Распределенные информационные системы

Тестирование и отладка РИС

Вопросы

- Общие определения (повтор?)
- Виды тестирования
- Особенности тестирования РИС
- Тестирование масштабируемости
- Инструменты тестирования РИС, мониторинг и диагностика

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

- Один из создателей структурного программирования
- Противник рассмотрения программирования как процесса «кодирование – тестирование – исправление ошибок»

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

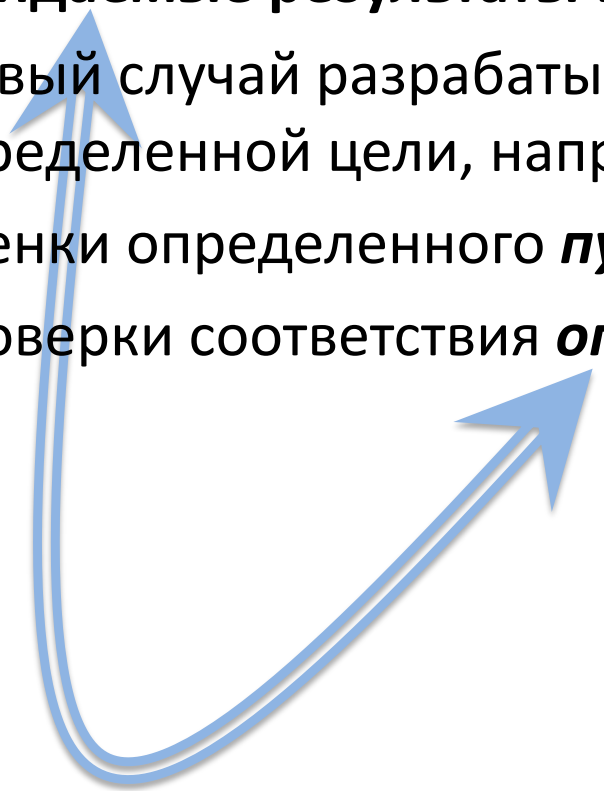
Определения – 1/5

- **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
  - Является подмножеством тестирования

Определения – 2/5

- Тестовый случай (тестовая процедура) включает:
  - набор входных данных
  - условия выполнения системы
  - процедуру (скрипт) выполнения системы
  - **ожидаемые результаты** выполнения

  Тестовый случай разрабатывается для достижения определенной цели, например:
  - оценки определенного ***пути выполнения*** приложения
  - проверки соответствия ***определенному требованию***



Определения – 3/5

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

Определения – 4/5

- 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 – 5/5

- Тестирование – процесс выполнения системы, демонстрирующий соответствие требованиям
  - 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) и локализовать причины этих ошибок или описать возможные пути избежания их проявления (work-around)
  - См. Верификация

Виды тестирования – 1/4

- По объекту тестирования:
  - Функциональное
    - Наличие функций, правильность реализации
  - «Примыкающие к тестированию функций»:
    - Тестирование масштабируемости
    - Тестирование производительности
      - Нагрузочное
      - Стресс-тестирование
      - Тестирование стабильности
      - Конфигурационное тестирование
  - Нефункциональное 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 (варианты):
    - 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 тогда, когда компоненты объединяются интерфейсами и получившаяся [под]система тестируется как единое целое
  - Выявляются ошибки взаимодействия компонентов и проблемы определения интерфейсов
- Системное тестирование – «высшая стадия» интеграционного тестирования: все компоненты объединены в конечный вариант РИС
  - NB! Иногда использование РИС предполагает комбинирование компонентов в разных вариантах
  - Отличие от интеграционного – тестируется внешний интерфейс системы, а не подробности внутреннего взаимодействия компонентов

По изолированности – 3/3

- Приемочное (приемо-сдаточное) 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
  - Практический совет: Иметь два набора исходных данных тестирования от заказчика:
    - Для отладки
    - Для приемного тестирования

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

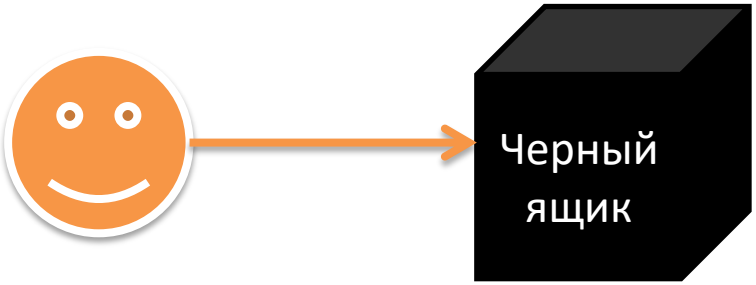
- В тестировании производительности различают следующие направления:
  - Нагрузочное (load)
    - Определяет способность обработать заданное количество запросов и/или объем данных за заданное время
  - Стресс (stress)
    - Определяет пределы устойчивости системы и ее поведение при нагрузках выше заданных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

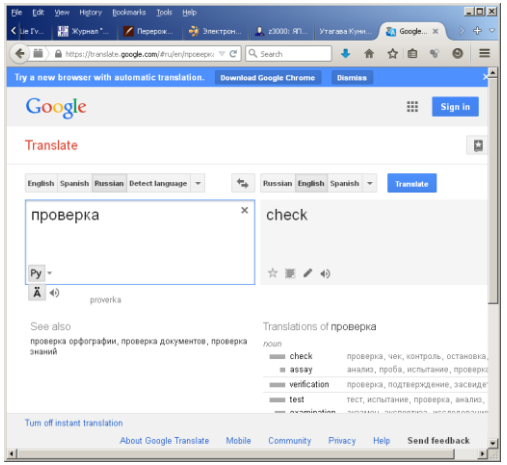
По знанию системы

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)



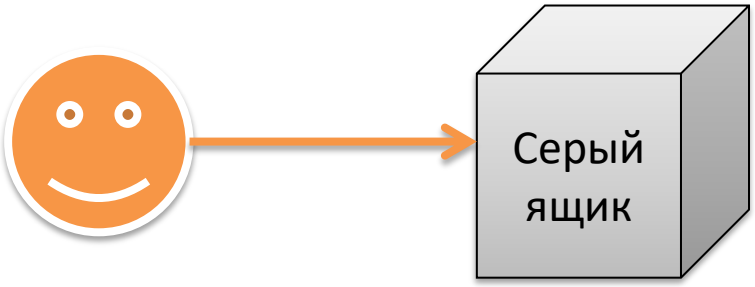


Виды: черный ящик – 2/2

- 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)

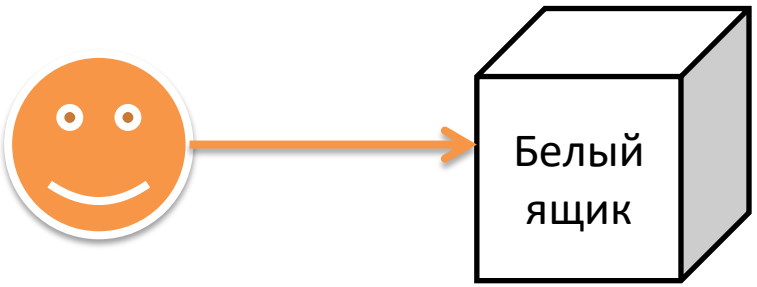


Виды: серый ящик – 2/2

- 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

Обычно выполняется с помощью статических анализаторов кода

- 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 по критерию исходного кода программы

