**Лекция 14 Тестирование и отладка программных систем.** Стратегии и методы тестирования и верификации. Динамический анализ кода программы. Программные средства автоматизации тестирования и верификации. Тестирования на проникновение

В настоящее время реальные программные продукты чаще всего разрабатываются в сжатые сроки и ограниченном бюджете. К сожалению, при таких условиях разработчики программного обеспечения часто игнорируют необходимость контроля и поддержки надлежащего качества разрабатываемого ими продукта, подвергая тем самым конечных пользователей неоправданному риску. Основным аспектом, доказывающим необходимость применения тестирования совместно с процессом разработки программного обеспечения (ПО), является минимизация затрат как для разработчика, так и для потребителя продукта.

Любой заказчик хочет получить надежное программное изделие, которое полностью удовлетворяет его потребности. Различные уровни надежности обеспечиваются разными инженерными подходами к тестированию.

Уровень надежности программных изделий определяется инженерией тестирования.

Существует несколько признаков, по которым принято производить классификацию видов тестирования:

* По объекту тестирования
* По знанию системы
* По степени автоматизации
* По степени изолированности компонентов
* По времени проведения тестирования
* По признаку позитивности сценариев
* По степени подготовленности к тестированию
* Уровни тестирования

**Тестирование программного кода.**

**Тестирование** – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением в специально заданных, искусственных условиях.

*Ожидаемое поведение программы*. Исходной информацией для тестирования является *знание* о том, как система должна себя вести, то есть требования к ней или к ее отдельной части. Самым распространенным способом тестирования является тестирование методом **черного ящика**, то есть когда реализация системы недоступна *тестировщикам*, а тестируется только ее *интерфейс*. Существует тестирование методом **белого ящика**, когда *код программ* доступен тестировщикам и используется в качестве источника информации о системе. Его схема представлена на следующем рисунке:

*Специально заданные, искусственные условия*, – те условия, где осуществляется тестирование. При этом ключевым аспектом здесь является наличие **тестов** – воспроизводимых шагов манипуляции с системой, приводящих к ее некорректной работе. Концепция теста очень важна, так как необходимо не просто обнаружить некорректное поведение системы, а создать и зафиксировать *алгоритм* воспроизведения ошибки – чтобы повторить его для разработчика или чтобы разработчик сам смог воспроизвести ошибку. Если ошибка не воспроизводится, то нет возможности ее исправить.

Тесты могут быть "ручными" и автоматизированными. "Ручной" тест – это последовательность действий тестировщика, которую он (или разработчик) может воспроизвести и ошибка произойдет. Автоматический тест – это некоторая *программа*, которая воздействует на систему и проверяет то или иное ее свойство.

Трудности *автоматического тестирования*:

Во-первых, для того, чтобы тесты автоматически запускать, нужны соответствующие *программные продукты.* Их будем называть **инструментами тестирования**. В их задачу входит *запуск* теста на системе, "прогон" целого пакета тестов, а также *анализ* получившихся результатов и их обработка.

Кроме того, немаловажной задачей инструментов тестирования является обеспечение доступа теста к системе через некоторый ее *интерфейс*. *Доступ* к системе может оказаться затруднительным.

Во-вторых, часто возникает проблема ресурсов для *автоматического тестирования*. Особенно при автоматической генерации тестов: часто есть возможность автоматически сгенерировать очень большое количество тестов, так что если их еще выполнять регулярно, в режиме непрерывной интеграции, то не хватит имеющихся системных ресурсов. При этом качество тестирования может оказаться неудовлетворительным – ошибки находятся редко или вообще не находятся.

**Виды тестирования**. Не претендуя на *полноту*, выделим следующие виды тестирования.

* **Модульное тестирование** - тестируется отдельный модуль, в отрыве от остальной системы.

В ходе модульного тестирования каждый модуль тестируется как на соответствие требованиям, так и на отсутствие проблемных участков программного кода, которые могут 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.

