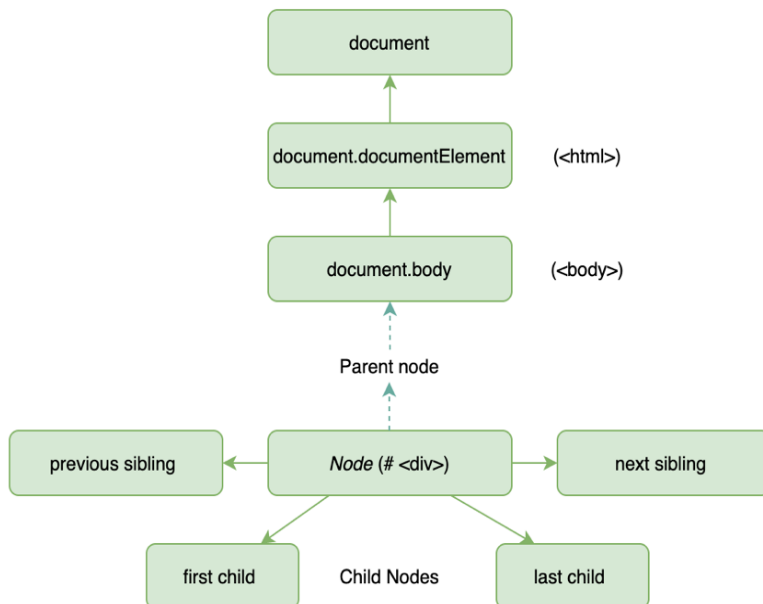

Рис. 34. — Навигация в DOM-дереве

<u>Узлы DOM-дерева</u>

Узлы DOM-дерева представлены различными тегами и соответственно имеют различные свойства. Имеют соответствующее различие и по типу текстовые узлы и узлы-элементы. Но все они имеют общие свойства и методы, так как все классы DOM-узлов образуют единую иерархию. Каждый DOM-узел принадлежит соответствующему встроенному классу, реализующему определённый интерфейс.

Корневым является EventTarget, представляет собой «абстрактный класс» и служит основой для поддержки «событий» DOM-узлами. Не возможности создать объект данного класса.

От него наследует Node, также «абстрактный» класс, являющийся основой для DOM-узлов. Node обеспечивает базовую функциональность. Не возможности создать объект данного класса. Есть классы узлов, которые наследуют от него, к ним относят:

Text – для текстовых узлов (опосредованно, через CharacterData),

Element – для узлов-элементов (опосредованно, через CharacterData),

Comment – для узлов-комментариев.

Неполная иерархия классов узлов DOM-дерева может выглядеть следующим образом:

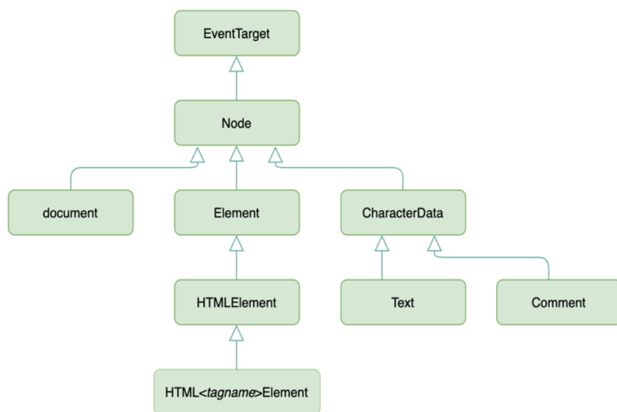

Рис. 35. — Фрагмент иерархии DOM-дерева

Для различных тегов существуют наследующие классу HTMLElement классы: например, HTMLTextAreaElement, HTMLInputElement, HTMLTableElement и т.д.

Взаимодействие с элементами документа

Для взаимодействия используются соотвествующие интерфейсы, так для работы с документом в целом используются window, document, для взаимодействия с элементами страницы соотвественно Element и т.д.

Среди наиболее распространённых свойств и методов для работы с элементами страницы можно назвать:

- для document (# см. DOM "Interface Document", MDN Web Docs "Document")

- document.getElementById(id)

- document.getElementsByTagName(name)
- document.createElement(name)

- для Node и ParentNode (# см. DOM "Interface Node", MDN Web Docs "Node", MDN Web Docs "ParentNode", w3schools.com "HTML DOM appendChild() Method")

- parentNode.appendChild(node) и ранее используемый Node.appendChild()

- для element (# см. DOM "Interface Element", MDN Web Docs "Element", w3schools.com "Element")

- element.innerHTML
- element.style.свойство
- element.setAttribute
- element.getAttribute
- elem.querySelectorAll(css)
- elem.matches(css)
- elem.closest(css)
- elem.getElementsByTagName(tag)
- elem.getElementsByClassName(className)
- element.addEventListener

- для window (# см. MDN Web Docs "Window")

- window.content
- window.onload

Использование библиотек и фреймворков

Библиотеки JavaScript содержат пользовательские функции, в целом использование библиотек упрощает разработку, в частности позволяет использовать готовое и проверенное решение. Среди них можно назвать jQuery, React.js, Mootools.