Сравнение «ящиков»

Тестовый случай определяется	Результат
Требованиями	Выходные данные сравниваются с заданными требованиями «Черный» ящик
Требованиями и элементами дизайна	Как для «серого» и «белого» ящиков
Элементами проектирования	Подтверждение ожидаемого поведения «Белый ящик»

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
    - Сами скрипты требуют тестирования и отладки

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 при сохранении объема тестирования
- Скорость проведения тестов
- Возможность выполнения тестов в нерабочее время (скорость + сокращение использования ресурсов)
- 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
- 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 (non-regression test) – 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

Масштабируемость

Производительность

Инструменты

ТЕСТИРОВАНИЕ 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

Что масштабируется

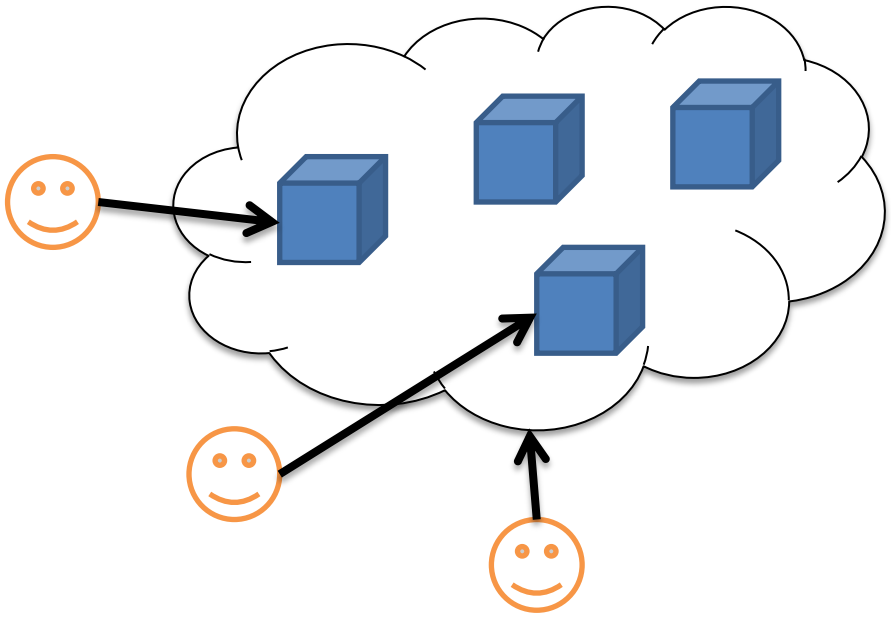
Примеры

ТЕСТИРОВАНИЕ МАСШТАБИРУЕМОСТИ

Тестирование масштабируемости

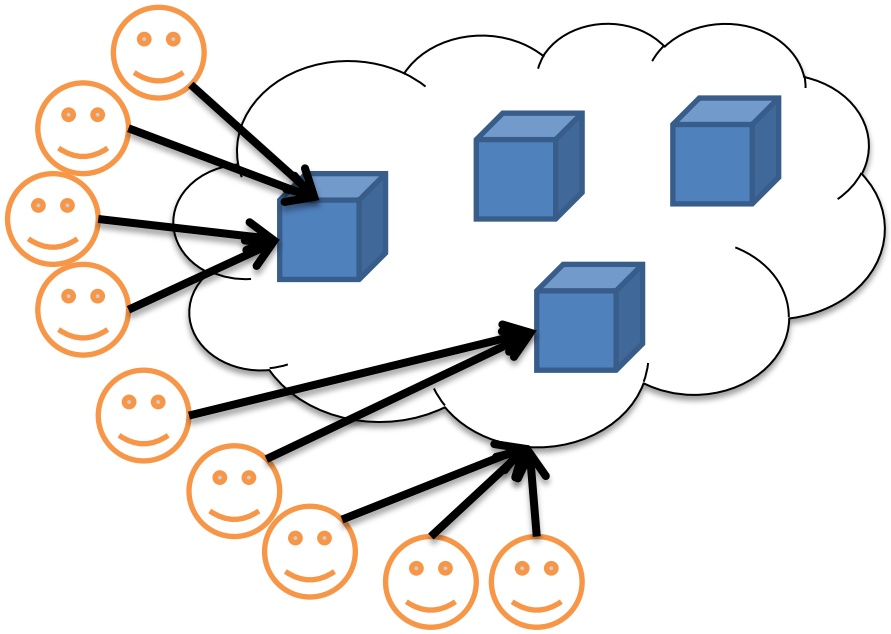
- Мера оценки производительности РИС не всегда четко определена
- Все особенности тестирования РИС проявляются в крайней степени
- Включает в себя:
  - Тестирование производительности
  - Устойчивость структуры РИС к увеличению числа элементов
  - Поведение системы при превышении предельной нагрузки

