МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра информационных систем управления

Отчет

по учебной практике (1 курс)

Выполнил:

Студент 1 курса 13 группы

Никончик Даниил Викторович

Минск 2020

Оглавление

1. Работа с Microsoft Word

Цель: углубить знания и улучшить навыки работы с Microsoft Office Word 2019.

1.1. Рассылки

С помощью рассылок создаём приглашение на свадьбу.

Исходный текст:

Здравствуйте уважаемый

Никончик Даниил Викторович

Дорогой Даниил Викторович.

Вы приглашены на учебу в нашем университете.

12.09.2019

Администрация

В качестве источника данных была использована таблица Microsoft Excel. Для создания фонового изображения использовалась программа Adobe Photoshop 2019

Скриншоты:

Фамилия Имя Отчество	Пол	Имя отчество
Аксютин Павел Евгеньевич	1	Павел Евгеньевич
Белевич Михаил Андреевич	1	Михаил Андреевич
Белицкий Евгений Игоревич	1	Евгений Игоревич
Бислюк Артём Сергеевич	1	Артём Сергеевич
Бокун Адам Антониевич	1	Адам Антониевич
Болотова Анастасия Михайловна	0	Анастасия Михайловна
Булах Артём Игоревич	1	Артём Игоревич
Врублевская Екатерина Александровна	0	Екатерина Александровна
Горбач Дмитрий Николаевич	1	Дмитрий Николаевич
Гущеня Артём Юрьевич	1	Артём Юрьевич
Доскоч Роман Дмитриевич	1	Роман Дмитриевич
Ильющенко Владислав Игоревич	1	Владислав Игоревич
Ищенко Иван Сергеевич	1	Иван Сергеевич
Кадол Валерий Александрович	1	Валерий Александрович
Клевко Надежда Александровна	0	Надежда Александровна
Козунов Алексей Леонидович	1	Алексей Леонидович
Коростелёва Анастасия Леонидовна	0	Анастасия Леонидовна
Кукса Андрей Фёдорович	1	Андрей Фёдорович
Левданская Елизавета Витальевна	0	Елизавета Витальевна
Лигай Владислва Евгеньевич	1	Владислав Евгеньевич
Ляхнович Кирилл Александрович	1	Кирилл Александрович
Мигас Злата Игоревна	0	Злата Игоревна
Никончик Даниил Викторович	1	Даниил Викторович
Овезов Ровшан Чалыевич	1	Ровшан Чалыевич
Руткевич Родион Александрович	1	Родион Александрович
Семенович Дмитрий Анатольевич	1	Дмитрий Анатольевич

Итоговый вариант:

Здравствуйте уважаемый

Никончик Даниил Викторович

Дорогой Даниил Викторович.

Вы приглашены на учебу в нашем университете.

12.09.2019

Администрация

2. Работа с СУБД на примере Microsoft Access

Цель: научиться работать с СУБД. Освоить запросы и формы Microsoft Access.

2.1. Запросы Access

Постановка задачи:

Создать различные запросы к предоставленной базе данных «Online Shop».

Запросы на соединение (при возможности выполнить внешнее соединение, левое или правое).

1. Соединить таблицы «заказ», «клиент», «состояние заказа» (в базе данных могут быть клиенты, ещё не оформившие ни одного заказа).

2. Соединить таблицы «заказ», «курьер», «состояние заказа» (в базе данных могут быть курьеры, ещё не доставившие ни одного заказа и заказы, не обработанные курьерами).

3. Соединить таблицы «заказ», «товар», «состав заказа».

4. Соединить таблицы «поставщик», «товар», «склад» (в базе данных могут быть клиенты, ещё не выполнившие ни одной поставки).

Поиск данных

1. Найти клиентов, заказы которых были отменены.

2. Найти товары, поставщики которых располагаются в Минске.

3. Найти товары, хранящиеся на складах с ёмкостью, не меньшей 1000.

4. Найти товары, отправленные заказчикам не ранее апреля 2017 года.

5. Найти заказы, в составе которых есть товары по цене, меньшей 1000.

Группировка данных

1. Посчитать количество заказов у каждого клиента, сделавшего хотя бы один заказ.

2. Посчитать количество товаров в составе каждого заказа.

3. Найти максимальную цену поставки каждого товара.

4. Посчитать суммарное количество товаров, хранящихся на каждом складе.

5. Посчитать количество товаров, доставленных каждым курьером.

6. Найти максимальную дату заказа в разрезе состояний заказа.

Параметризованные запросы

Выполнить один из запросов из раздела «поиск данных», задавая критерий запроса в качестве параметра.

Скриншоты:

НомерЗаказа	ФИО	НазваниеСс
1	Растопыркин І	принят
2	Растопыркин І	отменён
8	Растопыркин І	доставляется
6	Рукоблудов Л	принят
7	Рукоблудов Л	в обработке
10	Рукоблудов Л	доставлен
13	Рукоблудов Л	принят
3	Филинов В. Д.	доставляется
4	Филинов В. Д.	принят

