Тестирование

ФИТ Лекция №10. 25’

Чек-листы самопроверки для разработчиков

Чек-листы самопроверки для разработчиков

Это еще одна shift-left практика тестирования:
часть работы выполняет тестировщик (готовит чек-лист самопроверки), а другую часть - разработчик (выполняет happy path тесты).

В этом случае «сдвиг» по времени обеспечен подготовкой чек-листа **до** начала разработки и самопроверкой разработчика **до** передачи продукта в тестирование.

Этим мы решаем сразу две проблемы:

Исключаем отладку через тестирование и сокращение цикла возврата ПО на доработку

Сокращаем время тестирования в целом, т.к. разработчик самостоятельно отлаживает и оперативно правит критичные баги не прогоняя каждый баг через цикл внешнего тестирования через QA-инженера

Чек-листы самопроверки для разработчиков

Чек-лист может быть оформлен в любой системе или документе, главное, что всем было понятно какой его flow работы.

Важное для руководителя проекта:
если процесс самопроверок введён, то он должен легко контролироваться.

Пример организации такой работы:

- У нас есть PBI для нового функционала
- QA-инженер привязывает к ней подзадачу с чек-листом самопроверки.
- Разработчик пишет код, по прикрепленному чек-листу проходит тесты, ставит задаче на самопроверку статус «resolved»
- Для руководителя проекта и QA прозрачно следующее: можно быстро оценить все PBI, по которым уже пройдены проверки и их можно забирать в полноценное тестирование, сколько ещё осталось задач проверить разработчикам.

Чек-листы самопроверки для разработчиков

Что важно для таких чек-листов

1. **Разработчик не должен выполнять полноценное тестирование.** Он проходит так называемый «happy path» тест, на то что разработанный функционал выполняет все запрограммированные сценарии и не ломается на простейших данных и интеграциях.
   Проверка сложных случаев, граничных значений это уже работа QA-инженера, иначе разработчик будет избыточно заниматься тестированием.
2. **Такой чек-лист обязательно учитывает критерии приёмки** из постановки задачи. Все критерии должны быть покрыты тестами.
3. **Если нужны специфические данные для проверок** – должно быть указано как эти данные получить/сгенерировать.

Чек-листы самопроверки для разработчиков

Что важно для таких чек-листов

4. **Основные сценарии использования** должны легко проходиться вручную (например: регистрация, авторизация, ключевые операции).

5. **Учитывается, что пользователь может выполнить все целевые действия без ошибок** (открытие страниц, навигация, отправка данных).

6. **Валидация и обработка данных**
   - *обязательно то, что корректные данные обрабатываются без ошибок* (например, валидные email, пароли, числа).
   - *Обязательные поля и форматы соблюдены* (ошибки не возникают при правильном вводе).

Чек-листы самопроверки для разработчиков

Что важно для таких чек-листов

7. Интеграции

- Взаимодействие с API/БД/сторонними сервисами работает штатно (получение/отправка данных, корректные ответы).
- Нет ошибок при работе с внешними зависимостями (например, платежные шлюзы, SMTP) при отправке "удобных"/корректных данных.

8. Сохранение и отображение данных

- Данные корректно сохраняются в БД/кеше.
- Информация отображается верно после обновления страницы или повторного входа.

Чек-листы самопроверки для разработчиков

Что важно для таких чек-листов

9. **Юзабилити и интерфейс проверки**
   - Нет явных визуальных артефактов (например, скроллбары в неположенных местах, наложения текста).
   - Адаптивность проверена хотя бы на основном разрешении (типовое мобильное устройство/планшет/десктоп).

10. **Производительность, проверяемая очевидным способом**
    - Время отклика для ключевых операций в пределах нормы (например, загрузка страницы < 2 сек).
    - Нет явных "зависаний" интерфейса при типовых действиях.

    другие тесты производительности, нагрузку проверяет уже QA.

Ручное тестирование, чек-листы и тест-кейсы

Ручное тестирование, чек-листы и тест-кейсы

Ручное тестирование - это проверка ПО без использования автоматизированных инструментов.