Что масштабируется – 1/2



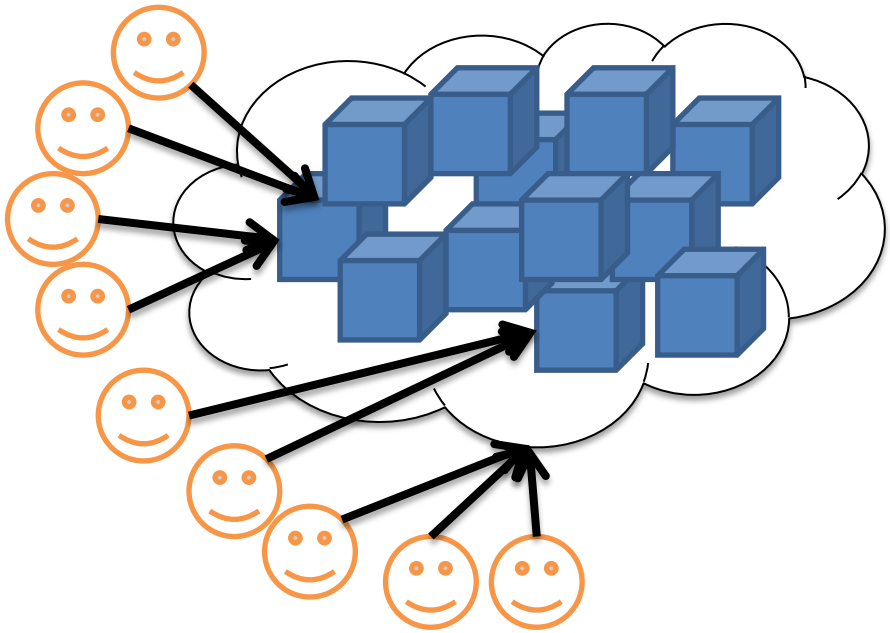
Исходный вариант

Что масштабируется – 2/2



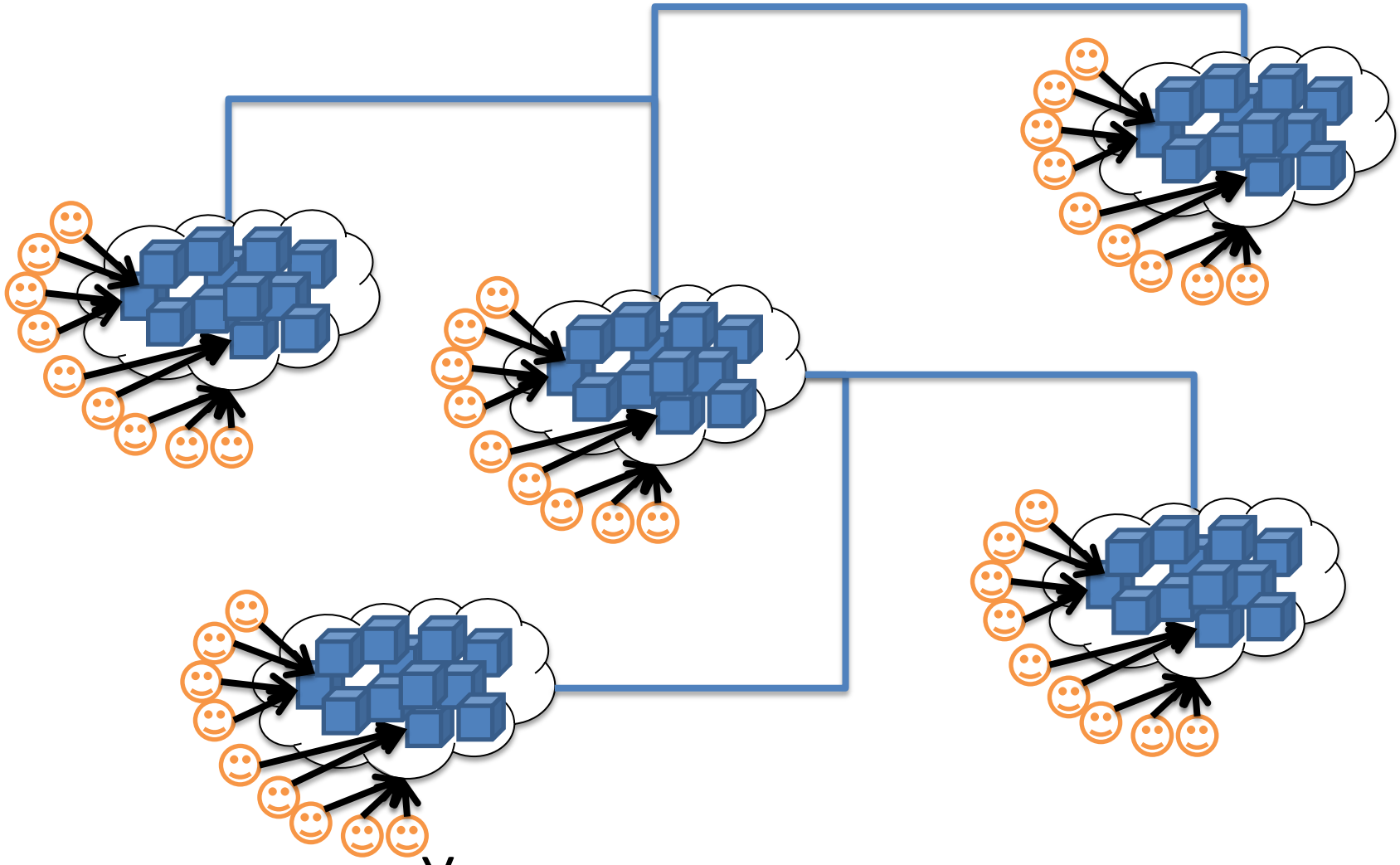
Увеличивается нагрузка

Примеры масштабируемости – 1/3



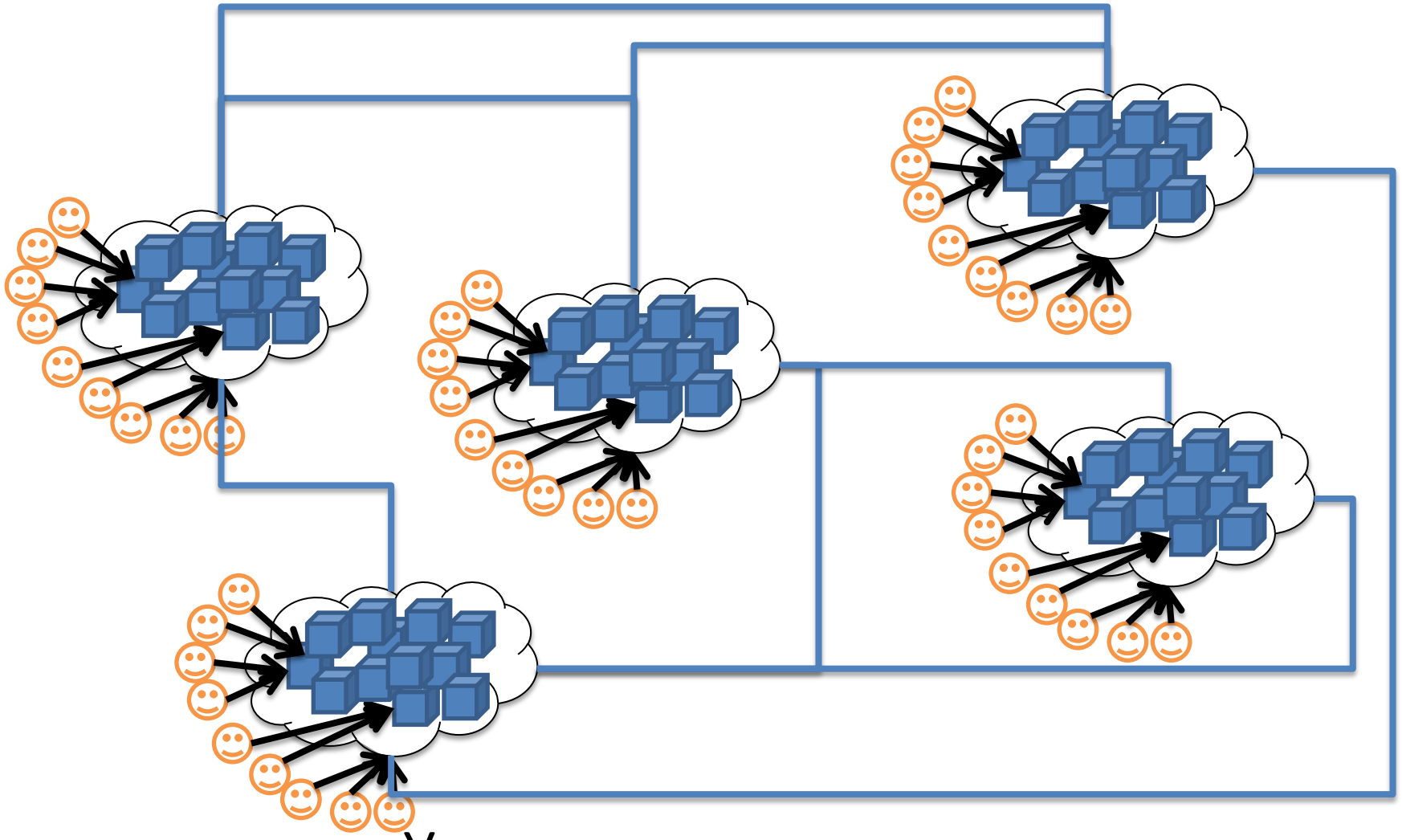
Увеличивается число элементов

Примеры масштабируемости – 2/3



Усложняется структура

Примеры масштабируемости – 3/3



Усложнение сети связи

Определения

Виды

ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ

Тестирование производительности

- 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

Виды тестов производительности

- 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 /soak / stability)
  - Проверяет способность выполнять функции при длительной работе
- Конфигурационное (configuration)
  - Определяет влияние различных конфигураций системы на ее производительность

