Билет №1

1. Классификация парадигм программирования. Роль парадигм программирования в процессе разработки ПО. Связь между парадигмами программирования и предметными областями, архитектурами вычислительных систем.
2. Выбор архитектуры программного обеспечения в зависимости от целевых задач и предметной области.

---------------------------------------------------------------------------------------------------------------------------------------------------------------Ответ:

**1. Классификация парадигм программирования**

— Императивные (процедурная, ООП): управление состоянием через последовательность инструкций.

— Деклагтивные (функциональная, логическая): описание результата без явного указания шагов.

Роль: управление сложностью кода, обеспечение соответствия между вычислительными моделями и задачами.

Связи:

— С предметными областями: ООП → инкапсуляция состояний, функциональная → потоковые/символьные вычисления.

— С архитектурами: распределенные системы → акторные модели, многопоточность → иммутабельность.

**2. Выбор архитектуры ПО**

Зависит от:

— Требований к масштабируемости (микросервисы, монолит).

— Предметной специфики (событийная → асинхронные данные, клиент-сервер → централизация).

— Ограничений среды (ресурсоемкие задачи → низкоуровневая оптимизация, распределенные системы → гибридные модели).

Билет №2

1. Приемы и подходы, обеспечивающие гибкое построение качественного кода. Понятие «Чистая архитектура».
2. Определение основных программных объектов, их функций и взаимодействий. Формирование поведенческих моделей.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

**3. Приемы гибкого кода и «Чистая архитектура»**  
— **Принципы**:

* *SOLID* (разделение ответственности, инверсия зависимостей),
* *DRY/KISS* (минимизация дублирования, упрощение логики).  
  — **Практики**:
* Паттерны проектирования (Фабрика, Адаптер),
* Тестирование (TDD, модульные тесты).  
  **Чистая архитектура** (Р. Мартин):
* Слоистая структура (домен → инфраструктура),
* Независимость от внешних компонентов (БД, UI),
* Управление зависимостями через инверсию.

**4. Определение объектов и поведенческие модели**  
— **Программные объекты**:

* Абстракции предметной области (сущности, сервисы),
* Функции: инкапсуляция данных, выполнение операций.  
  — **Взаимодействия**:
* Шаблоны (Наблюдатель, Стратегия),
* Контракты (интерфейсы, API).  
  — **Поведенческие модели**:
* UML-диаграммы (последовательностей, состояний),
* DDD (тактическое проектирование: агрегаты, доменные события).

Билет №3

1. Система управления версиями -git, особенности использования. Стратегии ветвления. Использование сторонних репозиториев.
2. Сравнение водопадной и итеративной моделей разработки ПО. Гибкие методы разработки (Agile).Манифест гибкой разработки и принципы. Основные характеристики различных гибких подходов. Причины популярности. Недостатки. Область применения.

---------------------------------------------------------------------------------------------------------------------------------------------------------------Ответ:

**5. Система управления версиями Git**  
— **Особенности**:

* Распределенная архитектура (локальные/удаленные репозитории),
* Низкоуровневый контроль изменений (коммиты, хеширование),
* Поддержка нелинейной разработки (ветвление, слияние).  
  — **Стратегии ветвления**:
* *Git Flow*: разделение на фичи, релизы, хотфиксы (долгие циклы),
* *GitHub Flow*: упрощенный подход (ветка → PR → main),
* *Trunk-Based*: минимум веток, частые коммиты в main (CI/CD).  
  — **Сторонние репозитории**:
* Интеграция через HTTPS/SSH,
* Submodules (вложенные проекты),
* Пакетные менеджеры (npm, Maven) + приватные репозитории (Artifactory).

**6. Водопадная vs. Итеративная модели и Agile**  
— **Сравнение моделей**:

* *Водопадная*: линейные этапы (требования → тестирование), жесткость, низкая адаптивность,
* *Итеративная*: цикличность (прототипы → доработки), гибкость, быстрая обратная связь.  
  — **Agile**:
* *Манифест*:
  + Люди и взаимодействия > процессы,
  + Рабочее ПО > документация,
  + Сотрудничество с заказчиком > контракты,
  + Готовность к изменениям > следование плану.
* *Принципы*: инкрементальная поставка, самоорганизация команд, рефакторинг.  
  — **Гибкие подходы**:
* *Scrum*: спринты, роли (Product Owner, Scrum Master), артефакты (бэклог),
* *Kanban*: визуализация потока (доска), WIP-лимиты,
* *XP*: парное программирование, TDD, непрерывная интеграция.  
  — **Преимущества Agile**: адаптивность, снижение рисков, фокус на ценности.  
  — **Недостатки**: сложность масштабирования, зависимость от дисциплины команды, риск «бесконечных» доработок.  
  — **Области применения**: стартапы, динамичные рынки, продукты с нечеткими требованиями. Водопад — регуляторные проекты (медицина, авионика), долгосрочные контракты.