Поставщик.НАИМЕНОВАНИЕ	Товар.НАИМЕНОВАНИЕ	НОМ_СКЛА	АДРЕС	ЕМКОСТЬ
ЗАО «БелХард Групп»	Стиральная машина	1	г. Минск Респ	1000
ЗАО «БелХард Групп»	Ноутбуки HP Pavilion	3	г. Минск Респ	250
ЗАО «БелХард Групп»	Монитор HP Pavilion	2	г. Минск Респ	500
Samsung	Телевизор 50"" серия 6 3D Ful	3	г. Москва Росс	250
BenQ	Фотоаппарат цифровой	1	г. Тайбэй о-в Т	1000
BenQ	Монитор	2	г. Тайбэй о-в Т	500
BenQ	Проектор	3	г. Тайбэй о-в Т	250
ОАО "Горизонт"	Телевизор	1	г. Минск ул. Ки	1000
ОАО "Горизонт"	Пылесос	1	г. Минск ул. Ки	1000
ОАО "Горизонт"	Электрочайник	3	г. Минск ул. Ки	250
ЗАО "Атлант"	Холодильник	1	220035 г. Минс	1000
ЗАО "Атлант"	Электрочайник	3	220035 г. Минс	250
ЗАО «БелХард Групп»	ПРИНТЕР HP COLOR LASERJET		г. Минск Респ	
Samsung	Смартфон	1	г. Москва Росс	1000
Samsung	"Ноутбук 15	2	г. Москва Росс	500
*		(№)		

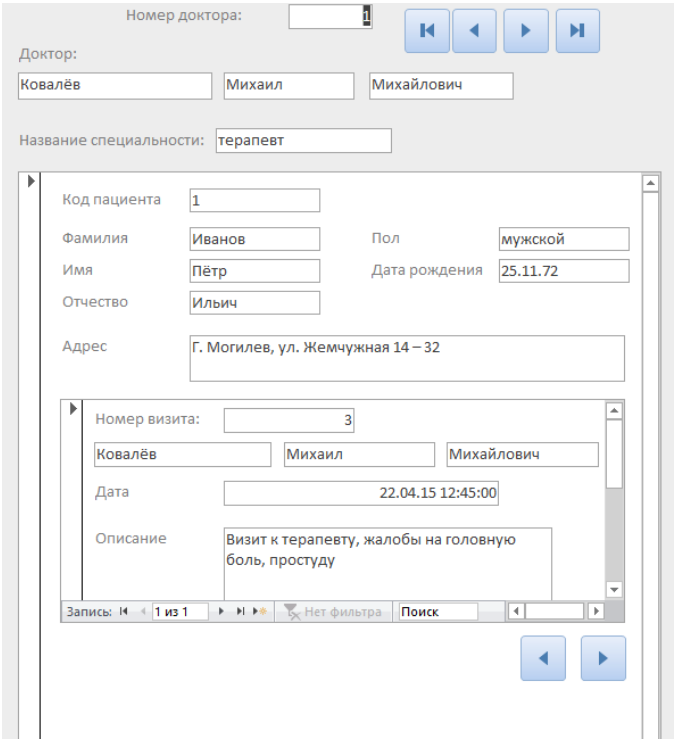
КОДКЛИЕНТА	НАИМЕНОВАНИЕ	АДРЕС	КОНТАКТ	НомерЗаказа	ДатаЗаказа	ДатаПостав	НазваниеСостояния
1	Бендер О. И.	г. Черноморск	o.bender@rogacopyta.com	1	09.03.15		принят
1	Бендер О. И.	г. Черноморск	o.bender@rogacopyta.com	2	10.03.15	12.03.15	в обработке
2	ООО "Рога и Копыта"	г. Черноморск	rogacopyta@mail.com	3	01.04.15	03.04.15	доставляется
2	ООО "Рога и Копыта"	г. Черноморск	rogacopyta@mail.com	4	05.04.15		принят
3	Корейко А. И.	г. Урюпинск	koreyko@aport.ru	5	17.05.15	19.05.15	доставлен
4	ИП "Безенчук и сыновья"	г. Артемовск	bezenchuk@yahoo.com				

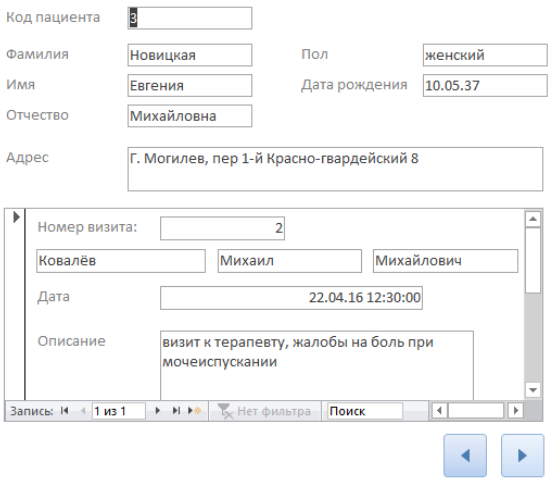
2.2. Формы Access

Постановка задачи:

Создать формы для базы данных для представления данных в виде, удобном для пользователя.

Скриншоты:





3. Работа с C# скриптами и Unity 2D

Цель: изучить C# скрипты и их функционал для создания движка для ролевой игры.

3.1. Создание скриптов для ролевой игры на C#

Постановка задачи:

Часть 1

Создать класс «персонаж ролевой игры». Включить в описание класса следующие поля:

- уникальный числовой идентификатор (*);
- имя персонажа (*);
- состояние (нормальное, ослаблен, болен, отравлен, парализован, мёртв);
- возможность разговаривать в текущий момент времени;
- возможность двигаться в текущий момент времени;
- раса (человек, гном, эльф, орк, гоблин) (*);
- пол (*);
- возраст;
- текущее значение здоровья персонажа (неотрицательная величина);
- максимальное значение для здоровья персонажа;
- количество опыта, набранное персонажем.