Определения

Основная идея

Приемы

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)**

Определения

История

XP PLANNING GAME (ИГРА В ПЛАНИРОВАНИЕ)

Planning game: Agile – 1/8

- Игра в планирование XP является специальной формой встреч и взаимодействия заказчиков и разработчиков, предназначенной для:
  - Улучшения архитектуры проекта
  - Упрощения организации процессов разработки и управления проектом
- Игра в планирование разнится на стадиях:
  - Планирования выпусков (release planning)
  - Планирования шагов (iteration planning)

Синяя таблетка позволит остаться в искусственно созданной реальности Матрицы

Planning game: Release planning – 2/8

- Планирование выпусков – первая часть игры в планирование. Предполагает совместную работу заказчика и разработчиков. Включает работу с проектными требованиями и планирование работ в целом. Разделяется на три фазы:
  - Исследовательскую (exploration phase)
  - Достижения договоренностей (commitment phase)
  - Управления (steering phase)

Planning game: Release planning – 3/8

- **Исследовательская фаза** предполагает описание заказчиками своего видения будущей системы, включая характеристики и поведение. Это описание заносится на карточки, разделяясь на задачи, которые надо будет реализовывать. Эти user story cards дальше используются как «направляющие» дальнейшую разработку

Planning game: Release planning – 4/8

- Во время фазы достижения договоренностей (commitment phase) заказчик и разработчики согласуют конкретную функциональность, которая должна быть реализована. Кроме этого принимаются решения о датах выпуска ПО

Planning game: Release planning – 5/8

- Фаза управления (steering phase) включает пересмотр текущих планов разработки, а также требований к ПО. Заказчик и разработчики согласуют наилучшие способы достижения результата (NB! определяемого заказчиком), что как раз и может привести к изменению требований и планирования разработки

Planning game: Iteration planning – 6/8

- Планирование шагов (iteration planning) является второй частью игры в планирование. Участие в этой части принимают только разработчики, участие заказчика не предполагается.
  Планирование шагов также разбивается на три фазы:
  - Исследовательскую (exploration phase)
  - Достижения договоренностей (commitment phase)
  - Управления (steering phase)

Planning game: Iteration planning – 7/8

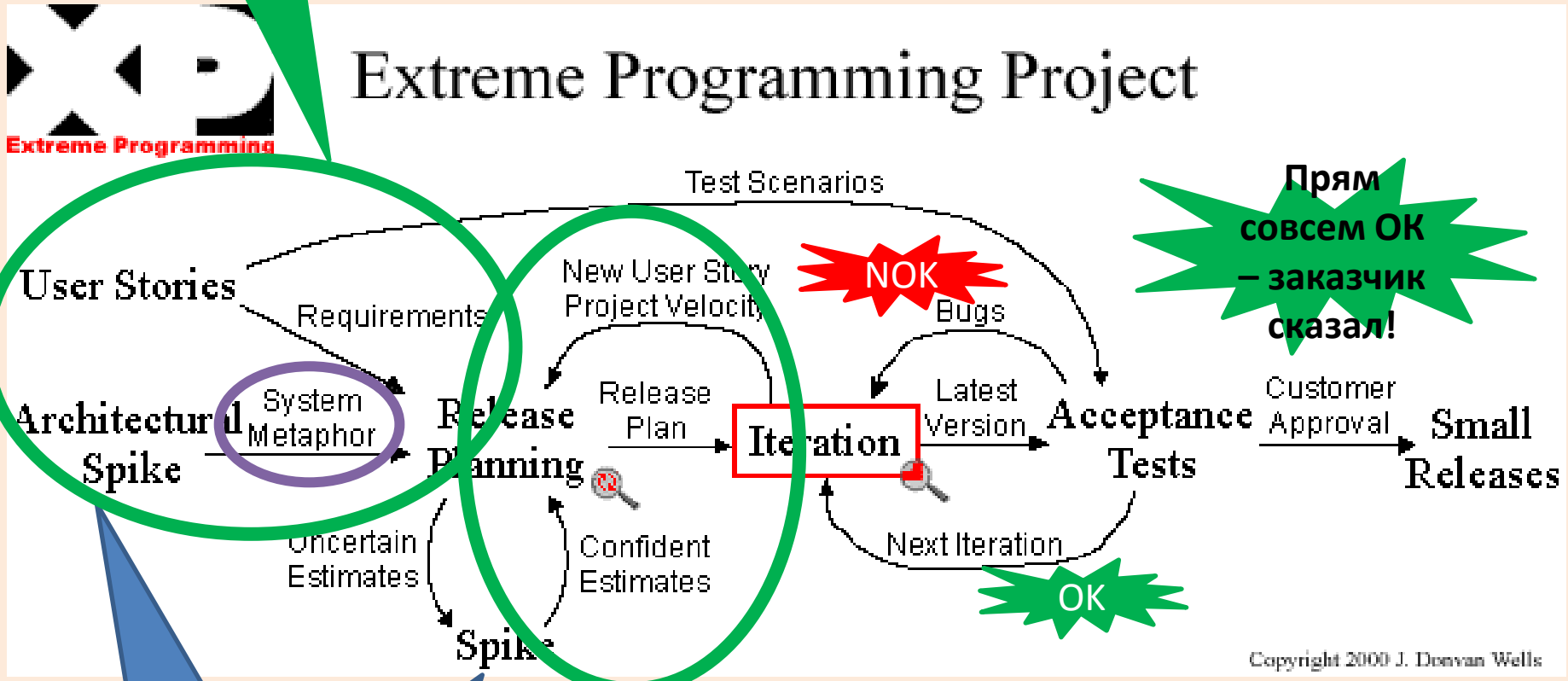
- Во время фазы исследования разработчики используют с пользовательскими требованиями, описанными user story card и «переводят» их конкретные задачи для решения. Полученные задачи оформляются в «карточек задач» (task card)
- На фазе достижения договоренностей задачи распределяются между разработчиками. Для задач, кроме назначения исполнителей, определяются сроки исполнения для решения задач учета (ensure accountability) и определения эффективности (efficiency)

Planning game: Iteration planning – 8/8

- Фаза управления включает учет завершенных задач и оценку результата сравнивая результат с задачей, определяемой user story card. В случае, когда результат отличается, проводятся доработки

Глубже в кроличью нору!

