| МГТУ | ИУ7 |

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лектор: Жаров С.М.

Москва, МГТУ им. Н. Э. Баумана.

Тестирование и отладка ПО | Теория к Экзамену

СПИСОК ВОПРОСОВ

1. Определение качества ПО. Что оценивается?

- **Качество ПО** – способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям (ISO/IEC 25000:2014)[1].
  - весь объём признаков и характеристик программ, который относится к их способности удовлетворять установленным или предполагаемым потребностям (ГОСТ Р ИСО/МЭК 9126-93, ISO 8402:94);
  - степень, в которой система, компонент или процесс удовлетворяют потребностям или ожиданиям заказчика или пользователя (IEEE Std 610.12-1990).
- **Качество программного обеспечения** (Software Quality) – это совокупность характеристик программного обеспечения, относящихся к его способности удовлетворять установленные и предполагаемые потребности. [Quality management and quality assurance][2].
- **Тестирование программного обеспечения** – проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование – это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

1. Wiki источник
2. Жаров

Рисунок 1. Качество ПО.

- Верификация (verification) – это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа[IEEE]. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.
- Валидация (validation) – это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе [BS7925-1].

- Также можно встретить иную интерпретацию:
  - Процесс оценки соответствия продукта явным требованиям (спецификациям) и есть верификация (verification), в то же время оценка соответствия продукта ожиданиям и требованиям пользователей – есть валидация (validation). Также часто можно встретить следующее определение этих понятий:
    - Validation – ’is this the right specification?’.
    - Verification – ’is the system correct to specification?’.

2. Определение тестирования ПО. Этапы тестирования ПО.

Тестирование ПО - проверка соответсвия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование – одна из техник контроля качества, включающая в себя активности по планированию работ (Test managment), проектирование тестов (Test Design), выполнение тестирования (Test Execution) и анализу полученных результатов (Test Analysis)

Этапы тестирования:

1. Анализ продукта
2. Работа с требованиями
3. Разработка стратегии и планирования процедур контроля качества
4. Создание тестовой документации
5. Тестирование прототипа
6. Основное тестирование
7. Стабилизация
8. Эксплатация

3. Какие стандарты к качеству ПО бывают? Для чего они нужны? Нужны ли?

А нужны ли стандарты? – **Скорее да**, чем нет.

- IEEE 829 (документация), IEEE 1008 (unit tests), IEEE 1061 (метрики)
- ISO 29119, ISO 8402, ISO 9126 (International Organization for Standardization)
  - **ISO 29119** – Но, почему так важен именно ISO? Стандарт ISO это общемировой стандарт, который принимается во многих странах мира. Если мы посмотрим на модель стандартов, то мы увидим, что многие стандарты являются профессиональными, коммерческими, национальными, но никак не международными.
  - **ISO 8402** – Управление качеством и обеспечение качества
  - **ISO 9126** – определяет оценочные характеристики качества программного обеспечения.
- BS 7925-1, BS 7925-2 (Бритиш стандарт)
- ГОСТ Р 56921-2016, ГОСТ Р 56920-2016, ГОСТ Р ИСО/МЭК 12119 (аналоги к тем, что выше, просто ГОСТ – РФовские)

4. Что такое черный\белый(стеклянный)\серый ящик с точки зрения тестирования?

Черный ящик

Тестирование черным ящиком *– техника тестирования, основанная на работе исключительно с внешними интерфейсами тестируемой системы*

Согласно ISTQB (International Software Testing Qualifications Board):

- *Тестирование как функциональное, и так не функциональное, не предполагающее знание внутреннего устройства компонента или системы*
- *Тест дизайн основанный на технике черного ящика – процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутренего устройства*

Целью данной техники является ошибок в следующих категориях:

1. Неправильно реализованные или недостающие функции;
2. Ошибки интерфейса;
3. Ошибки в структурах данных или организации доступа к внешним базам данных;
4. Ошибки поведения или недостаточная производительность системы

Пример: Тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация.

Преимущества:

- Тестирование проводится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
- Тестировщику не нужны ЯП.
- Можно писать тесты как готовы спецификация

Недостатки:

- Без спецификации почти невозможно составить эффективное тестирование;
- Тесты могут быть избыточными (если была проведено тестирование на уровне модулей);

Белый ящик

Тестирование методом белого ящика *– метод тестирования программного обеспечения, который предполагает, что внутренняя стуктура/ устройство/реализация/ системы известна тетсировщику.*

Согласно ISTQB: Т***естирование белым ящиком*** *– это тестирование основанное на анализе внутренней структуры компонента или системы.*

Пример: Тестировщик, который, как правило, является программистом, изучает реализацию кода поля ввода на веб-странице, определяет все предусмотренные (как правильные, так и неправильные) и не предусмотренные пользовательские вводы, и сравнивает фактический результат выполнения программы с ожидаемым. При этом ожидаемый результат определяется именно тем, как должен работать код программы.

Преимущество:

- *тестирование может производиться на ранних этапах: нет необходимости ждать создания пользовательского интерфейса;*
- *можно провести более тщательное тестирование, с покрытием большого количества путей выполнения программы.*

Недостатки:

- *для выполнения тестирования белого ящика необходимо большое количество специальных знаний*
- *при использовании автоматизации тестирования на этом уровне, поддержка тестовых скриптов может оказаться достаточно накладной, если программа часто изменяется.*

Критерий	Black Box	White Box
Определение	Тестирование, как функциональное, так и не функциональное, не предполагающее знание внутреннего устройства компонента или системы.	Тестирование, основанное на анализе внутренней структуры компонента или системы
Уровни, к которым применима техника	В основном: • Приемочное тестирование • Системное тестирование	В основном: • Юнит-тестирование • Интеграционное тестирование
Кто выполняет	Как правило, тестировщики	Как правило, разработчики
Знание программирования	Не нужно	Необходимо
Знание реализации	Не нужно	Необходимо
Основна для тест-кейсов	Спецификация, требования	Проектная документация

Серый ящик

Тестирование методом серого ящика – метод тестирования программного обеспечения, который предполагает, комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично.

Эту технику тестирования также называют методом полупрозрачного ящика: что-то мы видим, а что-то – нет.

***Пример**: Тестировщик изучает код программы с тем, чтобы лучше понимать принципы ее работы и изучить возможные пути ее выполнения. Такое знание поможет написать тест-кейс, который наверняка будет проверять определенную функциональность.*

5. Как с этим связана отладка (дебаг) ПО?

Термин отладка может иметь разные значения, но в первую очередь он означает устранение ошибок в коде. Делается это по-разному. Например, отладка может выполняться путем проверки кода на наличие опечаток или с помощью анализатора кода. Код можно отлаживать с помощью профилировщика[3] производительности. Кроме того, отладка может производиться посредством отладчика.

Отладчик – это узкоспециализированное средство разработки, которое присоединяется к работающему приложению и позволяет проверять код.

Отладчик – важнейший инструмент для поиска и устранения ошибок в приложениях. Однако большое значение имеет контекст. Важно использовать все средства, имеющиеся в вашем распоряжении, чтобы быстро устранять ошибки. Зачастую лучшим "средством" являются **правильные методики** написания кода. Зная, когда лучше использовать отладчик, а когда – другие средства, вы также сможете более эффективно использовать отладчик.

