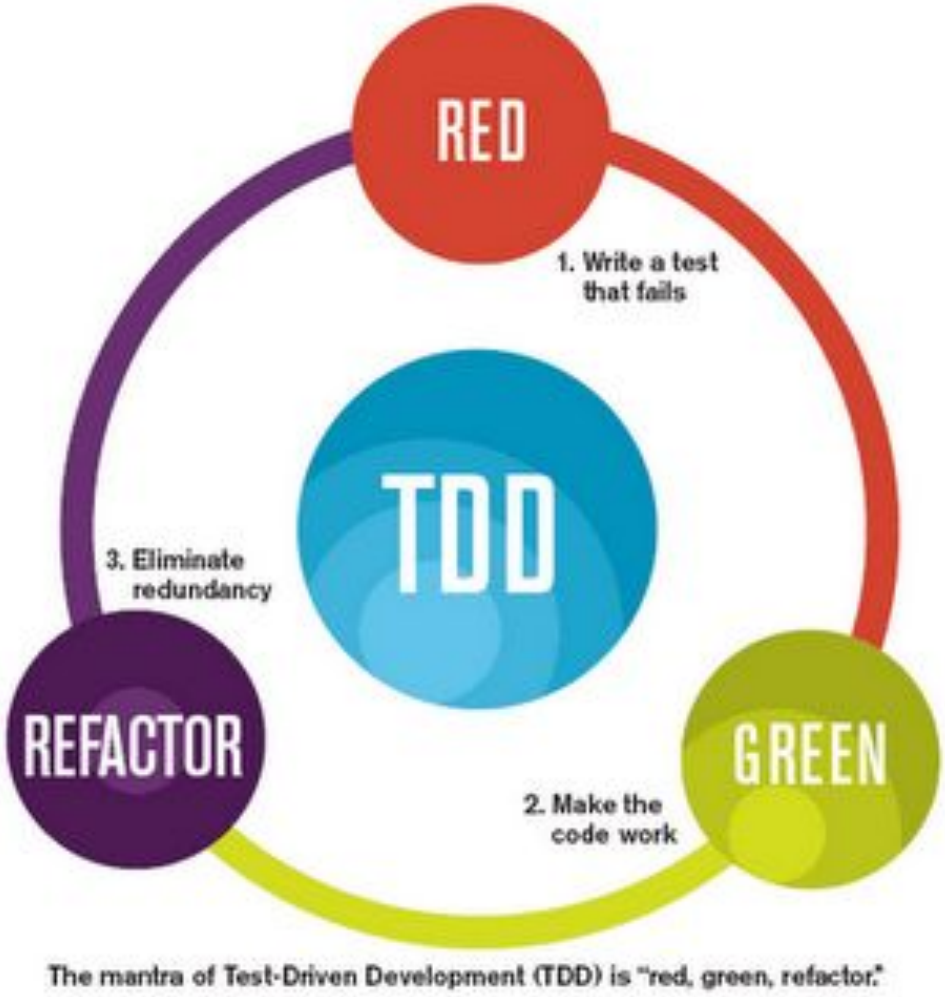
Лекция № 3



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Тема 2. Разработка через тестирование

2. Разработка через тестирование
    2.1. Модульное тестирование.
    2.2. Функциональное тестирование.
    2.3. Другие виды тестов (параллельный тест *(parallel test),* стресс-тест *(stress test) и др.*).
    2.4.Тесты как одна из форм документации.
    2.5. Сложности тестирования.
        2.5.1.Тестирование пользовательского интерфейса.
        2.5.2. Тестирование в ограниченном пространстве.
        2.5.3. Анализ покрытия кода тестами.

ГИБКИЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В соответствие с IEEE Std 829-1983 **Тестирование** — это процесс анализа ПО, направленный на выявление отличий между его реально существующими и требуемыми свойствами (дефект) и на оценку свойств ПО.

Невозможно полностью протестировать программу, поскольку число вариантов работы нетривиальной программы может быть бесконечно большим. Следовательно, тестирование не может доказать отсутствие ошибок в программном коде, оно может показать только наличие ошибок.

