Оглавление

1. Верификация. Основные понятия ([1] стр. 3-7)

Под *жизненным циклом программного обеспечения* обычно понимают весь интервал времени от момента зарождения идеи о том, чтобы создать или приобрести программную систему для решения определенных задач, до момента полного прекращения использования последней ее версии.

Вид деятельности в жизненном цикле ПО — это набор действий, направленных на решение одной задачи или группы тесно связанных задач в рамках разработки и сопровождения ПО.
Примерами видов деятельности являются анализ предметной области, выделение и описание требований, проектирование, разработка кода, тестирование, управление конфигурациями, развертывание.

Роль в жизненном цикле ПО — это профессиональная специализация людей, участвующих в работах по созданию или сопровождению ПО (или затрагиваемых ими) и имеющих одинаковые интересы или решающих одни и те же задачи по отношению к этому ПО. Примеры ролей: бизнес-аналитик, инженер по требованиям, архитектор, проектировщик пользовательского интерфейса, программист-кодировщик, технический писатель, тестировщик, руководитель проекта, пользователь, администратор системы.

Артефактами жизненного цикла ПО называются различные информационные сущности, документы и модели, создаваемые или используемые в ходе разработки и сопровождения ПО. Так, артефактами являются техническое задание, описание архитектуры, модель предметной области на каком-либо графическом языке, исходный код, пользовательская документация и т.д. Различные модели, используемые отдельными разработчиками при создании и анализе ПО, но не зафиксированные в виде доступных другим людям документов, не могут считаться артефактами.

2.Характеристика качества программного обеспечения

Для двух точек зрения — внешнего качества и внутреннего качества — в рамках ISO 9126 предложена модель качества, состоящая из 6 факторов и 27 атрибутов.
Определения факторов и атрибутов качества в этой модели приведены ниже.

1) Функциональность — способность ПО в определенных условиях решать задачи, нужные пользователям. Определяет, что именно делает ПО.

- Функциональная пригодность — способность решать нужный набор задач.
- Точность — способность выдавать нужные результаты.
- Способность к взаимодействию, совместимость — способность взаимодействовать с нужным набором других систем.
- Соответствие стандартам и правилам— соответствие ПО имеющимся стандартам, нормативным и законодательным актам, другим регулирующим нормам.
- Защищенность— способность предотвращать неавторизованный и не разрешенный доступ к данным, коммуникациям и др. элементам ПО.

2) Надежность — способность ПО поддерживать определенную работоспособность в заданных условиях.

- Зрелость, завершенность — величина, обратная частоте отказов ПО. Определяется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени.
- Устойчивость к отказам — способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением.
- Способность к восстановлению — способность восстанавливать определенный уровень работоспособности и целостность данных после отказа в рамках, заданных времени и ресурсов.
- Соответствие стандартам надежности.

3) Удобство использования или практичность — способность ПО быть удобным в обучении и использовании, а также привлекательным для пользователей.

- Понятность — показатель, обратный к усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание способов их использования для решения своих задач.
- Удобство обучения — показатель, обратный к усилиям, затрачиваемым пользователями на обучение работе с ПО.
- Удобство работы — показатель, обратный трудоемкости решения пользователями задач с помощью ПО.
- Привлекательность — способность ПО быть привлекательным для пользователей.
- Соответствие стандартам удобства использования.

4) Производительность или эффективность — способность ПО при заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам.

- Временная эффективность — способность ПО решать определенные задачи за отведенное время.
- Эффективность использования ресурсов — способность решать нужные задачи с использованием заданных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременная память, сетевые соединения, устройства ввода и вывода, и пр.

- Соответствие стандартам производительности.

5) Удобство сопровождения — удобство проведения всех видов деятельности, связанных с сопровождение программ.

- Анализируемость или удобство проведения анализа — удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий.
- Удобство внесения изменений — показатель, обратный трудозатратам на выполнение необходимых изменений.
- Стабильность — показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений.
- Удобство проверки — показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам.
- Соответствие стандартам удобства сопровождения.

6) Переносимость — способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения.

- Адаптируемость — способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных.
- Удобство установки — способность ПО быть установленным или развернутым в определенном окружении.
- Способность к сосуществованию — способность ПО сосуществовать в общем окружении с другими программами, деля с ними ресурсы.
- Удобство замены другого ПО данным — возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении.
- Соответствие стандартам переносимости.

Указанные выше характеристики используются для описания качества ПО с точки зрения его разработчиков и их руководства. Для пользовательской точки зрения, т.е. качества ПО при использовании, стандарт ISO 9126 предлагает другую систему факторов.

- Эффективность — способность решать задачи пользователей с необходимой точностью при использовании в заданном контексте.
- Продуктивность — способность предоставлять определенные результаты в рамках ожидаемых затрат ресурсов.
- Безопасность — способность обеспечивать необходимо низкий уровень риска нанесения ущерба жизни и здоровью людей, бизнесу, собственности или окружающей среде.
- Удовлетворение пользователей — способность приносить удовлетворение пользователям при использовании в заданном контексте.

3. Модели жизненного цикла ПО. Определение, классификация, преимущества и недостатки. Agile.

Жизненный цикл программного обеспечения – совокупность итерационных процедур, связанных с последовательным изменением состояния программного обеспечения от формирования исходных требований к нему до окончания его эксплуатации конечным пользователем.

Модели жизненного цикла

Кулямин рассматривает два вида моделей жизненного цикла ПО: *каскадная (*или *водопадная) модель* (см. Рис. 3, слева) и *итеративная модель* жизненного цикла, которая впервые описана в 1970 году (Рис. 3, справа).

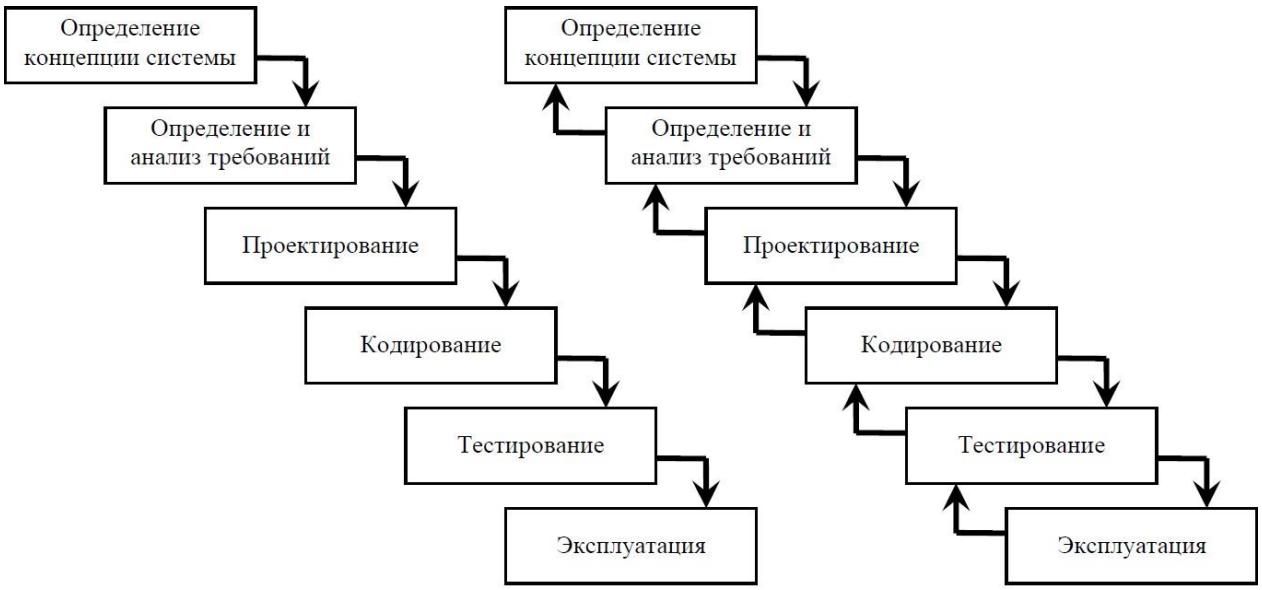


Рисунок 3. Схема каскадной и итеративной моделей жизненного цикла ПО.

Каскадная модель хорошо работает в тех случаях, когда требования к создаваемой системе удается полностью выявить и зафиксировать в начале проекта (что на практике случается не часто), и результаты всех выполняемых действий проходят тщательный анализ на внутреннюю корректность и соответствие исходным данным. В противном случае обнаруживаемые впоследствии ошибки и недоработки в результатах предыдущих шагов существенно затрудняют продвижение проекта и снижают его управляемость. Таким образом, в рамках каскадной модели верификация имеет должна выполняться в рамках всех видов деятельности для проверки корректности их результатов, и именно она в первую очередь обеспечивает успешное движение к конечной цели. Один из видов верификации — тестирование — даже выделяется в отдельный этап проекта.

В рамках итеративной модели отдельные виды деятельности уже не привязаны к этапам проекта и могут выполняться в разнообразных комбинациях. Итеративная модель позволяет быстро реагировать на изменения требований, но требует большего умения от руководителя проекта. В ее рамках различные методы верификации также имеют важнейшее значение, поскольку только с их помощью можно получить оценку качества результатов проекта, как конечных, так и промежуточных. Именно оценка качества

служит основной информацией для оценки продвижения к целям проекта, планирования следующих итераций, принятия решений о прекращении проекта или передаче его результатов заказчику.

В других источниках рассматривается несколько иная классификация моделей жизненного цикла ПО.
Каскадный жизненный цикл (Рис. 1).

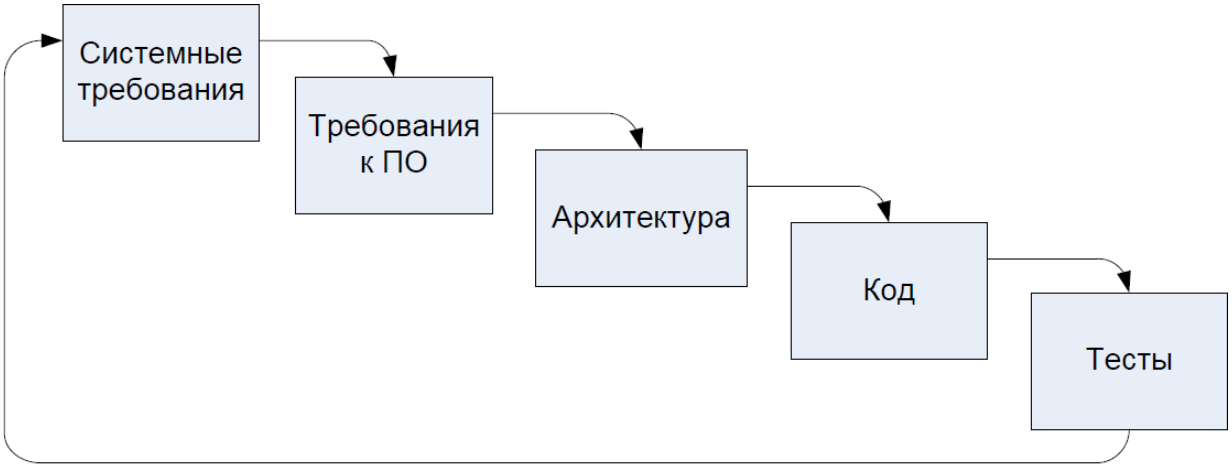


Рис. 1 Каскадная модель жизненного цикла

Особенность каскадного жизненного цикла состоит в том, что переход к следующему этапу происходит только тогда, когда полностью завершены все работы предыдущего этапа. То есть сначала полностью готовятся все требования к системе, затем по ним полностью готовятся все требования к программному обеспечению, полностью разрабатывается архитектура системы и так далее до тестирования.
Естественно, что в случае достаточно больших систем такой подход себя не оправдывает. Работа на каждом этапе занимает значительное время, а внесение изменений в первичные документы либо невозможно, либо вызывает лавинообразное изменения на всех других этапах.
Как правило, используется модификация каскадной модели, допускающая возврат на любой из ранее выполненных этапов. При этом фактически возникает дополнительная процедура принятия решения.

Особенности рассмотренных выше типов жизненного цикла сведены в таблицу 1. Из нее

Таблица 1 Сравнение различных типов жизненного цикла

Тип жизненного цикла	**Длина цикла**	**Верификация и внесение изменений**	**Интеграция отдельных компонент системы**
Каскадный	Все этапы разработки системы. Длинный	В конце разработки всей системы. Редко.	Четко определенные до начала кодирования интерфейсы.
V-образный	Все этапы разработки системы. Длинный	В конце полной разработки каждого из этапов системы. Средне.	Редко изменяемые интерфейсы.
Спиральный	Разработка одной версии системы. Средний.	В конце разработки каждого из этапов версии системы. Средне.	Периодически изменяемые интерфейсы, редко меняемые в пределах версии.
XP	Разработка одной истории. Короткий.	В конце разработки каждой истории. Очень часто	Часто изменяемые интерфейсы.

можно видеть, что различные типы жизненных циклов применяются в зависимости от планируемой частоты внесения изменений в систему, сроков разработки и ее сложности. Жизненные циклы с более короткими фазами больше подходят для разработки систем, требования к которым еще не устоялись и вырабатываются во взаимодействии с заказчиком системы во время ее разработки.

AGILE – гибкая система управления проектами.

Гибкая методология разработки (*Agile software development, agile-методы*) — серия подходов к разработке программного обеспечения, ориентированных на использование итеративной разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля. Существует несколько методик, относящихся к классу гибких методологий разработки, в частности экстремальное программирование, DSDM, Scrum, FDD.

Agile — семейство процессов разработки, а не единственный подход в разработке программного обеспечения, и определяется **Agile Manifesto**. Agile не включает практик, а определяет ценности и принципы, которыми руководствуются команды.

Манифест Agile включает в себя 4 базовых идеи и 12 принципов эффективного управления проектами.

Идеи Agile:

1. Люди и их взаимодействие важнее, чем процессы и инструменты
2. Рабочее ПО важнее, чем документация
3. Клиенты и сотрудничество с ними важнее, чем контракт и обсуждение условий
4. Готовность к внесению изменений важнее, чем первоначальный план

Принципы Agile:

1. Удовлетворять клиентов, заблаговременно и постоянно поставляя ПО (клиенты довольны, когда рабочее ПО поступает к ним регулярно и через одинаковые промежутки времени)
2. Изменять требования к конечному продукту в течение всего цикла его разработки
3. Поставлять рабочее ПО как можно чаще (раз в неделю, в две недели, в месяц и т.д.)
4. Поддерживать сотрудничество между разработчиками и заказчиком в течение всего цикла разработки
5. Поддерживать и мотивировать всех, кто вовлечен в проект (если команда мотивирована, она намного лучше справляется со своими задачами, нежели команда, члены которой условиями труда недовольны)
6. Обеспечивать непосредственное взаимодействие между разработчиками (возможность прямого контакта способствует более успешной коммуникации)
7. Измерять прогресс только посредством рабочего ПО (клиенты должны получать только функциональное и рабочее программное обеспечение)
8. Поддерживать непрерывный темп работы (команда должна выработать оптимальную и поддерживаемую скорость работы)
9. Уделять внимание дизайну и техническим деталям (благодаря эффективным навыкам и хорошему дизайну команда проекта получает возможность постоянного совершенствования продукта и работы над его улучшением)
10. Стараться сделать рабочий процесс максимально простым, а ПО – простым и понятным
11. Позволять членам команды самостоятельно принимать решения (если разработчики

могут сами принимать решения, самоорганизовываться и общаться с другими членами коллектива, обмениваясь с ними идеями, вероятность создания качественного продукта существенно возрастает)

12. Постоянно адаптироваться к меняющейся среде (благодаря этому конченый продукт будет более конкурентоспособен)

Ключевые моменты:

Agile-методология основывается, в первую очередь, на визуальном контроле. Чаще всего участники проекта, работая над достижением результата, пользуются специальными цветными карточками. Один цвет сигнализирует о завершении планирования какого-то элемента конечного продукта, другой – о завершении его разработки, третий – о готовности и т.п. Визуальный контроль позволяет команде иметь наглядное представление о текущем состоянии процесса и гарантирует одинаковое видение проекта всеми ее членами.

Еще одним важным моментом Agile-методологии является разделение всего объема проекта на несколько более мелких составных частей. Такой подход многократно упрощает процесс разработки, а отдельные группы команды могут фокусироваться каждая на своей конкретной задаче.

Работая над одним циклом, участники проекта овладевают новыми навыками и получают новые знания, а также анализируют допущенные в процессе ошибки. Все это сводит вероятность совершения подобных ошибок в будущем (в следующих циклах и других проектах) практически к нулю.

И, наконец, последний значимый элемент подхода – это спринты и ежедневные встречи. Спринтами называются ограниченные конкретными сроками (дедлайнами) отрезки времени, в течение которых команда успевает выполнить определенные задачи. Именно благодаря спринтам команда может видеть результаты своих действий.