Звездочкой помечены поля, не изменяющиеся после создания персонажа.
Реализовать:

- конструктор, задающий значения неизменяемых полей и обеспечивающий уникальность идентификатора для нового объекта;
- свойства для всех полей (доступ к полям может быть реализован только при помощи свойств);
- сравнение персонажей по опыту через реализацию интерфейса IComparable;
- если процент здоровья персонажа (отношение текущего здоровья персонажа к максимальному количеству здоровья) становится менее 10, персонаж

автоматически переходит из состояния «здоров» в состояние «ослаблен».
Если процент здоровья персонажа становится большим или равным 10, персонаж автоматически переходит из состояния «ослаблен» в состояние «здоров». Если текущее значение здоровья равно 0, персонаж автоматически

переходит из любого состояния в состояние «мертв».

- вывод информации о персонаже в строку (через метод ToString).

Создать класс-потомок «персонаж, владеющий магией». Дополнительно включить в описание этого класса следующие поля:

- текущее значение магической энергии (маны) (неотрицательная величина);
- максимальное значение маны.

Мана расходуется на произнесение заклинаний. Если текущее значение маны меньше того количества, которое требуется для произнесения какого-либо заклинания, заклинание не может быть произнесено, а количество маны остается
неизменным.

Некоторые заклинания обладают силой, причем сила заклинания задается волшебником в момент его произнесения. Расход маны в этом случае пропорционален силе заклинания. Сила заклинания ограничивается текущим значением маны.

Реализовать заклинание «добавление здоровья». Суть этого заклинания – увеличить текущее значение здоровья какого-либо персонажа (в том числе и себя) до
максимального или до предела, задаваемого текущим значением маны. На единицу добавленного здоровья расходуется две единицы маны.

Часть 2

Создать интерфейс «волшебство», включающий перегруженные методы «выполнить волшебное воздействие». В качестве параметра указать персонажа
ролевой игры, на которого может быть направлено воздействие, а также силу воздействия. Оба параметра могут отсутствовать.

Создать абстрактный класс «заклинание», реализующий указанный интерфейс.

Включить в описание класса следующие поля:

- минимальное значение маны, требуемое для выполнения заклинания (может быть равно 0);
- наличие вербальной компоненты (заклинание нужно произносить);
- наличие моторной компоненты (необходимо выполнять какие-то движения);

Реализовать классы заклинаний:

1) «Добавить здоровье». Суть этого заклинания – увеличить текущее значение здоровья какого-либо персонажа на заданную величину или до предела,
задаваемого текущим значением маны. На единицу добавленного здоровья расходуется две единицы маны.

2) «Вылечить». Суть этого заклинания – перевести какого-либо персонажа из состояния «болен» в состояние «здоров или ослаблен». Текущая величина здоровья не изменяется. Заклинание требует 20 единиц маны.

3) «Противоядие». Суть этого заклинания – перевести какого-либо персонажа из состояния «отравлен» в состояние «здоров или ослаблен». Текущая
величина здоровья не изменяется. Заклинание требует 30 единиц маны.

4) «Оживить». Суть этого заклинания – перевести какого-либо персонажа из состояния «мертв» в состояние «здоров или ослаблен». Текущая величина здоровья становится равной 1. Заклинание требует 150 единиц маны.

5) «Броня». Персонаж, на которого обращено заклинание, становится неуязвимым в течение некоторого промежутка времени, определяемого силой заклинания. Заклинание требует 50 единиц маны на единицу времени.

6) «Отомри!» Суть этого заклинания – перевести какого-либо персонажа из
состояния «парализован» в состояние «здоров или ослаблен». Текущая величина здоровья становится равной 1. Заклинание требует 85 единиц маны.

Создать абстрактный класс «артефакт», реализующий указанный интерфейс. Включить в описание класса следующие поля:

- мощность артефакта (величина, аналогичная количеству маны у персонажа, владеющего магией) – может быть равен 0;
- признак возобновляемости артефакта.

Реализовать классы артефактов:

1) Бутылка с живой водой – увеличивает здоровье персонажа. Здоровье
персонажа не может превысить максимальную величину, но артефакт используется полностью! Могут быть малые, средние и большие бутылки, увеличивающие здоровье соответственно на 10, 25 и 50 единиц. Не возобновляемый.

2) Бутылка с мертвой водой – увеличивает ману персонажа, владеющего магией. Мана не может превысить максимальную величину, но артефакт
используется полностью! Могут быть малые, средние и большие бутылки

увеличивающие ману соответственно на 10, 25 и 50 единиц. Не возобновляемый.

3) Посох «Молния». Уменьшает количество здоровья персонажа, против которого был применен этот артефакт, на величину, заданную мощностью (мощность задаётся персонажем при использовании артефакта). Мощность посоха уменьшается на эту величину. Возобновляемый, но непригоден для
использования, если его мощность равна нулю.

4) Декокт из лягушачьих лапок. Переводит какого-либо персонажа из состояния «отравлен» в состояние «здоров или ослаблен». Текущая величина здоровья не изменяется. Не возобновляемый.

5) Ядовитая слюна (накладка на зубы, через которую надо плевать). Переводит какого-либо персонажа из состояния «здоров или ослаблен» в состояние «отравлен». Текущая величина здоровья уменьшается на величину, задаваемую мощностью артефакта. При применении этого артефакта
персонаж, против которого он был применен, может умереть!
Возобновляемый.

6) Глаз василиска. Переводит любого не мёртвого персонажа в состояние «парализован». Не возобновляемый.

