**10. Системное тестирование.**

Системное тестирование - один из самых сложных видов тестирования. На этом этапе проводится не только функциональное тестирование, но и оценка характеристик качества системы - ее устойчивости, надежности, безопасности и производительности. На этом этапе выявляются многие проблемы внешних интерфейсов системы, связанные с неверным взаимодействием с другими системами, аппаратным обеспечением, неверным распределением памяти, отсутствием корректного освобождения ресурсов и т.п.

После завершения системного тестирования разработка переходит в фазу приемо-сдаточных испытаний (для программных систем, разрабатываемых на заказ) или в фазу альфа- и бета-тестирования (для программных систем общего применения).

Поскольку системное тестирование - процесс, требующих значительных ресурсов, для его проведения часто выделяют отдельный коллектив тестировщиков, а зачастую системное тестирование выполняется организацией, которая не связана с коллективом разработчиков и тестировщиков, выполнявших работы на предыдущих этапах тестирования. При этом необходимо отметить, что при разработке некоторых типов программного обеспечения (например, авиационного бортового) требование независимого тестирования на всех этапах разработки является обязательным.

Системное тестирование проводится в несколько фаз, на каждой из которых проверяется один из аспектов поведения системы, т.е. проводится один из типов системного тестирования. Все эти фазы могут протекать одновременно или последовательно.

Принято выделять следующие виды системного тестирования:

* функциональное тестирование;
* тестирование производительности;
* нагрузочное или стрессовое тестирование;
* тестирование конфигурации;
* тестирование безопасности;
* тестирование надежности и восстановления после сбоев;
* тестирование удобства использования.

В ходе системного тестирования проводятся далеко не все из перечисленных видов тестирования - конкретный их набор зависит от тестируемой системы.

Исходной информацией для проведения перечисленных видов тестирования являются два класса требований: функциональные и нефункциональные. Функциональные требования явно описывают, что система должна делать и какие выполнять преобразования входных значений в выходные. Нефункциональные требования определяют свойства системы, напрямую не связанные с ее функциональностью. Примером таких свойств может служить время отклика на запрос пользователя (например, не более 2 секунд), время бесперебойной работы (например, не менее 10000 часов между двумя сбоями), количество ошибок, которые допускает начинающий пользователь за первую неделю работы (не более 100), и т.п.

Функциональное тестирование. Данный вид тестирования предназначен для доказательства того, что вся система в целом ведет себя в соответствии с ожиданиями пользователя, формализованными в виде системных требований. В ходе данного вида тестирования проверяются все функции системы с точки зрения ее пользователей (как пользователей-людей, так и "пользователей" - других программных систем). Система при функциональном тестировании рассматривается как черный ящик, поэтому в данном случае полезно использовать классы эквивалентности. Критерием полноты тестирования в данном случае будет полнота покрытия тестами системных функциональных требований (или системных тест-требований) и полнота тестирования классов эквивалентности, а именно:

* все функциональные требования должны быть протестированы;
* все классы допустимых входных данных должны корректно обрабатываться системой;
* все классы недопустимых входных данных должны быть отброшены системой, при этом не должна нарушаться стабильность ее работы;
* в тестовых примерах должны генерироваться все возможные классы выходных данных системы;
* во время тестирования система должна побывать во всех своих внутренних состояниях, пройдя при этом по всем возможным переходам между состояниями.

Результаты системного тестирования протоколируются и анализируются совершенно аналогично тому, как это делается для модульного и интеграционного тестирования. Основная сложность здесь заключается в локализации дефектов в программном коде системы и определении зависимостей одних дефектов от других (эффект "четного числа ошибок").

Тестирование производительности. Данный вид тестирования направлен на определение того, что система обеспечивает должный уровень производительности при обработке пользовательских запросов. Тестирование производительности выполняется при различных уровнях нагрузки на систему, на различных конфигурациях оборудования. Выделяют три основных фактора, влияющие на производительность системы: количество поддерживаемых системой потоков (например, пользовательских сессий), количество свободных системных ресурсов, количество свободных аппаратных ресурсов.

Тестирование производительности позволяет выявлять узкие места в системе, которые проявляются в условиях повышенной нагрузки или нехватки системных ресурсов. В этом случае по результатам тестирования проводится доработка системы, изменяются алгоритмы выделения и распределения ресурсов системы.

