[**1. Жизненный цикл ПО. Понятие жизненного цикла ПО, этапы, их описание в терминологии различных стандартов (примеры).**](#_js8td1l8av5n) **4**

[**2.Процессы создания ПО. Управление процессами, 5 групп процессов, их характерные особенности.**](#_ha4vfsdfen86) **5**

[**3. Проект. Управление проектами. Понятие проектного офиса, типы проектных офисов. Программы и портфели.**](#_jnlglt66rnde) **6**

[**4. Требования к ПО. Виды требований. Техники сбора и описания.**](#_t66lmjdd4fec) **7**

[**5. Проектное управление. Ограничения и жизненный цикл проекта**](#_jc1e53l7jctx) **10**

[**6. Проектное управление. Методологии. PMBOOK и PRINCE2**](#_qdj84us7ep87) **11**

[**7. Гибкие принципы управление. Agile – идеи и принципы.**](#_g0w6stcyrjkh) **15**

[**8. Гибкие технологии. Scrum**](#_p52lko6f03wi) **16**

[**9. Гибкие технологии. Kanban**](#_e0juvhtct595) **17**

[**10. Масштабирование гибкой разработки. Проблемы и решения. SAFe как пример масштабируемого фреймворка**](#_bg39643j6c47) **18**

[**11. Гибкое управление. Особенности организации работы дистанционно работающих команд**](#_6heo2qw6jhtc) **19**

[**12. Поток поставки ценности. DevOps – понятия и основные принципы**](#_w8alc4az62gu) **20**

[**13. DevOps. Компоненты и процессы. Понятие безопасной разработки**](#_p842zc5uc9rv) **22**

[**14. Документация. Типы, назначение. Принципы создания, группы заинтересованных лиц, методики формирования пакетов документации.**](#_icyt2k36lstg) **24**

[**15. Тестирование ПО. Цели и принципы**](#_ufkijamohfex) **27**

[**16. Тестирование. Уровни и типы тестирования.**](#_mczmm9yphy88) **29**

[**17. Тестирование. Методы черного и белого ящика. Статическое и динамическое тестирование**](#_milf5hsf88je) **31**

[**18. Тестирование. Методы и инструменты. Классификация дефектов**](#_buhu3pawp067) **33**

[**19. Сопровождение и поддержка ПО. ITSM и ITIL общие понятия и принципы**](#_ew3lk87otqy8) **36**

[**20. Сопровождение и поддержка. ITIL. Базовые процессы**](#_s316o04uwjnl) **38**

[**21. Управление изменениями. ITIL и процесс управления изменениями**](#_xkfdz8xt5hz5) **40**

[**22. Процесс разработки. Архитектура решения. Назначение, принципы выбора. Анализ и оценка**](#_62m3kkqjaltt) **41**

[**23. Процесс разработки. Архитектура. Документирование и распространение сведений. Группы заинтересованных лиц.**](#_1zql04geu98c) **43**

[**24. Атрибуты качества систем и их связь с функциональными требованиями. Сценарии атрибутов качества**](#_xoefbm6hu7eo) **45**

[**25. Атрибуты качества и сценарии - подробное описание на примере одного из типа сценариев (Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)**](#_2c4xmqcjpkhv) **46**

[**26. Тактики обеспечения атрибутов качества (на примере одного из типов Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)**](#_jybc1sgu2o4c) **50**

[**27. Документирование программных систем. Заинтересованные лица, представления и нотации (с примерами).**](#_jtnkiigu7c7x) **52**

[**28. Документирование программных систем. Неформальные нотации. Плюсы и минусы, области применения**](#_lpc2jjku08o8) **55**

[**29. Документирование программных систем. Частично формальные и формальные нотации. Плюсы и минусы, области применения**](#_k5w58n89iooc) **57**

[**30. Нотации. Рассмотреть пример одной из нотаций на выбор. Основные типы диаграмм, области применения**](#_ik8yin3z1vw6) **59**

[**31. Стили и шаблоны проектирования. Понятие архитектурного стиля и архитектурного шаблона**](#_osfh3690652a) **61**

[**32. Архитектурные принципы проектирование. Основные принципы и подробно на примере 1 – 2 выбранных**](#_zc4wj2lvoaeq) **62**

[**33. Архитектурные стили. Уровни применения, типичные представителе (на примере выбранного - монолит, SOA, Микросервисы...)**](#_jqeztlosevs2) **64**

[**34. Архитектурные стили на уровне компонентов. Слоистая и модульная архитектура**](#_s87551moxkb0) **66**

[**35. Паттерны проектирования. Общие понятия, назначение, плюсы и минусы.**](#_6f0jzkieytpr) **68**

[**36. Паттерны проектирования. Порождающие паттерны (на выбранном примере)**](#_ghbgj9wlrtd5) **69**

[**37. Паттерны проектирования. Структурные паттерны (на выбранном примере)**](#_9xq9u5gdy1id) **71**

[**38. Паттерны проектирования. Поведенческие паттерны (на выбранном примере)**](#_pnkn98b8f7im) **74**

[**39. Управление качеством. Основные понятия, процессы и практики**](#_sp8so8r7ztlv) **77**

[**40. Управление качеством. Стандарты и модели**](#_1v9dq0d2knt2) **78**

[**41. Управление качеством. Эталонная модель**](#_x657oeoxsmyj) **80**

[**42. Методологии управления качеством. Классификация**](#_hxqymkngj2j3) **81**

[**43. Рефакторинг. Назначение и принципы**](#_4i51rkpnb969) **83**

[**44. Рефакторинг. Техники рефакторинга (на примере одной из техник)**](#_wpuz4nudoi0w) **84**

[**45. Метрики программного кода. Основные принципы и подходы, инструменты, практика применения.**](#_k55sjwuvtxrr) **86**

## 1. Жизненный цикл ПО. Понятие жизненного цикла ПО, этапы, их описание в терминологии различных стандартов (примеры).

**Жизненный цикл программного обеспечения** (Software Life Cycle Model) — это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

Все проекты могут иметь следующую структуру жизненного цикла:

* начало проекта;
* организация и подготовка;
* выполнение работ проекта;
* завершение проекта.

| Жизненный цикл по ISO/IEC 12207:   1. Формирование требований 2. Проектирование 3. Реализация 4. Тестирование 5. Ввод в действие 6. Эксплуатация и сопровождение 7. Вывод из эксплуатации   Жизненный цикл по ГОСТ Р 53622-2009:   1. Научно-исследовательские работы 2. Согласование и утверждение ТЗ 3. Проектирование 4. Реализация 5. Внедрение 6. Эксплуатация 7. Сопровождение 8. Снятие с эксплуатации | Жизненный цикл по ISO/IEC 15288:   1. Формирование концепции 2. Разработка 3. Реализация 4. Эксплуатация 5. Поддержка 6. Вывод из эксплуатации   Жизненный цикл по Oracle CDM:   1. Определение требований 2. Анализ 3. Проектирование 4. Реализация 5. Внедрение 6. Эксплуатация   Жизненный цикл услуг по ITSM:   1. Планирование 2. Моделирование 3. Внедрение 4. Эксплуатация 5. Непрерывное улучшение |
| --- | --- |

## 

## 2.Процессы создания ПО. Управление процессами, 5 групп процессов, их характерные особенности.

**Процесс создания ПО** – совокупность процессов, приводящих к созданию программного продукта.

Управление процессами в рамках менеджмента состоит в осуществлении определенным образом воздействий на составные части процесса для обеспечения исполнения предъявленных к процессу требований.

5 групп процессов:

* Инициация: процессы, выполняемые для определения нового проекта или новой фазы существующего проекта путем получения авторизации на начало проекта или фазы;
* Планирование: процессы, требуемые для установления содержания работ, уточнения целей и определения направления действий, требуемых для достижения целей проекта;
* Исполнение: процессы, применяемые для выполнения работ, указанных в плане управления проектом, с целью соответствия спецификациям проекта;
* Мониторинг и контроль: процессы, требуемые для отслеживания, анализа, а также регулирования исполнения проекта; выявления областей, требующих внесения изменений в план; и инициирования соответствующих изменений,
* Закрытие: процессы, выполняемые для завершения всех операций в рамках всех групп процессов в целях формального закрытия проекта или фазы.

## 

## 3. Проект. Управление проектами. Понятие проектного офиса, типы проектных офисов. Программы и портфели.

**Офис управления проектами/Проектный офис** — организационная структура, стандартизирующая процессы руководства проектами и способствующая обмену ресурсами, методологиями, инструментами и методами. Сфера ответственности ОУП может варьироваться от оказания поддержки в управлении проектами до прямого управления одним или более проектами.

**Типы проектных офисов:**

* Поддерживающий. Играет консультативную роль, предоставляя шаблоны, лучшие практики, обучение, доступ к информации и уроки, извлеченные из других проектов. Данный тип служит в качестве хранилища проекта. Степень контроля со стороны Проектного офиса низкая.
* Контролирующий. Предоставляет поддержку и требует соответствия требованиям с помощью различных средств. Соответствие может предполагать адаптацию структур или методологий управления проектами, использование специфических шаблонов, форм и инструментов или соответствие требованиям руководства. Степень контроля со стороны Проектного офиса средняя.
* Руководящий. Контролирует проекты путем непосредственного управления данными проектами. Степень контроля со стороны Проектного офиса высокая.

**Управление проектом** — это приложение знаний, навыков, инструментов и методов к работам проекта для удовлетворения требований, предъявляемых к проекту. Управление проектом осуществляется посредством надлежащего применения и интеграции логически сгруппированных 47 процессов управления проектом, объединенных в 5 групп процессов.

**Программа** — ряд связанных друг с другом проектов, подпрограмм и операций программы, управление которыми координируется для получения выгод, которые были бы недоступны при управлении ими по отдельности.

**Портфель** — это набор проектов, программ, подпортфелей и элементов операционной деятельности, управляемых как группа с целью достижения стратегических целей.

## 

## 4. Требования к ПО. Виды требований. Техники сбора и описания.

**Виды требований:** • Функциональные, • Требования к качеству, • Ограничения

**Задача** формирования требований - сбор целей и требований из различных источников

В соответствии с моделью **Кано**, в зависимости от степени влияния на удовлетворение, требования можно разделить на **три категории:** • базовые (ожидаемые); • основные (желаемые); • восхищающие (привлекательные).

**Различные техники сбора требования:**

* методы исследования (например, интервью, анкетирование);
* методы креативности (например, мозговой штурм, парадоксальный мозговой штурм, изменение перспективы, метод аналогии);
* документоориентированные методы (например, программная археология, «автозапись», повторно используемые требования, изучение существующей документации);
* методы наблюдения (например, полевые наблюдения, обучение);
* методы поддержки (например, ментальные карты, мастер-классы, карта "Класс-Ответственность-Кооперация", аудио- и видеозаписи, диаграммы вариантов использования, прототипы).

**Анкетирование** - данный способ подразумевает под собой составление листа-опросника (анкеты, брифа), который может содержать открытые и закрытые вопросы. Анкетирование используется для того, чтобы подтвердить или детализировать ранее известные требования, выбрать параметры для решений.

| **Преимущества**:  • Высокая скорость получения результатов; • Сравнительно небольшие материальные затраты. | **Недостатки**:  • Данный метод не подходит для выявления неявных требований; • При составлении опросника физически невозможно учесть все необходимые вопросы. |
| --- | --- |

**Интервью** - данный способ применяется, в основном, для получения информации по какой-либо конкретной теме и/или для уточнения требований. Необходимо гибко реагировать на реакцию интервьюируемого и в случае необходимости изменять порядок заготовленных вопросов или их формулировку.

| **Преимущества:**  • Возможность задавать вопросы в произвольной последовательности; • Возможность использовать вспомогательный материал; • Анализ невербальной реакции опрашиваемого человека, позволит сделать дополнительный вывод о достоверности его ответов | **Недостатки**:  • Интервью отнимает достаточно много времени и сил; • Дополнительной сложностью является получение одинаковых ответов от интервьюируемых. |
| --- | --- |

**«Автозапись»** - данный метод подразумевает под собой работу с записями, письмами (электронными письмами), а также с любым другим документом, автором которого является Заказчик или конечный пользователь.

| **Преимущество:**  • Помогает лучше понять сложные процедуры или процессы. | **Недостаток**:  • Данный метод сильно зависит от опыта Заказчика, а также от его умения формулировать и выражать свои мысли. |
| --- | --- |

**Изучение существующей документации** - данная методика может быть использована при наличии в организации документации, которая может помочь в определении потребностей Заказчика. Выявленные требования являются основой для дальнейшего анализа и должны быть детализированы. Данная методика применима, например, при автоматизации устоявшихся в организации регламентированных бизнес процессов.

| **Преимущество:**  • Быстрое получение информации. | **Недостаток**:  • Данный способ не применим при наличии в компании только базовых документов (или при их полном отсутствии) или, если в компании Заказчика не поддерживается актуальность документации. |
| --- | --- |

**Работа в «поле»** - работа «в поле» состоит из наблюдения за деятельностью и процессами будущих пользователей системы, и в определении требований, основанных на этом наблюдении. Метод позволяет избежать проблем, связанных с трудностями заказчиков в описании и выражении своих потребностей. В некоторых случаях процесс наблюдения может сопровождаться «интервьюированием» пользователей для уточнения особенностей и деталей их работы и задач.

| **Преимущества:**  • Позволяет наглядно увидеть проблему и разработать наиболее оптимальный вариант ее решения; • Помогает наиболее точно собрать требования, наблюдая за работой сотрудников. | **Недостатки:**  • В процессе наблюдения могут быть упущены некоторые альтернативные сценарии бизнес процесса; • Очень трудоемкий. |
| --- | --- |

**Мозговой штурм** - наиболее часто используемый метод получения требований, которые связаны с новыми или плохо изученными направлениями деятельности организации Заказчика или функциями системы. Во время мозгового штурма участники «накидывают» любые идеи, касающиеся решения данной проблемы.

| **Преимущества:**  • Можно проработать несколько различных вариантов решения заданной проблемы; • Можно заодно разрешить конфликты требований. | **Недостатки:**  • Участники мозгового штурма должны быть мотивированы на идеи; • Трудно применим в распределенных командах. |
| --- | --- |

**Диаграммы вариантов использования** или use cases позволяют собрать и сформировать функциональные требования от лица участников, определяют границы решения и показывают связи с внешними системами и участниками. Метод позволяет детализировать требования с точки зрения пользователей, а также помогает уточнить и систематизировать функционал, который требуется реализовать.

| **Преимущество:**  • Позволяет проработать все варианты развития сценария (основной и альтернативные сценарии). | **Недостаток:**  • Метод не применим для сбора нефункциональных требований. |
| --- | --- |

## 

## 5. Проектное управление. Ограничения и жизненный цикл проекта

**Проектное управление** является одним из двух основных видов управления разработкой (наряду с гибким управлением Agile). Стоит отметить, что НЕ существует идеальной системы управления проектами, подходящей для каждого из видов проектов. Также не существует системы, которая бы подходила каждому руководителю и была удобна для всех членов команды.

Под самим проектом понимается временное предприятие, направленное на создание уникального продукта, услуги или результата (улучшение существующего продукт или документ с результатами исследования).

**Ограничения проекта:**

● **Содержание** - содержание проекта, которое в простейшем случае может выглядеть как список работ по проекту, которые нужно реализовать, чтобы достигнуть целей проекта и получить запланированные результаты. Чем точнее описано содержание проекта, тем проще спрогнозировать сроки и бюджет проекта.

