Программная инженерия

Тестирование и отладка ПО

Вопросы

- Отладка и тестирование ПО: основные понятия
- Виды тестирования:
  - функциональное / модульное тестирование,
  - регрессивное тестирование,
  - нагрузочное тестирование
- Приемы и технологии тестирования
- Интеграция
- Верификация и валидация
- Инструментальные средства поддержки тестирования
- Экстремальное программирование: разработка через тестирование

Определения

Классификация видов

ОТЛАДКА И ТЕСТИРОВАНИЕ ПО

Пара цитат по теме

- «Тестирование программ может использоваться для демонстрации наличия ошибок, но оно никогда не покажет их отсутствие»
- «Если отладка — процесс удаления ошибок, то под программированием можно понимать процесс их внесения»
- Эдсгер Вибе Дейкстра

История подхода – 1/2

- 60-е: много внимания уделялось «исчерпывающему» тестированию, которое должно проводиться с использованием всех путей в коде или всех возможных входных данных
- 70-е: тестирование ПО это «процесс, направленный на демонстрацию корректности продукта»
- 80-е: появилось понятие «предупреждение дефектов»

История подхода – 2/2

- Начало 90-х: в понятие «тестирование» включили планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений → переход от тестирования к обеспечению качества, охватывающего весь цикл разработки ПО
- Середина 90-х: с развитием Интернета и распространением распределенных систем популярным стало «гибкое тестирование»
- 2000-е: добавлено понятие «оптимизация бизнес-технологий». Идея в оценке и максимизации значимости всех этапов жизненного цикла ПО для достижения необходимого уровня качества, производительности, доступности

Определения – 1/4

- Тестирование – процесс выполнения системы, программы или компонента, при определенных [тестировщиком] условиях, для наблюдения и для оценки некоторых аспектов их функционирования
- Выявление дефектов – анализ ПО, выявляющий несоответствия между существующими особенностями выполнения ПО и требованиями к ПО в процессе тестирования
- Тестовый случай – это набор входных данных, условий выполнения системы и ожидаемых результатов, разработанных для определенной цели, как то:
  - Для оценки определенного пути выполнения приложения или
  - Для проверки соответствия определенному требованию

Определения – 2/4

- Верификация — это подтверждение соответствия конечного продукта предопределённым эталонным требованиям
  - Внутренний процесс управления качеством
  - «Вы создали продукт таким, каким и намеревались его сделать»

Определения – 3/4

- Валидация подтверждает, что приведены доказательства того, что требования конкретного внешнего потребителя или пользователя продукта, услуги или системы удовлетворены
  - Согласно стандарту ГОСТ Р ИСО 9000-2008 (ISO 9000:2005): «Подтверждение на основе представления объективных свидетельств того, что требования, предназначенные для конкретного использования или применения, выполнены»
  - Внешний процесс
  - «Вы создали правильный продукт»

Определения – 4/4

- Тестирование – процесс выполнения системы, демонстрирующий соответствие требованиям ТЗ
  - При этом могут тестироваться: функциональность системы, ее характеристики, другие свойства
  - См. Валидация
- Отладка – процесс выполнения системы, позволяющий обнаружить ошибки функционирования системы (ее явные или неявные несоответствия требованиям ТЗ) и локализовать причины этих ошибок или описать возможные пути избежания их проявления (work-around)
  - См. Верификация

Виды тестирования – 1/4

- По объекту тестирования:
  - Функциональное
  - Тестирование производительности
    - Нагрузочное
    - Стресс-тестирование
    - Тестирование стабильности
    - Конфигурационное тестирование
  - Юзабилити-тестирование
  - Тестирование интерфейса пользователя
  - Тестирование безопасности
  - Тестирование локализации
  - Тестирование совместимости

Виды тестирования – 2/4

- По знанию системы:
  - Черный ящик
  - Серый ящик
  - Белый ящик
- По степени автоматизации
  - Ручное
  - Автоматизированное
  - Автоматическое

