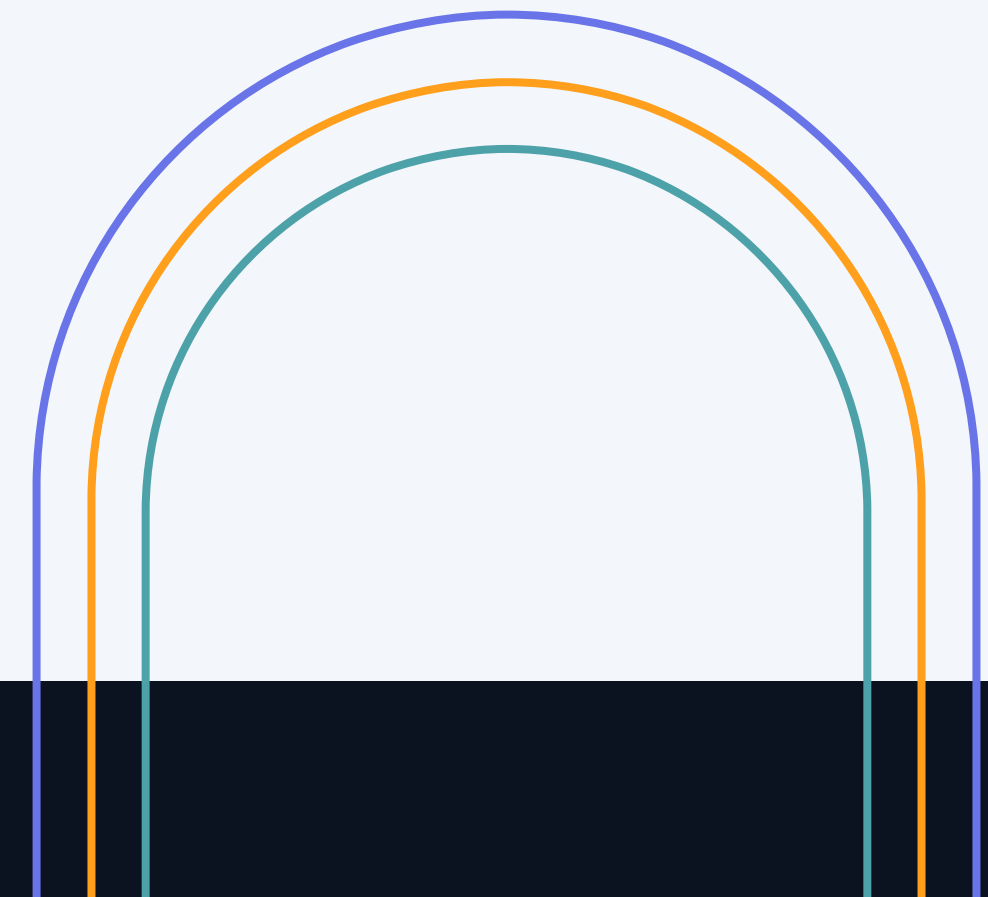




# Лекция 6. Тестирование (обзорная)

- Определение
- История развития тестирования
- Важность тестов
- Пирамида тестирования
- Тестирование белого/черного/серого ящиков
- Пример поиска и исправления ошибки
- Разработка через тестирование
- Используемые источники



# Определение

---



## Тестирование программного обеспечения

проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом

## В более широком смысле, тестирование

это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).



# История тестирования

50-60-х

- В 50–60-х годах прошлого века процесс тестирования был предельно формализован, отделён от процесса непосредственной разработки ПО и «математизирован». Фактически тестирование представляло собой скорее отладку программ

70-х

- В 70-х годах фактически родились две фундаментальные идеи тестирования: тести-рование рассматривалось как доказательство работоспособности программы в некоторых заданных условиях, а затем - наоборот: как процесс доказательства неработоспособности программы в некоторых заданных условиях. Это внутреннее противоречие не только не исчезло со временем, но и в наши дни отмечается как две взаимодополняющие цели тестирования.

80-х

- В 80-х годах произошло ключевое изменение места тестирования в разработке ПО: тестирование стало применяться на протяжении всего цикла разработки, что позволило в очень многих случаях не только быстро обнаруживать и устранять проблемы, но даже предсказывать и предотвращать их появление. В этот же период времени отмечено бурное развитие и формализация методологий тестирования и появление первых попыток автоматизировать тестирование.

90-х

- В 90-х годах произошёл переход от тестирования к более всеобъемлющему процессу, который называется «обеспечение качества» (Quality Assurance, QA), который охватывает весь цикл разработки ПО и затрагивает процессы планирования, проектирования, создания и выполнения тест-сценариев, поддержку имеющихся тест-сценариев и тестовых окружений.

Н.В.

- В настоящее время развитие тестирования продолжается в контексте поиска всё новых путей, методологий, техник и подходов к обеспечению качества. Серьёзное влияние на понимание тестирования оказало появление гибких методологий разработки и таких подходов, как «разработка через тестирование (test-driven development, TDD)». Автома-тизация тестирования уже воспринималась как обычная неотъемлемая часть большинства проектов, а также стали популярны идеи о том, что во главу процесса тестирования следует ставить не соответствие программы требованиям, а её способность предоставить конечному пользователю возможность эффективно решать свои задачи.