Фаза изучения



«Клинки» (в спину)

Фаза управления

Planning game: Реальность – 1/4

- Опуская идеологические рассуждения, можно сказать, что коллективное планирование может реализовываться на любом этапе проекта. Упрощая, должны быть выполнены следующие задачи:
  - Уточнение понимания планируемой к реализации (постановка задачи)
    - При этом обязательно привлечение специалистов в предметной области (заказчиков, экспертов, etc.)
    - Формирование описание тестовых процедур, которые докажут выполнение описанной задачи

Красная таблетка приведёт к бегству из Матрицы в реальный мир

Planning game: Реальность – 2/4

- Следующий шаг: LLD проектирование реализации описанной на предыдущем шаге функции
  - Элементы реализации общей бизнес логики разбиваются на элементы, реализующиеся в разных частях РИС (РИС – большие системы, за разные части которой отвечают разные ***коллективы***)
    - Например, необходимо реализовать расчетную задачу. Даже если сами вычисления производятся в одном компоненте РИС, задействуются и другие: предоставляющие исходные данные, получающие результаты, управляющие совместной работой
  - ~~Торг~~ согласование разбиения реализации на элементы, которые должны быть выполнены отдельными исполнителями, и объема требующейся работы (изменений и/или дополнений) влияет на разрабатываемую архитектуру, ~~потому что люди меркантильные, ленивые~~

Planning game: Реальность – 3/4

- Следующий шаг: планирование работ
  - Зависит от созданной ранее архитектуры:
    - Кто что конкретно делает (разбиение на реализуемые части и соотнесение их с конкретными разработчиками)
    - Длительности работ (зависят от объема задачи, других проектов исполнителей)
    - Последовательности реализации и зависимости между компонентами
    - Для каждой задачи определяется наглядный способ проверки ее выполнения
- Отслеживание выполнения проводится «обычным» порядком. Встречи-обсуждения проводятся тогда, когда необходимо согласовать изменения в планировании (не успели) и/или архитектуре (возникли проблемы в реализации)

Planning game: Реальность – 4/4

- Что необходимо внедрить (если нет):
  - Присутствие ведущего собрание, который готовит протокол совещания (minutes of meeting)
  - В протоколе отражаются:
    - Задачи (actions), для которых определяются: исполнитель, срок исполнения, описание задачи (отдельно ее результат, если необходимо)
    - Проектные решения. Хорошим способом их фиксации является копия схемы с белой доски (whiteboad). Белой доской может быть физическая доска (тогда подойдет фото) или результат работы специализированной программы (например MS Whiteboard). Кроме собственно архитектуры на схеме надо пометить ответственных, объемы работы, etc.

Определения

История

Процедуры

Методы

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ (TEST-DRIVEN DEVELOPMENT)

История TDD

- Появилась 1999 году как концепция «сначала тест» (test-first) из экстремального программирования
- Позже выделилась как независимая методология
- Сейчас (2020+) несколько смешалась с «простым» использованием Unit-тестов

TDD: Определения – 1/3

- 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
  - Далее (по мере необходимости) проводится рефакторинг кода

TDD: Определения – 2/3

- Другое название цикла TDD: «красный / зеленый / рефакторинг»:
  - Красный: Сначала новый тест НЕ проходит (теста нет → тест есть; кода еще нет)
  - Зеленый: Затем пишется код, который его проходит (тест есть; кода нет → код есть)
  - Рефакторинг: Последующие изменения кода (тест есть; код есть → измененный [лучший] код)

TDD: Определения – 3/3

- 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

Крайне маловероятно. Повторяющееся выполнение вручную сотен даже мелких тестов нереализуемо

TDD: Процедура – 1/2

- Задача: Реализовать некий алгоритм (создание)
- Выполнение этап 1:
  1. Продумать и реализовать тест, проверяющий правильность работы (например, через результат) алгоритма
     - Кода еще нет!
  2. Сразу после этого реализовать алгоритм
     - Непосредственно после – еще хорошо помнится, каков должен быть результат/поведение

Пример теста: getHost() – 1/2

- Функция ищет в списке хост по его имени номеру порта
- Что проверяем:
  - Что нашли в списке хост, который там есть (позитивный тест)
  - Что не нашли в списке хост, которого там нет (негативный тест)
- В ходе обдумывания тестов принимается решение о том, как сообщить о провале поиска, например:
  - Вернуть «пустой» указатель, если не нашли
  - Вернуть результат поиска (true/false), а хост вернуть, изменив значения параметра вызова функции
  - «Бросить» исключение

Очевидно!

Чуть менее очевидно…

Предпочтительные варианты могут быть описаны в стандарте программирования

Пример теста: getHost() – 2/2

- Неочевидные проблемы тестирования:
  - А список откуда взялся? И откуда мы знаем, какой хост там есть, а какого нет?
    - Решаем, например, так: заполняем список перед вызовом метода
  - А что является элементом списка хостов? Экземпляр класса, структуры, что-то еще? Кто и как эти элементы создает?
    - Решаем, например, так: создаем элементы
  - Итого:
    - При анализе и написании теста мы неминуемо приняли некоторые архитектурные решения
      → +3 к проектированию (LLD)
    - И подумали, прежде чем сделать!
      → +5 к качеству кода

Пример теста: calcStrangeFnc() – 1/4

- Вычислить функцию `sin(x)*cons(x)/tg(x)`
- Что проверяем:
  - Вычисленное значение (тест обычного выполнения)
  - Отказ от вычисления при x = 0 + 2π (деление на ноль) и в x = π/2 (тангенс не определен)
- В ходе обдумывания тестов принимаются решения:
  - В каких единицах передавать угол (по умолчанию для библиотеки math – радианы)
  - Как обозначить π? Так, чтобы программа поняла, что это π, а не просто некое число, ему точно не равное

Очевидно!

Чуть менее очевидно…

Пример теста: calcStrangeFnc() – 2/4

- Принятые решения (продолжение):
  - Как отказаться вычислять, например:
    - Вернуть специальное значение NAN, как в библиотеке math
    - Вернуть успешность вычисления (true/false), а результат вернуть, изменив значения параметра вызова функции
    - «Бросить» исключение

Предпочтительные варианты могут быть описаны в стандарте программирования

- Неочевидные проблемы тестирования:
  - А в каких единицах в проекте задают углы? А каковы границы значений? Углы ли это?
    - А используются ли только «единичные» углы, или их «группы» для описания ориентации – <крен, рыскание, тангаж> или <крен, курс, тангаж> или кватернионы?
      - А какая последовательность поворотов?

Проектные решения верхнего уровня

Пример теста: calcStrangeFnc() – 3/4

- Неочевидные проблемы (продолжение):
  - А как у нас в проекте вообще задают π?
    - А как его отличить от «просто числа»?
      - А как задать точность вычислений с плавающей точкой?
  - Хм… А почему нам вообще не упростить выражение?