Реализовать лечение, которое было описано в части 1, через произнесение соответствующего заклинания и использование артефакта.

Часть 3

Дополнить созданные в частях 1 и 2 классы следующими возможностями:

1) У каждого персонажа игры есть мешок (inventory), куда можно помещать различные артефакты (количество артефактов одного вида неограниченно) и использовать их. Если артефакт не является возобновляемым, он исчезает из мешка. Можно использовать только те артефакты, которые имеются в мешке.

2) Персонаж, владеющий магией, может изучить различные заклинания. После изучения заклинания могут быть реализованы. Можно реализовывать только изученные заклинания.

Реализовать методы:
«Подобрать артефакт и пополнить мешок»
«Выбросить артефакт из мешка»

«Передать артефакт другому персонажу»
«Использовать артефакт»
«Выучить заклинание»
«Забыть заклинание»
«Произнести заклинание»

Один из классов:

```
public class inventory
{
    public bool droped = true;
    public Artefact[] container = new Artefact[5] { null, null, null, null, null };
    public int count = 0;
    private void add_item(Artefact art)
    {

        for(int i = 0; i < 5; i++)
        {
            if (container[i] == null)
            {
                container[i] = art;
                count++;
                return;
            }
        }
    }
    private void check_add(Artefact art)
    {
        if (count < 5)
        {
            add_item(art);
            Debug.Log(art.name_ + "Добавлен");
        }
        else
            throw new Exception("Инвентраь переполнен");

    }
    public void add_func(Artefact art)
    {
        if (art.IsRenewable)
        {
            check_add(art);
        }
        else
        {

            for(int i = 0; i < 5; i++)
            {

                if (container[i]!=null && container[i].name_ == art.name_)
                {

                    container[i].count_of_bottles++;
                    return;
                }
```

```
            }
            check_add(art);
        }

    }

    public void throw_item(Artefact art)
    {
        Debug.Log("Наш артефакт " + art.name_ + " id= " + art.id_);
        for (int i = 0; i < 5; i++)
        {
            if (container[i] != null)
              Debug.Log("Сравниваем с  " + container[i].name_+" id=
"+container[i].id_);
            if (container[i]!=null && container[i].id_ == art.id_)
            {
                Debug.Log("Выбросисли " + art.name_);
                remove_item(i);
                return;
            }
        }

    }
    private void remove_item(int i)
    {
        container[i] = null;
        count--;

    }
    public void UseArtefact(int id, Character target)
    {
        Debug.Log("Ищем в рюкзаке " + id);
        for (int i = 0; i < 5; i++)
        {
            if (container[i] != null)
            {
                Debug.Log("Проверяем " + container[i].name_ + " id= " +
container[i].id_);
            }
               if (container[i]!=null &&  container[i].id_ == id)
            {
                Debug.Log("нашли!");
                container[i].SkillEffect(target);
                if (!container[i].IsRenewable && container[i].count_of_bottles < 2)
                {Debug.Log("Закончился! -выбрасываем");
                    throw_item(container[i]);

                }
                else
                {
                    container[i].count_of_bottles--;
                }
                return;
            }


        }
        Debug.Log(" не нашли...");
    }
```

3.2. Создание сценария ролевой игры на Unity 2D

Постановка задачи:

Разработать сценарий ролевой игры, демонстрирующий работу игрового движка.

Сценарий:

Игрок появляется в своем доме, который находится в лесу, переполненном грибами. Цель игры: построить 3 здания, собирая грибы, а затем продавая их для того чтобы купить необходимые материалы для постройки зданий.

Скриншоты

4. Изучение HTML, CSS, Java Script. Создание веб-сайтов.

Цель: изучить связку HTML + CSS + JS для создания веб-сайтов.

Постановка задачи:

Создать информативный веб-сайт на тему «Стройматериалы».

Ссылка: https://daniilnikonchik.wixsite.com/buildingmaterials

Интернет-магазин стройматериалов

Скриншоты:

Доска пола 21х90х1500 мм, хвоя, сорт АВ

3,55р.

Доска пола 21х90 длина 2 м, сорт I, материал Сосна

4,57р.

Бетоносмеситель СМ-160

557,00р.

Бетономешалка СМ-120

447,00р.

Дверь металлическая Промет Практик-504 левая 2066х980 мм беленый дуб

Дверь Новосёл-2050х950 мм R

244,99р.

Черепица битумная TILERCAT ПРИМА коричневая, 3 м2

Металлочерепица Germania 30 Simetric MAT AM 1215х720 мм черная2.256155

СВЯЖИТЕСЬ С ПОДДЕРЖКОЙ

Имя | Фамилия

Эл. почта | Телефон

Выберите проблему

Расскажите, чем мы можем вам помочь...

Связаться

ПОДРОБНЕЕ О НАС

Преданность клиентам

Интернет-магазин «Стройматериалы» предлагает большой ассортимент товаров для ремонта и обустройства домов и садовых участков, регулярно пополняет и обновляет его с учетом интересов покупателей. Благодаря круглосуточной службе поддержки ваш заказ будет оформлен без задержек и доставлен в срок. У нас есть все необходимое для любых задач, будь то капитальный ремонт, небольшое обновление или весенние работы в саду.

Приложение

В качестве приложения предлагается реферат на тему «Тестирование путем покрытия логики программы (“белый ящик”, “серый ящик”)».