Альтернатива: вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода – в консоль или файл. Вывод отладочных сведений в файл называется **логированием**.

3. Профилирование приложения – неизбежный компонент любой оптимизации, оптимизация без профилирования невозможна. Вы профилируете, находите узкие места, оптимизируете-профилируете, это постоянный цикл.

6. Что такое класс эквивалентности?

Класс эквивалентности (equivalence class) *– набор данных, обработка которых приводит к одному и тому же результату.*

Два теста можно считать эквивалентными, в случае когда:

- они проверяют одну и ту же часть системы (функцию, модуль);
- один тест находит ошибку, то и другой, скорее всего, найдет ошибку и наоборот (если один не находит ошибку – второй также не находит);
- они используют сходные наборы входных данных;
- чтобы выполнить тесты, необходимо совершить одни и те же операции;
- в результате проведения тестов получаем одинаковые выходные данные и система находится в одном и том же состоянии:
  - срабатывает один и тот же блок обработки ошибки;
  - не срабатывает блок обработки ошибки.

Разделение на классы эквивалентности *– это техника, при которой функционал (диапазон возможных входных значений) разделяется на группы значений эквивалентных по воздействию на систему.*

Вместе КЭ используется также ***анализ граничных значений*** *и считаются одним из самых выжным при тестировании:*

- *Позволяют минимзировать количество тестов;*
- *Интуитивно понятно;*

- *Неправильное использование данных техник может повлечь за собойпро-пуск критичных дефектов.*

7. Что такое функциональное и нефункциональное тестирование?

Функциональное

Функциональное тестирование рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом. Функциональные виды тестирования рассматривают внешнее поведение системы. Далее перечислены одни из самых распространенных видов функциональных тестов:

- Функциональное тестирование (Functional testing)
- Тестирование безопасности (Security and Access Control Testing)
- Тестирование взаимодействия (Interoperability Testing)

Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (компонентном, интеграционном, системном, приемочном). Как правило, эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (use cases).

Тестирование функциональности может проводиться в двух аспектах:

- требования
- бизнес-процессы

Тестирование в перспективе «требования» использует спецификацию функциональных требований к системе как основу для дизайна тестовых случаев (Test Cases). В этом случае необходимо сделать список того, что будет тестироваться, а что нет, приоритезировать требования на основе рисков (если это не сделано

в документе с требованиями), а на основе этого приоритезировать тестовые сценарии (test cases). Это позволит сфокусироваться и не упустить при тестировании наиболее важный функционал.

Тестирование в перспективе «бизнес-процессы» использует знание этих самых бизнес-процессов, которые описывают сценарии ежедневного использования системы. В этой перспективе тестовые сценарии (test scripts), как правило, основываются на случаях использования системы (use cases).

Преимущества функционального тестирования:

✔ имитирует фактическое использование системы;

Недостатки функционального тестирования:

✗ возможность упущения логических ошибок в программном обеспечении;

✗ вероятность избыточного тестирования.

Нефункциональное

Нефункциональное тестирование описывает тесты, необходимые для определения характеристик программного обеспечения, которые могут быть измерены различными величинами. В целом, это тестирование того, "Как" система работает. Качества оцениваемые им:

- **Надежность** (реакция системы на непредвиденные ситуации).
- **Производительность** (Работоспособность системы под разными нагрузками).
- **Удобство** (Исследование удобности работы с приложением с точки зрения пользователя).

- **Масштабируемость** (Требования к горизонтальному или вертикальному масштабированию приложения).
- **Безопасность** (Защищенность пользовательских данных).
- **Портируемость** (Переносимость приложения на различные платформы).

Далее перечислены **основные виды** нефункциональных тестов:

- **Тестирование установки** (Installation testing) – проверка успешности установки приложения, его настройки и удаления. Снижает риски потери пользовательских данных, потери работоспособности приложения и пр.
- **Тестирование удобства использования** (Usability testing) – характеризует систему с точки зрения удобства использования конечного пользователя.
- **Конфигурационное тестирование** (или тестирование портируемости) – исследование работоспособности программной системы в условиях различных программных конфигураций.
- **Тестирование на отказ и восстановление** (Failover and Recovery Testing) – исследование программной системы на предмет восстановления после ошибок, сбоев. Оценивание реакции защитных свойств приложения.
- **Тестирование на производительность**:
  - нагрузочное тестирование (Performance and Load Testing)
  - стрессовое тестирование (Stress Testing)
  - тестирование стабильности или надежности (Stability / Reliability Testing)
  - объемное тестирование (Volume Testing)

8. Для чего нужно логгирование и что нужно логгировать при разработке

Логирование *– вывод отладочных сведений в файл.*

Зачем нужны логи:

- *Знать что происходит в системе в данный момент*
- *Профилирование (на что тратится больше времени/ресурсов)*
- *Изучение обстоятельств которые привели к определенному состоянию системы*

Что нужно логировать?

- **Debug:** сообщения отладки, профилирования. В production системе обычно сообщения этого уровня включаются при первоначальном запуске системы или для поиска узких мест (bottleneck-ов).
- **Info:** обычные сообщения, информирующие о действиях системы. Реагировать на такие сообщения вообще не надо, но они могут помочь, например, при поиске багов, расследовании интересных ситуаций итд.
- **Warn:** записывая такое сообщение, система пытается привлечь внимание обслуживающего персонала. Произошло что-то странное. Возможно, это новый тип ситуации, ещё не известный системе. Следует разобраться в том, что произошло, что это означает, и отнести ситуацию либо к инфо-сообщению, либо к ошибке. Соответственно, придётся доработать код обработки таких ситуаций.
- **Error:** ошибка в работе системы, требующая вмешательства. Что-то не сохранилось, что-то отвалилось. Необходимо принимать меры довольно быст-

ро! Ошибки этого уровня и выше требуют немедленной записи в лог, чтобы ускорить реакцию на них. Нужно понимать, что ошибка пользователя – это не ошибка системы. Если пользователь ввёл в поле -1, где это не предполагалось – не надо писать об этом в лог ошибок.

- **Fatal:** это особый класс ошибок. Такие ошибки приводят к неработоспособности системы в целом, или неработоспособности одной из подсистем. Чаще всего случаются фатальные ошибки из-за неверной конфигурации или отказов оборудования. Требуют срочной, немедленной реакции. Возможно, следует предусмотреть уведомление о таких ошибках по SMS.

Обычно принято логировать последние 3 типа и взаимодействие с БД

9. Профилирование: что профилируют, как, по каким параметрам

Профилирование – это сбор характеристик программы во время ее выполнения. При профилировании замеряется время выполнения и количество вызовов отдельных функций и строк в коде программы. При помощи этого инструмента программист **может найти наиболее медленные участки кода** и провести их оптимизацию, либо чтобы **генерировать граф вызовов (Call Graph)**.

Профилировщики на основе событий – некоторые языки:

- **Java** API JVMTI – **перехват событий** (вызовы, загрузка класса, выгрузка, выход из потока).
- **Python** – модулем профиля **hotshot** (который основан на графике вызовов) и использование функции **sys.setprofile** для **перехвата событий**, таких как c_{call, return, exception}, python_ {call, return, exception}.