* **Интеграционное тестирование** – два и более компонентов тестируются на совместимость. Этот вид тестирования, безусловно, должен применяться самими программистами, чтобы, как минимум, удостовериться, что все живет вместе в первом приближении. Далее тонкости интеграции могут исследовать тестировщики. Необходимо отметить, что такого рода ошибки – "ошибки на стыках" - непросто обнаруживать и устранять. Во время разработки все компоненты все вместе не готовы, интеграция откладывается, а в конце обнаруживаются трудные ошибки (в том смысле, что их устранение требует существенной работы). Здесь выходом является ранняя интеграция системы и в дальнейшем использование практики постоянной интеграции.
* **Системное тестирование** – это тестирование всей системы в целом, как правило, через ее пользовательский интерфейс. При этом тестировщики, менеджеры и разработчики акцентируются на том, как ПО выглядит и работает в целом, удобно ли оно, удовлетворяет ли она ожиданиям заказчика. При этом могут открываться различные дефекты, такие как неудобство в использовании тех или иных функций, забытые или "скудно" понятые требования.
* **Регрессионное тестирование** – тестирование системы в процессе ее разработки и сопровождение на регресс. То есть проверяется, что изменения системы не ухудшили уже существующей функциональности. Для этого создаются пакеты регрессионных тестов, которые запускаются с определенной периодичностью – например, в пакетном режиме, связанные с процедурой постоянной интеграции.
* **Нагрузочное тестирование** – тестирование системы на корректную работу с большими объемами данных. Например, проверка баз данных на корректную обработку большого (предельного) объема записей, исследование поведение серверного ПО при большом количестве клиентских соединений, эксперименты с предельным трафиком для сетевых и телекоммуникационных систем, одновременное открытие большого числа файлов, проектов и т.д.
* **Стрессовое тестирование** – тестирование системы на устойчивость к непредвиденным ситуациям. Этот вид тестирования нужен далеко не для каждой системы, так как подразумевает высокую планку качества.
* **Приемочное тестирование** – тестирование, выполняемое при приемке системы заказчиков. Более того, различные стандарты часто включают в себя наборы приемочных тестов. Например, существует большой пакет тестов, поддерживаемых компанией *Sun Microsystems*, которые обязательны для прогона для всех новых реализаций Java-машины. Считается, что только после того, как все эти тесты успешно проходят, новая реализация вправе называться Java.

**Стратегии и методы тестирования и верификации.**

Процесс тестирования должен делаться не спонтанно, а на основании конкретной стратегии, которой следует придерживаться при тестировании.

Перед разработкой стратегии тестирования, необходимо собрать как можно больше информации о требованиях к тестируемому программному обеспечению (ПО) и оценить риски. Стратегия тестирования должна учитывать множество факторов и определять, что и когда должны выполнять разработчики и тестировщики. Цель стратегии по тестированию — минимизация рисков, принимая во внимание сроки, бюджет, лимит ресурсов и прочие нюансы разработки программного обеспечения.

В настоящее время сформировались две противоположных друг другу парадигмы тестирования — функциональное (метод черного ящика) и структурное (метод белого ящика).

В методе тестирования «черного ящика» программе подаются некоторые данные на вход и проверятся результаты, в надежде найти несоответствия. При этом как именно работает программа считается несущественным. Важно отметить, что при таком подходе обязательно необходимо иметь спецификацию программы для того, чтобы было с чем сравнивать полученные в результате тестирования результаты.

Методы функционального тестирования:

1) Эквивалентное разбиение — метод, когда по спецификации выделяются классы эквивалентности (множество тестов со сходными параметрами, протестировав один из них, можно считать, что протестированы и все остальные) входных данных и создаются тесты, моделирующие попадание данных в эти классы.

2) Анализ граничных условий — метод, при котором проверяются границы классов эквивалентности. Строятся тесты для границ классов, для минимальных и максимальных значений.

3) Метод функциональных диаграмм — метод, в котором функциональная диаграмма формально является текстом, в который транслируется спецификация. В диаграмму включаются причины (входные условия) и следствия (выходные условия или преобразование системы). После чего функциональная диаграмма преобразуется в булевский граф, связывающий причины и следствия. Каждый узел графа может находиться в состоянии 0 (существует) или 1 (не существует). После чего диаграмма снабжается комментариями, которые задают ограничения на комбинации причин и следствий. Диаграмма преобразуется в таблицу решений, выбирается следствие, которое устанавливается в 1, и находятся все комбинации причин, с учетом ограничений, которые устанавливают выбранное следствие в 1.

Выделяют следующие особенности методов функционального тестирования:

‒ Тестирование системы в целом, включая отдельные модули и интерфейсы между ними.

‒ Тестирование без знания исходного кода, что позволяет ускорить процесс тестирования и сделать его более беспристрастным.

‒ Тесты основаны на спецификации и не зависят от исходного кода.

Стратегия структурного (модульного) тестирования (Рис. 2) предполагает создание тестов на основе структуры системы и ее реализации. Такой подход иногда называют тестированием методом «белого ящика», чтобы отличать его от тестирования методом «черного ящика». Структурное тестирование так же называют тестированием путем покрытия логики.

Как правило, структурное тестирование применяется к относительно небольшим программным элементам. Набор тестовых входных данных определяется исходя из структуры программы, наличия и взаимного расположения в коде определенных конструкций языка программирования — циклов, условных операторов.

Основными методами структурного тестирования являются покрытие операторов программы, ветвей программы, условий.