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет прикладной математики и информатики
Кафедра информационных систем управления

Реферат по теме
«Тестирование путем покрытия логики программы (“белый ящик”, “серый ящик”)»

Выполнил студент группы № 13
Никончик Даниил Викторович

Оглавление

Введение

В данной работе рассматриваются особенности подхода к обеспечению качества программного обеспечения различными методами и видами тестирования. Определяются философия тестирования, принципы и фазы процесса тестирования. Рассматриваются различные виды ошибок и методы их обнаружения, стратегии тестирования на примере тестирование путём покрытия логики программы («белый, чёрный и серый ящики»). Определяются достоинства и недостатки этих методов на наглядном примере из жизни.

1. Философия тестирования.

1.1. Определение термина «тестирование»

Тестирование как объект изучения может рассматриваться с различных чисто технических точек зрения. Однако наиболее важными при изучении тестирования представляются вопросы его экономики и психологии разработчика. Иными словами, достоверность тестирования программы в первую очередь определяется тем, кто будет ее тестировать и каков его образ мышления, и уже затем определенными технологическими аспектами. Поэтому, прежде чем перейти к техническим проблемам, мы остановимся на этих вопросах. Вопросы экономики и психологии до сих пор тщательно не исследованы. Однако, необходимо разобраться в общих моментах экономики и тестирования. Поначалу может показаться тривиальным жизненно важный вопрос определения термина «тестирование». Необходимость обсуждения этого термина связана с тем, что большинство специалистов используют его неверно, а это в свою очередь приводит к плохому тестированию. Таковы, например, следующие определения:

«Тестирование представляет собой процесс, демонстрирующий отсутствие ошибок в программе»,

«Цель тестирования – показать, что программа корректно исполняет предусмотренные функции»,

«Тестирование – это процесс, позволяющий убедиться в том, что программа выполняет свое назначение».

Эти определения описывают нечто противоположное тому, что следует понимать под тестированием, поэтому они неверны. Оставив на время определения, предположим, что если мы тестируем программу, то нам нужно добавить к ней некоторую новую стоимость (так как тестирование стоит денег и нам желательно возвратить затраченную сумму, а это можно сделать только путем увеличения стоимости программы). Увеличение стоимости означает повышение качества или возрастание надежности программы, в противном случае пользователь будет недоволен платой за качество. Повышение качества или надежности программы связано с обнаружением и удалением из нее ошибок. Следовательно, программа тестируется не для того, чтобы показать, что она работает, а скорее наоборот – тестирование начинается с предположения, что в ней есть ошибки (это предположение справедливо практически для любой программы), а затем уже обнаруживается их максимально возможное число. Таким образом, сформулируем наиболее приемлемое и простое определение:

«Тестирование – это процесс исполнения программы с целью обнаружения ошибок».

Дело в том, что верный выбор цели дает важный психологический эффект, поскольку для человеческого сознания характерна целевая направленность. Если поставить целью демонстрацию отсутствия ошибок, то мы подсознательно будем стремиться к этой цели, выбирая тестовые данные, на которых вероятность появления ошибки мала. В то же время, если нашей задачей станет обнаружение ошибок, то создаваемый нами тест будет обладать большей вероятностью обнаружения ошибки. Такой подход заметнее повысит качество программы, чем первый.

1.2. Принципы тестирования

Сформулируем основные принципы тестирования. Наиболее важными в тестировании программ являются вопросы психологии. Эти принципы интересны тем, что в основном они интуитивно ясны, но в то же время на них часто не обращают должного внимания. Описание предполагаемых значений выходных данных или результатов должно быть необходимой частью тестового набора. Нарушение этого очевидного принципа представляет одну из наиболее распространенных ошибок. Ошибочные, но правдоподобные результаты могут быть признаны правильными, если результаты теста не были заранее определены. Здесь мы сталкиваемся с явлением психологии: мы видим то, что мы хотим увидеть. Другими словами, несмотря на то что тестирование

по определению – деструктивный процесс, есть подсознательное желание видеть корректный результат. Один из способов борьбы с этим состоит в поощрении детального анализа выходных переменных заранее при разработке теста. Поэтому тест должен включать две компоненты: описание входных данных и описание точного и корректного результата, соответствующего набору входных данных. Необходимость этого подчеркивал логик Копи в работе: «Проблема может быть охарактеризована как факт или группа фактов, которые не имеют приемлемого объяснения, которые кажутся необычными или которые не удается подогнать под наши представления или предположения. Очевидно, что если что-нибудь подвергается сомнению, то об этом должна иметься какая-то предварительная информация. Если нет предположений, то не может быть и неожиданных результатов». Следует избегать тестирования программы ее автором. К сожалению, реализация этого в целом верного принципа не всегда возможна в силу трех факторов:

1) людские ресурсы разработки, как правило, недостаточны;

2) для регулярного применения этого принципа к каждой программе требуется весьма высокая квалификация всех программистов или большой группы программистов, тестирующих все программы, что не всегда осуществимо;

3) необходим высокий уровень формализации ведения разработки; тщательные формализованные спецификации требований к программам и данным, тщательное описание интерфейса и формализация ответственности за качество продукта.

В настоящее время проводится значительная работа по созданию и внедрению формализованных методов в большинстве крупных разработок, но опыт подобного ведения разработок пока еще недостаточно массовый.

1.3. Фазы тестирования

В процессе тестирования выделяют следующие фазы:

1. **Определение целей** (требований к тестированию), включающее следующую конкретизацию: какие части системы будут тестироваться, какие аспекты их работы будут выбраны для проверки, каково желаемое качество и т.п.

