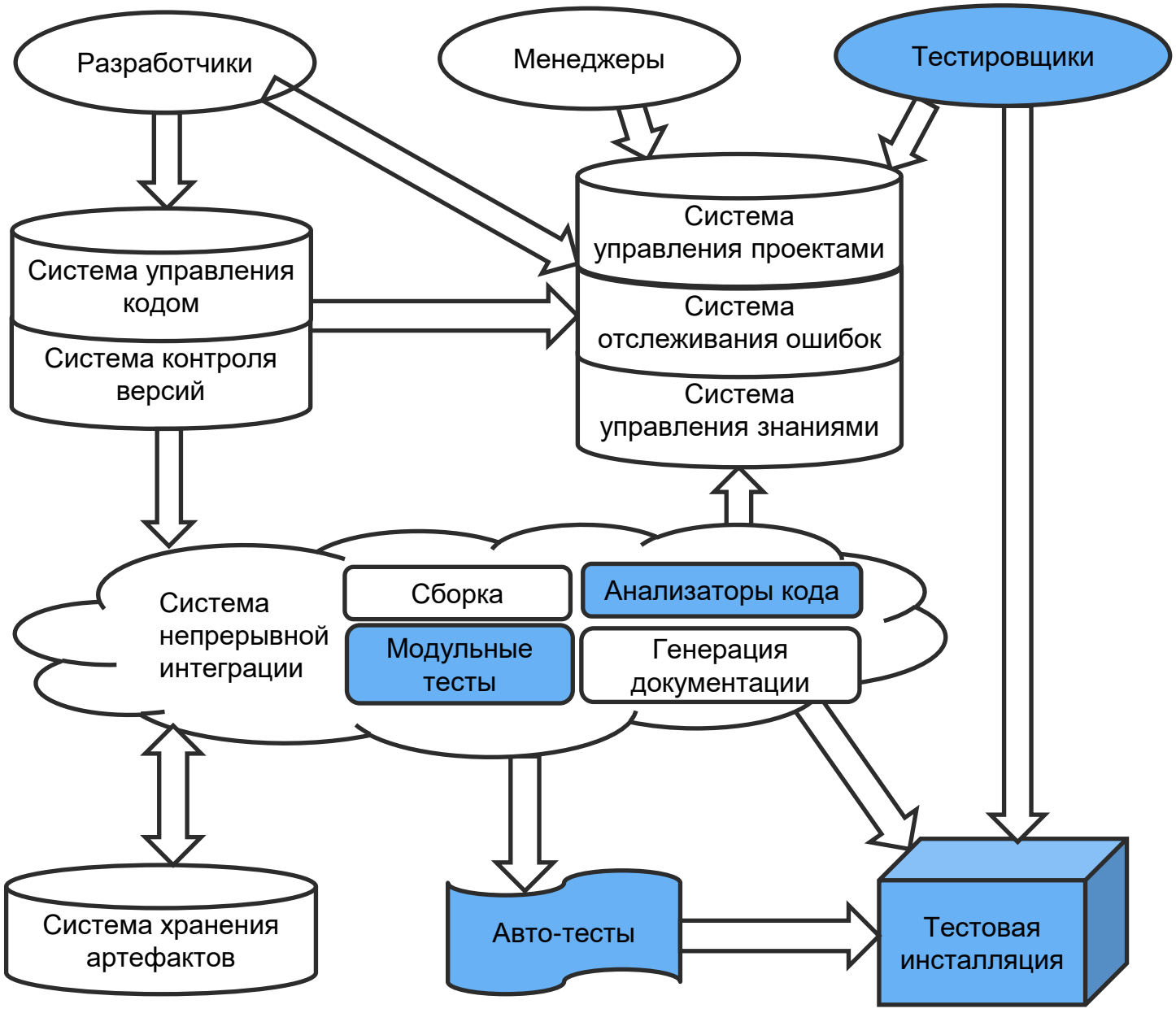
Управление производственным процессом разработки программного обеспечения

Quality Assurance


Разработчики
Менеджеры
Тестировщики
Система управления кодом
Система контроля версий
Система управления проектами
Система отслеживания ошибок
Система управления знаниями
Система непрерывной интеграции
Сборка
Анализаторы кода
Модульные тесты
Генерация документации
Система хранения артефактов
Авто-тесты
Тестовая инсталляция

План

- Основные определения
- Тестирование на пользователях
- Юнит тестирование
- Тестирование производительности

Обеспечение качества (Quality Assurance)

Quality Assurance

Планируемая систематическая активность, направленная на удовлетворение требований к качеству выпускаемого продукта или услуги.

