Лекция 7. Виды и методы тестирования

Видов тестирования достаточно много. Классификации разных авторов совпадают не по всем признакам классификации. Попытки систематизации методов и технологий тестирования встречаются сравнительно редко. В данном пособии также не будем пытаться «объять необъятное» и рассмотрим лишь некоторые виды и методы тестирования.

7.1. Классификация видов тестирования по цели тестирования

По цели можно выделить тестирование:

- с целью выявления факта наличия ошибки;
- с целью локализации ошибки, выявления причины;
- с целью аттестации программного средства.

Аттестация – процессы проверки и анализа ПП, в ходе которых выявляется соответствие программного обеспечения своей спецификации и требованиям заказчиков. Выполняется представительной комиссией, в состав которой входят эксперты, представители заказчика и представители разработчика.

Основными методиками проверки и анализа систем при аттестации являются инспектирование и тестирование ПО. *Инспектирование* относится к статическим методам и предполагает анализ и проверку полноты и точности различных представлений системы: спецификации требований, архитектурных схем, исходного кода (в том числе и документации по тестированию и отладке ПС). Может выполняться на разных этапах процесса разработки программной системы. Но статические методы не позволяют проверить правильность функционирования системы, ее производительность и надежность. *Тестирование* – это динамический метод аттестации. Тестирование включает запуск исполняемого кода с тестовыми данными и исследование выходных данных и рабочих характеристик программного продукта.

Основным видом испытаний при аттестации ПС являются приемо-сдаточные испытания, во время которых оценивают качество программных документов и выборочно пропускают тесты разработчиков, контрольные задачи пользователей и дополнительные тесты, подготовленные комиссией для оценки качества аттестуемого ПП.

Локализацией называют процесс определения оператора программы, вызвавшего нарушение нормального вычислительного процесса. При тестировании с целью локализации ошибок используют следующие **методы**:

***Метод ручного тестирования*.** Предполагает выполнение программы вручную с использованием тестового набора, при работе с которым обнаружена ошибка. Можно в необходимых местах расставить точки останова, подозрительные участки кода выполнить в пошаговом режиме, наблюдая значения переменных в окнах отладчика. Достоинство метода - простота. Но метод неприменим для больших программ, в которых необходимо выявить причины более сложных ошибок. Например, нужно исследовать программу на утечки памяти, или ускорить ее выполнение, или при выполнении программы компьютер просто “зависает” и т.п.

***Метод индукции*.** Индукция - познавательная процедура, посредством которой из сравнения имеющихся частных фактов выводится обобщающее их утверждение. Анализируя результаты теста, выявляют симптомы ошибки, записывают все ее проявления. Полученную информацию обобщают в виде предполагаемой причины ошибки. Если информация о симптомах ошибки недостаточно полна, гипотеза может оказаться ошибочной.

Метод дедукции. Опирается на некоторые общие теории, логический закон, аксиомы, предпосылки. Если посылки дедукции истинны, то истинны и её следствия. По методу дедукции вначале составляют список причин (как список болезней с описанием их симптомов), которые могли бы вызвать данное проявление ошибки. Затем, соотносят имеющиеся данные об ошибке с причинами. При наличии противоречий гипотезу из списка исключают.

Метод обратного прослеживания. Для точки вывода неправильного результата строится гипотеза о значениях основных переменных, которые могли к нему привести. Затем выявляют факторы, влияющие на значения этих переменных и далее в соответствии с логикой программы до оператора, вызвавшего ошибку.

Отладочный вывод. Предполагает включение в программу дополнительных операторов отладочного вывода. Это позволяет отследить значения переменных в узловых точках. Основным недостатком здесь является внесение в программу изменений, которые могут скрыть ошибку или добавить новую. Применение интегрированных средств отладки позволяет оставить программу без изменений: специальные макросы и функции будут выводить сообщения в окна отладчика.