2. **Планирование**: создание графика (расписания) разработки тестов для каждой тестируемой подсистемы; оценка необходимых человеческих, программных и аппаратных ресурсов; разработка расписания тестовых

циклов. Важно отметить, что расписание тестирования обязательно должно быть согласовано с расписанием разработки создаваемой системы, поскольку наличие исполняемой версии разрабатываемой системы (**Implementation Under Testing (IUT)** или **Application Under Testing (AUT)** – часто употребляемые обозначения для тестируемой системы) является одним из необходимых условий тестирования, что создает взаимозависимость в работе команд тестировщиков и разработчиков.

3. **Разработка тестов**, то есть тестового кода для тестируемой системы, если необходимо - кода системы автоматизации тестирования и тестовых процедур (выполняемых вручную).
4. **Выполнение тестов**: реализация тестовых циклов.
5. **Анализ результатов**.

После анализа результатов возможно повторение процесса тестирования, начиная с пунктов 3, 2 или даже 1.

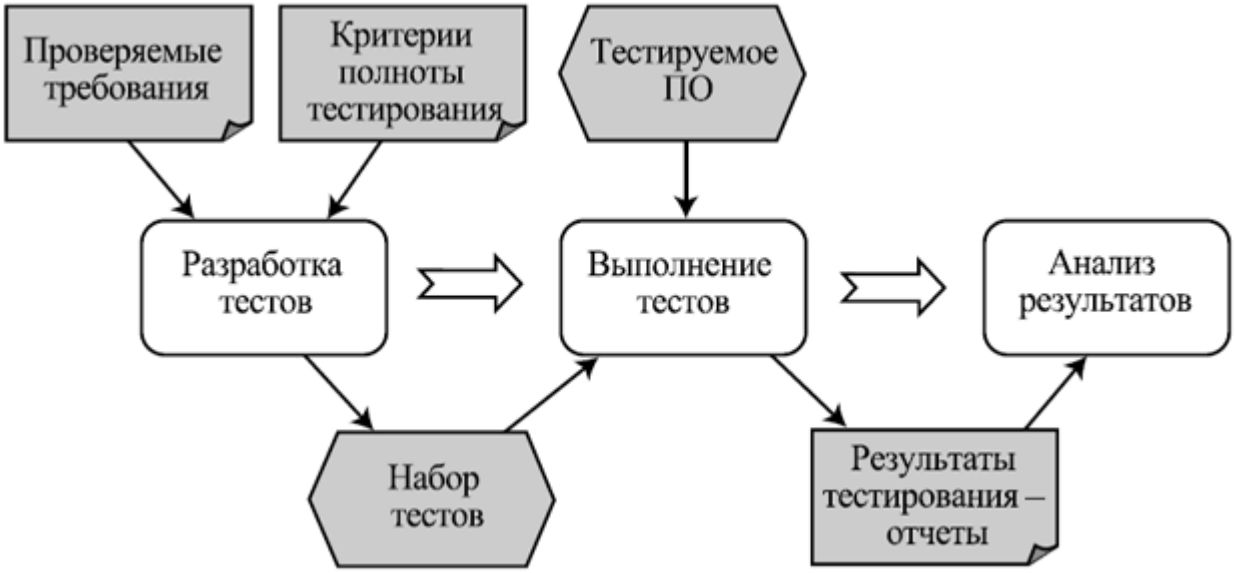


Рисунок 1. Процесс тестирования ПО

2. Типы ошибок и ручные методы тестирования

Задача любого тестировщика заключается в нахождении наибольшего количества ошибок, поэтому он должен хорошо знать наиболее часто допускаемые ошибки и уметь находить их за минимально короткий период времени. Остальные ошибки, которые не являются типовыми, обнаруживаются только тщательно созданными наборами тестов. Однако, из этого не следует, что для типовых ошибок не нужно составлять тесты. Далее будет дана классификация ошибок, что поможет сосредоточить наши усилия в правильном направлении.

2.1. Классификация ошибок

Для классификации ошибок мы должны определить термин «ошибка».

«Ошибка – это расхождение между вычисленным, наблюдаемым и истинным, заданным или теоретически правильным значением». [2]

Такое определение понятия «ошибка» не является универсальным, так как оно больше подходит для понятия «программная ошибка». В технологии программирования существуют не только программные ошибки, но и ошибки, связанные с созданием программного продукта, например, ошибки в документации программы. Но нас пока будут интересовать программные ошибки. Итак, по времени появления ошибки можно разделить на три вида:

• структурные ошибки набора;

• ошибки компиляции;

• ошибки периода выполнения.

Структурные ошибки возникают непосредственно при наборе программы. Что это за ошибки? Если кто-то работал в среде разработки Microsoft Visual Basic, то он знает, что если набрать оператор If, затем сравнение и нажать на клавишу Enter, не набрав слова Then, то Visual Basic укажет, что возникла ошибка компиляции. Это не совсем верно, так как компиляция в Visual Basic происходит только непосредственно при выполнении команды программы. В данном случае мы имеем дело именно со структурной ошибкой набора. Данный тип ошибок определяется либо при

