Билет №1

1. Теоретический вопрос: Модели жизненного цикла ПО

Модель жизненного цикла (ЖЦПО) — это структура, описывающая этапы разработки: от анализа требований до вывода программы из эксплуатации.

1. Каскадная (водопадная) модель

Это последовательный переход между этапами. Следующий этап начинается только после полного завершения предыдущего.

- **Достоинства:** Прозрачность управления, фиксированная стоимость и сроки, строго регламентированная документация.
- **Недостатки:** Сложность возврата на шаг назад, высокая стоимость ошибки (ошибка в требованиях обнаружится только в конце), отсутствие гибкости.
- **Применение:** Проекты с жесткими требованиями (госзаказы, системы управления в авиации или медицине).

2. Итерационная модель

Разработка ведется повторяющимися циклами (итерациями). С каждой новой итерацией создается более полная версия продукта.

- **Достоинства:** Возможность получать обратную связь от заказчика на ранних стадиях, гибкость к изменениям требований.
- **Недостатки:** Риск затягивания сроков («бесконечные правки»), сложность проектирования архитектуры сразу на весь проект.
- **Применение:** Большинство современных коммерческих проектов, веб-сервисы, мобильные приложения.

3. Спиральная модель

Основана на итерационном подходе с уникальным фокусом на **анализ рисков**. Каждый виток спирали включает: планирование, оценку рисков, конструирование и оценку результата.

- **Достоинства:** Самая высокая надежность, раннее выявление критических проблем.
- **Недостатки:** Высокая стоимость, необходимость в экспертах по оценке рисков.

- **Применение:** Уникальные, дорогостоящие и инновационные системы (космическая отрасль, банковское ядро).

2. Практическое задание: План перехода на Windows 10/11

Для компании из 30 сотрудников предлагается следующий 5-этапный план:

Этап 1: Аудит и совместимость

- **Инвентаризация железа:** Проверка ПК на соответствие системным требованиям (процессор, ОЗУ от 8 ГБ, наличие SSD).
- **Анализ ПО:** Составление списка критически важных программ (1С, CRM, ключи ЭЦП). Проверка их совместимости с новой ОС на сайте вендоров.

Этап 2: Резервное копирование (Backup)

- **Пользовательские данные:** Копирование рабочих файлов (Документы, Рабочий стол, почта Outlook) на сетевой диск или сервер.
- **Образ системы:** Создание образа текущего состояния диска (System Image) для быстрого отката в случае сбоя.

Этап 3: Управление правами и безопасностью

- **Групповые политики:** Настройка прав через Active Directory. Пользователи **не должны** иметь прав локального администратора (для безопасности).
- **Антивирус:** Убедиться, что корпоративный антивирус поддерживает новую версию ОС.

Этап 4: Поэтапное развертывание

- **Пилотная группа:** Обновить ОС сначала у 3-5 сотрудников из разных отделов.
- **Масштабирование:** При отсутствии жалоб в течение 2-3 дней — обновление остальных рабочих мест (в нерабочее время).

Этап 5: Обучение

- Подготовка краткой памятки (PDF) с описанием нового интерфейса.
- Проведение 15-минутного инструктажа по основным изменениям.

3. Ситуационная задача: Диагностика медленной работы ПК

Действия сисадмина (технический план):

1. **Диспетчер задач:** Проверка вкладок «Процессы» (нагрузка на ЦП и диск) и «Автозагрузка».
2. **Монитор ресурсов:** Анализ того, не перегружает ли новое приложение оперативную память или запись на диск.
3. **Логи системы:** Проверка «Просмотра событий» Windows на наличие конфликтов драйверов или ошибок после установки.
4. **Антивирус:** Проверка, не конфликтует ли новое ПО с защитными системами в режиме реального времени.

Билет №2

1. Теоретический вопрос: Системы управления версиями (VCS)

Система управления версиями представляет собой специальное программное обеспечение, предназначенное для облегчения работы с изменяющейся информацией. В контексте разработки ПО она фиксирует каждое изменение в файлах проекта, позволяя в любой момент вернуться к предыдущему состоянию, сравнить разные версии кода или узнать, кто именно внес те или иные правки. Основное назначение таких систем заключается в обеспечении надежного хранения истории проекта, возможности безопасного проведения экспериментов через создание изолированных копий и организации эффективной совместной работы многих программистов над общим продуктом без риска потери данных.

В основе работы большинства VCS лежат три ключевых понятия. **Репозиторий** — это своего рода хранилище или база данных, в которой содержатся не только актуальные файлы проекта, но и вся история их изменений.

Коммит является базовой единицей этой истории; это зафиксированный «снимок» состояния проекта в определенный момент времени, снабженный автором, датой и описанием.

Ветка представляет собой параллельную линию разработки, которая позволяет отклониться от основного кода для работы над новой функцией или исправления ошибки, не мешая при этом коллегам.

Среди популярных систем выделяются Git и SVN, которые используют разные подходы.

Git — это распределенная система, где каждый участник имеет на своем компьютере полную копию всей истории репозитория. Это делает работу быстрой, независимой от интернета и предоставляет мощные инструменты для слияния веток.

SVN (Subversion) — это централизованная система, где вся история хранится на одном сервере, а пользователи скачивают только рабочую копию текущих файлов. SVN проще в освоении и удобнее для контроля доступа к отдельным частям проекта, но требует постоянного подключения к сети для совершения большинства операций и медленнее работает с ветками.