● **Качество** - ограничение, связанное со степенью выполнения требований к результатам проекта. Например, заказчик проекта может согласиться с тем, что его устроит ситуация, когда могут быть реализованы лишь самые важные требования к результатам проекта

● **Расписание** - ограничения по продолжительности проекта, включающее дату, к которой в проекте должны получить ожидаемые результаты.

● **Бюджет** - ограничение по стоимости проекта

● **Ресурсы** - ограничения по ресурсам оказывают самое сильное влияние на сроки выполнения работ, но могут повлиять и на бюджет проекта.

● **Риски** - изменение требований к проекту или целей проекта может вызвать дополнительные риски. Риски, в свою очередь, могут повлиять на содержание проекта, сроки или бюджет проекта

**Жизненный цикл проекта** — набор фаз, через которые проходит проект с момента его инициации до момента закрытия.

Все проекты могут иметь следующую структуру жизненного цикла:

* Начало проекта;
* Организация и подготовка;
* Выполнение работ проекта;
* Завершение проекта.

## 

## 6. Проектное управление. Методологии. PMBOOK и PRINCE2

**Project Management Body Of Knowledge (PMBOK).**

Управление проектом — это приложение знаний, навыков, инструментов и методов к работам проекта для удовлетворения требований, предъявляемых к проекту. Управление проектом осуществляется посредством надлежащего применения и интеграции логически сгруппированных 47 процессов управления проектом, объединенных в 5 групп процессов.

**5 групп процессов:**

* **Инициация**: процессы, выполняемые для определения нового проекта или новой фазы существующего проекта путем получения авторизации на начало проекта или фазы;
* **Планирование**: процессы, требуемые для установления содержания работ, уточнения целей и определения направления действий, требуемых для достижения целей проекта;
* **Исполнение**: процессы, применяемые для выполнения работ, указанных в плане управления проектом, с целью соответствия спецификациям проекта;
* **Мониторинг** и контроль: процессы, требуемые для отслеживания, анализа, а также регулирования исполнения проекта; выявления областей, требующих внесения изменений в план; и инициирования соответствующих изменений,
* **Закрытие**: процессы, выполняемые для завершения всех операций в рамках всех групп процессов в целях формального закрытия проекта или фазы.

**Ограничения проекта:**• содержание, • качество, • расписание, • бюджет, • ресурсы, • риски.

**Жизненный цикл проекта** — набор фаз, через которые проходит проект с момента его инициации до момента закрытия.

Все проекты могут иметь следующую структуру жизненного цикла:   
• начало проекта; • организация и подготовка; • выполнение работ проекта; • завершение проекта.

Область знаний является всеобъемлющей системой понятий, терминов и действий, составляющих профессиональную область, область управления проектами или область деятельности.

**Области знаний** включают в себя:

• управление интеграцией проекта; • управление содержанием проекта;

• управление сроками проекта; • управление стоимостью проекта;

• управление качеством проекта; • управление человеческими ресурсами проекта;

• управление коммуникациями проекта; • управление рисками проекта;

• управление закупками проекта; • управление заинтересованными

**Планирование**. Типы зависимостей операций.

• **Финиш-старт** (finish-start, FS). Логическая связь, при которой старт последующей операции зависит от финиша предшествующей операции.  
• **Финиш-финиш** (finish-finish, FF). Логическая связь, при которой финиш последующей операции зависит от финиша предшествующей операции.  
• **Старт-старт** (start-start, SS). Логическая связь, при которой старт последующей операции зависит от старта предшествующей операции.  
• **Старт-финиш** (start-finish, SF). Логическая связь, при которой финиш последующей операции зависит от старта предшествующей операции.

**PRojects IN Controlled Environments version 2 (PRINCE2).**

Методология PRINCE2 в отличие от свода знаний PMBOK **не содержит**:

• Специализированных аспектов управления проектом, например, отраслевых;

• Конкретных практик и инструментов управления проектами, таких как диаграмма Ганта, WBS и т.п.

PRINCE2 концентрируется на управленческих сторонах проекта, выраженных в 7 принципах, 7 процессах и 7 темах проекта.

• 7 принципов определяют общие правила управления проектами по PRINCE2, определяют базу методологии;

• 7 процессов определяют шаги продвижения по проектному циклу;

• 7 тем – аспекты, по которым проводится контроль для достижения успеха проекта.

В PRINCE2 более чётко определённая структура команды проекта, чем у большинства подходов к проектному управлению.

**Процессы управления.**

**Начало проекта (Starting up a project)** - в ходе данного процесса назначается менеджер проекта и определяются общие требования к характеристикам продукта. Менеджер проекта, чья основная задача – внимание к деталям, отчитывается перед Управляющим комитетом проекта, который отвечает за общее руководство проектом. Именно Управляющий комитет следит за тем, чтобы проект не сбился с курса, и он же полностью отвечает за успех проекта.

**Инициация проекта (Initiation a project)** - в ходе данного процесса менеджер проекта составляет «Документацию по инициации проекта», в которой содержится план проекта по стадиям. Стадии могут длиться разное количество времени, но, как и в классическом подходе, они следуют строго друг за другом.

**Руководство проектом (Directing a project)** - данный процесс предоставляет возможность Управляющему комитету нести общую ответственность за успех проекта, не погружаясь в детали, которые находятся в границах полномочий менеджера проекта.

**Контроль стадии (Controlling a stage)** - при реализации проекта, даже в идеальных условиях, будут вноситься определённые изменения. Процесс «Контроль стадии» реализует один из принципов PRINCE2 – принцип управления по исключениям. В обязанности менеджера проекта входит отслеживать в ходе выполнения стадии отклонения от плановых параметров проекта по срокам, содержанию, бюджету и др. Если эти отклонения превышают данные руководителю проекта Управляющим комитетом полномочия (в терминологии PRINCE2 – допуски), менеджер проекта обязан проинформировать Управляющий комитет и предложить пути выхода из ситуации.

**Управление созданием продукта (Managing Product Delivery):**

Процесс управления созданием продукта представляет собой взаимодействие менеджера проекта и менеджера команды по созданию одного из продуктов проекта. В обязанности менеджера проекта в данном процессе входит делегирование полномочий по созданию продукта менеджеру команды и приемка созданного продукта.

**Управление границами стадии (Managing a stage boundary):**

В ходе данного процесса менеджер проекта предоставляет Управляющему комитету всю необходимую информацию для оценки результатов пройденной стадии и принятия решения о переходе на следующую стадию.

**Завершение проекта (Closing a project):**

Одно из отличий PRINCE2 в том, что процесс завершения проекта не выделяется в отдельный этап или стадию, как в классическом подходе, а выполняется в рамках финальной стадии создания продукта. Цель процесса – подтвердить, что продукт проекта принят, или проект больше не может принести ничего полезного.

Ограничения проекта:

• содержание, • качество, • деньги, • время, • выгоды, • риски.

Компоненты методологии

**Обоснование проекта.** Стоит ли его запускать, получится ли сделать и какими методами. Если первый «зеленый свет» загорелся, то дальше анализируйте стоимость проекта, время его выполнения и поджидающие рисках.

**Организация.** Управляющий комитет контролирует работу руководителя проекта и отчитывается менеджменту компании.

**Планы.** Составляйте на разных стадиях проекта. Одно из свойств Prince2 — планирование на основе Характеристик продукта: почему его создают, какие материалы или навыки нужны. Это позволяет сразу сосредоточиться на нужном результате, а не вносить потом поспешные изменения.

**Контроль.** Убедитесь на данном этапе, что проект соответствует заданным критериям, на него не уходит больше времени и денег, чем планировалось, и шансы довести дело до победного конца — также высокие.

**Управление рисками.** Проведите заранее риск-анализ.

**Качество.** Потребитель должен получить именно тот продукт, который описывался в Характеристиках.

**Изменения.** Они, как и риски, неизбежны: подготовить полностью предсказуемый проект едва ли возможно.

Принципы

На каждом этапе проверяйте, остается ли проект целесообразным.

**Учитесь на ошибках.** Записывайте их в отдельный список, чтобы не допустить

на следующей стадии и в следующем проекте.

**Четко указывайте роли и обязанности.**

**Управляйте поэтапно.** Даже очень сложные проекты можно довести до конца, если

следовать схеме процессов.

**Управляйте по необходимости.** Нет нужды в постоянных встречах с командой или стоять у подопечных над душой — им достаточно раздать исчерпывающие задания. Вмешивайтесь тогда, когда нарушаются одно из шести ограничений.

**Сосредоточьтесь на продукте.** Каждый этап нужно завершить промежуточным результатом, чтобы убедиться, что вы идете в правильном направлении и не отходите от плана.

**Проявляйте гибкость.** Если составление документов о проделанной работе занимает неоправданно много времени, то можно просто отчитаться устно перед комитетом или прислать им электронное письмо. Сам Управляющий комитет тоже способен упростить себе жизнь, если тема касается только одного из его членов. Тогда вместо голосования можно просто довериться тому, кому ближе вопрос — заказчику, старшему исполнителю или старшему пользователю.

## 

## 7. Гибкие принципы управление. Agile – идеи и принципы.

**Agile. Идеи.**

**Люди и взаимодействие** важнее процессов и инструментов

**Работающий продукт** важнее исчерпывающей документации

**Сотрудничество с заказчиком** важнее согласования условий контракта

**Готовность к изменениям** важнее следования первоначальному плану

Не отрицая важности того, что справа, мы всё-таки больше ценим то, что слева.

**Agile. Принципы.**

1 Удовлетворение клиента за счёт ранней и бесперебойной поставки ценного программного обеспечения;

2 Приветствие изменений требований даже в конце разработки (это может повысить

конкурентоспособность полученного продукта);

3 Частая поставка рабочего программного обеспечения (каждый месяц или неделю или ещё чаще);

4 Тесное, ежедневное общение заказчика с разработчиками на протяжении всего проекта;

5 Проектом занимаются мотивированные личности, которые обеспечены нужными условиями работы, поддержкой и доверием;

6 Рекомендуемый метод передачи информации — личный разговор (лицом к лицу);

7 Работающее программное обеспечение — лучший измеритель прогресса;

8 Спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп на неопределённый срок;

9 Постоянное внимание улучшению технического мастерства и удобному дизайну;

10 Простота — искусство не делать лишней работы;

11 Лучшие технические требования, дизайн и архитектура получаются у самоорганизованной команды;

12 Постоянная адаптация к изменяющимся обстоятельствам. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

## 

## 8. Гибкие технологии. Scrum

Скрам-мастер (Scrum Master) — проводит совещания (Scrum meetings), следит за соблюдением всех принципов, разрешает противоречия и защищает команду от отвлекающих факторов, проводит фасилитацию митингов.

Владелец продукта (Product Owner) — представляет интересы заказчика или конечных пользователей и других заинтересованных в продукте сторон. В целом можно считать заказчиком

Скрам-команда (Scrum Team) — кросс-функциональная команда разработчиков проекта, состоящая из специалистов разных профилей: тестировщиков, архитекторов, аналитиков, программистов и т. д.

Размер команды составляет от 5 до 9 человек.

Инкремент Продукта — это готовый продукт в конце спринта. Показывают заинтересованным на демонстрации, чтобы собрать отзывы и решить, что делать с продуктом дальше.

Истории (User Story) — бизнес-функциональность, которую добавляют в бэклог, часто называют историей. Зачастую история имеет следующую структуру: «Будучи пользователем <тип пользователя> я хочу сделать <действие>, чтобы получить <результат>». Такая структура удобна тем, что понятна как разработчикам, так и заказчикам.

Story Points — абстрактная метрика оценки сложности истории, которая не учитывает затраты в человекочасах

Гибкий фреймворк, созданный в 1986 году, считается самым структурированным из семейства Agile. И самым популярным.

**Scrum. Основные встречи.**

Основная структура процессов Scrum вращается вокруг 5 основных встреч

**Встреча по упорядочиванию беклога (Backlog Refinement Meeting, «Backlog Grooming»**): на встрече пересматривают бэклог, меняют приоритеты, уточняют оценки и проясняют требования.

**Планирование Спринта (Sprint Planning Meeting**): из бэклога достаются задачи, которые планируются к выполнению за спринт, каждая задача оценивается в стори пойнтах. Обсуждается, как будут реализованы задачи. Проводится в самом начале спринта, после Встречи по упорядочиванию продукта.

**Ежедневные встречи (Daily Scrum**): каждый день спринта, в идеале, в одно и то же время, члены команды тратят 15 минут на то, чтобы поделиться информацией о статусе задач. На ней не происходит обсуждений проблем или принятия решений. Нужна для обмена информации и поддержания всех членов команды в курсе состояния задач.

**Подведение итогов Спринта (Sprint review meeting)**: команда представляет результаты деятельности всем заинтересованным лицам. Основная задача – убедиться, что продукт этапа соответствует ожиданиям участников. Нельзя демонстрировать незавершенную функциональность.

**Ретроспектива Спринта (Retrospective meeting**): команда выясняет, что было сделано хорошо и что можно улучшить. Выполняют улучшение процесса разработки.

## 9. Гибкие технологии. Kanban

**Kanban. Основные положения**

Для работы с Kanban необходимо определить этапы потока операций (workflow). В Kanban они изображаются как столбцы, а задачи обозначают специальные карточки. Карточка перемещается по этапам, и на каждом этапе процент завершения становится выше. На выходе мы получаем готовый к поставке заказчику элемент продукта. Доска со столбцами и карточками может быть как настоящей, так и электронной.

Карточки: Для каждой задачи создаётся индивидуальная карточка, в которую заносится вся необходимая информация о задаче. Таким образом, вся нужная информация о задаче всегда под рукой.

Ограничение на количество задач на этапе: Количество карточек на одном этапе строго регламентировано. Благодаря этому сразу становится видно, когда в потоке операций возникает «затор». Отсюда следует важный принцип вытягивания задач, а не их проталкивания.

Непрерывный поток: Задачи из беклога попадают в поток в порядке приоритета. Таким образом, работа никогда не прекращается.

Постоянное улучшение («кайзен» (kaizen)): постоянный анализ производственного процесса и поиск путей повышения производительности.

## 

## 10. Масштабирование гибкой разработки. Проблемы и решения. SAFe как пример масштабируемого фреймворка

Опыт команд разработки ПО показал, что внедрение agile-методов, таких как scrum и kanban, позволяет поставлять клиентам решения быстрее и более предсказуемо, а также быстро реагировать на новую информацию. Применять принципы agile на уровне отдельных команд сравнительно просто.

Но реальные сложности возникают, когда подход agile пытаются внедрить сразу в несколько команд в крупной организации. Или, другими словами, при масштабировании agile.

Масштабирование Agile — это культурное преобразование, при котором сотрудники, практики и инструменты компании совершенствуют совместную работу и обеспечивают реализацию корпоративной стратегии.

В конечном счете изменения в этих сферах приведут к децентрализации процесса принятия решения, повышению прозрачности и согласованности работы, а также увеличению скорости вывода продукта на рынок. При этом ценности agile становятся основополагающими принципами организации.

**SAFe** — это фреймворк для управления компанией в которой требуется координация работы над некоторым проектом или связанными проектами для 5 или более SCRUM командами. Т.е. это некая надстройка над SCRUM позволяющая управлять коллективами из 100 и более человек.