7.2. Классификация по виду программного документа, на основе которого строятся тесты.

а) Комплексное тестирование охватывает ***весь комплект документации*** на ПП и включает тестирование архитектуры, тестирование внешних функций, тестирование качества ПС, тестирование документации, тестирование определения требований. Тестируется ПП в целом. Тестирование документов производится, как правило, в порядке, обратном их разработке. Но тестирование документации по применению лучше выполнить после завершения тестирования внешнего описания. Все тесты готовятся в форме, рассчитанной на пользователя.

Тестирование архитектуры направлено на поиск ошибок реализации архитектуры, несоответствия между описанием архитектуры и совокупностью программ ПП. Желательно проверить все пути взаимодействия подсистем.

Тестирование внешних функций. Производится по принципу черного ящика. Цель – выявить несоответствия между функциональной спецификацией и совокупностью программ ПП.

Тестирование качества. Цель – выявить несоответствия между требованиями спецификации качества и совокупностью программ ПП. Тестируются точность, устойчивость, защищенность, временная эффективность, эффективность по памяти, расширяемость и некоторые другие.

Тестирование документации по применению ПП. Ищут несогласованности документации по применению с совокупностью программ ПП. Фактически выполняется тестирование внешних функций, но тесты готовятся исключительно на основании документации по применению. Следует обратить внимание на неясные места в документации, на приведённые в документации примеры.

Тестирование определения требований к ПП. Осуществляется организацией-приобретателем, обычно с помощью типовых задач, для которых известен результат решения (например, с помощью предыдущей версии ПС).

Автономное тестирование предполагает последовательное раздельное тестирование различных частей ***программного обеспечения***. Цель – выявить ошибки компонентов ПП и показать, что по отдельности эти части работоспособны. Автономное тестирование может выполняться сразу после написания очередного участка кода, чтобы убедиться в корректной работе новой функции.

Обычно при автономном тестировании каждый модуль проверяется в некотором программном окружении, включающем компоненты с ним

взаимодействующие. Окружение может состоять из уже проверенных модулей и специально написанных отладочных модулей (драйверов или заглушек), управляющих тестированием. Состав отладочных модулей, входящих в окружение, зависит от последовательности тестирования модулей, их типа и от того, какой тест будет пропускаться. Модуль-драйвер - модуль, который содержит фиксированные тестовые данные, вызывает тестируемый модуль и отображает выходные результаты. Заглушка имитирует работу модуля нижнего уровня и может содержать только сообщение о том, что был вход в этот модуль, и возврат управления.

Достоинством автономного тестирования является возможность полной реализации плана тестирования модуля.

Недостатком – необходимость отдельного тестирования **сопряжения модулей:** спецификация каждого модуля используется не только при разработке текста (тела) этого модуля, но и при написании обращения к нему других компонент. Тестирование сопряжения модулей призвано обнаруживать ошибки взаимодействия компонент и несогласованности их интерфейсов.

7.3. Классификация по последовательности сборки и тестирования компонент

По последовательности сборки и тестирования компонент можно выделить монолитное, пошаговое, нисходящее, восходящее и некоторые другие виды тестирования.

При **монолитном** тестировании последовательность тестирования компонент не оговаривается. После независимого тестирования каждого модуля осуществляют сборку системы. Далее тестируется полученная версия (сборка, релиз) ПП. Монолитное тестирование позволяет сократить расход машинного времени и ускорить сроки сдачи программы.

При **пошаговом** тестировании каждый проверяемый модуль для тестирования подключается к набору ранее оттестированных модулей. При этом ошибки в межмодульных связях обнаруживаются раньше и легче.