Все требования, относящиеся к производительности системы, должны быть четко определены и обязательно должны включать в себя числовые оценки параметров производительности. Т.е., например, требование "Система должна иметь приемлемое время отклика на запрос пользователя" является непригодным для тестирования. Напротив, требование "Время отклика на запрос пользователя не должно превышать 2 секунды" может быть протестировано.

То же самое относится и к результатам тестирования производительности. В отчетах по данному виду тестирования сохраняют такие показатели, как загрузка аппаратного и системного программного обеспечения (количество циклов процессора, выделенной памяти, количество свободных системных ресурсов и т.п.). Также важны скоростные характеристики тестируемой системы (количество обработанных в единицу времени запросов, временные интервалы между началом обработки каждого последующего запроса, равномерность времени отклика в разные моменты времени и т.п.).

Для проведения тестирования производительности требуется наличие генератора запросов, который подает на вход системы поток данных, типичных для сеанса работы с ней. Тестовое окружение должно включать в себя кроме программной компоненты еще и аппаратную, причем на таком тестовом стенде должна существовать возможность моделирования различного уровня доступных ресурсов.

Стрессовое тестирование. Стрессовое тестирование имеет много общего с тестированием производительности, однако его основная задача - не определить производительность системы, а оценить производительность и    устойчивость системы в случае, когда для своей работы она выделяет максимально доступное количество ресурсов, либо когда она работает в условиях их критической нехватки. Основная цель стрессового тестирования - вывести систему из строя, определить те условия, при которых она не сможет далее нормально функционировать. Для проведения стрессового тестирования используются те же самые инструменты, что и для тестирования производительности. Однако, например, генератор нагрузки при стрессовом тестировании должен генерировать запросы пользователей с максимально возможной скоростью либо генерировать данные запросов таким образом, чтобы они были максимально возможными по объему обработки.

Стрессовое тестирование очень важно при тестировании web-систем и систем с открытым доступом, уровень нагрузки на которые зачастую очень сложно прогнозировать.

Тестирование конфигурации. Большинство программных систем массового назначения предназначено для использования на самом разном оборудовании. Несмотря на то, что в настоящее время особенности реализации периферийных устройств скрываются драйверами операционных систем, которые имеют унифицированный с точки зрения прикладных систем интерфейс, проблемы совместимости (как программной, так и аппаратной) все равно существуют.

В ходе тестирования конфигурации проверяется, что программная система корректно работает на всем поддерживаемом аппаратном обеспечении и совместно с другими программными системами. Необходимо также проверять, что система продолжает стабильно работать при горячей замене любого поддерживаемого устройства на аналогичное. При этом система не должна давать сбоев ни в момент замены устройства, ни после начала работы с новым устройством.

Также необходимо проверять, что система корректно обрабатывает проблемы, возникающие в оборудовании, как штатные (например, сигнал конца бумаги в принтере), так и нештатные (сбой питания).

Тестирование безопасности. Если программная система предназначена для хранения или обработки данных, содержимое которых представляет собой тайну определенного рода (личную, коммерческую, государственную и т.п.), то к свойствам системы, обеспечивающим сохранение этой тайны, будут предъявляться повышенные требования. Эти требования должны быть проверены при тестировании безопасности системы. В ходе этого тестирования проверяется, что информация не теряется, не повреждается, ее невозможно подменить, а также к ней невозможно получить несанкционированный доступ, в том числе при помощи использования уязвимостей в самой программной системе.

В отечественной практике принято проводить сертификацию программных систем, предназначенных для хранения данных для служебного пользования, секретных, совершенно секретных и совершенно секретных особой важности. Существует ряд отечественных стандартов, регламентирующих свойства программных систем по обеспечению необходимого уровня безопасности и по отсутствию недокументированных возможностей ("закладок"), которые могут быть использованы злоумышленником для несанкционированного доступа к данным.

Кроме того, существует международный стандарт Common Criteria, также регламентирующий вопросы защиты информации в программных системах.