SAFe — это слоеный пирог из различных методик Agile. На нижнем уровне находится практически традиционный SCRUM, с двух-трех недельными спринтами, командами по 3-9 человек. С одним ключевым отличием: спринты объединяются в Program Increments состоящие из обычно 5 спринтов.

На следующем уровне идут Agile Release Trains (ART). Для управления пятиспринтовыми отрезками появляются новые функции — системный архитектор, product manager и RTE. Здесь применяются некоторые наработки из Kanban, в частности: доска, способ назначения приоритетов и проецирование того, что будет построено в конце временного отрезка. Последний спринт из 5 объявляется организационным и во время него проводятся огромные собрания, проводится анализ технического долга, строятся планы по проработке архитектуры и синхронизируется работа всех команд.

Над уровнем поездов идет координация между отделами, директорами, и клиентом. Здесь проводится анализ экономической целесообразности изменений. В идеале любые изменения проходят через предварительный анализ где выдвигается измеримая гипотеза о предстоящем изменении и далее эта гипотеза либо подтверждается, либо нет. Здесь же создаются планы работ на 12-36 месяцев.

## 

## 11. Гибкое управление. Особенности организации работы дистанционно работающих команд

Изначально agile-методы создавались для команд, участники которых располагались в одном офисе. В соответствии с идеей о том, что самый эффективный и практичный способ передачи информации команде разработчиков или между ее участниками — это личное общение. Но сегодня в большинстве компаний работают распределенные команды. Это не просто дань моде; это обусловлено реальной необходимостью. Распределенные команды могут работать над проектами круглые сутки, и ценного сотрудника можно найти на рынке, где конкуренция не так остра (не говоря уже о том, что сотрудника проще удержать, если не требовать от него менять место жительства). Но помимо плюсов у распределенных команд есть и минусы. Например, многим распределенным командам сложно ввести в обиход такую составляющую agile, как личное общение.

Распределенные команды разработчиков могут столкнуться и с другими сложными задачами:

* координация работы сотрудников из разных часовых поясов;
* налаживание хороших взаимоотношений между людьми, работающими в разных офисах;
* совместная работа специалистов, исповедующих разные подходы к разработке;
* планирование совещаний или неформальных бесед, когда обе команды одновременно доступны через Интернет в течение лишь нескольких часов (или даже меньше).

Все это создает сложные, но разрешимые проблемы.

1. Придайте команде правильную структуру. Задайте время для созвонов, воспользуйтесь преимуществами распределенной команды для увеличения времени поддержки пользователей. Команды, в которых работа распределена по нескольким часовым поясам, могут извлечь пользу из стендапов: на них команда, которая заканчивает работу, может передать эстафету команде, только выходящей на смену. Иногда офисы расположены так далеко друг от друга, что общие совещания будут причинять определенные неудобства одной из команд. Чередуйте время проведения совещаний. Так будет справедливо для обеих команд, и ни одной из них не придется постоянно работать сверхурочно (а подобные ситуации серьезно подрывают моральный дух).
2. Наладьте взаимоотношения в команде. Команда обязательно должна общаться для создания эмоциональной связи и командного духа, желательно общение по видеосвязи
3. Привейте единую культуру разработки.

* Сообщайте даже о самых незначительных решениях во все регионы, где присутствуют представители команды.
* Максимально упростите развертывание среды разработки.
* Установите четкие критерии готовности работы.
* Напишите инструкции по составлению правильных отчетов о багах.

## 12. Поток поставки ценности. DevOps – понятия и основные принципы

Одно из ключевых понятий DevOps, заимствованное из бережливого производства — поток создания ценности (англ. Value Stream). Это весь процесс поставки ценности вашему клиенту от начала и до конца — от момента, когда вы начинаете разработку и создание продукта, до конечного получения вашим клиентом ценности. Value stream mapping — это процесс написания такой карты.

«**DevOps**» – сочетание «development» и «operations» (разработка и операции). Игра слов отражает основную идею, заложенную в DevOps – сотрудничество между различными дисциплинами разработки.

DevOps — это технологическая структура, которая обеспечивает взаимодействие между командами разработчиков и операционными командами для более быстрого развертывания кода в производственных средах с возможностью повторения действий и автоматизации.

Ценность DevOps для компании:

1.Сокращение времени для выхода на рынок;

2.Снижение частоты отказов новых релизов;

3.Сокращение времени выполнения исправлений;

4.Уменьшение количества времени на восстановления.

5.Автоматизация процессов разработки;

6.Ускорение обратной связи от пользователей.

Принципы - ключевые идеи, на которых базируется весь DevOps, без принятия и применения которых от DevOps остаётся совсем мало смысла. Практики - действия, выполняемые в соответствии с принципами, направленные на получение полезного эффекта. Принципы будут неизменны для любой организации, применяющей идеи DevOps, в то время как практики, скорее всего, будут выбраны и видоизменены в зависимости от конкретной ситуации, компании и решаемых задач.

**Принципы DevOps (CALMS)**

· Culture

Культура DevOps – кросс-функциональное сотрудничество разнопрофильных специалистов и команд за счет единого информационного пространства проектного контента, открытых каналов коммуникаций и постоянного общения всех участников. От культуры в организации зависит то, как будут сформированы все процессы в целом.

· Automation

Автоматизация устраняет избыточный ручной труд, заменяя его на быстрые, надежные автоматизированные процессы.

· Lean IT

Бережливость в IT подразумевает выявление и устранение потерь (времени и других ресурсов) при разработке и эксплуатации ПО. Для этого весь процесс визуализируется с помощью карты **потока создания ценности.**

· Measurement

В DevOps измерять необходимо вообще всё. При этом действительно важные метрики необходимо контролировать постоянно, а остальные данные – при необходимости. (производительности, например, продолжительность работы пользователей с продуктом, частота появления в логах сообщений о критических ошибках – необходимы ясные и четкие критерии оценки работы, показатели эффективности процессов)

· Sharing

Распространение знаний подчеркивает важность открытых коммуникаций в компании, для того чтобы каждый мог понять на каком этапе сейчас находится каждая конкретная задача.

+Непрерывное совершенствование, ориентирование на клиента, создание продуктов с учетом конечной цели, совместная ответственность

## 

## 13. DevOps. Компоненты и процессы. Понятие безопасной разработки

«**DevOps**» – сочетание «development» и «operations» (разработка и операции). Игра слов отражает основную идею, заложенную в DevOps – сотрудничество между различными дисциплинами разработки.

DevOps — это технологическая структура, которая обеспечивает взаимодействие между командами разработчиков и операционными командами для более быстрого развертывания кода в производственных средах с возможностью повторения действий и автоматизации.

Ценность DevOps для компании:

1.Сокращение времени для выхода на рынок;

2.Снижение частоты отказов новых релизов;

3.Сокращение времени выполнения исправлений;

4.Уменьшение количества времени на восстановления.

5.Автоматизация процессов разработки;

6.Ускорение обратной связи от пользователей.

**Компоненты:**

1. Кодирование — разработка и анализ кода, инструменты контроля версий, слияние кода;
2. Сборка — инструменты непрерывной интеграции, статус сборки;
3. Тестирование — инструменты непрерывного тестирования, обеспечивающие быструю и своевременную оценку бизнес-рисков;
4. Упаковка — репозиторий артефактов, предварительная установка приложения;
5. Релиз — управление изменениями, официальное утверждение выпуска, автоматизация выпуска;
6. Настройка — конфигурация и управление инфраструктурой, Инфраструктура как инструменты кода;
7. Мониторинг — измерение производительности приложений, взаимодействие с конечным пользователем;
8. Непрерывная поставка;
9. Непрерывная интеграция.

**Процессы:** (DevOps покрывает все этапы жизненного цикла ПО.)

Непрерывная интеграция:

-Unit-тестирование

-Анализ Кода

-Сборка и развертывание на среде разработки

Непрерывная поставка:

-Оркестрация средами тестирования

-Развертывание на средах тестирования

-Тестирование

Непрерывное развертывание

-Развертывание в промышленных средах

-Обеспечение отката

**DevSecOps** – это тот же DevOps, но с учетом включения в себя практик безопасной разработки.

На этапе анализа кода – статистический анализ на безопасность кода; при тестировании – динамический анализ на безопасность

**Безопасная разработка** — методика разработки программного обеспечения, предотвращающая случайное внедрение уязвимостей и обеспечивающая устойчивость к воздействию вредоносных программ и несанкционированному доступу. Баги и логические ошибки являются основной причиной появления уязвимостей программного обеспечения.

## 

## 14. Документация. Типы, назначение. Принципы создания, группы заинтересованных лиц, методики формирования пакетов документации.

Характер архитектуры любой системы обусловливается предъявляемыми к ней требованиями; это утверждение справедливо и по отношению к документации системы — содержание документации зависит от предполагаемых вариантов ее применения.

Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа.

Существует четыре основных типа документации:

1. архитектурная/проектная — обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО

Архитектурная/проектная документация обычно описывает продукт в общих чертах. Не описывая того, как что-либо будет использоваться, она скорее отвечает на вопрос «почему именно так». Например, в проектном документе программист может описать обоснование того, почему структуры данных организованы именно таким образом. Описываются причины, почему какой-либо [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) сконструирован определенным образом, выделяются [паттерны](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в некоторых случаях даже даются идеи как можно будет выполнить улучшения в дальнейшем.

1. техническая — документация на [код](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), [алгоритмы](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), интерфейсы, [API](https://ru.wikipedia.org/wiki/API)

При создании программы, одного лишь [кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4), как правило, недостаточно. Должен быть предоставлен некоторый текст, описывающий различные аспекты того, что именно делает код. Такая документация часто включается непосредственно в исходный код или предоставляется вместе с ним.

Часто при составлении технической документации используются автоматизированные средства — [генераторы документации](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D0%B8), такие как [Doxygen](https://ru.wikipedia.org/wiki/Doxygen), [javadoc](https://ru.wikipedia.org/wiki/Javadoc), [NDoc](https://ru.wikipedia.org/w/index.php?title=NDoc&action=edit&redlink=1) и другие.

1. пользовательская — руководства для конечных пользователей, администраторов системы и другого персонала

В отличие от технической документации, сфокусированной на коде и том, как он работает, пользовательская документация описывает лишь то, как использовать программу.

Обычно, пользовательская документация представляет собой [руководство пользователя](https://ru.wikipedia.org/wiki/%D0%A0%D1%83%D0%BA%D0%BE%D0%B2%D0%BE%D0%B4%D1%81%D1%82%D0%B2%D0%BE_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8F), которое описывает каждую функцию программы, а также шаги, которые нужно выполнить для использования этой функции.

Очень важно, чтобы документация не вводила в заблуждение и была актуальной. Логическая связность и простота также имеют большое значение.

1. маркетинговая

Для многих приложений необходимо располагать рядом с ними рекламные материалы, для того, чтобы заинтересовать людей, обратив их внимание на продукт. Такая форма документации имеет целью:

* подогреть интерес к продукту у потенциальных пользователей
* информировать их о том, что именно делает продукт, с тем чтобы их ожидания совпадали с тем, что они получат
* объяснить положение продукта по сравнению с конкурирующими решениями

Заинтересованные в составлении документации лица преследуют разные цели — от представленной в ней информации они требуют разной направленности, разных уровней детализации и разных трактовок. Стремиться следует к тому, чтобы любое заинтересованное лицо смогло быстро найти необходимую информацию, при этом не тратя время на незначительные (с его точки зрения) сведения.

Для того чтобы понять, как структурировать документацию и повысить удобство ее использования, следует составить представление о заинтересованных лицах и разобраться в том, для чего она им может понадобиться.

Примеры заинтересованных лиц:

1. Архитектор и разработчики требований
2. Проектировщики, составляющие систему элементов
3. Разработчики и проектировщики
4. Тестировщики и сборщики
5. Специалисты по сопровождению
6. Проектировщики сторонних систем, взаимодействующих с рассматриваемой системой
7. Специалисты по атрибутам качества
8. Руководители
9. Группа контроля качества

Вероятно, наиболее важным понятием документирования программной архитектуры является **«представление»** (view). С понятием представления, которое можно определить как способ фиксации структуры, связан основной принцип документирования программной архитектуры:

**Документирование архитектуры подразумевает документирование всех значимых представлений с последующей фиксацией сведений, относящихся одновременно к нескольким представлениям.**

Этот принцип полезен тем, что он помогает разбить проблему документирования архитектуры на ряд менее обширных элементов:

1. выбор значимых представлений;
2. документирование представления;
3. документирование сведений, относящихся к нескольким представлениям.

## 

## 15. Тестирование ПО. Цели и принципы

**Основные цели тестирования**

• Оценку рабочих продуктов, таких как требования, пользовательские

истории, проектирование и код

• Проверку, все ли указанные требования выполнены

• Проверку, завершен ли объект тестирования и работает, как ожидают пользователи и заинтересованные лица

• Создание уверенности в уровне качества объекта тестирования

• Предотвращение дефектов

• Обнаружение отказов и дефектов

• Предоставление заинтересованным лицам достаточной информации, позволяющей им принять обоснованные решения, особенно в отношении уровня качества объекта тестирования

• Снижение уровня риска ненадлежащего качества программного обеспечения (например, пропущенные сбои в работе)

• Соблюдение договорных, правовых или нормативных требований, или стандартов и/или проверка соответствия объекта тестирования таким требованиям и стандартам

**Принципы тестирования**

**Тестирование демонстрирует наличие дефектов, а не их отсутствие**

Тестирование может показать, что дефекты присутствуют, но не может доказать, что их нет.

**Исчерпывающее тестирование недостижимо**

Полное тестирование с использованием всех комбинаций вводов и предусловий физически невыполнимо, за исключением тривиальных случаев.

**Раннее тестирование сохраняет время и деньги**

Тестирование на ранних этапах жизненного цикла разработки программного обеспечения помогает сократить или исключить дорогостоящие изменения.

**Кластеризация дефектов**

Обычно небольшое количество модулей содержит большинство дефектов, обнаруженных во время тестирования перед выпуском, или отвечает за большинство эксплуатационных отказов.

**Парадокс пестицида**

Если одни и те же тесты будут выполняться снова и снова, в конечном счете эти тесты больше не будут находить новых дефектов. Для обнаружения новых дефектов может потребоваться изменение существующих тестов и тестовых данных, а также написание новых тестов.

**Тестирование зависит от контекста**

Тестирование выполняется по-разному в зависимости от контекста. Например, программное обеспечение управления производством, в котором критически важна безопасность, тестируется иначе, чем мобильное приложение электронной коммерции.

**Заблуждение об отсутствии ошибок**

Некоторые организации ожидают, что тестировщики смогут выполнить все возможные тесты и найти все возможные дефекты, но принципы 2 и 1, соответственно, говорят нам, что это невозможно.