При **восходящем** тестировании проверка компонент начинается с модулей нижнего уровня. При этом работу остальной части программы имитирует ведущий модуль (драйвер). Он готовит информационную среду для тестирования отлаживаемого модуля, осуществляет обращение к модулю и после окончания его работы выдает необходимые сообщения. Для разных тестов необходимы разные драйверы. Отсюда – большой объем отладочного программирования. Также к *недостатками* восходящего тестирования относят необходимость специального тестирования сопряжения модулей, искусственность тестовых данных (не рассчитаны на пользователя). *Достоинство* – так как тестовое состояние информационной среды готовится непосредственно перед обращением к проверяемому модулю, легко подготовить тесты, позволяющие полностью реализовать его план тестирования.

При **нисходящем** тестировании проверку компонент начинают с головного, модуля, обеспечивающего интерфейс с пользователем. Окружение отлаживаемого модуля включает уже проверенные компоненты верхнего уровня и отладочные имитаторы (заглушки) ещё не протестированных модулей, к которым он будет обращаться. К *преимуществам* подхода относят:

- возможность подготовки тестов в форме, рассчитанной на пользователя;
- попутное тестирование сопряжений модулей;

— малый объем отладочного программирования: заглушки весьма просты, каждая из них пригодна для большого числа тестов и, при необходимости, ее легко изменить для других тестов.

Однако, подготовка тестового состояние информационной среды перед обращением к отлаживаемому модулю косвенным способом затрудняет подготовку тестов и реализацию полного плана тестирования отлаживаемого модуля, требует высокой квалификации разработчика тестов.

Существуют некоторые разновидности метода нисходящего тестирования. Также методы восходящего и нисходящего тестирования могут различаться последовательностью тестирования компонент одного уровня и последовательностью тестирования ветвей ПП.

Воспользоваться достоинствами и восходящего, и нисходящего тестирования, а также в значительной степени нейтрализовать их недостатки. позволяет **метод сэндвича** (гибридное тестирование). Он представляет собой комбинацию восходящего и нисходящего тестирования: первый применяется для модулей нижнего уровня, второй – для верхнего. При использовании смешанной стратегии тестирования в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов.

7.4. Классификация по способу реализации

По способу реализации тестирование можно разделить на инструментальное (с привлечением технических средств) и безинструментальное или на ручное, машинное и автоматизированное.

Машинное тестирование предполагает работу ПО на предназначенном для него оборудовании или в среде, имитирующей работу реального оборудования. Тесты инициируются соответствующими специалистами.

При **автоматизированном** тестировании для создания сценариев тестирования, запуска на выполнение заданных тестов, создания отчетов применяются специальные инструменты. Подробнее этот вид тестирования рассматривается в главе 7.

Ручное тестирование проводится без исполнения тестируемой программы на компьютере. Обычно используется на ранних этапах разработки. Предполагает анализ всех проектных решений с точки зрения их правильности и целесообразности. При **статическом** подходе объекты анализа – структура, управляющие и информационные связи программы, ее входные и выходные данные. При **динамическом** подходе вручную моделируют процесс выполнения программы на заданных исходных данных. Ручной контроль позволяет найти 30 – 70 % ошибок логического проектирования и кодирования.

Основными **методами** ручного контроля являются: инспекция исходных текстов, сквозные просмотры, проверка за столом,

Инспекция исходных текстов является способом раннего выявления частей программы с большой вероятностью содержания ошибок. В ходе заседания программа анализируется группой специалистов по списку вопросов, позволяющему выявить наиболее типичные ошибки программирования.

В число таких вопросов, как правило, входят:

— контроль обращения к данным (все ли переменные инициализированы, не превышается ли допустимый размер массивов и т. д.);
— контроль вычислений (правильно ли записаны выражения, корректно ли выполняются вычисления над не арифметическими переменными);
— контроль передачи управления (в циклах, при вызове модулей и т. п.);

— контроль межмодульных интерфейсов (области действия глобальных и локальных переменных и т. п.).

С листингом программы и спецификацией на нее участники группы знакомятся заранее. Инспектируют программу координатор (компетентный программист, но не автор программы) проектировщик, специалист по тестированию. Автор программы рассказывает о логике работы программы и отвечает на вопросы инспекторов, возникшие в ходе ее анализа.