Если же мы разделим все время, отведенное на проект, на несколько спринтов, получим конкретное их количество; пусть их будет 15. Каждый спринт длится, к примеру, две недели. Вот как раз в течение этих двух недель (времени, отведенного на спринт) участники каждый день встречаются для обсуждения процесса и прогресса.

Ежедневные встречи не должны превышать 15 минут. Организуются они для того, чтобы каждый член команды дал себе же ответ на три вопроса:

- Что я делал вчера?
- Чем я буду занят сегодня?
- Что мешает мне работать?

Ответы на эти вопросы позволяют держать под контролем процесс, понимать, на какой стадии находится каждый из участников команды, и устранять потенциальные проблемы на пути к цели. Если же обобщить, то внедрение Agile-методологии возможно, если соблюдается несколько **условий:**

1. Четко обозначается значение проекта
2. В процессе реализации активно участвует клиент
3. Общий объем работ выполняется пошагово
4. Ориентироваться следует на конкретный результат
5. Численность одной рабочей группы: от 7 до 9 человек

Существует немало методов проект-менеджмента, которые применяются разными современными компаниями. Но самыми известными и востребованными среди них считаются Scrum (Скрам) и Kanban (Канбан).

Метод Scrum

Среди всех методов системы Agile Scrum отличается тем, что делает основной упор на качественный контроль рабочего процесса.

Метод заключается в том, что разработка проекта разделяется на спринты, по окончании которых клиент получает улучшенное ПО. Спринты строго фиксируются по времени, и могут длиться от 2 до 4 недель. Рабочий процесс в одном спринте включает в себя несколько стадий:

- Определяются объемы работы
- Каждый день проводятся 15-минутные встречи, чтобы члены команды могли скорректировать свою работу и подвести промежуточные итоги
- Демонстрируются полученные результаты
- Спринты обсуждаются для поиска удачных и неудачных решений и действий

В большинстве случаев Скрам применяется в работе со сложным ПО и для разработки продукта с использованием инкрементных и итеративных методов. Благодаря ему серьезно повышается производительность команды и сокращаются временные затраты на достижение цели.

Метод Kanban

Канбан – еще один метод, делающий командную работу более результативной и продуктивной. Смысл его сводится к приданию процессу разработки максимальной прозрачности и равномерному распределению нагрузки среди участников проекта.

Работа по методу Kanban выстраивается на нескольких принципах. Во-первых, вся информация о проекте должна быть визуализирована, что позволяет видеть накладки, ошибки и недочеты и активно их устранять. Во-вторых, работа над одной задачей должна вестись одновременно всей командой – это помогает сбалансировать усилия и получаемые результаты, исключает неравномерное распределение нагрузки. И, в-третьих, время на выполнение всех задач строго контролируется, благодаря чему оптимизируется процесс и экономится время.

Достоинства Agile:

В первую очередь стоит отметить, что Agile-управление очень гибкое. Если, например, традиционная методология указывает на конкретные этапы работы, то Agile легко подстраивается под потребителя конечного продукта и требования заказчика.

Собственно, и в конечном продукте число дефектов минимизируется, ведь он является результатом тщательной проверки качества, которая проводится по завершении каждого этапа-спринта.

Кроме того, Agile быстро запускается, легко реагирует на изменения, позволяет команде разработчиков и клиентов поддерживать постоянную связь в реальном времени.

Недостатки методологии состоят в том, что, во-первых, постоянная обратная связь может приводить к тому, что все время будет переноситься и дедлайн проекта, тем самым создавая угрозу бесконечно продолжающейся работы. Если заказчик видит, например, только результаты, но не имеет представления об усилиях, потребовавшихся для их достижения, он будет все время требовать улучшений.

Второй недостаток заключается в необходимости адаптировать под изменяющиеся условия проекта проектную документацию. При отсутствии надлежащего информирования команды об изменениях или дополнительных функциях документы с функциональными требованиями или архитектурой могут оказаться неактуальными на

текущий момент времени.
Третьим существенным минусом Agile можно назвать необходимость в частых встречах. Они, конечно, способствуют повышению эффективности работы, но все же постоянное отвлечение членов команды может сказаться на процессе отрицательно, ведь внимание людей систематически уходит в сторону от решаемых задач.
Сюда же можно отнести такие вещи как необходимость в постоянном присутствии клиента, невозможность выстраивать долгосрочные планы и потребность в мотивированных и высококвалифицированных специалистах.

4. Ролевой состав коллектива разработчиков, взаимодействие между ролями. ([2] стр. 16-18)

Когда проектная команда включает более двух человек неизбежно встает вопрос ораспределении ролей, прав и ответственности в команде. Конкретный набор ролейопределяется многими факторами – количеством участников разработки и их личнымипредпочтениями, принятой методологией разработки, особенностями проекта и другимифакторами. Практически в любом коллективе разработчиков можно выделитьперечисленные ниже роли. Некоторые из них могут вовсе отсутствовать, при этомотдельные люди могут выполнять сразу несколько ролей, однако общий состав меняетсямало.

Заказчик (заявитель). Эта роль принадлежит представителю организации,заказавшей разрабатываемую систему. Обычно заявитель ограничен в своемвзаимодействии и общается только с менеджерами проекта и специалистом посертификации или внедрению. Обычно заказчик имеет право изменять требования кпродукту (только во взаимодействии с менеджерами), читать проектную исертификационную документацию, затрагивающую нетехнические особенностиразрабатываемой системы.

Менеджер проекта. Эта роль обеспечивает коммуникационный канал междузаказчиком и проектной группой. Менеджер продукта управляет ожиданиями заказчика,разрабатывает и поддерживает бизнес-контекст проекта. Его работа не связана напрямуюс продажей продукта, он сфокусирован на продукте, его задача - определить и обеспечитьтребования заказчика. Менеджер проекта имеет право изменять требования к продукту ифинальную документацию на продукт.

Менеджер программы. Эта роль управляет коммуникациями и взаимоотношениямив проектной группе, является в некотором роде координатором, разрабатываетфункциональные спецификации и управляет ими, ведет график проекта и отчитывается посостоянию проекта, инициирует принятие критичных для хода проекта решений.Менеджер программы имеет право изменять функциональные спецификацииверхнего уровня, план-график проекта, распределение ресурсов по задачам. Часто напрактике роль менеджера проекта и менеджера программы выполняет один человек.

Разработчик. Разработчик принимает технические решения, которые могут бытьреализованы и использованы, создает продукт, удовлетворяющий спецификациям иожиданиям заказчика, консультирует другие роли в ходе проекта. Он участвует в обзорах,реализует возможности продукта, участвует в создании функциональных спецификаций,отслеживает и исправляет ошибки за приемлемое время. В контексте конкретного проектароль разработчика может подразумевать, например, инсталляцию программногообеспечения, настройку продукта или услуги. Разработчик имеет доступ ко всейпроектной документации, включая документацию по тестированию, имеет право наизменение программного кода системы в рамках своих служебных обязанностей.

Специалист по тестированию. Специалист по тестированию определяет стратегиютестирования, тест-требования и тест-планы для каждой из фаз проекта, выполняеттестирование системы, собирает и анализирует отчеты о прохождении тестирования.Тест-требования должны покрывать системные требования, функциональныеспецификации, требования к надежности и нагрузочной способности, пользовательскиеинтерфейсы и собственно программный код. В реальности роль специалиста потестированию часто разбивается на две – разработчика тестов и

тестировщика.Тестировщик выполняет все работы по выполнению тестов и сбору информации,разработчик тестов – всю остальные работы.

Специалист по контролю качества. Эта роль принадлежит члену проектнойгруппы, который осуществляет взаимодействие с разработчиком, менеджером программыи специалистами по безопасности и сертификации с целью отслеживания целостнойкартины качества продукта, его соответствия стандартам и спецификациям,предусмотренным проектной документацией. Следует различать специалиста потестированию и специалиста по контролю качества. Последний не является членомтехнического персонала проекта, ответственным за детали и технику работы. Контролькачества подразумевает в первую очередь контроль самих процессов разработки ипроверку их соответствия определенным в стандартах качества критериям.

Специалист по сертификации. При разработке систем, к надежности которыхпредъявляются повышенные требования, перед вводом системы в эксплуатацию требуетсяподтверждение со стороны уполномоченного органа (обычно государственного)соответствия ее эксплуатационных характеристик заданным критериям. Такоесоответствие определяется в ходе сертификации системы. Специалист по сертификацииможе либо быть представителем сертифицирующих органов, включенным в составколлектива разработчиков, либо наоборот – представлять интересы разработчиков всертифицирующем органе. Специалист по сертификации приводит документацию напрограммную систему в соответствие требованиям сертифицирующего органа, либоучаствует в процессе создания документации с учетом этим требованиям. Такжеспециалист по сертификации ответственен за все взаимодействие между коллективомразработчиков и сертифицирующим органом. Важной особенностью роли являетсянезависимость специалиста от проектной группы на всех этапах создания продукта.Взаимодействие специалиста с членами проектной группы ограничивается менеджерамипо проекту и по программе.

Специалист по внедрению и сопровождению. Участвует в анализе особенностейплощадки заказчика, на которой планируется проводить внедрение разрабатываемой системы, выполняет весь спектр работ по установке и настройке системы, проводитобучение пользователей.

Специалист по безопасности. Данный Специалист ответственен за весь спектрвопросов безопасности создаваемого продукта. Его работа начинается с участия в18написании требований к продукту и заканчивается финальной стадией сертификациипродукта.

Инструктор. Эта роль отвечает за снижение затрат на дальнейшее сопровождениепродукта, обеспечение максимальной эффективности работы пользователя. Важно, чторечь идет о производительности пользователя, а не системы. Для обеспеченияоптимальной продуктивности инструктор собирает статистику по производительностипользователей и создает решения для повышения производительности, в том числе сиспользованием различных аудиовизуальных средств. Инструктор принимает участие вовсех обсуждениях пользовательского интерфейса и архитектуры продукта.

Технический писатель. Лицо, осуществляющее эту роль, несет обязанности поподготовке документации к разработанному продукту, финального описанияфункциональных возможностей. Так же он участвует в написании сопроводительныхдокументов (системы помощи, руководство пользователя).

5. Задачи и цели процесса верификации.

Основная цель процесса верификации – доказательство того, что результат разработки соответствует предъявленным к нему требованиям.
Обычно процесс верификации проводится сверху вниз, начиная от общих требований, заданных в техническом задании и/или спецификации на всю информационную систему до детальных требований на программные модули и их взаимодействие.
В состав задач верификации входит последовательная проверка того, что в программной системе:

- общие требования к информационной системе, предназначенные для программной реализации, корректно переработаны в спецификацию требований высокого уровня к комплексу программ, удовлетворяющих исходным системным требованиям;
- требования высокого уровня правильно переработаны в архитектуру ПО и в спецификации требований к функциональным компонентам низкого уровня, которые удовлетворяют требованиям высокого уровня;
- спецификации требований к функциональным компонентам ПО, расположенным между компонентами высокого и низкого уровня, удовлетворяют требованиям более высокого уровня;
- архитектура ПО и требования к компонентам низкого уровня корректно переработаны в удовлетворяющие им исходные тексты программных и информационных модулей;
- исходные тексты программ и соответствующий им исполняемый код не содержат ошибок.

Кроме того, верификации на соответствие спецификации требований на конкретный проект программного средства подлежат требования к технологическому обеспечению жизненного цикла ПО, а также требования к эксплуатационной и технологической документации.
Цели верификации ПО достигаются посредством последовательного выполнения комбинации из инспекций проектной документации и анализа их результатов, разработки тестовых планов тестирования и тест-требований, тестовых сценариев и процедур, и последующего выполнения этих процедур. Тестовые сценарии предназначены для проверки внутренней непротиворечивости и полноты реализации требований.
Выполнение тестовых процедур должно обеспечивать демонстрацию соответствия испытываемых программ исходным требованиям.
На выбор эффективных методов верификации и последовательность их применения в наибольшей степени влияют основные характеристики тестируемых объектов:

- класс комплекса программ, определяющийся глубиной связи его функционирования с реальным временем и случайными воздействиями из внешней среды, а также требования к качеству обработки информации и надежности функционирования;
- сложность или масштаб (объем, размеры) комплекса программ и его функциональных компонентов, являющихся конечными результатами разработки;
- преобладающие элементы в программах: осуществляющие вычисления сложных выражений и преобразования измеряемых величин или обрабатывающие логические и символьные данные для подготовки и отображения решений.

Верификация решает следующие ***задачи***:

- Выявление дефектов (ошибок, недоработок, неполноты и пр.) различных артефактов разработки ПО (требований, проектных решений, документации или кода), что позволяет устранять их и поставлять пользователям и заказчикам более правильное и надежное ПО.

- Выявление наиболее критичных и наиболее подверженных ошибкам частей создаваемой или сопровождаемой системы.
- Контроль и оценка качества ПО во всех его аспектах.
- Предоставление всем заинтересованным лицам (руководителям, заказчикам, пользователям и пр.) информации о текущем состоянии проекта и характеристиках его результатов.
- Предоставление руководству проекта и разработчикам информации для планирования дальнейших работ, а также для принятия решений о продолжении проекта, его прекращении или передаче результатов заказчику.

6.Верификация различных артефактов жизненного цикла ПО

Артефакты жизненного цикла ПО можно разделить на технические и организационные. К техническим артефактам относятся описание требований (техническое задание), описание проектных решений (эскизный и технический проекты), исходный код (текст программы), документация пользователей и администраторов (рабочая документация), сама работающая система. Техническими также являются вспомогательные артефакты для проведения верификации и валидации — формальные модели требований и проектных решений, наборы тестов и компоненты тестового окружения, модели поведения реального окружения системы. Организационными артефактами являются структура работ, разнообразные проектные планы (план-график работ, план конфигурационного управления, план обеспечения качества, план обхода и преодоления рисков, планы проверок и испытаний и пр.), описания системы качества, описания процессов и процедур выполнения отдельных работ. Верификация может и должна проводиться для всех видов артефактов, создаваемых при разработке и сопровождении программных систем.

• При верификации организационных документов и процессов проверяется, насколько выбранные формы организации, планы и методы выполнения работ соответствуют задачам, решаемым в рамках проекта, и ограничениям по срокам и бюджету, то есть, что с помощью выбранных методов и технологий проект действительно можно выполнить в рамках контракта. Проверяется также, что команда проекта в достаточной степени владеет используемыми технологиями разработки, или же, что запланированы необходимые мероприятия по обучению. 17 В дальнейшем в рамках данной статьи рассматриваются, в основном, методы верификации, нацеленные на оценку качества технических, а не организационных артефактов процесса разработки.

• При верификации описания требований одной из первых задач верификации является оценка осуществимости требований с помощью технологий, взятых на вооружение в проекте и в рамках выделенных на проект ресурсов. Проверяются также характеристики требований, указанные в стандартах IEEE 830 [31] и IEEE 1233 [32], а именно следующие.

- Однозначность. Требования должны однозначно, недвусмысленно выражать нужные ограничения.
- Непротиворечивость или согласованность. Различные требования не должны противоречить друг другу или основным законам предметной области.
- Внутренняя полнота. Требования должны описывать поведения системы во всех возможных в контексте ее работы ситуациях. Все значимые законы предметной области и нормы действующих в ней стандартов должны быть учтены в требованиях как ограничения на работу системы.
- Минимальность. Требования не должны быть сводимы друг к другу на основе формальной логики и основных законов предметной области.
- Проверяемость. В каждой затрагиваемой требованием ситуации должен быть способ однозначно установить, выполнено оно или нарушено.
- Систематичность. Требования должны быть представлены в рамках единой системы, с четким указанием связей между ними, с уникальными идентификаторами и набором определенных атрибутов: приоритетом, риском внесения изменений, критичностью для пользователей и пр.

Кроме этого, требования должны адекватно и достаточно полно отражать нужды и потребности пользователей и других заинтересованных лиц. Требования должны затрагивать все существенные для пользователей аспекты качества системы: помимо функциональных требований, должны быть адекватно отражены требования к производительности, надежности, удобству использования, переносимости и удобству сопровождения. Для проверки 18 адекватности и полноты отражения реальных потребностей пользователей необходимо проводить валидацию.

• При верификации проектных решений проверяются следующие свойства.

o Все проектные решения связаны с требованиями и действительно нацелены на их реализацию. Все требования нашли отражение в проектных решениях.

o Проектные документы точно и полно формулируют принятые решения, отдельные их элементы не противоречат друг другу. o При оформлении проектных документов учтены все правила корректности составления документов такого типа на соответствующих языках. Если используются графические нотации, такие как DFD, ERD или UML, то все диаграммы составлены с соблюдением всех правил и ограничений этих языков.

o Для проектных решений, связанных с критически важными требованиями к системе, например, по ее безопасности и защищенности, необходимо с помощью максимально строгих методов установить их корректность, т.е. то, что они действительно реализуют соответствующие требования во всех возможных в контексте работы системы ситуациях.