## 

## 16. Тестирование. Уровни и типы тестирования.

**Уровни тестирования:**

• Компонентное тестирование

• Интеграционное тестирование

• Системное тестирование

• Приемочное тестирование

**Объекты тестирования**

**Компонентное:**

**•** Компоненты, модули

• Код и структуры данных

• Базы данных

• Модули БД

• Классы

**Интеграционное:**

• Подсистемы

• Интерфейсы

• Микросервисы

• Программные интерфейсы приложений (API)

• Базы данных

• Инфраструктура

**Системное:**

• Приложения

• Аппаратные / программные системы

• Тестируемая система

• Операционные системы

• Конфигурация системы и конфигурация данных

**Приемочное:**

• Тестируемая система

• Конфигурация системы и конфигурационные данные

• Бизнес-процессы для полностью интегрированной системы

• Восстановление системы и непрерывность

• Формы

• Отчеты

• Существующие и преобразованные данные

**Типы тестирования**

**Функциональное тестирование**

Функциональное тестирование системы включает тесты по оценке функций, которые должна выполнять система. Функциональные требования могут быть описаны в рабочих продуктах (требования, спецификация, бизнес-потребность, пользовательская история, сценарий использования) или в функциональной спецификации, а могут быть вообще не задокументированы. Функции системы дают ответ на вопрос «что делает система»

**Нефункциональное тестирование**

Нефункциональное тестирование системы выполняется для оценки таких характеристик системы и программного обеспечения, как удобство использования, производительность или безопасность.

**Тестирование методом белого ящика**

Тестирование методом белого ящика основывается на внутренней структуре системы или ее реализации. Под внутренней структурой подразумевается программный код, архитектура, принципы работы и/или потоки данных внутри системы.

**Тестирование, связанное с изменениями**

Когда в систему вносятся изменения, выполненные для исправления дефекта, либо из-за новой или изменяющейся функциональности, необходимо провести тестирование, чтобы подтвердить, что изменения исправили дефект или что функциональность правильно реализована и изменения не вызвали каких-либо непредвиденных неблагоприятных последствий.

## 

## 17. Тестирование. Методы черного и белого ящика. Статическое и динамическое тестирование

Категории методов проектирования тестов

**Методы черного ящика** (поведенческие, или методы, основанные на поведении)

основываются на анализе соответствующего базиса тестирования (формальных

требований, спецификаций, сценариев использования, пользовательских историй или

бизнес-процессов).

**Методы черного ящика**

**Эквивалентное разбиение**

Делит данные на группы (классы эквивалентности), которые обрабатываются схожим образом. Области эквивалентности могут быть как для правильных (позитивных), так и неправильных (негативных) значений.

**Анализ граничных значений**

Является продолжением метода эквивалентного разбиения, но может быть применим, только если классы состоят из упорядоченных числовых значений. Максимальное и минимальное значение класса являются его границами.

**Тестирование с помощью таблицы альтернатив**

Комбинаторные методы полезны при тестировании требований, содержащих условия, которые дают разные результаты в зависимости от комбинаций. Запись сложных бизнес-правил, которые должны быть реализованы в системе. В процессе создания таблицы, тестировщик определяет условия (входы) и результирующие действия системы (выходы).

**Тестирование с помощью таблицы переходов**

В зависимости от прошлых условий и состояния система может вести себя по-разному. Описать прошлое системы можно с помощью концепции состояний. Диаграмма состояний и переходов показывает начальное и конечное состояния системы, а также описывает переходы между состояниями.

**Тестирование с помощью сценариев использования (use case)**

Тесты можно разработать на основе сценариев использования, которые представляют собой способ описания взаимодействий с программными объектами.

**Методы белого ящика** (структурные, или методы, основанные на структуре) основаны на анализе архитектуры, детального проектирования, внутренней структуры или кода компонента либо системы.

**Методы белого ящика**

**Тестирование и покрытие операторов**

Направлено на проверку исполняемых операторов в коде. Покрытие вычисляется как отношение количества операторов, выполненных тестом, к общему числу операторов в тестируемом коде.

**Тестирование и покрытие условий**

Тестирование условий направлено на проверку логических условий в коде, а также кода, выполняемого в зависимости от исхода условия. Для этого тесты следуют потокам управления, которые выходят из условия (путь для выхода «истина» и для выхода «ложь»; для оператора выбора (CASE) тесты потребуются для всех возможных результатов, включая выход по умолчанию). Покрытие вычисляется как отношение числа исходов условий, проверенных тестом, к общему числу исходов тестируемых условий.

**Методы, основанные на опыте**, используют опыт разработчиков, тестировщиков и пользователей для проектирования, реализации и выполнения тестов. Их часто совмещают с методами черного и белого ящиков.

**Статическое тестирование**

**Статическое тестирование (static testing) — тестирование без запуска кода на исполнение.**

Типы:

• выполняемая человеком экспертиза

• инструментальная оценка кода

Проведение статического анализа также часто включается в автоматизированный процесс сборки и поставки систем, например, в разработке по гибкой методологии с применением практик DevOps. Является важной частью тестирования безопасности.

Основное отличие: статическое тестирование обнаруживает дефекты в рабочих продуктах напрямую без фактического исполнения кода, а не идентифицирует сбои, вызванные дефектами при запуске программного обеспечения

Типичные дефекты, которые легче и дешевле найти и исправить с помощью статического тестирования:

• Дефекты требований (например, несоответствия, неоднозначности, противоречия, упущения, неточности, избыточность)

• Конструктивные дефекты (например, неэффективные алгоритмы или структуры базы данных, сильные связи, низкая связность)

• Отклонения от стандартов (например, несоблюдение стандартов кодирования)

• Неправильные спецификации интерфейса (например, различные единицы измерения)

• Уязвимость системы (например, уязвимость переполнения буфера)

• Пробелы или неточности в отслеживаемости и охвате тестовой базы (например, отсутствие тестов для критериев приемки)

**Динамическое тестирование** (dynamic testing) — тестирование с запуском кода на исполнение. Запускаться на исполнение может как код всего приложения целиком (системное тестирование), так и код нескольких взаимосвязанных частей (интеграционное тестирование), отдельных частей (модульное или компонентное тестирование) и даже отдельные участки кода.

## 18. Тестирование. Методы и инструменты. Классификация дефектов

**Тестирование -** это деятельность, в целом направленная на оценку и улучшение качества ПО. Однако цели тестирования могут варьироваться в зависимости от контекста тестируемого компонента или системы, уровня тестирования и модели жизненного цикла разработки программного обеспечения.

Методы:

**1 Методы белого ящика** (структурные, или методы, основанные на структуре) основаны на анализе архитектуры, детального проектирования, внутренней структуры или кода компонента либо системы.

**2 Методы черного ящика** (поведенческие, или методы, основанные на поведении)

основываются на анализе соответствующего базиса тестирования (формальных

требований, спецификаций, сценариев использования, пользовательских историй или

бизнес-процессов).

**Некоторые инструменты**

• Инструменты управления дефектами

• Инструменты статического анализа

• Инструменты проектирования тестов

• Инструменты разработки тестов через приемочное тестирование (ATDD – acceptance

test driven development) и на основе поведения (BDD - behavior driven development)

• Инструменты разработки через тестирование (TDD – test driven development)

• Инструменты выполнения тестов (например, запуск регрессионных тестов)

• Инструменты тестирования производительности

**Регистрация дефектов**

**Типичный набор атрибутов:**

• Идентификатор дефекта

• Заголовок и краткое описание найденного дефекта

• Дату сообщения о дефекте, информацию об авторе сообщения

• Идентификацию элемента тестирования (проверяемого элемента конфигурации) и среды

• Фазу жизненного цикла разработки, в которой обнаружен дефект

• Описание дефекта, достаточное для его воспроизведения и принятия решения, включая системные журналы,

скриншоты, дампы базы данных или записи (если они созданы во время выполнения теста)

• Ожидаемые и фактические результаты

• Область или степень влияния дефекта на интересы заинтересованной стороны (критичность)

• Срочность/приоритет для исправления

• Статус дефекта (например, открыт, отложен, дубликат, ожидает исправления, ожидает проверки, повторно открыт,

закрыт)

• Выводы, рекомендации и согласования

• Глобальные проблемы, например, области, которые будут затронуты исправлением дефекта

• История изменений, например, последовательность действий членов команды проекта, чтобы изолировать дефект,

исправить и подтвердить исправление дефекта

• Ссылки, включая ссылку на тестовый сценарий, который обнаружил дефект

**Дефект –** это несоответствие фактического результата выполнения программы ожидаемому результату. Дефекты обнаруживаются на этапе тестирования программного обеспечения (ПО), когда тестировщик проводит сравнение полученных результатов работы программы (компонента или дизайна) с ожидаемым результатом, описанным в спецификации требований.

**Дефект -** это не обязательно ошибка кода. Это может быть ошибка тестировщика, ошибка архитектора, анализа, новое требование, ошибка третьестороннего приложения. В качестве классификации предлагается следующее: функциональные дефекты, системные дефекты (ошибка внутреннего взаимодействия, проблема в аппаратной части, дефект в архитектуре ПО, ошибка в системном ПО), связанные с процессом дефекты (ошибки в инициализации, последовательности, статической логике), связанные с данными дефекты, связанные со стандартами дефекты, дефекты пользовательского интерфейса.

Классификация дефектов, с точки зрения степени влияния(Severity)на работоспособность ПО:

**Blocker**. Ошибка, которая приводит программу в нерабочее состояние. Дальнейшая работа с программной системой или ее функциями – невозможна.

**Critical**. Критический дефект, приводящий некоторый ключевой функционал в нерабочее состояние. Так же это может быть существенное отклонение от бизнес логики, неправильная реализация требуемых функций, потеря пользовательских данных и т.д.

**Major**. Весьма серьезная ошибка, свидетельствующая об отклонении от бизнес логики или нарушающая работу программы. Не имеет критического воздействия на приложение.

**Minor**. Незначительный дефект, не нарушающий функционал тестируемого приложения, но который является несоответствием ожидаемому результату. Например, ошибка дизайна.

**Trivial**. Баг, не имеющий влияние на функционал или работу программы, но который может быть обнаружен визуально. Например, ошибка в тексте.

## 

## 19. Сопровождение и поддержка ПО. ITSM и ITIL общие понятия и принципы

Сопровождение (поддержка) программного обеспечения — процесс улучшения, оптимизации и устранения дефектов программного обеспечения (ПО) после передачи в эксплуатацию.

**ITSM (IT Service Management, управление ИТ-услугами)** — подход к управлению и организации ИТ-услуг, направленный на удовлетворение потребностей бизнеса. Управление ИТ-услугами реализуется поставщиками ИТ-услуг путём использования оптимального сочетания людей, процессов и информационных технологий

**ITIL (IT Infrastructure Library — библиотека инфраструктуры информационных**

**технологий)** — библиотека, описывающая лучшие из применяемых на практике

способы организации работы подразделений или компаний, занимающихся

предоставлением услуг в области информационных технологий.

**Основополагающие принципы ITIL .**

7 универсальных принципов, направляющих организацию в любых

условиях и при любых обстоятельствах:

· Фокусируйтесь на ценности

· Отталкивайтесь от текущей ситуации

· Продвигайтесь постепенно, шаг за шагом, используя обратную связь

· Сотрудничайте, действуйте открыто и понятно

· Используйте системный подход

· Не усложняйте, будьте прагматичны

· Оптимизируйте и автоматизируйте

Основополагающие принципы ITIL дают всесторонний и системный

взгляд на управление и работу сервисной организации. Фокус на

сотрудничестве и взаимодействии, автоматизации и упрощении отражает

принципы, которые мы можем найти в Agile, DevOps и Lean.

Базовые принципы методологии ITSM раскладывают процесс управления услугами на пять этапов, которые описывают жизненный цикл любой услуги:

1. формирование стратегии услуг,
2. проектирование услуг,
3. преобразование услуг,
4. эксплуатация услуг,
5. постоянное улучшение.

## 

## 20. Сопровождение и поддержка. ITIL. Базовые процессы

Сопровождение (поддержка) программного обеспечения — процесс улучшения, оптимизации и устранения дефектов программного обеспечения (ПО) после передачи в эксплуатацию.

**ITIL (IT Infrastructure Library — библиотека инфраструктуры информационных технологий)** — библиотека, описывающая лучшие из применяемых на практике способы организации работы подразделений или компаний, занимающихся предоставлением услуг в области информационных технологий.

**Базовые процессы ITIL**

Интересующие нас:

• Процесс управления инцидентами

• Процесс управления проблемами

• Процесс управления конфигурациями

• Процесс управления изменениями

• Процесс управления релизами

**Управление инцидентами (Incident Management)** - процесс, отвечающий за управление жизненным циклом всех инцидентов.

Основная цель Управления инцидентами - скорейшее восстановление услуги для пользователей.

**Инцидент (Incident)** - незапланированное прерывание услуги или снижение качества услуги. Сбой конфигурационной единицы, который еще не повлиял на услугу, также является инцидентом.

**Управление проблемами (Problem Management)** - процесс, отвечающий за управление жизненным циклом всех проблем. Ключевыми целями Управления проблемами являются предотвращение инцидентов и минимизация влияния тех инцидентов, которые не могут быть предотвращены.

**Проблема (Problem) -** причина одного или нескольких инцидентов

Процесс Управления проблемами предназначен для диагностирования первопричин возникновения инцидентов и поиска решений по их устранению. Он также контролирует то, что решение проблем будет осуществлено в рамках соответствующих процессов, в частности, Управления изменениями и Управления релизами.

Процесс Управления проблемами тесно связан с процессом Управления инцидентами, так как возникновение инцидентов происходит вследствие наличия проблем. Эти процессы часто используют одни инструменты, системы категорирования и расстановки приоритетов и т.п.

**Управление конфигурациями** отвечает за то, чтобы отдельные компоненты услуги, системы или продукта, были должным образом определены, снабжены всем необходимым и контролировались. Процесс также контролирует все изменения компонентов. Он предоставляет модель конфигураций со всеми связями между активами и конфигурациями. Объектом рассмотрения является Конфигурационная единица.

Конфигурационная единица (Configuration Item или CI) - любой компонент, который нуждается в управлении для того, чтобы предоставлять услугу. Информация о каждой КЕ регистрируется в форме Записи о КЕ в Системе управления конфигурациями и поддерживается актуальной в течение всего жизненного цикла процессом Управления конфигурациями. КЕ находятся под контролем Управления изменениями. Типичными примерами КЕ являются услуги, оборудование, программное обеспечение, здания, люди и документы, такие как Процессная документация и Соглашения об уровне услуг (SLA)

## 

## 21. Управление изменениями. ITIL и процесс управления изменениями

**ITIL (IT Infrastructure Library — библиотека инфраструктуры информационных технологий)** — библиотека, описывающая лучшие из применяемых на практике способы организации работы подразделений или компаний, занимающихся предоставлением услуг в области информационных технологий.

**Управление изменениями.**

**Изменение** – добавление, модификация или удаление чего-либо, способного оказать влияние на ИТ-услуги. В эти рамки необходимо включать все изменения в архитектурах, процессах, инструментах, метриках и документации, а также изменения в ИТ-услугах и других конфигурационных единицах.

Управление изменениями – процесс, отвечающий за управление жизненным циклом всех изменений, способствующий реализации полезных изменений с минимальным прерыванием ИТ-услуг.

Запрос на изменение – формальное предложение на выполнение изменения. Запрос на изменение включает в себя детали предложенного изменения и может быть записан в бумажном или электронном виде.

**Три типа изменений:**

Стандартное изменение – предавторизованное изменение, с низким риском,

относительно обычное и следующее какой-либо процедуре или рабочей инструкции.

Экстренное изменение – изменение, которое должно быть внедрено как можно

быстрее, например, для разрешения значительного инцидента или установки

обновления безопасности.

Нормальное изменение – изменение, не являющееся срочным или стандартным.

**Пример системы кодирования приоритетов:**

• Низкий приоритет — изменение желательно, но его внедрение может быть

отложено.