Критерии структурного тестирования:

С0 — условие тестирования команд, заключается в выполнении каждого оператора хотя бы один раз.

С1 — условие тестирование ветвей, требуется выполнение каждой ветви программы не менее 1 раза.

C2 — критерий покрытия всех путей в управляющем графе программы.

Тестирование в реальных условиях, как правило выполняется на месте использование программного продукта пользователями или совместно с заказчиком.

Среди особенностей тестирования в реальных условиях выделяются:

‒ Возможность проверки разработанного программного продукта в условиях реальной эксплуатации.

‒ Высокая стоимость устранения ошибки.

‒ Необходимость поддержки тестирование в виде консультаций специалистов по поддержке программных продуктов.

В результате исследования современных методов тестирования программного обеспечения (ПО) было установлено, что не существует универсального средства для избавления программного обеспечения от ошибок.

На этапе разработки наиболее эффективно представляется стратегия структурного тестирования. На этапе интеграции отдельных модулей в единую систему целесообразно использовать стратегию функционального тестирования.

**Динамический анализ кода**

Динамический анализ кода - это способ анализа программы непосредственно при ее выполнении. Процесс динамического анализа можно разбить на несколько этапов - подготовка исходных данных, проведение тестового запуска программы и сбор необходимых параметров, анализ полученных данных. При тестовом запуске исполнение программы может выполняться как на реальном, так и на виртуальном процессоре. Для этого из исходного кода в обязательном порядке должен быть получен исполняемый файл, то есть нельзя таким способом проанализировать код, содержащий ошибки компиляции или сборки.

Динамический анализ может производиться для программ, написанных на различных языках программирования C, C++, Java, C#, PHP, Python, Erlang и многих других.

Для динамического анализа кода используются специальные утилиты, предназначенные для запуска программы, сбора и анализа полученных данных. Многие современные среды разработки уже содержат в своем составе инструменты для динамического анализа кода, например отладчик и профилировщик в Microsoft Visual Studio 2012.

Программы для динамического анализа различаются по способу взаимодействия с проверяемой программой:

* инструментирование исходного кода - в исходный текст приложения, до начала компиляции, добавляется специальный код для обнаружения ошибок;
* инструментирование объектного кода - код добавляется непосредственно в исполняемый файл;
* инструментирование кода на этапе компиляции - проверочный код добавляется, используя специальные ключи компилятора (например, такой режим поддерживается компилятором GNU C/C++ 4.x);
* не изменяет исходную программу, используются специализированные библиотеки этапа исполнения - для обнаружения ошибок используются специальные отладочные версии системных библиотек.

Динамический анализ выполняется с помощью набора данных, которые подаются на вход исследуемой программе. Поэтому эффективность анализа напрямую зависит от качества и количества входных данных для тестирования. Именно от них зависит полнота [покрытия кода](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BA%D1%80%D1%8B%D1%82%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0), которая будет получена по результатам тестирования.

С помощью динамического тестирования могут быть получены следующие метрики:

* используемые ресурсы - время выполнения программы в целом или ее отдельных модулей, количество внешних запросов (например, к базе данных), количество используемой оперативной памяти и других ресурсов;
* цикломатическая сложность, степень покрытия кода тестами и другие метрики программы;
* программные ошибки - деление на ноль, разыменование нулевого указателя, утечки памяти, "состояние гонки";
* наличие в программе уязвимостей.

Динамический анализ имеет как слабые, так и сильные стороны.

Достоинства динамического анализа кода:

* В большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе; таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения;
* зачастую не требуется исходный код; это позволяет протестировать программы с закрытым кодом.

Недостатки динамического анализа кода:

* динамический анализ обнаруживает дефекты только на трассе, определяемой конкретными входными данными; дефекты, находящиеся в других частях программы, не будут обнаружены;
* не может проверить правильность работы кода, что код делает то, что должен;
* требуются значительные вычислительные ресурсы для проведения тестирования;
* только один путь выполнения может быть проверен в каждый конкретный момент времени, что требует большого количества тестовых запусков для большей полноты тестирования;
* при тестировании на реальном процессоре исполнение некорректного кода может привести к непредсказуемым последствиям.

Имея свои сильные и слабые стороны, динамический анализ наиболее эффективно может быть использован вместе со статическим анализом кода.

К основным преимуществам статического анализа можно отнести:

* Обнаружение ошибок на ранних этапах разработки программного обеспечения. Это существенно снижает стоимость устранения дефектов в программе, так как чем раньше выявлена ошибка, тем легче и, как следствие, дешевле её исправить.
* Позволяет точно определять местонахождение потенциальной ошибки в исходном коде.
* Полное покрытие кода. Вне зависимости от того, как часто получают управление те или иные участки кода во время исполнения программы, весь исходный код будет полностью проанализирован.
* Простота использования. Для запуска статического анализа не нужно заранее подготавливать какие-либо наборы входных данных.
* С помощью статического анализа кода можно достаточно легко и быстро обнаруживать опечатки и последствия использования Copy-Paste.