QA требует:

- Признания своей роли в рабочем процессе
- Наличия требований к качеству

Ресурсов для обеспечения качества

Роль QA в SCM

Контроль

Для того чтобы управлять качеством необходимо знать текущее состояние продукта.

Оптимизация

Чем позже в жизненном цикле найдена ошибка, тем дороже ее исправление.

Обеспечение качества

"Планируемая систематическая активность, направленная на удовлетворение требований к качеству выпускаемого продукта или услуги." [ASQ Definition]

Тестирование

Тестирование

Тестирование это целенаправленное исследование продукта направленное на обнаружение и идентификацию ошибок.

Для тестирование необходимы:

- Объект тестирования
- Проверяемые требования
- Ресурсы

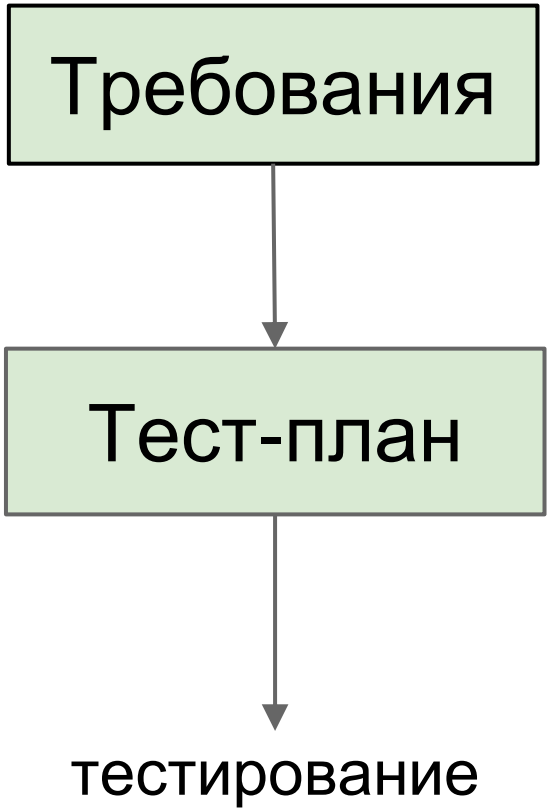
Тестирование — вероятностный процесс: невозможно найти все ошибки в программе.

Чаще всего ошибки сосредоточены на границах модулей.

Виды тестирования

- По степени подготовленности
- По объекту тестирования
- По знанию внутреннего устройства
- По отношению к исполняемому коду
- По времени выполнения
- По степени автоматизации

Тестирование по плану



план состоит из тестовых сценариев (test cases) — детализированных инструкций по тестированию

Тест-план обеспечивает повторяемость результатов между циклами тестирования. Хорошо составленный тест-план обеспечивает необходимое тестовое покрытие.

Метод свободного поиска

При отсутствии тест-плана и возможностей его составить тестировщики пользуются методом свободного поиска.

Функциональность делится на участки, которые проверяются один за другим на основании здравого смысла.

Достоинства:
Меньше бюрократии — больше дела.

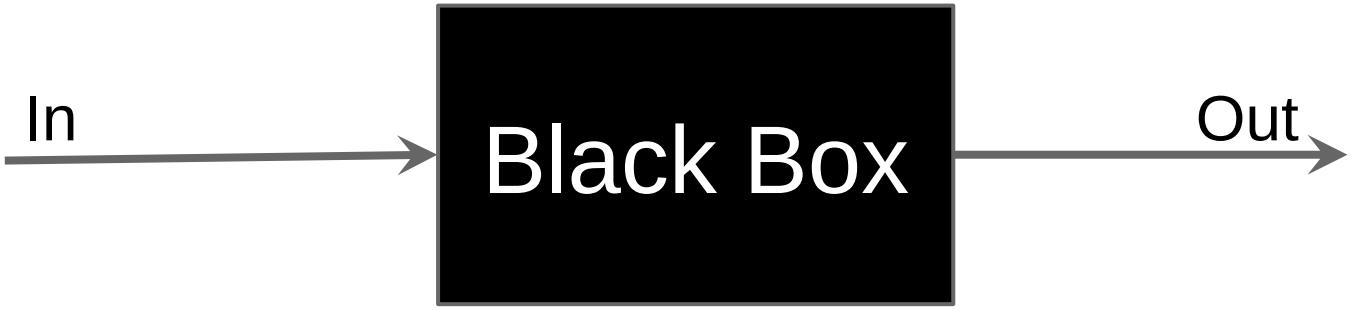
Недостатки:
Практически невозможно найти нетривиальные баги связанные с взаимодействием модулей

Объект тестирования

- Пользовательский интерфейс (UI testing)
- Отдельные модули (unit testing)
- Производительность (performance testing)
- Юзабилити (usability testing)
- Функциональность (functional testing)
- Безопасность (security testing)
- Локализация (localization testing)
- Совместимость (compatibility testing)
- Интеграционное тестирование (integration testing)

и т. д.