Основные задачи:

- Поиск дефектов.
- Проверка соответствия требованиям (не равно тестированию требований)
- Оценка пользовательского опыта (UX).

Когда применяется:

для функционального тестирования (особенно Web,МП), для сложных UI-сценариев, для регрессионного тестирования.

Ручное тестирование, чек-листы и тест-кейсы

Преимущества:

- Гибкость (адаптация к изменениям). Идеально для Agile! 😊
- Лучшее понимание контекста пользователя.
- Супер эффективно для небольших проектов.

Сложности:

- Трудоемкость.
- Риск человеческой ошибки.
- Сложность масштабирования – понадобится больше QA инженеров

Ручное тестирование, чек-листы и тест-кейсы

Чек-листы в ручном тестировании

Это список проверок для контроля выполнения ключевых функций.

Зачем нужны:

- Систематизация процесса, сохранение опыта
- Уменьшение вероятности пропуска ошибок

Пример для интернет-магазина

Сценарий	Ожидаемый результат	Статус (ok/not ok)
TS_01: Авторизация с валидными данными	Пользователь перенаправлен в личный кабинет	Ok
TS_02: Добавление товара в корзину	Товар отображается в корзине, обновляется итоговая сумма.	Not ok

Ручное тестирование, чек-листы и тест-кейсы

Чек-листы в ручном тестировании. Как создать чек-лист?

Ручное тестирование, чек-листы и тест-кейсы

Тест-кейсы в ручном тестировании

Тест-кейсы **это пошаговая инструкция для проверки конкретного сценария**. Структура:

Ручное тестирование, чек-листы и тест-кейсы

Тест-кейсы в ручном тестировании

Пример для сценария из чек листа «TS_02: Добавление товара в корзину»:

ID	Сценарий	Пердусловия	Шаги	Ожидаемый результат	Фактический результат
TS_02_case001	Успешное добавление одного товара в корзину	1. Пользователь авторизован. 2. Товар "X" доступен в каталоге.	1. Открыть страницу товара "X". 2. Нажать кнопку "Добавить в корзину". 3. Перейти в раздел "Корзина".	- Товар "X" отображается в корзине. - Сумма заказа равна цене товара "X".	Ok
TS_02_case002	Добавление нескольких товаров в корзину	1. Пользователь авторизован. 2. Товары "X" и "Y" доступны в каталоге.	1. Добавить товар "X" в корзину. 2. Вернуться в каталог, добавить товар "Y" в корзину. 3. Перейти в раздел "Корзина".	- В корзине отображаются товары "X" и "Y". - Сумма заказа равна сумме цен товаров "X" и "Y".	Not Ok (приложен скриншот)
TS_02_case003	Попытка добавить товар с нулевым количеством	1. Пользователь авторизован. 2. Товар "X" доступен в каталоге.	1. На странице товара "X" изменить количество на 0. 2. Нажать кнопку "Добавить в корзину".	- Появляется сообщение: "Укажите количество больше нуля". - Товар не добавляется в корзину.	Ok

Ручное тестирование, чек-листы и тест-кейсы

Тест-кейсы в ручном тестировании.
Написание эффективных тест-кейсов

Как должно быть:

- Однозначность формулировок.
- Независимость от других кейсов.
- Покрытие всех возможных сценариев (положительные/отрицательные).

Как не должно быть:

- Избыточная детализация.
- Нечеткие ожидаемые результаты.

Ручное тестирование, чек-листы и тест-кейсы

Чек-листы vs. Тест-кейсы

Чек-листы:	Тест-кейсы:
Описывают общие пункты и сценарии	Детальные шаги, можно использовать как основу для автотестов. Бонус - удобно обучать новых сотрудников
Отлично подходят для быстрых проверок, регресса, для опытных QA	Хорошо подходят для сложных сценариев и новых функций с непростым интерфейсом. Обеспечивают глубину
Экономят время на документацию	Проще проверять самого себя, новичок с меньшей вероятностью ошибётся

Unit-тесты и интеграционные тесты

Unit-тесты и интеграционные тесты

Unit(Юнит)-тестирование - это тестирование минимальных единиц кода (функции, классы, методы) изолированно.