Виды тестирования – 3/4

- По времени проведения:
  - Альфа-тестирование
  - Дымовое
  - Регрессионное
  - Тестирование новой функции (new feature testing)
  - Подтверждающее тестирование
  - Приёмочное тестирование
  - Бета-тестирование

Виды тестирования – 4/4

- По степени изолированности:
  - Компонентное
  - Интеграционное
  - Системное
- По предполагаемому результату
  - Позитивное тестирование
  - Негативное тестирование
- По степени подготовленности к тестированию
  - Формальное тестирование (по документации)
  - Интуитивное тестирование (ad hoc testing)

По изолированности

ВИДЫ ТЕСТИРОВАНИЯ ПО

По изолированности – 1/3

- Компонентное тестирование (юнит-тестирование): тестируются отдельные компоненты системы для определения корректности их работы
  - Каждый компонент тестируется независимо от других
  - Компонентами могут быть отдельные функции, классы, библиотеки
  - Размер приложения:
    - Обычно под юнитом понимают часть кода, которая не может выполниться самостоятельно
    - Под компонентом понимается отдельное приложение из состава РИС
    - Возможны и промежуточные варианты ^_^

По изолированности – 2/3

- Интеграционное тестирование проводится когда компоненты объединяются интерфейсами и получившаяся система тестируется как единое целое
  - Выявляются ошибки взаимодействия компонентов и проблемы определения интерфейсов
- Системное тестирование – «высшая стадия» интеграционного тестирования, когда все компоненты объединены так, что образуют конечный вариант РИС
  - NB! Иногда использование РИС предполагает комбинирование компонентов в разных вариантах

По изолированности – 3/3

- Приемочное тестирование: финальная стадия процесса тестирования. Вид системного тестирования
  - Система тестируется на данных, предоставленных заказчиком (пользователем)
  - Могут выявиться ошибки интерпретации данных и требований к системе, так как система может вести себя по-другому на реальных данных

По объекту тестирования

ВИДЫ ТЕСТИРОВАНИЯ ПО

Функциональное тестирование

- Функциональное — тестирование ПО для проверки реализации функциональных требований
- Функциональные требования включают в себя:
  - Функциональная пригодность (suitability)
  - Точность (accuracy)
  - Способность к взаимодействию (interoperability)
  - Соответствие стандартам и правилам (compliance)
  - Защищённость (security)

Тестирование производительности – 1/2

- Тестирование производительности проводится с целью определения, как [быстро] работает вычислительная система или её часть под определённой нагрузкой
- Также может служить для проверки и подтверждения таких свойств системы, как:
  - Масштабируемость
  - Надёжность
  - Потребление ресурсов

Тестирование производительности – 2/2

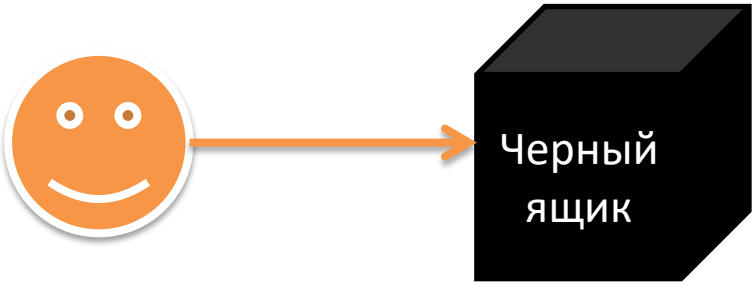
- В тестировании производительности различают следующие направления:
  - Нагрузочное (load)
    - Определяет способность обработать заданное количество запросов и/или объем данных за заданное время
  - Стресс (stress)
    - Определяет пределы устойчивости системы и ее поведение при нагрузках выше заданных
  - Тестирование стабильности (endurance or soak or stability)
    - Проверяет способность выполнять функции при длительной работе
  - Конфигурационное (configuration)
    - Определяет влияние различных конфигураций системы на ее производительность