На сегодняшний день существует множество различных инструментов фротенд-разработки для решения широкого круга задач, охватывающих все этапы ЖЦ разработки. Это различные библиотеки, фреймворки, в том числе специфичные инструменты такие как

менеджеры пакетов, загрузчики модулей, CSS-пре и постпроцессоры, средства автоматизации сборки и компиляции, разработаны инструменты тестирования клиентского кода, а также инструменты, обеспечивающие качество кода.

JavaScript фреймворки в целом представляют собой уже не только библиотеку функций, но комплекс технологий разработки, в частности обеспечивают работу с HTML, CSS, JavaScript и т.д., что позволяет используя только фреймворк создать приложение. На сегодняшний день фреймворки представляют собой целые экосистемы для разработки. В частности, React включает в себя пакет react-dom для работы с DOM, React Router для маршрутизации язык разметки JSX, различные вспомогательные библиотеки, такие как Redux для управления состоянием приложений или Axios для обмена данными с серверными API.

Статистику по использованию инструментов разработки можно посмотреть в отчёте The 2021 Stack Overflow Developer Survey.

Список источников

1. A Complete Guide to Grid. — URL: https://css-tricks.com/snippets/css/complete-guide-grid/ (дата обращения 08.02.2022).
2. A Complete Guide to Flexbox. — URL: https://css-tricks.com/snippets/css/a-guide-to-flexbox/ (дата обращения 08.02.2022)
3. CSS Flexible Box Layout . — URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout (дата обращения 08.02.2022).
4. CSS Tutorial. — URL: https://www.w3schools.com/css/default.asp (дата обращения 08.02.2022).
5. DOM-дерево. — URL: https://learn.javascript.ru/dom-nodes (дата обращения 08.02.2022)
6. HTML. — URL: https://developer.mozilla.org/ru/docs/Web/HTML (дата обращения 08.02.2022).
7. HTML Versus XHTML. — URL: https://www.w3schools.com/html/html_xhtml.asp (дата обращения 08.02.2022).
8. HTML5 Semantic Tags: What Are They and How To Use Them! — URL: https://www.semrush.com/blog/semantic-html5-guide/ (дата обращения 08.02.2022).

9. JavaScript HTML DOM. — URL: https://www.w3schools.com/js/js_htmldom.asp (дата обращения 08.02.2022).
10. Media queries. — URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries (дата обращения 08.02.2022).
11. Media Queries. — URL: https://www.w3.org/TR/css3-mediaqueries/#syntax (дата обращения 08.02.2022).
12. Object-Oriented CSS. — URL: http://oocss.org/ (дата обращения 08.02.2022).
13. Shadow tree. — URL: https://dom.spec.whatwg.org/#shadow-trees (дата обращения 08.02.2022).
14. SVG Document Object Model (DOM). — URL: https://www.w3.org/TR/SVG11/svgdom.html (дата обращения 08.02.2022).
15. UI Events. — URL: https://www.w3.org/TR/DOM-Level-3-Events/#event-type-focus (дата обращения 08.02.2022).
16. Using media queries . — URL: https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries (дата обращения 08.02.2022).
17. What is XHTML? — URL: https://www.w3.org/TR/xhtml1/#xhtml (дата обращения 08.02.2022).
18. XHTML — Краткое руководство. — URL: https:// coderlessons.com/tutorials/veb-razrabotka/vyuchit-xhtml/xhtml-kratkoe-rukovodstvo (дата обращения 08.02.2022).
19. Быстрый старт/ Методология/ БЭМ. — URL: https:// ru.bem.info/methodology/quick-start/ (дата обращения 08.02.2022).
20. Виртуальный DOM и детали его реализации в React. — URL: https://ru.legacy.reactjs.org/docs/faq-internals.html (дата обращения 08.02.2022).
21. Навигация по DOM-элементам. — URL: https://learn.javascript.ru/dom-navigation (дата обращения 08.02.2022)
22. Руководство по DOM. — URL: https://developer.mozilla.org/ru/docs/Web/API/Document_Object_Model (дата обращения 08.02.2022).

2.6. Лабораторная работа «Реализация back-end приложения»

Цель работы:

Разработка серверной части приложения.

Форма проведения: Групповое выполнение задания (реализация распределённых в команде задач).

Форма отчетности: Защита результата преподавателю: демонстрация серверной части приложения, содержащей всю необходимую функциональность для выполнения пользовательских историй (со стороны серверной части приложения), демонстрация работы в репозитории на GitHub

Теоретические основы

Использование виртуального окружения

Для организации изолированного виртуального окружения python (virtual environment) можно использовать virtualenv (при помощи pip — package installer for Python).

Для создания виртуального окружения при помощи virtualenv необходимо, перейдя в директорию локального репозитория выполнить virtualenv nameVE. В результате будет создана директория с указанным именем и поддериректориями /bin, /lib и конфигурационным файлом pyvenv.cfg. Далее можно созданное изолированное окружение активировать (перейти в него), для этого необходимо выполнить source nameVE/bin/activate. В созданном изолированном окружении можно установить необходимое ПО, в частности Django требуемой версии (также используя pip).

Начиная с версии Python 3.4 вместе с ним идёт пакет venv, выполняющий те же функции — создание виртуальных окружений Python. Для создания используется соответствующая команда `python3 -m venv /path/to/new/virtual/environment`.