# Важность тестов



# Важность тестов

## Первое:

Какая бы методология разработки программного обеспечения не применялась, роль процесса тестирования для обеспечения качества продукта трудно переоценить. Тестирование является составляющей частью процесса отладки ПО, т.к. после выявления ошибок, дефекты должны быть устранены разработчиками в программном коде.

## Третье:

Понимание важности процесса тестирования приводит к возникновению тенденций, направленных на применение промышленных способов проверки качества программного обеспечения. Наиболее важным направлением здесь является внедрение различных систем автоматизированного тестирования.

## Второе:

Задачами современного тестирования является не только обнаружение ошибок в программах, но и выявление причин их возникновения. Такой подход позволяет разработчикам функционировать максимально эффективно, быстро устраняя возникающие ошибки..

## Тестирование позволяет:

- *выявлять значительное количество дефектов программы на как можно более ран-них стадиях;*
- *позволяют выявить такое количество ошибок, чтобы продукт мог поступить к конечному пользователю.*

# Пирамида тестирования



# Пирамида тестирования

---



Автоматизированное тестирование тесно связано с именем Майка Кона, который представил процесс автоматизации проверки систем и продуктов в форме пирамиды. Разработанная Майком пирамида состоит из трёх элементов.

Юнит-тесты  
(модульные тесты)

Сервисные тесты

Тесты пользовательского  
интерфейса  
(тесты User Interface, UI)



*Тем не менее, суть тестовой пирамиды представляет хорошее правило, и из этой пирамиды главное запомнить два принципа:*

- 1. Писать тесты разной детализации;**
- 2. Чем выше уровень, тем меньше тестов (на 1000 юнит-тестов должно приходиться около 300 сервисных тестов и порядка 30 тестов пользовательского интерфейса).**

*Иногда данная концепция кажется недостаточной. С современной точки зрения, пирамида тестов кажется чрезмерно упрощённой и поэтому может вводить в заблуждение.*

# Пирамида тестов Майка Кона



# Юнит-тесты

**Существует несколько правил написания**



## Правило 1

Быть достоверными

## Правило 2

Не зависеть от окружения,  
на котором они  
выполняются

## Правило 3

Легко читаться и быть простыми  
для понимания (даже новый  
разработчик должен понять что  
именно тестируется);

## Правило 4

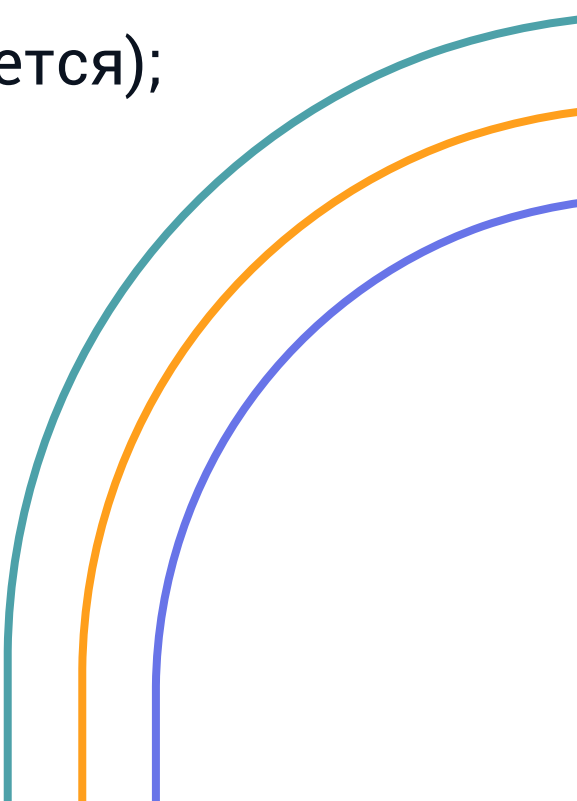
Легко поддерживаться

## Правило 5

Соблюдать единую  
конвенцию именования

## Правило 6

Запускаться регулярно в  
автоматическом режиме;



# Для выполнения этих пунктов, есть несколько советов:

## 1. Выберите способ именования проектов с тестами

*Используйте такой же способ именования для тестовых классов*  
У вас есть класс *ProblemResolver*? Добавьте в тестовый проект *ProblemResolverTests*.  
*Каждый тестирующий класс должен тестировать только одну сущность.*

## 2. Выберите «говорящий» способ именования методов тестирующих классов

**TestLogin** – не самое лучшее название метода. Что именно тестируется? Каковы входные параметры? Могут ли возникать ошибки и исключительные ситуации? Можно ввести подобную конвенцию наименования тестов:

**[Тестируемый метод]\_[Сценарий]\_[Ожидаемое поведение].**

Рассмотрим пример:

Предположим, что у нас есть класс *Calculator*, а у него есть метод *Sum*, который должен складывать два числа. В этом случае наш тестирующий класс будет выглядеть так:

```
class CalculatorTests {  
    public void Sum_2Plus5_7Returned() {  
        // ...  
    }  
}
```

## 3. Тестируйте один аспект поведения за один раз

*Каждый тест должен проверять только одну вещь. Если процесс слишком сложен (например, покупка в интернет магазине), разделите его на несколько частей и протестируйте их отдельно.*  
*Если вы не будете придерживаться этого правила, ваши тесты станут нечитаемыми, и вскоре вам окажется очень сложно их поддерживать.*



# Интеграционные тесты



Все приложения интегрированы с некоторыми другими компонентами (базы данных, файловые системы, сетевые вызовы к другим приложениям). Для тестирования корректного взаимодействия взаимодействия предназначены интеграционные тесты. Они проверяют интеграцию приложения со всеми частями вне приложения.