3. Ситуационная задача: Разрешение конфликта слияния в Git

Конфликт слияния в Git возникает в ситуации, когда система не может автоматически объединить изменения, например, если два разработчика одновременно изменили одну и ту же строку в одном и том же файле. Для разрешения этой проблемы необходимо выполнить последовательность четких действий.

Сначала следует обнаружить конфликт: Git сообщит о нем при попытке слияния веток или подтягивания обновлений, остановив процесс. С помощью команды git status можно точно определить, в каких файлах возникли противоречия. Далее разработчик должен открыть конфликтный файл в текстовом редакторе. Внутри файла появятся специальные маркеры: блок между <<<<<<< HEAD и разделителем ======= покажет локальные изменения, а блок после разделителя до >>>>>>> — изменения из вливаемой ветки.

Процесс непосредственного разрешения заключается в том, чтобы вручную отредактировать файл, удалив все служебные маркеры и оставив финальный, корректно работающий вариант кода. Это может быть либо одна из версий, либо их комбинация. После сохранения исправленного файла его необходимо пометить как разрешенный, добавив в индекс с помощью команды git add. Завершающим этапом является создание коммита слияния через команду git commit, что официально фиксирует разрешение конфликта в истории проекта.

Билет №3

1. Теоретический вопрос: Тестирование ПО

Тестирование программного обеспечения представляет собой комплексный процесс проверки того, насколько поведение готовой программы соответствует ожиданиям разработчика и требованиям заказчика. Главная задача этого процесса заключается в раннем выявлении дефектов, что позволяет значительно снизить стоимость исправления ошибок и повысить общую надежность продукта. В современной разработке тестирование принято разделять на несколько уровней и видов в зависимости от преследуемых целей.

Уровни тестирования определяют масштаб и глубину проверки кода. Первый и самый детальный уровень — **модульное (unit) тестирование**.

На этом этапе проверяются минимальные части программы, такие как отдельные функции или методы, в полной изоляции от остального кода. Это позволяет быстро найти логические ошибки в конкретном блоке. Далее следует **интеграционное тестирование**, задача которого — проверить, как ранее протестированные модули взаимодействуют друг с другом.

Здесь выявляются ошибки в интерфейсах передачи данных и несовместимость компонентов. Высшим уровнем является **системное тестирование**, в рамках которого проверяется всё приложение целиком.

Оно проводится в среде, максимально близкой к реальной, чтобы убедиться в готовности продукта к эксплуатации.

Виды тестирования классифицируются по целям и характеристикам, которые нужно проверить.

Функциональное тестирование фокусируется на том, *что* делает программа: выполняет ли она все заложенные функции, правильно ли реагируют кнопки и корректно ли производятся расчеты.

Нагрузочное тестирование относится к нефункциональным проверкам и исследует поведение системы под большим давлением, например, при одновременной работе тысяч пользователей. Это критически важно для определения стабильности системы.

Наконец, **регрессионное тестирование** — это повторная проверка системы после внесения любых изменений (исправления багов или добавления функций). Его цель — гарантировать, что новые правки не «сломали» тот функционал, который ранее работал исправно.

3. Ситуационная задача: Запрос на изменение в конце проекта

Ситуация, когда заказчик просит внедрить новую функцию в практически готовый проект, является классическим примером «раздувания рамок» (scope creep). В такой момент важно действовать профессионально, чтобы сохранить хорошие отношения с клиентом, но при этом не поставить под удар успех всего релиза.

Последовательность действий: Прежде всего, необходимо официально зафиксировать запрос и провести оперативный анализ влияния (Impact Analysis). Нужно оценить, какие части системы затронет новая функция, сколько времени потребуется на разработку и последующее тестирование, и как это сдвинет финальные сроки. С этой оценкой следует выйти на переговоры. Наиболее конструктивным решением будет предложение выпустить текущую версию проекта согласно исходному ТЗ, а новую функцию включить в план следующего обновления (версия 1.1). Если же внедрение требуется немедленно, необходимо оформить это как официальное дополнительное соглашение с пересмотром сроков и бюджета.

Аргументы для обсуждения: Главный аргумент — **соблюдение сроков**. Внедрение незапланированного функционала на финальной стадии требует проведения полного цикла регрессионного тестирования, что неизбежно отодвинет запуск проекта.

Вторым важным фактором является **стабильность системы**: внесение правок в спешке в отлаженный код резко повышает риск возникновения новых критических багов. Также стоит упомянуть **бюджетные риски**: использование ресурсов команды сверх плана требует дополнительного финансирования.

Наконец, необходимо подчеркнуть важность **качества**: приоритетом на текущем этапе является доведение до совершенства уже реализованных функций, чтобы пользователь получил качественный продукт, а не набор недоработанных фич.

Билет №4

1. Теоретический вопрос: Документирование ПО

Документирование программного обеспечения — это процесс создания текстовых и графических материалов, которые описывают устройство, функции и правила использования продукта. Качественная документация критически важна для поддержки проекта, передачи знаний между разработчиками и успешного освоения программы пользователями. Всю документацию принято делить на три основные категории в зависимости от целевой аудитории и назначения.

Проектная документация создается на начальных этапах жизненного цикла ПО. К ней относятся план проекта, описание архитектуры, концепция продукта (Vision and Scope) и графики работ. Эти документы нужны менеджерам, заказчикам и ведущим разработчикам для контроля сроков, ресурсов и общего вектора развития. Она отвечает на вопрос «Зачем и в какие сроки мы это делаем?».

Техническая документация предназначена для специалистов (разработчиков, системных администраторов, тестировщиков). Она включает в себя описание API, схемы баз данных, руководства по развертыванию (Deployment Guide) и комментарии внутри самого кода. Особое место здесь занимает

Техническое задание (ТЗ) — фундаментальный документ, описывающий требования к системе. Техническая документация отвечает на вопрос «Как это устроено и как это поддерживать?».