Несмотря на то, что сертификация - процесс, следующий за верификацией, требования этих стандартов могут быть использованы и при тестировании системы. Выделяют следующие группы свойств программной системы, подлежащие проверке:

* разграничение и контроль доступа - предотвращение доступа к "чужой" информации;
* очистка и защита памяти - предотвращение доступа к остаточной информации после удаления объектов из памяти;
* маркировка и защита информации, передаваемой во внешний мир - сохранение уровня секретности даже вне системы;
* идентификация и аутентификация - предоставление доступа только санкционированным пользователям и отказ в доступе всем остальным;
* регистрация (аудит событий) - регистрация в специальном журнале всех событий системы, связанных с безопасностью для последующего анализа;
* гарантии проектирования и архитектуры - система должна быть спроектирована таким образом, чтобы гарантировать защищенность информации с определенным уровнем уверенности;
* тестирование - все функции по обеспечению безопасности должны быть протестированы во всех режимах;
* целостность и восстановление средств защиты - система должна иметь средства контроля корректности всех правил разграничения доступа и системы безопасности в целом, а также средства их восстановления при сбое;
* документация разработчика, администратора и пользователя - все средства системы по обеспечению безопасности должны быть описаны в соответствующих руководствах.

При разработке и верификации программной системы, которая будет подвергаться последующей сертификации, работы по сертификации должны включать в себя проверку всех перечисленных свойств.

Тестирование надежности и восстановления после сбоев. Для корректной работы системы в любой ситуации необходимо удостовериться в том, что она восстанавливает свою функциональность и продолжает корректно работать после любой проблемы, прервавшей ее работу. При тестировании восстановления после сбоев имитируются сбои оборудования или окружающего программного обеспечения либо сбои программной системы, вызванные внешними факторами. При анализе поведения системы в этом случае необходимо обращать внимание на два фактора - минимизацию потерь данных в результате сбоя и минимизацию времени между сбоем и продолжением нормального функционирования системы

Тестирование удобства использования. Отдельная группа нефункциональных требований - требования к удобству использования пользовательского интерфейса системы.

В результате выполнения всех рассмотренных выше видов тестирования делается заключение о функциональности и свойствах системы, после чего узкие места системы дорабатываются до реализации необходимой функциональности или до достижения системой необходимых свойств.

**30. Тестовое окружение: драйверы и заглушки, генераторы сигналов (событийно-управляемый код).**

Окружение (environment) -  это среда, место, машина, на которой находится приложение, сайт. Например, хостинг - это место, где хранится сайт. Тестовое окружение (test environment) - это то место, где тестировщик тестирует сайт, приложение, программу. Локальное окружение - это всего одна машина, на которой разрабатывают, тестируют приложение. В процессе разработки, как правило, существует несколько тестовых окружений. Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем, часто реализованной не на том языке программирования, на котором написана система), оно тоже должно быть протестировано. Целью тестирования тестового окружения является доказательство того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

Тестовое окружение для программного кода на структурных языках программирования состоит из двух компонентов - драйвера, который обеспечивает запуск и выполнение тестируемого модуля, и заглушек, которые моделируют функции, вызываемые из данного модуля. Разработка тестового драйвера представляет собой отдельную задачу тестирования, сам драйвер должен быть протестирован, чтобы исключить неверное тестирование. Драйвер и заглушки могут иметь различные уровни сложности, требуемый уровень сложности выбирается в зависимости от сложности тестируемого модуля и уровня тестирования. Так, драйвер может выполнять следующие функции:

* вызов тестируемого модуля
* передача в тестируемый модуль входных значений и прием результатов
* вывод выходных значений
* протоколирование процесса тестирования и ключевых точек программы

Заглушки могут выполнять следующие функции:

* не производить никаких действий (такие заглушки нужны для корректной сборки тестируемого модуля);
* выводить сообщения о том, что заглушка была вызвана;
* выводить сообщения со значениями параметров, переданных в функцию;
* возвращать значение, заранее заданное во входных параметрах теста;
* выводить значение, заранее заданное во входных параметрах теста;
* принимать от тестируемого ПО значения и передавать их в драйвер.

Для тестирования программного кода, написанного на процедурном языке программирования, используются драйверы, представляющие собой программу с точкой входа (например, функцией main() ), функциями запуска тестируемого модуля и функциями сбора результатов. Обычно драйвер имеет как минимум одну функцию - точку входа, которой передается управление при его вызове.

Функции-заглушки могут помещаться в тот же файл исходного кода, что и основной текст драйвера. Имена и параметры заглушек должны совпадать с именами и параметрами "заглушаемых" функций реальной системы. Это требование важно не столько с точки зрения корректной сборки системы (при сборке тестового драйвера и тестируемого ПО может использоваться приведение типов), сколько для того, чтобы максимально точно моделировать поведение реальной системы по передаче данных.