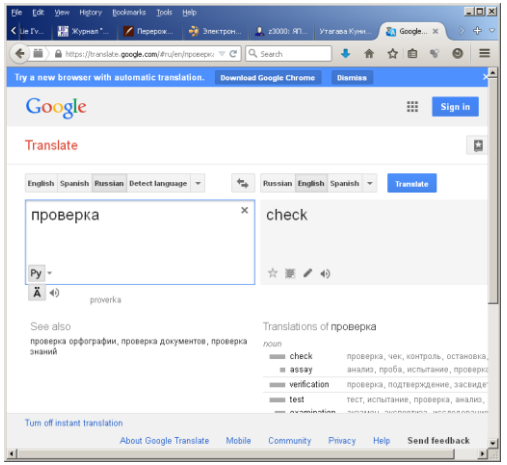
По знанию системы

ВИДЫ ТЕСТИРОВАНИЯ ПО

Виды: черный ящик – 1/2

- Тестировщик не имеет представления о исходном коде, таблицах БД, и т.д. В этом случае система для него является чёрным ящиком.
- При тестировании по стратегии чёрного ящика руководствуются спецификацией системы, оценивается её функциональность
- Обычно тестировщиками являются пользователи (или бета-тестерами)



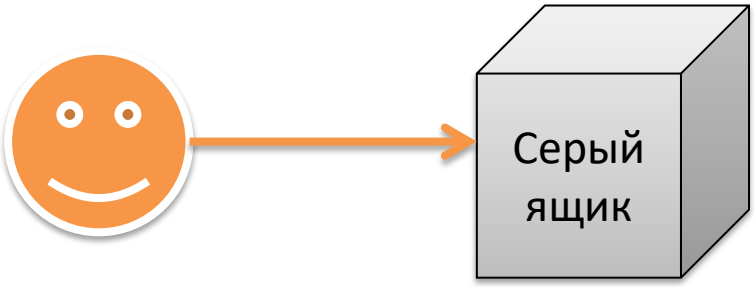


Виды: черный ящик – 2/2

- Функции системы
  - Делает ли система то, что должна делать согласно ТЗ (спецификации)?
  - Правильно ли система реагирует на рутинные действия пользователя?
- Контроль реакций
  - Как система реагирует на некорректный ввод данных?
- Выходные данные
  - Правильно ли формируются результаты работы системы?
- Исчерпывающее тестирование (перебор всех возможных комбинаций) как правило невозможно. Обычно выбирают несколько тестов, максимально покрывающих проверяемый функционал

Виды: серый ящик – 1/2

- Тестировщик знаком со спецификацией и ключевыми элементами проекта. Он проверяет не только что делает система, но и то, как она это делает. В этом случае разрабатываемая система является серым ящиком
- Обычно тестировщиками являются другие разработчики (не вовлеченные в разработку конкретного ПО)

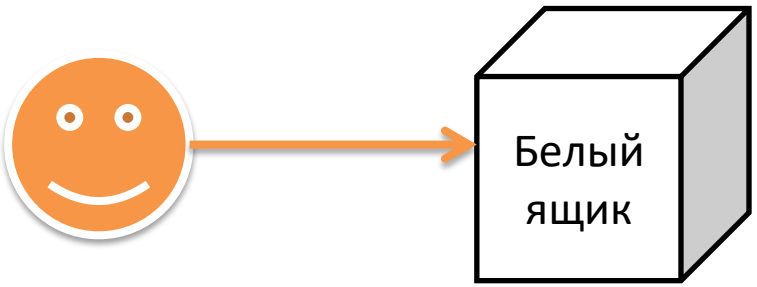


Виды: серый ящик – 2/2

- Контроль ведения аудита вводимой информации:
  - Ведутся ли логи во время использования системы?
- Проверка информации, создаваемой самой системой:
  - Временных меток (timestamps) с учётом временных зон, хэш-сумм, внешних ключей БД и т.д.
- Удаление временных файлов и очистка памяти
  - Не возникает ли утечка памяти во время исполнения программы?
- Эта стратегия объединяет в себе цели белого и чёрных ящиков
- Программно её можно реализовать, добавив специализованные вызовы (запись в журнал, assert, и т.д.)