Для автоматизированных тестов это означает, что нужно запустить не только собственное приложение, но и интегрируемый компонент. Если вы тестируете интеграцию с БД, то при выполнении тестов надо запустить БД. Чтобы проверить чтение файлов с диска нужно сохранить файл на диск и загрузить его в интеграционный тест.

Что касается пирамиды тестов, то интеграционные тесты находятся на более высоком уровне, чем модульные. Интеграция файловых систем и БД обычно гораздо медленнее, чем выполнение юнит-тестов с их имитациями. Их также труднее писать, чем маленькие изолированные модульные тесты.

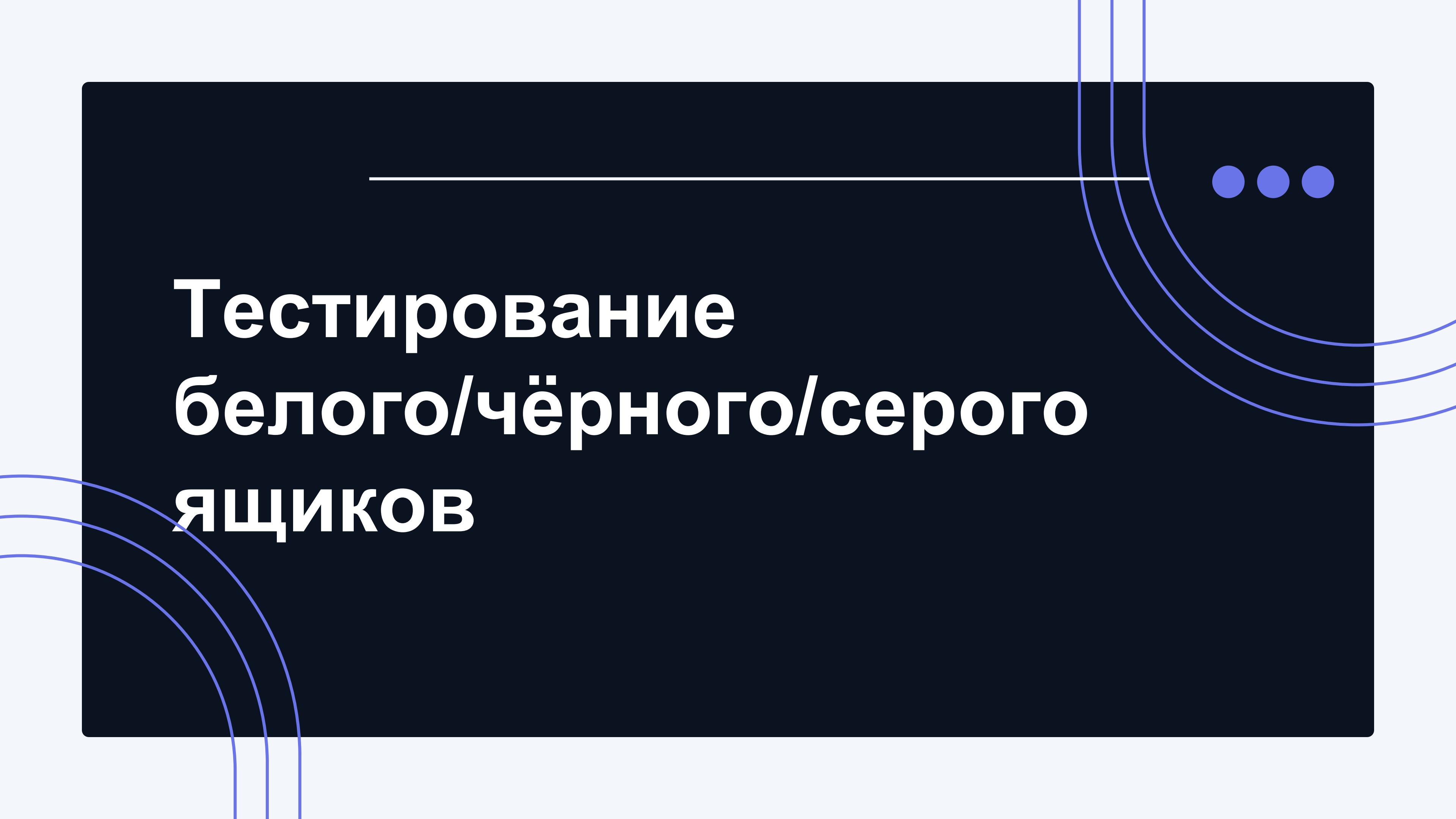
---

Тем не менее, они имеют преимущество, потому что дают уверенность в правильной работе приложения со всеми внешними компонентами

# Тесты пользовательского интерфейса—

Основные плюсы UI тестирования:	
Покрывает большую часть пользовательских действий и позволяет, со стороны пользователя, «потрогать» приложение;	
Проверяет взаимодействие компонентов и сервисов между собой;	
Увеличивает надежность приложения.	

Основные минусы UI тестирования:	
Огромное количество времени, потраченного на тест всех компонентов и сер-висов;	
Мало применимо для маленьких по размеру приложений;	
Сами тесты выполняются дольше, чем Unit, из-за сложности используемых сервисов;	
Более сложный процесс автоматизации.	