Генераторы сигналов (событийно‑управляемый код). Монолитные  программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных.

Значительная часть сложных программ в настоящее время использует ту или иную форму межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла следующие этапы:

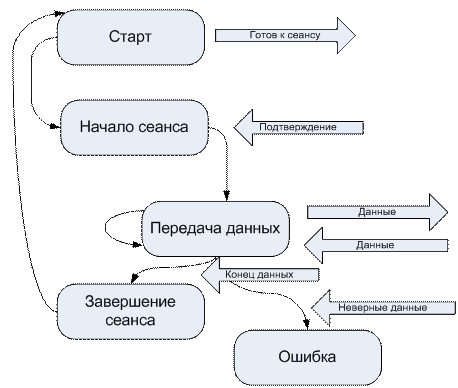
1. Монолитные программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных. При запуске таких программ образуется один процесс, который выполняет всю необходимую работу.

2. Модульные программы, которые состоят из отдельных программных модулей с четко определенными интерфейсами вызовов. Объединение модулей в программу может происходить либо на этапе сборки исполняемого файла (статическая сборка или static linking), либо на этапе выполнения программы (динамическая сборка или dynamic linking). Преимущество модульных программ заключается в достижении некоторого уровня универсальности – один модуль может быть заменен другим. Однако, модульная программа все равно представляет собой один процесс, а данные, необходимые для решения задачи, передаются внутри процесса как параметры функций.

3. Программы, использующие межпроцессное взаимодействие. Такие программы образуют программный комплекс, предназначенный для решения общей задачи. Каждая запущенная программа образует один или более процессов. Каждый из процессов использует для решения задачи либо свои собственные данные и обменивается с другими процессами только результатом своей работы, либо работает с общей областью данных, разделяемых между разными процессами. Для решения особо сложных задач процессы могут быть запущены на разных физических компьютерах и взаимодействовать через сеть. Преимущество использования межпроцессного взаимодействия заключается в еще большей универсальности – взаимодействующие процессы могут быть заменены независимо друг от друга при сохранении интерфейса взаимодействия. Другое преимущество состоит в том, что вычислительная нагрузка распределяется между процессами. Это позволяет операционной системе управлять приоритетами выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам. При выполнении многих процессов, решающих общую задачу, используются несколько типичных механизмов взаимодействия между ними, направленных на решение следующих задач:

* + передача данных от одного процесса к другому;
  + совместное использование одних и тех же данных несколькими процессами;
  + извещения об изменении состояния процессов.

Во всех этих случаях типичная структура каждого процесса представляет собой конечный автомат с набором состояний и переходов между ними. Находясь в определенном состоянии, процесс выполняет обработку данных, при переходе между состояниями – пересылает данные другим процессам или принимает данные от них. Для моделирования конечных автоматов используются stateflow или SDL-диаграммы, акцент в которых делается соответственно на условиях перехода между состояниями и пересылаемыми данными. Так, на рисунке показана схема процесса приема/передачи данных. Закругленными прямоугольниками указаны состояния процесса, тонкими стрелками – переходы между состояниями, большими стрелками – пересылаемые данные. Находясь в состоянии «Старт» процесс посылает во внешний мир (или процессу, с которым он обменивается данными) сообщение о своей готовности к началу сеанса передачи данных. После получения от второго процесса подтверждения о готовности начинается сеанс обмена данными. В случае поступления сообщения о конце данных происходит завершение сеанса и переход в стартовое состояние. В случае поступления неверных данных (например, неправильного формата или с неверной контрольной суммой), процесс переходит в состояние «Ошибка», выйти из которого возможно только завершением и перезапуском процесса.



Пример конечного автомата процесса приема-передачи данных.

Тестовое окружение для такого процесса также должно иметь структуру конечного автомата и пересылать данные в том же формате, что и тестируемый процесс.  Целью тестирования в данном случае будет показать, что процесс обрабатывает принимаемые данные в соответствии с требованиями, форматы передаваемых данных корректны, а также что процесс во время своей работы действительно проходит все состояния конечного автомата, моделирующего его поведение.

**94. Нагрузочное тестирование.**

Нагрузочное тестирование (англ. *load testing*) — подвид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Для исследования времени отклика системы на высоких или пиковых нагрузках производится стресс-тестирование, при котором создаваемая на систему нагрузка превышает нормальные сценарии её использования. Не существует чёткой границы между нагрузочным и стресс-тестированием, однако эти понятия не стоит смешивать, так как эти виды тестирования отвечают на разные бизнес-вопросы и используют различную методологию.