• Обычный приоритет — нет особой срочности и высокой степени воздействия, но

изменение не следует откладывать.

• Высокий приоритет — изменение касается серьезной ошибки, затрагивающей ряд

пользователей.

• Наивысший приоритет — запрос на изменение (RFC) касается проблемы, серьезно

влияющей на важнейший для заказчиков сервис. Изменения с таким приоритетом

классифицируются как «экстренные».

## 22. Процесс разработки. Архитектура решения. Назначение, принципы выбора. Анализ и оценка

**Процесс разработки -** Процесс, посредством которого потребности пользователей преобразуются в программный продукт. Процесс разработки программного обеспечения является составной частью программной инженерии.

Архитектура

Архитектура это объективная реальность. Любое существующее приложение имеет архитектуру. Даже если о ней никто не задумывался. Даже если это выглядит как абсолютный хаос. Даже если все делалось «как получится».

Архитектура

● Позволяет организовывать коммуникацию с заинтересованными лицами

● определяет ограничения реализации

● сдерживает или способствует реализации атрибутов качества системы

● облегчает анализ изменений и их организацию

● позволяет более точно рассчитывать стоимость и сроки

Архитектура может быть реализована с различной степенью детализации – от набора стикеров с общими схемами до детально проработанной схемы

Создание проходит 3 этапа:

1. Архитектурный образец (Архитектурный стиль) – описание типов элементов и отношений и изложение ряда ограничений на их использование.
2. Эталонная модель – разделение функциональных возможностей и потоков данных по блокам. Стандартная декомпозиция известной проблемы на части, которые взаимодействуя могут решить проблему.
3. Эталонная архитектура – это эталонная модель отображенная на программные элементы (реализующие функциональность, определенную в эталонной модели). Эталонная модель обеспечивает разделение функций, эталонная архитектура отображает эти функции на декомпозицию системы

Далее не уверена, что нужно именно это -

Планирование:

Этап планирования включает в себя сбор требований где вся необходимая информация собирается у клиента и документируется. Спецификация требований к программному обеспечению создается на следующем этапе.

Анализ рисков:

На этом этапе выбирается лучшее решение с учетом имеющихся рисков и проводится анализ путем создания прототипа.

Пример: риск, связанный с доступом к данным из удаленной базы данных, может заключаться в том, что скорость доступа к данным может быть слишком низкой. Риск может быть устранен путем создания прототипа подсистемы доступа к данным.

Разработка:

Как только завершен анализ рисков, следует программирование и тестирование.

Оценка:

Клиент оценивает разработанную систему и планируется следующая итерация.

## 

## 23. Процесс разработки. Архитектура. Документирование и распространение сведений. Группы заинтересованных лиц.

**Процесс разработки -** Процесс, посредством которого потребности пользователей преобразуются в программный продукт. Процесс разработки программного обеспечения является составной частью программной инженерии.

**Архитектура**

Архитектура это объективная реальность. Любое существующее приложение имеет архитектуру. Даже если о ней никто не задумывался. Даже если это выглядит как абсолютный хаос. Даже если все делалось «как получится».

Архитектура

● Позволяет организовывать коммуникацию с заинтересованными лицами

● определяет ограничения реализации

● сдерживает или способствует реализации атрибутов качества системы

● облегчает анализ изменений и их организацию

● позволяет более точно рассчитывать стоимость и сроки

Архитектура может быть реализована с различной степенью детализации – от набора стикеров с общими схемами до детально проработанной схемы

**Создание проходит 3 этапа:**

1. Архитектурный образец (Архитектурный стиль) – описание типов элементов и отношений и изложение ряда ограничений на их использование.
2. Эталонная модель – разделение функциональных возможностей и потоков данных по блокам. Стандартная декомпозиция известной проблемы на части, которые взаимодействуя могут решить проблему.
3. Эталонная архитектура – это эталонная модель отображенная на программные элементы (реализующие функциональность, определенную в эталонной модели). Эталонная модель обеспечивает разделение функций, эталонная архитектура отображает эти функции на декомпозицию системы

**Документирование**

Для того, чтобы архитектура стала основой проекта, ее суть должна быть четко донесена до всех заинтересованных лиц, документирование архитектуры должно быть информативным и понятным людям различных профессий. Заинтересованные лица преследуют различные цели – требуют разной направленности документов, разных уровней детализации и разных трактовок. Необходимо стремиться к тому, чтобы **любое заинтересованное лицо смогло быстро найти необходимую именно ему информацию**.

**Группы заинтересованных лиц**

● Спонсоры и менеджеры проекта

● Представляющие заказчика разработчики требований

● Архитектор(-ы)

● Тестировщики и сборщики

● Специалисты по сопровождению

## 

## 24. Атрибуты качества систем и их связь с функциональными требованиями. Сценарии атрибутов качества

Архитектурно-экономический цикл, как правило, основан на требованиях к качеству, определяемыми прямыми коммерческими мотивами, то есть, функциональностью разрабатываемой программы или программного комплекса. Это предполагает включения атрибутов качества как элементов требований к функциональности – при описании возможностей и поведения системы и оказываемых ей услуг. Функциональность становится ведущим показателем качества и занимает в схеме разработки центральное, или даже единоличное положение

**Классификатор атрибутов качества:**

1. Атрибуты качества системы. Будем рассматривать такие атрибуты как готовность, модифицируемость производительность, безопасность, контролепригодность и практичность
2. Коммерческие атрибуты качества (например, Time-to-Market, срок вывода продукта на рынок). Имеет смысл рассматривать только те из атрибутов, реализация которых определяется архитектурой системы
3. Атрибуты качества самой архитектуры (например, концептуальная целостность), которые влияют на другие качества – например, модифицируемость

Сценарием атрибута качества называется требование, путем выполнения которого этот атрибут реализуется.

Сценарий состоит из шести элементов:

1. Источник стимула. Это некий субъект (человек, вычислительная система) который порождает символ
2. Стимул. Стимулом называется наблюдаемое в системе явление, требующее к себе внимания
3. Условия. Стимул возникает при конкретных условиях. К примеру, система может находиться в состоянии перегрузки или исполняться в обычном режиме
4. Артефакт. Объектом воздействия стимула является некий артефакт. В этом качестве выступает система в целом или ее отдельный элемент
5. Реакция. Реакция – это действие, предпринятое в ответ на появление стимула
6. Количественная мера реакции. Предпринимаемые в ответ на стимул действия должны быть измеримы – только тогда соответствие требованию можно проверить

## 

## 25. Атрибуты качества и сценарии - подробное описание на примере одного из типа сценариев (Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)

\*\*\*Из примеров сценариев нужно выбрать только один на ваш вкус\*\*\*

Вопросы воплощения атрибутов качества решаются в периоды проектирования, реализации и развертывания. Не существует ни одного атрибута качества, который бы зависел только от какого-то одного этапа программного цикла.

* **Практичность** имеет как архитектурные, так и не архитектурные аспекты. Примеры последних – обеспечение ясности и простоты применения пользовательского интерфейса. Что выбрать: переключатель или флажок? Какая схема размещения на экране наиболее интуитивна? Эти вещи имеют большое значение для конечного пользователя но при этом являются локальными решениями стадии реализации
* **Модифицируемость** определяется тем, каким образом происходит разделение функциональности (архитектурный аспект) и методик кодирования в рамках отдельного модуля (аспект тактической реализации).
* **Производительность** также зависит от архитектурных и неархитектурных факторов. Она зависит от объемов информации, передающейся между компонентами, от распределения функциональности между компонентами, управления общими ресурсами – все это архитектурные аспекты. Но на производительность влияют и выбранные для реализации функциональности алгоритмы, и качество их реализации (кодирования) – а это неархитектурные аспекты.

Атрибуты качества:

* Атрибуты качества системы. Такие атрибуты как готовность, модифицируемость, производительность, безопасность, контролепригодность и практичность
* Коммерческие атрибуты качества (например, Time-to-Market, срок вывода продукта на рынок). Имеет смысл рассматривать только те из атрибутов, реализация которых определяется архитектурой системы
* Атрибуты качества самой архитектуры (например, концептуальная целостность), которые влияют на другие качества – например, модифицируемость

Реализация ряда атрибутов качества по отдельности в сколько-либо сложной системе невозможна. Реализация одного атрибута всегда каким-то образом (положительно или отрицательно) влияет на реализацию других. К примеру, безопасность и надежность, как правило, взаимно конфликтуют

У всех проведенных атрибутов качества есть три основных недостатка:

* Непрактичные определения
* Несогласованность классификаций
* У разных сообществ свои словари.

Первые две проблемы (непрактичные определения и несогласованность классификаций) решаются путем применения для составления характеристик сценариев атрибутов качества (quality attribute scenarios)

**Сценарием атрибута качества** называется требование, путем выполнения которого этот атрибут реализуется. Сценарий состоит из шести элементов

* Источник стимула. Это некий субъект (человек, вычислительная система) который порождает символ
* Стимул. Стимулом называется наблюдаемое в системе явление, требующее к себе внимания
* Условия. Стимул возникает при конкретных условиях. К примеру, система может находиться в состоянии перегрузки или исполняться в обычном режиме
* Артефакт. Объектом воздействия стимула является некий артефакт. В этом качестве выступает система в целом или ее отдельный элемент
* Реакция. Реакция – это действие, предпринятое в ответ на появление стимула
* Количественная мера реакции. Предпринимаемые в ответ на стимул действия должны быть измеримы – только тогда соответствие требованию можно проверить

**Пример сценария готовности:**

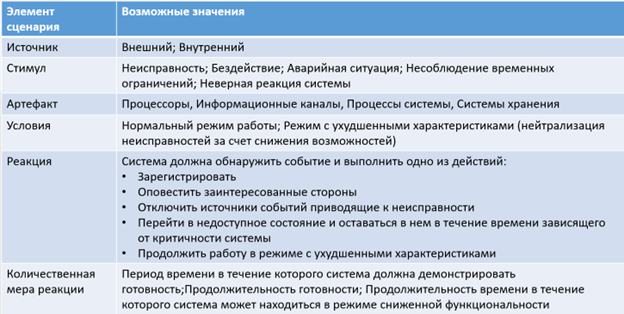
****

Готовность отражает ситуацию отказа системы и его возможных последствий. Отказом системы называется положение, при котором система теряет способность предоставления услуг, заявленных в ее спецификации. Отказ заметен для пользователей системы, в качестве которых могут выступать люди или другие системы.

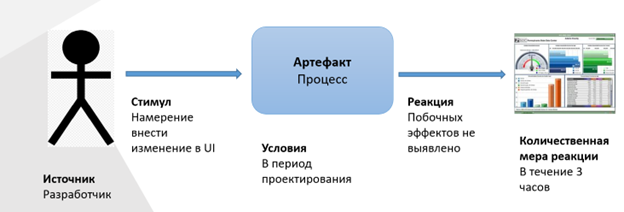
Следует четко различать Отказ (failure) и неисправность (fault). Если неисправность не устранена или не замаскирована, она может перейти в состояние отказа. Пользователь системы может наблюдать ситуацию отказа, но не неисправность. Когда неисправность становится обозримой для пользователя, она становится отказом.

После отказа системы на первый план выходит вопрос о том, сколько времени потребуется на его устранение. Из того, что отказ системы обозрим для пользователей, можно сделать вывод, что период устранения отказа эквивалентен периоду, по истечении которого отказ перестает быть обозримым.

В количественном выражении готовность это вероятность функционирования системы на отрезках времени когда в этом есть необходимость.



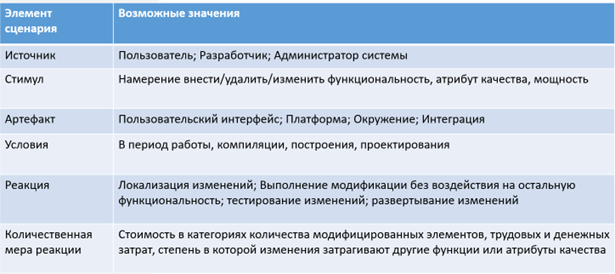
**Пример сценария модифицируемости:**



Качество модифицируемости выражает стоимость внесения изменений. Оно решает два вопроса:

1. Под воздействием чего артефакт может быть изменен? Изменения можно внести в любой аспект системы — как правило, это ее функции, платформа (аппаратная часть, операционная система, промежуточное программное обеспечение и т. д.)

2. Когда производится модификация и кто ее проводит (условия)? В прошлом изменения чаще всего вносились в исходный код. При этом разработчик проводил модификацию, после чего она тестировалась и размещалась в новой версии. Сегодня проблема времени модификации все больше переплетается с проблемой ее исполнителя. Изменения можно вносить в реализацию (путем редактирования исходного кода), в периоды компиляции (при помощи переключателей периода компиляции), построения (путем выбора тех или иных библиотек), настройки конфигурации (разными методами, в частности — путем установки параметров



## 

## 26. Тактики обеспечения атрибутов качества (на примере одного из типов Модифицируемость, Ремонтопригодность, Производительность, Безопасность...)

Тактики реализации атрибутов качества используются воплощения атрибутов качества системы.

Тактика — это проектное решение, которое влияет на управление реакцией по атрибуту качества. Совокупность тактик называется архитектурной стратегией (architectural strategy).

Цели стратегий безопасности — это строгое выполнение обязательств, конфиденциальность, целостность и гарантирование.

Достичь этих целей помогают следующие тактики:

• Аутентификация пользователей. Это проверка, является ли пользователь или удаленный компьютер тем, за кого себя выдает

• Авторизация. Проверка, обладает ли аутентифицированный пользователь или процесс достаточными правами на запрашиваемый ресурс.

• Обеспечение конфиденциальности данных. Например, шифрация каналов и данных

• Обеспечение целостности. Данные должны передаваться в неизменном виде. Содержащаяся в них служебная информация – например, контрольные суммы или хэши – может шифроваться вместе с данными или отдельно.

• Минимизация подверженности внешним воздействиям. Распределение данных и служб по независимым хостам

• Ограничения доступа. Проверка доступа на firewall с ограничением портов отправки и назначения.

**Тактики восстановления** после атак могут быть ориентированы на восстановление предшествующего состояния или на идентификацию исполнителя атаки (последние можно причислить как к превентивным, так и к карательным мерам).



Тактики реализации производительности

1. Тактика снижения потребления ресурсов

a. Повышение производительности за счет оптимизации алгоритмов, отказа от использования медленных ресурсов, выявления узких мест.

b. Сокращение непроизводственных издержек. Отказ от посредников, лишних выводов и операций копирования данных.

c. Уменьшения частоты срабатывания.

d. Ограничение времени обработки входящих запросов.

e. Ограничение длины очереди входящих запросов.

2. Тактика управления ресурсами

a. Параллельная обработка поступающих запросов.

b. Дублирование данных или вычислений.

c. Увеличение вычислительной мощности системы.

3. Тактика доступа к ресурсам

a. FIFO.

b. На основе приоритетов.

i.Статические приоритеты в зависимости от времени запуска задачи

ii.Статические приоритеты в зависимости от предельного срока обработки

iii.Статические приоритеты в зависимости от частоты поступления запросов.

iv.Динамические приоритеты с циклическим обслуживанием запросов.

v.Динамические приоритеты в зависимости от предельного срока обработки

## 

## 27. Документирование программных систем. Заинтересованные лица, представления и нотации (с примерами).

Содержание документации зависит от предполагаемых вариантов ее применения.

Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа.

Заинтересованные в составлении документации лица преследуют разные цели.

Стремиться следует к тому, чтобы любое заинтересованное лицо смогло быстро найти необходимую информацию, при этом не тратя время на незначительные (с его точки зрения) сведения.

Для того чтобы понять, как структурировать документацию и повысить удобство ее использования, следует составить представление о заинтересованных лицах и разобраться в том, для чего она им может понадобиться.

Наиболее важным понятием документирования программной архитектуры является «представление» (view) - способ фиксации структуры, с ним связан основной принцип документирования программной архитектуры:

Документирование архитектуры подразумевает документирование всех значимых представлений с последующей фиксацией сведений, относящихся одновременно к нескольким представлениям

Этот принцип полезен тем, что он помогает разбить проблему документирования архитектуры на ряд менее обширных элементов:

♦ выбор значимых представлений;

♦ документирование представления;

♦ документирование сведений, относящихся к нескольким представлениям.

Общие подходы к документированию программных систем

Типы нотаций

* **формальные нотации**

•Семантика полностью стандартизована, смысл диаграмм определяется однозначно

• Возможен формальный анализ как синтаксиса, так и семантики модели

• Примеры: AADL, ER

Особенности:

• Избыточно сложны

• Требуются глубокие знания языка описания

• ВСЕГДА требуется дополнительное ПО

• Больше подходят для технического описания аппаратных комплексов

* **Частично формальные нотации**

Нотация стандартизована, есть наборы графических примитивов и правила их использования

• Полная семантика и возможные трактовки элементов диаграмм не зафиксированы

• Анализ семантики упрощен по сравнению с неформальными нотациями

• Примеры: UML, BPMN, ERD

Особенности:

Требуется изучение нотации всеми участниками процесса

• Как правило, требуется дополнительное ПО

• Фокусировка на определенных областях, атрибутах

* **Неформальные нотации**

• Диаграммы в свободной форме

• Семантика моделей на естественном языке (цвета, форма, надписи)

• Информация диаграмм не может быть формально проанализирована

• Легко понятные, но при этом высок риск неправильного понимания

• Пример: классические boxes-and-lines например в PowerPoint

Особенности:

• Для понимания требуется «ключ»

• Содержимое раскрывается с помощью фигур, цветов и стрелок

• Легко создавать

• Самые популярные

• Хороши для презентаций

Выбор нотации всегда должен учитывать:

Какие заинтересованные лица (stakeholders) будут вовлечены в работу с документацией

Стандарты организации/проектной группы

Более формальные нотации требуют больших затрат, но дают новые возможности (анализ и т.д.)

Менее формальные нотации легче создавать и применять, но они дают меньше гарантий

Нотации: типичные представители

• UML

• BPMN

• C4

• FlowChart/swimlane

## 

## 28. Документирование программных систем. Неформальные нотации. Плюсы и минусы, области применения

Содержание документации зависит от предполагаемых вариантов ее применения.

Документация ни при каких обстоятельствах не может быть универсальной. С одной стороны, она должна быть абстрактной и, следовательно, доступной для понимания новыми сотрудниками, но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа.

Наиболее важным понятием документирования программной архитектуры является «представление» (view) - способ фиксации структуры, с ним связан основной принцип документирования программной архитектуры:

Документирование архитектуры подразумевает документирование всех значимых представлений с последующей фиксацией сведений, относящихся одновременно к нескольким представлениям

Этот принцип полезен тем, что он помогает разбить проблему документирования архитектуры на ряд менее обширных элементов:

♦ выбор значимых представлений;

♦ документирование представления;

♦ документирование сведений, относящихся к нескольким представлениям.

* **Неформальные нотации**

• Диаграммы в свободной форме

• Семантика моделей на естественном языке (цвета, форма, надписи)

• Информация диаграмм не может быть формально проанализирована

• Легко понятные, но при этом высок риск неправильного понимания

• Пример: классические boxes-and-lines например в PowerPoint

Особенности:

• Для понимания требуется «ключ»

• Содержимое раскрывается с помощью фигур, цветов и стрелок

• Легко создавать

• Самые популярные

• Хороши для презентаций

Выбор нотации всегда должен учитывать:

Какие заинтересованные лица (stakeholders) будут вовлечены в работу с документацией

Стандарты организации/проектной группы

Более формальные нотации требуют больших затрат, но дают новые возможности (анализ и т.д.)

Менее формальные нотации легче создавать и применять, но они дают меньше гарантий

## 

## 29. Документирование программных систем. Частично формальные и формальные нотации. Плюсы и минусы, области применения

Документация

Документирование ПО помогает команде понять структуру системы и назначение ее элементов.

Также часть документации предназначена для конечных пользователей и администраторов — она позволяет понять, как нужно устанавливать и разворачивать систему, какие функции она предоставляет и каким образом можно бороться с ошибками, возникающими при ее работе.

Содержание документации зависит от предполагаемых вариантов ее применения.

С одной стороны документация должны быть абстрактной (доступной для понимания новыми сотрудниками), но с другой — весьма детальной — настолько, чтобы ее можно быть использовать как план проведения анализа.

Частично формальные нотации

• Такая нотация стандартизована, в ней есть наборы графических примитивов и правила их использования

• Но возможные трактовки элементов диаграмм четко не зафиксированы

• Анализ семантики упрощен по сравнению с неформальными нотациями

• Требуется изучение нотации всеми участниками процесса

• Частично формальные нотации фокусируются на определенных областях, атрибутах

• Примеры: UML (пример ПО – StarUML), BPMN (пример ПО - ARIS), ERD (Lucidchart)

Плюсы: Анализ значения элементов упрощен по сравнению с неформальными нотациями

Минусы: Полная семантика и возможные трактовки элементов диаграмм не зафиксированы

Область применения : Моделирование процессов и систем при составлении бизнес требований и системных требований, которые будут доступны для понимания всех участников команд

Формальные нотации

• Семантика элементов нотации полностью стандартизована

• Возможен формальный анализ синтаксиса и семантики модели

• Существует специализированное ПО для анализа полноты описания и генерации программного кода на целевых платформах

• Примеры: AADL (ПО - Open Source AADL Tool Environment), ER (пример ПО - Oracle SQL Developer Data Modeler)

Плюсы: Значение элементов нотации идентифицируется однозначно

Минусы: Нотации избыточно сложны, требуются глубокие знания языка описания

Область применения: подходят для технического описания аппаратных комплексов

## 

## 30. Нотации. Рассмотреть пример одной из нотаций на выбор. Основные типы диаграмм, области применения

***BPMN***

Это метод описания бизнес-процессов, отображающий этапы выполнения процесса от начала до конца.BPMN-схемы наглядно и подробно демонстрируют последовательность рабочих действий и перемещение информационных потоков, необходимых для выполнения процесса, а потому являются одним из ключевых инструментов управления бизнесом.

**BPMN элементы “Пул” и “Дорожка”**

Весь бизнес-процесс состоит из пулов: совокупности операций + лиц, которые эти операции выполняют (пример - набор действий по погрузке товара и отправке его клиенту).

При этом выделяют так называемые “дорожки”, из которых состоит любой пул (например, первая дорожка - оформление документов, касающихся погрузки и отправки товара; вторая – погрузка нужной партии товаров в автомобиль)

**BPMN элемент “Событие”**

“Событие” является одним из главных элементов BPMN и служит для описания того, что должно случиться.

В зависимости от положения на схеме процесса бывают:

- Начальное событие (инициирующее бизнес-процесс)

- Промежуточное событие

- Конечное событие (заканчивающее бизнес-процесс).

**BPMN элемент “Действие”**

Под “действием” понимается единица работы, выполняемой в ходе исполнения бизнес-процесса. Действия могут быть как элементарными (задача), так и составными (подпроцесс).

Основные типы действий:

- Абстрактная задача – простое действие или операция

- Подпроцесс - отображение свернутого процесса, включенного в состав рассматриваемого процесс

- Процесс-ссылка - Используется для обозначения ссылки на один из наиболее часто повторяющихся процессов.

**BPMN элементы “Развилка” /“Шлюз”**

Два основных типа:

- Шлюз исключающего «или» - нужен для создания альтернативных потоков процесса (на развилке выбираем только один путь). Чтобы выбрать ветку применяется условие

- Параллельный шлюз – нужен для для создания параллельных путей без оценки какого бы то ни было условия

**BPMN элементы “Потоки”**

Поток – это последовательность действий, которая обозначается стрелкой. Элемент “поток” показывает какое действие после какого необходимо совершить (стрелочка между действиями)

**BPMN элементы “Артефакт”**

Под артефактами в BPMN понимают объекты, которые не влияют на исполнение бизнес-процесса напрямую. Это могут быть документы, данные.

***UML***

UML был разработан с целью обеспечить единый визуальный язык с богатой семантикой и развернутым синтаксисом для проектирования и внедрения программных систем со сложной структурой и комплексным поведением. UML применяется не только в разработке программ, но и в других сферах, например, в схематизации потоков производственных процессов.

Диаграммы UML описывают границы, структуру и поведение как всей системы, так и отдельных объектов в ее составе. UML не является языком программирования, однако на базе диаграмм UML можно сгенерировать код на разных языках, с помощью специализированного ПО.

**Структурные диаграммы UML**

- Диаграммы классов — отображают классы внутри системы, а также атрибуты, операции(методы) и отношения между классами. Диаграммы классов применяются при схематизации крупных систем, так как позволяют объединять классы в группы.

- Диаграммы компонентов - отражают структурные отношения между элементами программной системы и чаще всего применяются при работе с комплексными структурами, состоящими из множества компонентов. Взаимодействие между компонентами осуществляется посредством интерфейсов.

- Диаграммы композитной структуры - применяются для схематизации внутренней структуры класса.

- Диаграммы развертывания - иллюстрируют аппаратное и программное обеспечение системы. Удобны в тех случаях, когда программный продукт применяется на нескольких компьютерах с индивидуальными настройками.

- Диаграмма объектов - представляет собой конкретный экземпляр диаграммы классов в определенный момент времени. Диаграмма объектов фокусируется на атрибутах набора объектов и на том, как эти объекты связаны друг с другом.

**Поведенческие диаграммы UML**

- Диаграммы активности - схематично отображают потоки рабочих или операционных процессов, иллюстрируя активность в разных частях или компонентах системы.

- Диаграммы последовательностей (сиквенсы)- показывают, как и в какой последовательности объекты взаимодействуют между собой. Взаимодействие иллюстрирует определенный сценарий.

- Схемы коммуникации - напоминают диаграммы последовательностей, однако во главу угла ставят сообщения, передаваемые между объектами.

## 31. Стили и шаблоны проектирования. Понятие архитектурного стиля и архитектурного шаблона

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



**Архитектурный шаблон**

Выражает фундаментальную структурную схему организации программных систем. Он предоставляет набор предопределенных подсистем, определяет их обязанности и включает правила и рекомендации по организации взаимосвязей между ними.



## 

## 32. Архитектурные принципы проектирование. Основные принципы и подробно на примере 1 – 2 выбранных

· ПРИНЦИП ЕДИНСТВЕННОЙ ОТВЕТСТВЕННОСТИ

Модуль, компонент, объект выполняет минимально возможную функциональность, обеспечивающую решение дискретной задачи

· ПРИНЦИП НАИМЕНЬШЕГО ЗНАНИЯ

Модуль, компонент, объект не знает о внутренней реализации других модулей, компонентов, объектов с которыми взаимодействует

· ПРИНЦИП СЛАБОЙ СВЯЗНОСТИ

Связи модулей, совместно составляющих систему минимальны

· ПРИНЦИП СИЛЬНОГО ЗАЦЕПЛЕНИЯ

Внутренние компоненты модуля работают совместно для решения общих задач и сильно «сцеплены» друг с другом

· ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ

Зависимости компонент направлены в сторону абстракции, а не в сторону деталей реализации

**Разбор принципа слабой связности**

Связность — это мера зависимости программного блока от других программных блоков.

Программный модуль может иметь разную степень детализации: методы в классе, классы, подсистемы, пакеты, модули, сервисы. И чем больше элементы программного модуля будут зависеть от элементов других программных модуля, тем более высокой связностью будет обладать этот модуль.

Некоторые примеры типов связности:

-Классы могут взаимно получать доступ к (частным) данным друг друга. Это очень *сильная форма* связи, поскольку вы больше не можете изменять один из классов, не рассматривая другой.

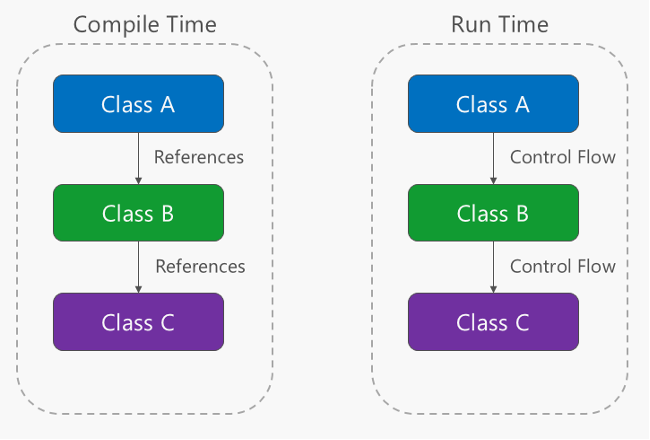
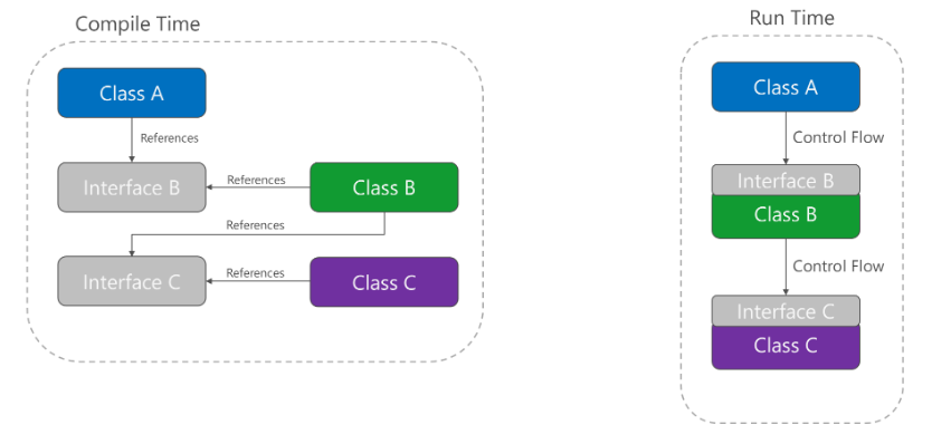
- *Более слабая связь* присутствует, если классы взаимодействуют через глобальную структуру данных. Прямые зависимости между классами освобождаются и передаются на аутсорсинг глобальной структуре данных. Несмотря на это, связь по-прежнему очень сильна: все изменения, которые влияют на глобальные данные, также влияют на все классы, которые работают с данными.

-Если классы взаимодействуют только через параметры метода, *связь значительно ниже*: задействованные методы содержат только необходимые данные. Таким образом, изменения в этой связке данных вызывают только локальные изменения в соответствующих методах задействованных классов. Отсутствие связи (система связанных объектов, которые обмениваются данными посредством сообщений)

**Разбор принципа инверсии зависимостей**

Инверсия зависимостей является важной частью процесса создания слабо связанных приложений, так как детали реализации могут описывать зависимости и реализовывать абстракции более высокого уровня, а не компоненты того же уровня. В результате получаются приложения с более высоким уровнем тестируемости, модульности и удобства в обслуживании. Практика *внедрения зависимостей* базируется на соблюдении принципа инверсии зависимостей.