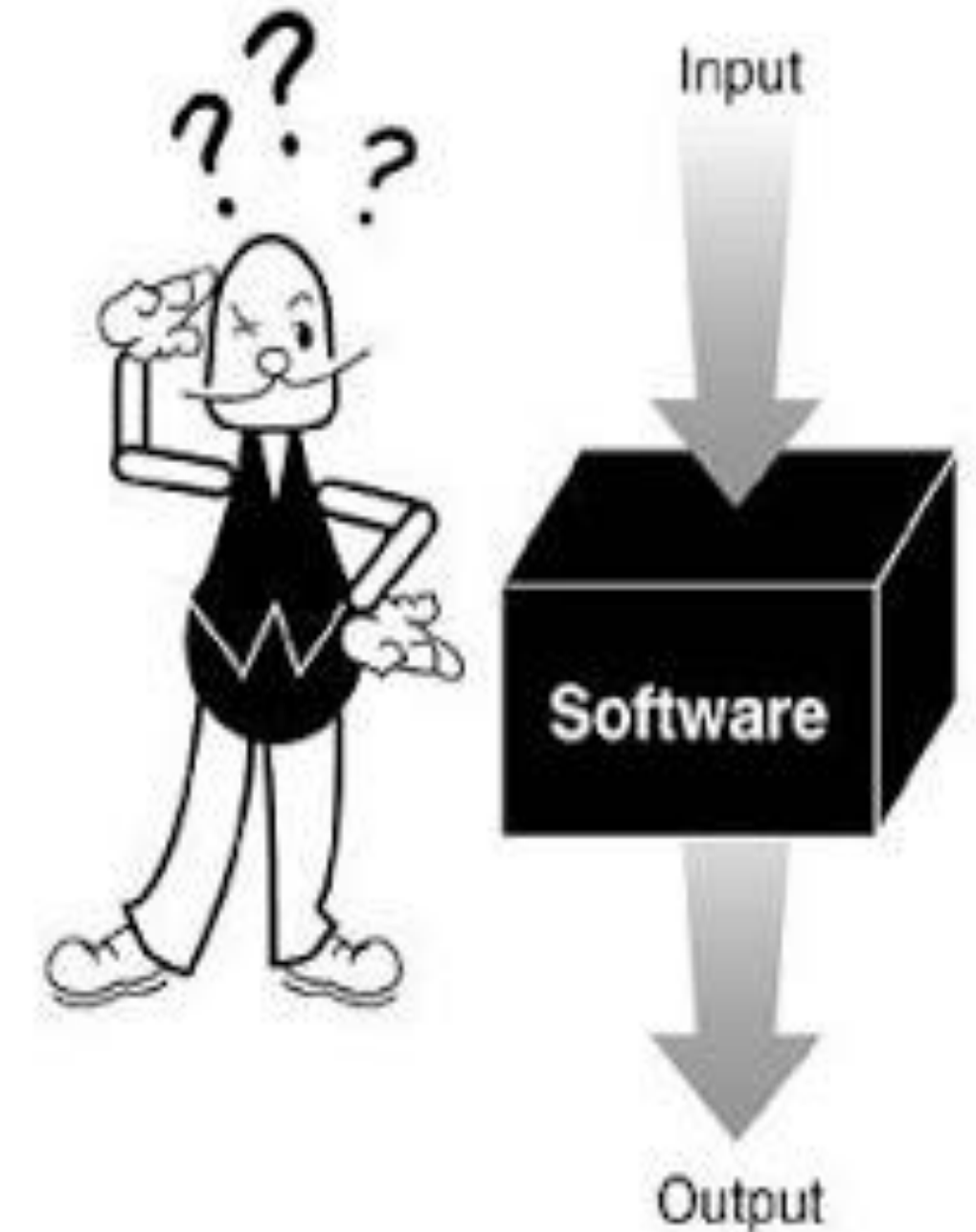
# Тестирование белого/чёрного/серого ящиков

**Тестирование способом черного ящика** - это стратегия или метод тестирования, базируется только лишь на тестировании по функциональной спецификации и требованиям, при этом не имея доступа во внутреннюю структуру кода и базу данных.

Целью этой техники является поиск ошибок в таких категориях:

- неправильно реализованные или недостающие функции;
- ошибки интерфейса;
- ошибки в структурах данных или организации доступа к внешним базам данных;
- ошибки поведения или недостаточная производительности системы

*Таким образом, мы не имеем представления о структуре и внутреннем устройстве системы. Нужно концентрироваться на том, что программа делает, а не на том, как она это делает.!*



Black-Box Testing

# Черный ящик (Black Box Testing)



# Черный ящик (Black Box Testing)

## Преимущества

1

- тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
- тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
- можно начинать писать тестовые сценарии, как только готова спецификация.