Сквозные просмотры выполняются группой специалистов, состоящей из 3 – 5 человек: председатель (координатор), секретарь, специалист по тестированию, независимый эксперт, программист. Участники группы заранее изучают листинг программы и спецификацию на нее. На заседание выносится несколько тестов. Каждый специалист выполняет тесты в соответствии с логикой программы. Секретарь отслеживает состояние программы (значения переменных) на бумаге или доске, фиксирует ошибки. При необходимости программисту задаются вопросы о логике проектирования и принятых допущениях (при опросе автора программы выявляется обычно больше ошибок, чем при выполнении тестов).

Проверка за столом осуществляется одним специалистом по тестированию (проверку не должен выполнять автор программы). Текст программы проверяется на наличие возможных дефектов по специальному списку часто встречающихся ошибок. Также тестировщик пропускает через программу наборы тестовых данных. Вследствие неупорядоченности процесса тестирования, отсутствия обмена мнениями и конкуренции, результативность данного метода ниже, чем групповых методов.

7.5. Классификация по исполнителю

В зависимости от того, кем выполняется, тестирование бывает внутреннее и внешнее **(**альфа- бета- и гамма-тестирование)

Внутреннее тестирование (самотестирование). Включает проверку ПП а) самими программистами (создателями кода), б) группой, производящей данное программное обеспечение (разработчиками, и в частности, тестировщиками). Цель – выявить ошибки, которые могут быть найдены посредством малых затрат в простейшей детерминированной среде путем модульного или низкоуровневого системного тестирования. Отмечают высокую эффективность внутреннего тестирования.

Внешнее (независимое, пользовательское) тестирование. При этом, как правило, исследуются функциональные возможности ПП для решения задач пользователя и реализации сценариев использования системы в различных условиях, проверяется удобство установки и эксплуатации. Цель внешнего тестирования **–** разнообразить область применения программного средства, обеспечить тестирование в среде с более широкими, чем это доступно разработчику, возможностями.

По степени привлечения конечных пользователей выделяют альфа-, бета- и гамма-тестирование.

Альфа-тестирование – первая стадия пользовательского тестирования нового ПП внутри разработавшей его компании перед тем, как он выйдет за ее пределы (лабораторные испытания). Обычно заключается в систематической апробации всех функций программы. Выполняется штатными разработчиками, возможно, с привлечением потенциальных пользователей. Может являться формой внутреннего приёмочного тестирования.

Бета-тестирование (ОКЭ – опытно-конструкторская эксплуатация) – вторая стадия пользовательского тестирования, тестирование в реальных

производственных условиях, опытная эксплуатация ПП, предварительное тестирование типичными пользователями и партнерами для удаления ошибок, неадекватностей с целью усовершенствования ПП (может быть начато при неполной готовности ПП). В этом случае определенной группе лиц для пробной эксплуатации поставляются версии программного продукта с ограничениями (по функциональности или времени работы). Может являться формой внешнего приёмочного тестирования.

Гамма-тестирование – третья стадия тестирования ПП перед его коммерческим выпуском. Производится на основе отчетов реальных пользователей в процессе эксплуатации продукта. ПП д.б. полностью завершен (кроме документации и упаковки). Может являться формой внешнего приёмочного тестирования.

7.6. Классификация по объекту тестирования

Это, пожалуй, самая сложная классификация. К объектам тестирования относят функциональность, структуру, взаимосвязи компонент. Надо проверить работоспособность отдельных компонент и системы в целом. Существует большой список системных испытаний: тестирование удобства установки, конфигураций, нагрузочной способности, безопасности, удобства использования и т.д.