Тестирование черного ящика (black box)

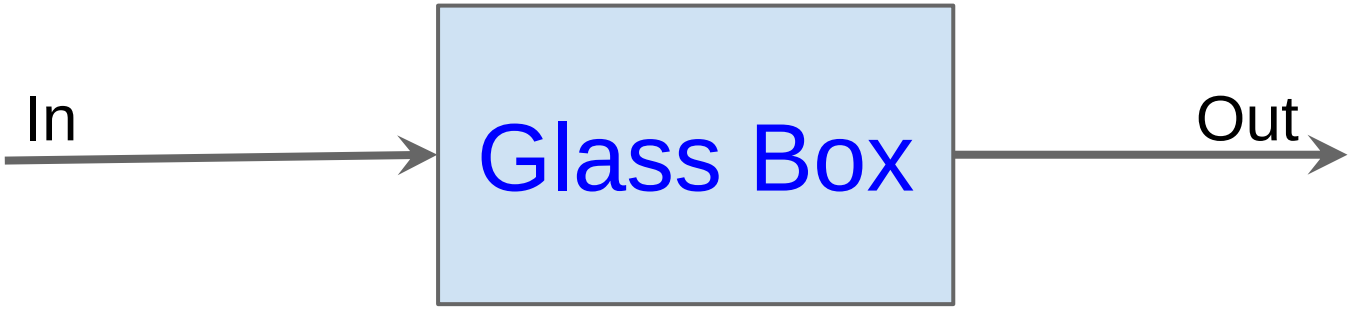


Тестирование программы как "черного ящика", без использования знаний об ее внутреннем устройстве.

Достоинства:
Позволяет легко находить несоответствия спецификации.

Недостатки:
Высока вероятность пропустить внутренние ошибки реализации.

Тестирование белого ящика (glass box)



Тестирование программы с использованием знаний об ее внутреннем устройстве.

Достоинства:
Позволяет легко находить внутренние ошибки реализации.

Недостатки:
Требуется время и квалификация для анализа исходного кода.

Знание внутреннего устройства

- Стратегия черного ящика

Функциональное и приемочное тестирование.

- Стратегия белого ящика

Тестирование безопасности, модульное тестирование.

- Стратегия серого ящика

Первые две стратегии являются идеализированными моделями.
На практике обычно применяется тестирование серого ящика.

Статическое тестирование

Статическое тестирование происходит без запуска кода.