- Использует **моки**/стабы для изоляции компонентов.
- Выполняются быстро, запускаются автоматизировано.
- Инструменты: JUnit (Java), pytest (Python), NUnit (.NET).
- Пишут, как правило, сами разработчики

Зачем нужно?

- Раннее обнаружение ошибок до передачи в отдел тестирование
- Упрощение отладки для разработчика

Важно для руководителя проектов – при оценке нового функционала разработчик должен *закладывать время на написание Unit-тестов, а не только создание нового кода.*

Unit-тесты и интеграционные тесты

Интеграционное тестирование - это проверка взаимодействия модулей, сервисов или систем. Основная цель – проверить, что все или группа модулей работают как единый сервис.

Подходы:

- Снизу вверх: Постепенная интеграция с нижних уровней.
- Сверху вниз: Начинается с верхних модулей.
- Big Bang: Все компоненты объединяются сразу.

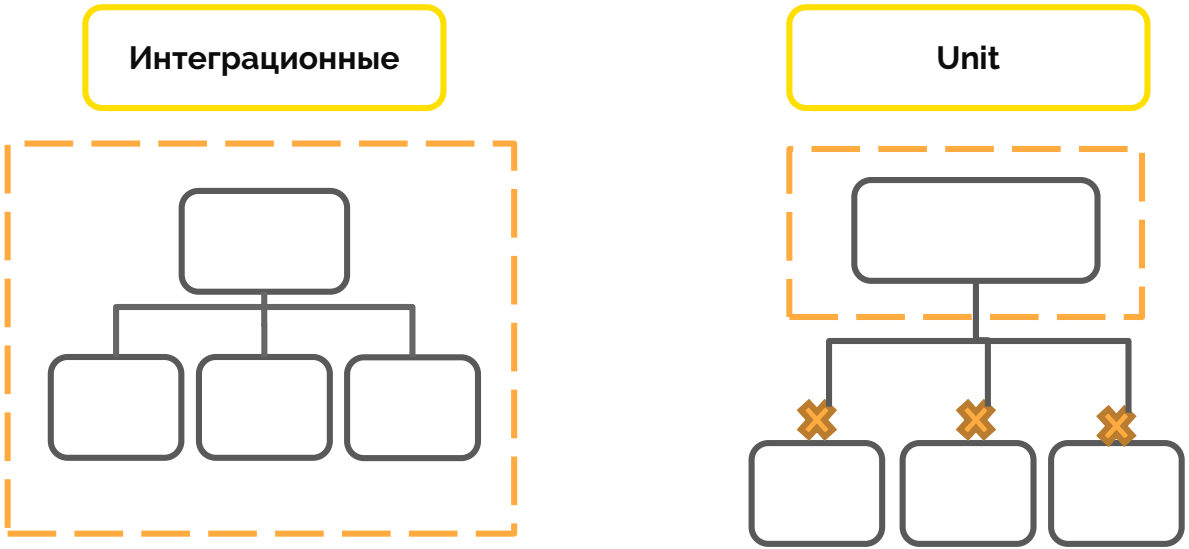
Инструменты: TestNG, Postman, Insomnia (API), Selenium, Cypress (веб-интерфейсы).

Можно делать различные группировки модулей под разный функционал. Какие-то добавлять, какие-то отключать. Например, можно сделать тест на поведение системы при недоступности системы из которой получаем курс валюты.

Unit-тесты и интеграционные тесты

Критерий	Unit-тесты	Интеграционные тесты
Уровень	Отдельные компоненты	Взаимодействие компонентов
Скорость	Быстрые	Медленные
Сложность	Низкая (изоляция)	Высокая (настройка окружения)
Цель	Проверка логики	Проверка интеграции

Unit-тесты и интеграционные тесты



Unit-тесты и интеграционные тесты

Использование на проекте

Юнит-тесты:

пишутся и проводятся на этапе разработки, желательно для каждой функции. По сути это **фундамент качества** кода.