7.6.1. Структурное, функциональное и регрессионное тестирование

Структурное тестирование направлено на управляющие связи элементов ПО, реже на информационные связи. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Для тестирования применяют тесты, составленные в соответствии со стратегией «белого ящика» по критерию покрытия всех маршрутов программы (но, если независимых маршрутов очень много, вероятно, придётся выбирать наиболее важные). Следует помнить, что даже исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней, не позволяет обнаружить ошибки, появление которых зависит от обрабатываемых данных, и пропуск какого-либо маршрута (ошибка реализации алгоритма).

Функциональное тестирование акцентирует внимание на информационной области определения программной системы и направлено на полную проверку всех функциональных требований к ПП. Может применяться на уровне программных модулей и на уровне программной системы. Обеспечивает поиск таких категорий ошибок, которые иными способами не выявляются. Например, ошибки инициализации и завершения, ошибки интерфейса, некорректные или отсутствующие функции.

Популярным методом тестирования пользовательских функций является **параллельное тестирование**, когда одновременно испытываются существующее и новое решение: одни и те же данные последовательно загружаются в оба приложения, запускаются одни и те же пользовательские сценарии использования системы. Метод позволяет быстро проверить, ведет ли себя решение именно так, как ожидается. На основании результатов такого исследования делается вывод о ценности нового решения для пользователей. Еще вариант применения метода: несколько тестировщиков или систем автоматизации одновременно (параллельно) выполняют одни и те же действия.

Регрессионное тестирование можно рассматривать, как частный случай тестирования функционального. Предполагает проверку ПО на корректность функционирования после внесения в него изменений. Выполняется, обычно, после каждой удачной компиляции вручную или с помощью инструментов для

автоматизированного тестирования. При этом прогоняются все предыдущие тесты: и те, на которых были выявлены ошибки, и те, результаты которых соответствовали ожидаемым. Нередко в результате регрессионного тестирования «заваливаются» ранее успешные тесты (перестает работать то, что до внесения изменений в программу работало правильно). Такие ошибки называют регрессионными.

Нефункциональное тестирование направлено на проверку правильности реализации нефункциональных требований.

7.6.2. Модульное, интеграционное и системное тестирование

Эта классификация связана с уровнем детализации приложения. Разбиение на модульное, интеграционное, системное и приёмочное тестирование также может использоваться в контексте управления проектом (разделения областей ответственности).

При **модульном** тестировании проверяются отдельные небольшие части приложения, работоспособность которых можно исследовать изолированно от других подобных частей. Как разновидности модульного тестирования можно рассматривать юнит-тестирование и компонентное тестирование. *Юнит-тестирование*, как правило, отвечает за проверку атомарных участков кода, *компонентное* – за тестирование библиотек и структурных частей приложения. В этом случае модульное тестирование занимает промежуточное положение и направлено на тестирование классов и небольших библиотек.

Цель **интеграционного** тестирования – выявление проблем (например, информация не передается, передается неправильно) в интерфейсах и взаимодействии между интегрируемыми ранее проверенными корректными компонентами.

В ходе **системного** тестирования проверяется на соответствие исходным требованиям к ней интегрированная система, как единое целое, собранное из частей, проверенных на предыдущих стадиях. Системное тестирование, как правило, не требует знаний о внутреннем устройстве системы, проводится на тестах, составленных в соответствии со стратегией «чёрного ящика», и дает возможность взаимодействовать с приложением с позиций конечного пользователя.

Разновидности системного тестирования: функциональное тестирование, тестирование пользовательского интерфейса, юзабилити-тестирование, тестирование совместимости, тестирование на основе модели, тестирование безопасности, тестирование производительности, регрессионное тестирование и некоторые другие.

Например, **тестирование совместимости** направлено на проверку способности приложения к взаимодействию с существующими системами и программными решениями. Проверяется совместимость с браузерами и их версиями, с мобильными устройствами, с аппаратной платформой, операционной системой, сетевой инфраструктурой и так далее. К такому тестированию часто привлекаются группы, внешние по отношению к проектной команде. В ряде случаев это объясняется невозможностью выполнения тестов в полностью изолированной среде. Участие сторонней команды следует предусмотреть в плане.