- аудит кода (code review)
- анализ кода при помощи специальных инструментов

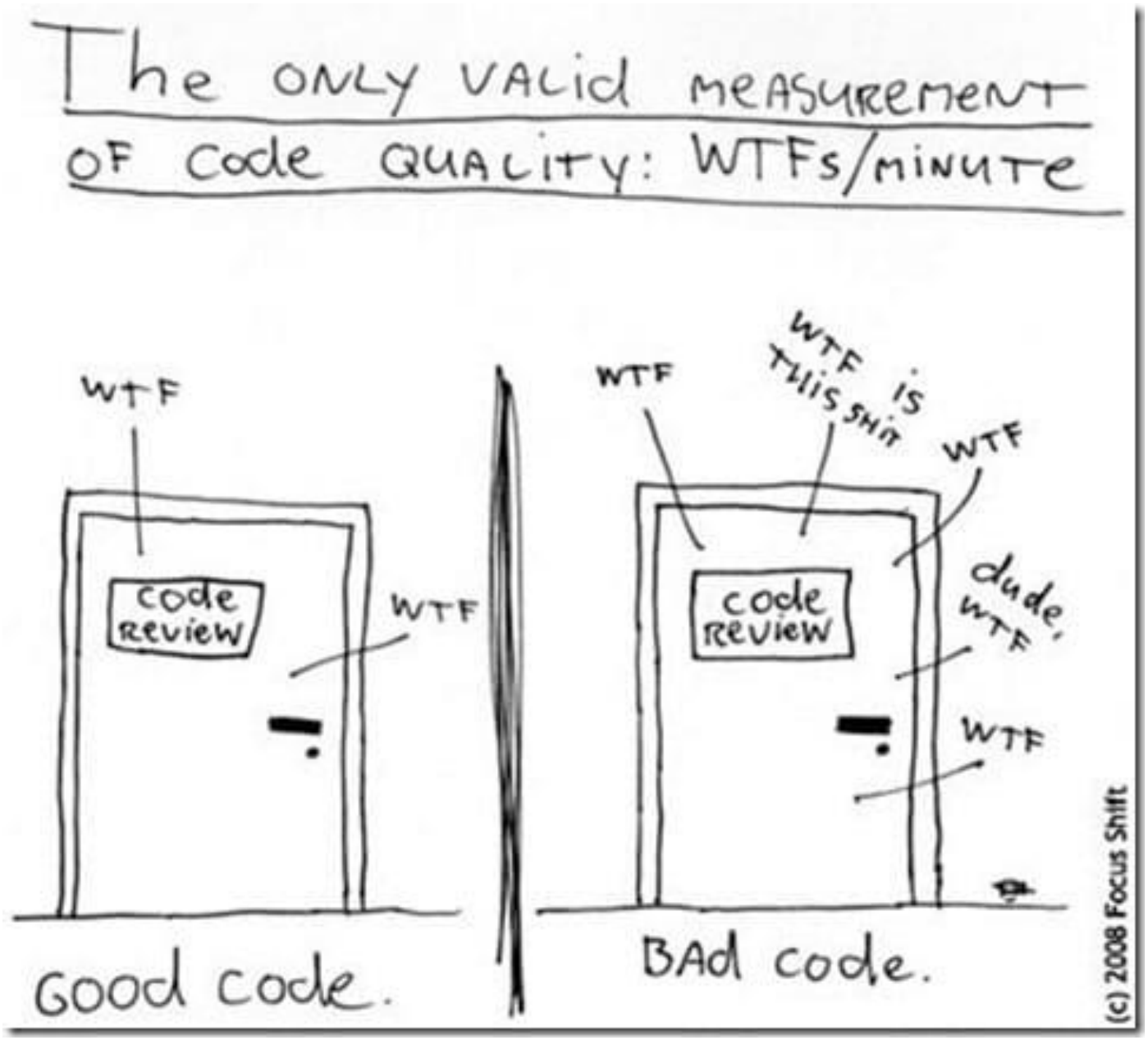
Существуют ошибки, которые легко обнаружить "методом пристального вглядывания". Классические примеры:

Java:

```
if (myString == "const") { ... }
```

C:

```
int* i;
i[10] = -1
```


The ONLY VALid MEASUREMENT OF Code QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF
THIS SHIT
IS
WTF
code review
dude.
WTF
WTF
BAd code.
(c) 2008 Focus Shift

Распространенные инструменты

C: lint, PV-Studio
Java: FindBugs, Sonar, PMD
JavaScript: JSlint, Closure compiler
Python: pylint
Perl: PerlTidy
.NET: FxCop, ReSharper

Практически все инструменты можно встроить в процесс сборки.

Недостатки инструментального статического тестирования

- Много ложных срабатываний (false positives)
- Многие правила плохо соотносятся с конкретным проектом
- Сложно составить набор правил обязательных к исполнению

Тестирование по времени выполнения

- Smoke testing
- Тестирование новой функциональности
- Регрессионное тестирование (regression testing)
- Приемочное тестирование (acceptance testing)
- Бета тестирование (beta testing)

Тестирование на реальных пользователях

У тестирования собственными силами есть несколько недостатков. Внутренне тестирование

- требует ресурсов
- замедляет поставку продукта пользователям
- не всегда соответствует ожиданиям пользователей
- не может справиться с достаточно большими продуктами.

Для борьбы с этими недостатками применяют бета-тестирование на реальных пользователях.

К моменту бета теста продукт уже должен соответствовать определенному уровню качества.

Недостатки бета-тестирования

- Бета-тестирование требует дополнительной инфраструктуры.
- Бета-тестирование не бесплатно. Затраты сокращаются ценой дополнительной поддержки.
- Злоупотребление бета-тестированием может отпугивать потенциальных пользователей.

Автоматическое тестирование

- Модульное тестирование (unit testing)
- Автоматические функциональные тесты
- Тестирование пользовательского интерфейса
- Тестирование производительности

Юнит-тесты

Юнит-тест это код, который тестирует код.

```
int sum(int a, int b) {
    return a + b;
}

void testSum() {
    assertEquals(4, sum(2, 2));
}
```

Как правило юнит тесты пишутся самими программистами по стратегии белого ящика.