Статистические профилировщики – Некоторые профилировщики работают по **выборке** . Профилировщик выборки проверяет стек вызовов целевой программы через регулярные промежутки времени, используя прерывания операционной системы . **Профили выборки** обычно менее точны в числовом отношении и конкретны, но позволяют целевой программе работать почти на полной скорости.

10. Уровни тестирования\Пирамида тестирования

1. **Модульное тестирование (Unit Testing)**
    - *Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).*
2. **Интеграционное тестирование (Integration Testing):**
    - *Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.*
3. **Системное тестирование (System Testing)**
    - *Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д.*
4. **Операционное тестирование (Release Testing).**
    - *Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес модели системы. Следует учесть, что и бизнес модель может содержать ошибки. Поэтому так важно провести операционное тестирование как*

финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения – критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.

5. **Приемочное тестирование (Acceptance Testing)**
   - Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью
     - определения удовлетворяет ли система приемочным критериям;
     - вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

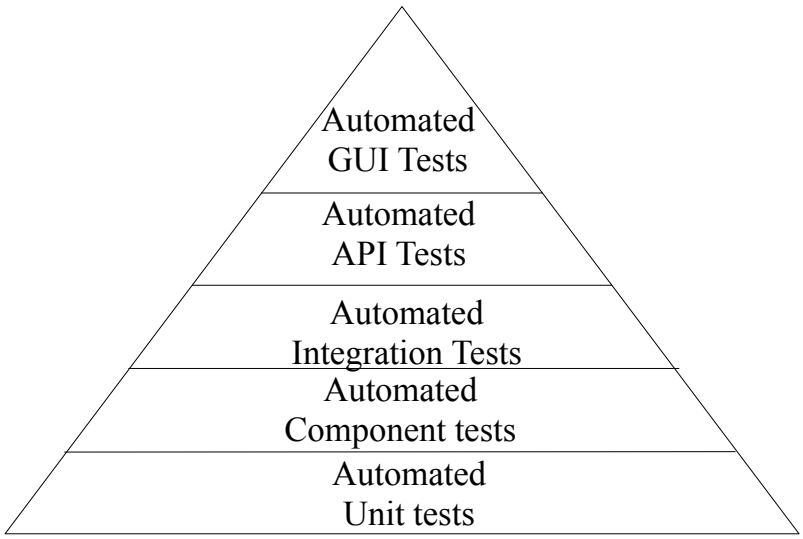

Рисунок 2. Пирамида тестирования

11. Unit-тестирование: зачем нужно, что бывает если не писать юнит-тесты, что бывает если писать бесполезные юнит-тесты, что такое software entropy

Модульное тестирование (unit testing) – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к **регрессии**[4].

Таким образом, юнит-тестирование – это первый этап из пирамиды, на борьбе с багами. За ним еще интеграционное, приемочное и, наконец, ручное тестирование, в том числе «свободный поиск».

Конечно, это нужно не всегда. Выделим примеры, когда не нужно писать тесты:

- ✔ Простой сайт-визитка из статических html-страниц и с одной формой отправки письма. Здесь нет никакой особенной логики, быстрее просто все проверить «руками»;
- ✔ Рекламным сайт, простые флеш-игры или баннерами – сложная верстка/ анимация или большой объем статики. Никакой логики нет, только представление;
- ✔ Делаете проект для выставки. Срок – от двух недель до месяца, ваша система – комбинация железа и софта, в начале проекта не до конца известно, что именно должно получиться в конце. Софт будет работать недолго;
- ✗ Вы Ванга. (**это шуточный вариант!**) – вы всегда пишете код без ошибок, обладаете идеальной памятью и даром предвидения.

Критерии тестов. Тесты должны:

- ✔ Быть достоверными
- ✔ Не зависеть от окружения, на котором они выполняются
- ✔ Легко поддерживаться

4. Появление ошибок в уже оттестированных местах программы.

- ✔ Легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется)
- ✔ Соблюдать единую конвенцию именования
- ✔ Запускаться регулярно в автоматическом режиме

Лондонский и классический (плюсы и общий минус)

Лондонский:

- ✔ Лучшее гранулирование, тест-класс проверяет конкретный класс и только его;
- ✔ Чем больше проект, тем сложней использовать классический подход из-за транзитивных зависимостей между классами;
- ✔ Если тест не пройден – очевидно в каком именно классе ошибка;
- ✔ Логично подводит к TDD и прочим современным методикам.

Классический:

- ✔ Естественный способ строить интеграционные тесты;
- ✔ Единственный подход к End-to-end тестам (E2E);
- ✗ **Обе эти категории относятся уже не к чистому unit-тестированию.**

Энтропия ПО (software entropy)

По типу физики[5] – при модификации системы её беспорядок может только расти, это и называется энтропия ПО.

Сложность ПО возрастает по мере её модификации.

5. Второй закон **термодинамики** – беспорядок в замкнутой системе не может уменьшаться, он может только оставаться неизменным или расти. **Мера** беспорядка является **энтропия**.

12. Unit-тестирование: атрибуты отдельного теста и набора тестов

Атрибуты теста:

- Быть достоверными;
- Не зависеть от окружения, на котором выполняются;
- Легко поддерживаться
- Легко читаться и быть простыми для понимания (даже новый разработчик должен понять, что именно тестируется)
- Соблюдать единую конвенцию именования
- Запускаться регулярно в авторежиме.

13. Unit-тестирование: два метода оценки покрытия кода, к какому покрытию нужно стремиться

Как определить покрытие кода тестами? Source Lines of Code – SLOC (физические строки, логические строки и строки комментариев):

$$\text{Code coverage} = \frac{\text{Lines of code executed}}{\text{Total numbers of lines}}$$

Дизайн кода и тестов непосредственно влияет процент покрытия, но не гарантирует проверки всех возможных случаев.

Стремиться нужно не к 100% покрытию всего кода, а охвату функциональности на все 100%.

Тут уже лучше почитать, т. к. написать можно много, но суть коротка.

14. Unit-тестирование: организация тестирования в CI\CD, какие части кода нужно покрывать тестами?

Типичный CD-конвейер состоит из этапов сборки, тестирования и развертывания. Более сложные конвейеры включают в себя следующие этапы:

- *Получение кода из системы контроля версий и выполнение сборки;*
- *Настройка инфраструктуры, автоматизированной через подход "инфраструктура как код";*
- *Копирование кода в целевую среду.*
- *Настройка переменных окружения для целевой среды.*
- *Развертывание компонентов приложения (веб-серверы, API-сервисы, базы данных).*
- *Выполнение дополнительных действий, таких как перезапуск сервисов или вызов сервисов, необходимых для работоспособности новых изменений.*
- *Выполнение тестов и откат изменений окружения в случае провала тестов.*
- *Логирование и отправка оповещений о состоянии поставки.*

Что должно покрываться тестами?

- Механизмы алгоритмов
- Основные методы бизнес-логики
- Предикаты проверки простых запросов к БД
- Методы с высоким риском

15. Unit-тестирование: mock, stub и прочие Test Double; изоляция объекта тестирования – private, shared, out-of-process, volatile, mutable, immutable, etc., как их тестировать?