Такой подход может быть удобен для работы с несколькими проектами, требующими, например, различных версий тех или иных библиотек, для каждого из которых может быть создано своё виртуальное окружение.

Django-проект

Находясь в локальном репозитории необходимо создать Django-проект: `django-admin startproject` ***`projname`***. В репозитории будет создана директория с соответствующим именем — *projname* (имя может быть любым, за исключением совпадающих с именами директорий Django проекта). Имя *projname* можно изменить при необходимости, оно является внешним и не является значимым для Django. Внутри будут созданы следующие файлы:

- manage.py: утилита командной строки для взаимодеqствия с Django (см. django-admin and manage.py).
- внутренняя директория с таким же именем *projname* представляет собой Python package для созданного проекта, это имя значимо и не должно изменяться.
- *projname*/__init__.py: инициализационный файл, указывающий, что это директория для Python package (см. more about packages).
- *projname*/settings.py: настроийки и конфигурация для Django проекта (см. Django settings)
- *projname*/urls.py: содержит объявления URL для Django проекта (“table of contents” of your Django-powered site, см. URL dispatcher).
- *projname*/asgi.py: кофигурация для использовния ASGI (см. How to deploy with ASGI).
- *projname*/wsgi.py: кофигурация для использовния WSGI (см. How to deploy with WSGI).

Структура Django-приложения

Django проект в общем случае представляет собой совокупность приложений и конфигурации сайта. В данном проекте может быть добавлено одно или несколько Web-приложений, представляющих собой некий определённый функционал. Таким образом, в одном проекте может быть несколько приложений, а одно приложение может использоваться в нескольких проектах. Для создания приложения необходимо выполнить команду: `python manage.py startapp` ***`appname`***.

Django при создании приложения создаёт его базовую структуру директорий. Django-приложение использует паттерн MVT (Model-View-Template), в созданной директории содержатся соотвествующие файлы models.py и views.py.

Для корректной обработки запросов приложением необходимо создать конфигу-рационный файл URL, так называемый URLconf, с именем urls.py в директории проекта.

```python
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

Рис. 36. — Фрагмент конфигурационного файла

Обработка запроса в Django-приложении происходит следующим образом (при поступлении запроса приложение определяет какой Python-код необходимо выполнить):

1. Django определяет используемый корневой модуль URLconf, как правило это значение параметра *ROOT_URLCONF* (параметр задаётся в файле settings.py, расположенном в директории проекта: ROOT_URLCONF = 'stdproj.urls', где stdproj.urls, указание на файл urls.py находящийся в директории Django-проекта). Корневой модуль URLconf может быть задан через атрибут urlconf входящего HttpRequest-объекта (устанавливается промежуточным программным обеспечением), и в этом случае будет использоваться значение атрибута вместо параметра ROOT_URLCONF.

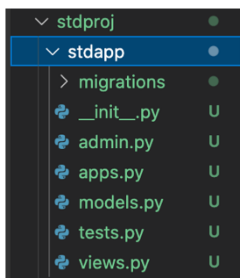

Рис. 37. — Структура Django-приложения

2. Django загружает этот модуль Python и ищет переменную urlpatterns.
3. Django проходит по каждому шаблону URL по порядку и останавливается на первом соответствующем запрошенному URL-

адресу шаблону (на основании значения атрибута path_info HttpRequest-объекта).

4. При совпадении шаблона URL Django импортирует и вызывает заданное View. Это может быть функция Python или class-based view. Django предоставляет подходящие для широкого круга приложений классы базовых представлений. Все представления наследуются от класса View, который позволяет обрабатывать связывание представления с URL-адресами, осуществлять отправку HTTP-запросов с соответствующими методами и т.д.
5. При возникновении ошибки на одном из шагов должна быть возвращена соответствующая ошибка.

Запуск Django development server

Для запуска Django development server необходимо перейти в директорию проекта: `cd` ***`projname`*** и выполнить команду `python manage.py runserver`. Команда должна выполняться именно из директории проекта, т.к. утилита manage.py находится там, вне её утилита недоступна.

```
System check identified no issues (0 silenced).
April 02, 2021 - 04:33:40
Django version 3.1.7, using settings 'stdproj.settings'
Starting development server at http://127.0.0.1:8000/
```

Рис. 37. — Результат запуска Django development server в консоли

По умолчанию команда runserver запускает сервер на порту 8000. Номер порта можно задать в качестве аргумента команды.

Открыв URI http://127.0.0.1:8000/ в браузере, можно убедиться, что сервер запущен:

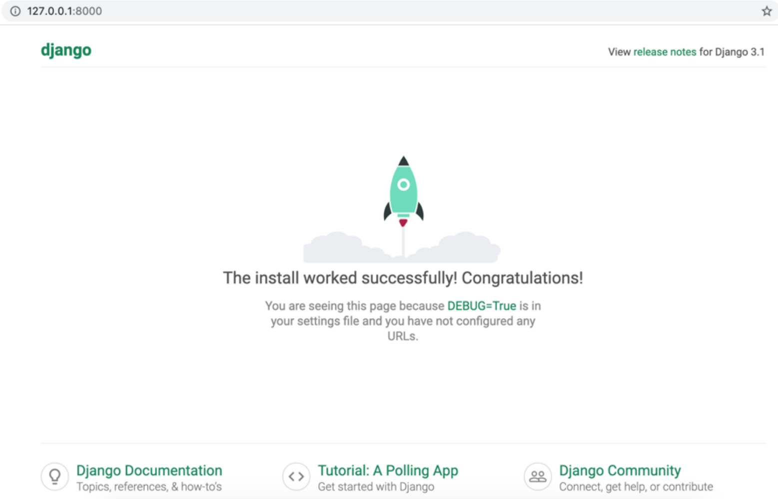

Рис. 38. — Результат успешного запуска Django development server в консоли

При этом при обращении к http://127.0.0.1:8000/ в консоли будут вводиться соответствующие логи.

