V-образная модель тестирования

Тестирование появляется с самого начала проекта

общее планирование

общая информация об объекте, сроках, сложности тестирования и тд

пользовательские требования

задачи, которые пользователь может выполнить с помощью разрабатываемой системы (реакцию системы на действия пользователя, сценарии работы пользователя)

системные требования

поведение и свойства системы

функциональные

описывают поведение системы, т.е. её действия (вычисления, преобразования, проверки, обработку и т.д.) В контексте проектирования функциональные требования в основном или «вот-вот» на дизайн системы.
Стоит помнить, что к поведению системы относится не только то, что система должна делать, но и то, что она не должна делать (например: «приложение не должно выгружать из оперативной памяти фоновые документы в течение 30 минут с момента выполнения с ними последней операции).

нефункциональные

описывают свойства системы (удобство использования, безопасность, надёжность, расширяемость и т.д.), которыми она должна обладать при реализации своего поведения. Здесь приводится более технически и детальное описание атрибутов качества. В контексте проектирования нефункциональные требования в основном влияют на архитектуру системы.

техническая архитектура

взаимодействие персонала и данных и в системе

Архитектура приложения-это абстрактное представление приложений, подсистем, служб и структур оркестровки, а также их взаимосвязи друг с другом и с данными. Документирование архитектуры программного обеспечения (ПО) упрощает процесс коммуникации между разработчиками, позволяет зафиксировать принятые проектные решения и предоставить информацию о них эксплуатационному персоналу системы, повторно использовать компоненты и шаблоны проекта в других.

Архитектура, управляемая событиями (EDA)

Этот вид архитектуры подходит в том случае, если процесс работы приложения распадается на несколько шагов, которые могут выполняется отдельными обработчиками. Основными компонентами являются «фильтр» (filter) и «канал» (pipe). Иногда дополнительно выделяют «источник данных» (data source) и «потребитель данных» (data sink).

Каждый поток обработки данных – это серия чередующихся фильтров и каналов, начинающаяся источником данных и заканчивающаяся их потребителем. Каналы обеспечивают передачу данных и синхронизацию. Фильтр же принимает на вход данные и обрабатывает их, трансформируя в некое иное представление, а затем передает дальше.

«Каналы и фильтры»

Является одной из самых известных архитектур, в которой каждый слой выполняет определенную функцию. В зависимости от ваших нужд вы можете реализовать любое количество уровней, но слишком большое их количество приведет к чрезмерному усложнению системы. Часто выделяют три основных уровня: уровень представления, уровень логики и уровень данных, хотя не обязательно знать, что делают его соседи. Здесь проявляется такое свойство как разделение ответственности. Если все три слоя являются закрытыми, то запрос пользователя к верхнему уровню инициирует цепочку обращений с верхнего уровня до самого нижнего. В этом случае уровень представления отвечает за пользовательский интерфейс и отображение данных для пользователя и ничего не знает о существовании физического хранилища данных. Ничего о существовании базы данных не знает и уровень логики – его «беспокоить» только правила бизнес-логики. Доступ к базе данных имеет лишь через уровень управления данными.

Достоинствами применения такой архитектуры являются простота разработки (в основном из-за того, что этот вид архитектуры всем знаком) и простота тестирования. Среди недостатков можно выделить возможные сложности с производительностью и масштабированием – всему виной необходимость прохождения запросов и данных по всем уровням (опять же, в том случае, если все слои являются закрытыми).

Многоуровневая архитектура

Архитектура, управляемая событиями (EDA)

в этой схеме существует два варианта событий: инициализирующее событие и событие, на которое реагируют обработчики. Обработчики являются изолированными независимыми компонентами, отвечающими (в идеале) за какую-нибудь одну задачу, и содержат бизнес-логику, необходимую для работы.

Микроядерная архитектура

Паттерн состоит из двух компонентов: основной системы (ядра) и плагинов. Ядро содержит минимум бизнес-логики, но руководит загрузкой и запуском необходимых плагинов. Таким образом, плагины оказываются несвязанными друг с другом.

Поскольку плагины могут разрабатываться независимо друг от друга, такие системы обладают очень высокой гибкостью и, как следствие, легко тестируются. Производительность приложения, построенного на основе такой архитектуры, напрямую зависит от количества подключенных и активных модулей.

Микросервисная архитектура

приложение разбивается на множество небольших сервисов, называемых микросервисами. Каждый микросервис включает в себя бизнес-логику и представляет собой совершенно независимый компонент. Сервисы одной системы могут быть написаны на различных языках программирования и общаться друг с другом, используя различные протоколы.