Дублеры

Термин Test Double (дублер) – как обозначение для объекта, который заменяет реальный объект, от которого зависит SUT (system under test), в тестовых целях.

Когда нам **нужно** использовать **double**?

- ✔ Низкая скорость выполнения тестов с реальными объектами (если это, например, работа с базой, файлами, почтовым сервером и т.п.)
- ✔ Когда есть необходимость запуска тестов независимо от окружения (например, на машине у любого разработчика)
- ✔ Система, в которой работает код, не дает возможности (или дает, но это сложно делать) запустить код с определенным входным набором данных.
- ✔ Нет возможности проверить, что SUT отработал правильно, например, он меняет не свое состояние, а состояние внешней системы. И там эту проверку сделать сложно.

Stubs – обеспечивают жестко зашитый ответ на вызовы во время тестирования. Применяются для замены тех объектов, которые обеспечивают SUT входными данными. Также они могут сохранять в себе информацию о вызове (например параметры или количество этих вызовов) - такие иногда называют своим термином Test Spy. Такая "запись" позволяет оценить работу SUT, если состояние самого SUT не меняется(По Жарову: *Stub – эмулировать и входящие взаимодействия. Это вызовы и взаимодействия, которые исполняются SUT к зависимым объектам, чтобы запросить и получить данные*).

Mocks – объекты, которые настраиваются (например специфично каждому тесту) и позволяют задать ожидания в виде своего рода спецификации вызовов, которые мы планируем получить. Проверки соответствия ожиданиям проводятся через вызовы к Mock-объекту. (По Жарову: *Mock – эмулировать и проверять исходящие взаимодействия. Это вызовы и взаимодействия, которые исполняются SUT к зависимым объектам, чтобы изменить их состояние*)

Spy – вариант Mock без использования фреймворка мокирования.

Dummy – примитивный Stub (возвращать любое not null value).

Fake – Stub для функционала, которого ещё нет.

Какой использовать? Плюсы и минусы есть у обоих. Остановимся на **минусах**.

При использовании классического подхода (**stub**), когда все в тестах - это реальные объекты, в случае ошибки в одном из методов реального объекта, "красными" станут все тесты, в которых этот метод используется (независимо от того, тестируется он в них или просто используется другим объектом). **Локализация проблемы может стать трудной задачей**, если при написании тестов, вы не задумывались над гранулярностью теста: его минимально необходимыми зависимостями.

В обратную сторону, при мокировании (**mock**) всего и вся может возникнуть другая проблема: поведенческие тесты жестко фиксируют внутреннюю реализацию SUT. Что и как вызывается, с какими аргументами, какое API используется и тп. Все это создает трудности при рефакторинге и возможных изменениях в SUT - вместе с кодом все тесты придется писать заново. Всё просто, лучше использовать **mock** только тогда, когда это действительно нужно.

16. Unit-тестирование: AAA, fixtures, Data builder / Object Mother AAA:

1. **Arange –** подготовить объект тестирования, контекс выполнения методов, инициализируются переменные среды;
2. **Act –** выполенение метода, который понаджелит тестированию;
3. Assert – проверка выходного результата и параметра среды после выполнения метода

Рекомендации по наполнению AAA:

1. **Act –** один тест, одна строчка в секции, один вызов одного метода
2. Arrange – почти половина всего кода теста будет тут:
    1. Object Mother pattern – создание, модификация и хранение объектов, которые нужны для выполнения тестов
    2. Builder pattern – генерация тестовых данных для избегания хардкода и обеспечения покрытия разных наборов входных данных для одного и того же теста
3. Assert – ставшаяся почти половина кода теста будет тут. Unit – это не только про Unit – про поведение объекта тестирования.

GWT – given, when, then аналогичный паттерн используемый в BDD/TDD

Fixtures

Автоматические тесты необходимо выполнять неоднократно. Мы хотели бы выполнять тесты в некоторых известных состояниях для гарантии повторяемости процесса тестирования. Эти состояния называются ***фикстуры*.**

Object Mother AND Data Builder

Вкратце, Object Mother - это набор фабричных методов, которые позволяют нам создавать похожие объекты в тестах.

Test Data Builder использует Builder шаблон для создания объектов в модульных тестах. Краткое напоминание о Builder – шаблон проектирования программного обеспечения для создания объектов. […] Целью паттерна строителя является поиск решения антишаблона телескопического конструктора.

Ни одно из решений не является идеальным. Но что, если мы их объединим? Представьте себе, что Object Motherвозвращает a Test Data Builder. Имея это, вы можете затем манипулировать состоянием Builder перед вызовом операции терминала. Это своего рода шаблон.

17. Unit-тестирование: принципы поддержки и ведения тестов в проекте: как рефакторить, как добавлять, как удалять

Базовые принципы ведения Unit-тестов в проекте

Тесты должны предоставлять защиту от регрессий

- ✔ Увеличиваем покрытие нашего кода
- ✔ Уменьшаем количество транзитивных зависимостей
- ✔ Тестируем критичный код компонентов

Тесты должны быть устойчивы к рефакторингу

- ✔ Сама суть рефакторинга подразумевает, что тесты сломаться не должны
- ✔ А если сломались, то стоит задуматься, а стоило ли рефакторить
- ✔ false positive / false negative / ошибки 1 и 2 рода

18. Unit-тестирование: consistency, availability, partition tolerance – почему нельзя максимизировать все три параметра?

CAP - *В любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств*

- согласованность данных (англ. consistency) – во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
- доступность (англ. availability) – любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;
- устойчивость к разделению (англ. partition tolerance) – расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

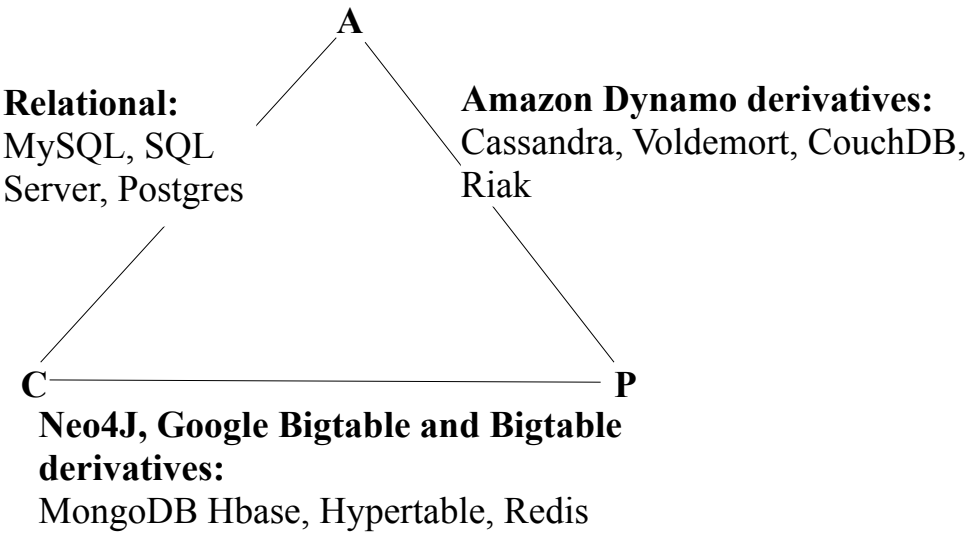

Рисунок 3: Три параметра.