Виды: белый ящик – 1/2

- Тестировщик имеет доступ к исходному коду и/или элементам детального дизайна. Он определяет уместность паттернов проектирования, обоснованность структуры классов. В этом случае разрабатываемая система является белым ящиком
- Обычно тестировщиками являются «соседи» и/или непосредственные руководители разработки



Виды: белый ящик – 2/2

- Тестирование всех возможных веток в коде:
  - В каком случае условие условного оператора не будет выполняться?
  - Может ли цикл стать бесконечным?
- Надлежащая обработка всех возможных исключительных ситуаций:
  - Намеренно спровоцировать вызов всех возможных, проверяем их возникновение и обработку
- «Краевые» тесты:
  - Как поведёт себя метод, если передать нулевые данные или ничего не передавать?
  - Что будет, если не хватит ресурсов?
- С тестированием по стратегии белого ящика тесно связано вычисление процента тестового покрытия

Сравнение «ящиков»

Входные данные тестирования определяются	Результат
Требованиями	Выходные данные сравниваются с заданными требованиями
Требованиями и элементами дизайна	Как для «черного» и «белого» ящиков
Элементами проектирования	Подтверждение ожидаемого поведения

По степени автоматизации

ВИДЫ ТЕСТИРОВАНИЯ ПО

Определения

- Автоматизация тестирования ПО заключается в использовании готовых программных средств для выполнения тестов и проверки результатов выполнения
- Виды автоматизированного тестирования:
  - Тестирование на уровне кода
    - Например, модульное тестирование
  - Тестирование пользовательского интерфейса
    - Например, имитация действий пользователя с помощью специальных приложений (тестовых фреймворков)
  - Тестирование производительности

Тестирование GUI – 1/4

- Утилиты записи и воспроизведения
  - Записывают действия тестировщика во время ручного тестирования
  - Преимущества:
    - Универсальность
    - Простота и распространенность
  - Недостатки:
    - Переделки ПО требуют перезаписи действий оператора
    - Не контролируют результаты и реакцию тестируемого ПО

Тестирование GUI – 2/4

- Написание сценария (scripting)
  - Действия оператора описываются на специальных языках
  - Преимущества:
    - Гибкость
    - Относительная распространенность
  - Недостатки:
    - Требуют работы специалистов (не только тестировщиков)
    - Изменения в ПО требуют переписывания скриптов

Тестирование GUI – 3/4

- Тестирование управляемое данными (Data-driven testing)
  - Развитие технологии скриптов. Тестовые скрипты выполняются и верифицируются на основе данных, которые хранятся в центральном хранилище данных или БД
  - Преимущества:
    - Можно менять входные данные без переписывания скриптов
    - Анализируются результаты выполнения программы
  - Недостатки:
    - Сложность
    - Дороговизна

Тестирование GUI – 4/4

- Тестирование по ключевым словам (keyword-based)
  - Создание тестов из более мелких кусочков. Конечный тест представляет собой не отдельную программу, а суперпрограмму, оперирующую более мелкими процедурами
  - Предполагается, что тестовый фреймворк распознает операции по ключевым словам, реализованным во фреймворке. Таким образом, тесты могут создавать не программисты, описывая их на языке, похожем на ЕЯ

Модульное тестирование – 1/4

- Модульное тестирование (unit testing) — процесс, позволяющий проверить корректность отдельного модуля исходного кода
- Что делается: пишется тест для каждой [нетривиальной] функции или метода
- Для чего:
  - Облегчает отладку (обнаружение и устранение ошибок)
  - Можно быстро проверить, что изменение кода не привело к регрессии (появлению ошибок в отлаженном коде)

Модульное тестирование: применение – 2/4

- Поощрение улучшения кода
  - Можно не опасаться, что улучшая код внесешь ошибку
- Упрощение интеграции
  - Отлаженные куски лучше интегрируются
- Документирование кода
  - Тестовые процедуры могут использоваться как «черновики» для сопрягаемых модулей
- Отделение интерфейса от реализации
  - При написании модульных тестов легко обнаруживаются недопустимые кросс-модульные связи
- **Экстремальное программирование же!**