К объективным недостаткам статического анализа кода относятся следующие факторы:

* Неизбежное появление так называемых ложно-позитивных срабатываний.
* 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.

**Тестирование на проникновение**

Тестирование на проникновение является одной из методик выявления областей системы, уязвимых для вторжения и компрометации целостности и достоверности со стороны неавторизованных и злонамеренных пользователей или сущностей. Процесс тестирования проникновения включает в себя умышленные санкционированные атаки на систему, способные выявить как ее наиболее слабые области, так и пробелы в защите от сторонних проникновений, и тем самым улучшить атрибуты безопасности.

Данная методика также может быть использована в качестве дополнения к другим методам проверки для оценки эффективности комплекса защиты системы от различных типов неожиданных вредоносных атак.

Каковы причины уязвимостей системы?

Пробелы в безопасности появляются на разных стадиях процесса и зависят от множества факторов:

* + ошибка проектирования (например, недоработки в дизайне – один из наиболее важных факторов возникновения лазеек в безопасности);
  + некорректная настройка и неудачная конфигурация связанного оборудования и программного обеспечения;
  + проблемы сетевого подключения (безопасное подключение устраняет возможность вредоносных атак, а небезопасная сеть обеспечивает шлюз хакерам для нападения на систему);
  + человеческая ошибка (ошибка, совершенная преднамеренно или непреднамеренно отдельным лицом или командой при проектировании, развертывании и обслуживании системы или сети);
  + погрешность коммуникации (неправильная или открытая передача конфиденциальных данных и информации среди команд или отдельных лиц);
  + чрезмерная сложность системы (контролировать механизм безопасности простой сетевой инфраструктуры легко, а отслеживать утечки или любую злонамеренную деятельность в сложных системах трудно);
  + недостаточность обучения (отсутствие знаний и должной подготовки по вопросам безопасности как у внутренних сотрудников, так и у тех, кто работает за пределами организационной структуры).

Чем отличаются тестирование на проникновение и оценка уязвимости?

Обе эти методики преследуют одну цель – сделать программный продукт безопасным, но имеют разные рабочие процессы.

Тестирование на проникновение – это проверка в реальном времени вручную или с помощью инструментов автоматизации; система и связанный с ней компонент подвергаются воздействию сэмулированных злонамеренных атак для выявления недостатков безопасности.

Оценка уязвимости включает в себя изучение и анализ системы с помощью инструментов тестирования с целью обнаружения лазеек в защите для нескольких вариантов вредоносных атак. Благодаря этой методике выявляются уязвимые области, которые могут предоставить хакерам возможность скомпрометировать систему. Кроме того, в процессе оценки уязвимости предусмотрены различные корректирующие меры, направленные на устранение выявленных недостатков.

Оценка уязвимости следует заранее определенной и установленной процедуре, в то время как тестирование на проникновение решает единственную задачу — разрушения системы вне зависимости от принятых подходов.

Для чего нужно тестирование на проникновение?

Пробелы в безопасности обеспечивают неавторизованному пользователю или незаконному объекту возможность для атаки на систему, влияющей на ее целостность и конфиденциальность. Таким образом, тестирование программных продуктов на проникновение помогает избавиться от этих уязвимостей и сделать систему достаточно компетентной для защиты от ожидаемых и даже неожиданных вредоносных угроз и атак.

Рассмотрим результаты применения данной методики подробнее. Итак, тестирование на проникновение предоставляет:

* + Способ выявления слабых и уязвимых областей системы еще до того, как их заметит хакер. Частые и сложные обновления системы могут повлиять на соответствующее оборудование и программное обеспечение, что приводит к проблемам безопасности, – следовательно, уместно контролировать все эти обновления.
  + Возможность оценки существующего механизма безопасности системы. Это позволяет разработчикам оценить свою компетентность в защите и поддерживать уровень стандартов безопасности, установленный в системе. Помимо уязвимости системы рекомендуется также с помощью бизнес- и технической команд оценивать различные бизнес-риски и проблемы, включая любой компромисс с разрешенными и конфиденциальными данными организации. Это помогает организации структурировать и устанавливать приоритеты, смягчая или полностью исключая различные бизнес-риски и проблемы.
  + Наконец (но не в последнюю очередь), инструмент для выявления и удовлетворения определенных основных стандартов, норм и практик безопасности.

Как выполнить тестирование на проникновение?