• При верификации исходного кода системы проверяют указанные ниже характеристики.

o Все элементы кода связаны с проектными решениями и требованиями и корректно реализуют соответствующие проектные решения.

o Код написан в соответствии с синтаксическими и семантическими правилами выбранных языков программирования, а также с принятыми в организации и данном проекте стандартами оформления текстов программ (coding rules, coding conventions). Выполнены требования к удобству сопровождения кода, в коде отсутствуют неясные места, все его элементы можно протестировать с помощью сценариев возможной работы системы.

o В исходном коде отсутствуют пути выполнения, достижимые в условиях работы системы и приводящие к ее сбоям, зацикливаниям или тупиковым 19 ситуациям, разрушению процессов и данных проверяемой системы или объемлющей, исключительным ситуациям, непредусмотренным в требованиях и проектных решениях, и пр. Во всех возможных в контексте работы системы сценариях выполнения кода принятые проектные решения и требования соблюдаются, и нет элементов кода, выполняющих непредусмотренные требованиями действия. Эти правила на практике невозможно проверить полностью, но при верификации стремятся как можно более достоверно подтвердить его. При возрастании критичности требований, связанных с компонентами и элементами кода, требуется более строгое подтверждение, и используются более строгие и трудоемкие методы.

• Верификация самой работающей системы или ее компонентов, которые можно выполнять независимо, призвана проверить следующее.

o Система или ее компоненты действительно способны работать в том окружении, в котором они нужны пользователям, или же в рамках достаточно точной имитации этого окружения.

o Поведение системы или ее компонентов на возможных сценариях их использования соответствует требованиям по всем измеримым характеристикам. Это, снова, невозможно проверить полностью. Однако, для наиболее критичных требований и сценариев использования применяются более строгие и полные методы проверки соответствия. Часто проверяется также соответствие поведения системы и ее компонентов реальным нуждам пользователей — это уже является валидацией.

• При верификации пользовательской документации проверяется следующее.

o Документация содержит полное, точное и непротиворечивое описание поведения системы.

o Описанное в документации поведение соответствует реальному поведению системы.

• Верификации также должны подвергаться тестовые планы или планы других мероприятий по верификации, а также тесты или материалы, подготовленные 20 для проведения верификации других артефактов, например различные формальные модели. В

этих случаях проверяются такие характеристики.

- Подготовленные планы соответствуют основным рискам проекта и уделяют различным его артефактам ровно такое внимание, которое требуется, исходя из их зрелости и важности для проекта.
- Методы верификации, которые планируется применять, действительно способны дать лучшие результаты (с точки зрения обнаружения ошибок и получения достоверных оценок качества, отнесенных к затратам) в намеченных для них областях.
- Подготовленные материалы (тесты, списки возможных ошибок для инспекций, формальные модели требований или окружения системы и пр.) соответствуют контексту использования системы, требованиям к проверяемым артефактам и связанным с ними проектным решениям и могут быть использованы в качестве входных данных для выбранных методов проведения верификации.

7.Требования ФСТЭК РД НДВ. Общие положения. Термины и определения. Перечень требований.

1. ОБЩИЕ ПОЛОЖЕНИЯ

1.1. Классификация распространяется на ПО, предназначенное для защиты информации ограниченного доступа.
1.2. Устанавливается четыре уровня контроля отсутствия недекларированных возможностей. Каждый уровень характеризуется определенной минимальной совокупностью требований.
1.3. Для ПО, используемого при защите информации, отнесенной к государственной тайне, должен быть обеспечен уровень контроля не ниже третьего.
1.4. Самый высокий уровень контроля - первый, достаточен для ПО, используемого при защите информации с грифом «ОВ».
Второй уровень контроля достаточен для ПО, используемого при защите информации с грифом «СС».
Третий уровень контроля достаточен для ПО, используемого при защите информации с грифом «С».
1.5 Самый низкий уровень контроля - четвертый, достаточен для ПО, используемого при защите конфиденциальной информации.

2. ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

2.1. Недекларированные возможности - функциональные возможности ПО, не описанные или не соответствующие описанным в документации, при использовании которых возможно нарушение конфиденциальности, доступности или целостности обрабатываемой информации.
Реализацией недекларированных возможностей, в частности, являются программные закладки.
2.2. Программные закладки – преднамеренно внесенные в ПО функциональные объекты, которые при определенных условиях (входных данных) инициируют выполнение не описанных в документации функций ПО, приводящих к нарушению конфиденциальности, доступности или целостности обрабатываемой информации.
2.3. Функциональный объект – элемент программы, осуществляющий выполнение действий по реализации законченного фрагмента алгоритма программы.
В качестве функциональных объектов могут выступать процедуры, функции, ветви, операторы и т.п.
2.4. Информационный объект - элемент программы, содержащий фрагменты информации, циркулирующей в программе. В зависимости от языка программирования в качестве информационных объектов могут выступать переменные, массивы, записи, таблицы, файлы, фрагменты оперативной памяти и т.п.
2.5. Маршрут выполнения функциональных объектов – определенная алгоритмом последовательность выполняемых функциональных объектов.
2.6. Фактический маршрут выполнения функциональных объектов – последовательность фактически выполняемых функциональных объектов при определённых условиях (входных данных).
2.7. Критический маршрут выполнения функциональных объектов – такой маршрут, при выполнении которого существует возможность неконтролируемого нарушения установленных правил обработки информационных объектов.
2.8. Статический анализ исходных текстов программ – совокупность методов контроля (не)соответствия реализованных и декларированных в документации функциональных

возможностей ПО, основанных на структурном анализе и декомпозиции исходных текстов программ.

2.9. Динамический анализ исходных текстов программ – совокупность методов контроля (не)соответствия реализованных и декларированных в документации функциональных возможностей ПО, основанных на идентификации фактических маршрутов выполнения функциональных объектов с последующим сопоставлением маршрутам, построенным в процессе проведения статического анализа.

3. ТРЕБОВАНИЯ К УРОВНЮ КОНТРОЛЯ

3.1. ПЕРЕЧЕНЬ ТРЕБОВАНИЙ

Таблица 1

N	Наименование требования	Уровень контроля			
		4	3	2	1
	Требования к документации				
1	Контроль состава и содержания документации				
1.1.	Спецификация (ГОСТ 19.202-78)	+	=	=	=
1.2.	Описание программы (ГОСТ 19.402-78)	+	=	=	=
1.3.	Описание применения (ГОСТ 19.502-78)	+	=	=	=
1.4.	Пояснительная записка (ГОСТ 19.404-79)	-	+	=	=
1.5.	Тексты программ, входящих в состав ПО (ГОСТ 19.401-78)	+	=	=	=
	Требования к содержанию испытаний				
2.	Контроль исходного состояния ПО	+	=	=	=
3.	Статический анализ исходных текстов программ				
3.1.	Контроль полноты и отсутствия избыточности исходных текстов	+	+	+	=
3.2.	Контроль соответствия исходных текстов ПО его объектному (загрузочному) коду	+	=	=	+
3.3.	Контроль связей функциональных объектов по управлению	-	+	=	=
3.4.	Контроль связей функциональных объектов по информации	-	+	=	=
3.5.	Контроль информационных объектов	-	+	=	=
3.6.	Контроль наличия заданных конструкций в исходных текстах	-	-	+	+
3.7.	Формирование перечня маршрутов выполнения функциональных объектов	-	+	+	=
3.8.	Анализ критических маршрутов выполнения функциональных объектов	-	-	+	=
3.9.	Анализ алгоритма работы функциональных объектов на основе блок-схем, диаграмм и т. п., построенных по исходным текстам контролируемого ПО	-	-	+	=
4.	Динамический анализ исходных текстов программ				

4.1.	Контроль выполнения функциональных объектов	-	+	+	=
4.2.	Сопоставление фактических маршрутов выполнения функциональных объектов и маршрутов, построенных в процессе проведения статического анализа	-	+	+	=
5.	Отчетность	+	+	+	+

Обозначения
"-" - нет требований к данному уровню;
"+" - новые или дополнительные требования;
"=" - требования совпадают с требованиями предыдущего уровня.

Все, что идет дальше подробнее раскрывает уровни контроля, возможно, не обязательно при ответе.

8.Требования к средствам ЭП. Классификация в зависимости от способностей противостоять атакам (перечислить наименования классов). Пункты требований – кратко, своими словами, какие аспекты средств ЭП затрагивают.

Принятый ещё в апреле 2011 года Федеральный закон N 63-ФЗ « Об электронной подписи » определил следующие два вида электронной подписи (ЭП): **простая ЭП и усиленная ЭП**, из которых вторая в свою очередь бывает неквалифицированной и квалифицированной.

Простая ЭП, по большому счёту, даже и не совсем электронная подпись в классическом математическом понимании — в качестве простой ЭП можно использовать хоть одноразовые коды, хоть и вовсе пароли:

Простой электронной подписью является электронная подпись, которая посредством использования кодов, паролей или иных средств подтверждает факт формирования электронной подписи определенным лицом.

Усиленная же электронная подпись обязательно использует криптографические преобразования. Вот требования для усиленной неквалифицированной ЭП (или просто неквалифицированной ЭП):

1) получена в результате криптографического преобразования информации с использованием ключа электронной подписи;
2) позволяет определить лицо, подписавшее электронный документ;
3) позволяет обнаружить факт внесения изменений в электронный документ после момента его подписания;
4) создается с использованием средств электронной подписи.

Как видно, усиленная ЭП в отличие от простой ЭП должна позволять обнаруживать изменения в электронном документе (понятно, что с помощью пароля такое не реализовать), а формировать её нужно с использованием Средств ЭП:

средства электронной подписи — шифровальные (криптографические) средства, используемые для реализации хотя бы одной из следующих функций — создание электронной подписи, проверка электронной подписи, создание ключа электронной подписи и ключа проверки электронной подписи;

Квалифицированная усиленная подпись отличается от неквалифицированной усиленной двумя принципиальными дополнительными требованиями:

1) ключ проверки электронной подписи указан в квалифицированном сертификате;
2) для создания и проверки электронной подписи используются средства электронной подписи, имеющие подтверждение соответствия требованиям, установленным в соответствии с настоящим Федеральным законом.

Квалифицированный сертификат — сертификат ключа проверки электронной подписи, соответствующий требованиям, установленным настоящим Федеральным законом и иными принимаемыми в соответствии с ним нормативными правовыми актами, и созданный аккредитованным удостоверяющим центром либо федеральным органом исполнительной власти, уполномоченным в сфере использования электронной подписи;

Виды электронных подписей (ЭП) и их особенности

	Простая электронная подпись	Усиленная электронная подпись	
		Неквалифицированная	Квалифицированная
Как создаётся	коды, пароли или иные средства	получается в результате криптографического преобразования информации с использованием ключа электронной подписи	
Средство ЭП	нет	создается с использованием средств электронной подписи	создается с использованием средств электронной подписи, имеющих подтверждение соответствия требованиям, установленным в соответствии с настоящим Федеральным законом
Сертификат ключа проверки ЭП	нет	любой	квалифицированный
Что должна делать	подтверждает факт формирования электронной подписи определенным лицом	позволяет определить лицо, подписавшее электронный документ	
		позволяет обнаружить факт внесения изменений в электронный документ после момента его подписания	

Средства ЭП — это определённый вид шифровальных (криптографических) средств, которые используются для какой-либо одной или любой комбинации из функций:

- создание электронной подписи,
- проверка электронной подписи,
- создание ключа электронной подписи и ключа проверки электронной подписи.

Требования для Средств ЭП:
1) позволяют установить факт изменения подписанного электронного документа после момента его подписания;
2) обеспечивают практическую невозможность вычисления ключа электронной подписи из электронной подписи или из ключа ее проверки.

Статья 11 определяет, что :
Статья 11. Средства ЭП должны противостоять угрозам, представляющим собой целенаправленные действия с использованием аппаратных и (или) программных средств с целью нарушения безопасности защищаемой средством ЭП информации или с целью создания условий для этого (далее — атака).
В последующих статьях 12-18 определяются классы Средств ЭП КС1, КС2, КС3, КВ2, КА1 в зависимости от их способности противостоять таким атакам.

К основным особенностям СЗИ класса **КС1** относится их возможность противостоять атакам, проводимым из-за пределов контролируемой зоны. При этом подразумевается, что создание способов атак, их подготовка и проведение осуществляется без участия специалистов в области разработки и анализа криптографических СЗИ. Предполагается, что информация о системе, в которой применяются указанные СЗИ, может быть получена из открытых источников. Если криптографическое СЗИ может противостоять атакам, блокируемым средствами класса КС1, а также проводимым в пределах контролируемой зоны, то такое СЗИ соответствует классу **КС2**. При этом допускается, например, что при подготовке атаки могла стать доступной информация о физических мерах защиты информационных систем, обеспечении контролируемой зоны и пр. В случае возможности противостоять атакам при наличии физического доступа к средствам вычислительной техники с установленными криптографическими СЗИ говорят о соответствии таких средств классу **КС3**. Если криптографическое СЗИ противостоит атакам, при создании которых участвовали специалисты в области разработки и анализа указанных средств, в том числе научно-исследовательские центры, была возможность проведения лабораторных исследований средств защиты, то речь идет о соответствии

классу КВ. Если к разработке способов атак привлекались специалисты в области использования НДВ системного программного обеспечения, была доступна соответствующая конструкторская документация и был доступ к любым аппаратным компонентам криптографических СЗИ, то защиту от таких атак могут обеспечивать средства класса **КА**.

12. В зависимости от способностей противостоять атакам средства ЭП подразделяются на классы.
13. Средства ЭП класса КС1 противостоят атакам, при создании способов, подготовке и проведении которых используются следующие возможности:
13.1. Самостоятельное осуществление создания способов атак, подготовки и проведения атак.
13.2. Действия на различных этапах жизненного цикла средства ЭП[5].
13.3. Проведение атаки только извне пространства, в пределах которого осуществляется контроль за пребыванием и действиями лиц и (или) транспортных средств (далее - контролируемая зона[6]).

13.4. Проведение на этапах разработки, производства, хранения, транспортировки средств ЭП и этапе ввода в эксплуатацию средств ЭП (пусконаладочные работы) следующих атак:

- внесение несанкционированных изменений в средство ЭП и (или) в компоненты СФ, в

том числе с использованием вредоносных программ;
- внесение несанкционированных изменений в документацию на средство ЭП и на компоненты СФ.

13.5. Проведение атак на следующие объекты:

- документацию на средство ЭП и на компоненты СФ;
- защищаемые электронные документы;
- ключевую, аутентифицирующую и парольную информацию средства ЭП;
- средство ЭП и его программные и аппаратные компоненты;
- аппаратные средства, входящие в СФ, включая микросхемы с записанным микрокодом BIOS, осуществляющей инициализацию этих средств (далее - аппаратные компоненты СФ);
- программные компоненты СФ;
- данные, передаваемые по каналам связи;
- помещения, в которых находится совокупность программных и технических элементов систем обработки данных, способных функционировать самостоятельно или в составе других систем (далее - СВТ), на которых реализованы средства ЭП и СФ;
- иные объекты атак, которые при необходимости указываются в ТЗ на разработку (модернизацию) средства ЭП с учетом используемых в информационной системе информационных технологий, аппаратных средств (далее - АС) и программного обеспечения (далее - ПО).

13.6. Получение следующей информации:

- общих сведений об информационной системе, в которой используется средство ЭП (назначение, состав, оператор, объекты, в которых размещены ресурсы информационной системы);
- сведений об информационных технологиях, базах данных, АС, ПО, используемых в информационной системе совместно со средством ЭП;
- сведений о физических мерах защиты объектов, в которых размещены средства ЭП;
- сведений о мерах по обеспечению контролируемой зоны объектов информационной системы, в которой используется средство ЭП;
- сведений о мерах по разграничению доступа в помещения, в которых находятся СВТ, на которых реализованы средства ЭП и СФ;
- содержания находящейся в свободном доступе документации на аппаратные и программные компоненты средства ЭП и СФ;
- общих сведений о защищаемой информации, используемой в процессе эксплуатации средства ЭП;
- всех возможных данных, передаваемых в открытом виде по каналам связи, не защищенным от несанкционированного доступа (далее - НСД) к информации организационно-техническими мерами;
- сведений о линиях связи, по которым передается защищаемая средством ЭП информация;
- сведений обо всех проявляющихся в каналах связи, не защищенных от НСД к информации организационно-техническими мерами, нарушениях правил эксплуатации средства ЭП и СФ;
- сведений обо всех проявляющихся в каналах связи, не защищенных от НСД к информации организационно-техническими мерами, неисправностях и сбоях аппаратных компонентов средства ЭП и СФ;
- сведений, получаемых в результате анализа любых сигналов от аппаратных компонентов средства ЭП и СФ, которые может перехватить нарушитель.