Билет №4

1. Контрактное программирование и проектирование по контракту. Решаемые проблемы, особенности реализации и использования в различных языках программирования.
2. Scrum-артефакты, активности и роли. Проблемы внедрения. Выбор методологии для проекта, этапа жизненного цикла проекта.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответ:

**7. Контрактное программирование**  
— **Суть**: формализация обязательств между компонентами через *предусловия*, *постусловия* и *инварианты*.  
— **Решаемые проблемы**:

* Ошибки на границах модулей (некорректные входные данные, нарушение состояний),
* Упрощение отладки за счет явных ограничений.  
  — **Реализация**:
* *Eiffel*: встроенная поддержка (ключевые слова require, ensure),
* *D*: атрибуты @pre, @post,
* *Java/C#*: библиотеки (Spring Contracts, Code Contracts) или Assertion-механизмы.  
  — **Особенности**:
* Повышает надежность и читаемость кода,
* Увеличивает накладные расходы на проверки (требует баланса между строгостью и производительностью).

**8. Scrum: артефакты, активности, роли**  
— **Артефакты**:

* *Product Backlog*: иерархизированный список требований,
* *Sprint Backlog*: задачи текущего спринта,
* *Инкремент*: рабочий продукт по итогам спринта.  
  — **Активности**:
* *Sprint Planning*: определение целей и задач спринта,
* *Daily Scrum*: синхронизация команды (15 мин),
* *Sprint Review*: демонстрация результатов,
* *Retrospective*: анализ улучшений процесса.  
  — **Роли**:
* *Product Owner*: управление бэклогом, приоритезация,
* *Scrum Master*: устранение препятствий, соблюдение процесса,
* *Команда разработки*: реализация задач.  
  — **Проблемы внедрения**:
* Сопротивление изменениям в культуре команды,
* Недостаточное понимание ролей (например, смешение функций PO и SM),
* Формальное использование артефактов без фокуса на ценность.  
  — **Выбор методологии**:
* Зависит от сложности проекта, уровня неопределенности требований, зрелости команды.
* Scrum подходит для продуктов с динамично меняющимся контекстом, Kanban — для поддержки и постепенных улучшений.  
  — **Этапы жизненного цикла**:
* В Scrum этапы итеративны (спринты = микроциклы),
* В классических моделях (водопад) — последовательные стадии (анализ, дизайн, разработка, тестирование, поддержка).

Билет №5

1. Сравнение механизмов «сборки мусора» в С++ и Java. Особенности JVM-принципы работы компилятора времени исполнения и динамического профилировщика, алгоритмы сборки мусора в JVM.
2. Типичная схема процесса управления проектом. Варианты организации персонала. Инструментальные средства поддержки управления проектом.

Ответы:

**9. Сборка мусора в C++ vs Java, JVM и алгоритмы GC**  
— **С++**:

* Ручное управление памятью через new/delete, умные указатели (unique\_ptr, shared\_ptr).
* Нет встроенного GC → риск утечек/ошибок, но полный контроль над производительностью.  
  — **Java**:
* Автоматический GC → безопасность, но возможны паузы (stop-the-world).
* **JVM-механизмы**:
  + **JIT-компилятор**: преобразует байт-код в машинный код с оптимизацией "на лету".
  + **Профилировщик**: собирает runtime-метрики для оптимизации кода (например, HotSpot).
* **Алгоритмы GC**:
  + **Generational**: разделение объектов на Young/Old поколения (Minor/Major сборки).
  + **G1 (Garbage-First)**: баланс между задержками и пропускной способностью.
  + **ZGC/Shenandoah**: низколатентные алгоритмы для больших объемов памяти.