```
[02/Apr/2021 04:34:43] "GET / HTTP/1.1" 200 16351
[02/Apr/2021 04:34:44] "GET /static/admin/css/fonts.css HTTP/1.1" 200 423
[02/Apr/2021 04:34:44] "GET /static/admin/fonts/Roboto-Bold-webfont.woff HTTP/1.1" 200 86184
[02/Apr/2021 04:34:44] "GET /static/admin/fonts/Roboto-Regular-webfont.woff HTTP/1.1" 200 85876
[02/Apr/2021 04:34:44] "GET /static/admin/fonts/Roboto-Light-webfont.woff HTTP/1.1" 200 85692
```

Рис. 39. — Логи запуска Django development в консоли

Из логов видно, что при запуске сервера выполняются определённые HTTP-запросы с методом GET для получения требуемых данных.

Добавление приложения в проект

Для добавления созданного приложения в проект необходимо указать его конфигурационный класс в переменной INSTALLED_APPS в файле настроек проекта settings.py:

1. В файле *appname*/apps.py декларируется конфигурационный класс приложения с именем

2. В файле проекта settings.py в переменной INSTALLED_APPS указываются все приложения проекта, при добавлении нового приложения, необходимо добавить в данную переменную ссылку на его конфигурационный класс, путь указывается через точку:

<u>База данных</u>

По умолчанию Django использует SQLite, она включена в проект и не требует установки.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Рис. 40. — База данных Django-приложения

При необходимости использовать другую СУБД необходимо в файле настроек проекта settings.py в DATABASES изменить значение на требуемое и задать дополнительные параметры: USER, PASSWORD, HOST, PORT (см. DATABASES).

Django поддерживает:

- PostgreSQL (ENGINE : 'django.db.backends.postgresql'),
- MySQL (jango.db.backends.mysql),
- Oracle (django.db.backends.oracle).

Также может быть использован ряд других СУБД, в частности Microsoft SQL Server некоторых версий, но для них ряд предоставляемых ORM-функций может достаточно сильно отличаться от тех, что предоставляются официально поддерживаемым СУБД.

<u>База данных</u>

Модель в Django представляет собой источник информации о данных. Она содержит основные поля и поведение хранимых данных. Как правило, каждая модель отображается в одну таблицу базы

данных. В Django модели представлены классами Python, т.е. конкретная сущность описывается отдельным классом. При этом каждая модель представляет собой подклассы Python django.db.models.Model, соответственно, могут быть использованы методы Model. Атрибут модели — поле таблицы в базе данных. Для работы с моделями Django предоставляет автоматически генерируемый API доступа к базе данных (Model API reference).

```
class Note(models.Model):
    note_text = models.CharField(max_length=200)
```

Рис. 41. — Фрагмент кода модели Django

Поля модели представлены экземплярами класса Field, он определяет тип поля, так например для символьных полей используют CharField, для десятичных чисел DecimalField и т.д.

После того как модель описана, её необходимо активировать, для это необходимо выполнить команду `python manage.py makemigrations` ***`appname`***. Т.е. выполнить миграцию, в Django миграция описывает, каким образом Django сохраняет изменения в моделях и, соответственно, в схеме базы данных.

При успешном выполнении команды в консоль будет выведено:

```
Migrations for 'stdapp':
  stdapp/migrations/0001_initial.py
    - Create model Note
```

Рис. 42. — Результат в консоли

stdapp — название приложения, а Note — имя модели. Все описанные модели в файле приложения models.py будут активированы и также перечислены в выводе выполнения команды `python manage.py makemigrations`.

Выполненную миграцию можно посмотреть в сгенерированном файле *appname*/migrations/0001_initial.py, где 0001 — номер выполненной миграции:

```python
from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Note',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True,
                ('note_text', models.CharField(max_length=200)),
            ],
        ),
    ]
```

Рис. 43. — Миграции Django

Данный файл при необходимости может быть отредактирован вручную.

Помимо описанных в модели полей дополнительно будет создано поле id. При добавлении новых описаний моделей или изменений в имеющихся, миграции будут выполняться только для внесённых изменений.

Для автоматического запуска миграций используется команда `migrate`, команда имеет два параметра [app_label] и [migration_name], по умолчанию будут выполнены все миграции для всех приложений, при указании параметра [app_label] для конкретного приложения (также может запустить связанные миграции в других приложениях), при указании [migration_name], будет выполнена конкретная миграция, при этом более поздние миграции выполнены не будут (# см. Migrations).

Для просмотра SQL запроса миграции можно воспользоваться командой `python manage.py sqlmigrate` ***`appname`*** `0001`, где 0001 — номер миграции (см. рис. 44).