13.7. Использование:

- находящихся в свободном доступе или используемых за пределами контролируемой зоны АС и ПО, включая аппаратные и программные компоненты средства ЭП и СФ;
- специально разработанных АС и ПО.

13.8. Использование в качестве среды переноса от субъекта к объекту (от объекта к субъекту) атаки действий, осуществляемых при подготовке и (или) проведении атаки (далее - канал атаки):

- не защищенных от НСД к информации организационно-техническими мерами каналов связи (как вне контролируемой зоны, так и в ее пределах), по которым передается защищаемая средством ЭП информация;
- каналов распространения сигналов, сопровождающих функционирование средства ЭП и СФ.

13.9. Проведение атаки из информационно-телекоммуникационных сетей, доступ к которым не ограничен определенным кругом лиц.

13.10. Использование АС и ПО из состава средств информационной системы, используемых на местах эксплуатации средства ЭП (далее - штатные средства) и находящихся за пределами контролируемой зоны.

14. Средства ЭП класса КС2 противостоят атакам, при создании способов, подготовке и проведении которых используются возможности, перечисленные в подпунктах 13.1 - 13.10 настоящих Требований, и следующие дополнительные возможности:

14.1. Проведение атаки при нахождении как вне пределов, так и в пределах контролируемой зоны.

14.2. Использование штатных средств, ограниченное мерами, реализованными в информационной системе, в которой используется средство ЭП, и направленными на предотвращение и пресечение несанкционированных действий.

15. Средства ЭП класса КС3 противостоят атакам, при создании способов, подготовке и проведении которых используются возможности, перечисленные в подпунктах 13.1 - 13.10, 14.1, 14.2 настоящих Требований, и следующие дополнительные возможности:

15.1. Доступ к СВТ, на которых реализованы средство ЭП и СФ.

15.2. Возможность располагать аппаратными компонентами средства ЭП и СФ в объеме, зависящем от мер, направленных на предотвращение и пресечение несанкционированных действий, реализованных в информационной системе, в которой используется средство ЭП.

16. Средства ЭП класса КВ1 противостоят атакам, при создании способов, подготовке и проведении которых используются возможности, перечисленные в подпунктах 13.1 - 13.10, 14.1, 14.2, 15.1, 15.2 настоящих Требований, и следующие дополнительные возможности:

16.1. Создание способов атак, подготовка и проведение атак с привлечением

специалистов, имеющих опыт разработки и анализа средств ЭП, включая специалистов в области анализа сигналов, сопровождающих функционирование средства ЭП и СФ.

16.2. Проведение лабораторных исследований средства ЭП, используемого вне контролируемой зоны, в объеме, зависящем от мер, направленных на предотвращение и пресечение несанкционированных действий, реализованных в информационной системе, в которой используется средство ЭП.

17. Средства ЭП класса КВ2 противостоят атакам, при создании способов, подготовке и проведении которых используются возможности, перечисленные в подпунктах 13.1 - 13.10, 14.1, 14.2, 15.1, 15.2, 16.1, 16.2 настоящих Требований, и следующие дополнительные возможности:

17.1. Создание способов атак, подготовка и проведение атак с привлечением специалистов, имеющих опыт разработки и анализа средств ЭП, включая специалистов в области использования для реализации атак возможностей прикладного ПО, не описанных в документации на прикладное ПО.

17.2. Постановка работ по созданию способов и средств атак в научно-исследовательских центрах, специализирующихся в области разработки и анализа средств ЭП и СФ.

17.3. Возможность располагать исходными текстами входящего в СФ прикладного ПО.

18. Средства ЭП класса КА1 противостоят атакам, при создании способов, подготовке и проведении которых используются возможности, перечисленные в подпунктах 13.1 - 13.10, 14.1, 14.2, 15.1, 15.2, 16.1, 16.2, 17.1 - 17.3 настоящих Требований, и следующие дополнительные возможности:

18.1. Создание способов атак, подготовка и проведение атак с привлечением специалистов, имеющих опыт разработки и анализа средств ЭП, включая специалистов в области использования для реализации атак возможностей системного ПО, не описанных в документации на системное ПО.

18.2. Возможность располагать всей документацией на аппаратные и программные компоненты СФ.

18.3. Возможность располагать всеми аппаратными компонентами средства ЭП и СФ.

9. Принципы разработки и модернизации шифровальных (криптографических) средств защиты информации. Введение. Общие принципы построения СКЗИ. Базовые положения для программного обеспечения СКЗИ.

Ниже я выписал из общего стандарта ответ на поставленный вопрос. Прочитать весь стандарт можно здесь.

Область применения

Настоящие рекомендации распространяются на шифровальные (криптографические) средства защиты информации, предназначенные для использования на территории Российской Федерации. Настоящие рекомендации определяют принципы разработки и модернизации шифровальных (криптографических) средств защиты информации, не содержащей сведений, составляющих государственную тайну. Принципы обеспечения безопасности защищаемой информации до ее обработки в СКЗИ в настоящем документе не рассматриваются. Принципы разработки и модернизации шифровальных (криптографических) средств защиты информации, перечисленных в пункте 4 Положения ПКЗ – 2005 [1], могут регулироваться отдельными рекомендациями по стандартизации.

Общие принципы построения СКЗИ

В настоящем разделе приводятся общие принципы, на которых основывается разработка новых или модификация действующих СКЗИ.

1. СКЗИ должно обеспечивать безопасность защищаемой информации при реализации атак в процессе обработки защищаемой информации в СКЗИ и/или при условии несанкционированного доступа к защищенной СКЗИ информации в процессе ее хранения или передачи по каналам связи.
2. СКЗИ должно реализовывать одну или несколько криптографических функций. В зависимости от реализуемых криптографических функции СКЗИ может быть отнесено к одному или нескольким средствам:
   - средству шифрования;
   - средству имитозащиты;
   - средству электронной подписи;
   - средству кодирования;
   - средству изготовления ключевых документов;
   - ключевому документу.
3. В настоящем документе средства кодирования не рассматриваются.
4. Все СКЗИ подразделяются на 5 классов, упорядоченных по старшинству:
   - класс КС1 – младший по отношению к классам КС2, КС3, КВ и КА;
   - класс КС2 – младший по отношению к классам КС3, КВ, КА и старший по отношению к классу КС1;
   - класс КС3 – младший по отношению к классам КВ, КА и старший по отношению к классам КС1, КС2;
   - класс КВ – младший по отношению к классу КА и старший по отношению к классам КС1, КС2, КС3;
   - класс КА – старший по отношению к классам КС1, КС2, КС3, КВ.

5. Класс разрабатываемого (модернизируемого) СКЗИ определяется заказчиком СКЗИ путем формирования перечня подлежащих защите объектов ИС и совокупности возможностей, которые могут быть использованы при создании способов, подготовке и проведении атак на указанные объекты, с учетом применяемых в ИС информационных технологий, среды функционирования и аппаратных средств.

6. Совокупность возможностей, которые могут быть использованы при создании способов, подготовке и проведении атак, должна формироваться на основе приведенной в приложении А базовой совокупности и согласовываться с ФСБ России.

7. Класс разрабатываемого (модернизируемого) СКЗИ, совокупность возможностей, которые могут быть использованы при создании способов, подготовке и проведении атак, и состав криптографических функций СКЗИ должны определяться в ТЗ на разработку (модернизацию) СКЗИ.
8. Разрабатываемые (модернизируемые) СКЗИ должны удовлетворять требованиям по безопасности информации, устанавливаемым в соответствии с законодательством Российской Федерации (Положение ПКЗ-2005 [1], статья 1, пункт 12). Совокупность предъявляемых к СКЗИ требований определяется:
   - а) классом СКЗИ;
   - б) составом криптографических функций, которые должны быть реализованы в СКЗИ.
9. В ТЗ на разработку (модернизацию) СКЗИ могут предъявляться дополнительные требования к СКЗИ, не противоречащие принципам настоящего документа.
10. Допускается проведение процедуры оценки соответствия СКЗИ произвольной совокупности из предъявленных к нему требований с выдачей заключения ФСБ России о соответствии (несоответствии) СКЗИ данной совокупности требований.
11. Сертификат соответствия СКЗИ выдается только в том случае, если для ввода СКЗИ в эксплуатацию не требуется проведение дополнительных тематических исследований СКЗИ после утверждения положительного заключения ФСБ России о соответствии СКЗИ всем предъявляемым к нему требованиям.
12. 4.12 В отдельных случаях, при наличии соответствующего обоснования по решению ФСБ России может быть разрешена эксплуатация СКЗИ, когда отдельные положения требований по безопасности информации к ним не выполнены.

Базовые положения для программного обеспечения СКЗИ

При разработке и встраивании ПО СКЗИ рекомендуется учитывать следующие положения:

1. Для проведения тематических исследований ПО СКЗИ для всех классов должно быть представлено в виде исходных текстов, исполняемого кода и документации.
2. Исходные тексты ПО СКЗИ должны удовлетворять следующим условиям: исходные тексты ПО СКЗИ рекомендуется выполнять в соответствии с ГОСТ 19.401. При этом специализированная организация, проводящая тематические исследования, может устанавливать собственные требования к содержанию и оформлению текстов ПО СКЗИ;
   - исходные тексты ПО СКЗИ должны содержать комментарии, достаточные для понимания алгоритма функционирования ПО СКЗИ;
   - исходные тексты ПО СКЗИ должны содержать полный набор файлов, необходимый для воспроизведения из них исполняемого кода, идентичного представленному для проведения тематических исследований;
   - бинарные и ассемблерные вставки, вставки информационных массивов и входящие в состав исходных текстов фрагменты, не имеющие прямого отношения к реализации криптографических функций СКЗИ, должны быть документированы и обоснованы.
3. Документация на ПО СКЗИ должна включать в себя:
   - спецификацию ПО СКЗИ;
   - описание ПО СКЗИ;
   - описание применения ПО СКЗИ;

- пояснительную записку.

4. Спецификация ПО СКЗИ (см. 6.2.3, перечисление а) должна быть выполнена в соответствии с ГОСТ 19.202.
5. Описание ПО СКЗИ (см. п. 6.2.3, перечисление б) должно быть выполнено в соответствии с ГОСТ 19.402 и содержать, в частности:
   - основные сведения о составе ПО СКЗИ (с указанием контрольных сумм файлов, входящих в состав ПО СКЗИ);
   - логическую структуру ПО СКЗИ;
   - описание методов, приемов и правил эксплуатации средств технологического оснащения, использованных при создании ПО СКЗИ;
   - инструкцию по сборке из исходных текстов ПО СКЗИ загрузочных и исполняемых модулей СКЗИ.
6. Описание применения ПО СКЗИ (см. п. 6.2.3, перечисление в) должно быть выполнено в соответствии с ГОСТ 19.502-78 и содержать:
   - сведения о назначении ПО СКЗИ;
   - сведения об области применения ПО СКЗИ;
   - сведения о классе решаемых ПО СКЗИ задач;
   - сведения об ограничениях при применении ПО СКЗИ;
   - сведения о минимальной конфигурации технических средств;
   - сведения о СФ;
   - сведения о порядке работы СКЗИ.
7. Пояснительная записка (см. п. 6.2.3, перечисление г) должна содержать, в частности:
   - все сведения о назначении компонентов, входящих в состав ПО СКЗИ;
   - перечень всех реализованных в ПО СКЗИ функций;
   - сведения о параметрах (аргументах) реализованных в ПО СКЗИ функций;
   - сведения о формируемых кодах возврата реализованных в ПО СКЗИ функций;
   - перечень экспортируемых функций ПО СКЗИ;
   - описание используемых переменных;
   - описание алгоритмов функционирования ПО СКЗИ;
   - описание критериев, методики и результатов тестирования реализованных в ПО СКЗИ функций.
8. Подача на вход любых значений параметров экспортируемых функций ПО СКЗИ не должна приводить к появлению уязвимостей, позволяющих реализовывать успешные атаки на СКЗИ.
9. В случае выявления при проведении тематических исследований значений параметров экспортируемых функций ПО СКЗИ, приводящих к появлению уязвимостей, позволяющих реализовывать успешные атаки на СКЗИ, составляется список таких функций и значений параметров. Этот список исключается из документации на СКЗИ (см. 6.5.4, перечисление е), представляемой разработчикам, осуществляющим встраивание СКЗИ в ИС. Действие сертификата соответствия СКЗИ на функции из указанного списка не распространяется.

10. Требования PA-DSS. Назначение. Требование 1.1, 2.2, 2.5

Требования и процедуры аудита безопасности стандартов безопасности данных платежных приложений (PA-DSS) индустрии платежных карт (PCI) регламентируют требования к безопасности и процедурам аудита для поставщиков платежных приложений.
Стандарт PA-DSS распространяется на поставщиков приложений и иных разработчиков приложений, которые хранят, обрабатывают или передают данные держателей карт и (или) критичные аутентификационные данные

Требования:
1.1 Запрещается хранить критичные аутентификационные данные после авторизации (даже в зашифрованном виде). В случае получения критичных аутентификационных данных все данные должны стать невосстановимыми по завершении процесса авторизации. К критичным аутентификационным данным относятся данные, перечисленные в требованиях 1.1.1 – 1.1.3. (

-содержимое дорожки (содержимое магнитной полосы, находящейся на обратной стороне карты, его аналог на чипе либо в ином месте),

-кода CVC или значения, используемого для подтверждения транзакций,

-персонального идентификационного номера (PIN), а также зашифрованного PINблока.)

Пояснение: Критичные аутентификационные данные состоят из полных данных на магнитной дорожке, кода или значения подтверждения подлинности карты и данных PIN-кода. Хранение критичных аутентификационных данных запрещается. Эти данные представляют интерес для злоумышленников, поскольку позволяют им генерировать поддельные платежные карты и осуществлять мошеннические операции. Эмитенты платежных карт или компании, предоставляющие услуги эмиссии или поддерживающие этот процесс, часто создают и управляют критичными аутентификационными данными в рамках процесса эмиссии. Эмитенты и компании, обеспечивающие услуги эмиссии, могут иметь обоснованную необходимость хранения критичных аутентификационных данных. Такая необходимость должна иметь обоснование с точки зрения бизнеса, а хранимые данные должны быть надежно защищены. Для неэмитентов сохранение критичных аутентификационных данных после аутентификации запрещено, и приложение должно иметь механизм надежного удаления данных без возможности их восстановления.

2.2 Следует маскировать основной номер держателя карты при его отображении (максимально возможное количество знаков для отображения – первые шесть и последние четыре), чтобы только сотрудники с обоснованной коммерческой необходимостью могли видеть весь основной номер держателя карты

Примечание. Это требование не заменяет собой иные более строгие требования к отображению данных держателей карт (например, юридические требования или требования к брендированию платежных карт на чеках кассовых терминалов (в местах продаж)

Пояснение: полное отображение основного номера держателя карты на экранах компьютеров, квитанциях об операциях с платежными картами, факсах, в бумажных отчетах и т. п. может привести к тому, что эти данные станут известны неавторизованным лицам и могут быть использованы в мошеннических целях. Это требование касается защиты основного номера держателя карты, отображаемого на экранах, бумажных квитанциях, распечатках и т. д., и его следует отличать от требования 2.3 стандарта PA-DSS, которое касается защиты основного номера

держателя карты при его хранении в файлах, базах данных и т. Д

2.5 В платежные приложения должны быть полностью внедрены все процессы и процедуры управления ключами шифрования данных держателей карт, включая по крайней мере:

- Генерация стойких криптографических ключей *(Платежное приложение должно генерировать стойкие ключи, как описано в документе "Глоссарий PCI DSS и PADSS: основные определения, аббревиатуры и сокращения" в определении термина "стойкая криптография".)*
- Безопасное распространение ключей *(Платежное приложение должно распространять ключи безопасным образом, то есть в зашифрованном виде и только посредством авторизованных процессов.)*
- Безопасное хранение ключей шифрования *(Платежное приложение должно хранить*
- *ключи безопасным образом (например,шифруя их при помощи ключа шифрования ключей).)*
- Смена ключей шифрования, период действия которых истек *(Период действия ключа — это период времени, в течение которого ключ шифрования можно использовать для решения определенной задачи. Аспекты, рассматриваемые при определении криптопериода, включают, но не ограничиваются: надежность базового алгоритма, размер или длина ключа, риск кражи ключа и конфиденциальность данных, зашифрованных с помощью ключа. Периодическая замена ключей шифрования является обязательной для минимизации рисков несанкционированного получения ключей шифрования и последующего дешифрования данных.)*
- Изъятие или смена ключей (например, архивация, уничтожение и (или) аннулирование) при нарушении целостности (например, увольнение сотрудника, обладающего информацией об открытом коде ключа и т. д.), а также ключей, относительно которых существуют подозрения в их компрометации. *(Ключи, которые больше не используются или в которых нет необходимости, а также ключи, относительно которых существуют подозрения о компрометации, должны быть изъяты и (или) уничтожены, чтобы устранить возможность их использования. Если требуется хранение таких ключей (например, для поддержки архивированных зашифрованных данных), то они должны быть надежно защищены.*
- *Платежное приложение должно обеспечивать возможность смены ключей, которые необходимо заменить или относительно которых существуют подозрения о компрометации.)*
- Если платежное приложение поддерживает ручное управление ключами шифрования в открытом виде, соответствующие процедуры обязаны предусматривать разделение знания и двойной контроль ключей.
- Защита от неавторизованной смены ключей *(Платежное приложение должно определить*
- *для пользователей приложения методы, обеспечивающие возможность только*
- *авторизованной смены ключей. Конфигурация приложения не должна допускать или принимать подмену ключей, инициированную неавторизованными источниками или неожиданными процессами)*

12. Наиболее опасные программные ошибки согласно CWE.