Тестирование на проникновение системы может осуществляться с использованием любого из следующих подходов:

* 1. ручное тестирование;
  2. автоматическое тестирование;
  3. сочетание ручного и автоматического тестирования.

**1. Ручное тестирование на проникновение**

Для проведения ручного тестирования на проникновение программного продукта используется последовательный стандартный подход, включающий следующие этапы:

* 1. **Планирование тестирования проникновения.** Этот этап включает в себя сбор требований, определение сферы применения, стратегий и целей тестирования проникновения в соответствии с нормами безопасности. Кроме того, он может содержать оценку и перечисление проверяемых областей, типы планируемых испытаний и другие связанные с этим проверки.
  2. **Разведка.** Сбор и анализ максимально подробной информации о системных и связанных с ними атрибутах безопасности, полезных для таргетинга и атаки на каждый блок, для эффективного и результативного тестирования системы проникновения в систему. Различают две формы сбора и анализа информации о целевой системе: пассивная и активная разведка (в первом случае не предполагается прямое взаимодействие с системой).
  3. **Анализ уязвимости.** На этом этапе тестировщики выявляют и обнаруживают уязвимые области системы, которые в дальнейшем будут использоваться для входа и атаки с помощью тестов на проникновение.
  4. **Эксплуатация.** Фактическое испытание на проникновение в систему, включающее внутренние и внешние атаки. Внешние атаки – это сэмулированные атаки со стороны внешнего мира, преобладающие за пределами границы системы / сети (например, получение несанкционированного доступа к функциям и данным системы, относящимся к приложениям и серверам, обращенным к общественности). Внутренние атаки начинаются уже после вторжения авторизованных объектов в систему или сеть и имеют целью различные действия (при достижении компромисса с целостностью и правдивостью системы), которые способны преднамеренно или непреднамеренно скомпрометировать систему.
  5. **Пост-эксплуатация.** Следующий шаг – анализ каждой атаки на систему для оценки ее цели и задачи, а также ее потенциального воздействия на системные и бизнес-процессы.
  6. **Отчетность.** На самом деле, отчетность включает в себя документационную работу по мероприятиям, проводимым на всех упомянутых этапах. Кроме того, она может описывать различные риски, выявленные проблемы, уязвимые области (использованные или нет) и предлагаемые для устранения недостатков решения.

**2. Автоматическое тестирование на проникновение**

Этот полезный и эффективный подход к проведению испытаний на проникновение предполагает использование специализированного инструментария. Автоматическое тестирование надежно, удобно, оно происходит очень быстро и легко поддается анализу. Инструменты проверки эффективны для точного обнаружения дефектов безопасности, присутствующих в системе, за короткий промежуток времени, а также для создания «кристально чистых» отчетов.

Назовем лишь некоторые из популярных и широко используемых инструментов тестирования на проникновение:

* Nmap;
* Nessus;
* Metasploit;
* Wireshark;
* OpenSSL;
* Cain & Abel;
* THC Hydra;
* w3af.

Многие инструменты для автоматизированного тестирования можно найти в готовых сборках Linux (Kali Linux, Mantra OS).

Для работы над конкретным проектом придется выбирать инструмент, отвечающий целому ряду требований и критериев:

* + удобство развертывания, использования и обслуживания;
  + обеспечение простого и быстрого сканирования системы;
  + возможность автоматизации процесса проверки выявленных уязвимостей;
  + доступность проверки ранее обнаруженных уязвимостей;
  + способность создания простых и подробных отчетов об уязвимостях.

**3. Сочетание ручного и автоматического тестирования на проникновение**

Данный подход может быть признан оптимальным, так как он сочетает в себе преимущества первых двух вариантов и обеспечивает оперативный контроль с помощью надежного и точного проникновения в программный продукт.

**Типы испытаний на проникновение**

Тестирование на проникновение в зависимости от используемых элементов и объектов может быть отнесено к следующим типам:

* 1. **Социальная инженерия.** Тестирование с подключением «человеческого контингента», способного четко выявлять и получать конфиденциальные данные и другую информацию через Интернет или телефон (к этой группе могут относиться сотрудники организации или любые другие уполномоченные лица, присутствующие в сети организации).
  2. **Веб-приложение.** Используется для обнаружения прорех в безопасности и иных проблем в нескольких вариантах веб-приложений и сервисов, размещенных на стороне клиента или сервера.
  3. **Сетевая служба.** Тестирование проникновения в сеть для выявления и обнаружения возможности доступа хакерам или любому неавторизованному объекту.
  4. **Клиентская часть.** Как видно из названия, этот тест используется для тестирования приложений, установленных на клиентском сайте / приложении.
  5. **Удаленное подключение.** Тестирование vpn или аналогичного объекта, который может обеспечить доступ к подключенной системе.
  6. **Беспроводные сети.** Тест предназначен для беспроводных приложений и сервисов, включая их различные компоненты и функции (маршрутизаторы, фильтрационные пакеты, шифрование, дешифрование и т.д.).