Конфигурационное тестирование перекликается с тестированием совместимости. Задача конфигурационного тестирования проверить работоспособность приложения в каждой из возможных его конфигураций.

Тестирование **удобства установки** (настройки, инсталляции). направлено на выявление дефектов стадии инсталляции приложения. Проверяют различные сценарии установки: первичная установка в новой среде, «переинсталляция»,

обновление существующей версии, удаление приложения, автоматическая инсталляция и некоторые другие.

При тестировании пользовательского **интерфейса** (в т.ч. и интерфейса командной строки) исследуется правильность его работы. Следует помнить, что корректно работающий интерфейс может быть неудобным, а удобный может работать некорректно.

Цель тестирования **удобства использования** выяснить, насколько конечному пользователю понятно, как работать с полученной программной системой, какова его удовлетворённость от этой работы. В такой постановке решение этой задачи требует сбора статистической информации от значительного числа конечных пользователей, проведение маркетинговых исследований. Другой путь – тестирование всей сопровождающей ПП **документации и справочной системы.** Документы и файлы справочной системы проверяются на предмет соответствия целям проекта. Ведется поиск некорректных инструкций, устаревшей информации и снимков экранов, опечаток и т.п. Проверяется значимость сообщений программы, понятность диагностики ошибок, единообразие стиля пользовательских интерфейсов.

Для многопользовательских систем важную роль играет **нагрузочное** тестирование (тестирование производительности). Рассмотрим его отдельно.

7.6.3. Тестирование производительности

Является важным элементом обеспечения стабильности работы приложений: проблемы производительности, могут стать причиной отказа потребителей от использования ПП. Тестирование производительности направлено на исследование показателей скорости реакции приложения на внешние воздействия при различной по характеру и интенсивности нагрузке и позволяет определить, как быстро работает система или её часть в данных условиях, какой элемент нагрузки или часть системы негативно влияет на производительность.

Основными показателями производительности приложения являются:

1. ***Время выполнения запроса*** (request response time, ms). Один из самых главных показателей производительности приложения. Может быть измерено на серверной стороне, как время обработки запроса сервером, и на клиентской время, которое требуется на сериализацию/десериализацию, пересылку и обработку запроса).

2. ***Потребление ресурсов центрального процессора*** (CPU, %). Показывает, какая часть заданного интервала времени потрачена процессором на вычисления для выбранного процесса.

3. ***Потребление оперативной памяти*** (Memory usage, Mb). Метрика, показывает количество памяти, использованной приложением. При работе приложения память заполняется ссылками на используемые объекты и периодически нуждается в очистке. На время, требующееся для очистки памяти, доступ процесса к страницам выделенной памяти может быть заблокирован, что может повлиять на конечное время обработки этим процессом данных.

4. Длительность работы с дисковой подсистемой (I/O Wait). Значительное время, затраченное на чтение/запись может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления CPU и времени отклика.

5. Потребление сетевых ресурсов. Непосредственно с производительностью приложения не связано, но при недостаточной пропускной способности сетевого канала требуемые показатели производительности системы выдержаны не будут.

Критерием успешности тестирования производительности является *соответствие* характеристик работы приложения *предъявленным требованиям*. В идеале, требования к производительности должны быть сформулированы на стадии разработки требований к создаваемой системе и задокументированы.

Пример требований:

При нагрузке до 100 транзакций в секунду типа «ping» и 10 транзакций в секунду типа «action» должно быть:

- среднее время отклика для транзакций типа «action» не более 2,5 секунд;
- количество отказов не более 1%,
- дисперсия не более 5%,
- сервер приложений должен потреблять не более 50% CPU и не более 1,2 гигабайта ОЗУ,
- система должна расходовать не более трёх соединений с СУБД.