```sql
BEGIN;
--
-- Create model Note
--
CREATE TABLE "stdapp_note" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "note_text" varchar(200) NOT NULL);
COMMIT;
```

Рис. 44. — SQL запрос миграции Django

Пользователь-администратор

Для создания пользователя администратора используется команда `python manage.py createsuperuser`. После будет предложено ввести желаемое имя, по умолчанию это будет имя пользователя в системе (если оставить строку ввода пустой), указать эл. почту и задать пароль. После того как пользователь-администратор создан можно запустить сервер (`python manage.py runserver`) и в браузере перейти http://127.0.0.1:8000/admin/.

В браузере будет загружен соответствующий ресурс:

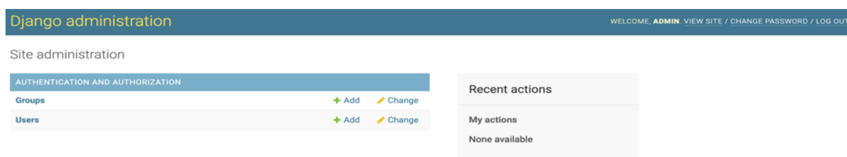

Рис. 45. — Панель администратора Django

Для добавления интерфейса администратора для приложения необходимо в файле приложения admin.py указать модели приложения, которые должны быть доступны администратору — admin.site.register (*appname*).

```
from django.contrib import admin

from .models import Note

admin.site.register(Note)
```

Рис. 46. — Определение моделей доступных через интерфейс администратора

Через интерфейс администратора можно добавлять новые объекты, редактировать и удалять уже имеющиеся. А также просматривать историю изменений («History»).

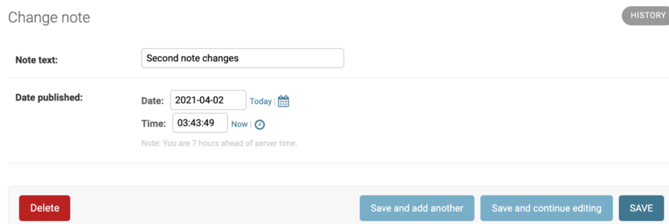

Рис. 46. — Интерфейс администратора

Использование Template для представления данных модели

Django использует MVT подход и для представления данных моделей используется шаблоны, их основное предназначение — динамическое представление данных. Шаблоны представляют собой статический HTML (либо в другом поддерживаемом формате XML, CSV и т.д.) и динамические данные, рендеринг которых описывается специальным синтаксисом. Для размещения шаблонов в директории приложения необходимо создать поддиректорию template, где должны быть размещены файлы, содержащие шаблоны.

Произвольную директорию можно указать в соответствующей переменной TEMPLATE — DIR[ ]. При указании APP_DIRS': True, загрузчик шаблонов будет искать шаблоны в каталоге шаблонов приложения.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
```

Рис. 47. — Template (фрагмент)

Основными конструкциями шаблонов являются теги и переменные.

Переменная позволяет выводить некое значение контекста, которое, по сути, является dict-подобным объектом (изменяемый объект-отображение): {{ *variable name* }}. Контекст в данном случае — словарь содержащий пары ключ-значение, где имена переменных и их значения выступают, соответственно, в качестве ключа и его значения. Для обращения и поиска используется точечная нотация: *some_object.attribute*.

Теги обеспечивают произвольную логику в процессе рендеринг. Тег задаётся следующей конструкцией: {% *name* %}. Теги могут использоваться для вывода контекста, представлять структуру управления, например для организации цикла. Часть тегов требует открывающей и закрывающей конструкции, так парный тег используется для организации таких циклов как if и for:

{% if … %} {% endif %} {% else %}

{% for … %} {% endfor %}

К шаблону могут применяться фильтры: {{ some_date|date:"Y-m-d" }}, где date — фильтр, а “Y-m-d" — аргумент фильтра (но, большая часть фильтров Django не принимает аргументы).

Для комментариев имеется соотвествующий тег:

{# comment #} — для однострочных комментариев;

{% comment %} — для многострочных комментариев.

Django содержит достаточно большое число различных тегов и фильтров , и также позволяет создавать пользовательские теги и пользовательские фильтры. Стоит обратить внимание, что в языке шаблонов Django нет обработки исключений, при получении исключения выдаётся ошибка сервера.

Для представления данных пользователю посредством шаблонов, необходимо обратиться к шаблону во View (в представленном примере шаблон «Notes.html»):

```
from django.shortcuts import render
from django.http import HttpResponse
from django.template import loader

from .models import Note

def index(request):

    notes_list = Note.objects.all()
    template = loader.get_template('Notes.html')
    context = {
        'notes_list': notes_list,
    }
    return HttpResponse(template.render(context, request))
```

Рис. 48. — View (фрагмент)

Данный View будет представлять пользователю список сохранённых заметок из Note. Здесь запрашиваются все имеющиеся объекты. Для вывода текста заметок используется шаблон:

```
{% if notes_list %}
    {% for note in notes_list %}
        <ul class="list-group">
            <li class="list-group-item">
                {{ note.note_text }}
            </li>
    {% endfor %}
        </ul>
{% else %}
    <p>No one note are available.</p>
{% endif %}
```

Рис. 49. — Запрос данных с использованием шаблона

{{ note.note_text }} — переменная, чьё значение будет выведено пользователю, в данном случае это будет значение атрибута note_text объекта note.