Классифицировать тестирование на проникновение также можно и на основе используемых подходов к тестированию:

* 1. **Белый ящик.** При таком подходе тестировщик будет иметь полный доступ к глубоким знаниям о функционировании и основных атрибутах системы. Это тестирование очень эффективно, так как понимание каждого аспекта системы очень полезно при проведении обширных испытаний на проникновение.
  2. **Черный ящик.** Тестировщикам предоставляется только высокоуровневая информация (например, URL или IP-адрес организации) для проведения тестирования на проникновение. Специалист может ощутить себя хакером, который ничего не знает о системе / сети. Это весьма трудоемкий подход, так как тестировщику требуется значительное количество времени для изучения свойств и деталей системы; кроме того, высока вероятность пропустить часть областей из-за недостатка времени и информации.
  3. **Серый ящик.** Тестировщик получает ограниченную информацию (например, знания алгоритма, архитектуры, внутренних состояний) для имитации внешней атаки на систему.

Ограничения тестирования на проникновение.

У тестирования на проникновение существует ряд ограничений:

* + недостаток времени и высокая стоимость тестирования;
  + ограниченный объем испытаний, основанный на требованиях за данный период времени (что может привести к игнорированию других важных областей);
  + возможность разрушения системы или потери системы в состоянии отказа в результате испытания на проникновение;
  + уязвимость данных (потеря, коррупция или ущерб).

#### Управление процессом тестирования

Все способы тестирования ПС объединяются базой данных, где помещаются результаты тестирования системы. В ней содержатся все компоненты, тестовые контрольные данные, результаты тестирования и информация о документировании процесса тестирования.

База данных проекта поддерживается специальными инструментальными средствами типа CASE, которые обеспечивают ведение анализа ПрО, сборку данных об их объектах, потоках данных и тому подобное. База данных проекта хранит также начальные и эталонные данные, которые используются для сопоставления данных,накопленных в базе, с данными, которые получены в процессе тестирования системы.

При тестировании выполняются разные *виды расчетов* характеристик этого процесса и способы планирования и управления.

1. Расчет продолжительности выполнения функций путем сбора средних показателей скорости выполнения операторов без выполнения программы на машине. Выявляются компоненты, которые требуют большого времени выполнения в реальной среде.
2. Управление выполнением тестирования путем подбора тестов проверки, их выполнения, селекции результатов тестирования и проведения сопоставления с эталонными значениями. Результаты данного процесса отображаются на дисплее, например, в графической форме (пути прохождения по графу программы), в виде диаграмм UML, данных об отказах и ошибках или конкретных значений исходных параметров программы. Эти данные анализируются разработчиками для формулирования выводов о направлениях дальнейшей проверки правильности программы или их завершении.
3. Планирование тестирования предназначено для распределения сроков работ по тестированию, распределения тестировщиков по отдельным видам работ и составления ими тестов проверки системы. Определяются стратегия и пути тестирования. В диалоге запрашиваются данные о реальных значениях процесса выполнения системы, структуре ветвления вершин графа и параметрах циклов. Проверенные циклы, как правило, изымаются из путей выполнения программы. При планировании путей выполнения создаются соответствующие тесты, критерии и входные значения.

Документирование результатов тестирования в соответствии с действующим стандартом ANSI/IEEE 829 включает:

* описание задач, назначение и содержание ПС, а также перечень функций в соответствии с требованиями заказчика;
* технологии разработки системы;
* планов тестирования различных объектов, необходимых ресурсов, соответствующих специалистов для проведения тестирования и технологических способов;
* тестов, контрольных примеров, критериев и ограничений, оценки результатов программного продукта, а также процесса тестирования;
* учета процесса тестирования, составление отчетов об аномальных событиях, отказах и дефектах в итоговом документе системы.

### Инфраструктура процесса тестирования ПС

Под инфраструктурой процесса тестирования понимается:

* выделение объектов тестирования;
* проведение классификации ошибок для рассматриваемого класса тестируемых программ;
* подготовка тестов, их выполнение и поиск разного рода ошибок и отказов в компонентах и в системе в целом;
* служба проведения и управление процессом тестирования;
* анализ результатов тестирования.

**Объекты тестирования**- компоненты, группы компонентов, подсистемы и система. Для каждого из них формируется стратегия проведения тестирования. Если *объект* тестирования относится к "белому ящику" или "черному ящику", состав компонентов которого неизвестный, то тестирование проводится посредством ввода внего входных тестовых данных для получения выходных данных.