Преимущества модульных тестов

- Способствуют раннему обнаружению проблем
- Очень легко встраиваются в системы сборки
- Сильно облегчают регрессионное тестирование
- Удешевляют внесение изменений
- Упрощают интеграцию
- Улучшают низкоуровневую архитектуру

Недостатки модульных тестов

- Требуют ресурсов на их написание (в среднем 3-5 строк тестового кода на строку рабочего кода)
- Требуют ресурсов на поддержку в актуальном состоянии
- Не обнаруживают проблемы интеграции
- Не обнаруживают ошибки в многопоточной среде
- Сами тесты могут содержать ошибки

Mock(dummy)-объекты

Очень часто невозможно написать "чистый" unit test из-за внешних зависимостей.

Mock-объект это объект моделирующий отельные аспекты поведения внешнего окружения (по отношению к тестируемому коду).

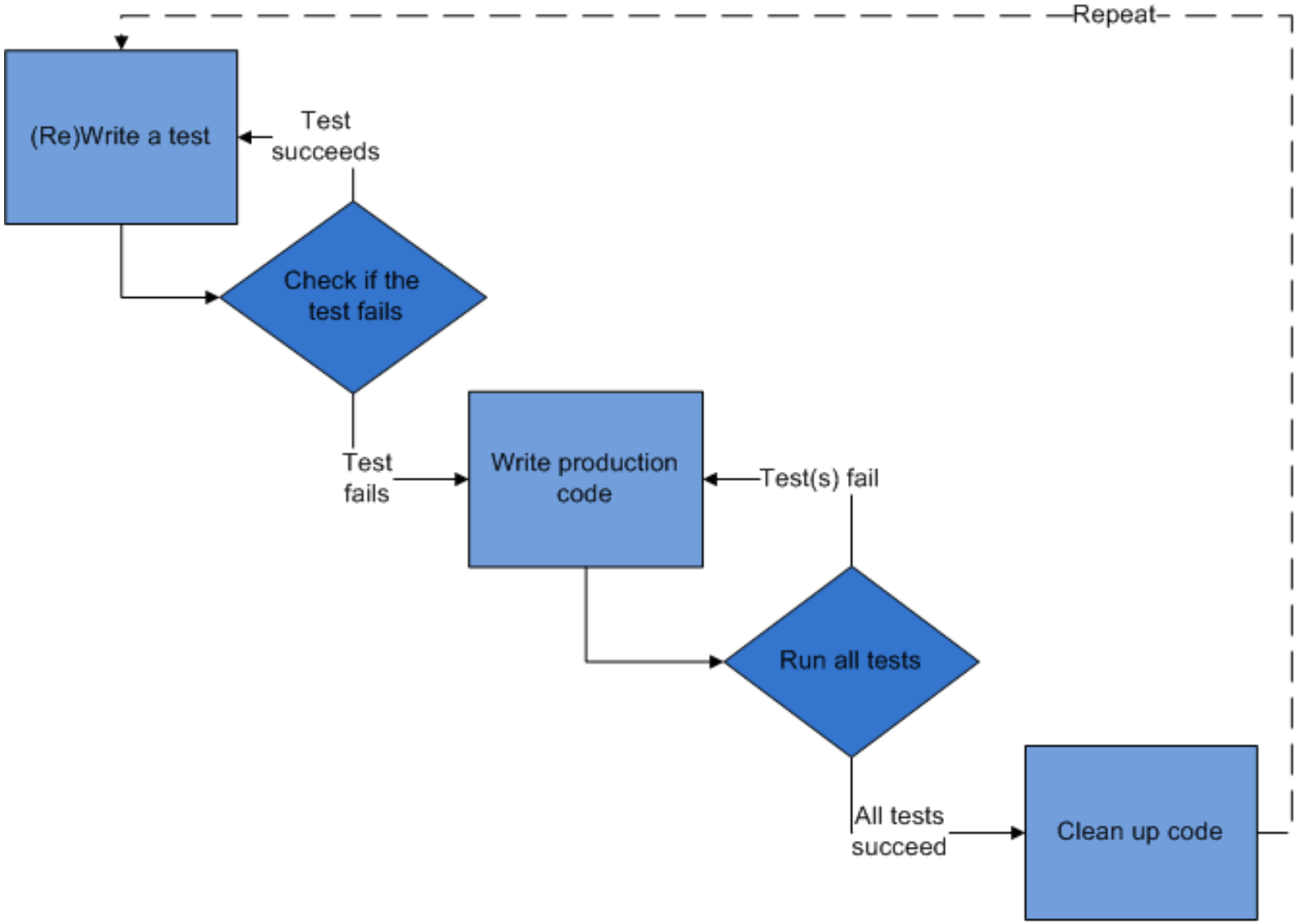
Mock-объект умеет выдавать заранее определенную реакцию на внешние вызовы или отслеживать порядок вызовов.

В Java для создания mock объектов используется библиотека JMock.