Интеграционные тесты:

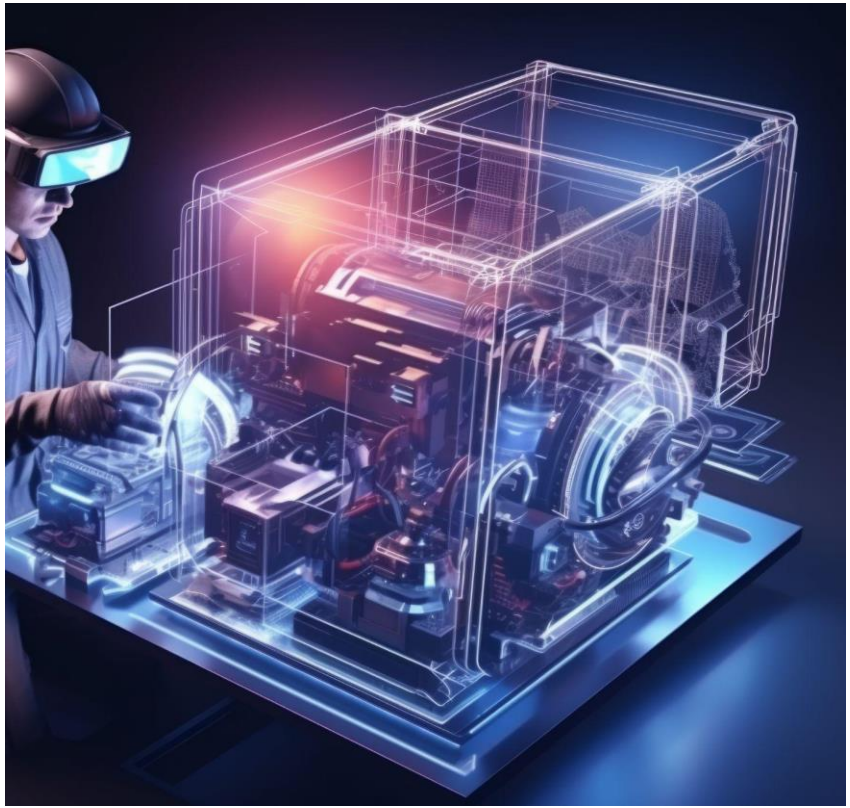
проводятся после сборки модулей, перед релизом, страхуемся от **системных** сбоев

Идеальный процесс:

1. Пишем **юнит-тесты** для каждой функции

2. Проводим **интеграционное тестирование** после объединения модулей

3. Комбинируем оба подхода для полного покрытия.



Автотесты

Автотесты

Зачем руководителю проекта автотесты?

1. Снижение рисков:

Автотесты минимизируют человеческие ошибки, повышают стабильность продукта, если покрытие достаточно полное.

2. Экономия времени и бюджета:

- Автотесты относятся к ранним этапам обнаружения багов. Выявление автотестами бага снижает стоимость исправления до 10 раз, чем на этапе продакшена.
- Качественно работающие автотесты высвобождают команду для задач, требующих креатива и нестандартного подхода к решению.

Автотесты

Зачем руководителю проекта автотесты?

3. Прозрачность процессов:

- Отчеты автотестов дают объективную метрику качества (например, 95% покрытие кода).
- Упрощение коммуникации с заказчиком: демонстрация прогресса через успешные тест-кейсы.

4. Ускорение релизов:

- Автотесты позволяют гораздо легче внедрять CI/CD (непрерывную интеграцию и доставку).
- Очевидный пример: ежедневные деплои вместо еженедельных.

Для руководителя проекта это полезный инструмент управления рисками и оптимизации процессов.

Регрессионное тестирование

Регрессионное тестирование

Регрессионное тестирование это проверка существующего функционала после изменений в коде.

С точки зрения QA процесс нужен для подтверждения того, что правки не вызвали непредвиденных ошибок в старом функционале.

Обычно проводится после исправлений, до релиза, при обновлении зависимостей.

С точки зрения руководителя проекта и Agile методолгии – регрессионное тестирование просто необходимо, т.к. итерации меняющие код и потенциально ломающие старые функции происходят регулярно.