Поскольку каждый микросервис является отдельным проектом, вы можете распределить работу над ними между командами разработчиков, то есть над системой могут одновременно трудиться несколько десятков программистов. Микросервисная архитектура позволяет с легкостью масштабировать приложение – если вам понадобилось внедрить новую функцию (развертывать каждый микросервис можно по отдельности), просто напишите новый сервис, а если какой-то функцией никто не пользуется – отключите сервис.

инсталляционное тестирование

проверка установки, удаления и обновления ПО

Проверяется корректность инсталляции и деинсталляции программного продукта в среде максимально приближенной к эксплуатационной. Позволяет проверить возможность инсталляции/деинсталляции продукта при различных условиях: новая инсталляция, обновление системы (upgrade), инсталляция по умолчанию, полная инсталляция; Выборочная инсталляция.

интеграция и модульные тесты

объединение блоков программы и их проверка малыми частями

Юнит-тест (unit test), или модульный тест, — это программа, которая проверяет работу небольшой части кода. Разработчики регулярно обновляют сайты и приложения, добавляют фичи, рефакторят код и вносят правки, а затем проверяют, как всё работает.
Тестировать систему целиком после каждого обновления — довольно мутотно и неэффективно. Поэтому обновлённые или исправленные части кода проганяют через юнит-тесты.

обмен данными между системами с возможной последующей их обработкой.Смысл интеграции заключается в том, чтобы данные, которые пользователь вводит в одну систему, автоматически переносились в другую. Продукт, в который пользователь вводит данные, называется источник. А получатель данных, соответственно, приемник.

Достаточно часто данные переносят в обе стороны, например, после преобразования в системе-приемнике результаты отправляются обратно в источник. А потому интеграция бывает как односторонней, так и двухсторонней.

детализированный дизайн

визуальный облик программы, который подстроен под пользовательское поведение

разработка и отладка

этап разработки продукта и обнаружение, локализация и устранение ошибок

итоговая отчетность

документ, содержащий информацию о выполненных действиях, результатах проведённой работы

В финальном отчете важно показать общий взгляд на проделанную работу (в контексте установленных метрик) и эволюцию продукта. Так же, надо дать исчерпывающую информацию о статусе продукта в данный момент (количество оставшихся неисправленных ошибок, полностью ли протестирован продукт или требуется дополнительный цикл тестирования, оценка возможности выпуска продукта во «внешний мир» и т.д).

приемочное тестирование

система проверяется на приемлемость — готовность к передаче заказчику (клиентам). Оценивается соответствие продукта бизнес-требованиям и требованиям пользователей.

проверка соответствия требованиям пользователей и бизнес-требованиям. проверка соответствия критериям приемки. После завершения приемочного тестирования пользователи/заказчики решают, принимать ли систему в пользование.

системное тестирование

процесс тестирования системы, на котором проводится не только функциональное тестирование, но и оценка характеристик качества системы — ее устойчивости, надежности, безопасности и производительности.

Выполняемое на полностью интегрированной системе с целью оценить качество выполнения системных требований.

Системному тестированию подлежат компоненты, прошедшие интеграционное тестирование.

Цель системного тестирования: определить несовместимые/плохо совместимые интегрированные модули, а также оценить качество системы как единого целого (почему и называется системным).

Результатом системного тестирования является подтвержденное ожидаемое поведение компонентов и всей системы.

Этот тип тестирования сосредоточен на проверке системных спецификаций и/или функциональных требований, в зависимости от проекта.

Оно проверяет дизайн и поведение системы, часто выходя за пределы, прописанные в требованиях.

Обычно выполняется QA-командой, независимой от разработчиков, что помогает сохранить объективность.

Системное тестирование может быть как функциональным, так и нефункциональным.

Системное тестирование типологически относится к черному ящику.

Порядок выполнения системного тестирования: после интеграционного и перед приемочным

Некоторые разновидности системного тестирования
Тестирование производительности, проверка скорости, расширяемость (масштабируемость), стабильности и надежности продукта
Нагрузочное тестирование: проверка поведения системы под большой нагрузкой
Стресс-тестирование: проверка «точки невозврата» системы, подвергая ее «стрессу» экстремальной нагрузкой / недостатком ресурсов (памяти)
Тестирование масштабируемости: проверка продуктивности системы в плане ее готовности к расширению, «подстройке» к постепенному росту