Что такое CWE? Что такое «дефект безопасности ПО»?
Общий перечень дефектов безопасности ПО (Common Weakness Enumeration, CWE) предназначен для разработчиков и специалистов по обеспечению безопасности ПО. Он представляет собой официальный реестр или словарь общих дефектов безопасности, которые могут проявиться в архитектуре, проектировании, коде или реализации ПО, и могут быть использованы злоумышленниками для получения несанкционированного доступа к системе. Данный перечень был разработан в качестве универсального формального языка для описания дефектов безопасности ПО, а также в качестве стандарта для измерения эффективности инструментов, выявляющих такие дефекты, и для распознавания, устранения и предотвращения этих дефектов.

Дефекты безопасности ПО — это дефекты, сбои, ошибки, уязвимости и прочие проблемы реализации, кода, проектирования или архитектуры ПО, которые могут сделать системы и сети уязвимыми к атакам злоумышленников, если их вовремя не исправить. К таким проблемам относятся: переполнения буферов, ошибки форматной строки и т.д.; проблемы структуры и оценки валидности данных; манипуляции со специальными элементами; ошибки каналов и путей; проблемы с обработчиками; ошибки пользовательского интерфейса; обход каталога и проблемы с распознаванием эквивалентности путей; ошибки аутентификации; ошибки управления ресурсами; недостаточный уровень проверки данных; проблемы оценки входящих данных и внедрение кода; проблемы предсказуемости и недостаточная «случайность» случайных чисел.

Список Топ 25 SANS — инструмент для обучения и ознакомления программистов, чтобы помочь им предотвратить возникновение уязвимостей в разрабатываемом ПО за счет выявления и предотвращения наиболее общих ошибок до введения ПО в эксплуатацию. Пользователи ПО могут использовать этот список при составлении ТЗ на безопасное программное обеспечение. Исследователи безопасности программного обеспечения могут использовать Топ 25, чтобы сосредоточиться на конкретном критичном подмножестве всех известных слабых мест безопасности. Наконец, менеджеры по программному обеспечению и главные менеджеры по информационным технологиям могут использовать список Топ 25 в качестве измерителя прогресса в их работе по обеспечению безопасности их программных продуктов.

Список Топ 25 — результат сотрудничества между Институтом SANS, MITRE и многими ведущими экспертами по безопасности программного обеспечения США и Европы.

Уязвимости разбиты на 3 категории:

• Опасное взаимодействие между компонентами (6 ошибок)

• Рискованное управление ресурсами (8 ошибок)

• Проницаемая защита (11 ошибок)

1.1 Опасное взаимодействие между компонентами

Эти уязвимости связаны с опасными путями, по которым данные посылают и получают между отдельными компонентами, модулями, программами, процессами, потоками или системами (Таблица 1).

Таблица 1

CWE ID	Name
CWE-89	Некорректная нейтрализация специальных элементов, используемых в команде SQL (‘SQL-инъекция ‘)
CWE-78	Некорректная нейтрализация специальных элементов, используемых в команде OS (‘Инъекция Команды OS’)
CWE-79	Некорректная нейтрализация входных данных при создании веб-страницы (‘Межсайтовый скриптинг’)
CWE-434	Неограниченная закачка файла опасного типа
CWE-352	Подделка запроса с перекрёстной ссылкой (межсайтового скриптинга) (CSRF)
CWE-601	Перенаправление с URL к недоверенному сайту (‘открытое перенаправление’)

1.2 Опасное управление ресурсами

Уязвимости этой категории связаны с некорректными способами управления ПО созданием, использованием, передачей или удалением важных системных ресурсов (Таблица 2).

Таблица 2

CWE ID	Name
CWE-120	Копирование в буфер без проверки размера входных данных (‘Классическое переполнение буфера’)
CWE-22	Некорректное ограничение имени пути к каталогу ограниченного доступа (‘обход каталога’)
CWE-494	Загрузка кода без проверки целостности
CWE-829	Добавление функциональности из недоверенной сферы управления
CWE-676	Использование потенциально опасной функции
CWE-131	Неправильное вычисление размера буфера
CWE-134	Неконтролируемая строка форматирования
CWE-190	Целочисленное переполнение или циклический возврат

1.3 Проницаемая защита

Уязвимости этой категории связаны с защитными методами, которые часто неправильно используются или просто игнорируются (Таблица 3).

Таблица 3

CWE ID	Name
CWE-306	Отсутствие аутентификации для критической функции
CWE-862	Отсутствие авторизации
CWE-798	Использование встроенных учетных данных
CWE-311	Отсутствие шифрования критичных данных

CWE ID	Name
CWE-807	Доверие ненадёжным входным данным в принятии решений по безопасности
CWE-250	Выполнение с чрезмерными полномочиями
CWE-863	Некорректная авторизация
CWE-732	Некорректное присвоение разрешения на критический ресурс
CWE-327	Использование взломанного или опасного алгоритма шифрования
CWE-307	Некорректное ограничение числа дополнительных попыток аутентификации
CWE-759	Использование одностороннего хэширования без использования случайного числа (пароля)

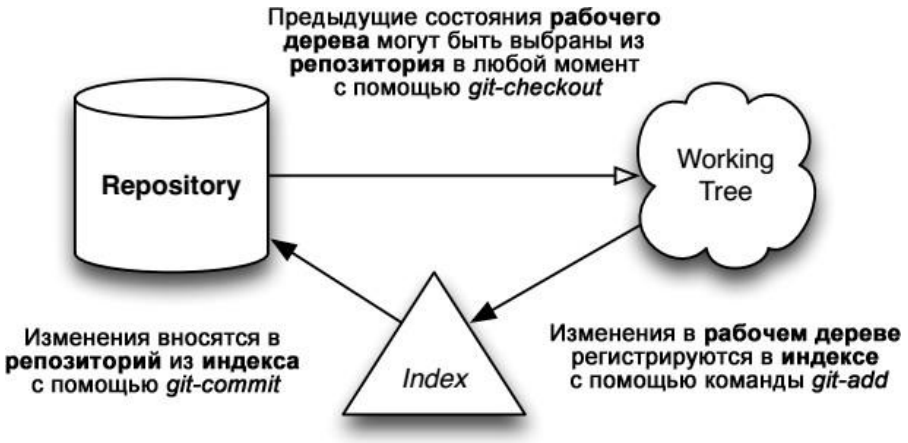
13.Инструменты поддержки процесса верификации. Системы контроля версий исходных текстов. Git: основные сущности.

Список терминов, встречающихся в тексте и необходимых для его понимания.

Рабочее дерево (Working tree) — Любая директория в вашей файловой системе, связанная с *репозиторием* (что можно видеть по наличию в ней поддиректории «.git»). Включает в себя все файлы и поддиректории.

- **Коммит (Commit)**. «моментальный снимок» *рабочего дерева* в какой-то момент времени.
- **Репозиторий (Repository)** — это набор *коммитов*, т.е. просто архив прошлых состояний *рабочего дерева* проекта на вашей или чьей-то машине.
- **Ветка (Branch)** — просто имя для *коммита*, также называемое *ссылкой* (reference). Определяет происхождение — «родословную» *коммита*, и таким образом, является типичным представлением «ветки разработки»
- **Checkout** — операция переключения между *ветками* или восстановления файлов *рабочего дерева*
- **Метка (Tag)** — также имя для *коммита*, отличающееся от *ветки* тем, что оно всегда постоянно указывает на один и тот же *коммит*, а также может иметь свое текстовое описание
- **Мастер (Master)**. Условно «главная» или «основная» *ветка репозитория*, но по сути ничем не отличающаяся от прочих *веток*
- **Индекс (Index)**. В отличие от других подобных инструментов, Git не передает изменения из *рабочего дерева* в *репозиторий* напрямую. Вместо этого изменения сначала регистрируются в *индексе*, или «области подготовки» (staging area). Это можно рассматривать как способ «подтверждения» ваших изменений перед совершением *коммита*, который запишет в *репозиторий* все одобренные изменения.
- **HEAD** — заголовок. Используется *репозиторием* для определения того, что выбрано с помощью *checkout*
  - Если субъект *checkout* — ветка, то *HEAD* будет ссылаться на нее, показывая, что имя ветки должно быть обновлено во время следующего *коммита*
  - Если субъект *checkout* — коммит, то *HEAD* будет ссылаться только на него. В этом случае *HEAD* называется обособленным (detached)

Взаимодействие с Git обычно выглядит так:
После создания репозитория работа происходит в рабочем дереве. Как только достигается значительная веха — устранение бага, момент, когда, наконец, все начинает компилироваться — вы добавляете свои изменения в индекс. Как только все, что вы собираетесь коммитить, оказывается в индексе, вы записываете его содержимое в репозиторий. На диаграмме ниже — типичный цикл жизни проекта:



Теперь давайте посмотрим, как каждая из показанных на картинке сущностей работает в git.

Репозиторий: отслеживание содержимого директории

Git содержит моментальные снимки содержимого директории.

Файловая система начинается с корневой директории, которая обычно состоит из других директорий, множество которых имеет узлы-листья, т.е. файлы, содержащие данные. Метаданные файлов хранятся как в директории (имена), так и в i-узлах, которые ссылаются на содержимое этих файлов (размер, тип, разрешения доступа и тп), Каждый i-узел имеет уникальный номер, идентифицирующий содержимое соответствующего файла. Хотя в директории может существовать множество объектов, указывающих на конкретный i-узел (т.е. хард-линки), именно i-узел «владеет» контентом, хранящимся в вашей файловой системе.

Git представляет содержимое ваших файлов в виде так называемых «фрагментов» («blobs»), которые являются узлами-листьями в структуре, очень похожей на директорию и называемой деревом. Фрагмент в Git маркируется путем вычисления SHA-1 хэша от его размера и содержания. Для всех мыслимых применений это всего лишь произвольный номер, как и у i-узла, за исключением двух дополнительных свойств: во-первых, он контролирует неизменность содержимого фрагмента, а во-вторых гарантирует, что одно и то же содержимое будет всегда представлено одним и тем же фрагментом независимо от того, где оно будет встречаться — в разных коммитах, репозиториях, или даже в разных частях Интернета. Если несколько деревьев ссылаются на тот же фрагмент, то это похоже на хард-линки: фрагмент не исчезнет из вашего репозитория до тех пор, пока на него существует хотя бы одна ссылка.

Разница между файлом в файловой системе и фрагментом в Git состоит в том, что сам фрагмент не хранит метаданных о его содержимом. Вся эта информация хранится в дереве, к которому принадлежит фрагмент. Одно дерево может считать это содержимое файлом «foo», созданным в августе 2004, в то время как другое дерево может знать то же содержимое под именем файла «bar», созданным на пять лет позже. В нормальной файловой системе такие два файла с совпадающим содержимым, но различающимися метаданными, будут всегда представлены как два независимых файла. Все объекты с идентичным содержимым будут общими независимо от их местонахождения.

Знакомимся с фрагментом

Теперь, когда общая картина нарисована, посмотрим на практические примеры. Создадим

тестовый репозиторий и покажем, как Git работает в нем с самого низа.

```
$ mkdir sample; cd sample
```

```
$ echo 'Hello, world!' > greeting
```

Здесь создается новая директорию *sample*. Репозиторий еще не создан, но уже можно начать использовать некоторые команды Git, чтобы понять, что он собирается делать. Для начала можно узнать, под каким хэшем Git будет хранить приветствие:

```
$ git hash-object greeting
```

```
af5626b4a114abcb82d63db7c8082c3c4756e51b
```

Следующий шаг — это инициализация нового репозитория и коммит в него:

```
$ git init
```

```
$ git add greeting
```

```
$ git commit -m "Added my greeting"
```

На этой стадии фрагмент должен быть в системе и использовать определенный выше хэш id. Для удобства Git требует только начальные цифры хэша, однозначно определяющие фрагмент в репозитории. Обычно 6 или 7 цифр для этого достаточно.

```
$ git cat-file -t af5626b
```

```
blob
```

```
$ git cat-file blob af5626b
```

```
Hello, world!
```

Данное содержимое будет иметь тот же самый идентификатор независимо от времени жизни репозитория или положения файла в нем. То есть, данные гарантированно сохранены навсегда.

Таким образом, фрагмент — это фундаментальная единица данных в Git.

Как образуются деревья?

Каждый коммит содержит единственное дерево. Фрагменты создаются при помощи «нарезки» содержимого вашего файла, и деревья владеют фрагментами, но мы еще не видели, как образуются эти деревья и как деревья связываются со своими родительскими коммитами.

Давайте снова начнем с нового репозитория, но на этот раз сделаем все вручную.

```
$ rm -fr greeting.git
```

```
$ echo 'Hello, world!' > greeting
```

```
$ git init
```

```
$ git add greeting
```

Все начинается с добавления файла в индекс. Пока можно считать, что индекс — это то, что используется для первоначального создания фрагментов из файлов. Когда был добавлен файл *greeting*, в репозитории произошли изменения. Хотя это еще и не коммит, есть способ на них посмотреть:

```
$ git log  # не сработает -коммиты отсутствуют
```

```
fatal: bad default revision 'HEAD'
```

```
$ git ls-files --stage #покажет фрагменты, на которые ссылается индекс
```

```
100644 af5626b4a114abcb82d63db7c8082c3c4756e51b 0 greeting
```

Коммитов еще нет, а объект уже есть. У него тот же хэш id, с которого все начиналось, так что он представляет содержимое файла *greeting*.
На этот фрагмент еще не ссылается ни дерево ни коммиты. Пока ссылка на него есть только в файле *.git/index*, содержащем ссылки на фрагменты и деревья, которые собственно и образуют текущий индекс. А теперь давайте создадим в репозитории дерево, на котором и будут висеть фрагменты:

```
$ git write-tree # записать содержимое индекса в дерево
```

```
0563f77d884e4f79ce95117e2d686d7d6e282887
```

Дерево, содержащее одни и те же фрагменты (и поддеревья) будет всегда иметь тот же самый хэш. Хотя до сих пор нет объекта коммита, но зато в этом репозитории есть объект дерево, содержащее фрагменты. Цель низкоуровневой команды *write-tree* — взять содержимое индекса и поместить его в новое дерево для дальнейшего создания коммита. Новый объект коммита можно создать вручную используя данное дерево напрямую. Именно это и делает команда *commit-tree* — берет хэш id дерева и создает для него объект коммита. Если бы было необходимо, чтобы у коммита был родительский объект, то нужно было бы указать его явно с использованием ключа -p.

```
$ echo "Initial commit" | git commit-tree 0563f77
```

```
5f1bc85745dcccce6121494fdd37658cb4ad441f
```

Чтобы ветка «master» теперь ссылалась на данный коммит, нужно использовать команду *update-ref*:

```
$ git update-ref refs/heads/master 5f1bc857
```

После создания ветки *master*, мы должны связать с ней наше дерево. Это обычно происходит когда вы переключаете ветку:

```
$ git symbolic-ref HEAD refs/heads/master
```

Эта команда создает символическую ссылку *HEAD* на ветку *master*. Это очень важно, так как все дальнейшие коммиты из рабочего дерева теперь будут автоматически обновлять значение *refs/heads/master*.

Теперь можно использовать команду *log* для просмотра свежесозданного коммита.

```
$ git log
```

```
commit 5f1bc85745dcccce6121494fdd37658cb4ad441f
```

```
Author: John Wiegley <johnw@newartisans.com>
```

```
Date: Mon Apr 14 11:14:58 2008 -0400
```

```
Initial commit
```

Вся прелесть коммитов

В Git ветки не являются отдельной сущностью — здесь есть исключительно фрагменты, деревья и коммиты. Так как у коммита может иметься один и более родителей, и эти коммиты в свою очередь могут принадлежать своим родителям, мы можем рассматривать единичный коммит как ветвление — ведь он знает всю свою «родословную».