Модульное тестирование: ограничения – 3/4

- Сложный код
  - Для сложного алгоритма сложность и размер теста очень велики
- Нет простого способа проверить результат
  - Например, результат выполнения модели заранее неизвестен
- Проблемы масштабируемости
  - При интеграции модулей может выясниться, что при сопряжении проверенных модулей они совместно работают неверно (или неэффективно)
- Низкая культура программирования
  - Для внедрения требуется (как минимум) использование средств контроля версий ПО

Модульное тестирование: инструменты – 4/4

- Java
  - JUnit (JUnit.org)
- C++
  - Boost Test
  - Google C++ Testing Framework
- .NET
  - Nunit
- Perl
  - Test::Simple

Недостатки автоматизации

- Трудоемкость
  - Ресурсы тратятся на создание и обновление самих тестов
- Кто следит за проверяющими?
  - Подтверждение результатов автоматизированного тестирования
- Ограниченность
  - «Живой» тестировщик «уронит» программу так, как никому и в голову не придет
- Искусственность
  - Часть обнаруживаемых ошибок может в реальной жизни никогда не проявиться и потраченные на их исправление ресурсы будут потрачены зря

Преимущества автоматизации

- Повторяемость результатов
- Скорость
- Улучшение качества кода

По времени проведения

ВИДЫ ТЕСТИРОВАНИЯ ПО

Виды тестирования – 3/4

- По времени проведения:
  - Дымовое
  - Альфа-тестирование (проверка бизнес-логики)
  - Альфа-тестирование (внутренняя сдача)
  - Приёмочное тестирование
  - Бета-тестирование
  - Тестирование новой функции (new feature testing)
  - Регрессионное
  - Подтверждающее тестирование

Что такое: α, β, ɣ

- Альфа-тестирование:
  - Имитация реальной работы с системой выполняемая штатными разработчиками, либо
  - Реальная работа с системой заказчиком
  - Обычно в начале разработки, но иногда в конце как внутренняя сдача
- Бета-тестирование:
  - Распространение предварительной версии для того, чтобы убедиться, что продукт содержит мало ошибок
  - Иногда используется для того, чтобы получить обратную связь с будущими пользователями

«Мелкие» тесты

- Дымовое тестирование: минимальный набор тестов на явные ошибки
  - Обычно выполняется самим разработчиком
  - Не проходящую такой тест программу не имеет смысла отдавать на более глубокое тестирование
  - Например: установка ПО на «чистую» машину
- Тестирование новой функции:
  - Изолированный тест, проверяющий работу отдельной (добавленной) функции системы
  - Обычно используется при доработках системы или при пошаговой разработке системы

Регрессионное тестирование – 1/2

- Регрессионное тестирование – общее название для всех видов тестирования, направленных на обнаружение ошибок в уже протестированных участках исходного кода
- Выделяют:
  - new bug-fix
    - проверка исправления вновь найденного дефекта
  - old bug-fix
    - проверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова
  - side-effect bug-fix
    - проверка того, что не нарушилась работоспособность работающей ранее функциональности, если её код мог быть затронут при исправлении некоторых дефектов в другой функциональности

Регрессионное тестирование – 2/2

- Методы регрессионного тестирования:
  - Повторные прогоны предыдущих тестов
  - Проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода
- Для этого создают специализированные (желательно автоматические) тесты для проверки наличия отдельных ошибок

Особенности РИС

Инструменты

Масштабируемость

ОСОБЕННОСТИ ТЕСТИРОВАНИЯ РИС

Особенности тестирования РИС – 1/2

- Количество взаимодействующих элементов > 1
  - Требуется управлять в процессе выполнения теста
  - Сложность анализа результатов – он распределен
- Распределенность устройств
  - Распределенный характер скрипта
  - Синхронизация действий операторов и/или элементов ПО (входящих в РИС и других)
  - Сеть связи
- Доступность специального оборудования
  - Наличие и доступность оборудования (в достаточном количестве)
  - Наличие для него средств отладки / тестирования

Особенности тестирования РИС – 2/2

- Уникальность разрабатываемых систем
- «Многомерность» тестируемых параметров
- Сложно доказать допустимость выбранных ограничений / исходных данных / условий
- При разработке middleware предсказать конкретные варианты использования практически невозможно