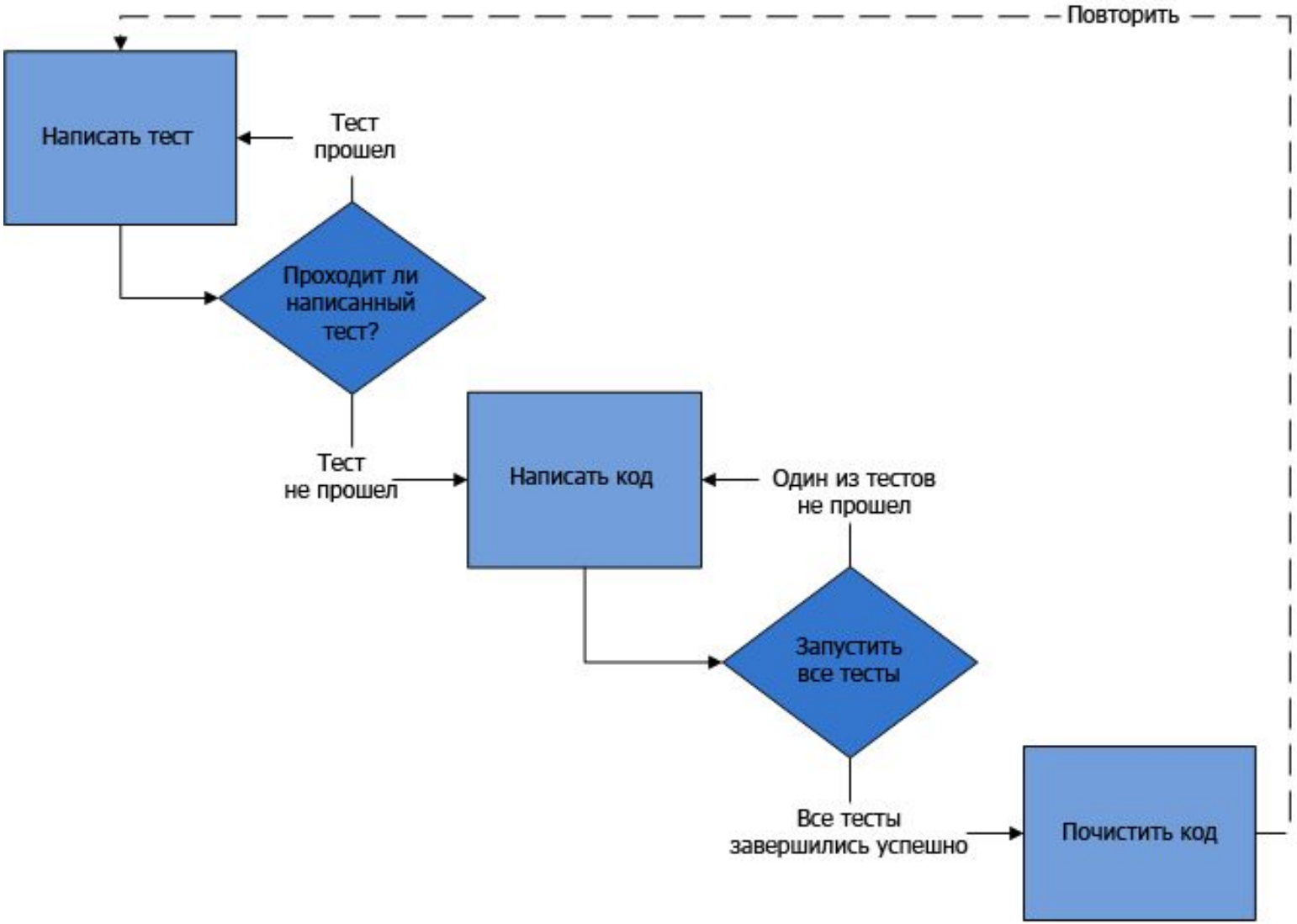
В гибких технологиях применяется подход, известный как *разработка через тестирование (test-driven development).* Благодаря использованию этого подхода гарантируется тестирование всего кода разрабатываемой программной системы.

Тест – это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода.

Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную. В данном контексте тест состоит из двух этапов: стимулирование кода и проверки результатов его работы. Автоматический тест выполняется иначе: вместо программиста стимулированием кода и проверкой результатов занимается компьютер, который отображает на экране результат выполнения теста: код работоспособен или код неработоспособен.

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 и под конец проводится рефакторинг нового кода к соответствующим стандартам. Кент Бек, считающийся изобретателем этой техники, утверждал в 2003 году, что разработка через тестирование поощряет простой дизайн и внушает уверенность (*inspires confidence*).