Обновление данных модели

Для отображения данных пользователю приложения используются теплейты (templates), посредством которых представление отображает данные модели. Сама модель представляет собой соответствующую таблицу в БД (базе данных), а конкретная запись в таблице является экземпляром модели. Данные модели могут обновляться различными способами, в частности через интерфейс администратора или непосредственно пользователем. В Django для получения данных для изменений в БД, в общем, случае используются формы. Они позволяют создавать, обновлять и удалять экземпляры модели.

HTML-формы

HTML-формы представляют собой набор элементов помещённых в тег <form>…</form>, в частности включают в себя виджеты для ввода различных типов данных (текст, дата, ссылка и т.д.). Для работы с элементами HTML-формы как и для HTML-документа в целом применяются CSS и JavaScript. Как правило, форма содержит <input> элементы, в том числе <input type="submit">, элемент типа «submit» используется для отправки данных формы в соответствующий обработчик формы.

```
<form method="POST">
    <label for="team_name">Note: </label>
    <input id="notename" type="text" name="name_field" value="Note header">
    <label for="team_name">Note text: </label>
    <input id="notetext" type="text" name="name_field" value="Text">
    <input type="submit" value="OK">
</form>
```

Рис. 50. — Запрос данных с использованием шаблона

Тег <form> имеет два важных атрибута method и action. Атрибут метод определяет какой HTTP-метод требуется использовать, для форм используются методы POST или GET:

- метод POST используется в случае, если требуется отправить данные для внесения изменений в БД;
- метод GET используется в случае, если требуется получить данные, в том числе для выполнения запросов к БД.

Атрибут action определяет ресурс/URL-адрес куда требуется отправить данные для обработки. В случае если атрибута не задано, введённые данные будут отправлены в код представления (функцию, или класс), сформировавший текущую страницу.

В общем случае, на сервер возлагается отправка начального состояния формы, обработка полученных от клиента данных, в частности их валидация. В случае, если данные не корректны сообщить об этом пользователь, например, отправив вновь начальное состояние формы и соответствующее сообщение с описанием проблемы. В случае, если данные не корректны сервер должен выполнить предусмотренные действия, например, сохранить данные, возвратить результата поиска, загрузить файл и т.д. И при необходимости проинформировать пользователя о совершённых действиях.

Формы в Django

Использование HTML-форм относится в большей степени к темплейтам (templates), в Представлениях (views) в Django используются собственные формы, определяемые через класс Form, который позволяет создавать HTML-формы: описывает форму и определяет, как форма работает и выглядит.

Как правило, в директории проекта создаётся соответствующей файл forms.py.

```
forms.py
from django import forms

class UserForm(forms.Form):
    username = forms.CharField(max_length=50)
    password = forms.CharField(max_length=20)
```

Рис. 51. — Форма Django *(фрагмент)*

Каждое поле формы определенно своим классом, например, FileField, DateField, TextField и т.д. (см. Field types) и представлено соответствующим виджетом. Так, например, поле обозначенное как CharField по умолчанию представлено виджетом TextInput, который создает тег <input type="text"> в HTML. Если необходимо использовать другой виджет, он может быть предопределён при определении поля формы: вместо forms.CharField() можно указать forms.CharField(widget=forms.Textarea), при этом будет создан тег <textarea> вместо тега <input type="text">.

```
forms.py
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    comment = forms.CharField(widget=forms.Textarea)
```

Рис. 52. — Назначение требуемого виджета в форме Django

Форма рендерится для клиента сходным образом с рендерингом данных модели образом.

```
userform = UserForm()
return render(request, "index2.html", {"form": userform})
```

Рис. 53. — Рендеринг формы Django

- в Представлении (View) определён соответствующий код для формы;

- в используемом темплейте указана соответствующая переменная контекста ({{ form }}, указанный в данном примере {% csrf_token %} используется для защиты от межсайтовой подделки запроса, не рекомендован для использования с ссылающимися на внешние URL формами); при этом все поля формы будут добавлены в HTML из переменной контекста {{ form }} при рендеринге шаблона.

Если для <form> не указан атрибут action, то данные формы будут возвращены в тот же фрагмент кода, из которого была вызвана форма, если атрибут определён, то по указанному в атрибуте URL.

Для рендеринга полей формы в тегах <tr>, <p>, <li> могут быть заданы соответствующие значения переменной контекста:

- {{ form.as_table }} — поля формы будут выведены в таблице, в ячейках тега <tr>, при этом тег <table> в HTML должен быть создан заранее;
- {{ form.as_p }} — поля формы будут выведены в теге <p>
- {{ form.as_ul }} — поля формы будут выведены в теге <li>, при этом тег <ul> в HTML должен быть создан заранее.

При необходимости конкретные поле можно рендерить вручную, доступ к полю можно получить через атрибут формы name_of_field: {{ form.name_of_field}}.