В общем случае под нагрузочным тестированием понимается практика моделирования ожидаемого использования приложения с помощью эмуляции работы нескольких пользователей одновременно. Таким образом, подобное тестирование больше всего подходит для многопользовательских систем, чаще — использующих клиент-серверную архитектуру (например, веб-серверов). Однако и другие типы систем ПО могут быть протестированы подобным способом. Например, текстовый или графический редактор можно заставить прочесть очень большой документ; а финансовый пакет — сгенерировать отчёт на основе данных за несколько лет. Наиболее адекватно спроектированный нагрузочный тест даёт более точные результаты.

В идеальном случае в качестве критериев успешности нагрузочного тестирования выступают требования к производительности системы, которые формулируются и документируются на стадии разработки функциональных требований к системе до начала программирования основных архитектурных решений. Однако часто бывает так, что такие требования не были четко сформулированы или не были сформулированы вовсе. В этом случае первое нагрузочное тестирование будет являться *пробным* (англ. *exploratory load testing*) и основываться на разумных предположениях об ожидаемой нагрузке и потреблении аппаратной части ресурсов.

Одним из оптимальных подходов в использовании нагрузочного тестирования для измерений производительности системы является тестирование на стадии ранней разработки. Нагрузочное тестирование на первых стадиях готовности архитектурного решения с целью определить его состоятельность называется 'proof-of-concept' тестированием.

Уникальность запросов — даже сформировав реалистичный сценарий работы с системой на основе статистики её использования, необходимо понимать, что всегда найдутся исключения из этого сценария.

Время отклика системы — в общем случае время отклика системы подчиняется функции нормального распределения. В частности, это означает, что, имея достаточное количество измерений, можно определить вероятность с которой отклик системы на запрос попадёт в тот или иной интервал времени.

Зависимость времени отклика системы от степени распределённости этой системы — дисперсия нормального распределения времени отклика системы на запрос пропорциональна отношению количества узлов системы, параллельно обрабатывающих такие запросы и количеству запросов, приходящихся на каждый узел. То есть, на разброс значений времени отклика системы влияет одновременно количество запросов приходящихся на каждый узел системы и само количество узлов, каждый из которых добавляет некоторую случайную величину задержки при обработке запросов.

Разброс времени отклика системы — при достаточно большом количестве измерений величины времени обработки запроса в любой системе всегда найдутся запросы, время обработки которых превышает определённые в требованиях максимумы; причем, чем больше суммарное время проведения эксперимента тем выше окажутся новые максимумы. Этот факт учитывается при формировании требований к производительности системы, а также при проведении регулярного нагрузочного тестирования.

Точность воспроизведения профилей нагрузки — необходимая точность воспроизведения профилей нагрузки тем дороже, чем больше компонент содержит система. Часто невозможно учесть все аспекты профиля нагрузки для сложных систем, так как чем сложнее система, тем больше времени будет затрачено на проектирование, программирование и поддержку адекватного профиля нагрузки для неё, что не всегда является необходимостью. Оптимальный подход в данном случае заключается в балансировании между стоимостью разработки теста и покрытием функциональности системы, в результате которого появляются допущения о влиянии на общую производительность той или иной части тестируемой системы.

Одним из результатов, получаемых при нагрузочном тестировании и используемых в дальнейшем для анализа, являются показатели производительности приложения.

Потребление ресурсов центрального процессора — метрика, показывающая сколько времени из заданного определённого интервала было потрачено процессором на вычисления для выбранного процесса. В современных системах важным фактором является способность процесса работать в нескольких потоках, для того, чтобы процессор мог производить вычисления параллельно. Анализ истории потребления ресурсов процессора может объяснять влияние на общую производительность системы потоков обрабатываемых данных, конфигурации приложения и операционной системы, многопоточности вычислений, и других факторов.