**10. Управление проектами**  
— **Типовая схема процесса**:

1. Инициация (цели, стейкхолдеры).
2. Планирование (ресурсы, риски, график).
3. Выполнение (реализация задач).
4. Мониторинг (отслеживание прогресса, корректировки).
5. Завершение (документация, ретроспектива).  
   — **Организация персонала**:

* **Функциональная**: разделение по специализациям (разработка, тестирование).
* **Матричная**: двойное подчинение (проектный + функциональный менеджер).
* **Проектная**: временные команды под конкретный продукт.  
  — **Инструменты**:
* Планирование: MS Project, Jira, Asana.
* Коммуникация: Slack, Teams.
* Контроль версий: GitLab, GitHub.
* CI/CD: Jenkins, GitLab CI.  
  — **Ключевой аспект**: выбор методологии (Agile, Waterfall) в зависимости от гибкости требований и масштаба проекта.

Билет №6

1. Умные указатели – решаемые проблемы, особенности реализации. Идиома RAП – примеры, проблемы.
2. Документирование процесса разработки ПО. Стандарты документация. Управление документацией. Согласованность и целостность документации.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**11. Умные указатели и RAII**  
— **Умные указатели** (C++):

* **Решаемые проблемы**:
  + Утечки памяти (неосвобожденные ресурсы),
  + Висячие ссылки (dangling pointers),
  + Двойное освобождение памяти.
* **Типы**:
  + unique\_ptr: эксклюзивное владение (перемещение, запрет копирования),
  + shared\_ptr: разделяемое владение через счетчик ссылок,
  + weak\_ptr: доступ к ресурсу без увеличения счетчика (для избежания циклических зависимостей).
* **Особенности реализации**:
  + Перегрузка операторов (\*, ->),
  + Использование шаблонов для обобщения типов.

— **Идиома RAII (Resource Acquisition Is Initialization)**:

* **Суть**: привязка жизненного цикла ресурса (память, файлы, сокеты) к времени жизни объекта.
* **Примеры**:
  + std::fstream (автоматическое закрытие файла),
  + std::lock\_guard (автоматическое освобождение мьютекса).
* **Проблемы**:
  + Неправильная реализация копирования/перемещения (риск двойного освобождения),
  + Ограниченная применимость в языках без деструкторов (например, Java).

**12. Документирование процесса разработки ПО**  
— **Стандарты документации**:

* **ISO/IEC/IEEE 26530**: спецификации требований и архитектуры,
* **ISO/IEC 12207**: жизненный цикл ПО,
* **Внутрикорпоративные стандарты** (Google, Microsoft).  
  — **Типы документации**:
* Техническая (API, архитектура),
* Пользовательская (руководства, FAQ),
* Процессная (планы, отчеты).  
  — **Управление документацией**:
* **Версионирование**: интеграция с Git, Confluence,
* **Генерация**: инструменты (Doxygen, Sphinx, Swagger),
* **Хранение**: централизованные репозитории (SharePoint, Notion).  
  — **Согласованность и целостность**:
* Автоматическая проверка через CI/CD (соответствие кода и комментариев),
* Регулярные ревью документации,
* Использование единого глоссария и шаблонов.  
  — **Проблемы**:
* Устаревание документации при быстрых изменениях кода,
* Субъективность описаний (особенно в архитектурных решениях).

Билет №7

1. Организация модульного тестирования. Примеры использования модульного тестирования и различных парадигмах.
2. Документирование архитектурных решений. Диаграммы развертывания.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**13. Организация модульного тестирования**  
— **Цель**: проверка отдельных компонентов (функций, классов) на корректность.  
— **Принципы**:

* Изоляция тестов (моки, стабы),
* Автоматизация (интеграция в CI/CD),
* Покрытие кода (метрики вроде line/branch coverage).  
  — **Примеры по парадигмам**:
* *Процедурная*: тестирование функций (напр., add(x, y)),
* *ООП*: проверка методов классов и инкапсуляции,
* *Функциональная*: тестирование чистых функций и композиций (напр., Haskell + QuickCheck).  
  — **Инструменты**: JUnit (Java), pytest (Python), RSpec (Ruby).

**14. Документирование архитектурных решений**  
— **Методы**:

* *ADL* (Architecture Description Language): формальное описание компонентов и связей,
* *Диаграммы*: UML (классов, последовательностей), C4-модель.  
  — **Диаграммы развертывания (UML)**:
* **Элементы**:
  + Узлы (серверы, устройства),
  + Артефакты (исполняемые файлы, БД),
  + Связи (протоколы, зависимости).
* **Цель**: визуализация физического размещения компонентов в среде выполнения.  
  — **Инструменты**: PlantUML, Draw.io, Enterprise Architect.  
  — **Управление**:
* Версионирование диаграмм (вместе с кодом),
* Синхронизация с документацией (Confluence, Markdown),
* Валидация через ревью с архитекторами.