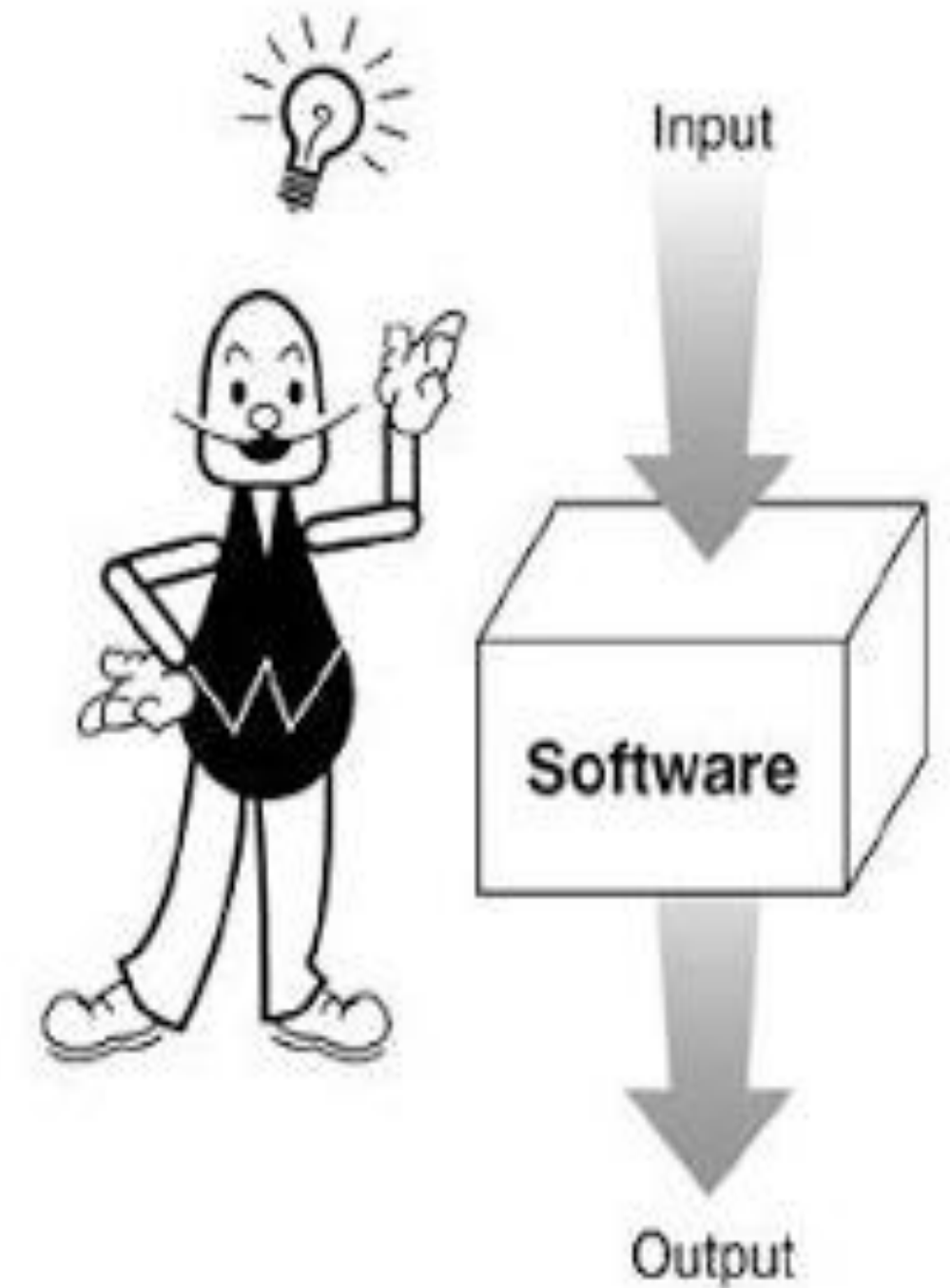
## Недостатки

2

- тестируется только очень ограниченное количество путей выполнения программы;
- без четкой спецификации достаточно трудно составить эффективные тестовые сценарии;
- некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования;

**Тестирование способом белого ящика** - (также: прозрачного, открытого, стеклянного ящика; основанное на коде или структурное тестирование) – метод тестирования программного обеспечения, который предполагает, что внутренняя устройство системы известны тестировщику.

- Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутренне устройство системы, за пределы ее внешних интерфейсов