19. Unit-тестирование: принцип сегрегации команд и запросов – когда нужны mock, когда stubs и почему

Принцип сегрегации команд и запросов

Принцип гласит, что метод должен быть либо командой, выполняющей какое-то действие, либо запросом, возвращающим данные, но не одновременно. Другими словами, **задавание вопроса не должно менять ответ**. Более формально, возвращать значение можно только чистым (т. е. детерминированным и не имеющим побочных эффектов) методом. Следует отметить, что строгое соблюдение этого принципа делает невозможным отслеживание количества вызовов запросов.

Теорема CAP и распределение тестов по уровням (не знаю почему оно именно тут...)

В любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств:

- ➔ согласованность данных (англ. Consistency) – во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
- ➔ доступность (англ. availability) – любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;
- ➔ устойчивость к разделению (англ. partition tolerance) – расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

(СМОТРИ тыкай ТУТ)

20. Integration Testing и System Integration Testing в чем разница для разработчика? Подход к написанию Integration Тестов

Интеграционное тестирование, на мой взгляд, **наиболее сложное** для понимания. Есть определение – это тестирование взаимодействия нескольких классов, выполняющих вместе какую-то работу. Однако как по такому определению тестировать не понятно. Причем важен **только результат взаимодействия**, а не детали и порядок вызовов. Поэтому на тесты не влияет рефакторинг кода. Не происходит избыточного или недостаточного тестирования – тестируются только те взаимодействия, которые встречаются при обработке реальных данных. Сами тесты легко поддерживать, так как спецификация хорошо читается и ее просто изменять в соответствии с новыми требованиями.

Тестирование интеграции системы (SIT) выполняется для проверки взаимодействия между модулями программной системы. Он занимается проверкой требований к программному обеспечению высокого и низкого уровня, указанных в Спецификации / данных требований к программному обеспечению и Документе по разработке программного обеспечения. Он также проверяет сосуществование программной системы с другими и тестирует интерфейс между модулями программного приложения. В этом типе тестирования модули сначала тестируются индивидуально, а затем объединяются в систему. Программные и / или аппаратные компоненты объединяются и постепенно тестируются, пока не будет интегрирована вся система.

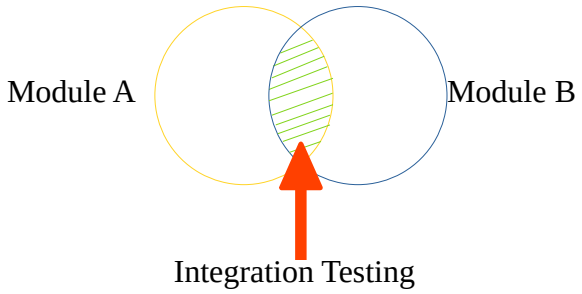

Рисунок 4: Тестирование системной интеграции (SIT).

21. System Testing – что такое? когда надо автоматизировать? надо ли?

Основной задачей системного тестирования является проверка как функциональных так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренное сценарий использования, отсутствующая или неверная функциональность, неудобства использования и т. д. Для минимизации рисков, связанных с особенностями поведения системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которой будет установлен продукт после выдачи.

Можно выделить два подхода к системному требованию:

- на базе требований для каждого требования пишутся тестовые случаи(test cases), проверяющие выполнение данного требования.
- На базе требований (use case based) случаев использования на основе предоставления о способах использования продукта создается случае использования системы (use cases). По конкретному случаю использования можно определить один или более сценарий в точка на проверку каждого сценария пишутся тест-кейсы (test cases), которые должны быть протестированы.

22. E2E тесты и тестирование процессов

E2E - это когда вы тестируете всю свою систему от начала до конца. Это включает в себя обеспечение того, чтобы все интегрированные части приложения функционировали и работали вместе, как ожидалось.

E2E тесты моделируют реальные сценарии пользователя, в основном проверяя, как реальный пользователь будет использовать приложение.

Примером для E2E для регистрации пользователя. Тест включал бы:

- ✔ Открытие сайта в браузере и поиск определенных элементов.
- ✔ Затем заполнить несколько форм регистрации.
- ✔ Затем убедитесь, что пользователь успешно создан.

Зачем это все?

- ✔ Для регрессии
- ✔ Для описания системы
- ✔ CI/CD

Тесты E2E позволяют нам охватывать разделы приложения, которые не проверяются unit тестами и интеграционными тестами. Это связано с тем, что unit тесты и интеграционные тесты покрывают отдельные части приложения и тестируют изолированную часть функционала.

Даже если эти части работают хорошо сами по себе, вы не уверены, будут ли они работать вместе. Таким образом, наличие набора E2E тестов поверх unit и интеграции позволяет нам тестировать все наше приложение.

23. Инструменты для E2E/Integration тестирования – Soap UI, Postman, Jmeter, Fiddler, Wireshark – для чего нужны и как используются

SoapUI

Представляет собой консольный инструмент, предназначенный для тестирования API и позволяющий пользователям легко тестировать API REST и SOAP, а также Web-сервисы. При помощи SoapUI, пользователи могут получить полный исходный документ и встроить предпочтительный набор функций, в дополнение к перечисленным ниже:

- ✔ Создать тест легко и быстро при помощи технологий перетаскивания объектов «мышкой» (Drag and drop) и метода «указания и щелчка» (Point-and-click)
- ✔ Быстро создать пользовательский код при помощи Groovy
- ✔ Мощное тестирование на основе данных: Данные загружаются из файлов, баз данных и Excel, поэтому они могут создать симуляцию взаимодействия пользователя и API.
- ✔ Создание комплексных сценариев и поддержка асинхронного тестирования
- ✔ Повторное использование скриптов: загрузка тестов и сканирование безопасности могут повторно использоваться в случае функционального тестирования всего в несколько шагов

Postman

Будучи изначально плагином браузера Chrome, теперь Postman расширяет свои технические решения вместе с оригинальными версиями как для Mac, так и для Windows. Postman является отличным выбором API тестирования для тех, кто

не желает иметь дела с кодировками в интегрированной среде разработки, используя тот же язык программирования, что и разработчик.

- ✔ Легкий в использовании клиент REST
- ✔ Богатый интерфейс делает этот инструмент простым и удобным в использовании
- ✔ Может использоваться как при автоматическом, так и при исследовательском тестировании
- ✔ Работает в приложениях для Mac, Windows, Linux и Chrome
- ✔ Обладает пакетом средств интеграции, таких как поддержка форматов Swagger и RAML
- ✔ Обладает функциями Run, Test, Document и Monitoring
- ✔ Не требует изучения нового языка программирования
- ✔ Позволяет пользователям легко делиться опытом и знаниями с другими членами команды, поскольку позволяет упаковывать все запросы и ожидаемые ответы и отправлять их коллегам.

Важно знать && [Общее]

Ни одно решение не может совместить все инструменты.

Некоторые могут посчитать, что свойств коммерческих продуктов (Postman, Tricentis Tosca,…) будет достаточно, однако цена вопроса будет служить серьезным сдерживающим фактором. Бесплатные и общедоступные решения (Rest-Assured, Karate DSL,…) являются довольно-таки приемлемыми, но требуют квалифицированных умений и много усилий для имплементации правильной

платформы. Инструменты, которые удерживают баланс между ценой и другими факторами (Katalon Studio, Postman), могут иметь недостатки для некоторых типов проектов, и эти недостатки требуют пристальной оценки.