Проектные решения верхнего уровня

$$\frac{\sin(x) * \cos(x)}{tg(x)} = \frac{\sin(x) * cos(x)}{\frac{\sin(x)}{\cos(x)}} = \frac{\sin(x) * \cos(x) * cos(x)}{\sin(x)} = \cos^2(x)$$

…а ведь у квадрата косинуса нет ограничений на значения параметров…

Понимаем, что и зачем мы делаем…

- Какой физический / практический смысл именно в том виде формулы, который был задан, можно ли привести к квадрату косинуса?

Пример теста: calcStrangeFnc() – 4/4

- Итого при анализе и написании теста мы:
  - Неминуемо приняли некоторые архитектурные решения

    → +3 к проектированию (LLD)
  - Разобрались в принятых архитектурных решениях верхнего уровня

    → +5 к пониманию архитектуры проекта (HLD)
  - Разобрались в семантике (почему такая формула) и прагматике (как ее применять) предметной области (как минимум, конкретной задачи)

    → +5 к пониманию задачи
  - И подумали, прежде чем сделать!

    → +5 к качеству кода

TDD: Процедура – 2/2

- Задача: **Улучшить** алгоритм (рефакторинг)
- Выполнение этап 2:
  1. Сформулировать понятие «лучшести» кода, придумать способ проверки этого
  2. Изучить код и изменить его так, чтобы он стал лучше
     - Переписать код
     - Проверить код (старыми тестами – на правильность функционирования)
  3. Если тесты прошли – ОК, проверяем «лучшесть»
     - Если стало лучше, то фиксируем изменения (вместе с описанием теста «лучшести»)
     - Если тесты «лучшести» автоматизируются, то добавляем их в список автоматических, нет – оставляем как документ
  4. Если тест не прошли – не ОК, отлаживаемся или откатываемся

Рефакторинг – 1/2

- Рефакторинг – это улучшение кода. Под улучшением понимается изменение нефункциональных свойств. К таким свойствам относятся (как минимум):
  - Производительность
  - Масштабируемость
  - Удобство использования (юзабилити)
- В любом случае проверка улучшений выполняется на уровне интеграционном или системном и автоматизации поддается плохо

Рефакторинг – 2/2

- Не следует путать рефакторинг с:
  - исправлением ошибок – это изменение функциональности кода для приведения в соответствие с требованиями: тестирование (обычно пользователями – черный ящик) и отладка
  - добавлением функциональности – хотя оно и может привести к изменению существующего кода, но тесты приходится менять вместе с кодом, чтобы покрыть (отразить) новую функциональность

Unit-тестирование и TDD

- Количество единичных тестов велико, поэтому используются средствами [автоматизированного] тестирования (testing frameworks), с помощью которых создают и автоматизируют запуск наборов единичных тестов

  →Поэтому в повседневности часто смешивают TDD и использование unit-test средств. Отличия «обычного» unit-тестирования:

  - последовательность создания кода и тестов («обычно» сначала код)
  - покрытии кода тестами («обычно» тестируют только ключевой код)

Приемы TDD – 1/2

- 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

getHost(): KISS и рефакторинг – 1/3

- KISS: поиск ведем прямым перебором
- Что может быть улучшено:
  - Скорость поиска
  - Поиск как по IP адресу в строчном виде, так и по имени хоста (разрешаем имена)
- Как ускорить:
  - ~~Придумываем более быстрый алгоритм поиска~~ ~~Берем том Кнута~~ Выбираем более быстрый алгоритм
  - Меняем способ хранения списка хостов на подходящий для поиска. Например, вместо std::vector используем std::map

Рефакторинг как он есть

Изменение функциональности!
Нерелевантно

getHost(): KISS и рефакторинг – 2/3

- Примечания:
  - На эффективность алгоритма поиска может влиять упорядоченность хостов в списке. Для ее достижения может понадобиться изменить метод добавления хостов в список
  - Изменение алгоритма поиска на более быстрый может потребовать заодно и изменения формата хранения списка хостов. Например, гораздо эффективнее «переходить» по элементам массива, чем по элементам односвязного списка
    - Изменение формата хранения списка хостов может повлиять на «лучшесть» и даже на правильность выполнения других методов класса

getHost(): KISS и рефакторинг – 3/3

- Примечания:
  - Необходимость ускорения поиска может «всплыть» как минимум, в случаях, когда:
    - У нас очень много хостов и прямым перебором искать долго
      - Эффективнее использовать более быстрый алгоритм поиска
    - Поиск вызывается очень часто
      - Может оказаться эффективным введение «кэширования запросов», если один и тот же хост ищется много раз
      - Может оказаться более логичным и эффективным изменение использующего кода, когда вместо постоянного вызова метода getHost() с одними и теми же параметрами лучше сохранить результат поиска и использовать его

calcStrangeFnc(): KISS и рефакторинг

- KISS: вычисляем функцию «как есть», без попыток ее упростить математически
- Что может быть улучшено:
  - Упростить выражение – ускорит вычисление
  - Определить точность вычислений и зафиксировать в коде
  - Добавить в параметры требующуюся точность вычислений
- Проверки улучшений:
  - Скорость – очевидно, прогнать оба варианта множество раз
  - Для определения точности написать проверку количества знаков после запятой, проверить на [всем] множестве значений

Рефакторинг как он есть

Тоже рефакторинг, но посложнее

Изменение функциональности и интерфейса! Нерелевантно

Приемы TDD – 2/2

- 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 покрытию тестами всей функциональности. Иначе разработчики склонны переходить к реализации других функций, не протестировав только что реализованную

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

Общее описание

Мониторинг и диагностика РКС (не РИС!)

ИНСТРУМЕНТЫ ТЕСТИРОВАНИЯ

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

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

Утилиты

SNMP

Системы мониторинга

МОНИТОРИНГ И ДИАГНОСТИКА РКС

Мониторинг и диагностика РКС

- Мониторинг РКС – [постоянное] наблюдение за элементами компьютерной сети с целью обнаружения:
  - Отказов элементов (хостов или сетевого оборудования)
  - Сбоев в выполнении операций различного рода
  - Деградации характеристик элементов или сети в целом
- Задачи мониторинга является подмножеством задач управления РКС
- Диагностика РКС – ~~процесс установления диагноза~~ методы и средства определения технического состояния элементов РКС. Обычно служит для выявления причин сбоев выявленных в процессе мониторинга

Что мониторится – 1/2

- Сеть:
  - Конфигурация
  - Доступность (других сетей, хостов, сетевого оборудования)
  - Пропускная способность
- Сетевое оборудование
  - Доступность
  - Состояние

Что мониторится – 2/2

- Хосты:
  - Доступность
  - Загрузка:
    - ЦП (средняя, по ядрам, etc.)
    - Памяти (оперативной, подкачки)
    - Свободное место на смонтированных томах
- Общее ПО (ОС, СУБД, etc.):
  - Состояние
  - Ошибки / отказы
- Специальное ПО: ?