наборе программы (самой IDE (Integrated Development Environment) – интегрированной средой разработки) или при ее компиляции, если среда не разделяет первые два типа ошибок. К данному типу ошибок относятся такие как: несоответствие числа открывающих скобок числу закрывающих, отсутствие парного оператора (например, try без catch), неправильное употребление синтаксических знаков и т. п. Во многих средах разработки программного обеспечения данный тип ошибок объединяется со следующим типом, так как раннее определение ошибок вызывает некоторое неудобство при наборе программ (скажем, я задумал что-то написать, а потом вспомнил, что в начале пропустил оператор, тогда среда разработки может выдать мне ошибку при попытке перейти в другую строку). Еще раз нужно отметить, что данный тип ошибок достаточно уникален и выделяется в отдельный тип только некоторыми средами разработки программного обеспечения. Ошибки компиляции возникают из-за ошибок в тексте кода. Они включают ошибки в синтаксисе, неверное использование конструкций языка (оператор else в операторе for и т. п.), использование несуществующих объектов или свойств, методов у объектов. Среда разработки (компилятор) обнаружит эти ошибки при общей компиляции приложения и сообщит о последствиях этих ошибок. Необходимо подчеркнуть слово «последствия» – это очень важно. Дело в том, что часто, говоря об ошибках, мы не разделяем проявление ошибки и саму ошибку, хотя это и не одно и то же. Например, ошибка «неопределенный класс» не означает, что класс не определен. Он может быть неподключенным, так как не подключен пакет классов. Ошибки периода выполнения возникают, когда программа выполняется и компилятор (или операционная система, виртуальная машина) обнаруживает, что оператор делает попытку выполнить недопустимое или невозможное действие. Например, деление на ноль. Предположим, имеется такое выражение:

```
ratio = firstValue / sum.
```

Если переменная sum содержит ноль, то деление – недопустимая операция, хотя сам оператор синтаксически правилен. Прежде, чем программа обнаружит эту ошибку, ее необходимо запустить на выполнение. Хотя данный тип ошибок называется «ошибками периода выполнения», это не означает, что ошибки находятся только после запуска программы. Вы можете выполнять программу в уме и обнаружить ошибки данного типа, однако, понятно, что это крайне неэффективно. Если проанализировать все типы ошибок согласно первой классификации, то можно прийти к заключению, что при тестировании приходится иметь дело с ошибками периода выполнения, так как первые два типа ошибок определяются на этапе кодирования. В теоретической

информатике программные ошибки классифицируют по степени нарушения логики на:

• синтаксические;

• семантические;

• прагматические.

Синтаксические ошибки заключаются в нарушении правописания или пунктуации в записи выражений, операторов и т. п., т. е. в нарушении грамматических правил языка. В качестве примеров синтаксических ошибок можно назвать:

• пропуск необходимого знака пунктуации;

• несогласованность скобок;

• пропуск нужных скобок;

• неверное написание зарезервированных слов;

• отсутствие описания массива.

Все ошибки данного типа обнаруживаются компилятором. Семантические ошибки заключаются в нарушении порядка операторов, параметров функций и употреблении выражений. Например, параметры у функции add (на языке Java) в следующем выражении указаны в неправильном порядке: GregorianCalendar.add(1, Calendar.MONTH). Параметр, указывающий изменяемое поле (в примере – месяц), должен идти первым. Семантические ошибки также обнаруживаются компилятором. Надо отметить, что некоторые исследователи относят семантические ошибки к следующей группе ошибок. Прагматические ошибки (или логические) заключаются в неправильной логике алгоритма, нарушении смысла вычислений и т. п. Они являются самыми сложными и крайне трудно обнаруживаются. Компилятор может выявить только следствие прагматической ошибки (см. выше пример с делением на ноль, компилятор обнаружит деление на ноль, но, когда и почему переменная sum стала равна нулю – должен найти программист). Таким образом, после рассмотрения двух классификаций ошибок можно прийти к выводу, что на этапе тестирования ищутся прагматические ошибки периода выполнения, так как остальные выявляются в процессе программирования. На этом можно было бы закончить рассмотрение классификаций, но с течением времени накапливался опыт обнаружения ошибок и сами ошибки, некоторые,

из которых образуют характерные группы, которые могут тоже служить характерной классификацией.

Ошибка адресации – ошибка, состоящая в неправильной адресации данных (например, выход за пределы участка памяти).

Ошибка ввода-вывода – ошибка, возникающая в процессе обмена данными между устройствами памяти, внешними устройствами.

Ошибка вычисления – ошибка, возникающая при выполнении арифметических операций (например, разнотипные данные, деление на нуль и др.).

Ошибка интерфейса – программная ошибка, вызванная несовпадением характеристик фактических и формальных параметров (как правило, семантическая ошибка периода компиляции, но может быть и логической ошибкой периода выполнения).

Ошибка обращения к данным – ошибка, возникающая при обращении программы к данным (например, выход индекса за пределы массива, не инициализированные значения переменных и др.).

Ошибка описания данных – ошибка, допущенная в ходе описания данных.

2.2. Первичное выявление ошибок

В течение многих лет большинство программистов убеждено в том, что программы пишутся исключительно для выполнения их на машине и не предназначены для чтения человеком, а единственным способом тестирования программы является ее исполнение на ЭВМ. Это мнение стало изменяться в начале 70-х годов в значительной степени благодаря книге Вейнберга «Психология программирования для ЭВМ». [3] Вейнберг показал, что программы должны быть удобочитаемыми и что их просмотр должен быть эффективным процессом обнаружения ошибок. По этой причине, прежде чем перейти к обсуждению традиционных методов тестирования, основанных на применении ЭВМ, рассмотрим процесс тестирования без применения ЭВМ («ручное тестирование»), являющийся по сути первичным обнаружением ошибок. Эксперименты показали, что методы ручного тестирования достаточно эффективны с точки зрения нахождения ошибок,