Инструменты тестирования РИС

- Системы мониторинга РКС
- Инструменты мониторинга сети
  - Настраиваемость на специализированные протоколы обмена
- Средства мониторинга и отладки, входящие в состав middleware
- Средства журналирования (с возможностью воспроизведения)
- Средства анализа журналов

Тестирование масштабируемости

- Мера оценки производительности РИС не всегда четко определена
- Все особенности тестирования РИС проявляются в крайней степени
- Включает в себя:
  - Тестирование производительности
  - Устойчивость структуры РИС к увеличению числа элементов

Определения

Основные тезисы

Test-Driven Development

ЭКСТРЕМАЛЬНОЕ ПРОГРАММИРОВАНИЕ

XP: Введение – 1/2

- Экстремальное программирование (Extreme Programming, XP) — одна из гибких методологий разработки программного обеспечения
- Основная идея:
  - Собрать полезные традиционные методы и практики разработки ПО
  - Поднять их на новый «экстремальный» уровень

XP: Введение – 2/2

- Например:
  - Практика выполнения ревизии кода
    - Проверка одним программистом кода, написанного другим программистом
  - «Экстремальный» вариант → «парное программирование»
    - Один программист занимается кодированием, а его напарник в это же время непрерывно просматривает только что написанный код
    - Затем меняются в парах и между парами

XP: Основные приемы – 1/3

- Всего 12 приемов, разбитых на 4 группы
- Короткий цикл обратной связи (Fine-scale feedback)
  - **Разработка через тестирование (Test-driven development)**
  - Игра в планирование (Planning game)
  - Заказчик всегда рядом (Whole team, Onsite customer)
  - Парное программирование (Pair programming)

XP: Основные приемы – 2/3

- Непрерывность процесса разработки
  - Непрерывная интеграция (Continuous integration)
  - Рефакторинг (Design improvement, Refactoring)
  - Частые небольшие релизы (Small releases)
- Социальная защищённость программиста (Programmer welfare):
  - 40-часовая рабочая неделя (Sustainable pace, Forty-hour week)

XP: Основные приемы – 3/3

- Согласованное представление системы (понимание, разделяемое всеми)
  - Простота (Simple design)
  - Метафора системы (System metaphor)
  - Коллективное владение кодом/ шаблонами проектирования (Collective code/patterns ownership)
  - **Стандарт кодирования (Coding standard or Coding conventions)**

Test-Driven Development – 1/2

- Разработка через тестирование (test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки:
  - Сначала пишется тест, покрывающий желаемое изменение
  - Затем пишется код, который позволит пройти тест
  - В конце проводится рефакторинг нового кода к соответствующим стандартам

Test-Driven Development – 2/2

- Появилась 1999 году как концепция «сначала тест» (test-first) из экстремального программирования
- Позже выделилась как независимая методология

TDD: Технология – 1/5

- Тест — [автоматическая] процедура, которая подтверждает, либо опровергает работоспособность [небольшого куска] кода. Он содержит проверки условий, которые могут либо выполняться, либо нет
  - Когда условия выполняются, говорят, что тест пройден
  - Прохождение теста подтверждает поведение, предполагаемое программистом
- Тест может выполняться вручную или автоматически

TDD: Технология – 2/5

- TDD требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду ***непосредственно перед*** написанием самого кода
- Разработчики часто пользуются средствами для [автоматизированного] тестирования (testing frameworks) для создания и автоматизации запуска наборов тестов

TDD: Технология – 3/5

- Принцип keep it simple, stupid (KISS, «делай проще, дурачок»)
- Принцип «вам это не понадобится» (you ain’t gonna need it, YAGNI)
  - отказ добавления функциональности, в которой нет непосредственной надобности

→ Дизайн может быть чище и яснее, при написании лишь того кода, который необходим для прохождения теста