24. Тесты с использованием БД: характеристика зависимости от БД, способы инициализации тестовой базы

База данных – внепроцессная зависимость, находится вне контекста SUT (system under test, объект тестирования)

Можно выделить несколько подходов к включению БД в тестовый контур:

- ✔ Хранить схему БД в системе контроля версий (**НЕ ВСЮ БД ЦЕЛИКОМ**)
- ✔ Использовать тестовую БД для каждого разработчика (общую или персональную)
- ✔ Реализовать механизм миграции данных

Хранить модель БД – антипаттерн:

- ✗ нет истории изменений
- ✗ нет golden source

Сложность хранения схемы*: нужно генерировать данные*

Для подготовки схемы БД необходимы:

- SQL скрипты с созданием таблиц, представлений, индексов, курсоров, хранимых процедур и т.д. и т.п.
- Справочная информация (reference data) – необходимая информация для инициализации объекта (например БД), не может быть изменена без нанесения вреда целостности объекта, который требуется инициализировать. Пример: типы данных для сущностей, связь между сущностями,

Сложности использования общей тестовой БД:

- запуски тестов разных разработчиков могут влиять на результаты выполнения
- не обратносовместимые изменения могут заблокировать выполнение тестов или работу БД в целом

✗ **Отдельная БД для каждого разработчика** – не всегда целесообразно с точки зрения распределения ресурсов

25. Тестирование API: информационная совместимость, программная совместимость, двоичная совместимость, что такое двоичный интерфейс приложений?

API расшифровывается как “интерфейс прикладного программирования” или “интерфейс программирования приложений”.

Он позволяет осуществлять связь и обмениваться данными между двумя отдельными модулями программы. Система программного обеспечения, реализующая API, содержит функции/подпрограммы, которые могут быть выполнены с помощью другого программного обеспечения.

Тестирование API полностью отличается от тестирования графического интерфейса и в основном концентрируется на слое бизнес-логики архитектуры программного обеспечения. GUI в API тестировании практически не нужен.

Вместо стандартных видов ввода пользовательских данных (заполнение форм) здесь для передачи данных используется программное обеспечение.

Для API тестирование нам потребуется само тестируемое приложение и приложение для работы с API. Также потребуется написать собственный код для работы с API.

Выходные данные API

- ✔ Любой тип данных
- ✔ Статус (true, или false)
- ✔ Вызов другой функции API

Виды тестов в тестировании API

1. Обзорное тестирование – тесты должны выполнить набор вызовов, задекларированных в API, чтобы проверить общую работоспособность системы;
2. Юзабилити-тестирование – проверяет, является ли API функциональным и обладает ли удобным интерфейсом, также проверяется интеграция с другими;
3. Тестирование безопасности – проверяет используемый тип аутентификации и шифрование данных с помощью HTTP ;
4. Автоматизированное тестирование – создание скриптов, программ или настройка приложений, которые смогут тестировать API на регулярной основе;
5. Документация – проверяется полнота описаний функций API, её понятность и, в свою очередь, является финальным результатом.

Лучшие практики тестирования API

1. Тест-кейсы должны быть сгруппированы по тестовым категориям;
2. Каждый тест должен включать декларацию тестируемой функции;
3. Выбор параметров должен быть явно упомянут в самом тесте;
4. Установка приоритетов вызова функций API;
5. Каждый тест должен быть самодостаточным и независимым друг от друга;
6. Особую осторожность следует соблюдать при обращении к функциям удаления, закрытия окна и прочим;
7. Вызов последовательности действий должен быть хорошо спланирован и выполнен;

8. Для обеспечения полного тестового покрытия создавайте тестовые случаи для всех возможных комбинаций входных данных.

Проблемы в тестировании API

1. Комбинация и выбор параметров
2. Отсутствие графического интерфейса
3. Валидация и верификация выходных данных в разных системах
4. Обязательная проверка обработки исключений
5. Тестировщикам необходимы знания в программировании

Разница API-тестированием и Unit-тестированием

Unit-тестирование	API-тестирование
Выполняется разработчиками	Выполняется тестировщиками
Тестируются отдельные модули	Тестируется end-to-end функционал
Разработчик имеет доступ к исходному коду	Тестировщик не имеет доступа к исх. коду
Возможно тестирование UI	Тестируется только API
Тестируется только базовый функционал	Тестируется весь функционал
Ограничен в возможностях	Широкие возможности
Запускается до заливки на SCV	Запускается после билда

Вывод

API состоит из множества классов / функций / процедур, которые представляют собой слой бизнес-логики. Если API не проверяется должным образом, то это может вызвать проблемы не только в применении API, но и в вызывающем приложении.

26. Тестирование API: чек-лист на тестирование API и что такое статический анализ кода?

Чек-лист (вроде). Перед нами подробный список полей, которые должны быть в тест-дизайне:

- ✔ Функция пользователя;
- ✔ Проверка;
- ✔ Ожидаемый результат;
- ✔ Планируемый временной отрезок;
- ✔ Комментарии;
- ✔ Статус теста;
- ✔ Дата проверки;
- ✔ Ревизия, тестовый стенд;
- ✔ Клиент для проверки.

Процесс тестирования API постоянно и неразрывно связан с проверкой имеющейся документации и составлением кейсов для тестов. Этим нежелательно пренебрегать, так как большой риск того, что самые важные клиентские сценарии не будут протестированы в полной мере.

Статический анализ

Статический анализ кода это процесс выявления ошибок и недочетов в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода. Остановимся на обзоре кода чуть подробнее.

Обзор кода (code review) – один из самых старых и надежных методов выявления дефектов. Он заключается в совместном внимательном чтении исходного кода и высказывании рекомендаций по его улучшению. В процессе чтения кода выявляются ошибки или участки кода, которые могут стать ошибочными в будущем. Также считается, что автор кода во время обзора не должен давать объяснений, как работает та или иная часть программы. Алгоритм работы должен быть понятен непосредственно из текста программы и комментариев. Если это условие не выполняется, то код должен быть доработан.

Задачи, решаемые программами статического анализа кода **можно разделить на 3 категории**:

1. Выявление ошибок в программах. Подробнее про это будет рассказано ниже.
2. Рекомендации по оформлению кода. Некоторые статические анализаторы позволяют проверять, соответствует ли исходный код, принятому в компании стандарту оформления кода. Имеется в виду контроль количества отступов в различных конструкциях, использование пробелов/символов табуляции и так далее.
3. Подсчет метрик. Метрика программного обеспечения - это мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций. Существует большое количество разнообразных метрик, которые можно подсчитать, используя те ли иные инструменты.

Другие преимущества статического анализа кода

1. Полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получают управление крайне редко. Такие участки

кода, как правило, не удается протестировать другими методами. Это позволяет находить дефекты в обработчиках редких ситуаций, в обработчиках ошибок или в системе логирования.