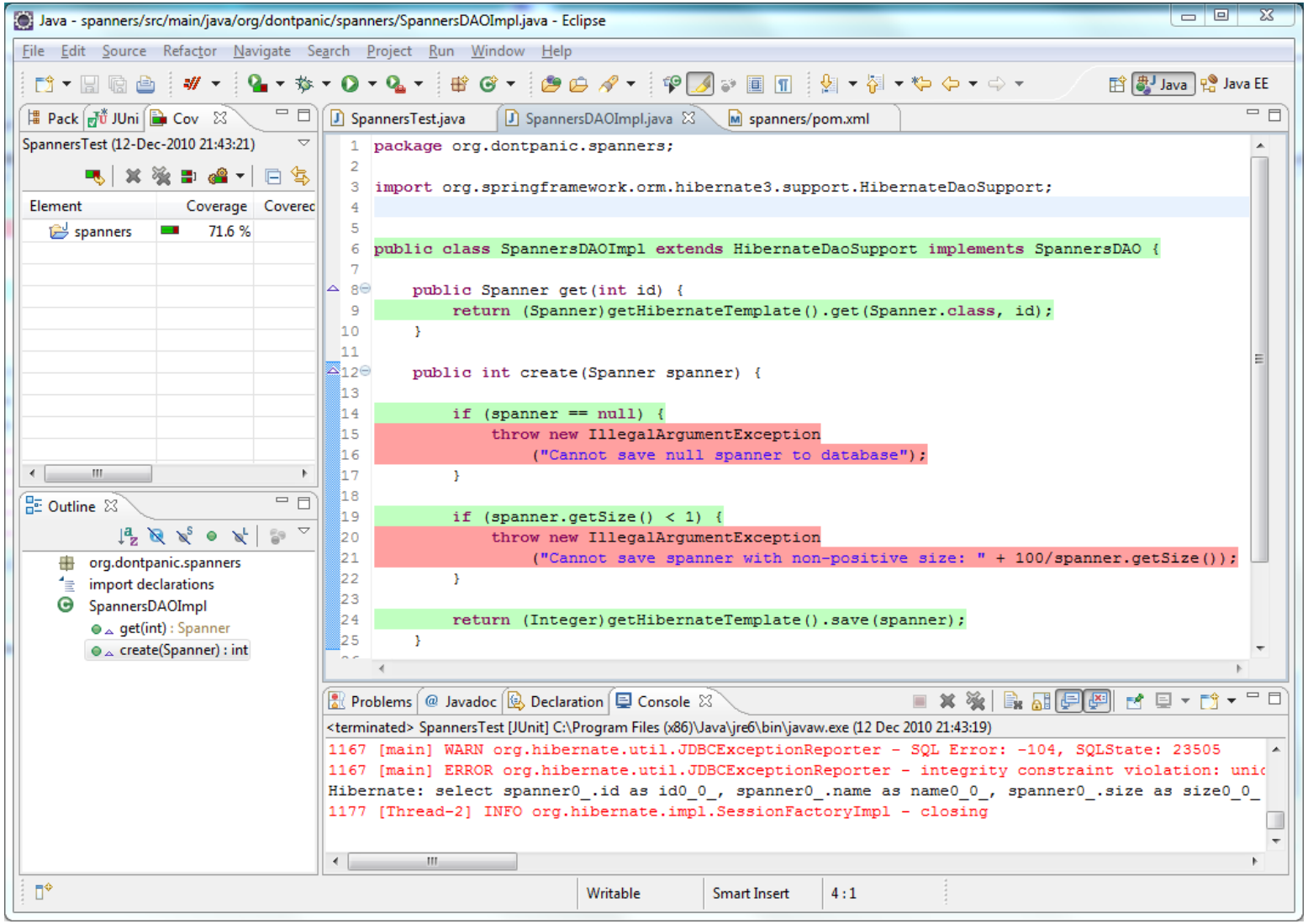
Когда использовать юнит-тесты

- Хорошо изолированный код
- Границы спецификаций и крайние случаи
- Сложные алгоритмы
- Редко модифицируемые библиотеки
- Утилитарные классы общего назначения (например, для работы со временем, форматами данных)