#### Методы поиска ошибок в программах

Международный стандарт ANSI/IEEE-729-83 разделяет все ошибки в разработке программ на следующие типы.

*Ошибка*(error) - состояние программы, при котором выдаются неправильные результаты, причиной которых являются изъяны (*flaw*) в операторах программы или в технологическом процессе ее разработки, что приводит к неправильной интерпретации исходной информации, следовательно, и к неверному решению.

*Дефект*(fault) в программе - следствие ошибок разработчика на любом из этапов разработки, которая может содержаться в исходных или проектных спецификациях, текстах кодов программ, эксплуатационной документация и т.п. В процессе выполнения программы может быть обнаружен дефект или сбой.

*Отказ*(failure) - это отклонение программы от функционирования или невозможность программы выполнять функции, определенные требованиями и ограничениями, что рассматривается как событие, способствующее переходу программы в неработоспособное состояние из-за ошибок, скрытых в ней дефектов или сбоев в среде функционирования. Отказ может быть результатом следующих причин:

* ошибочная спецификация или пропущенное требование, означающее, что спецификация точно не отражает того, что предполагал пользователь;
* спецификация может содержать требование, которое невозможно выполнить на данной аппаратуре и программном обеспечении;
* проект программы может содержать ошибки (например, база данных спроектирована без средств защиты от несанкционированного доступа пользователя, а требуется защита);
* программа может быть неправильной, т.е. она выполняет несвойственный алгоритм или он реализован не полностью.

Таким образом, отказы, как правило, являются результатами одной или более ошибок в программе, а также наличия разного рода дефектов.

**Ошибки на этапах процесса тестирования**.Приведенные типы ошибок распределяются по этапам ЖЦ и им соответствуют такие источники их возникновения:

* непреднамеренное отклонение разработчиков от рабочих стандартов или планов реализации;
* спецификации функциональных и интерфейсных требований выполнены без соблюдения стандартов разработки, что приводит к нарушению функционирования программ;
* организации процесса разработки - несовершенная или недостаточное управление руководителем проекта ресурсами (человеческими, техническими, программными и т.д.) и вопросами тестирования и интеграции элементов проекта.

Рассмотрим процесс тестирования, исходя из рекомендаций стандарта ISO/IEC 12207, и приведем типы ошибок, которые обнаруживаются на каждом процессе ЖЦ.

*Процесс разработки требований*. При определении исходной концепции системы и исходных требований к системе возникают ошибки аналитиков при спецификации верхнего уровня системы и построении концептуальной модели предметной области.

Характерными ошибками этого процесса являются:

* неадекватность спецификации требований конечным пользователям;- некорректность спецификации взаимодействия ПО со средой функционирования или с пользователями;
* несоответствие требований заказчика к отдельным и общим свойствам ПО;
* некорректность описания функциональных характеристик;
* необеспеченность инструментальными средствами всех аспектов реализации требований заказчика и др.

*Процесс проектирования*.Ошибки при проектировании компонентов могут возникать при описании алгоритмов, логики управления, структур данных, интерфейсов, логики моделирования потоков данных, форматов ввода-вывода и др. В основе этих ошибок лежат дефекты спецификаций аналитиков и недоработки проектировщиков. К ним относятся ошибки, связанные:

* с определением интерфейса пользователя со средой;
* с описанием функций (неадекватность целей и задач компонентов, которые обнаруживаются при проверке комплекса компонентов);
* с определением процесса обработки информации и взаимодействия между процессами (результат некорректного определения взаимосвязей компонентов и процессов);
* с некорректным заданием данных и их структур при описании отдельных компонентов и ПС в целом;
* с некорректным описанием алгоритмов модулей;
* с определением условий возникновения возможных ошибок в программе;
* с нарушением принятых для проекта стандартов и технологий.

*Этап кодирования*.На данном этапе возникают ошибки, которые являются результатом дефектов проектирования, ошибок программистов и менеджеров в процессе разработки и отладки системы. Причиной ошибок являются:

* бесконтрольность значений входных параметров, индексов массивов, параметров циклов, выходных результатов, деления на 0 и др.;
* неправильная обработка нерегулярных ситуаций при анализе кодов возврата от вызываемых подпрограмм, функций и др.;
* нарушение стандартов кодирования (плохие комментарии, нерациональное *выделение модулей* и компонент и др.);
* использование одного имени для обозначения разных объектов или разных имен одного объекта, плохая мнемоника имен;- несогласованное внесение изменений в программу разными разработчиками и др.