Графическое представление цикла разработки, в виде блок-схемы



ГИБКИЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В гибких технологиях программирования в основном используется два вида тестирования:

- ✔модульное тестирование *(unit testing);*
- ✔функциональное тестирование (*functional testing).*

Кроме этих двух основных видов тестов могут использоваться и другие виды тестирования, использование которых может быть оправдано в определенных ситуациях.

2.1. Модульное тестирование

Модульное тестирование, или **юнит-тестирование** (англ. *unit testing*) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к *регрессии*, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

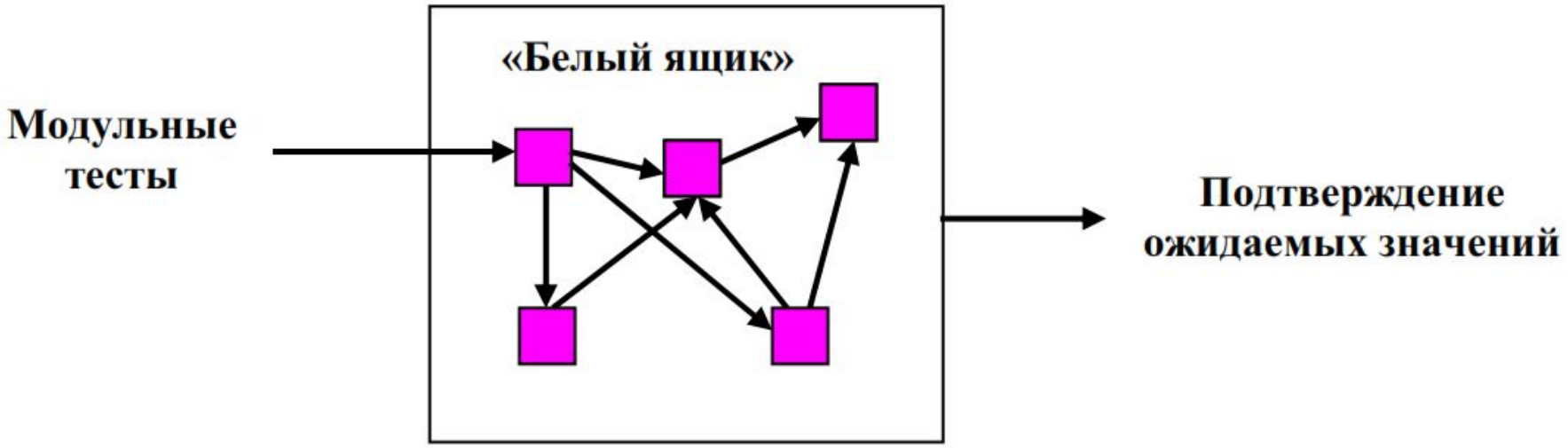


Рис. 3.1. Модульное тестирование

В различный источниках выделяют следующие цели модульного тестирования

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны. Этот тип тестирования обычно выполняется программистами.

Цель модульного тестирования — п*олучение работоспособного кода с наименьшими затратами. И его применение оправдано тогда и только тогда, когда оно дает больший эффект, нежели другие методы.*

Из целей модульного тестирования следует несколько выводов:

✔ **Нет смысла писать тесты на весь код.** Некоторые ошибки проще найти на более поздних стадиях. Так, например, для ООП данное правило может звучать так: нет смысла писать тесты на класс, который используется только одним классом. Эффективней написать тесты на вызывающий класс и создать тесты тестирующие все участки кода.

- ✔ **Писать тесты для кода потенциально подверженного изменениям более выгодно, чем для кода, изменение которого не предполагается.** Сложная логика меняется чаще, чем простая. Следовательно, в первую очередь имеет смысл писать модульные тесты на сложную логику. А на простую логику писать позднее или вообще тестировать другими методами.

✔ **Для того чтобы как можно реже изменять тесты следует хорошо планировать интерфейсы.** То же самое можно сказать и применительно к написанию исходного кода. Действительно, создание хорошей архитектуры часто определяет дальнейший ход проекта. И есть оптимум, на каком этапе архитектура «достаточно хороша».

Если в проекте применяется модульное тестирование, то тщательное планирование интерфейсов становится более выгодным. Внедрению модульного тестирования должно предшествовать внедрение планирования интерфейсов.

Вильям Уэйк (William Wake) для описания порядка написания тестов и кодирования использует следующую ***метафору светофора*.**

Желтый свет - пишите тест и компилируйте. Скомпилировать его не удастся.