Вы можете в любой момент посмотреть все коммиты верхнего уровня используя команду branch

```
$ git branch -v
```

```
* master 5f1bc85 Initial commit
```

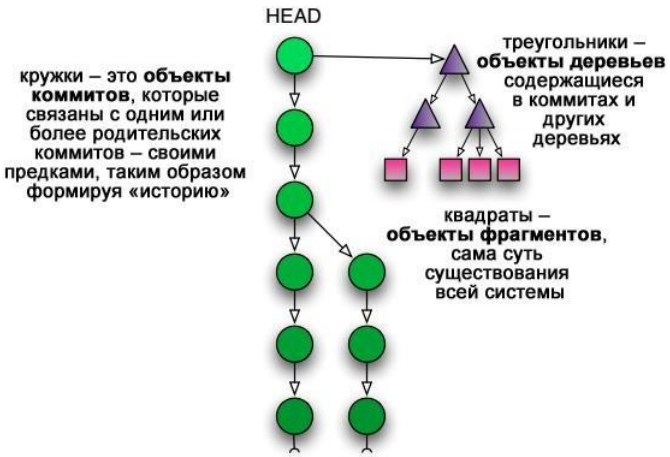
Ветка — это просто именованная ссылка на коммит.

Способ перейти к конкретному коммиту — использовать команду *checkout*:

```
$ git checkout 5f1bc85
```

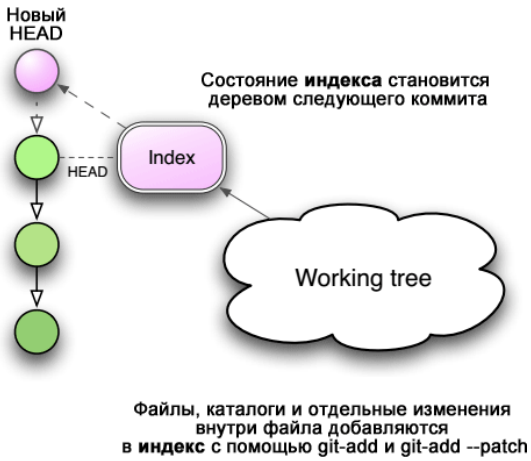
Еще одно достоинство системы, основанной на коммитах, это возможность перефразировать термины даже самых сложных систем контроля версий на

простом языке. Например, если у коммита несколько родителей, то это — объединенный (merged) коммит. Или, если у коммита несколько потомков, то он представляет собой предка ветки и тп. Но для Git между этими сущностями нет никакой разницы для него мир представляет собой просто набор коммитов, каждый из которых содержит дерево, ссылающееся на другие деревья и фрагменты, хранящие ваши данные.
Вот иллюстрация того, как все это работает:



Индекс: Встречайте посредника

Между вашими файлами, хранящимися в файловой системе и фрагментами Git, хранящимися в репозитории, стоит сущность — Git индекс. Это индекс в том смысле, что он ссылается на набор новых деревьев и фрагментов, которые вы создали с помощью команды add. Эти новые объекты, возможно, скоро будут объединены в новое дерево с целью коммита в ваш репозиторий. Но до тех пор на них ссылается только индекс. Это значит, что если вы разрегистрируете изменения в индексе с помощью reset, то ваши фрагменты будут, фактически потеряны, и впоследствии удалены. Индекс — это просто область подготовки для вашего следующего коммита, и для его существования есть уважительная причина: он поддерживает модель разработки, которая дает возможность построения коммита в несколько стадий.



Но, при этом, есть способ почти полностью игнорировать индекс — путем передачи ключа -a команде commit.

При использовании в Git команды *commit –a* происходит следующее: новые неизвестные файлы игнорируются, а новые, добавленные в команде add, попадут в репозиторий, так же, как и любые изменения существующих файлов.

Рассмотрим следующий пример: пусть у вас имеется исходный файл foo.c, в котором вы сделали два набора независимых изменений. И теперь вы хотите разделить эти изменения на 2 отдельных коммита, каждый со своим описанием.

```
$ git add --patch foo.c
```

```
<интерактивно выбираем изменения для первого коммита>
```

```
$ git commit -m "Сообщение первого коммита"
```

```
$ git add foo.c # добавляем оставшиеся изменения
```

```
$ git commit -m "Сообщение второго коммита"
```

reset — это редактор ссылок, индекса и рабочего дерева.

```
$ git add foo.c # добавляем изменения в индекс в виде нового фрагмента
$ git reset HEAD # убираем все изменения, внесенные в индекс
$ git add foo.c # мы ошиблись, возвращаем все назад
```

Как сделать откат

Опция --hard у команды *reset* — штука потенциально опасная, так как если вы сделаете hard reset для текущего HEAD, то это приведет к стиранию всех изменений в вашем рабочем дереве так, что ваши текущие файлы станут соответствовать содержимому HEAD.
Существует еще одна команда — *checkout*, которая действует так же, как и reset --hard, но только в случае, если индекс пуст. В противном случае она приводит ваше рабочее дерево в соответствие с индексом.

1. git log - просмотр логов, показывает дельту (разницу/diff), привнесенную каждым коммитом.

```
commit 9a452d9cdbdb57e7e4f2b09f8ce2f776cd56657a
Author: devlabuser <user@mail.ru>
Date: Wed Jul 31 18:35:47 2013 +0400

    first commit

commit d528335724dfc15461996ed9d44d74f23ce6a075
Author: devlabuser <user@mail.ru>
Date: Wed Jul 31 06:24:57 2013 -0700
```

```
10
11    Initial commit
```

2. Копируем идентификатор коммита, до которого происходит откат.
3. Откатываемся до последнего успешного коммита (указываем последний коммит):

```
1 $ git reset --hard 9a452d955bdb57e7e4f2b09f8ce2fbb6cd56377a
2 HEAD is now at 9a45779 first commit
```

Можно откатить до последней версии ветки:

```
1 $ git reset --hard origin/dev
2 HEAD is now at 9a45779 first commit
```

После того, как откат сделан, и выполнен очередной локальный коммит, при попытке сделать push в удаленный репозиторий, git может начать ругаться, что версия вашей ветки младше чем на github и вам надо сделать pull. Это лечится принудительным коммитом:

```
1 git push -f origin master
```

14. Методы верификации. Тестирование. Задачи и цели тестирования программного кода.

Тестирование программного кода – процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под дефектом здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям).

Задача тестирования при таком подходе – определение условий, при которых проявляются дефекты системы и протоколирование этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных участков программного кода и никогда не входит исправление дефектов – это задача отладки, которая выполняется по результатам тестирования системы.

Цель применения процедуры тестирования программного кода – минимизация количества дефектов, в особенности существенных, в конечном продукте. Тестирование само по себе не может гарантировать полного отсутствия дефектов в программном коде системы. Однако, в сочетании с процессами верификации и валидации, направленными на устранение противоречивости и неполноты проектной документации (в частности – требований на систему), грамотно организованное тестирование дает гарантию того, что система удовлетворяет требованиям и ведет себя в соответствии с ними во всех предусмотренных ситуациях.

При разработке систем повышенной надежности, например, авиационных, гарантии надежности достигаются при помощи четкой организации процесса тестирования, определения его связи с остальными процессами жизненного цикла, введения количественных характеристик, позволяющих оценивать успешность тестирования. При этом, чем выше требования к надежности системы (ее уровень критичности), тем более жесткие требования предъявляются.

Таким образом, в первую очередь мы рассматриваем не конкретные результаты тестирования конкретной системы, а общую организацию процесса тестирования, используя подход «хорошо организованный процесс дает качественный результат». Такой подход является общим для многих международных и отраслевых стандартах качества, о которых более подробно будет рассказано в конце данного курса. Качество разрабатываемой системы при таком подходе является следствием организованного процесса разработки и тестирования, а не самостоятельным неуправляемым результатом.

Поскольку современные программные системы имеют весьма значительные размеры, при тестировании их программного кода используется метод функциональной декомпозиции. Система разбивается на отдельные модули (классы, пространства имен и т.п.), имеющие определенную требованиями функциональность и интерфейсы. После этого по отдельности тестируется каждый модуль – выполняется модульное тестирование. Затем выполняется сборка отдельных модулей в более крупные конфигурации – выполняется интеграционное тестирование, и наконец, тестируется система в целом – выполняется системное тестирование.

С точки зрения программного кода, модульное, интеграционное и системное тестирование имеют много общего, поэтому в данной теме основное внимание будет уделено модульному тестированию.

В ходе модульного тестирования каждый модуль тестируется как на соответствие требованиям, так и на отсутствие проблемных участков программного кода, могущих вызвать отказы и сбои в работе системы. Как правило, модули не работают вне системы – они принимают данные от других модулей, перерабатывают их и передают дальше. Для того, чтобы с одной стороны, изолировать модуль от системы и исключить влияние потенциальных ошибок системы, а с другой стороны – обеспечить модуль всеми

необходимыми данными, используется тестовое окружение.

Задача тестового окружения – создать среду выполнения для модуля, эмулировать все внешние интерфейсы, к которым обращается модуль. Об особенностях организации тестового окружения пойдет речь в данной теме.

Типичная процедура тестирования состоит в подготовке и выполнении тестовых примеров (также называемых просто тестами). Каждый тестовый пример проверяет одну «ситуацию» в поведении модуля и состоит из списка значений, передаваемых на вход модуля, описания запуска и выполнения переработки данных – тестового сценария, и списка значений, которые ожидаются на выходе модуля в случае его корректного поведения. Тестовые сценарии составляются таким образом, чтобы исключить обращения к внутренним данным модуля, все взаимодействие должно происходить только через его внешние интерфейсы.

Выполнение тестового примера поддерживается тестовым окружением, которое включает в себя программную реализацию тестового сценария. Выполнение начинается с передачи модулю входных данных и запуска сценария. Реальные выходные данные, полученные от модуля в результате выполнения сценария сохраняются и сравниваются с ожидаемыми. В случае их совпадения тест считается пройденным, в противном случае – не пройденным. Каждый не пройденный тест указывает либо на дефект в тестируемом модуле, либо в тестовом окружении, либо в описании теста.

Совокупность описаний тестовых примеров составляет тест-план – основной документ, определяющий процедуру тестирования программного модуля. Тест-план задает не только сами тестовые примеры, но и порядок их следования, который также может быть важен.

При тестировании часто бывает необходимо учитывать не только требования к системе, но и структуру программного кода тестируемого модуля. В этом случае тесты составляются таким образом, чтобы детектировать типичные ошибки программистов, вызванные неверной интерпретацией требований. Применяются проверки граничных условий, проверки классов эквивалентности. Отсутствие в системе возможностей, не заданных требованиями, гарантируют различные оценки покрытия программного кода тестами, т.е. оценки того, какой процент тех или иных языковых конструкций выполнен в результате выполнения всех тестовых примеров.

15. Методы верификации. Тестирование. Виды тестирования

Классификация видов тестирования достаточно сложна, потому что может проводиться по нескольким разным аспектам.

- По уровню или масштабу проверяемых элементов системы тестирование делится на следующие виды.
  - Модульное или компонентное (unit testing, component testing) — проверка корректности работы отдельных компонентов системы, выполнения ими своих функций и предполагаемых проектом характеристик.
  - Интеграционное (integration testing) — проверка корректности взаимодействий внутри отдельных групп компонентов.
  - Системное (system testing) — проверка работы системы в целом, выполнения ею своих основных функций, с использованием определенных ресурсов, в окружении с заданными характеристиками.
- По проверяемым характеристикам качества тестирование может быть тестирование функциональности, производительности (и по времени, и по другим ресурсам), надежности, переносимости или удобства использования. Более специфические виды тестирования, нацеленные на оценку отдельных атрибутов, — тестирование защищенности, совместимости или восстановления при сбоях.

Специфическим видом тестирования, нацеленным на минимизацию риска того, что в результате доработки или внесения ошибок качество системы изменилось в худшую сторону, является регрессионное тестирование (regression testing). При его проведении используется уже применявшийся ранее набор тестов, и оно должно выявить различия между результатами, полученными на этих тестах ранее, и наблюдаемыми после внесения изменений.

- По источникам данных, используемых для построения тестов, тестирование относится к одному из следующих видов.
  - Тестирование черного ящика (black-box testing, часто также называется тестированием соответствия, conformance testing, или функциональным тестированием, functional testing) — нацелено на проверку соблюдения требований. Использует критерии полноты, основанные на требованиях, и техники построения тестов, использующие только информацию, заданную в требованиях к проверяемой системе. Частными случаями этого вида тестирования являются тестирование на соответствие стандартам и квалификационное или сертификационное тестирование, нацеленное на получение некоторого сертификата соответствия определенным требованиям или стандартам.
  - Тестирование белого ящика (white-box testing, glass-box testing, также структурное тестирование, structural testing) [201] — нацелено на проверку корректности работы кода. Использует критерии полноты и техники построения тестов, основанные на структуре проверяемой системы, ее исходного кода.
  - Тестирование серого ящика (grey-box testing) использует для построения тестов как информацию о требованиях, так и коде. На практике оно встречается чаще, чем предыдущие крайние случаи.
  - Тестирование, нацеленное на ошибки — использует для построения тестов гипотезы о возможных или типичных ошибках в ПО такого же типа, как проверяемое. К этому типу относятся, например, следующие виды тестирования.
    - ♣ Тестирование работоспособности (sanity testing, smoke testing), нацеленное на проверку того, в систему включены все ее компоненты и операции, и система не дает сбоев при выполнении своих основных функций в простейших сценариях использования.
    - ♣ Тестирование на отказ, пытающееся найти ошибки в ПО, связанные с контролем корректности входных данных.

- ♣ Нагрузочное тестирование (load testing), проверяющее работоспособность ПО при больших нагрузках — больших объемах входных, выходных или промежуточных данных, большой сложности решаемых задач, большом количестве пользователей, работающих с ПО и пр.
- ♣ Тестирование в предельных режимах (stress testing), проверяющее работоспособность ПО на границах его возможностей и на границах той области, где оно должно использоваться.

- По роли команды, выполняющей тестирование, оно может относиться к следующим видам.
  - o Внутреннее тестирование выполняется в рамках проекта по разработке системы силами организации-разработчика ПО.
  - o Независимое тестирование выполняется третьими лицами (не разработчиками, не заказчиками и не пользователями) для получения объективных и аккуратных оценок качества системы.
  - o Аттестационное тестирование (приемочные испытания) выполняется представителями заказчика непосредственно перед приемкой системы в эксплуатацию для проверки того, что основные функции системы реализованы. Обычно аттестационные тесты являются очень простыми тестами функциональности и производительности.
  - o Пользовательское тестирование осуществляется силами пользователей системы. У него есть два часто упоминаемых частных случая.
    - ♣ Альфа-тестирование (alpha-testing) выполняется самими разработчиками, но в среде, максимально приближенной к рабочему окружению системы и на наиболее вероятных сценариях ее реального использования.
    - ♣ Бета-тестирование (beta-testing) выполняется пользователями, желающими познакомиться с возможностями системы до ее официального выпуска и передачи в эксплуатацию.

16. Методы верификации. Тестирование. Техники построения тестов

Тест представляет собой процедуру, обеспечивающую создание некоторой специфической ситуации с точки зрения поведения тестируемой системы и проверку правильности работы системы в этой ситуации. Соответственно, при построении теста, нужно выбрать нужную ситуацию, определить способ ее создания и определить процедуру проверки правильности работы тестируемой системы.

Тестовая ситуация обычно включает в себя следующие элементы:

1) Оказываемое на систему воздействие с некоторыми данными, результаты которого надо проверить в первую очередь. Это может быть вызов операции с определенными аргументами, посылка сообщения определенного содержания, нажатие на кнопку диалога после того, как другие поля этого диалога заполнены некоторыми данными, выполнение команды в командной строке с некоторыми опциями и аргументами и т.п.
2) Внутреннее состояние тестируемой системы. Поскольку состояние чаще всего невозможно установить напрямую, для его достижения нужно использовать другие воздействия на систему. При этом их результаты тоже необходимо проверять, так как само состояние чаще всего недоступно для прямого наблюдения. В результате достижение определенной ситуации чаще всего требует использования тестовой последовательности (test sequence) — последовательности обращений к системе с определенными данными.
3) Иногда на работу систему влияют внешние условия, воспринимаемые системой, помимо оказываемых на нее воздействий. Это могут быть измеряемые автоматически физические показатели окружающей среды или результаты взаимодействий с другими системами. Часто такие условия можно промоделировать программно, задавая определенную конфигурацию системы, имитируя деятельность других систем или заменив датчики физических показателей управляемыми модулями. В тех случаях, когда это сделать нельзя, нужно создавать модель рабочего окружения системы, позволяющую изменять внешние условия при тестировании.

Для выбора (или построения) тестовых ситуаций и составления тестового набора используют техники следующих типов:

1) Вероятностное тестирование (random testing). При использовании этого подхода каждая возможная тестовая ситуация описывается некоторым набором параметров, все значения которых генерируются как псевдослучайные данные с определенными распределениями. Проще всего таким способом получать тестовые наборы для систем без внутреннего состояния, при этом достаточно задать распределения вероятностей значений различных параметров. Для систем с состоянием используются описания распределения вероятностей возможных сценариев работы с ними в виде марковских цепей.