Пользовательская документация ориентирована на конечного потребителя. К ней относятся руководства пользователя (Manuals), краткие инструкции по установке, справки и FAQ. Её цель — максимально простым языком объяснить человеку, не обладающему глубокими техническими знаниями, как решить свои задачи с помощью программы. Она отвечает на вопрос «Как этим пользоваться?».

Стандарты оформления помогают сделать документы понятными и унифицированными. В России для ТЗ часто применяются государственные стандарты, такие как **ГОСТ 34** (автоматизированные системы) или **ГОСТ 19** (единая система программной документации). На международном уровне ориентируются на стандарты **ISO/IEC**. В современной гибкой разработке (Agile) чаще используют облегченные форматы документирования в Wiki-системах (например, Confluence), где ТЗ может состоять из набора User Stories и критериев приемки (Acceptance Criteria), оформленных по правилам Markdown.

3. Ситуационная задача: Работа с «плохим» кодом коллеги

Ситуация, когда код работает, но написан запутанно и без пояснений (так называемый «спагетти-код»), часто встречается в реальной практике. Работа с таким модулем требует осторожности, чтобы не сломать существующую логику при добавлении новых функций.

Последовательность действий:

Первым делом необходимо провести **анализ и исследование**. Не стоит сразу переписывать код. Нужно запустить его и проследить путь данных, чтобы точно понять, какие входные параметры он принимает и какой результат выдает. Если коллега доступен, стоит организовать короткую встречу, чтобы он объяснил неочевидные моменты.

Вторым важным шагом является **покрытие тестами**. Прежде чем вносить изменения, нужно написать модульные тесты (Unit-тесты) для текущего состояния кода. Это создаст «сетку безопасности»: если в процессе доработки вы что-то сломаете, тесты сразу об этом сообщат.

Третий этап — **поэтапный рефакторинг**. В процессе добавления новой функции следует придерживаться «правила бойскаута»: оставлять код чуть чище, чем он был до вашего прихода. Это включает в себя переименование переменных на более понятные, разбиение длинных функций на мелкие и логически завершенные блоки.

Завершающим действием станет **документирование**. После того как логика станет понятна, необходимо добавить комментарии к сложным участкам и описать работу модуля в технической документации проекта. Это избавит вас и ваших коллег от повторения этой трудоемкой работы в будущем. Таким образом, вместо простого добавления функции вы превратите «проблемный» модуль в поддерживаемый и качественный компонент системы.

Билет №5

1. Теоретический вопрос: Рефакторинг кода

Рефакторинг кода — это процесс контролируемого изменения внутренней структуры программного обеспечения, который не затрагивает его внешнее поведение. Основная цель рефакторинга заключается в том, чтобы сделать код более чистым, понятным и легким для последующей поддержки и расширения. В отличие от оптимизации, которая направлена на ускорение работы программы, рефакторинг фокусируется на улучшении «читаемости» кода для человека. Это критически важный процесс, позволяющий предотвратить превращение проекта в запутанную и трудноуправляемую систему.

Критерием необходимости рефакторинга обычно выступают так называемые «запахи кода» (code smells). К ним относятся избыточное дублирование логики, слишком длинные методы, огромные классы с множеством зон ответственности или непонятные названия переменных. Основными приемами рефакторинга являются выделение части кода в отдельный метод (Extract Method), переименование переменных для более точного отражения их сути, замена сложных условий полиморфизмом или упрощение вложенных конструкций. Рефакторинг должен проводиться небольшими шагами при обязательном наличии модульных тестов, чтобы гарантировать, что функциональность системы осталась неизменной.

Понятие рефакторинга неразрывно связано с **техническим долгом**. Технический долг возникает, когда ради скорости выпуска продукта разработчики принимают быстрые и не самые качественные решения. Со временем этот «долг» накапливает «проценты» в виде сложности добавления новых функций и роста количества ошибок. Рефакторинг — это основной инструмент «выплаты» этого долга. Если регулярно не проводить рефакторинг, стоимость поддержки системы может стать настолько высокой, что разработка новых функций практически остановится.

3. Ситуационная задача: Баг или «фича»?

Спор между тестировщиком и разработчиком о природе найденной проблемы — классическая ситуация в процессе разработки. Для её конструктивного разрешения необходимо отойти от личных мнений и обратиться к объективным источникам информации.

Последовательность действий для разрешения ситуации:

Первым и главным шагом является обращение к **проектной документации**. Необходимо проверить техническое задание (ТЗ), пользовательские истории (User Stories) или макеты. Если в документации четко описано ожидаемое поведение и программа ему не соответствует, то это однозначно баг. Если же поведение системы совпадает с описанным, но тестировщик считает его неудобным или нелогичным, это может рассматриваться как предложение по улучшению.

Если документация не дает однозначного ответа или требования допускают двоякую трактовку, следующим шагом должно стать привлечение **аналитика или владельца продукта (Product Owner)**. Именно эти специалисты обладают правом решающего голоса, так как они отвечают за бизнес-ценность продукта и понимают, как именно пользователь должен взаимодействовать с системой.

Также важно провести **анализ влияния на пользователя**. Если найденная проблема мешает пользователю выполнить целевое действие или приводит к потере данных, её стоит классифицировать как баг, даже если она не была явно описана в требованиях. В случае, если стороны все равно не приходят к согласию, обсуждение следует перевести в плоскость приоритетов: насколько критична данная проблема для текущего релиза. Итоговое решение должно быть зафиксировано в системе трекинга задач (например, Jira), чтобы избежать повторных споров в будущем и сохранить прозрачность процесса разработки.