С точки качества. Кроме функционала новым кодом можно снизить производительность, может пострадать UI качество (например «уедут» шрифты), появятся дыры в безопасности и т.п.

Регрессионное тестирование. Как проводить

Собираем набор регрессионных тестов.

Best-practice является проверка критических возможностей:

- Функции, стоящие на ключевых бизнес-потоках.
- Часто используемые функции.

Дополнение: **то что может быть автоматизировано - должно быть автоматизировано.** Регресс это повторяющаяся операция, соответственно легче поддаётся автоматизации. Если регресс покрыт автотестами на 85-90% - TTM сократится в разы.

1. Совместно с командой разработки QA-инженеры определяют модули с изменениями.

2. QA-команда обсуждает, какие изменения следует подвергнуть всестороннему тестированию, а какие могут обойтись без него.

3. Тестировщики определяют точки входа и выхода в выбранных модулях и наборы тестов

(как правило все чек-листы и кейсы для проверок уже есть в команде, нужно из них выбрать наиболее подходящие с учетом изменений)

Регрессионное тестирование. Как проводить

Полное повторное тестирование

Применяется ко всем существующим наборам тестов. Очень надёжно, но требует значительных затрат времени и ресурсов.

Когда применять полный регресс:

1. Когда происходит значительное изменение, например, при адаптации приложения к новой платформе или стеку (языки, фреймфорвки, БД), а также при значительном обновлении операционных систем (особенно подвержены влиянию на приложения мобильные ОС).

2. Небольшой размер и простота приложения. QA-команды могут запускать их полный прогон для получения максимального покрытия.

3. «По запросу» - т.е. запускать время от времени, раз в квартал или полугодие, для проверки, что в скрытых областях не копятся дефекты.

Регрессионное тестирование. Как проводить

Выборочное тестирование

При таком подходе QA-инженеры выбирают только то что затронуто изменениями и проводят регресс только на них. Выбрав соответствующие области, можно применить ограниченные и релевантные тестовые случаи. Это позволит сократить время и усилия.

Такой подход подходит для сложных или масштабных приложений, в которых количество тестовых сценариев велико и требует прохождения до нескольких дней.

Тестирование по приоритетам

Выборка для регресса в этом случае происходит по приоритетности функционала, важному для бизнеса в моменте времени, например:

1. Учитывается частота отказов
2. Влияние на бизнес.
3. Клиент-ориентированные функции.
4. Безопасность и т.п.

Регрессионное тестирование. Как проводить

Корректирующее регрессионное тестирование

1. Такой вид тестирования применяют, когда продукт обновляют, но не меняют его основные функции. Например, изменили дизайн, но не меняли функциональность.

2. Второе направление - проверка того, что исправление дефекта не вызвало новых ошибок в уже протестированных частях системы. Оно выполняется после устранения конкретного бага, чтобы убедиться, что изменения не нарушили существующую функциональность.

Фиксация багов (дефектов)

Фиксация багов (дефектов)

Фиксация багов - это ключевой этап процесса тестирования программного обеспечения, который позволяет формализовать обнаруженные проблемы, передать их разработчикам для исправления и отслеживать их статус. Правильное документирование дефектов повышает эффективность работы команды и ускоряет устранение ошибок.

Дефект — это отклонение фактического поведения системы от ожидаемого, описанного в требованиях, спецификациях или пользовательских сценариях. Например:

- Кнопка не реагирует на нажатие.
- Данные не сохраняются в базе.

Фиксация дефектов — это не просто формальность, а важный этап, который напрямую влияет на качество продукта. Чем точнее и полнее описан баг, тем быстрее он будет устранен, а команда сможет сосредоточиться на дальнейшем развитии проекта.