*Процесс тестирования*.На этом процессе ошибки допускаются программистами и тестировщиками при выполнении технологии сборки и тестирования, выбора тестовых наборов и сценариев тестирования и др. Отказы в программном обеспечении, вызванные такого рода ошибками, должны выявляться, устраняться и не отражаться на статистике ошибок компонент и программного обеспечения в целом.

*Процесс сопровождения*.На процессе сопровождения обнаруживаются ошибки, причиной которых являются недоработки и дефекты эксплуатационной документации, недостаточные показатели модифицируемости и удобочитаемости, а также некомпетентность лиц, ответственных за сопровождение и/или усовершенствование ПО. В зависимости от сущности вносимых изменений на этом этапе могут возникать практически любые ошибки, аналогичные ранее перечисленным ошибкам на предыдущих этапах.

Все ошибки, которые возникают в программах, принято подразделять на следующие классы:

* логические и функциональные ошибки;
* ошибки вычислений и времени выполнения;
* ошибки вводавывода и манипулирования данными;
* ошибки интерфейсов;
* ошибки объема данных и др.

*Логические ошибки*являются причиной нарушения логики алгоритма, внутренней несогласованности переменных и операторов, а также правил программирования. Функциональные ошибки - следствие неправильно определенных функций, нарушения порядка их применения или отсутствия полноты их реализации и т.д.

*Ошибки вычислений*возникают по причине неточности исходных данных и реализованных формул, погрешностей методов, неправильного применения операций вычислений или операндов. Ошибки времени выполнения связаны с необеспечением требуемой скорости обработки запросов или времени восстановления программы.

*Ошибки ввода-вывода*и манипулирования данными являются следствием некачественной подготовки данных для выполнения программы, сбоев при занесении их в базы данных или при выборке из нее.

*Ошибки интерфейса*относятся к ошибкам взаимосвязи отдельных элементов друг с другом, что проявляется при передаче данных между ними, а также при взаимодействии со средой функционирования.

*Ошибки объема*относятся к данным и являются следствием того, что реализованные методы доступа и размеры баз данных не удовлетворяют реальным объемам информации системы или интенсивности их обработки.

**Стандарты**

Состав и содержание документации, сопутствующей процессу тестирования, определяется стандартом IEEE 829-2008 IEEE Standard for Software and System Test Documentation

В стандарте IIЕЕ 1209-1992 сформулированы *общие требования к функциям средств автоматизации тестирования,* входящим в CASE-средства, которые должны обеспечивать:

• определение тестов — реализацию процесса тестирования пользователем: ввод тестовых наборов, генерацию тестовых наборов:

• генерацию тестовых данных, ввод ожидаемых, эталонных результатов, генерацию ожидаемых результатов;

• выполнение участка тестируемой программы между контрольными точками, для которого CASE-средство может перехватить операторский ввод (клавиатуры, мыши и т.д.) и для которого вводимые данные могут быть отредактированы и включены в последующие тестовые наборы;

• управление тестами и участком программы, для которого CASE-средство может автоматически выполнять тестовые наборы;

• регрессионное тестирование с возвратом от более сложных тестов к простым, возможность перезапускать предыдущие тесты и возможность модифицировать предыдущие тесты, чтобы учесть различия в системе и/или среде (например, дату и

время);

• анализ тестовых результатов — возможность CASE-средства автоматически анализировать тестовые результаты: сравнение ожидаемых и реальных результатов, сравнение файлов, статистический анализ результатов;

• анализ покрытия тестами исходного кода для обнаружения операторов (элементов текста программы, выражающих законченные действия), которые были/не были выполнены, процедур, которые были/не были вызваны, и переменных, к которым были/ не были обращения;

• анализ производительности программы, когда она выполняется: загрузку центрального процессора, загрузку памяти, обращения к специфицированным элементам данных и/или сегментам кода, временные характеристики;

• верификацию условий или исключительных ситуаций во время выполнения теста;

• моделирование среды — поддержку процесса тестирования с помощью модели, например, возможность CASE-средства автоматически генерировать входы моделируемых систем на основе полученных системных выходов.

ГОСТ Р ИСО/МЭК 12119-2000 содержит указания, которые определяют порядок тестирования продукта на соответствие его требованиям к качеству. Эти указания охватывают как тестирование для характеристик, присущих аналогичным продуктам, так и тестирование для характеристик, указанных в описании продукта. Указания охватывают тестирование путем проверки документов и тестирование программ и данных по принципу “черного ящика».

ГОСТ описывает только функциональное тестирование (тестирование по принципу “черного ящика”), а структурное тестирование не охватывается, потому что для его проведения необходимо наличие исходного кода. В нем рассматривают только тестирование продукта в необходимых для него системах. Эргономическую оценку на рабочем пространстве вычислительной системы в настоящем стандарте тоже не рассматривают.