Билет №6

1. Теоретический вопрос: Системы контроля версий Git и GitHub

Система контроля версий (VCS) — это фундаментальный инструмент в современной разработке, который позволяет отслеживать изменения в исходном коде и управлять ими. Важно различать **Git** и **GitHub**: Git — это сама консольная программа, работающая локально на компьютере разработчика, а GitHub — это облачная платформа для хранения репозиториев, которая позволяет командам взаимодействовать удаленно.

Основные понятия:

- **Репозиторий** — это своего рода база данных проекта, которая содержит не только текущие файлы, но и всю историю их изменений с момента создания.
- **Коммит** — это зафиксированное состояние проекта. Каждый коммит можно представить как «снимок» (snapshot) всех файлов, имеющий уникальный идентификатор, автора и описание того, что было изменено.
- **Ветка** — это изолированная линия разработки. Она позволяет работать над новой функцией или исправлением ошибки отдельно от основного стабильного кода.
- **Слияние (Merge)** — процесс объединения изменений из одной ветки в другую, например, когда работа над новой функцией завершена и её нужно добавить в основной проект.

Зачем нужен контроль версий? Без VCS совместная работа была бы практически невозможна: разработчики постоянно затирали бы код друг друга. Контроль версий обеспечивает «машину времени», позволяя вернуться к любой точке в прошлом, если в коде появилась критическая ошибка. Кроме того, он служит надежным бэкапом и позволяет документировать процесс разработки через описания к коммитам.

2. Практическое задание: Организация учебного проекта на GitHub

Для правильной организации учебного проекта необходимо выполнить ряд последовательных шагов через веб-интерфейс GitHub.

Создание и описание: Сначала на главной странице GitHub нажимается кнопка «New repository», указывается понятное имя и описание. При создании важно сразу отметить галочку «Add a README file». Файл

README.md — это «лицо» проекта; в нем в формате Markdown описывается предназначение программы, инструкции по установке и используемые технологии.

Структура и именование: Хорошим тоном считается разделение файлов по назначению. В папке **src** (source) размещается исходный код, в **docs** — расширенная документация, в **tests** — проверочные скрипты. Для именования файлов рекомендуется использовать строчные буквы и дефисы (kebab-case), например: user-profile.js. Для коммитов стоит придерживаться стандарта **Conventional Commits**:

- feat: для новых функций;
- fix: для исправления багов;
- docs: для изменений в документации.

Первый коммит: Чтобы создать структуру папок прямо в браузере, можно нажать «Add file» -> «Create new file». Чтобы создать папку, нужно написать её имя и поставить слэш (например, src/index.js). В нижней части страницы заполняется заголовок коммита (например, feat: create initial project structure) и нажимается кнопка «Commit changes».

3. Ситуационная задача: Восстановление удаленного файла

Удаление файла в проекте под управлением Git не является фатальным, так как вся история хранится в репозитории.

Последовательность действий:

1. **Если файл удален в рабочей директории, но изменения еще не зафиксированы (не сделан коммит):** Самый простой способ — использовать команду git restore <путь_к_файлу>. Это мгновенно вернет файл в то состояние, в котором он находился в последнем коммите.
2. **Если удаление уже было зафиксировано коммитом:** Сначала нужно найти идентификатор (хеш) коммита, в котором файл еще существовал. Это делается командой git log -- [путь_к_файлу]. Найдя нужный коммит, можно восстановить файл командой git checkout [хеш_коммита] -- [путь_к_файлу].
3. **Завершение восстановления:** После того как файл снова появился в папке проекта, его необходимо заново добавить в индекс (git add) и зафиксировать новым коммитом с описанием вроде fix: restore accidentally deleted file. Таким образом, файл вернется в актуальную версию проекта.

Билет №7

1. Теоретический вопрос: Совместная разработка на GitHub

Совместная работа над кодом требует инструментов, которые позволяют объединять усилия многих людей, минимизируя риск поломки проекта. GitHub предоставляет для этого три ключевых механизма.

Fork (Форк) — это создание полной копии чужого репозитория в своем аккаунте. Это позволяет свободно экспериментировать с кодом, не имея прав на запись в оригинальный проект. Обычно форки используются в Open Source проектах: разработчик делает форк, вносит изменения и затем предлагает их автору оригинала.

Pull Request (PR) — это предложение внести изменения из вашей ветки (или вашего форка) в основной репозиторий. PR — это не просто технический запрос, а место для обсуждения. Здесь коллеги видят, какие файлы изменились, могут оставлять комментарии к конкретным строкам кода и просить внести правки перед тем, как код попадет в общую базу.

Code Review (Ревью кода) — этап проверки кода другими участниками команды. Цель ревью не в том, чтобы «найти виноватого», а в том, чтобы повысить качество кода, найти потенциальные ошибки, убедиться в соблюдении стандартов оформления и поделиться знаниями внутри команды.

Процесс работы в команде обычно выглядит так: разработчик создает отдельную ветку под задачу, пишет код, отправляет его на GitHub и открывает Pull Request. Коллеги проводят Code Review, после исправления всех замечаний изменения одобряются и вливаются в основную ветку.

2. Практическое задание: Организация workflow командной разработки

Для эффективной работы команды необходимо настроить четкий процесс (workflow), который будет понятен каждому участнику.