White-Box Testing

**Белый ящик**  
**(White Box Testing)**



# Белый ящик (White Box Testing)

## Преимущества

1

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

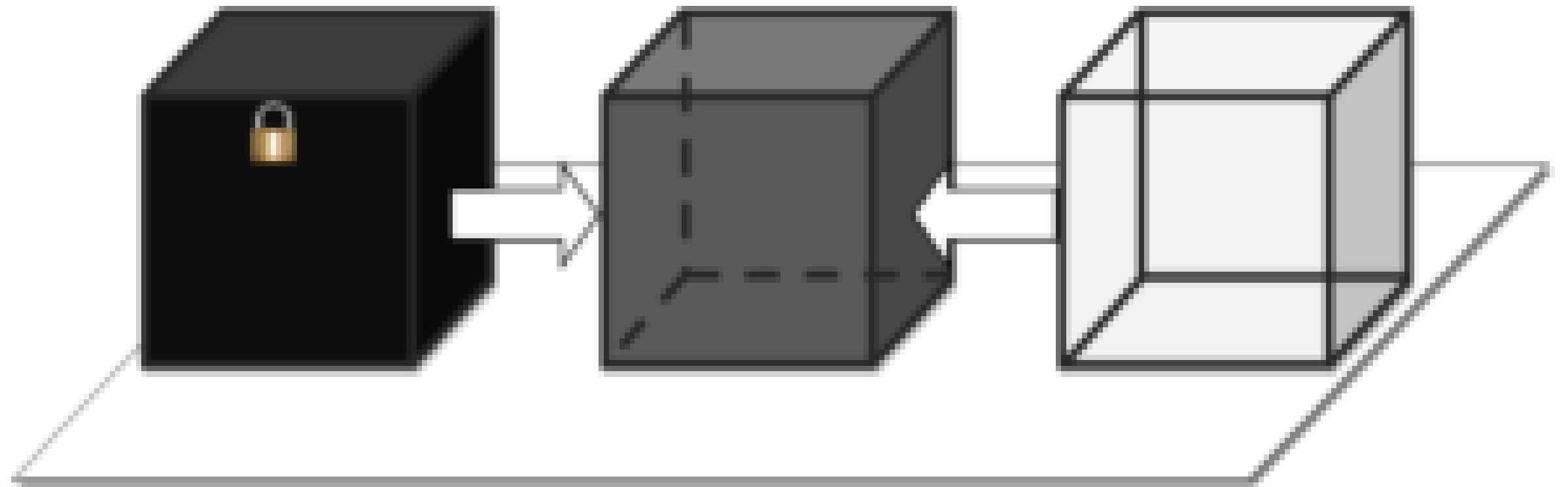
## Недостатки

2

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

**Тестирование способом серого ящика** - метод тестирования программного обеспечения, который предполагает, комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ к внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов, но само тестирование проводится с помощью техники черного ящика, то есть, с позиции пользо

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



**Серый ящик**  
**(Grey Box Testing)**



# Пример поиска и исправления ошибки



# Пример поиска и исправления ошибки



**Отладка обеспечивает локализацию ошибок, поиск причин ошибок и соответствующую корректировку программы**

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double Power(double x, int n)
{
    double z=1;

    for (int i=1;n>=i;i++)
    {
        z = z*x;
    }
    return z;
}
```

2.1. Исходный текст метода Power

```
double Power(double x,int n)
{
    double z=1;
    int i;
    for(i=1;n>=i;i++)
    {
        z=z*x;
    }
    return z;
}
```

**Если вызвать метод Power с отрицательным значением степени n Power(2,-1), то получим некорректный результат 1. Исправим метод так, чтобы ошибочное значение параметра (недопустимое по спецификации значение) идентифицировалось специальным сообщением, а возвращаемый результат был равен 1.**

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double
PowerNonNeg(double x, int n)
{
    double z=1;
    if (n>0)
    {
        for (int i=1;n>=i;i++)
        {
            z = z*x;
        }
    }
    else Console.WriteLine(
        "Ошибка ! Степень числа n" +
        " должна быть больше 0.");
    return z;
}
```

2.2. Скорректированный исходный текст  
double PowerNonNeg(double x, int n)

```
{
    double z=1;
    int i;
    if (n>0)
    {
        for (i=1;n>=i;i++)
        {
            z = z*x;
        }
    }
    else printf("Ошибка! Степень числа n
должна быть больше 0.\n");
    return z;
}
```

*Если вызвать скорректированный метод PowerNonNeg(2,-1) с отрицательным значением параметра степени, то сообщение об ошибке будет выдано автоматически.*