TDD: Технология – 4/5

- Принцип «подделай, пока не сделаешь» (fake it till you make it) – тесты должны писаться для тестируемой функциональности. Преимущества:
  - Разработчик с самого начала обдумывает, как приложение будет тестироваться → Помогает убедиться, что приложение пригодно для тестирования
  - Способствует тому, что тестами будет покрыта вся функциональность. Иначе разработчики склонны переходят к реализации следующей функциональности, не протестировав эту

TDD: Технология – 5/5

- Приём «красный/зеленый/рефакторинг»:
  - Красный: Сначала новый тест НЕ проходит (кода еще нет)
  - Зеленый: Затем пишется код, который его проходит
  - Рефакторинг: Последующие изменения кода (тест уже есть)

TDD: Применимость – 1/2

- Полный набор тестов для всего приложения **чрезвычайно** велик

→Обычно модульные тесты покрывают критические и нетривиальные участки кода

- Например:
  - Код, который подвержен частым изменениям
  - Код, от работы которого зависит работоспособность большого количества другого кода
  - Код с большим количеством зависимостей

TDD: Применимость – 2/2

- Для [эффективного] применения TDD:
  - Архитектура разрабатываемого ПО должна базироваться на использовании множества сильно связанных компонентов, которые слабо сцеплены друг с другом

  → TDD влияет на дизайн программы
- Опираясь на тесты, разработчики могут быстрее представить функциональность необходимую пользователю

  → Детали интерфейса появляются задолго до окончательной реализации решения

TDD: Ограничения

- Сложное поведение будущего кода сложно описать в терминах тестов
- Даже мелкие изменения структуры требуют изменения тестов
- Неясно, как проверить правильность самих тестов

TDD: Преимущества – 1/4

- Если тесты достаточно мелки, то если некоторые из тестов неожиданно перестают проходить, откат к версии, которая проходит все тесты, может быть более продуктивным, нежели отладка
- Если тесты описывают функциональность, нужную пользователю, то разработчик продумывает детали интерфейса до реализации

TDD: Преимущества – 2/4

- Создается код, более приспособленный для тестирования. Например:
  - Исключаются глобальные переменные, синглетоны
  - Классы создаются менее связанными и легкими для использования
  - Модульное тестирование способствует формированию четких и небольших интерфейсов

TDD: Преимущества – 3/4

- Суммарные затраты [времени и денег] на разработку при разработке через тестирование обычно оказывается меньше, чем в обычном случае
  - Тесты защищают от ошибок → Снижается время, затрачиваемое на отладку
- Рефакторинг кода упрощается
  - Тесты позволяют производить его без риска испортить код

TDD: Преимущества – 4/4

- Набор тестов достаточно полон
- Тесты могут использоваться в качестве документации
  - Примечание: причем соответствующей реальному состоянию дел (sic!)

TDD: Недостатки – 1/3

- Некоторые вещи нельзя или сложно протестировать универсальным способом:
  - Безопасность данных
  - Взаимодействия между процессами
- TDD сложно применять для тестирования функциональности
  - Интерфейсы пользователя
  - Программы, работающие с БД
  - ПО, зависящего от специфической конфигурации сети

TDD: Недостатки – 2/3

- Тесты и код создает один человек / коллектив → Если разработчик неправильно истолковал требования к приложению, то и тест, и тестируемый модуль будут содержать ошибку
- Большое количество используемых тестов могут создать ложное ощущение надежности, приводящее к меньшему количеству действий по контролю качества

TDD: Недостатки – 3/3

- Плохо написанные тесты сложно поддерживать
  - Например, hardcoded строки с сообщениями об ошибках
- Поддерживать приходится систему (разрабатываемое ПО + тесты)
  - Исходные тесты становятся всё более ценными с течением времени. При попытке изменить этот набор на поздних этапах может привести к необнаруживаемым пробелам в покрытии тестами

ATDD

- Разработка через приёмочное тестирование (Acceptance Test-driven development, ATDD) – разновидность TDD, в котором требования ТЗ автоматизируются в тесты
- Такой процесс позволяет гарантировать, что приложение удовлетворяет ТЗ

Все пропало

Ужас-ужас

ПРАКТИЧЕСКИЕ ЗАДАНИЯ