Настройка репозитория: Создается центральный репозиторий организации. В настройках (Settings -> Collaborators) добавляются участники команды с правами доступа. Важным шагом является настройка **Branch Protection Rules**: устанавливается запрет на прямые

коммиты в ветку main, чтобы изменения могли попадать туда только через одобренные Pull Request.

Стратегия работы с ветками: Для учебных и небольших проектов чаще всего выбирают **GitHub Flow**. Это простая модель, где есть одна стабильная ветка (main), а любая новая задача или исправление ошибки делается в отдельной ветке с говорящим названием (например, feature/user-login). После завершения задачи ветка удаляется. Более сложный **Git Flow** с отдельными ветками develop и release обычно избыточен для малых команд.

Создание Pull Request: При создании PR важно заполнить описание. Хорошее описание включает:

1. Заголовок с номером задачи.
2. Краткое пояснение: что именно было сделано.
3. Список изменений.
4. Скриншоты (если затронут интерфейс).

Чек-лист для ревьюера:

- **Логика:** решает ли код поставленную задачу? Нет ли лишних изменений?
- **Стиль:** соблюдены ли правила именования и структура проекта?
- **Безопасность:** нет ли в коде паролей или уязвимых мест?
- **Тесты:** покрыта ли новая функция тестами и проходят ли они?

Слияние и конфликты: После одобрения (Approve) выполняется **Merge**. В GitHub лучше использовать **Squash and Merge**, чтобы объединить все мелкие коммиты разработчика в один аккуратный коммит в основной ветке. Если возникают конфликты, они разрешаются локально или через веб-интерфейс перед слиянием.

3. Ситуационная задача: Разрешение конфликта при работе вдвоем

Если два разработчика изменили одну и ту же строку, Git не сможет автоматически решить, чей вариант правильный. Возникает конфликт.

Последовательность действий:

1. **Синхронизация:** Разработчик, который пытается влить свои изменения последним, должен сначала «подтянуть» актуальное состояние

основной ветки в свою рабочую ветку (команды git checkout feature-branch и git merge main).

2. **Поиск конфликта:** Git пометит файл как конфликтный. Разработчик открывает его и видит маркеры: <<<<<<< HEAD (его код) и >>>>>>> main (код коллеги, который уже в репозитории).
3. **Ручное исправление:** Нужно решить, какой код оставить. Возможно, нужно объединить оба решения. Разработчик удаляет маркеры и оставляет только чистый код.
4. **Фиксация:** После правки файл добавляется в индекс (git add [имя_файла]).
5. **Завершение:** Создается коммит слияния (git commit -m "docs: resolve merge conflict with main") и изменения отправляются на сервер (git push).

Билет №8

1. Теоретический вопрос: Понятие «Запах кода» (Code Smell)

Термин «запах кода» был введен Мартином Фаулером для обозначения определенных признаков в исходном коде, которые указывают на наличие глубоких проблем в дизайне или архитектуре системы. Важно понимать, что «запах» — это не техническая ошибка или баг (программа может работать корректно), а симптом, свидетельствующий о том, что код будет трудно поддерживать, тестировать или развивать в будущем.

Примеры запахов кода:

1. **Длинный метод (Long Method).** Это метод, который содержит слишком много строк кода и выполняет сразу несколько действий. **Проблема:** такой код крайне сложно читать, в нем легко запутаться при отладке, и его практически невозможно переиспользовать. **Устранение:** применяется прием «выделение метода» (Extract Method) — логически законченные части кода выносятся в отдельные небольшие методы с понятными названиями.
2. **Большой класс (Large Class).** Класс пытается делать слишком много вещей, нарушая принцип единственной ответственности. **Проблема:** класс становится «божественным объектом», накопление кода в нем ведет к хаосу, а малейшее изменение затрагивает множество зависимостей. **Устранение:** необходимо провести декомпозицию, используя приемы «выделение класса» (Extract Class) или «выделение подкласса», чтобы распределить обязанности между несколькими компактными сущностями.
3. **Дублирование кода (Duplicated Code).** Одинаковые или очень похожие структуры кода встречаются в разных местах проекта. **Проблема:** это самый коварный запах, так как при необходимости внести изменения разработчику приходится искать и править код во всех местах. Если одно место будет пропущено, возникнет баг. **Устранение:** общая логика выносится в отдельный метод, класс или базовый родительский класс.
4. **Одержимость примитивами (Primitive Obsession).** Использование простых типов данных (строк, чисел) там, где должны быть маленькие объекты (например, использование строки для хранения номера телефона или адреса электронной почты без валидации). **Проблема:**

логика проверки данных «размазывается» по всему проекту, а типы данных не защищают от ошибок. **Устранение:** замена значения-данного объектом (Replace Data Value with Object), создание специальных классов для таких сущностей.

5. **Стрельба дробью (Shotgun Surgery).** Ситуация, когда при внесении одного небольшого изменения в логику вам приходится вносить массу мелких правок во множество разных классов. **Проблема:** это признак слишком сильной связанности компонентов. Вы легко можете забыть поправить какой-то файл, что приведет к нестабильности системы. **Устранение:** необходимо переместить методы или поля так, чтобы вся логика, которая меняется одновременно, была сосредоточена в одном месте (Move Method / Move Field).

3. Ситуационная задача: Диагностика и решение системных проблем

Ситуация, когда изменения в одном модуле непредсказуемо ломают другие, называется «хрупкостью системы». Это классический признак высокой связанности (High Coupling) и плохой модульности архитектуры.