Способы мониторинга РКС

- Системные утилиты:
  - `ping`
  - Диспетчер задач / `top` / `ntop` / `htop` / `df` / `uptime` / `uname -a` / etc.
- Сервера SNMP
- Мониторинг журналов (системных и специальных)
- Системы мониторинга РКС

Способы диагностики РКС

- Системные утилиты:
  - `tracert` / `traceroute`
  - `ipconfig` / `ifconfig`
  - `ping`
- Средства удаленного управления:
  - RDP (Remote Desktop Protocol)
  - VNC (Virtual Network Computing)
  - Удаленный терминал / telnet
- Сервера SNMP
- Анализ журналов (системных и специальных)

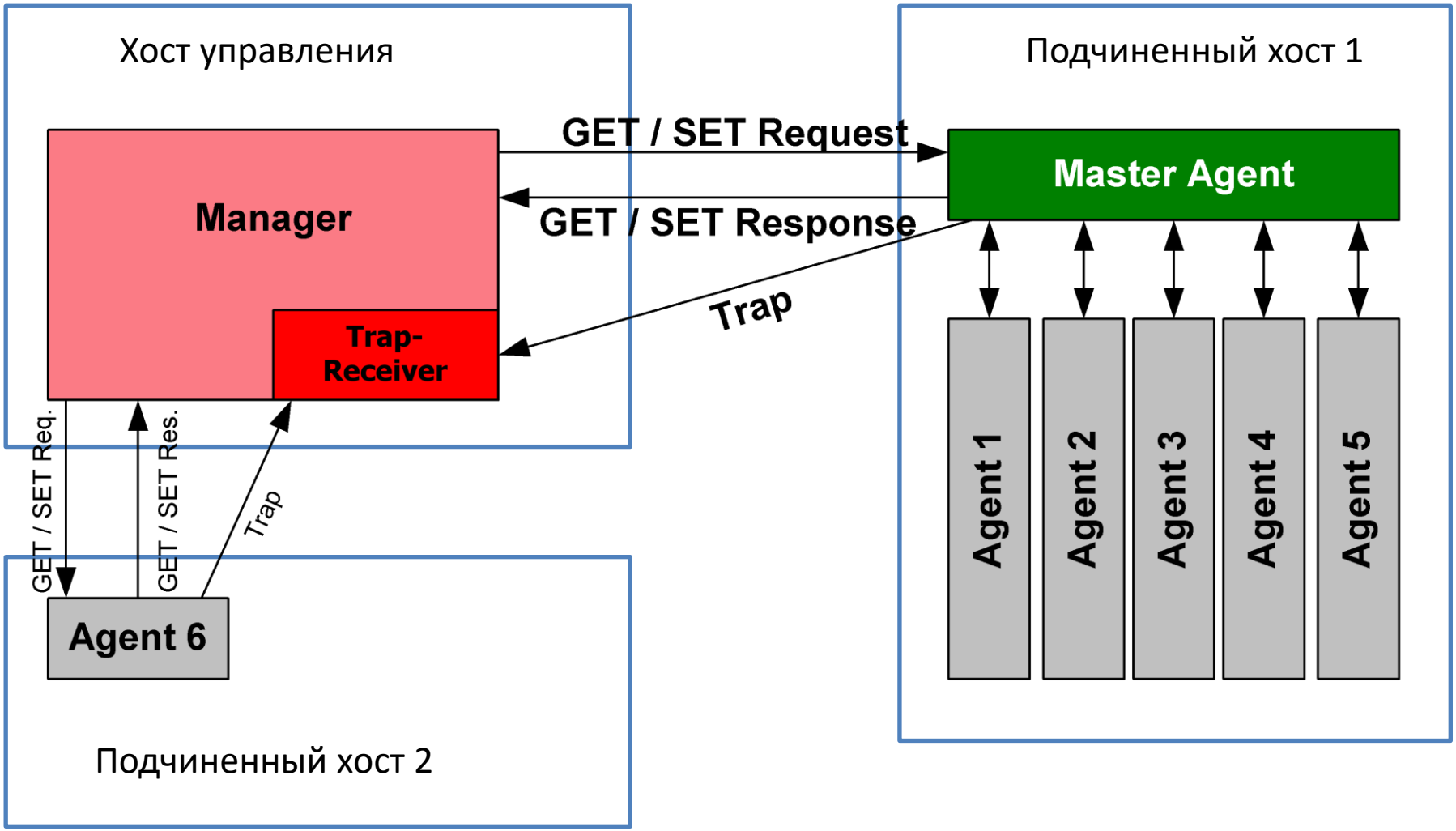
SNMP – 1/6

- SNMP (Simple Network Management Protocol) – стандартный интернет-протокол для управления устройствами в IP-сетях на основе архитектур TCP/UDP. Включает:
  - Протокол прикладного уровня
  - Схему БД
  - Набор объектов данных
- SNMP поддерживают:
  - Маршрутизаторы / коммутаторы
  - Серверы
  - Рабочие станции
  - Принтеры
  - Сетевые ОС (sic!), etc.

SNMP – 2/6

- Под управлением (management) понимается:
  - Сбор информации о состоянии и параметрах настроек управляемых устройств (мониторинг)
  - Изменение параметров настроек для изменения поведения устройств (собственно управление)
- Управляющий компьютер (возможно, не один) – компьютер, где функционирует управляющее СПО – Менеджер
- На каждой управляемой системе есть одна или несколько постоянно запущенных программ – Агентов, которые через SNMP обмениваются данными с Менеджером

SNMP – 3/6


Хост управления
Manager
Trap-
Receiver
GET / SET Request
GET / SET Response
Trap
GET / SET Req.
GET / SET Res.
Trap
Agent 6
Подчиненный хост 2
Подчиненный хост 1
Master Agent
Agent 1
Agent 2
Agent 3
Agent 4
Agent 5

SNMP – 4/6

- Master Agent может скрывать от Менеджера несколько Агентов на одном хосте
- Обмен данными Менеджер – Агент (Master Agent)
  - Get Request / Get Response – запрос текущих значений переменных и получение их
  - Set Request / Set Response – запрос на изменение значений переменных и подтверждение выполнения
  - Trap – передача данных от Агента Менеджеру по инициативе Агента

SNMP – 5/6

- Адреса объектов устройств задаются в цифровом формате и не являются человеко-читаемыми. Для упрощения их использования (интерпретации человеком) применяются базы управляющей информации (MIB – Management Information Base)
- База MIB описывает структуру управляемых данных устройства
  - Имена организованы иерархически
  - Каждый элемент дерева содержит идентификатор объекта (OID). OID состоит из двух частей
    - Читаемого текстового имени
    - SNMP адреса объекта в цифровом (нечитаемом) виде

SNMP – 6/6

- MIB необязательны и используются лишь для перевода имен объектов из читаемого формата в цифровой (ср. DNS)
- Структура объектов на устройствах разных производителей не совпадает (sic!) → без MIB практически невозможно определить цифровые SNMP адреса нужных объектов

Системы мониторинга РКС

- IBM Tivoli
- Zabbix
- СиПод

IBM Tivoli – 1/4

- IBM Tivoli — набор кроссплатфоменных решений для управления информационными ресурсами
- Основные функции:
  - Cloud Management – управление системой облачных вычислений и/или хранения
  - Virtualization Management – управление виртуальными серверами и предоставляемыми ими сервисами

IBM Tivoli – 2/4

- Основные функции (продолжение):
  - Storage Management – управление системами хранения предприятия
    - Online storage
    - Backup
    - Archiving
    - Disaster recovery
  - IT Service Management (ITSM) – набор операций (зависящих от «политик») организованный в процессы и поддерживающие процедуры для обеспечения заданного качества услуг Application Performance Management

IBM Tivoli – 3/4

- Основные функции (продолжение):
  - Application Performance Management – поддержание производительности СПО
  - Network Management – управление компьютерной сетью
  - System and Workload Automation – балансировка нагрузки [на сервера] (планировка задач)
  - Server, Desktop, Mobile Device Management & Security – управление устройствами и системами защиты от НСД

IBM Tivoli – 4/4

- Основные функции (продолжение):
  - Enterprise Asset Management – планирование, управление и мониторинг состояния технических и программных средств как имущества
  - Facilities Management – управление оборудованием и ПО с точки зрения физического размещения, организации работ, etc.