   Исходной информацией для задания распределений значений параметров является частота их использования при реальной работе системы, величина риска, связанного с ошибками в системе в соответствующей ситуации. Чаще всего, однако, используется вероятностное тестирование с равномерными распределениями, просто потому, что оно является наиболее дешевым способом получения большого количества тестов.
2) Тестирование на основе классов эквивалентности (partition testing).Для выбора тестов по этой технике все возможные ситуации разбиваются на конечное

множество классов эквивалентности. Обычно это делается так, чтобы различия в поведении тестируемого ПО в эквивалентных ситуациях были несущественны, а в неэквивалентных — достаточно велики. Часто за основу разбиения выбирается используемый критерий покрытия — ситуации, которые соответствуют одному покрываемому элементу в рамках этого критерия, считают эквивалентными. Далее тесты строятся так, чтобы в каждом классе эквивалентности был хотя бы один тест.

3) Комбинаторное тестирование. В этом случае произвольная тестовая ситуация также описывается набором параметров, каждый из которых может принимать только конечное множество значений. Ситуации для тестирования выбираются таким образом, чтобы реализовать определенные комбинации значений параметров, например, это могут быть вообще все комбинации, или комбинации всех пар значений, или два параметра с одинаковыми множествами значений должны принимать равные значения, а остальные — произвольные, и т.п. Для получения конечного множества значений параметра обычно используют выделение классов эквивалентности этих значений.
4) Сценарное тестирование (scenario-based testing).Тесты строятся на основе сценариев использования системы, сценариев ее взаимодействия с другими системами или сценариев взаимодействия ее компонентов друг с другом. Такие сценарии классифицируются, и для каждого выделенного типа сценариев создается тест, повторяющий общую структуру таких сценариев.
5) Тестирование, нацеленное на определенные ошибки. (fault-based testing, riskbased testing). С помощью таких техник строят тесты, прямо нацеленные на обнаружение ошибок некоторого типа. Они достаточно часто используются на практике, поскольку позволяют существенно снижать риски проекта. Возможность автоматизации определяется видом ошибок, на которые нацеливаются тесты. Примером такой техники является тестирование граничных значений (boundary testing), в рамках которых в качестве данных для тестов используются значения, расположенные на границах областей, в которых тестируемая операция ведет себя по-разному.
6) В рамках техник автоматного тестирования тесты строятся как пути на графе переходов автоматной модели, описывающей требования к поведению или проект тестируемого ПО.
7) Смешанные техники. В большинстве случаев на практике используют комбинацию из техник нескольких типов, поскольку ни один из них не покрывает полностью область применения и достоинства ни одного другого. Кроме этого, часто применяется адаптивное тестирование (adaptive testing), которое опирается на получаемую в ходе выполнения тестов информацию для выявления наиболее проблемных частей системы и построения новых тестов, более прямо нацеленных на вероятные ошибки. Одной из техник такого рода является исследовательское тестирование (exploratory testing) . Оно дополнительно делает упор на получение человеком более полной информации о свойствах ПО в процессе выполнения тестов и его способности отмечать странности в работе системы, которые можно использовать для эффективного выявления более существенных ошибок. Примером комбинации техник вероятностного тестирования и нацеливания на некоторые классы ситуаций является техника генерации структурных тестов на

основе генетических алгоритмов. В ее рамках исходный набор тестов генерируется случайно, после чего запускается генетический алгоритм, отбирающий такие тесты, который обеспечивает максимальное покрытие реализации тестируемых операций. Тест считается тем «лучше» с точки зрения дальнейшего отбора, чем ближе его выполнение подходит к еще не покрытым элементам кода.

Помимо выбора набора ситуаций для тестирования и определения способа их достижения, необходимо определить процедуру проверки получаемых при тестировании результатов на корректность. Такая процедура называется тестовым оракулом (test oracle). **Существуют следующие способы организации оракулов:**

1) При неавтоматизированном тестировании роль оракула играет человек, выполняющий тесты. Именно он оценивает, насколько получаемые результаты соответствуют требованиям, проектным решениям, задачам проекта, нуждам пользователей.
2) Очень часто при автоматизированном тестировании проверяется отсутствие сбоев и исключительных ситуаций. Это частичный оракул, который выражает только малую часть требований к ПО.
3) При автоматизации выполнения тестов иногда используют оракул, организованный в виде проверки равенства полученных в тесте результатов ожидаемым, вычисленным заранее.
4) Иногда тестовый оракул сравнивает полученные в тесте результаты с вычисляемыми другой реализацией тех же функций. В качестве другой реализации могут использоваться другие системы с похожей функциональностью, более ранние версии тестируемой системы, прототипы и исполнимые модели, созданные на более ранних этапах проекта.
5) Если проверяемая функция имеет легко вычисляемую обратную, ее можно использовать для построения оракула — вычислить обратную функцию от результатов и сравнить полученные данные с входными данными проверяемой функции.
6) Часто известны некоторые свойства результатов, которые заведомо должны быть выполнены, свойства значений переменных проверяемой программы в определенных точках кода или инварианты внутренних данных тестируемого компонента. Оракул может проверять эти свойства и инварианты в ходе тестирования.
7) Наиболее полно проверить корректность поведения тестируемой системы можно с помощью оракула, получаемого из формальной модели требований и проектных решений. Такие оракулы используются в тестировании на основе моделей.

17. Методы верификации. Тестирование. Тестовые примеры

Непосредственно для тестирования кроме тестового окружения необходимо определить проверочные задачи, которые будет выполнять система или ее часть. Такие проверочные задачи называют тестовыми примерами.
Как уже было сказано выше, каждый тестовый пример состоит из входных значений для системы, описания сценария работы примера и ожидаемых выходных значений. Целью выполнения любого тестового примера является либо продемонстрировать наличие в системе дефекта, либо доказать его отсутствие.
Начальный этап работы тестировщика заключается в формировании тест-требований, соответствующих функциональным требованиям. Основная цель тест-требований – определить, какая функциональность системы должна быть протестирована. В самом простом случае одному функциональному требованию соответствует одно тесттребование. Однако чаще всего тест- требования детализируют формулировки функциональных требований.

Типы тестовых примеров

- **Допустимые данные**

Чаще всего дефекты в программных системах проявляются при обработке нестандартных данных, не предусмотренных требованиями – при вводе неверных символов, пустых строк, слишком большой скорости ввода информации. Однако, перед поиском таких дефектов необходимо удостовериться в том, что программа корректно обрабатывает верные данные, предусмотренные спецификацией, т.е. проверить работу основных алгоритмов.

- **Граничные данные**

Отдельный вид допустимых данных, передача которых в систему может вскрыть дефект – граничные данные, т.е. например, числа, значения которых являются предельными для их типа, строки предельной или нулевой длины и т.п. Обычно при помощи тестирования граничных условий выявляются проблемы с арифметическим сравнением чисел или с итераторами циклов.

- **Отсутствие данных**

Дефекты могут проявиться и в случае, если системе не передается никаких данных или передаются данные нулевого размера.

- **Повторный ввод данных**

В случае повторной передачи на вход системы тех же самых данных могут получаться различия в выходных данных, не предусмотренные в требованиях. Как правило, дефекты такого типа проявляются в результате того, что система не устанавливает внутренние переменные в исходное состояние или в результате ошибок округления.

- **Неверные данные**

При проверке поведения системы необходимо не забывать проверять ее поведение при передаче ей данных, не предусмотренных требованиями – слишком длинных или слишком коротких строк, неверных символов, чисел за пределами вычислимого диапазона и т.п. Неверные данные, как и допустимые, также можно разделять на различные классы эквивалентности.

- **Реинициализация системы**

Механизмы повторной инициализации системы во время ее работы также могут содержать дефекты. В первую очередь эти дефекты могут проявляться в том, что не все внутренние данные системы после реинициализации придут в начальное состояние. В

результате может произойти сбой в работе системы.

- **Устойчивость системы**

Под устойчивостью системы можно понимать ее способность выдерживать нештатную нагрузку, явно не предусмотренную требованиями. Например, сохранит ли система работоспособность после 10 тысяч вызовов.

- **Нештатные состояния среды выполнения**

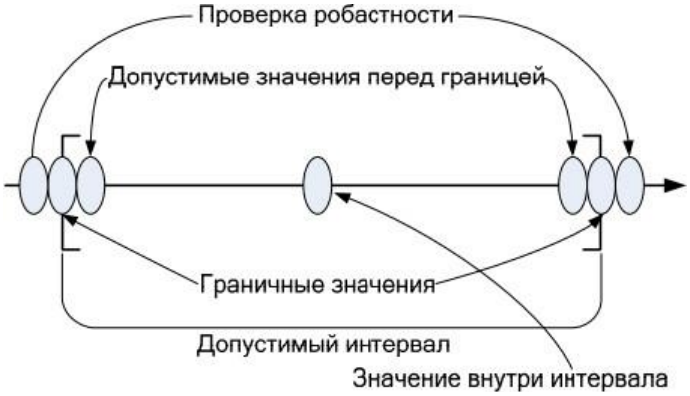
Нештатные состояния среды выполнения (например, исчерпание памяти, дискового пространства или длительная нехватка процессорного времени) могут затруднять работу системы, либо делать ее невозможной. Основная задача системы в такой ситуации – корректно завершить или приостановить свою работу.

Граничные условия

В тестовых примерах, прямо соответствующих тест-требованиям обычно используются входные значения, находящиеся заведомо внутри допустимого диапазона. Один из способов проверки устойчивости системы на значениях, близких к предельным – создавать для каждого входа как минимум три тестовых примера:
• Значение внутри диапазона
• Минимальное значение
• Максимальное значение
Для еще большей уверенности в работоспособности системы используют пять тестовых примеров:
• Значение внутри диапазона
• Минимальное значение
• Минимальное значение + 1
• Максимальное значение
• Максимальное значение – 1

Проверка робастности (выхода за границы диапазона)

Для тестирования робастности к тестовым примерам, рассмотренным в предыдущем разделе добавляются еще два тестовых примера:
• Минимальное значение - 1
• Максимальное значение + 1



Классы эквивалентности

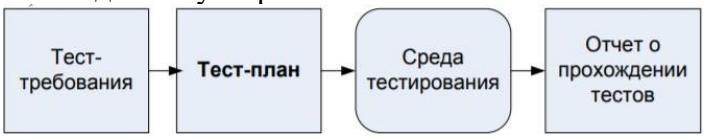
Рассмотренные выше граничные условия могут служить примером классов эквивалентности:
1. Значение из середины интервала.
2. Граничные значения.
3. Недопустимые значения за границами интервала.
Таким образом, тестирование граничных условий и робастности является частным случаем тестирования с использованием классов эквивалентности – вместо того, чтобы

тестировать все недопустимые значения, выбираются только соседние с граничными.
При определении классов эквивалентности следует руководствоваться следующими правилами:
• Всегда будет, по меньшей мере, два класса: корректный и некорректный
• Если входное условие определяет диапазон значений, то, как правило бывает три класса: меньше чем диапазон, внутри диапазона и больше чем диапазон. (Значения на концах диапазона могут трактоваться как граничные значения.)
• Если элементы диапазона обрабатываются по-разному, то каждому варианту обработки будут соответствовать разные требования.

18. Методы верификации. Тестирование. Тестпланы

Технологические цепочки и роли участников проекта, использующих тест-планы. Связь тест-планов с другими типами проектной документации.

На основании тест-требований составляются тест-планы - программы испытаний (проверки, тестирования) программной реализации системы. В отличие от тесттребований в тест-плане описываются конкретные способы проверки функциональности системы, т.е. то, как должна проверяться функциональность. Как правило, тест-план состоит из отдельных тестовых примеров, каждый из которых проверяет некоторую функцию или набор функций системы. Для каждого тестового примера однозначно определяется критерий успешного прохождения (pass/fail criteria), при помощи которого можно судить о том – соответствует ли поведение системы заданному в требованиях или нет.



Критерием качества тест-плана является покрытие (выполнение) всех требований к проверке правильности функционирования программной реализации. Желательной характеристикой тест-плана является проверка исполнения всех веток схемы программной реализации.
Структура тест-плана может соответствовать структуре тест-требований или следовать логике внешнего поведения системы. Каждый пункт тест-плана описывает, как производится проверка правильности функционирования программной реализации, и содержит:

- ссылку на требование(я), которое проверяется этим пунктом;
- конкретное входное воздействие на программу (значения входных данных);
- ожидаемую реакцию программы (тексты сообщений, значения результатов)
- описание последовательности действий, необходимых для выполнения пунктов тест-плана.

В состав тест-плана рекомендуется дополнительно включать пункты, служащие для проверки ветвей программы, не выполнявшихся при проверке удовлетворения функциональных требований. Такие пункты тест-плана могут иметь указание “Для полноты покрытия” в поле ссылки.
Тест-план может готовиться в формализованной форме и служить входным документом для тестовой оснастки, по которому тесты будут выполняться в автоматическом режиме с автоматической фиксацией результатов. В случае, если тестплан готовится в виде текстового документа, возможно только ручное тестирование системы по данному тест-плану.

Возможные формы подготовки тест-планов

Форма представления тест-плана в первую очередь зависит от того, каким образом тест-план будет использоваться в процессе тестирования. При ручном тестировании удобно представление тест-планов в виде текстовых документов, в которых отдельные разделы представляют собой описания тестовых примеров. Каждый тестовый пример в таком случае включает в себя перечисление последовательности действий, которые необходимо выполнить тестировщику для проведения тестирования – сценария теста, а также ожидаемые отклики системы на эти действия. Такая форма представления тестплана неудобна для автоматизации тестирования, поскольку описания на естественном языке практически не поддаются формализации.
Для автоматизированного тестирования сценарий теста может записываться на каком-либо формальном языке, в этом случае возможно непосредственное использованиетест-планов как входных данных для среды тестирования.
Другой формой представления тест-планов является таблица. Эта форма наиболеечасто используется при четко и формально определенных входных потоках данных системы.

Например, каждый столбец таблицы может представлять собой тестовый пример, каждая строка – описание входного потока данных, а в ячейке таблицы записывается передаваемое в данном тестовом примере в данный поток значение. Ожидаемые значения для данного теста записываются в аналогичной таблице, в которой в строках перечисляются выходные потоки данных.
И, наконец, третьей формой представления тестовых примеров является определение примеров в виде конечного автомата. Такая форма представления используется при тестировании протоколов связи или при тестировании программных модулей, взаимодействие которых с внешним миром производится при помощи обмена сообщениями по заранее заданному интерфейсу. Модуль при этом может быть представлен как конечный автомат с набором состояний, а тест-план будет состоять из двух частей – описания переходов между состояниями и их параметров и тестовых примеров, в которых задается маршрут перехода между состояниями, параметры переходов и ожидаемые значения. Такое представление тест-плана может быть пригодно как для ручного, так и для автоматизированного тестирования.

Сценарии

Представление сценариев, удобное для ручного тестирования – тест-план в виде текстового документа, в котором каждый тестовый пример представляет один раздел. Для каждого тестового примера в этот документ записывается следующая информация:

- идентификатор;
- описание теста и его цель;
- ссылки на тестируемую часть системы;
- ссылки на используемую проектную документацию, в частности тест-требования;
- перечисление действий сценария;
- ожидаемая реакция системы на каждый пункт сценария.

Подразумевается, что действия сценария должны быть описаны таким образом, чтобы их мог воспроизвести человек с практически любым уровнем подготовки. Описание ожидаемой реакции системы должно также быть записано таким образом, чтобы можно было однозначно судить о том – соответствует реакция ожидаемой или нет. Так, неудачной ожидаемой реакцией при ручном тестировании была бы запись

Сообщение «Загрузка» пропадает через приемлемое время

Степень приемлемости здесь будет зависеть от терпеливости тестировщика, и обеспечить повторяемость тестирования будет затруднительно. Более удачной формой описания той же самой ожидаемой реакции будет

Сообщение «Загрузка» исчезает с экрана не более, чем через 10 секунд после появления.