Test-Driven Development



Тестовое покрытие



Библиотеки для юнит-тестирования

Java: JUnit, TestNG
.NET: NUnit
Python: PyUnit
JavaScript: JsUnit
Ruby: RSpec

Автоматические функциональные тесты

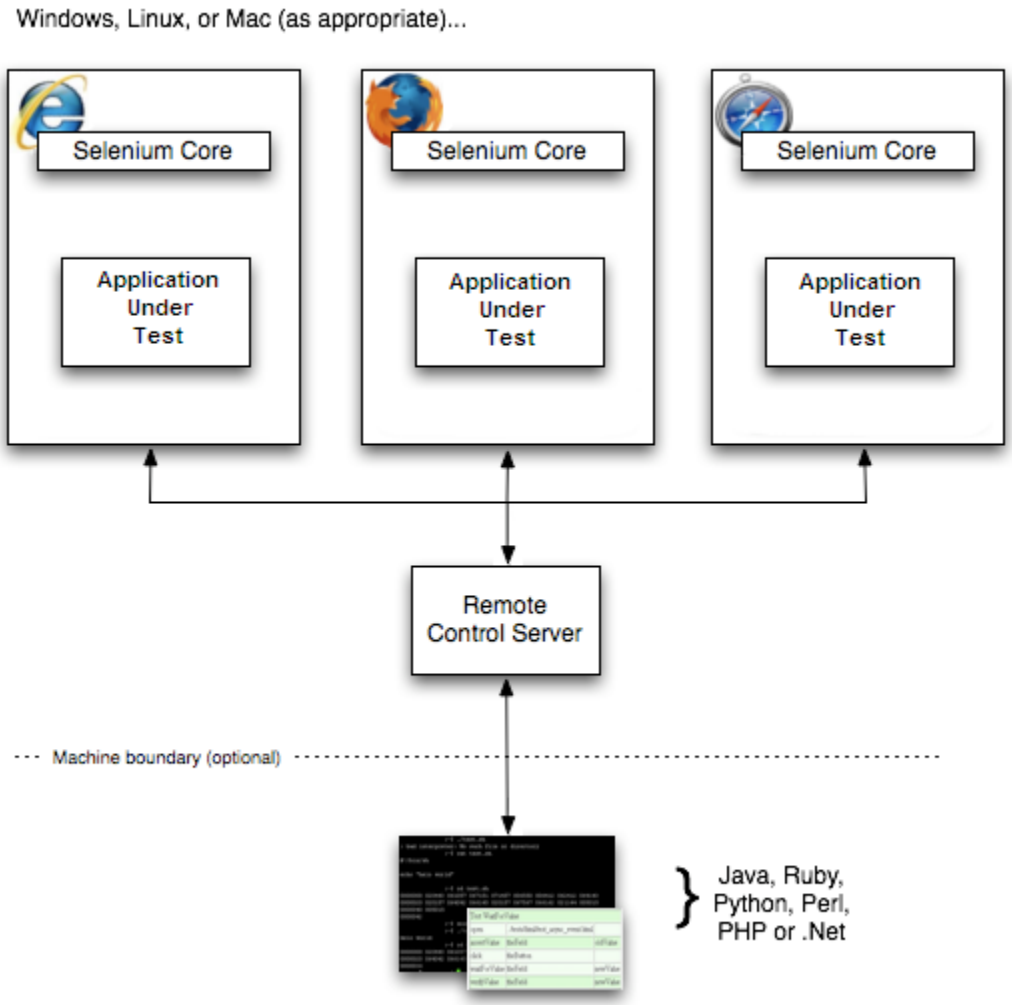
В отличие от юнит-тестов функциональные и интеграционные тесты проверяют значительные части системы в сборе. Точками входа могут служить:

- Внешнее API
- Внутренне API
- Пользовательский интерфейс

Автоматические функциональные тесты требуют для своей работы значительное окружение.

Для работы через API используются те же библиотеки что и для юнит тестирования.

Selenium


Windows, Linux, or Mac (as appropriate)...
Selenium Core
Application Under Test
Selenium Core
Application Under Test
Selenium Core
Application Under Test
Remote Control Server
Machine boundary (optional)
Java, Ruby, Python, Perl, PHP or .Net

Тестирование производительности

- Нагрузочное тестирование (load testing)

Отвечает на вопрос справляется система с запланированной нагрузкой или нет.

- Стресс-тестирование (stress testing)

Отвечает на вопросы с какой максимальной нагрузкой справится система, что произойдет с системой при увеличении нагрузки выше критической.

Тестирование производительности всегда осуществляется в условиях приближенных к реальным.