2. Статический анализ не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа. Это позволяет находить скрытые ошибки, которые могут проявить себя только через несколько лет. Например, это ошибки неопределенного поведения. Такие ошибки могут проявить себя при смене версии компилятора или при использовании других ключей для оптимизации кода. Другой интересный пример скрытых ошибок приводится в статье "Перезаписывать память - зачем?".
3. Можно легко и быстро обнаруживать опечатки и последствия использования Copy-Paste. Как правило, нахождение этих ошибок другими способами является кране неэффективной тратой времени и усилий. Обидно после часа отладки обнаружить, что ошибка заключается в выражении вида "strcmp(A, A)". Обсуждая типовые ошибки, про такие ляпы, как правило, не вспоминают. Но на практике на их выявление тратится существенное время.

Недостатки статического анализа кода

1. Статический анализ, как правило, слаб в диагностике утечек памяти и параллельных ошибок. Чтобы выявлять подобные ошибки, фактически необходимо виртуально выполнить часть программы. Это крайне сложно реализовать. Также подобные алгоритмы требуют очень много памяти и процессорного времени. Как правило, статические анализаторы ограничиваются диагностикой простых случаев. Более эффективным способом выявления

утечек памяти и параллельных ошибок является использование инструментов динамического анализа.

2. Программа статического анализа предупреждает о подозрительных местах. Это значит, что на самом деле код, может быть совершенно корректен. Это называется ложно-позитивными срабатываниями. Понять, указывает анализатор на ошибку или выдал ложное срабатывание, может только программист. Необходимость просматривать ложные срабатывания отнимает рабочее время и ослабляет внимание к тем участкам кода, где в действительности содержатся ошибки.

27. Динамический анализ кода и его связь с оценкой производительности

Проще говоря, статический анализ собирает информацию на основе исходного кода, а **динамический анализ основан на выполнении системы**, часто с использованием инструментария. На рынке существует множество инструментов динамического анализа, и самый известный из них – **отладчики**.

Преимущества динамического анализа

- ✔ Способен обнаруживать зависимости, которые невозможно обнаружить в статическом анализе. Например: динамические зависимости с использованием отражения, инъекции зависимостей, полиморфизма.
- ✔ Может собирать временную информацию.
- ✔ Имеет дело с реальными входными данными. Во время статического анализа трудно узнать, какие файлы будут переданы в качестве входных данных, какие WEB запросов поступят, какой пользователь нажмет и т. д.

Недостатки динамического анализа

- ✗ Может негативно сказаться на производительности приложения.
- ✗ Невозможно гарантировать полный охват исходного кода, так как его запуски основаны на взаимодействии с пользователем или автоматических тестах.

28. Виды тестирования производительности и профили нагрузки для каждого вида

Стресс-тестирование (Stress Test)

- Этот тест проводится первым. Нагрузка постепенно увеличивается до тех пор, пока приложение не перестанет работать корректно. В конце теста фиксируется количество пользователей, которое приложение выдерживало, соответствуя требованиям производительности, и сколько выдержать не смогло. Первое значение и будет пределом производительности вашего приложения. Часто этот вид тестирования проводится, если заказчик предвидит резкое увеличение нагрузки на систему. Например, для e-commerce это могут быть дни распродаж.

Нагрузочный тест (Load Test)

- Нагрузка на систему подается на протяжении 4-8 часов. В это время собираются метрики производительности: количество запросов в секунду, транзакций в секунду, время отклика от сервера, процент ошибок в ответах, утилизация аппаратных ресурсов и т д. Собранные метрики проходят проверку на соответствие заданным требованиям. В результате получаем ответ на вопрос: соответствует ли система требованиям производительности?
- Также на выходе имеем локализацию узких мест в производительности приложения и дефектов, подробное профилирование всех компонентов системы и утилизацию аппаратных ресурсов под целевой нагрузкой.

Тестирование на больших объемах данных (Volume Test)

- Данный вид тестирования помогает сделать прогноз относительно работоспособности приложения. Форма подаваемой нагрузки та же, что и при

нагрузочном тестировании. Задача теста – узнать, какое влияние окажет увеличение объема данных на систему. Таким образом, можем найти ответ на вопрос: как изменится производительность приложения спустя X лет, если аудитория приложения вырастет в Y раз?

Тестирование отказоустойчивости (Stability Test)

- Продолжительность нагрузки может варьироваться в зависимости от целей и возможностей проекта, доходя до семи дней и более. В результате получаем представление о том, как изменится производительность системы в течение длительного периода времени под нагрузкой, например, в течение недели. Снизится ли уровень производительности? Способно ли приложение выдерживать стабильную нагрузку без критических сбоев?

Тестирование масштабируемости (Scalability Test)

Профиль нагрузки тот же, что и при нагрузочном тестировании. Что получаем в результате? Ответы на следующие вопросы:

- Увеличится ли производительность приложения, если добавить дополнительные аппаратные ресурсы?
- Увеличится ли производительность пропорционально количеству добавленных аппаратных средств?

Какие же метрики собираются во время тестирования производительности?

Время отклика измеряется с момента отправки запроса к серверу до получения последнего байта от сервера.

Запросы в секунду. Клиентское приложение формирует HTTP-запрос и отправляет его на сервер. Серверное ПО данный запрос обрабатывает, формирует

ответ и передает его обратно клиенту. Общее число запросов в секунду и есть интересующая нас метрика.

Транзакции в секунду. Пользовательские транзакции – это последовательность действий пользователя в интерфейсе. Сравнивая реальное время прохождения транзакции с ожидаемой (или количество транзакций в секунду), вы сможете сделать вывод о том, насколько успешной системой было пройдено нагрузочное тестирование.

Число виртуальных пользователей в единицу времени также позволяет выяснить, отвечает ли производительность приложения заявленным требованиям. Если вы все делаете правильно и ваши сценарии максимально приближены к поведению пользователя, то один виртуальный пользователь будет равен одному реальному пользователю.

Процент ошибок рассчитывается как отношение невалидных ответов к валидным за промежуток времени.

29. BDD/TDD – плюсы, минусы, место по уровням в пирамиде тестирования, связь с паттерном AAA, как используются mock/stub

В TDD (Разработка через тестирование), тест написан для проверки реализации функциональности, но по мере развития кода тесты могут давать ложные результаты. BDD (Разработка, управляемая поведением) также является подходом, ориентированным на сначала тестирование, но отличается тем, что проверяет фактическое поведение системы с точки зрения конечных пользователей. Это подходы к разработке, когда сначала пишутся тесты, а потом код.

- ***DD** (*что-то* Driven Development) – разработка, основанная на чем-то.
- **TDD** (Test Driven Development) – Разработка на основе тестов.
- **BDD** (Behavior Driven Development) – Разработка на основе поведения. **BDD**, на самом деле, является расширением TDD-подхода. Тем не менее, они предназначены для разных целей и для их реализации используются разные инструменты. В разных командах эти понятия могут интерпретировать по-разному, и часто возникает путаница между ними.

Разница

- TDD хорошо подходит для юнит-тестирования, т.е. для проверки работы отдельных модулей самих по себе. BDD – для интеграционного (т.е. для проверки, как отдельные модули работают друг с другом) и e2e (т.е. для проверки всей системы целиком) тестирования.
- TDD: тесты сразу реализуются в коде, для BDD чаще всего описываются шаги на языке, понятном всем, а не только разработчикам.