Фиксация багов (дефектов)

Основные шаги фиксации дефектов

1. **Обнаружение:** Тестировщик выявляет проблему в ходе выполнения тест-кейсов, исследовательского тестирования или других методов.
2. **Документирование:** Дефект подробно описывается в специальной системе (например, Jira, Bugzilla, Azure DevOps).
3. Приоритизация: Определяется критичность дефекта (блокирующий, критический, средний, низкий).
4. **Исправление:** Разработчик анализирует проблему, вносит изменения в код.
5. **Повторное тестирование (Re-testing):** Тестировщик проверяет, устранена ли ошибка.
6. **Закрытие:** Если дефект исправлен, его статус меняется на «Закрыт»

Анализ результатов тестирования

Анализ результатов тестирования

Анализ результатов тестирования – это процесс интерпретации данных, полученных в ходе проверки ПО.

Его цель - оценить качества продукта, принять решение о его готовности к релизу и улучшение процессов разработки.

Этот этап включает не только техническую оценку дефектов, но и коммуникацию с заказчиком, а также предоставление менеджеру проекта ключевых метрик для управления рисками.

Анализ полезно получить в нескольких разрезах:

1. Отчетность заказчику

2. Анализ для менеджера проекта

3. Типизация найденных ошибок и дополнительные аспекты анализа

Анализ результатов тестирования

Отчетность заказчику – дает прозрачность в демонстрации прогресса, выявленных рисках, обосновании необходимости доработок. Что включать в отчет?

Общие сведения:

- Общее количество тест-кейсов и процент успешно пройденных.
- Статистика дефектов: количество найденных, исправленных, отклоненных, критических/блокирующих ошибок.
- Графики и диаграммы (например, динамика открытия/закрытия багов).

Ключевые проблемы:

- Список критических дефектов, которые могут повлиять на сроки или функциональность.
- Примеры сценариев, где требования не выполнены.

Рекомендации:

- Нужен ли дополнительный цикл тестирования.
- Какие модули требуют повышенного внимания.

Риски:

Потенциальные угрозы для пользователей или бизнеса, если дефекты не будут исправлены.

Анализ результатов тестирования

Для PM анализ результатов тестирования — это основа для:

1. Управления рисками: понимание, какие дефекты могут сорвать сроки или бюджет (например, блокирующие ошибки в ключевом функционале).

2. Принятия решений о релизе: оценка, готов ли продукт к выпуску, или нужен еще один цикл доработок.

3. Коммуникации с заказчиком: обоснование переноса дедлайнов, запрос дополнительных ресурсов.

4. Оптимизации процессов: если много ошибок возникает на этапе проектирования — нужно усилить работу с требованиями.

5. Оценки команды: анализ скорости исправления багов, качества кода (например, соотношение новых дефектов к закрытым).

Какие метрики может выбрать PM:

- Соотношение времени затраченного на фичу и время на исправление багов в ней
- Time-to-Fix - среднее время на исправление дефекта
- Reopened Bugs Rate - процент повторно открытых багов

Анализ результатов тестирования

Классификация дефектов:

Инструмент призванный кластеризовать проблемы и принять управленческие решения QA-lead и руководителю проекта.

При составлении баг репорта QA-инженер типизируют баги по нескольким метрикам. Выбор метрик зависит от специфики ПО, команды, важности определенного функционала.

Пример типизации:

Метод обнаружения бага: регрессионное тестирование

Место возникновения: бэк система

Окружение: developer-контур

Корневая причина проблемы (root-cause): чек-лист самопроверки не пройден.
Для руководителя проекта отчет поможет изменить процессы в команде. Можно выделить места возникновения, например, если часто возникает ошибка в бэке, стоит глубже понять процесс разработки бэк-энд разработчиков.

Анализ результатов тестирования

Классификация дефектов.

Другие примеры использования:

1. При нескольких релизах в отчётах есть заметный процент ошибок с root cause «Ошибка со стороны интеграции внешнего сервиса».

Это повод для более пристального изучения. Скорее всего улучшение качества лежит в области процессов, т.к. смежный сервис вносит изменения в API, соответственно нужно договориться о согласованных изменениях до начала разработки.

2. После внесения изменений в процесс тестирования требований должно измениться количество ошибок с причиной «ошибка в требованиях».

Если этого не происходит, стоит обратить внимание на само тестирование, скорее всего QA подходил к процессу формально, не проводя полноценное тестирование требований аналитика.

Вопросы

Спасибо
за внимание