При использовании метода GET Представление должно добавлять в контекст шаблона пустую форму для рендеринга и отправки. При запросе POST представление можно отправлять форму с соответствующими данными из запроса, при этом форма будет связана с этими данными. Для уточнения связана ли форма с данными можно проверить значение атрибута is_bound.

Класс Form имеет метод для проверки данных формы is_valid() для проверки валидности данных формы. Он возвращает True/False и в зависимости от результата пользователю должна быть возвращена форма с требованием ввести корректные данные (возвращено False) или же при необходимости сообщение об успешном завершении действия (возвращено True). При этом так как данные и формы связаны, то можно вернуть пользователю форму с введёнными ранее данными с указанием какие из них некорректны. Помимо метода is_valid() для проверки валидности данных существует множество различных методов (см. Form and field validation). Для работы с ошибками формы существуют соответствующие инструменты (Form.errors).

```
{% if form.non_field_errors %}
<ul>
  {% for error in form.non_field_errors %}
    <li>{{ error }}</li>
  {% endfor %}
</ul>
{% endif %}
```

Рис. 54. — Обработка ошибок

Ошибки форм также требуют соответствующего рендеринга, например, можно использовать {{ form.non_field_errors }} для работы с ошибками, не относящихся к определённому полю, так же можно получить ошибку по конкретному полю через атрибут errors ({{ form.fieldname.errors }}).

Создание формы на основе модели

Как правило, приложение использует базу данных, и многие формы аналогичны моделям. И, соответственно, создание формы дублирующую модель избыточно, поэтому в Django имеется возможность создавать формы на основе классов модели. Для этого используется класс ModelForm.

```
from django.forms import ModelForm
from myapp.models import Note

class NoteForm(ModelForm):
    class Meta:
        model = Note
        fields = ['note_name','note_text','pub_date']
```

Рис. 56. — Создание формы на основе модели

При создании формы на основании модели необходимо указать модель и перечислить соответствующие поля модели. Сгенерированный класс формы будет содержать соответствующее поле формы для каждого поля модели в том порядке, в котором они указаны в атрибуте fields. При этом каждому полю модели соответствует стандартное поле формы. Так, поле модели CharField будет представлено на форме как CharField. Но не все классы полей совпадают в моделях и формах, так поле модели ManyToManyField будет представлено как поле формы MultipleChoiceField. Соответствие между квасами полей можно посмотреть в Field types.

Для созданной на основе модели формы также проводится валидация. Для этого также может быть использован is_valid() для выполнения проверки всех полей, включенных в форму. Для проверки полей модели используется Model.clean_fields(), для проверки объекта модели целиком — Model.clean(), проверка уникальности полей проводится при помощи Model.validate_unique().

Менеджер модели Django

Для работы с базой и обеспечения операций с данными в базе модель использует соответсвующий интерфейс — менеджер модели. По умолчанию Django для каждого класса модели добавляет Manager с именем objects. Имя менеджера можно изменить, для этого необходимо определить для требуемого класса атрибут класса типа models.Manager() с требуемым именем (model_manager_name = models.Manager()). Используя, менеджер модели, можно обращаться к конкретным полям объектов класса модели.

Операции с объектами и данными модели

Модель неразрывно связана с базой данных и, следовательно, работа с моделью подразумевает выполнение запросов к базе данных. Django предоставляет API-интерфейс для базы данных, который позволяет создавать, извлекать, обновлять и удалять объекты.

Для получения объектов из базы данных необходимо создать QuerySet, используя менеджер соответствующего класса модели. QuerySet представляет коллекцию объектов из базы данных. Например, Note.objects.all() позволяет получить все объекты класса Note, хранящиеся в базе данных. Для работы с QuerySet используется соответствующий API.

Для получения конкретных объектов или конкретных групп объектов используются фильтры по заданным параметрам filter(**kwargs), он возвращает новый QuerySet, содержащий объекты, которые соответствуют заданным параметрам поиска, например, Note.objects.filter(pub_date__year=2020) вернёт все записи за 2020 год. QuerySet можно ограничить при помощи соответствующего синтаксиса нарезки массивов в Python, например, Note.objects.all()[:5], вернёт массив из первых пяти объектов. Или же можно извлечь конкретный объект Note.objects.all()[0], при необходимости объекты можно упорядочить и выбрать конкретный из них, например, Entry.objects.order_by('note_name')[0] и т.д.

Также может быть использован метод exclude(**kwargs), он возвращает новый QuerySet, содержащий объекты, которые не соответствуют указанным параметрам поиска. Данные методы могут использоваться в комбинации при необходимости.

Если соотнести QuerySet с SQL, то его получение приравнивается к работе оператора SELECT, а фильтр при этом

выступает в качестве ограничивающего предложения, как, например, WHERE (# …(pub_date__year=2020)) или LIMIT (# …[:5]).

Для работы с объектами модели в Django используются различные методы, для простых действий таких как создание обновление и удаления объекта применяются:

save() — для создания и сохранения изменений объектов модели;

delete() — соответственно, для удаления объектов.

Для работы с объектами в наборах связанных объектов (объектов, находящихся в отношениях «один ко многим» или «многие ко многим»):

add() — для добавления объектов в набор связанных объектов используется метод например, при сохранении ForeignKey и ManyToManyField;

create() — для создания объекта помещаемого в связанный список объектов;

remove() — соответственно, для его удаления.