- TDD: юнит-тесты пишут сами разработчики. BDD требует объедения усилий разных членов команды. Обычно тест-кейсы (шаги) описываются ручным тестировщиком или аналитиком и воплощаются в код тестировщиком-автоматизатором. В нашей команде мы (фронтенедеры) описываем шаги вместе с тестировщиками, а код тестов пишет фронтенд-команда.
- TDD проверяет работу функций, BDD – пользовательские сценарии.

Как к задаче, используя TDD подход

1. Пишем тест, в котором проверяем, что функция getCatFood() возвращает нужные значения в разных ситуациях
2. Проверяем, что тесты упали (кода еще нет)
3. Пишем код функции очень просто – так чтобы тесты прошли
4. Проверяем, что тесты прошли
5. На этом шаге можем задуматься о качестве кода. Можем спокойно рефакторить и изменять код как угодно, т.к. у нас есть тесты, которые с уверенностью скажут, что мы где-то ошиблись
6. Повторяем все вышеуказанные шаги еще раз

Как подойти к этой задаче, используя BDD подход

1. Процесс начинается с того что пользователь открывает форму
2. Нам нужно протестировать числа которые выдает форма
3. Нам нужно ввести 10–20 разных значений
4. Проверка в данном случае это нажатие на Submit кнопку и проверка значения

5. Тест пройдет если результат на форме соответствует “правильным” значениям

Вывод

Хотя мы говорим, что BDD – **лучший подход**, мы не должны забывать, что BDD на самом деле произошел от TDD как способ устранения недостатков TDD. Таким образом, нет абсолютно никакого вреда в реализации обоих подходов - один для поддержки качества кода, который пишет разработчик, а другой для поддержки поведения системы, определенного владельцем продукта.

30. DDD и DDT – разные варианты подготовки данных, входные\выходные состояния объекта тестирования

Предметно-ориентированное проектирование не является какой-либо конкретной технологией или методологией. DDD – это набор правил, которые позволяют принимать правильные проектные решения. Данный подход позволяет значительно ускорить процесс проектирования программного обеспечения в незнакомой предметной области.

Предметно-ориентированное проектирование (реже проблемно-ориентированное, англ. Domain-driven design, DDD) – это набор принципов и схем, направленных на создание оптимальных систем объектов. Процесс разработки сводится к созданию программных абстракций, которые называются моделями предметных областей. В эти модели входит бизнес-логика, устанавливающая связь между реальными условиями области применения продукта и кодом.

Подход DDD особо полезен в ситуациях, когда разработчик не является специалистом в области разрабатываемого продукта. К примеру: программист не может знать все области, в которых требуется создать ПО, но с помощью правильного представления структуры, посредством предметно-ориентированного подхода, может без труда спроектировать приложение, основываясь на ключевых моментах и знаниях рабочей области.

Основная цель Domain-Driven Design — это борьба со сложностью бизнес-процессов, их автоматизации и реализации в коде. «Domain» переводится как «предметная область», и именно от предметной области отталкивается разработка и проектирование в рамках данного подхода.

Ключевым понятием в DDD является «единый язык» (ubiquitous language). Ubiquitous language способствует прозрачному общению между участниками проекта. Единый он не в том смысле, что он один на все случаи жизни. Как раз наоборот. Все участники общаются на нём, всё обсуждение происходит в терминах единого языка, и все артефакты максимально должны излагаться в терминах единого языка, то есть, начиная от ТЗ, и, заканчивая кодом.

Минусы DDD

- ✗ требуется высокая квалификация разработчиков, особенно, на старте проекта;
- ✗ не все клиенты готовы пойти на такие затраты, DDD нужно учиться всем участникам процесса разработки.

DDT

Data Driven Testing (DDT) – подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще всего применимо к backend тестированию), при котором тест умеет принимать набор входных параметров, и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров. Такое сравнение и есть assert такого теста. Притом как часть входных параметров, могут передаваться опции выполнения теста, или флаги, которые влияют на его логику.

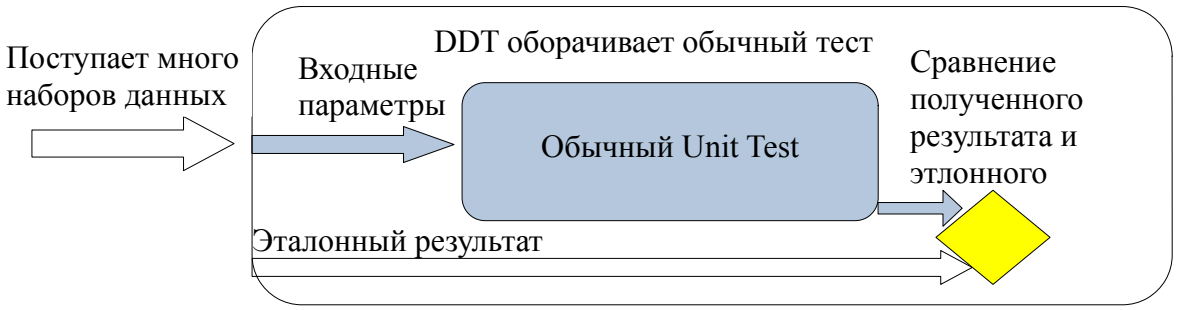

Рисунок 5: Схема DDT.

Вход/Выход

- ✔ Частью входного описания может быть состояние базы данных, т.е. в функционал DDT будет входить setUp базы данных. Это состояние можно брать из SQL дампа, а можно генерировать программно на java (второй способ легче поддерживать).
- ✔ Частью эталонного выходного состояния опять-таки может быть состояние базы данных. Но бывает, что всё намного сложнее, и нам нужно проверить не просто состояние, а последовательность вызовов, ивентов, и даже, контекст после каждого вызова внутри системы, тогда хорошей идеей может быть построение дерева вызовов во время прогонки теста, т.е. логирование нужного в нужном формате (к примеру, XML или JSON), и тогда эталонными данными будет проверенное ранее дерево вызовов. Кстати, эталонное дерево вызовов можно записать в первой прогонке, потом вручную проверить, что оно – правильное, и далее – уже его использовать для тестов.

Плюсы DDT

Особым плюсом хорошо-спроектированного DDT является возможность ввода входных значений и эталонного результата в виде, удобном для всех ролей на проекте – начиная от мануального тестировщика и заканчивая менеджером (тест менеджером) проекта, и даже, product owner-а (на практике автора такие случаи случались). Соответственно, когда Вы способны загрузить мануальных тестировщиков увеличением покрытия и увеличением наборов данных – это удешевляет тестирование. А также в целом, удобный и понятный формат позволяет более наглядно видеть, что покрыто, а что нет, это – по сути, и есть документация тести-

рования. К примеру, это может быть XLS файл с понятной структурой (хотя чаще всего properties файла достаточно).

Нюансы DDT

Отмечу, что при определённом подходе в использовании, Robot Framework тесты могут быть очень DDT-ориентированными, и давать массу выгод.

Простейшая форма DDT – параметризованные тесты в JUnit, когда с помощью Data Providers в тест поставляются наборы данных для тестирования.

DDT – не панацея, но грамотное применение в нужных местах помогает очень и очень.