# Разработка через тестирование

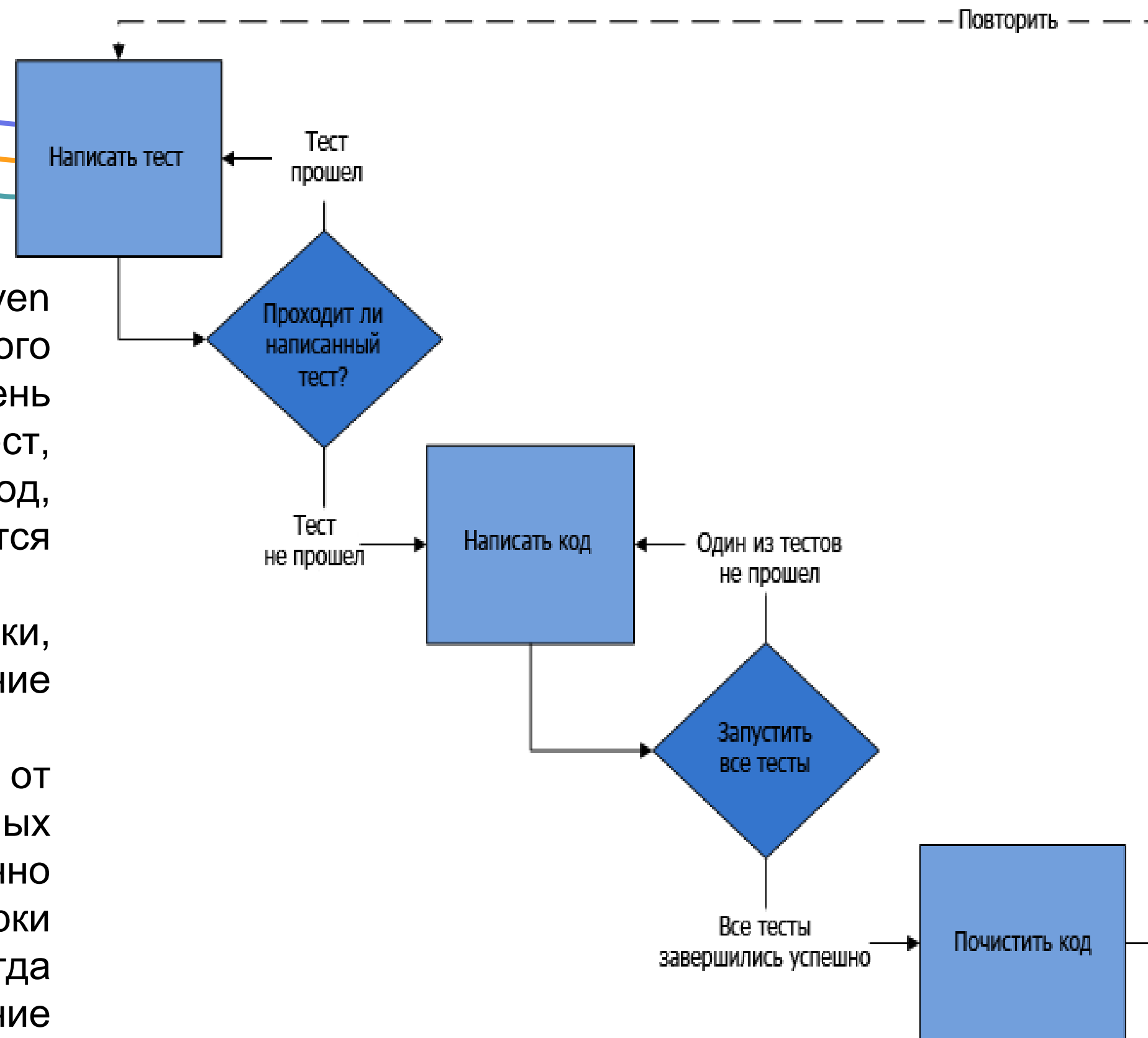


Одной из современных методологий разработки является такая интересная технология, как разработка через тестирование.

**Разработка через тестирование** (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, за-тем 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.

Разработка через тестирование требует от разработчика создания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом.



Графическое представление цикла разработки методом TDD

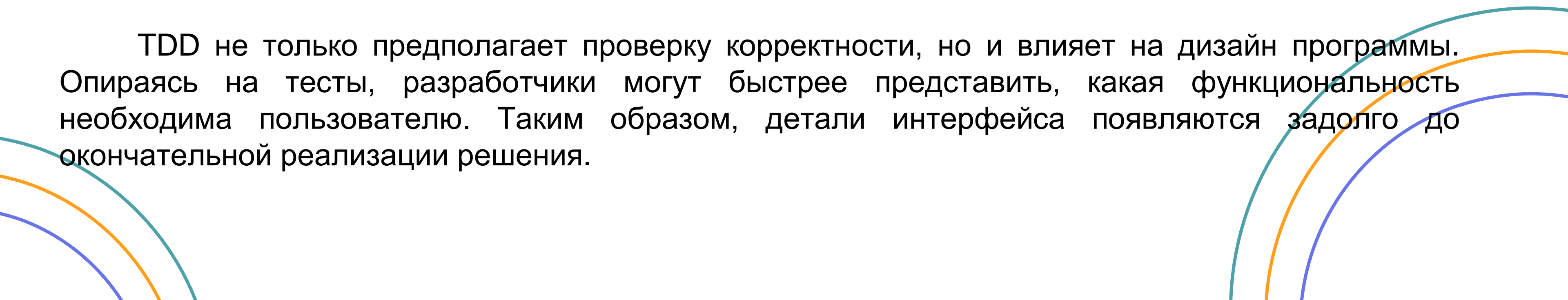




Разработчики часто пользуются библиотеками для тестирования для создания и автоматизации запуска наборов тестов. На практике модульные тесты покрывают критические и нетривиальные участки кода. Это может быть код, который подвержен частым изменениям, код, от работы которого зависит работоспособность большого количества другого кода, или код с большим количеством зависимостей.

Среда разработки должна быстро реагировать на небольшие модификации кода. Архитектура программы должна базироваться на использовании множества сильно связанных компонентов, которые слабо сцеплены друг с другом (low coupling, high cohesion, один из принципов проектирования в ООП, подробнее можно прочитать [тут](#)), благодаря чему тестирование кода упрощается.

TDD не только предполагает проверку корректности, но и влияет на дизайн программы. Опираясь на тесты, разработчики могут быстрее представить, какая функциональность необходима пользователю. Таким образом, детали интерфейса появляются задолго до окончательной реализации решения.