Диагностика проблемы: Чтобы найти корень проблемы, я бы провел анализ зависимостей между модулями. Скорее всего, диагностика покажет, что модули знают слишком много о внутреннем устройстве друг друга (нарушение инкапсуляции) или зависят от конкретных реализаций, а не от абстракций. Также отсутствие или недостаточность регрессионного тестирования (автоматических тестов) позволяет этим поломкам доходить до поздних стадий, вместо того чтобы быть обнаруженными сразу.

Предложения для решения на системном уровне:

1. **Внедрение принципов SOLID.** Особое внимание стоит уделить принципу инверсии зависимостей (Dependency Inversion): модули должны зависеть от интерфейсов, а не от конкретных классов. Это позволит менять реализацию одного модуля, не затрагивая другие.
2. **Внедрение автоматизированного тестирования.** Необходимо покрыть критические части системы модульными (Unit) и интеграционными тестами. Это создаст «сеть безопасности», которая будет мгновенно сигнализировать о поломке в несвязанном модуле сразу после внесения правок.

3. **Рефакторинг в сторону уменьшения связанности.** Я предложу команде выделить четкие границы между модулями и использовать паттерны проектирования (например, «Наблюдатель», «Фасад» или «Медиатор»), чтобы минимизировать прямые вызовы между компонентами.
4. **Усиление Code Review.** На проверке кода следует акцентировать внимание на архитектурной чистоте и на том, не вносятся ли новые неявные зависимости между частями системы.
5. **Документирование архитектурных границ.** Четкое описание того, как модули должны взаимодействовать, поможет команде избегать «быстрых», но грязных решений, которые увеличивают технический долг.

Билет №9

1. Теоретический вопрос: Системы контроля версий Git и GitHub

Система контроля версий (VCS) — это программный инструмент, который позволяет разработчикам фиксировать изменения в исходном коде проекта, обеспечивая возможность возврата к любой предыдущей версии. Это своего рода «машина времени» для кода.

Основные понятия:

- **Репозиторий** — это хранилище вашего проекта, где хранится не только текущая рабочая копия файлов, но и вся история их изменений.
- **Коммит** — это зафиксированное состояние файлов в определенный момент. Каждый коммит имеет автора, дату и уникальный идентификатор (хеш). Делая коммит, вы «сохраняете игру» в конкретной точке.
- **Ветка** — это изолированная линия разработки. Она позволяет работать над новым функционалом, не затрагивая основной стабильный код.
- **Слияние (Merge)** — процесс объединения изменений из одной ветки в другую (например, добавление протестированной функции из рабочей ветки в основную).

Зачем нужен контроль версий? Без VCS совместная разработка превратилась бы в хаос: программисты постоянно перезаписывали бы файлы друг друга. Контроль версий позволяет безопасно экспериментировать, легко находить причины багов (проверяя историю изменений) и гарантирует, что ни одна строчка кода не будет потеряна безвозвратно. **GitHub**, в свою очередь, является облачной платформой, которая делает этот процесс социальным: она позволяет хранить репозитории в сети, обсуждать код и управлять доступом команды.

2. Практическое задание: Организация работы с GitHub через командную строку

Для организации учебного проекта по шагам необходимо выполнить следующую последовательность действий в терминале:

Шаг 1. Инициализация и README. Перейдите в папку проекта и создайте локальный репозиторий: git init Создайте файл описания: echo "# Имя проекта" > README.md. В нем стоит кратко описать суть программы и инструкции по запуску.

Шаг 2. Структура папок. Создайте стандартную структуру: mkdir src docs tests Чтобы Git «увидел» пустые папки, в них часто добавляют пустой файл .gitkeep.

Шаг 3. Настройка .gitignore для Python. Это критически важный шаг. Создайте файл .gitignore и добавьте в него названия файлов, которые не должны попасть в репозиторий (временные файлы, библиотеки, конфиденциальные данные): __pycache__/ *.pyc .venv/ (виртуальное окружение) .env (секретные ключи)

Шаг 4. Именование коммитов (Conventional Commits). Придерживайтесь стандарта: тип: описание.

- feat: для нового функционала.
- fix: для исправления ошибок.
- docs: для правок в документации.
- chore: для рутинных задач (например, настройка .gitignore).

Шаг 5. Первый коммит и публикация. Добавьте файлы в индекс: git add . Создайте коммит: git commit -m "feat: initial project structure and docs" Свяжите локальный репозиторий с удаленным на GitHub: git remote add origin https://github.com/vash-login/project-name.git Отправьте код на сервер: git push -u origin main (или master).

3. Ситуационная задача: Работа с «плохим» кодом коллеги

Ситуация, когда вам достается работающий, но нечитаемый код без документации, требует системного подхода, чтобы не внести новые ошибки.

Ваши действия:

1. **Исследование и запуск.** Прежде чем менять хоть одну строчку, убедитесь, что код действительно работает. Проследите движение данных: что подается на вход и что получается на выходе.
2. **Покрытие тестами.** Это самый важный этап. Напишите несколько тестов (Unit-tests), которые фиксируют текущее поведение модуля. Если после ваших правок тесты «упадут», значит, вы нарушили логику.

3. **Постепенный рефакторинг.** Не пытайтесь переписать всё сразу. Применяйте «правило бойскаута»: делайте код чуть лучше в том месте, где вы сейчас работаете. Переименуйте переменные с a и b на осмысленные имена, выделите длинные участки кода в маленькие функции.
4. **Добавление комментариев и документации.** Как только вы разберетесь в логике, задокументируйте её. Напишите docstrings для функций, объясняющие их назначение.
5. **Обратная связь.** Если это возможно, обсудите правки с автором в формате Code Review. Делайте это конструктивно, объясняя, что изменения направлены на упрощение будущей поддержки модуля всей командой.