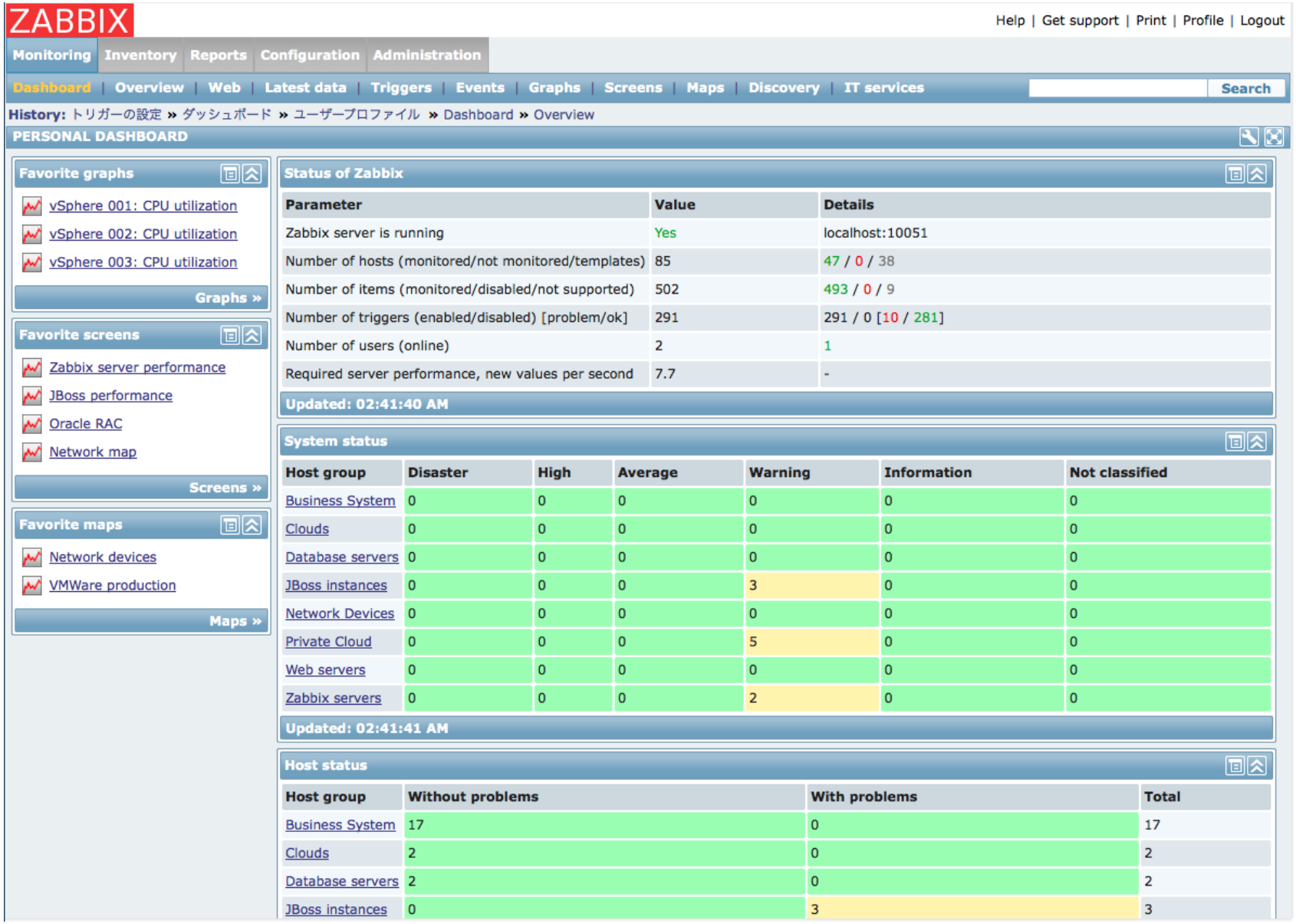
Zabbix – 1/3

- ZABBIX – свободная система мониторинга и отслеживания статусов разнообразных сервисов компьютерной сети, серверов и сетевого оборудования, написанная Алексеем Владышевым
- Особенности:
  - Данные хранятся в СУБД
  - Есть веб-интерфейс на PHP

Zabbix – 2/3

- Способы мониторинга
  - Simple checks – доступность (наличие реакции) стандартных сервисов проверяется выполнением запросов к ним без установки СПО на наблюдаемом хосте
    - SMTP, HTTP
  - ZABBIX agent – устанавливается на UNIX- и Windows-хостах
    - Загрузка ЦП, использование сети, дисковое пространство
  - External check — выполнение внешних программ
  - Мониторинг через SNMP

Zabbix – 3/3


ZABBIX
Help | Get support | Print | Profile | Logout
Monitoring Inventory Reports Configuration Administration
Dashboard | Overview | Web | Latest data | Triggers | Events | Graphs | Screens | Maps | Discovery | IT services
Search
History: トリガーの設定 » ダッシュボード » ユーザープロファイル » Dashboard » Overview
PERSONAL DASHBOARD
Favorite graphs
vSphere 001: CPU utilization
vSphere 002: CPU utilization
vSphere 003: CPU utilization
Graphs »
Favorite screens
Zabbix server performance
JBoss performance
Oracle RAC
Network map
Screens »
Favorite maps
Network devices
VMWare production
Maps »
Status of Zabbix
Parameter Value Details
Zabbix server is running Yes localhost:10051
Number of hosts (monitored/not monitored/templates) 85 47 / 0 / 38
Number of items (monitored/disabled/not supported) 502 493 / 0 / 9
Number of triggers (enabled/disabled) [problem/ok] 291 291 / 0 [10 / 281]
Number of users (online) 2 1
Required server performance, new values per second 7.7 -
Updated: 02:41:40 AM
System status
Host group Disaster High Average Warning Information Not classified
Business System 0 0 0 0 0 0
Clouds 0 0 0 0 0 0
Database servers 0 0 0 0 0 0
JBoss instances 0 0 0 3 0 0
Network Devices 0 0 0 0 0 0
Private Cloud 0 0 0 5 0 0
Web servers 0 0 0 0 0 0
Zabbix servers 0 0 0 2 0 0
Updated: 02:41:41 AM
Host status
Host group Without problems With problems Total
Business System 17 0 17
Clouds 2 0 2
Database servers 2 0 2
JBoss instances 0 3 3