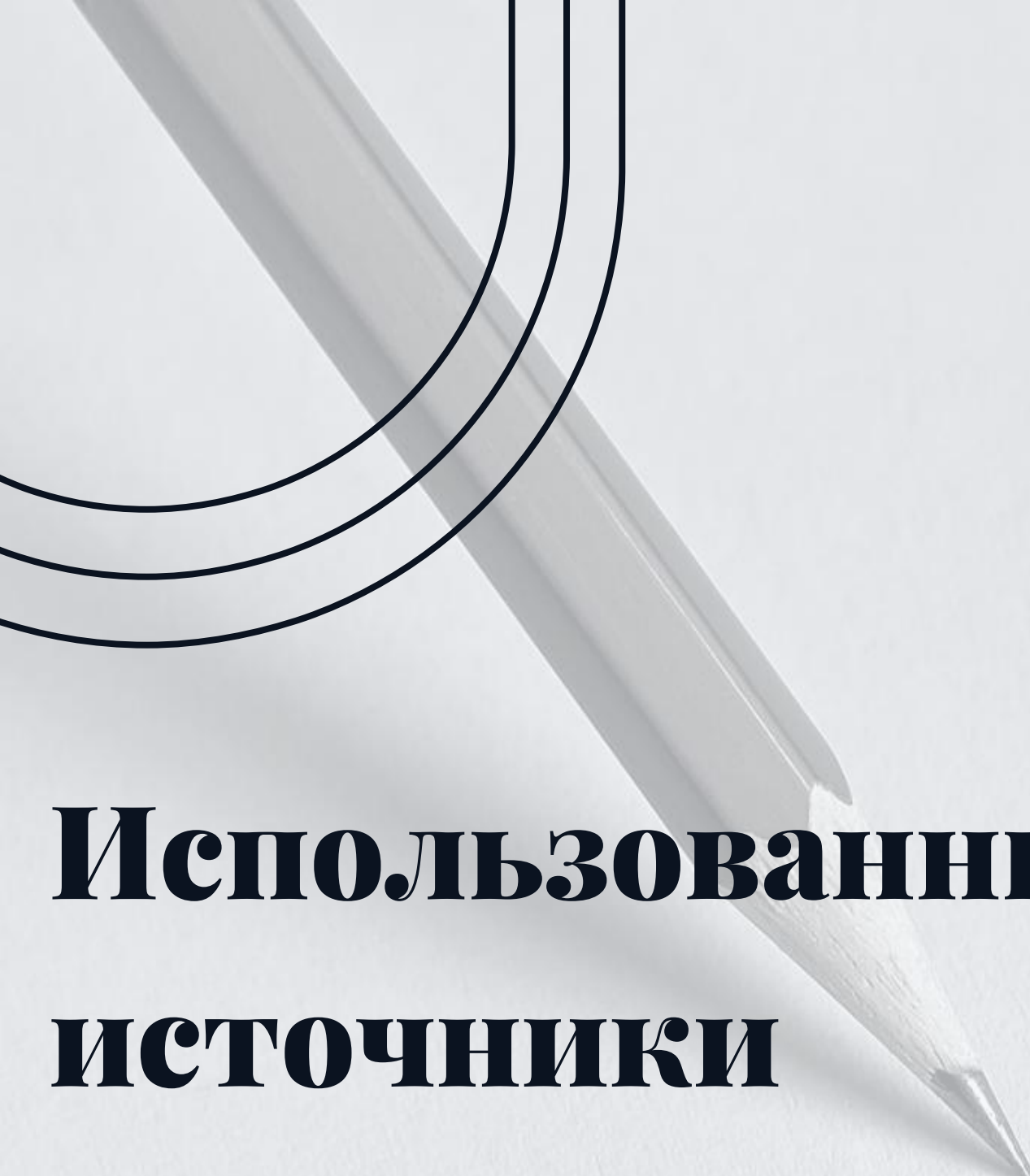
# Разработка через тестирование

## Преимущества

- Программисты, использующие TDD на проектах, отмечают, что они реже ощущают необходимость использовать отладчик;
- Разработка через тестирование предлагает больше, чем просто проверку корректности, она также влияет на дизайн программы. Изначально сфокусировавшись на тестах, проще представить, какая функциональность необходима пользователю. Та-ким образом, разработчик продумывает детали интерфейса до реализации;
- Несмотря на то, что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше. Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно;
- Тесты позволяют производить рефакторинг (улучшение качества кода без изменения поведения) кода без риска его испортить. При внесении изменений в хорошо протестированный код риск появления новых ошибок значительно ниже;
- Разработка через тестирование способствует более модульному, гибкому и расширяемому коду;
- Поскольку пишется лишь тот код, что необходим для прохождения теста, автоматизированные тесты покрывают все пути исполнения.

## Недостатки

- Существуют задачи, которые невозможно решить только при помощи тестов. В частности:
- TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами;
- Разработку через тестирование сложно применять в тех случаях, когда для тестирования необходимо прохождение функциональных тестов (разработка интерфейсов пользователя, программ, работающих с базами данных, а также того, что зависит от специфической конфигурации сети);
- Тесты сами по себе являются источником накладных расходов. Плохо написанные тесты, например, содержат жёстко вшитые строки с сообщениями об ошибках или подвержены ошибкам, дороги при поддержке. Чтобы упростить поддержку тестов, следует повторно использовать сообщения об ошибках из тестируемого кода;



# Использованные источники

<https://habr.com/post/358950/>

[http://svyatoslav.biz/software\\_testing\\_book\\_download/](http://svyatoslav.biz/software_testing_book_download/)

<http://www.infosoftcom.ru/article/rol-testirovaniya-pri-razrabotke-programm>

[http://quality-lab.ru/bad\\_automation\\_testing\\_organization\\_example\\_and\\_its\\_debriefing/](http://quality-lab.ru/bad_automation_testing_organization_example_and_its_debriefing/)

<http://ru.qatestlab.com/knowledge-center/qa-testing-materials/test-automated-pyramid/>

<https://qalight.com.ua/baza-znaniy/white-black-grey-box-testirovanie/>

<https://software-testing.org/testing/testirovanie-chernogo-yaschika-black-box-testing.html>

<https://ru.wikipedia.org/wiki/>

<https://habr.com/post/169381/>





# Спасибо!

Жду ваших вопросов...