БИЛЕТ №10

1. Теоретический вопрос: Диаграмма вариантов использования (Use Case Diagram)

Диаграмма вариантов использования (Use Case Diagram) — это графическое представление в языке моделирования UML, которое описывает функциональные требования к системе с точки зрения пользователя. Она позволяет визуализировать, какие функции должна выполнять система и кто будет с ними взаимодействовать, не вдаваясь в детали внутренней реализации. Это «взгляд снаружи», который помогает разработчикам и заказчикам прийти к общему пониманию того, что именно будет делать программный продукт.

Основные элементы диаграммы:

- **Акторы (Actor):** Это внешние сущности, которые взаимодействуют с системой. Акторами могут быть не только люди (роли пользователей, например, «Администратор» или «Клиент»), но и другие программные системы или аппаратные устройства, обменивающиеся данными с нашим приложением. На диаграмме они обычно изображаются в виде стилизованных фигурок людей. Важно помнить, что актор — это именно роль, а не конкретный человек.
- **Варианты использования (Use Case):** Это конкретные цели или задачи, которые актор может решить с помощью системы. Вариант использования описывает последовательность действий, приносящую значимый результат. Например, «Оформить заказ» или «Сменить пароль». Обычно они изображаются в виде овалов.
- **Ассоциации (Association):** Это простые линии, соединяющие актора с вариантом использования. Ассоциация показывает, что данный актор участвует в выполнении конкретной функции.

- **Зависимости (Dependencies):** Отношения между самими вариантами использования. Выделяют два основных типа:
    1. **Включение («include»):** Показывает, что один вариант использования обязательно включает в себя другой. Например, «Оформить заказ» всегда включает в себя «Проверить остаток на складе».
    2. **Расширение («extend»):** Описывает дополнительное или альтернативное поведение, которое происходит только при определенных условиях. Например, функция «Применить промокод» расширяет функцию «Оплатить товар», но выполняется не всегда, а только по желанию пользователя.

3. Ситуационная задача: Анализ пожеланий заказчика через Use Case

Когда заказчик предоставляет список пожеланий в свободной форме, он часто бывает неполным, противоречивым или перегруженным лишними деталями. Процесс построения диаграммы вариантов использования является мощным аналитическим фильтром в этой ситуации.

Как построение диаграммы помогает структурировать работу:

1. **Выявление ролей (Акторов):** Первым делом аналитик выделяет из текста всех, кто будет пользоваться системой. Это помогает понять, для кого мы создаем продукт. Если в пожеланиях заказчика есть функции, для которых не найден «хозяин» (актор), это повод уточнить: действительно ли эта функция нужна?
2. **Определение целей (Use Cases):** Каждое пожелание «хочу, чтобы система могла...» превращается в конкретный вариант использования. Это позволяет отсеять абстрактные фразы и перевести их на язык функциональных требований. Если два пожелания заказчика противоречат друг другу, на диаграмме это станет заметно в виде конфликтующих или дублирующих друг друга овалов.
3. **Уточнение границ проекта (System Boundary):** Это один из важнейших этапов. При рисовании диаграммы мы четко определяем, что находится внутри системы (рамка системы), а что остается снаружи. Если заказчик просит функцию, которая на самом деле должна выполняться сторонним сервисом (например, банком при оплате), диаграмма визуально покажет, что это внешняя зависимость, а не часть нашей разработки. Это помогает избежать «раздувания» бюджета и сроков.
4. **Поиск пробелов:** Когда вы соединяете акторов с функциями, часто выясняется, что в списке пожеланий заказчика не хватает логических шагов. Например, есть функция «Удалить статью», но нет функции «Авторизация». Построение связей include и extend заставляет аналитика задавать правильные вопросы и достраивать недостающие элементы логики еще до начала написания кода.

   Таким образом, Use Case Diagram превращает хаотичный текст в логическую карту проекта, которая служит фундаментом для будущего технического задания.

Билет №11

1. Теоретический вопрос: Диаграмма последовательности (Sequence Diagram)

Диаграмма последовательности — это одна из наиболее важных поведенческих диаграмм в языке UML. Она отображает взаимодействие объектов во времени, фокусируясь на последовательности сообщений, которыми они обмениваются для выполнения определенного сценария. В отличие от других диаграмм, здесь ключевым фактором является время: оно течет сверху вниз. Такая визуализация позволяет наглядно увидеть, как именно данные передаются между компонентами системы для достижения конкретного результата.

Назначение основных элементов диаграммы:

- **Объекты и линии жизни (Lifelines):** Представляют участников взаимодействия (объекты, классы или компоненты системы). Графически это вертикальные пунктирные линии, отходящие вниз от прямоугольника с именем участника. Линия жизни показывает время существования объекта в ходе описываемого процесса.
- **Сообщения (Messages):** Это горизонтальные стрелки, соединяющие линии жизни разных объектов. Они показывают передачу управления или данных. Сплошная стрелка с закрашенным наконечником обычно обозначает синхронный вызов (ожидание ответа), открытая стрелка — асинхронный, а пунктирная линия — возвращаемое значение (ответ).
- **Активационные полосы (Activation Bars):** Это узкие прямоугольники, наложенные на линии жизни. Они показывают период времени, в течение которого объект фактически выполняет какую-то операцию или находится в активном состоянии, обрабатывая полученное сообщение. Если объект вызывает сам себя, на полосе появляется еще одна, более узкая полоса («рекурсия»).
- **Рамки взаимодействия (Interaction Frames):** Это прямоугольные области, которые позволяют описывать логику управления внутри диаграммы. Например, рамка с пометкой **alt** (alternative) показывает условия «если — иначе», рамка **loop** обозначает циклы, а **opt** (optional) — действия, которые могут произойти, а могут и нет. Это делает диаграмму более гибкой и информативной.