Билет №8

1. Архитектурные шаблоны проектирования. Примеры, назначение, решаемые задачи.
2. Парадигмы программирования. Факторы, обуславливающие популярность парадигм программирования в конкретных предметных областях.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**15. Архитектурные шаблоны проектирования**  
— **Назначение**: стандартизация решений для типовых проблем проектирования, управление сложностью системы.  
— **Примеры и решаемые задачи**:

* **Слоистая архитектура**:
  + *Задача*: разделение ответственности (UI, бизнес-логика, БД).
  + *Пример*: веб-приложения (презентационный слой → слой услуг → DAL).
* **MVC (Model-View-Controller)**:
  + *Задача*: декoupling данных, логики и представления.
  + *Пример*: фреймворки (Ruby on Rails, Angular).
* **Микросервисы**:
  + *Задача*: масштабируемость и изоляция сервисов.
  + *Пример*: распределенные системы (Netflix, Uber).
* **Событийно-ориентированная архитектура**:
  + *Задача*: обработка асинхронных событий (IoT, трейдинг).
  + *Пример*: Apache Kafka, RabbitMQ.
* **Шина данных (Event Bus)**:
  + *Задача*: централизованное управление коммуникацией компонентов.

**16. Парадигмы программирования и их популярность**  
— **Факторы популярности**:

* **Предметная специфика**:
  + *ООП* → GUI, игры (инкапсуляция состояний, наследование),
  + *Функциональное* → Big Data, ML (иммутабельность, параллелизм),
  + *Логическое* → экспертные системы (Prolog для правил и выводов).
* **Производительность**:
  + *Императивное* (C, Rust) → системы реального времени, драйверы.
* **Экосистема и инструменты**:
  + *JavaScript* → веб (поддержка событийно-ориентированной модели),
  + *Python* → научные вычисления (библиотеки: NumPy, Pandas).
* **Управление сложностью**:
  + *Декларативное* (SQL, HTML) → фокус на "что", а не "как".  
    — **Ключевой принцип**: выбор парадигмы определяется балансом между требованиями предметной области, ресурсными ограничениями и экосистемой языка.

Билет №9

1. Паттерны проектирования. Применение паттернов при разработке программ.
2. Статическая и динамическая модели системы. Диаграммы активностей, диаграммы классов.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**17. Паттерны проектирования**  
— **Определение**: типовые решения повторяющихся проблем проектирования, обеспечивающие гибкость и поддерживаемость кода.  
— **Категории**:

* *Порождающие* (Creational):
  + **Singleton**: гарантирует единственный экземпляр класса (напр., логгер, конфигуратор).
  + **Factory Method**: делегирует создание объектов подклассам (инкапсуляция логики инициализации).
* *Структурные* (Structural):
  + **Adapter**: обеспечивает совместимость интерфейсов (интеграция legacy-кода).
  + **Decorator**: динамическое добавление функциональности (напр., кэширование, логирование).
* *Поведенческие* (Behavioral):
  + **Observer**: уведомление зависимых объектов об изменениях (напр., UI-события).
  + **Strategy**: инкапсуляция алгоритмов для взаимозаменяемости (сортировка, валидация).  
    — **Применение**:
* Устранение жестких зависимостей (Dependency Injection),
* Оптимизация взаимодействия компонентов (Facade для упрощения сложных систем),
* Повышение тестируемости (Mock-объекты через паттерн Proxy).

**18. Статическая и динамическая модели системы**  
— **Статическая модель**:

* **Описание**: структура системы (классы, атрибуты, связи, интерфейсы).
* **Диаграммы классов (UML)**:
  + Классы с полями/методами,
  + Ассоциации (наследование, агрегация, композиция),
  + Пример: модель БД (сущности и их связи).  
    — **Динамическая модель**:
* **Описание**: поведение системы во времени (взаимодействия, состояния, процессы).
* **Диаграммы активностей (UML)**:
  + Визуализация потоков операций (напр., бизнес-процесс "Оформление заказа"),
  + Узлы: действия, ветвления (decision nodes), параллельные потоки (fork/join).  
    — **Сравнение**:
* *Статика*: "Что есть в системе?" (архитектура),
* *Динамика*: "Как система работает?" (сценарии использования).  
  — **Инструменты**:
* UML-редакторы (Enterprise Architect, Lucidchart),
* Генерация кода из диаграмм (напр., Hibernate для ORM на основе моделей классов).  
  — **Роль в разработке**:
* Статическая модель → основа для реализации,
* Динамическая модель → валидация логики через сценарии.