Нагрузочное тестирование

Для того чтобы провести нагрузочное тестирование необходимы:

- профиль нагрузки
- инструмент позволяющий симулировать нагрузку
- требования к системе под нагрузкой

Результатом нагрузочного тестирования является набор метрик, отражающих поведение системы, таких как:

- среднее/максимальное/минимальное время ответа
- загрузка процессора
- использование памяти
- процент необработанных запросов

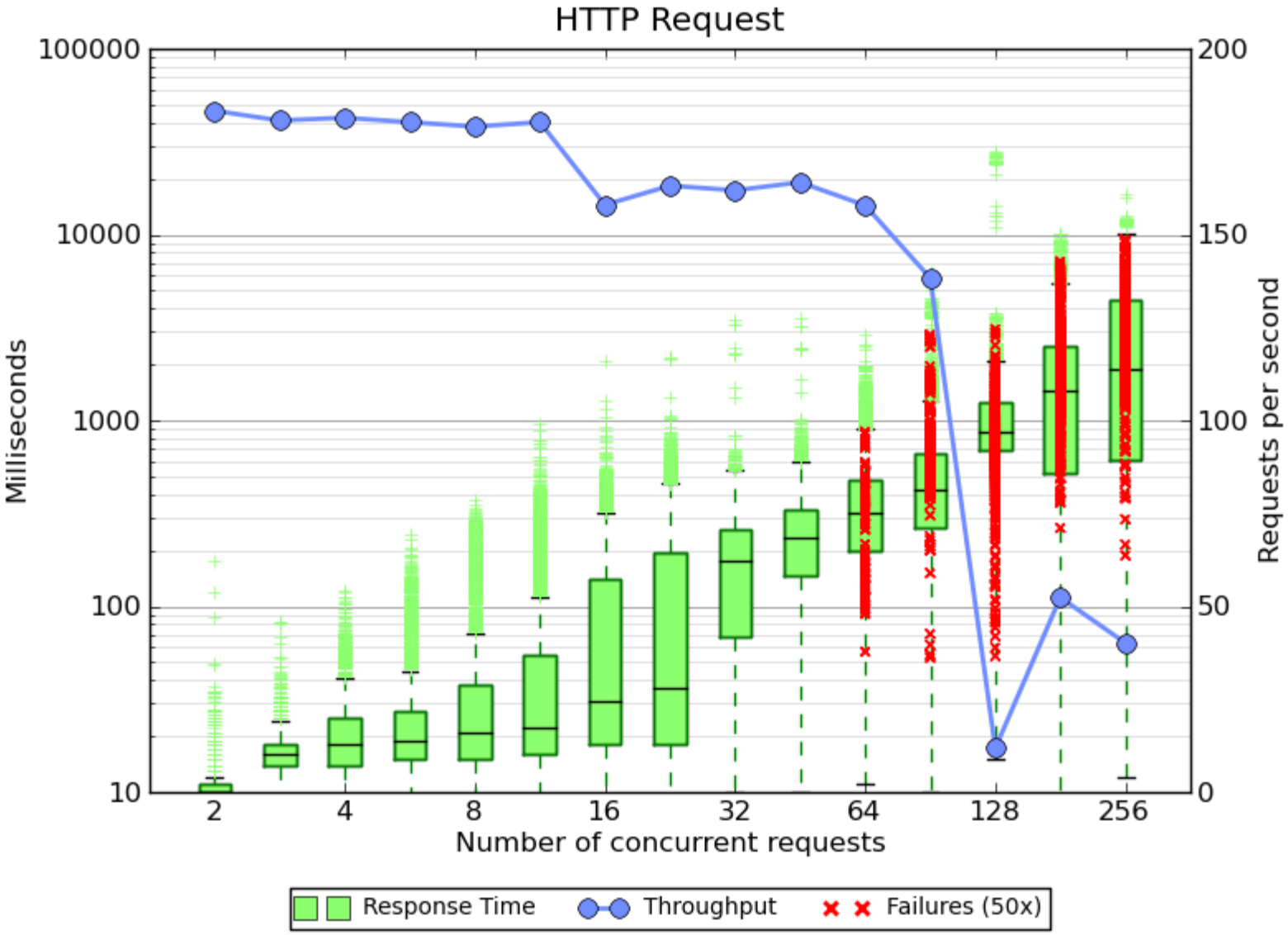
и т.п.

Стресс-тестирование

Стресс тестирование служит для поиска "бутылочных горлышек". Для стресс тестирования необходимы:

- профиль нагрузки
- инструмент позволяющий симулировать постепенное увеличение нагрузки
- тестовое окружение соответствующее реальному

Результатом стресс-тестирования являются графики метрик системы в зависимости от нагрузки.


HTTP Request
Milliseconds
Requests per second
Number of concurrent requests
100000
10000
1000
100
10
200
150
100
50
0
2
4
8
16
32
64
128
256
Response Time
Throughput
Failures (50x)

Рекомендуемая литература

- Список литературы на http://software-testing.ru/
- Блог "Тестирование" на Хабре
- Кент Бек. Экстремальное программирование: разработка через тестирование