Потребление оперативной памяти — метрика, показывающая количество памяти, использованной приложением. Использованная память делится на несколько категорий:

виртуальная память — объём виртуального адресного пространства, которое использует процесс. Этот объём подразумевает, как использование соответствующего дискового пространства так и оперативной памяти. Система виртуальной памяти гарантирует, что потоки одного процесса не получат доступа к памяти принадлежащей другому процессу;

приватная память — объём адресного пространства, занятого процессом и не разделяемого с другими процессами;

рабочее множество — набор страниц памяти, недавно использованных процессом. В случае, когда свободной памяти достаточно, страницы остаются в наборе, даже если они не используются. В случае, когда свободной памяти остаётся мало, использованные страницы перемещаются из оперативной памяти на накопитель;

разделяемая память — объём используемой процессом физической памяти, которая может использоваться совместно с другими процессами. Хотя память выделенная процессу должна быть изолированной, процессам, иногда, необходимо иметь возможность обмениваться информацией. Общая память является самым быстрым способом межпроцессного взаимодействия.

При работе приложения память заполняется ссылками на объекты, которые, в случае неиспользования, могут быть очищены специальным автоматическим процессом, называемым сборщиком мусора. Время затрачиваемое процессором на очистку памяти таким способом может быть значительным, в случае, когда процесс занял всю доступную память (в Java — так называемый «постоянный Full GC») или когда процессу выделены большие объёмы памяти, нуждающиеся 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.

Потребление сетевых ресурсов — метрика, не связана непосредственно с производительностью приложения, однако её показатели могут указывать на пределы производительности системы в целом.

Показатели подсистемы ввода-вывода могут значительно влиять на производительность системы, поэтому сбор статистики по работе с накопителями может помогать выявлять узкие места в этой области. Большое количество чтений или записей может приводить к простаиванию процессора в ожидании обработки данных с диска и в итоге увеличению потребления процессорных ресурсов и увеличению времени отклика.

Время выполнения запроса приложением остаётся одним из самых главных показателей производительности системы или приложения. Это время может быть измерено на серверной стороне, как показатель времени, которое требуется серверной части для обработки запроса; так и на клиентской, как показатель полного времени, которое требуется на сериализацию и десериализацию, пересылку и обработку запроса. При этом не каждое приложение для тестирования производительности может измерить оба этих времени.

**111. Понятие решения об автоматизации тестирования.**

Автоматизация тестирования — это методика автоматической проверки и утверждения программного продукта, например веб-приложения. Она позволяет удостовериться, что продукт соответствует установленным стандартам качества в отношении оформления кода, функциональности (бизнес-логики) и удобства пользователя.

Существуют разные методики тестирования.

**Модульное тестирование**: проверка работы отдельных модулей кода, например функций.

**Интеграционное тестирование**: проверка способности нескольких фрагментов кода работать вместе без нежелательных последствий.

**Сквозное тестирование**: проверка приложения на соответствие ожиданиям пользователя.

**Глубокое тестирование**: неструктурированный подход к анализу многочисленных компонентов приложения с точки зрения пользователя для выявления функциональных и визуальных проблем.

Часто эти типы тестирования изображают в виде пирамиды. По мере подъема по ступеням количество тестов уменьшается, а стоимость их создания и запуска возрастает.

Раньше все тестирование в пирамиде выполнялось вручную. До создания инструментов автоматического тестирования это был медленный, затратный и ненадежный процесс.

Сегодня почти все модульные тесты полностью автоматизированы и автоматическое модульное тестирование считается стандартом. В значительной степени автоматизированы и интеграционные тесты. Если же автоматизация не предусмотрена, вместо такого тестирования выполняют ручное сквозное тестирование. Сегодня усилия по автоматизации в большей степени сосредоточены на сквозных тестах, за счет чего предполагается снизить необходимость интеграционного тестирования.

Несмотря на то что инструменты автоматизации появились более десяти лет назад, многие из них все еще требуют навыков программирования. Часто результат их работы нестабилен и ненадежен, а при большом масштабе слишком возрастают расходы на техническое обслуживание и устранение неполадок. В итоге многие команды создают собственные системы автоматизации тестирования. Это затрудняет и растягивает во времени адаптацию новых участников команды из-за сложного обучения. Кроме того, собственные системы автоматизации тоже нуждаются в техническом обслуживании и оптимизации, чтобы соответствовать актуальному стеку технологий. Как следствие, большая часть сквозных тестов до сих пор выполнялась вручную.

По мере того как организации совершенствуют методики работы, возникает потребность в автоматизации тестирования на протяжении всего цикла разработки. Среды необходимо стандартизировать, а развертывание на этапе контроля качества — автоматизировать. Операции предварительного тестирования, очистки, последующего тестирования и т. д. понадобится автоматизировать и согласовать с циклом непрерывной интеграции.