Красный свет - пишите код, необходимый для компиляции теста. Скомпилируйте тест, запустите его. Тестирование не пройдет успешно.

Зеленый свет - пишите код, который необходим для успешного прохождения теста. Запустите его. Он пройдет успешно.

Переход от состояния к состоянию, не соответствующий последовательности «желтый - красный - зеленый», является признаком проблемы.

Рассмотрим разные варианты переходов.

От зеленого к зеленому

Только что написанный тест скомпилирован и работает. Это означает, что либо тест тестирует написанные ранее методы, либо тест фиктивный. Чтобы убедиться, что тест корректный, в него необходимо внести ошибку.

От зеленого к красному

Компиляция прошла успешно, но тест не сработал. Если такая ситуация возникает при работе над новым тестом для существующих методов - это нормально. Если же это тест для новых методов, то допущена какая-то ошибка.

От желтого к желтому

В коде заглушки допущена синтаксическая ошибка, которую обнаруживает компилятор.

От желтого к зеленому

Тест для кода заглушки (пустой метод) сработал. Пустые методы, которые не реализуют никаких функций, не должны проходить тесты. Однако если в качестве языка программирования используется C++ или Java, то для методов, возвращающих значение, необходимо ввести оператор возврата, который должен будет в случае успешного прохождения теста возвратить 0 или *null.*

От красного к желтому

В методе, который только что добавлен, есть синтаксическая ошибка, которую обнаруживает компилятор.

От красного к красному

Добавленный метод не работает. Необходимо исправить ошибку и запустить тесты снова.

Уэйк утверждает, что на описанную процедуру должно уходить около 10 - 15 минут для каждого теста. Если разработчик не укладывается в это время, необходимо писать более короткие тесты.

Планирование тестов

Первый вопрос, который встает перед нами: *«Сколько нужно тестов?».* Ответ, который часто дается: тестов должно быть столько, чтобы не осталось неоттестированных участков, т.е.
Код с не оттестированными участками не может быть опубликован.

В тестировании вопрос *«Как я могу сломать?»* гораздо эффективней вопроса *«Как я могу подтвердить правильность?»*. Поэтому в первую очередь тесты должны соответствовать не коду, а требованиям. Правило, которое следует применять:
Тесты должны базироваться на спецификации.

Модульные тесты лежат в основе выполнения **рефакторинга кода**. При этом тестирование необходимо проводить как *перед факторингом,* так и *после рефакторинга.*

Необходимо следить за чистотой тестов столь же ревностно, сколь и за чистотой кода. Чем лучше будут тесты, тем лучше будет код и тем проще будет выполняться его модернизация и добавление в него новых возможностей.

2.2. Функциональное тестирование

Функциональное тестирование соответствует идеологии тестирования «черного ящика», когда разработчик тестов ничего не знает о внутреннем устройстве программы.

Рис. 3.2. Функциональное тестирование

Цель функционального тестирования состоит в том, чтобы убедиться в надлежащем функционировании объекта тестирования. Тестируется правильность навигации по объекту, а также ввод, обработка и вывод данных.

Функциональные тесты рассматривают систему как «черный ящик» и рассчитаны на получение вполне определенного результата работы программного продукта. При тестировании методом «черного ящика» каждый тест никак не взаимодействует с остальными тестами, поэтому сбой в одном из тестов не является причиной несрабатывания других тестов.

Функциональные тесты разрабатываются при участии заказчика для каждого из пожеланий. В идеале разработка функциональных тестов для некоторой итерации завершается еще до того, как разработчики завершат работу над этой итерацией.

Функциональные тесты могут быть описаны на обратной стороне карточки с пожеланиями заказчика или на отдельной карточке (рис.3.3). Во втором случае пожелание заказчика и функциональный тест должны ссылаться друг на друга с помощью уникального номера пожелания заказчика.

Функциональный тест состоит из трех частей:

1. Установка теста - описание минимальных действий, которые необходимо выполнить перед запуском приемочного теста.

2. Сам **тест**, который часто называют *операцией.*

3. Подтверждение - результат выполнения теста (значение или состояние, которое достигается после корректного срабатывания теста).

ГИБКИЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рассмотрим пример создания функционального теста для пожелания заказчика. Пусть, результатом корректного выполнения теста должен быть выведенный на экран список гостиниц со свободными одноместными номерами. Заказчик предлагает рассмотреть несколько вариантов:

- ✔ в базе данных гостиниц имеется всего одна гостиница, удовлетворяющая заданным условиям;
- ✔ в базе данных гостиниц имеется несколько гостиниц, удовлетворяющих заданным условиям;
- ✔ в базе данных гостиниц отсутствует гостиница, удовлетворяющая заданным условиям.

На рис. 3.3 приведено описание функционального теста для первого варианта.

№ 56

Установка: Занести в базу данных гостиницу, в которой есть свободный номер с *05.08.05* по *15.08.05*. Адрес гостиницы: *ул. Мира, д. 14*, стоимость номера 2500 руб.

Операция: Просмотреть базу данных гостиниц и выбрать гостиницу, в которых есть свободные номера на 10.08.05.

Подтверждение: На экран выводится адрес: *ул. Мира, д. 14*, стоимость номера 2500 руб.

Рис. 3.3. Пример приемочного теста

Функциональное тестирование системы должно быть автоматизировано. С помощью автоматизации можно после каждой версии вернуться на предыдущий шаг. Это означает, что все функциональные тесты выполняются в каждой версии, даже те, которые ранее использовались в предыдущих версиях. Если при работе программного продукта возникает ошибка, в набор автоматизированных функциональных тестов добавляется тест, с помощью которого была обнаружена ошибка, и эта ошибка никогда вновь не появится в выпускаемых версиях.

2.3. Другие виды тестов (параллельный тест *(parallel test),* стресс-тест *(stress test) и др.*).

Кроме модульных и функциональных тестов в разработке через тестирование могут использоваться:

Параллельный тест *(parallel test)* предназначен для того, чтобы доказать, что новый программный продукт работает так же, как старый. На самом деле этот тест может только продемонстрировать, что новый программный продукт отличается от старого. При этом только заказчик может принять решение о том, насколько удовлетворительным для него является различие и можно ли он допустить программный продукт в эксплуатацию.

Стресс-тест *(stress test).* Этот вид тестов разрабатывается для имитации максимальной нагрузки на программный продукт. Стресс-тесты должны обязательно разрабатываться для тестирования сложных программных продуктов, для которых трудно делать предположения о производительности.

2.4. Тесты как одна из форм документации

Составление тестов - это неотделимая составная часть разработки кода, поэтому ***тесты являются наилучшей формой документации*** на уровне модулей (классов). Любые потери времени, связанные с написанием тестов, быстро окупаются благодаря существенному сокращению времени, которое приходится тратить на отладку и написание документации.

Тесты являются более подробной формой документации, чем документация, написанная на бумаге, вместе с тем они не содержат никаких лишних сведений. Они без лишних слов коротко и ясно объясняют, как работает тестируемый код.

Однако, отсюда не следует, что документация при использовании гибких технологий разработки вообще не нужна. Письменное описание программного продукта может потребоваться для презентации, защиты проекта, получения лицензии и в других подобных ситуациях.

2.5. Сложности тестирования

В настоящее время программные продукты разрабатываются для самых разных проблемных областей и аппаратных платформ, поэтому разработка тестов может оказаться довольно сложным делом. Рассмотрим ряд примеров

2.5.1. Тестирование пользовательского интерфейса

Часть программной системы, обеспечивающая работу **интерфейса с пользователем** - один из наиболее нетривиальных объектов для верификации. Нетривиальность заключается в двояком восприятии термина "пользовательский интерфейс".

С одной стороны, пользовательский интерфейс - часть программной системы. Соответственно, на пользовательский интерфейс пишутся функциональные и низкоуровневые требования, по которым затем составляются тест-требования и тест-планы. При этом, как правило, требования определяют реакцию системы на каждый ввод пользователя (при помощи клавиатуры, мыши или иного устройства ввода) и вид информационных сообщений системы, выводимых на экран, печатающее устройство или иное устройство вывода. При верификации таких требований речь идет о проверке функциональной полноты пользовательского интерфейса - насколько реализованные функции соответствуют требованиям, корректно ли выводится информация на экран.

С другой стороны, пользовательский интерфейс - "лицо" системы, и от его продуманности зависит эффективность работы пользователя с системой. Факторы, влияющие на эффективность работы, слабо поддаются формализации в виде конкретных требований к отдельным элементам, однако должны быть учтены в виде общих рекомендаций и принципов построения пользовательского интерфейса программной системы. Проверка интерфейса на эффективность человеко-машинного взаимодействия получила название **проверки удобства использования** (usability verification ; в русскоязычной литературе в качестве перевода термина usability часто используют слово "практичность").