Ниже приведен пример описания тестового примера в виде сценария, предназначенного для ручного тестирования:

№	Шаг сценария	Ожидаемый результат
1	Запустить терминальный клиент и соединиться с системой по адресу 127.0.0.1	Должно появиться приглашение терминала TRANSFER>
2	Запустить процесс передачи данных при помощи ввода команды SEND DATA	Должно появиться приглашение DATA TRANSFER INITIATED и следующими двумя строками Enter your credentials… Login:
3	Ввести имя учетной записи default	Должна появиться строка Password:
4	Ввести пароль default	Должно появиться сообщение Default user blocked – system set to High security и соединение с терминалом должно быть прервано

Как можно видеть, такая форма представления действительно неудобна для автоматизации тестирования и предназначена исключительно для ручного тестирования. Иногда такие тест-планы совмещают с отчетами о проведении тестирования, добавляя в таблицу описания сценария третью и четвертые колонки – «Реальный результат» и

«Соответствует», в который заносятся реальная реакция системы и указание на совпадение/несовпадение результатов соответственно. В конце описания каждого тестового примера добавляется графа «Пройден/не пройден», в которую заносится информация о том, пройден ли тестовый пример в целом. В конце всего тест-плана совмещенного с отчетом помещается графа «Тестовых примеров пройдено/всего», в которую заносится число пройденных тестовых примеров и общее их число.
Сценарии тестирования для автоматического тестирования часто описывают на том или ином языке программирования. Например, методы в тестирующих классах Microsoft Visual Studio Team Edition представляют собой именно пошаговые описания действий, которые необходимо выполнить тестовому окружению для проведения тестирования. При такой форме представления сценарий каждого тестового примера состоит из последовательности вызовов функций, которые передают данные в среду тестирования.

Таблицы

Как уже говорилось выше, табличное представление тестов удобно при четко формализованных входных и выходных потоках данных системы.
Табличное представление, как правило, используется для упрощения работы по подготовке и сопровождению большого количества однотипных тестов. Среда тестирования, использующая табличное описание тестовых примеров в качестве входных данных включает в себя интерпретатор таблиц, преобразующих это описание в последовательность команд, выполняемых средой для проведения тестирования, т.е. своего рода сценарий.
В случае, когда однотипными являются не только входные и выходные данные, но и их значения, может использоваться альтернативная форма представления табличных данных. Тестовые примеры в ней также нумеруются по горизонтали, а входные потоки данных – по вертикали. Однако, под каждым из потоков данных перечисляются возможные входные значения, а факт того, что это входное значение должно быть передано в данном тестовом примере, отмечается помещением специальной метки (например, символа X) на пересечении значения и тестового примера в таблице:

INPUTS:	a	b	c	d	e	f
Power_On_Mode						
COLD	X	X				X
WARM			X	X	X	
Configuration_Store_Id						
0xFFFD	X	X	X	X	X	X
IR_Access_Mode						
1	X	X	X			X
0				X		
0xFFFF					X	
Reset_Mode						
0	X	X	X	X	X	X
Reset_Source						
0		X				X
1	X					
2			X	X	X	

Конечные автоматы

Форма подготововки тест-планов в виде описания конечных автоматов удобна при тестировании программных модулей или систем, поведение которых также может быть описано в виде конечного автомата. В этом случае процесс тестирования представляет собой обмен сообщениями между двумя конечными автоматами, изменяющими свое состояние в процессе обмена. Критерием полноты такого тестирования будет достижимость всех состояний тестируемой системы всеми возможными способами.
Описание тест-планов в виде конечного автомата обычно состоит из двух частей – определения самого тестирующего конечного автомата и определения сценариев перехода между состояниями – тестовых примеров.

Генераторы тестов

В некоторых случаях для упрощения процедуры тестирования используются специальные инструментальные средства, автоматически генерирующие тестовые примеры. Эти системы различаются по используемым методам генерации тестовых примеров, а

получаемые тестовые примеры различаются по областям применимости.

Различают следующие способы генерации тестовых примеров:

- по формализованным требованиям;
- случайным образом;
- по программному коду.

Первый способ генерации тестовых примеров приемлем для тестирования системы как «черного ящика», но требует чтобы тест-требования (или системные/функциональные требования) были подготовлены на специальном формальном языке оформления требований, например RDL (Requirements Definition Language). Затем по требованиям строятся тестовые примеры, которые проверяют функциональность системы с точки зрения требований, т.е. в этом случае достигается основная цель верификации – проверить, ведет ли себя система в соответствии с требованиями.

К сожалению, этот путь достаточно трудоемок и экономия времени от автоматической генерации тестов зачастую сводится на нет необходимостью в выделении дополнительного времени на перевод всех требований в формальную форму. В связи с этим рекомендуется применять данный метод только для тестирования систем, требования на которые могут быть сравнительно легко формализованы с использованием того или иного языка, например, системы поддержки коммуникационных протоколов.

Второй метод генерации тестовых примеров – на основе случайных данных. В этом случае не может идти и речи о систематизированном тестировании и гарантиях качества системы. Такой подход может применяться только в случае необходимости проверить поведение системы в случае передачи в нее большого количества неверных данных или определить количественные параметры поведения системы под большой нагрузкой.

Третий метод тестирования основан на анализе исходных текстов системы и построения тестов, которые выполняют каждое логическое условие и каждый оператор системы. В результате достигается очень высокий уровень покрытия программного кода. Однако, в этом случае тесты проверяют не то, что система должна делать в соответствии с требованиями, а то, как она делает то, что уже запрограммировано. Перед тестировщиком в этом случае стоит задача анализа программного кода системы на соответствие требованиям, что зачастую представляет собой задачу не менее сложную, чем ручное написание тестов для проверки требований. Обычно рекомендуется вначале написать все тесты по требованиям, а затем, в случае необходимости, воспользоваться генератором тестов по программному коду. При этом целью использования генератора будет не достижение максимально возможного покрытия любой ценой, а анализ причин непокрытия при выполнении тестов требований, и, в случае необходимости, коррекции требований.

19. Статический анализ исходных текстов ПО. Использование в жизненном цикле разработки программ. Методы статического анализа.

Статические анализаторы осуществляют поиск ошибок в программе без её фактического запуска. При этом, как правило, сразу анализируется множество путей исполнения. Благодаря тому, что статические анализаторы просматривают сразу все пути исполнения и анализируют пути независимо от вероятности их выполнения, анализаторы находят многие ошибки на редко исполняемых путях, которые часто остаются незамеченными во время тестирования. Другим преимуществом статических анализаторов является диагностика места ошибки. При выдаче предупреждения об ошибке статические анализаторы сразу показывают место ошибки и описывают, почему именно это является ошибкой.

Примеры дефектов, которые может обнаружить статический анализатор: ошибки разыменования нулевого указателя, переполнение буфера, использование неинициализированных переменных, утечки памяти, двойные блокировки, наличие недостижимого кода, несогласованность конструкторов и деструкторов, ошибки деления на ноль, возвращение адреса локальных переменных, использование объектов после удаления.

Использование в жизненном цикле разработки программ.

Использовании статического анализатора в жизненном цикле разработки программ не имеет задачи найти все возможные дефекты в программе. Анализ проводится регулярно во время ночной сборки программы, поэтому время анализа ограничено примерно 12 часами. Количество ложных срабатываний должно быть достаточно низким (каждое ложное срабатывание отнимает время программиста и фактически выражается в убытках для компании). Для удовлетворения этих требований анализатор может в случае, когда нет достаточных оснований, что предупреждение будет истинным, просто не выдавать предупреждение. Это позволяет существенно уменьшить количество выдаваемых ложных срабатываний и в ряде случаев сократить время анализа. Такой анализатор может пропустить реальную ошибку, но, т.к. большинство выдаваемых предупреждений будут истинными, то увеличивается шанс, что каждый реально найденный дефект будет исправлен, и уменьшается стоимость каждого исправленного дефекта (стоимость будет пропорциональна количеству ложных срабатываний, которые необходимо просмотреть на каждое истинное срабатывание).

Долю истинных срабатываний в 60-70% можно считать приемлемой. При таком соотношении истинных и ложных срабатываний на одно истинное срабатывание приходится не более одного ложного.

Методы статического анализа.

По типу используемых абстракций методы поиска ошибок можно разделить на следующие группы:

- **Лексические анализаторы**, рассматривающие программу как поток токенов. С их помощью можно найти только самые простейшие виды дефектов;
- **Легковесные анализаторы (анализаторы 1го уровня)**, осуществляющие анализ преимущественно с помощью просмотра абстрактного синтаксического дерева (АСД), а также с использованием других абстракций уровня синтаксического анализа.;
- **Более сложные анализаторы (анализаторы 2го уровня)**, которые используют абстракции, связанные с фазами после синтаксического анализа.

В результате работы синтаксического анализатора компилятором строится абстрактное

синтаксического дерево, которое позже передаётся на следующие фазы компиляции программы. Анализаторы, построенные на базе АСД, осуществляют проход по узлам АСД и делают относительно простые проверки анализируемых правил.
В группе легковесных анализаторов можно выделить более сложные анализы, использующие некоторую модель памяти. Такие анализаторы позволяют реализовать довольно сложные детекторы и найти ошибки, которые невозможно найти простым проходом по АСД.
Методы анализа 2го уровня также можно разделить по тому какие свойства они учитывают при анализе: чувствительность к потоку, межпроцедурность, чувствительность к контексту и др.
Методы анализа 2го уровня:

- **Потоково-нечувствительные** анализы рассматривают программу как неотсортированный набор инструкций;
- **Потоковочувствительный** анализ учитывает порядок инструкций. Виды:
    - **Анализ с чувствительностью к путям**. Учитывает по какому пути прошло выполнение программы;
    - **Анализ без чувствительности к путям**;
- **Межпроцедурный анализ**. Обнаруживает дефекты, являющиеся результатом неправильного использования нескольких функций. Виды:
    - **Контекстно-чувствительный** анализ отличает эффекты вызова функции в зависимости от контекста её вызова.

Учёт какого-либо свойства требует большего количества памяти и времени работы анализа, но зато даёт лучшую точность. Недостаточная точность анализа приводит либо к пропуску ошибок, либо к появлению ложных срабатываний. Более сложный анализ учитывает больше свойств. При этом анализы могут иметь разную степень чувствительности к какому-либо свойству. Например, при чувствительности к контексту вызова можно учитывать значения не всех переменных или анализировать разную высоту стека вызовов.
Критичность найденных ошибок, как правило, не зависит от сложности анализатора. Многие критичные ошибки являются результатом опечатки и часто могут быть найдены с помощью анализаторов 1го уровня. Более простые правила желательно реализовывать с помощью легковесных анализаторов на основе АСД. В этом случае скорость написания правила и скорость анализа будет лучше. Кроме того, часть информации, которая доступна в АСД, может быть недоступной на более поздних фазах анализа. Например, наличие или отсутствие фигурных скобок можно проверить только на этапе синтаксического анализа, т.к. далее эта информация не сохраняется.
Любой статический анализатор находит приближённое решение (из-за проблемы неразрешимости задачи поиска дефектов). Анализ является **консервативным**, если приближённое решение гарантированно включает все возможные варианты при выполнении программы. Консервативность гарантирует, что все полученные выводы являются корректными. Это очень важно для оптимизирующих компиляторов, которые не должны изменять семантику программы. Если нет достаточно уверенности, что оптимизация безопасна, то лучше вообще её не применять, чем изменить поведение программы. При поиске же ошибок, иногда желательно выдать предупреждение об ошибке, которое может оказаться ложным, либо основано на некоторых эвристиках. Использование **неконсервативного** анализа позволяет создать более простой и быстрый анализатор, и в тоже время находить довольно сложные типы ошибок. Расплатой за это является недостаточно точная модель программы, что в некоторых случаях может приводить к непонятным ложным срабатываниям.

20. Статический анализ исходных текстов ПО. Инструментальные средства проведения. Примеры, особенности. [PVS-Studio, flawfinder, rats, cppcheck]

Статический анализ кода это процесс выявления ошибок и недочетов в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода. Остановимся на обзоре кода чуть подробнее.

Задачи, решаемые программами статического анализа кода можно разделить на 3 категории:

- Выявление ошибок в программах. Подробнее про это будет рассказано ниже.
- Рекомендации по оформлению кода. Некоторые статические анализаторы позволяют проверять, соответствует ли исходный код, принятому в компании стандарту оформления кода. Имеется в виду контроль количества отступов в различных конструкциях, использование пробелов/символов табуляции и так далее.
- Подсчет метрик. Метрика программного обеспечения - это мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций. Существует большое количество разнообразных метрик, которые можно подсчитать, используя те ли иные инструменты.

Другие преимущества статического анализа кода:

- Полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получают управление крайне редко. Такие участки кода, как правило, не удается протестировать другими методами. Это позволяет находить дефекты в обработчиках редких ситуаций, в обработчиках ошибок или в системе логирования.
- Статический анализ не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа. Это позволяет находить скрытые ошибки, которые могут проявить себя только через несколько лет. Например, это ошибки неопределенного поведения. Такие ошибки могут проявить себя при смене версии компилятора или при использовании других ключей для оптимизации кода. Другой интересный пример скрытых ошибок приводится в статье "Перезаписывать память - зачем?".
- Можно легко и быстро обнаруживать опечатки и последствия использования Copy-Paste. Как правило, нахождение этих ошибок другими способами является кране неэффективной тратой времени и усилий. Обидно после часа отладки обнаружить, что ошибка заключается в выражении вида "strcmp(A, A)". Обсуждая типовые ошибки, про такие ляпы, как правило, не вспоминают. Но на практике на их выявление тратится существенное время.

Недостатки статического анализа кода

- Статический анализ, как правило, слаб в диагностике утечек памяти и параллельных ошибок. Чтобы выявлять подобные ошибки, фактически необходимо виртуально выполнить часть программы. Это крайне сложно реализовать. Также подобные алгоритмы требуют очень много памяти и процессорного времени. Как правило, статические анализаторы ограничиваются диагностикой простых случаев.

Более эффективным способом выявления утечек памяти и параллельных ошибок является использование инструментов динамического анализа.

- Программа статического анализа предупреждает о подозрительных местах. Это значит, что на самом деле код, может быть совершенно корректен. Это называется ложно-позитивными срабатываниями. Понять, указывает анализатор на ошибку или выдал ложное срабатывание, может только программист. Необходимость просматривать ложные срабатывания отнимает рабочее время и ослабляет внимание к тем участкам кода, где в действительности содержатся ошибки.

PVS-Studio

PVS-Studio - это инструмент для выявления ошибок и потенциальных уязвимостей в исходном коде программ, написанных на языках C, C++, C# и Java. Работает в среде Windows, Linux и macOS.

PVS-Studio выполняет статический анализ кода и генерирует отчёт, помогающий программисту находить и устранять ошибки. PVS-Studio выполняет широкий спектр проверок кода, но наиболее силён в поисках опечаток и последствий неудачного Copy-Paste. Показательные примеры таких ошибок: V501, V517, V522, V523, V3001.

Особенности PVS-Studio

- Автоматический анализ отдельных файлов после их перекомпиляции.
- Удобная и простая интеграция с Visual Studio 2010-2017.
- Удобная online-справка по всем диагностикам, которая доступна и из программы, и на сайте, а также документация в .pdf одним файлом.
- Запуск из командной строки для проверки всего решения: позволяет интегрировать PVS-Studio в ночные сборки.
- Интерактивная фильтрация результатов анализа (лога) в окне PVS-Studio: по коду диагностики, по имени файла, по включению слова в текст диагностики.
- Возможность безболезненно внедрить PVS-Studio в существующий процесс разработки и сфокусироваться на ошибках только в новом коде.
- Статистика ошибок в Excel – можно посмотреть темпы правки ошибок, количество ошибок во времени и т.п.
- Использование относительных путей в файлах отчета для возможности переноса отчета на другую машину.
- Возможность исключить из анализа файлы по имени, папке или маске;
- Возможность проверять файлы, модифицированные за последние N дней.

Для поиска ошибок переполнения буфера и ошибок форматных строк используют следующие статические анализаторы:

1. **RATS**. Утилита RATS (Rough Auditing Tool for Security) обрабатывает код, написанный на Си/Си++, а также может обработать еще и скрипты на Perl, PHP и Python. RATS просматривает исходный текст, находя потенциально опасные обращения к функциям. Цель этого инструмента - не окончательно найти ошибки, а обеспечить обоснованные выводы, опираясь на которые специалист сможет вручную выполнять проверку кода. RATS использует сочетание проверок надежности защиты от семантических проверок в ITS4 до глубокого семантического анализа в поисках дефектов, способных привести к переполнению буфера, полученных из MOPS.

2. **Flawfinder**. Как и RATS, это статический сканер исходных текстов программ, написанных на Си/Си++. Выполняет поиск функций, которые чаще всего используются некорректно, присваивает им коэффициенты риска (опираясь на такую информацию, как передаваемые параметры) и составляет список потенциально уязвимых мест, упорядочивая их по степени риска.

Cppcheck — статический анализатор кода для языка C/C++

Анализатор способен проверять нестандартные участки кода, включающие использование расширений компилятора, встраиваемый ассемблер и т. п.

Возможности

Обнаруживает различные типы ошибок в программах.

- Проверяет выход за пределы.
- Обнаруживает утечки памяти.
- Обнаруживает возможное разыменовывание NULL-указателей.
- Обнаруживает неинициализированные переменные.
- Обнаруживает неправильное использование STL.
- Проверяет обработку исключительных ситуаций на безопасность.
- Находит устаревшие и неиспользуемые функции.
- Предупреждает о неиспользуемом или бесполезном коде.
- Находит подозрительные участки кода, которые могут содержать в себе ошибки.