Помимо данных методов для работы с моделью имеется множество различных методов, они описаны в соответствующей документации (в частности, см. Models, QuerySet).

Список источников

1. Django documentation 4.0. — URL: https://docs.djangoproject.com/en/4.0/ (дата обращения 08.02.2022).
2. Python. — URL: https://www.python.org/ (дата обращения 08.02.2022).
3. venv — Creation of virtual environments. — URL: https://docs.python.org/3/library/venv.html#module-venv (дата обращения 08.02.2022).
4. Virtualenv. — URL: https://virtualenv.pypa.io/en/stable/ (дата обращения 08.02.2022).

3 Методические указания для организации самостоятельной работы

3.1 Общие положения

Целями самостоятельной работы являются систематизация, расширение и закрепление теоретических знаний, приобретение навыков исследовательской деятельности.

Самостоятельная работа студента по дисциплине «Технологии программирования» включает следующие виды его активности:

1. проработка лекционного материала;

2. подготовка к практическим занятиям;

3. подготовка к зачёту с оценкой.

3.2 Проработка лекционного материала

Данный вид самостоятельной работы направлен на получение навыков работы с конспектом, структурирования материала, а также умения выделить основные пункты и положения, изложенные на лекции. Кроме того, проработка лекционного материала способствует более глубокому пониманию и прочному запоминанию теоретической части дисциплины.

При проработке лекционного материала необходимо:

1. отработать прослушанную лекцию, то есть прочитать конспект, прочитать учебник и сопоставить его материал с конспектом; восполнить пробелы, если они остались после лекции в силу того, что студент что-то не понял или не успел записать;

2. перед каждой последующей лекцией прочитать предыдущую, чтобы не тратилось много времени для восстановления контекста изучения дисциплины при продолжающейся теме, а также чтобы максимально правильно ответить на вопросы теста, который проводится на каждой лекции.

Для наиболее эффективной работы с конспектом рекомендуется сначала просмотреть его целиком, чтобы выделить структуру лекции. Эту структуру полезно выписать в виде плана. Затем по каждому пункту нужно выделить основные положения, определения и формулы, если они есть. Формулы тоже полезно записывать, чтобы кроме зрительной, включалась еще и моторная память.

3.3 Подготовка к лабораторным работам

Для подготовки к практическим занятиям необходимо изучить теоретические вопросы по теме работы, ознакомится с рекомендованными источниками, проработать основные понятия, необходимые для решения практических задач и выполнения задания по лабораторной работе.

Лабораторная работа «Анализ идеи проекта»

В рамках подготовки к лабораторной работе «Анализ идеи проекта» рекомендуется проработать конспект лекций и ознакомиться с источниками по тематике работы.

Лабораторная работа «Карта пользовательских историй»

В рамках подготовки к лабораторной работе «Карта пользовательских историй» рекомендуется проработать конспект лекций и ознакомиться с источниками по тематике работы.

Лабораторная работа «Проектирование»

В рамках подготовки к лабораторной работе «Проектирование» рекомендуется проработать конспект лекций и ознакомиться с источниками по тематике работы.

Лабораторная работа «Внедрение системы контроля версий»

В рамках подготовки к лабораторной работе «Внедрение системы контроля версий» рекомендуется проработать конспект лекций и ознакомиться с источниками по тематике работы.

Лабораторная работа «Реализация front-end приложения»

В рамках подготовки к лабораторной работе «Реализация front-end приложения» рекомендуется проработать конспект лекций и ознакомиться с источниками по тематике работы.

Лабораторная работа «Реализация back-end приложения»

В рамках подготовки к лабораторной работе «Реализация back-end приложения» рекомендуется проработать конспект лекций и ознакомиться с источниками по тематике работы.

4. Рекомендуемая литература

1. Галиаскаров, Э.Г. Анализ и проектирование систем с использованием UML: учебное пособие для вузов / Э. Г. Галиаскаров, А.С. Воробьев. — Москва: Издательство Юрайт, 2021. — 125 с. — (Высшее образование). — ISBN 978-5-534-14903-6. — Текст: электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/485415 (дата обращения 08.02.2022).

2. Лаврищева, Е. М. Программная инженерия и технологии программирования сложных систем: учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. и доп. — М.: Юрайт, 2021. — 432 с. — (Высшее образование). — ISBN 978-5-534-07604-2. — Текст: электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/470923 (дата обращения 08.02.2022).

3. Проектирование информационных систем: учебник и практикум для вузов / Д.В. Чистов, П.П. Мельников, А.В. Золотарюк, Н.Б. Ничепорук; под общей редакцией Д.В. Чистова. — 2-е изд., перераб. и доп. — Москва: Издательство Юрайт, 2021.— 258 с. — URL: https://urait.ru/bcode/469199.

4. Сысолетин, Е. Г. Разработка интернет-приложений: учебное пособие для вузов / Е. Г. Сысолетин, С. Д. Ростунцев; под научной редакцией Л. Г. Доросинского. — Москва: Издательство Юрайт, 2022. — 90 с. — URL: https://urait.ru/bcode/492224 (дата обращения 08.02.2022).