Однако, часто у разработчиков и заказчика нет зафиксированного представления о максимальном времени отклика для заданного числа пользователей. В этом случае первое тестирование производительности будет пробным, основанным на разумных предположениях об ожидаемой нагрузке и потреблении аппаратной части ресурсов. Полученную информацию о производительности используют для принятия решения.

Можно выделить следующие **подвиды тестирования производительности:**

А) **Нагрузочное** тестирование – исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).

Как правило, это автоматизированное тестирование, имитирующее работу определенного количества пользователей на каком-либо общем (разделяемом ими) ресурсе. При этом используется ограниченное число гипотетических сценариев пользовательского поведения и выбранная в соответствии с целями модель нагрузки.

Наиболее распространены постоянная, непрерывно возрастающая нагрузка и нагрузка «С всплесками» (рис. 7.1)

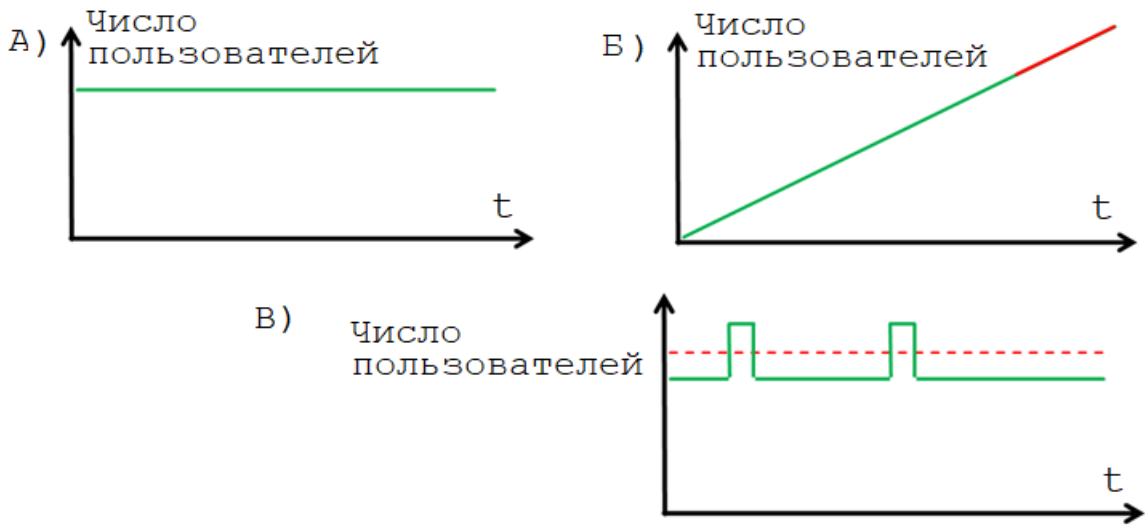

Рисунок 7.1 – Виды нагрузки: А) постоянная, Б) непрерывно возрастающая, В) нагрузка «С всплесками»

1. ***Постоянная нагрузка*** позволяет проверить стабильность приложения. Этот шаблон нагрузки следует использовать при выполнении теста с одинаковым числом пользователей в течение длительного времени. Если при постоянном шаблоне нагрузки указать большое количество пользователей, рекомендуется также указать период разогрева. Указание периода разогрева поможет избежать чрезмерной нагрузки на веб-узел, вызванной сотнями новых пользовательских сеансов одновременно.

2. Цель ***непрерывно возрастающей нагрузки –*** поиск точки насыщения. Пошаговый шаблон является одним из наиболее распространенных и полезных, поскольку он позволяет отследить производительность системы при повышении нагрузки, а значит, определить число пользователей, которых можно поддерживать с приемлемым временем отклика, и число пользователей, при котором производительность станет недостаточной.

3. С помощью нагрузки ***«С всплесками»*** проверяем устойчивость системы к периодическим нагрузкам. Важно, чтобы после всплеска все показатели работы системы вернулись к состоянию до всплеска.