С другой стороны, пользовательский интерфейс - "лицо" системы, и от его продуманности зависит эффективность работы пользователя с системой. Факторы, влияющие на эффективность работы, слабо поддаются формализации в виде конкретных требований к отдельным элементам, однако должны быть учтены в виде общих рекомендаций и принципов построения пользовательского интерфейса программной системы. Проверка интерфейса на эффективность человеко-машинного взаимодействия получила название проверки удобства использования (usability verification ; в русскоязычной литературе в качестве перевода термина usability часто используют слово "практичность").

Функциональное тестирование пользовательского интерфейса состоит из пяти фаз:

- ✔анализ требований к пользовательскому интерфейсу;
- ✔разработка тест-требований и тест-планов для проверки пользовательского интерфейса;
- ✔выполнение тестовых примеров и сбор информации о выполнении тестов;
- ✔определение полноты покрытия пользовательского интерфейса требованиями;
- ✔составление отчетов о проблемах в случае несовпадения поведения системы и требований либо в случае отсутствия требований на отдельные интерфейсные элементы.

Все эти фазы точно такие же, как и в случае тестирования любого другого компонента программной системы. Отличия заключаются в трактовке некоторых терминов в применении к пользовательскому интерфейсу и в особенностях автоматизированного сбора информации на каждой фазе.

Этапы тестирования удобства использования пользовательского интерфейса.

Исследовательское - проводится после формулирования требований к системе и разработки прототипа интерфейса. Основная цель на этом этапе - провести высокоуровневое обследование интерфейса и выяснить, позволяет ли он с достаточной степенью эффективности решать задачи пользователя.

Оценочное - проводится после разработки низкоуровневых требований и детализированного прототипа пользовательского интерфейса. Оценочное тестирование углубляет исследовательское и имеет ту же цель. На данном этапе уже проводятся количественные измерения характеристик пользовательского интерфейса: измеряются количество обращений к системе помощи по отношению к количеству совершенных операций, количество ошибочных операций, время устранения последствий ошибочных операций и т.п.

.

Валидационное - проводится ближе к этапу завершения разработки. На этом этапе проводится анализ соответствия интерфейса программной системы стандартам, регламентирующим вопросы удобства интерфейса (например ISO 13407, ISO 9126), проводится общее тестирование всех компонент пользовательского интерфейса с точки зрения конечного пользователя. Под компонентами интерфейса здесь понимается как его программная реализация, так и система помощи и руководство пользователя. Также на данном этапе проверяется отсутствие дефектов удобства использования интерфейса, выявленных на предыдущих этапах.

Сравнительное - данный вид тестирования может проводиться на любом этапе разработки интерфейса. В ходе сравнительного тестирования сравниваются два или более вариантов реализации пользовательского интерфейса.

ГИБКИЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Наиболее часто тестирование пользовательских интерфейсов сводится к тестированию *удобства и простоты использования* программного продукта. Критерии оценки удобства и простоты использования должны быть сформулированы заранее. Например, можно использовать следующие критерии.

Доступность. Насколько легко пользователи могут входить в систему, ориентироваться и выходить из системы?

Способность реагировать. Насколько быстро программный продукт позволяет пользователям достичь определенных целей?

Эффективность. Сколько шагов необходимо сделать, чтобы получить выбранную функциональность?

Ясность. Насколько часто пользователи пользуются документацией и вызывают помощь?

Необходимое количество пользователей, которых необходимо привлечь к тестированию, определяется статистически и зависит от предполагаемого числа пользователей продукта и желаемой вероятности правильного заключения.

2.5.2. Тестирование в ограниченном пространстве

Разработчики, создающие программное обеспечение для миниатюрных компьютерных устройств, например, сотовых телефонов или цифровых фотоаппаратов, работают в среде, далекой от условий применения программного обеспечения. Например, при разработке Java-приложений для обычных компьютеров можно использовать любые из стандартных классов и возможностей Java, в то время, как небольшой размер устройства не позволит разместить все классы Java. В этом случае необходимо создать новую среду для тестирования, при этом дополнительные усилия, затраченные на разработку, модификацию или адаптацию библиотеки тестирования для миниатюрного компьютерного устройства, как правило, окупят себя.