3. Ситуационная задача: Локализация ошибки через Sequence Diagram

Когда функция работает нестабильно и ошибка возникает только в определенных условиях, это часто указывает на проблемы в логике взаимодействия компонентов или на некорректную обработку специфических входных данных.

Как диаграмма последовательности помогает в этой ситуации:

1. **Визуализация «проблемного сценария»:** Команде следует построить диаграмму именно для того случая, когда возникает ошибка (edge case). Это заставляет разработчика пошагово восстановить всю цепочку вызовов. Часто уже на этапе рисования становится ясно, что какой-то объект ожидает данные в одном формате, а получает в другом, или что сообщение отправляется не вовремя.
2. **Поиск «тихого отказа»:** Диаграмма позволяет увидеть, на каком именно этапе цепочка сообщений обрывается или возвращает некорректный статус. Например, если объект А отправляет запрос объекту Б, а объект Б вместо данных возвращает ошибку или пустой ответ, активационная полоса объекта Б покажет некорректное завершение операции.
3. **Анализ условий в рамках взаимодействия:** Если ошибка возникает при определенных условиях, использование рамок **alt** на диаграмме поможет команде детально разобрать каждую ветку логики. Можно будет наглядно сравнить, чем «правильный» путь выполнения программы отличается от «ошибочного».
4. **Единое поле для обсуждения:** Диаграмма становится «картой», на которую может смотреть вся команда (разработчики, тестировщики, аналитики). Вместо того чтобы спорить о абстрактном коде, участники могут указать на конкретную стрелку (сообщение) и спросить: «Почему здесь возвращается null?» или «Почему этот запрос уходит до того, как завершилась авторизация?».

Таким образом, диаграмма последовательности превращает скрытые внутренние процессы системы в явную схему, где сбой локализуется в конкретной точке взаимодействия между конкретными объектами.

Билет №12

1. Теоретический вопрос: Диаграмма последовательности (Sequence Diagram)

Диаграмма последовательности — это одна из ключевых диаграмм взаимодействия в языке UML. Ее главная задача — показать, как объекты в системе взаимодействуют друг с другом в динамике, строго соблюдая временную последовательность. В отличие от диаграмм классов, которые описывают структуру (что из чего состоит), диаграмма последовательности описывает процесс (кто, кому и в каком порядке отправляет команды или данные). Время на такой диаграмме всегда течет сверху вниз.

Основные элементы диаграммы:

- **Объекты и линии жизни (Lifelines):** Сверху диаграммы располагаются прямоугольники, представляющие участников взаимодействия (классы, компоненты или пользователи). От каждого из них вниз идет вертикальная пунктирная линия — это «линия жизни». Она обозначает период времени, в течение которого объект существует и может участвовать в процессе.
- **Сообщения (Messages):** Это горизонтальные стрелки между линиями жизни. Они представляют собой вызовы методов, сигналы или передачу данных. Сплошная линия с закрашенным наконечником означает синхронный вызов (отправитель ждет завершения операции), а пунктирная стрелка — это ответ (return message), который возвращает результат выполнения операции отправителю.
- **Активационные полосы (Activation Bars):** На линиях жизни можно увидеть узкие вертикальные прямоугольники. Они показывают моменты, когда объект не просто «существует», а активно выполняет какую-то работу или ждет ответа от другого компонента. Если прямоугольник длинный, значит, операция занимает много времени в общем сценарии.
- **Рамки взаимодействия (Interaction Frames):** Это специальные блоки, которые позволяют добавить в диаграмму логику управления. Самые частые из них: **alt** (выбор между несколькими вариантами, аналог if-else), **loop** (повторение действий в цикле) и **opt** (необязательное действие). Это помогает описывать сложные алгоритмы, не создавая десятки отдельных диаграмм.

3. Ситуационная задача: Локализация ошибки через Sequence Diagram

Когда разработчик говорит, что код работает «в целом правильно», но выдает ошибку при специфических условиях, проблема часто кроется в нарушении логики обмена сообщениями между объектами или в неправильной обработке ответов.

Как диаграмма последовательности помогает решить эту проблему:

Во-первых, она служит инструментом **визуальной отладки**. Команде следует построить диаграмму именно для того «ошибочного» сценария, который упоминает разработчик. Когда весь путь данных прорисован по шагам, часто обнаруживается, что один из объектов отправляет запрос раньше, чем получил необходимые данные от другого, или что цепочка вызовов обрывается на полпути. Это позволяет точно локализовать этап, на котором логика программы расходится с ожидаемым поведением.

Во-вторых, диаграмма помогает выявить **«тихие сбои»**. Бывает так, что функция не «падает» с ошибкой, а просто возвращает некорректный результат. На диаграмме это будет выглядеть как возврат сообщения (return message) с пустым значением или ошибочным статусом. Прорисовывая это, команда может увидеть, какой именно компонент системы первым генерирует неверный ответ, спровоцировавший общую ошибку.

В-третьих, использование **рамок взаимодействия (interaction frames)** позволяет разобрать «условные» баги. Если ошибка возникает только при определенном условии (например, только для незарегистрированных пользователей), рамка **alt** на диаграмме наглядно покажет, по какому пути идут сообщения в «хорошем» и «плохом» случаях. Это лучший способ обсудить проблему с командой: вместо чтения сотен строк кода все смотрят на одну схему, где стрелка, ведущая к ошибке, становится очевидной.