Билет №10

1. Системы статического анализа кода и поиска утечек памяти, их возможности -valgrind, VLD.
2. Регулярные выражения. Решаемые проблемы, особенности применения.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**19. Системы анализа кода и поиска утечек**  
— **Статический анализ** (без запуска кода):

* *Цель*: поиск потенциальных ошибок (утечки, небезопасные конструкции).
* *Инструменты*: Clang Static Analyzer, PVS-Studio.  
  — **Динамический анализ** (в runtime):
* **Valgrind**:
  + Обнаружение утечек памяти (Memcheck),
  + Анализ использования неинициализированных данных,
  + Профилирование производительности (Cachegrind).
* **Visual Leak Detector (VLD)**:
  + Интеграция с Visual Studio для C/C++,
  + Детектирование утечек в Windows-приложениях.

**20. Регулярные выражения**  
— **Решаемые проблемы**:

* Поиск/замена шаблонов в тексте (например, email, даты),
* Валидация форматов данных,
* Парсинг структурированных логов или конфигов.  
  — **Особенности применения**:
* **Синтаксис**: метасимволы (.\*, \d, []), группы, квантификаторы,
* **Производительность**: избегать сложных шаблонов (например, backtracking),
* **Поддержка**: языки (Python, JavaScript), инструменты (grep, sed).  
  — **Примеры**:
* ^\w+([.-]?\w+)\*@\w+([.-]?\w+)\*(\.\w{2,3})+$ → валидация email,
* (\d{2})-(\d{2})-(\d{4}) → извлечение даты из строки.

Билет №11

1. Развитие языков программирования. Цели, факторы, парадигмы.
2. Язык UML. Способы применения. Валидация архитектурных решений.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**21. Развитие языков программирования**  
— **Цели**:

* Повышение выразительности и читаемости кода,
* Оптимизация производительности и безопасности,
* Адаптация к новым вычислительным моделям (распределенные системы, квантовые вычисления).  
  — **Факторы развития**:
* *Технологические*: эволюция аппаратуры (многоядерные CPU, GPU),
* *Практические*: потребности индустрии (веб, ML, IoT),
* *Теоретические*: исследования в области формальных методов и парадигм.  
  — **Влияние парадигм**:
* Мультипарадигменность (Python, Scala) → гибкость в выборе подходов,
* Доминирование ООП → стандарт для корпоративных приложений,
* Рост функциональных элементов в императивных языках (Java Streams, C++ лямбды).

**22. Язык UML**  
— **Способы применения**:

* **Визуализация**:
  + Диаграммы классов (статическая структура),
  + Диаграммы последовательностей (динамическое взаимодействие),
  + Диаграммы развертывания (физическая инфраструктура).
* **Проектирование**:
  + Моделирование требований (Use Case),
  + Описание состояний системы (State Machine).
* **Документирование**:
  + Стандартизация описания архитектуры,
  + Создание технической спецификации для разработчиков.  
    — **Валидация архитектурных решений**:
* Проверка согласованности диаграмм (напр., соответствие классов и последовательностей),
* Ревью с заинтересованными сторонами (архитекторы, разработчики),
* Инструменты анализа (напр., проверка цикломатической сложности на диаграммах активностей).  
  — **Ключевые аспекты**:
* UML как "мост" между бизнес-требованиями и технической реализацией,
* Поддержка Agile через итеративное уточнение моделей.

Билет №12

1. Непрерывная интеграция (continuous integration). Разработка через тестирование (TDD).
2. Шаблоны проектирования: Итератор, Адаптер. Решаемые проблемы, особенности и проблемы реализации.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**23. Непрерывная интеграция (CI) и TDD**  
— **Непрерывная интеграция (CI)**:

* **Цель**: автоматизация сборки, тестирования и интеграции кода при каждом коммите.
* **Принципы**:
  + Частые коммиты в основную ветку,
  + Автоматизированные тесты и сборка,
  + Быстрое обнаружение конфликтов/ошибок.
* **Инструменты**: Jenkins, GitHub Actions, GitLab CI.
* **Преимущества**: снижение рисков интеграции, ускорение доставки кода.

— **TDD (Test-Driven Development)**:

* **Цикл**:
  1. Написание теста (на ещё не реализованный функционал),
  2. Реализация минимального кода для прохождения теста,
  3. Рефакторинг (оптимизация без изменения поведения).
* **Преимущества**:
  1. Высокое покрытие тестами,
  2. Четкие требования через тесты,
  3. Снижение числа дефектов.
* **Сложности**:
  1. Требует дисциплины и времени,
  2. Не всегда применим (напр., UI-логика).