Разработка модели нагрузки должна опираться на:

- список тестируемых операций. В него должны войти операции, критичные с точки зрения бизнеса и с технической точки зрения, которые оказывают реальное влияние на ухудшение производительности;
- информацию об интенсивности выполнения операций;
- зависимость изменения интенсивности выполнения операций от времени.

Б). ***Стрессовое тестирование*** (Stress Testing). Направлено на исследование поведения приложения при экстремальных нагрузках или в ситуациях недоступности значительной части необходимых приложению ресурсов. Подвергая решение перегрузке, что, как правило, означает нагрузку большую, чем та, на которую оно рассчитано, удается обнаружить дефекты, не проявляющие себя в нормальных условиях, а также оценить способность системы к регенерации после прекращения воздействия стресса, оценить деградацию производительности.

Разновидность стрессового тестирования – тестирование на разрушение имеет другие цели и представляет собой крайнюю форму негативного тестирования.

В). ***Объемное тестирование*** (Volume Testing). Направлено на оценку производительности при обработке различных (как правило, значительных) объёмов данных.

Г) Тестирование масштабируемости (scalability testing) – исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов.

Д) Конкурентное тестирование (concurrency testing) – оценивают влияние на производительность конкуренции за ресурсы (базу данных, память, канал передачи данных, дисковую подсистему и т. д.) между большим количеством одновременно поступающих запросов. Может исследоваться также работа многопоточных приложений и корректность синхронизации действий, производимых в разных потоках.

Е). ***Тестирование стабильности или надежности*** (Stability / Reliability Testing). Проверяется работоспособность приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки. Акцент делается на отсутствие утечек памяти, перезапусков серверов под нагрузкой и другие аспекты, влияющие именно на стабильность работы.

В рамках тестирования производительности могут применяться тестирование использования ресурсов, тестирование восстанавливаемости, тестирование отказоустойчивости и другие.

7.7. Другие классификации

По способу формирования наборов тестовых данных выделяют тестирование:

а) **стохастическое** (случайное): исходные тестовые данные берутся случайным образом, как правило, с использованием статистического распределения;

б) **детерминированное**. Тестирование ПП проводится на наборах данных, сформированных в соответствии с заданными критериями. Применяют различные техники тест-дизайна. По ***стратегии*** проектирования тестов детерминированное тестирование делят на тестирование белого, серого (применяется комбинация стратегий белого и черного ящика, еще называется оптимальным) и черного ящика.

По направленности тестирования его делят на:

а) **статическое.** Направлено на анализ исходного кода, который вычитывается вручную, либо анализируется специальными инструментами. При статическом тестировании программный код не выполняется;

б) **динамическое.** Динамическое тестирование – процедура выявления дефектов в процессе работы программного продукта. В зависимости от того, какие наборы тестовых данных при этом используются, различают функциональное (основано на принципе «черного ящика») и структурное (или тестирование «белого ящика») тестирование.

По принципам работы с приложением бывает:

а) **Позитивное** тестирование. Исследуется работа приложения в идеальных условиях: пользователи строго следуют инструкции, не допускают неправильных действий, ошибок при вводе данных и т.п. Позитивные тесты можно объединять. Это уменьшит время тестирования, но, если ошибки всё-таки есть, может усложнить их диагностику.

б) **Негативное** тестирование. Используются данные, потенциально приводящие к ошибкам, выполняются некорректные операции, злоумышленники осознанно «ломают» приложение, в среде работы приложения возникают проблемы и т.д. Количество негативных тестов может оказаться значительным, а объединять их нежелательно: совокупность неверных входных данных может скрыть ошибку.

Например, проверяется правильность умножения вещественных чисел a и b. Подготовили тест: $a=b=(-1)^{1/2}$ – невещественные числа (и программа позволила их этим переменным присвоить). Получили –1. Ошибка о неверных исходных данных не проявилась.