“Прямой” граф зависимостей “Инвертированный” граф зависимостей

## 

## 33. Архитектурные стили. Уровни применения, типичные представителе (на примере выбранного - монолит, SOA, Микросервисы...)

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

Уровни применения:

1) Уровень приложения

- монолит

- микросервисы

- Event-driven

- SOA

2) Уровень компонент

- модульная архитектура

- слоистая архитектура

-архитектура каналов сообщений

**Микросервисы**

Микросервисная архитектура (или просто «микросервисы») представляет собой метод организации архитектуры, основанный на ряде независимо развертываемых служб. У этих служб есть собственная бизнес-логика и база данных с конкретной целью. Обновление, тестирование, развертывание и масштабирование выполняются внутри каждой службы. Микросервисы разбивают крупные задачи, характерные для конкретного бизнеса, на несколько независимых баз кода.

Ключевые атрибуты:

- "Облегченный" аналог SOA

- Компактны (легко понять, легко выложить)

- Loose Coupling, High Cohesion

- Независимы (можно выкладывать по одному)

- Изоляция ошибок

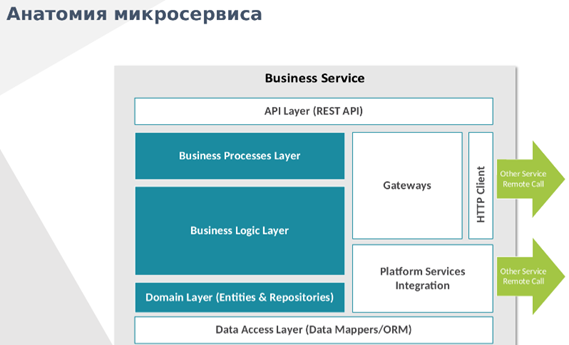
- Совместимы с SOAP, REST, Event-driven подходами

Проблемы:

- N монолитов -> N x М сервисов

- Трудно правильно делить систему

- API Gateway, Service Registry, Load Balancer, Fault Tolerance/Monitoring, Configuration/Properties Management



## 

## 34. Архитектурные стили на уровне компонентов. Слоистая и модульная архитектура

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

Уровни применения:

1) Уровень приложения

- монолит

- микросервисы

- Event-driven

- SOA

2) Уровень компонент

- модульная архитектура

- слоистая архитектура

-архитектура каналов сообщений

Архитектура программной системы, как правило, сочетает в себе несколько архитектурных стилей.

**Слоистая архитектура:**

• Может быть логической и физической

• Структура системы организована в набор слоев

• Каждый слой укладывается поверх другого слоя

• Четко определенные интерфейсы между слоями

• Снижает сложность, улучшает модульность, возможность повторного использования, ремонтопригодность

• Различные критерии для расслоения: в первую очередь абстракция

**Модульная архитектура:**

* Модульность -> более обслуживаемая и легкая в поддержке система
* Приложения должны иметь возможности расширения функционала без простоя
* Модуль как независимая единица поставки
* У модуля есть имя, версия и список зависимостей
* Взаимодействие с другими модулями через четкий интерфейс, детали реализации скрыты

Проблемы:

* Объекты, пакеты и архивы не являются независимыми

Решение:

* Контейнеризация

## 

## 35. Паттерны проектирования. Общие понятия, назначение, плюсы и минусы.

Общие понятия

Паттерн проектирования — это часто встречаемое решение определённой проблемы при проектировании архитектуры программ.

Паттерны необходимы для:

● проверенных решений

● стандартизации кода

● общего программистского стиля

Виды паттернов и их назначение:

● поведенческие - Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы

● структурные - Эти паттерны отвечают за построение удобных в поддержке иерархий классов

● порождающие - Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов

Преимущества от применения паттернов:

● **повышает эффективность труда** отдельных исполнителей и всей группы в целом.

● позволяет также создавать **более модифицируемое и гибкое** программное обеспечение

● помогают **общему пониманию** основных принципов объектно-ориентированного проектирования

● позволяют разработчику находить проектные решения для сложных проблем, **не создавая громоздкой иерархии** наследования классов.

Недостатки паттернов:

● **неэффективные решения -** разработчики используют паттерны как догмы и не подстраивают под проект

● **неоправданное применение** – использование там, где можно обойтись кодом попроще

● **“костыль” для слабых языков программирования** - применяют паттерны для тех проектов, которые написаны на языке программирования с низким уровнем абстракции

## 36. Паттерны проектирования. Порождающие паттерны (на выбранном примере)

**Паттерн проектирования** – повторяемая архитектурная конструкция, часто встречающееся решение какой-либо проблемы проектирования в рамках некоторого возникающего контекста. Он представляет собой общую концепцию решения той или иной проблемы, которую необходимо подстроить под собственную программу. Он не является четким набором действий, а представляет только высокоуровневое описание решения.

Паттерны проектирования помогают сэкономить время и сделать более качественное решение, а также их знание упрощает понимание между программистами.

Зачем?

* проверенные решения
* стандартизация кода
* общий программистский стиль

**Порождающие.** Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.

* Абстрактная фабрика - создает семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов
* Фабричный метод - определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов
* Строитель - Создает сложные объекты пошагово
* Одиночка - Создает гарантированно единственный экземпляр класса
* Прототип - Создает копию объекта, не вдаваясь в подробности его реализации

В качестве примера рассмотрим паттерн одиночка (Singleton). Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

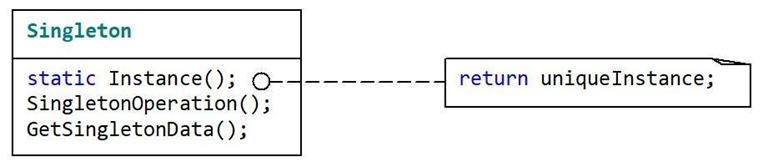
Проблема: Вы делаете программу для игры в шахматы. Программа предусматривает возможность множества последовательных игр. При создании фигур используются картинки стилей, скачиваемые с удаленного сервера по ненадежной линии (долго). Есть желание «кешировать»

Решение: Паттерн одиночка для создания кеша фигур (которые потом клонируются для каждой новой партии)

В жизни хороший пример одиночки – это правительство государства. В государстве может быть только одно официальное правительство. Вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».

В программном продукте паттерн одиночка используется тогда, когда есть сущности, существующие только в единственном экземпляре, например, система ведения системного журнала сообщений, или для доступа к какому-то общему ресурсу, например базе данных.

Структурная схема паттерна:



На рисунке отображается класс Singleton, который содержит три метода:

● Instance() — это главный метод, в котором осуществляется создание экземпляра класса и контроль за количеством созданных экземпляров класса. Если количество экземпляров класса не превышает некоторое максимальное (например 1), то метод возвращает экземпляр класса Singleton;

● SingletonOperation() — метод класса, который выполняет некоторую работу. Здесь может быть несколько методов с любыми именами (например SingletonOperation1(), SingletonOperation2() и т.д.), реализующие функционал класса;

● GetSingletonData() — метод класса, возвращающий данные. Здесь может быть реализовано разнообразное количество методов в зависимости от требований, устанавливаемых для класса.

Создание экземпляра класса в клиентском коде осуществляется вызовом функции Instance(). Создать экземпляр класса другим способом не удастся.

## 37. Паттерны проектирования. Структурные паттерны (на выбранном примере)

**Паттерн проектирования** – повторяемая архитектурная конструкция, часто встречающееся решение какой-либо проблемы проектирования в рамках некоторого возникающего контекста. Он представляет собой общую концепцию решения той или иной проблемы, которую необходимо подстроить под собственную программу. Он не является четким набором действий, а представляет только высокоуровневое описание решения.

Паттерны проектирования помогают сэкономить время и сделать более качественное решение, а также их знание упрощает понимание между программистами.

Зачем?

* проверенные решения
* стандартизация кода
* общий программистский стиль

**Структурные.**

Эти паттерны отвечают за построение удобных в поддержке иерархий классов. Определяют различные сложные структуры, которые изменяют интерфейс существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

* Адаптер (Adapter) позволяет объектам с несовместимыми интерфейсами работать вместе
* Мост (Bridge) деление классов на независимые абстракцию и реализацию
* Компоновщик (Composite pattern) позволяет сгруппировать множество объектов в древовидную структуру
* Декоратор (Decorator) динамически добавляет объектам новую функциональность, оборачивая их в полезные «обёртки»
* Фасад (Facade) предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку
* Легковес (Flyweight) экономит память, разделяя общее состояние объектов между собой
* Заместитель (Proxy) позволяет подставлять вместо реальных объектов специальные объекты-заменители

В качестве примера рассмотрим адаптер. **Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

**Проблема.** Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики. В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда — библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением. Вы могли бы переписать библиотеку, чтобы та поддерживала формат XML. Но, во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у вас может просто не быть доступа к её исходному коду.

**Решение.** Вы можете создать адаптер. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту. При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы. Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

Иногда возможно создать даже двухстороннийадаптер, который работал бы в обе стороны.

Таким образом, в приложении биржевых котировок вы могли бы создать класс XML\_To\_JSON\_Adapter, который бы оборачивал объект того или иного класса библиотеки аналитики. Ваш код посылал бы адаптеру запросы в формате XML, а адаптер сначала транслировал входящие данные в формат JSON, а затем передавал бы их методам обёрнутого объекта аналитики.

**Аналогия из жизни.** Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются. Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

## 

## 38. Паттерны проектирования. Поведенческие паттерны (на выбранном примере)

**Паттерн проектирования** – повторяемая архитектурная конструкция, часто встречающееся решение какой-либо проблемы проектирования в рамках некоторого возникающего контекста. Он представляет собой общую концепцию решения той или иной проблемы, которую необходимо подстроить под собственную программу. Он не является четким набором действий, а представляет только высокоуровневое описание решения.

Паттерны проектирования помогают сэкономить время и сделать более качественное решение, а также их знание упрощает понимание между программистами.

Зачем?

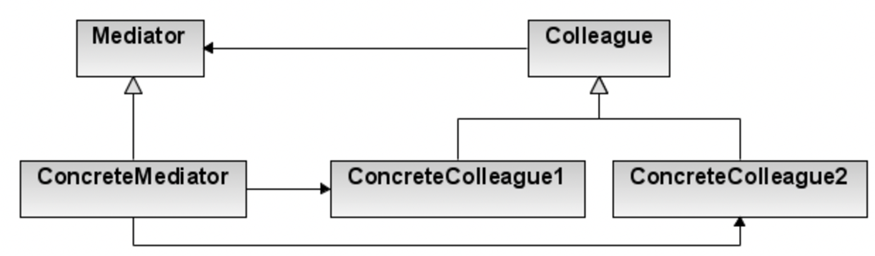
* проверенные решения
* стандартизация кода
* общий программистский стиль

**Поведенческие**

Шаблоны проектирования,определяющие алгоритмы и способы реализации эффективного и безопасного взаимодействия различных объектов и классов. Поведенческие шаблоны проектирования определяют общие закономерности связей между объектами, реализующими данные паттерны.

* **Цепочка обязанностей** — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.
* **Команда** — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.
* **Итератор** — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.
* **Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.
* **Хранитель или снимок** — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.
* **Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.
* **Состояние** — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.
* **Стратегия** — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.
* **Шаблонный метод** — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.
* **Посетитель —** это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Рассмотрим на примере посредника:



Проблема: предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок. Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы

Решение: заставить объекты общаться не напрямую друг с другом, а через отдельный объект посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.

Аналогия из жизни: Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно, а это приводило бы к частым катастрофам в небе.

## 

## 39. Управление качеством. Основные понятия, процессы и практики

Управление качеством - это процесс, который помогает организациям достичь и удерживать высокие стандарты качества продукции и услуг. Оно основано на принципах и практиках, которые помогают организациям понять и отвечать на потребности и ожидания клиентов.

Основные понятия управления качеством включают в себя понятия такие как качество, процессы и практики управления качеством.

Качество - это общее понятие, которое определяет степень соответствия продукта или услуги ожиданиям и потребностям клиентов.

Процессы управления качеством определяют порядок действий, которые должны быть предприняты для достижения и удержания высокого уровня качества. Исследования качества помогают организациям понять, как их продукты и услуги могут быть улучшены

Одной из наиболее эффективных практик является применение стандартов качества. Это позволяет определить минимальные требования к продукции и услугам, а также помогает обеспечить их высокое качество. Кроме того, стандарты качества помогают определить процессы и процедуры, необходимые для достижения желаемого уровня качества.

Основные практики:

1. Определение требований к качеству: процесс определения требований к качеству продукции и услуг для достижения наилучших результатов.

2. Планирование качества: процесс планирования и разработки процессов и процедур для достижения желаемого уровня качества.

3. Управление проектами: процесс управления проектами для достижения желаемого

## 

## 40. Управление качеством. Стандарты и модели

Разработка ПО успешна тогда, когда в ее результате за заданное время и бюджет материализуется затребованная заказчиком функциональность. При этом, немаловажной составляющей функциональности являются показатели качества системы.

Основными государственными стандартами, регламентирующими использование терминологии по качеству программного обеспечения в Российской Федерации, являются:

•**ГОСТ 28806-90** «КАЧЕСТВО ПРОГРАММНЫХ СРЕДСТВ. Термины и определения (Software quality. Terms and definitions)»

•**ГОСТ 28195-89** «ОЦЕНКА КАЧЕСТВА ПРОГРАММНЫХ СРЕДСТВ. Общие положения (Quality control of software systems. General principles)»

•**ГОСТ Р ИСО/МЭК 25000**. «Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов»

Согласно ГОСТ 28806-90 качество программного средства – это совокупность свойств, которые обуславливают его пригодность удовлетворять заданные или подразумеваемые потребности в соответствии с его назначением.

Программное средство (ПС) определяется как объект, состоящий из программ, процедур, правил, а также, если предусмотрено в сопутствующих им документации, и данных, относящихся к функционированию системы обработки информации.

Модель качества (quality model) – это определенное множество характеристик и взаимосвязей между ними, которые обеспечивают основу для определения требований к качеству и оценки качества (ИСО/МЭК 25000).

Характеристики качества ПС часто называют свойствами, атрибутами, показателями или факторами качества. Характеристика качества ПС является структурным многоуровневым объектом, и определяется как набор свойств, которые могут быть выражены числовыми и (реже) лингвистическими оценками.

Согласно ГОСТ 28195-89 [2] характеристика качества декомпозируется на 4 уровня:

•**Фактор качества** определяет группу основных показателей качества, характеризующую потребительские свойства (требования)

•**Критерий качества** задает программно-ориентированные свойства (требования) ПС. Для вычисления значения критерия используют одну или несколько метрик.