**24. Шаблоны: Итератор и Адаптер**  
— **Итератор (Iterator)**:

* **Цель**: предоставить унифицированный интерфейс для обхода коллекций.
* **Решаемые проблемы**:
  + Сокрытие реализации структуры данных,
  + Упрощение клиентского кода.
* **Пример**: Iterator в Java (обход элементов List или Set).
* **Особенности реализации**:
  + Поддержка разных типов итераторов (напр., обратный обход),
  + Потокобезопасность (если требуется).
* **Проблемы**:
  + Усложнение кода для простых коллекций,
  + Ограничения в многопоточной среде.

— **Адаптер (Adapter)**:

* **Цель**: обеспечить совместимость несовместимых интерфейсов.
* **Решаемые проблемы**:
  + Интеграция legacy-кода или сторонних библиотек,
  + Устранение зависимости от конкретных реализаций.
* **Пример**: адаптер для преобразования данных из XML в JSON-формат.
* **Типы**:
  + *Объектный адаптер*: оборачивает целевой объект,
  + *Классовый адаптер*: использует множественное наследование (если разрешено языком).
* **Проблемы**:
  + Риск создания "божественного объекта" (нарушение SRP),
  + Накладные расходы на обертки.

**Ключевой вывод**:

* **Итератор** → стандартизация доступа к данным.
* **Адаптер** → обеспечение взаимодействия компонентов с разными интерфейсами.

Билет №13

25. Инструментальная поддержка модульного тестирования консольных приложений, приложений с графическим интерфейсом, сайтов. Тестирование API Google Postman.

26. Соглашения о кодировании: назначения; развитие; факторы; влияющие на развитие. Системы генерации документации на основе исходного кода: назначения, возможности, примеры синтаксических конструкций.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

Ответы:

**25. Инструменты модульного тестирования и Postman**  
— **Консольные приложения**:

* **Инструменты**:
  + *pytest* (Python) — тестирование функций и CLI-скриптов с параметризацией и проверкой вывода.
  + *JUnit* (Java) — создание тестовых сценариев для проверки логики консольных утилит.
* **Подходы**: запуск команд через эмуляцию ввода, анализ выходных данных и кодов завершения.

— **Приложения с графическим интерфейсом (GUI)**:

* **Инструменты**:
  + *TestFX* (Java) — автоматизация действий пользователя (клики, ввод) для JavaFX-приложений.
  + *Appium* — кроссплатформенное тестирование мобильных и десктопных интерфейсов.
* **Особенности**: эмуляция взаимодействия с элементами интерфейса (кнопки, поля ввода).

— **Веб-сайты**:

* **Инструменты**:
  + *Cypress/Jest* (JavaScript) — тестирование рендеринга страниц, обработки событий и API-запросов.
  + *Selenium* — автоматизация браузеров для проверки кросс-платформенной совместимости.

— **Тестирование API через Postman**:

* **Функционал**:
  + Создание коллекций запросов с параметрами и переменными,
  + Автоматическая валидация ответов (проверка статус-кодов, структуры JSON/XML),
  + Интеграция с CI/CD через Newman для выполнения тестов в пайплайнах.

**26. Соглашения о кодировании и генерация документации**  
— **Соглашения о кодировании**:

* **Назначение**:
  + Унификация стиля кода для повышения читаемости,
  + Снижение ошибок из-за неочевидных практик.
* **Факторы развития**:
  + *Языковые стандарты* (например, PEP8 для Python),
  + *Корпоративные требования* (Google, Microsoft),
  + *Инструменты линтинга* (ESLint, flake8).
* **Примеры правил**:
  + Ограничение длины строки,
  + Правила именования переменных и функций,
  + Рекомендации по форматированию (отступы, пробелы).

— **Генерация документации из кода**:

* **Инструменты**:
  + *Doxygen* — создание документации для C++, Java и других языков на основе аннотаций в комментариях.
  + *Javadoc* — генерация API-документации для Java-классов и методов.
  + *Sphinx* — работа с Python-проектами, поддержка reStructuredText и Markdown.
* **Принципы**:
  + Использование специальных тегов (например, @param, @return) для описания параметров и возвращаемых значений,
  + Автоматическое построение навигации и перекрестных ссылок,
  + Экспорт в форматы HTML, PDF или LaTeX.
* **Преимущества**:
  + Синхронизация документации с кодом,
  + Упрощение поддержки за счет централизации информации.