2.5.3. Анализ покрытия кода тестами

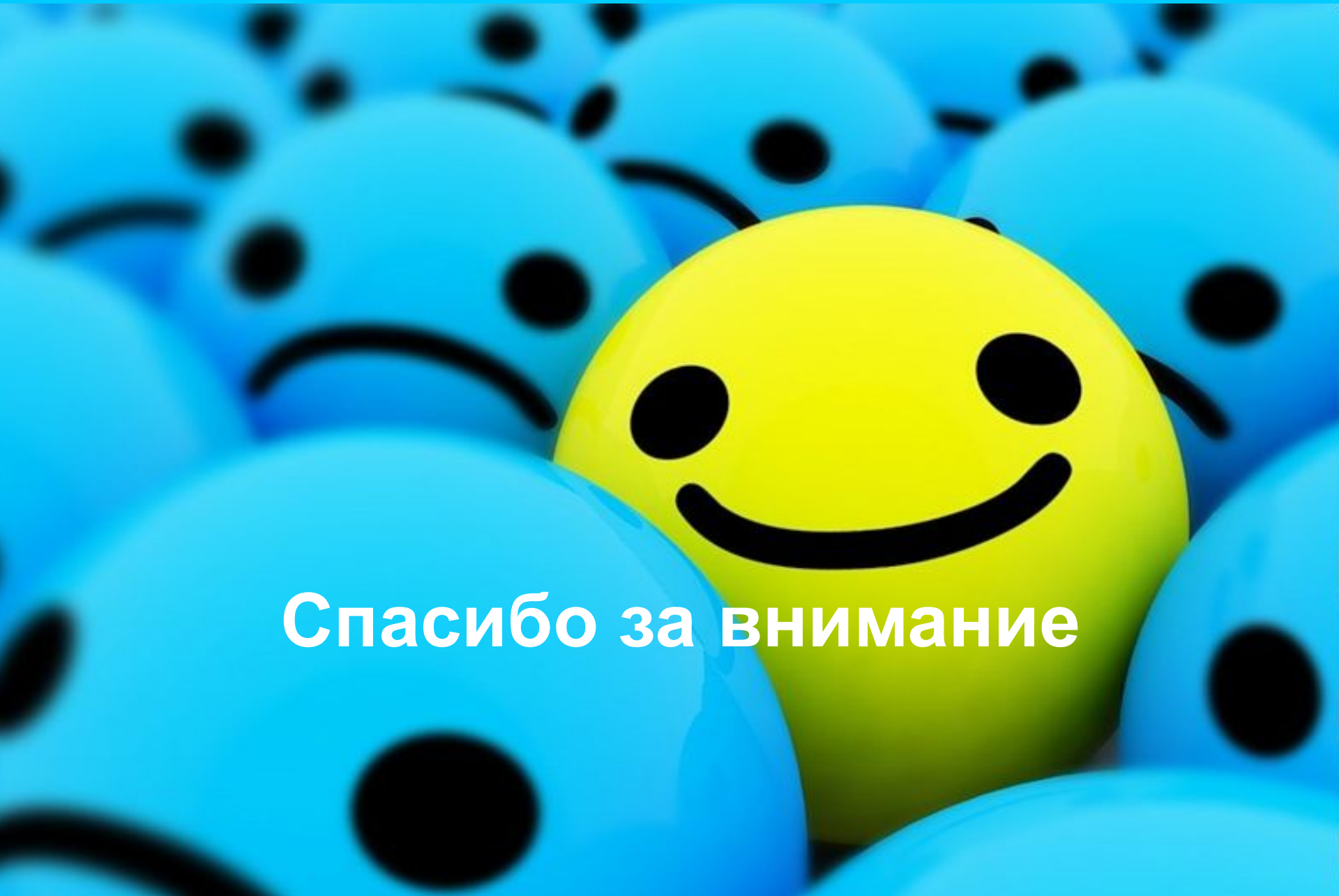
Один из эффективных инструментов, для определения полноты тестового набора — **матрица покрытия**.

Обязательно учитывайте следующее:

✔На каждое требование должен быть, как минимум, один тест. Неважно, ручной или автоматический.

✔Простой позитивный тест нужен обязательно т.к. несмотря на малую вероятность нахождения ошибки, цена пропущенной ошибки чрезмерно высока.

✔Наиболее эффективный способ создания тестового набора — совместное использование методов черного и белого ящиков., чтобы обеспечить проверку полноты тестового набора.


Спасибо за внимание