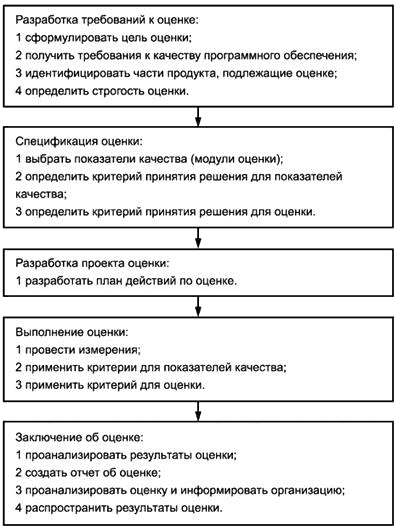
•**Метрика** используется для количественной оценки качества ПО по заданному критерию. Она определяется как шкала и метод измерения свойства.

•**Оценочный элемент** (элемент показателя качества) измеряет заданное в метрике свойство числом от 0 до 1 (ГОСТ 28195-89)

## 

## 41. Управление качеством. Эталонная модель

Эталонная модель процесса оценки качества программного продукта описывает процесс и детализирует действия и задачи, обеспечивающие их цели, а также предоставляет дополнительную методическую информацию, которую допускается использовать для проведения оценки качества программного продукта

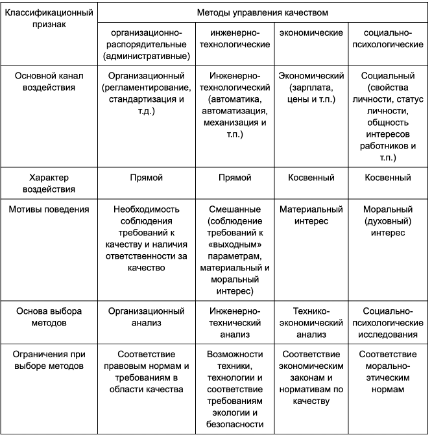


## 

## 42. Методологии управления качеством. Классификация

Методы управления качеством представляют собой способы и приемы осуществления управленческой деятельности и воздействия на управляемые объекты для достижения поставленных целей в об­ласти качества.

В практике управления качеством используются:  
- организационно-распорядительные (административные),  
- инженерно-технологические,  
- экономические,  
- социально-психологичес­кие методы.  
Все они могут быть классифицированы по различным признакам



Все указанные методы могут найти применение как при управле­нии качеством на всех стадиях жизненного цикла управляемых объ­ектов (продукции, услуг), так и при улучшении любых видов дея­тельности.

Наряду с этими способами в практике принятия решений в области управления качеством широко используются экспертные методы.

Организационно-распорядительные методы управления качест­вом осуществляются посредством обязательных для исполнения ди­ректив, приказов и других предписаний, направленных на повыше­ние и обеспечение необходимого уровня качества.

В группу организационно-распорядительных методов управления качеством следует включать методы:

-регламентирования (общеорганизационного, функционального, должностного, структурного);

-стандартизации (на основе стандартов различного уровня и статуса);

-нормирования (на базе норм времени, численности и др.);

-инструктирования (ознакомления, объяснения, разъяснения);

-распорядительных воздействий (на основе приказов, распоря­жений, указаний, постановлений, контроля исполнения с использованием превентивного и оперативного воздействий и т.п.). Включают:

-разработку и реализацию политики в области качества, утверждаемую первыми руководи­телями организаций;

-разработку и реализацию стандартов предприятия (СТП), руко­водств, процедур и другой НТД и НМД по управлению качеством;

-разработку и внедрение положений о подразделениях и должностных инструкций;

подбор, расстановку, воспитание и аттестацию кадров по управлению качеством; внедрение передо­вого опыта по управлению качеством;

обеспечение выполнения целевых программ по качеству;

приказы и распоряжения по управлению качеством;

обеспечение выполнения требований МС, ГОСТ и ТУ;

контроль за исполнением требований НТД, НМД и решений по управлению и обеспечению качества.

## 

## 43. Рефакторинг. Назначение и принципы

**Рефакторинг** представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода вы улучшаете его дизайн уже после того, как он написан.

**«Улучшение кода после его написания»** — непривычная фигура речи. В нашем сегодняшнем понимании разработки программного обеспечения мы сначала создаем дизайн системы, а потом пишем код. Сначала создается хороший дизайн, а затем происходит кодирование. Со временем код модифицируется, и целостность системы, соответствие ее структуры изначально созданному дизайну постепенно ухудшаются. Код медленно сползает от проектирования к хакерству.

**Рефакторинг представляет собой противоположную практику.** С ее помощью можно взять плохой проект, даже хаотический, и переделать его в хорошо спроектированный код. Каждый шаг этого процесса прост до чрезвычайности. Перемещается поле из одного класса в другой, изымается часть кода из метода и помещается в отдельный метод, какой-то код перемещается в иерархии в том или другом направлении. Однако суммарный эффект таких небольших изменений может радикально улучшить проект. Это прямо противоположно обычному явлению постепенного распада программы.

**При проведении рефакторинга оказывается, что соотношение разных этапов работ изменяется.** Проектирование непрерывно осуществляется во время разработки, а не выполняется целиком заранее. При реализации системы становится ясно, как можно улучшить ее проект. Происходящее взаимодействие приводит к созданию программы, качество проекта которой остается высоким по мере продолжения разработки.

Принципы рефакторинга

**Рефакторинг (Refactoring):** изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.

**Производить рефакторинг (Refactor):** изменять структуру программного обеспечения, применяя ряд рефакторингов, не затрагивая его поведения.

Рефакторинг не меняет видимого поведения программного обеспечения. Оно продолжает выполнять прежние функции. Никто — ни конечный пользователь, ни программист — не сможет сказать по внешнему виду, что что-то изменилось.

**Назначение рефакторинга**

§ Рефакторинг улучшает композицию программного обеспечения

§ Рефакторинг облегчает понимание программного обеспечения

§ Рефакторинг помогает найти ошибки

§ Рефакторинг позволяет быстрее писать программы

## 

## 44. Рефакторинг. Техники рефакторинга (на примере одной из техник)

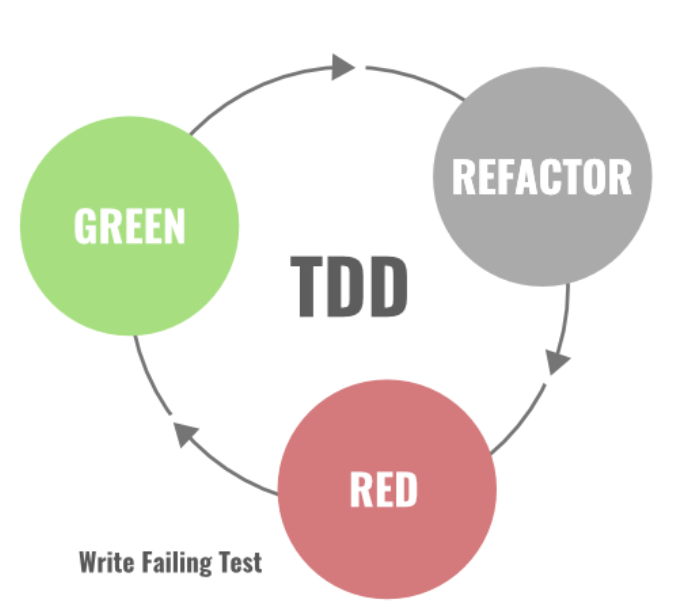
**Рефакторинг** представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода вы улучшаете его дизайн уже после того, как он написан.  
**Техники:**   
**1. Рефакторинг с помощью абстракции**

Этот метод в основном используется разработчиками, когда необходимо выполнить большой объем рефакторинга. В основном мы используем эту технику для уменьшения избыточности (дублирования) в нашем коде. Это включает в себя наследование классов, иерархию, создание новых классов и интерфейсов, извлечение, замену наследования делегированием и наоборот.  
**2. Метод составления**

На этапе разработки приложения мы часто пишем длинные методы в нашей программе. Эти длинные методы делают ваш код чрезвычайно трудным для понимания и изменения. В этих случаях в основном используется метод компоновки.В этом подходе мы используем методы оптимизации, чтобы уменьшить дублирование в нашем коде.   
**3. Методы упрощения**В этом подходе задействованы два метода.  
1)**Упрощение условных выражений Рефакторинг:** условный оператор в программированиисо временем становится более логичным и сложным. Вам нужно упростить логику в вашем коде, чтобы понять всю программу.  
**2)Упрощение вызовов методов Рефакторинг:** при таком подходе мы делаем вызовы методов более простыми и понятными. Мы работаем над взаимодействием между классами и упрощаем интерфейсы для них.

**4. Перемещение объектов между объектами**

В этой технике мы создаем новые классы и безопасно перемещаем функциональность между старыми и новыми классами. Мы скрываем детали реализации от публичного доступа.  
**5. Подготовительный рефакторинг**Этот подход лучше всего использовать, когда вы замечаете необходимость рефакторинга при добавлении некоторых новых функций в приложение. По сути, это часть обновления программного обеспечения с отдельным процессом рефакторинга. Вы избавите себя от будущих технических долгов, если заметите, что код нуждается в обновлении на ранних этапах разработки функций.  
**6. Рефакторинг пользовательского интерфейса**Вы можете вносить простые изменения в пользовательский интерфейс и реорганизовывать код. Например: выровнять поле ввода, применить шрифт, изменить слово активным голосом, указать формат, применить общий размер кнопки и увеличить цветовой контраст и т. Д.  
**7. Красно-зеленый рефакторинг (+ пример)**  
Красно-зеленый - самый популярный и широко используемый метод рефакторинга кода в процессе гибкой разработки программного обеспечения. Этот метод следует подходу “сначала тестирование” к проектированию и реализации, это закладывает основу для всех форм рефакторинга. Разработчики берут на себя инициативу по рефакторингу в цикл разработки, основанный на тестировании, и он выполняется в три этапа.

  
 **Красный:** первый шаг начинается с написания неудачного “красного теста”. Вы останавливаетесь и проверяете, что нужно разработать.

**Зеленый:** На втором этапе вы пишете достаточно простой код и получаете разрешение на разработку “зеленого” тестирования.  
**Рефакторинг:** на последнем и третьем этапах вы сосредотачиваетесь на улучшении и улучшении своего кода, сохраняя тест "зеленым".

Итак, в основном этот метод состоит из двух отдельных частей: первая часть включает в себя написание кода, который добавляет новую функцию в вашу систему, а вторая часть посвящена рефакторингу кода, который выполняет эту функцию. Имейте в виду, что вы не должны делать оба одновременно во время рабочего процесса.

## 45. Метрики программного кода. Основные принципы и подходы, инструменты, практика применения.

**Метрика** — это количественная мера позволяющая оценить, в какой степени система, компонент системы или процесс обладают заданным качеством.

**Основные цели применения метрик программного кода:**

· Измерение качества продукта либо процесса

· Прогнозирование качества продукта либо процесса

· Повышение качества продукта либо процесса.

**Метрики позволяют решать задачи:**

• Оценка производительности применения новых средств и методов

• Определение тенденций производительности с течением времени

• Оценка стоимости и графика будущих проектов

• Прогноз будущих потребностей в персонале

• Контроль стабильности качества ПО

• Прогноз стоимости владения (поддержки и развития) ПО

**Инструменты:**

• Количественные метрики(размерно-ориентированные)

- LOC-оценка и ее производные

LOC-оценка (Lines Of Code)-это самая простая и старая методика, основанная на подсчете строк кода(физических – SLOC или логических DSI). Эти метрики чувствительны к используемому языку программирования, что накладывает ограничения на их применение.

- Метрики Хольстеда (Haulsted)

Данные метрики основаны на следующих показателях:

n1 — число уникальных операторов программы (словарь операторов)

n2 — число уникальных операндов программы (словарь операндов),

N1 — общее число операторов в программе,

N2 — общее число операндов в программе,

n1' — теоретическое число уникальных операторов,

n2' — теоретическое число уникальных операндов.

• Метрики сложности потока управления программы

- Цикломатическая сложность

Самой распространенной оценкой, основанной на анализе графа управления, является

цикломатическая сложность программы (цикломатическое число Мак-Кейба).

Цикломатическая сложность определяется как

V(G) = e — n + 2p

где e — количество дуг

n — количество вершин

p — число компонент связности.

Дополнительные методики на основе графа управления

Для устранения недостатков оценки цикломатической сложности разрабатывались похожие методики:

метод Хансена. Мера сложности программы в данном случае представляется в виде пары (цикломатическая сложность, число операторов). Преимуществом данной меры является ее чувствительность к структурированности ПО.

Метрика Пивоварского**.** Модификация меры цикломатической сложности. Позволяет отслеживать различия не только между последовательными и вложенными управляющими конструкциями, но и между структурированными и неструктурированными программами.

Мера Вудворда — количество пересечений дуг управляющего графа. Так как в хорошо структурированной программе таких ситуаций возникать не должно, то данная метрика применяется в основном в слабо структурированных языках (ассемблер)

• Метрики сложности потока управления данными

- Метрика Чепина

Метрика Чепина: суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

• Метрики объектно-ориентированных программных систем

Набор метрик Чидамбера и Кемерера

**Метрика 1: Взвешенные методы на класс WMC (Weighted Methods Per Class)** учитывает комплексный показатель на основе количества методов в классе и их сложности

**Метрика 2: Высота дерева наследования DIT (Depth of Inheritance Tree)** максимальная длинна цепочки наследования классов. Рекомендация не более 7

**Метрика 3: Количество детей NOC (Number of children)** количество непосредственных потомков одного класса. Рекомендация не более 7 (9)

**Метрика 4: Сцепление между классами объектов СВО (Coupling between object classes):** Количество классов, о которых «знает» данный класс

**Метрика 5: Отклик для класса RFC (Response For a Class):** количество методов класса плюс количество методов других классов, вызываемых из данного класса. Метрика RFC является мерой потенциального взаимодействия данного класса с другими классами, позволяет судить о динамике поведения

**Метрика 6: Недостаток связности в методах LСOM (Lack of Cohesion in Methods)** показывает, насколько методы не связаны друг с другом через свойства (переменные). Можно определить как количество пар методов, не связанных по свойствам класса, минус количество пар методов, имеющих такую связь.

Метрики Лоренца и Кидда

**Метрики, ориентированные на классы:**

**Метрика 1: Размер класса CS (Class Size) –** число операций (методов) класса. CS <=20

**Метрика 2: Количество операций, переопределяемых подклассом, NOO (Number of Operations Overridden by a Subclass) –** число унаследованных операций, переопределяемых классом. NOO <=3

**Метрика 3: Количество операций, добавленных подклассом, NOA (Number of Operations Added by a Subclass) –** число новых методов, вводимых подклассом. NOA <= 4 для листового класса

**Метрика 4: Индекс специализации SI (Specialization Index) –** насколько методы класса НЕ используют совместно атрибуты

**Операционно-ориентированные метрики:**

**Метрика 5: Средний размер операции OSAVG (Average Operation Size)** – количество сообщений, посылаемых операцией. <= 25

**Метрика 6: Сложность операции ОС (Operation Complexity) --** число унаследованных операций, переопределяемых классом. NOO <=3

**Метрика 7: Среднее количество параметров на операцию NPAVG (Average Number of Parameters per operation)**

**Проектные метрики:**

**Метрика 8: Количество описаний сценариев NSS (Number of Scenario Scripts)**

**Метрика 9: Количество ключевых классов NKC (Number of Key Classes)**

**Метрика 10: Количество подсистем NSUB (NumberofSUBsystem)**

**ИСПОЛЬЗОВАНИЕ МЕТРИК ПРИМЕРЫ:**

Базовая плотность ошибок



Выработка в зависимости от размера проекта