так что один или несколько из них должны использоваться в каждом программном проекте. Описанные здесь методы предназначены для периода разработки, когда программа закодирована, но тестирование на ЭВМ еще не началось. Аналогичные методы могут быть получены и применены на более ранних этапах процесса создания программ (т. е. в конце каждого этапа проектирования). Следует заметить, что из-за неформальной природы методов ручного тестирования (неформальной с точки зрения других, более формальных методов, таких, как математическое доказательство корректности программ) первой реакцией часто является скептицизм, ощущение того, что простые и неформальные методы не могут быть полезными. Однако их использование показало, что они не «уводят в сторону». Скорее эти методы способствуют существенному увеличению производительности и повышению надежности программы. Во-первых, они обычно позволяют раньше обнаружить ошибки, уменьшить стоимость исправления последних и увеличить вероятность того, что корректировка произведена правильно. Во- вторых, психология программистов, по-видимому, изменяется, когда начинается тестирование на ЭВМ. Возрастает внутреннее напряжение и появляется тенденция «исправлять ошибки так быстро, как только это возможно». В результате программисты допускают больше промахов при корректировке ошибок, уже найденных во время тестирования на ЭВМ, чем при корректировке ошибок, найденных на более ранних этапах. Кроме того, скептицизм связан с тем, что это «первобытный метод». Да, сейчас стоимость машинного времени очень низка, а стоимость труда программиста, тестировщика высока и ряд руководителей пойдут на все, чтобы сократить расходы. Однако, есть другая сторона ручного тестирования – при тестировании за компьютером причины ошибок выявляются только в программе, а самая глубокая их причина – мышление программиста, как правило, не претерпевает изменений, при ручном же тестировании, программист глубоко анализирует свой код, попутно выявляя возможные пути его оптимизации, и изменяет собственный стиль мышления, повышая квалификацию. Таким образом, можно прийти к выводу, что ручное тестирование можно и нужно проводить на первичном этапе, особенно, если нет прессинга времени и бюджета.

2.3. Инспекции и сквозные просмотры

Инспекции исходного текста и сквозные просмотры являются основными методами ручного тестирования. Так как эти два метода имеют много общего, они рассматриваются здесь совместно. Инспекции и сквозные просмотры включают в себя чтение или визуальную проверку программы группой лиц. Эти методы развиты из идей Вейнберга. Оба метода предполагают некоторую подготовительную работу. Завершающим этапом является «обмен мнениями» – собрание, проводимое участниками проверки. Цель такого собрания – нахождение ошибок, но не их устранение (т. е. тестирование, а не отладка). Инспекции и сквозные просмотры широко практикуются в настоящее время, но причины их успеха до сих пор еще недостаточно выяснены. Заметим, что данный процесс выполняется группой лиц (оптимально три-четыре человека), лишь один из которых является автором программы. Следовательно, программа, по существу, тестируется не автором, а другими людьми, которые руководствуются изложенными ранее принципами (в разделе 1), обычно не эффективными при тестировании собственной программы. Фактически «инспекция» и «сквозной просмотр» – просто новые названия старого метода «проверки за столом» (состоящего в том, что программист просматривает свою программу перед ее тестированием), однако они гораздо более эффективны опять-таки по той же причине: в процессе участвует не только автор программы, но и другие лица. Результатом использования этих методов является, обычно, точное определение природы ошибок. Кроме того, с помощью данных методов обнаруживают группы ошибок, что позволяет в дальнейшем корректировать сразу несколько ошибок. С другой стороны, при тестировании на ЭВМ обычно выявляют только симптомы ошибок (например, программа не закончилась или напечатала бессмысленный результат), а сами они определяются поодиночке.

3. Стратегии тестирования

«Отлаженная программа – это программа, для которой пока еще не найдены такие условия, в которых она окажется неработоспособной»

Каждый, кто сталкивается с тестированием, первое что слышит – это метод черного, серого и белого ящика. И хотя их общая идея проста как все гениальное, но то, что на самом деле это не два метода, а классы методов или стратегии, удивляет даже специалистов. В данной главе будут рассмотрены классические методы, которые относятся к стратегиям чёрного, белого и серого ящика. Это методы, которые предназначены для тестирования не программного комплекса в целом, а для тестирования, прежде всего, программного кода. Понимание данных методов позволит вам оценивать остальные методы с точки зрения полноты тестирования и подхода к тестированию.

Хотя перечисленные методы будут рассматриваться здесь по отдельности, при проектировании эффективного теста программы рекомендуется использовать если не все эти методы, то, по крайней мере, большинство из них, так как каждый метод имеет определенные достоинства и недостатки (например, возможность обнаруживать и пропускать различные типы ошибок). Правда, эти методы весьма трудоемки, поэтому некоторые специалисты, ознакомившись с ними, могут не согласиться с данной рекомендацией. Однако следует представлять себе, что тестирование программы – чрезвычайно сложная задача. Для иллюстрации этого приведу известное изречение:

«Если вы думаете, что разработка и кодирование программы – вещь трудная, то вы еще ничего не видели». Г. Майерс

Рекомендуемая процедура заключается в том, чтобы разрабатывать тесты, используя стратегию черного ящика, а затем как необходимое условие – дополнительные тесты, используя методы белого ящика.

3.1. Тестирование путём покрытия логики программы

3.1.1. Проверка «чёрного ящика»

Проверка «черного ящика» – это метод тестирования программного обеспечения, при котором функциональность исследуется без рассмотрения кода, деталей реализации и знаний о внутреннем устройстве программного обеспечения (ПО). Тестировщики пишут тест-кейсы, опираясь только на требования и спецификацию программного обеспечения.

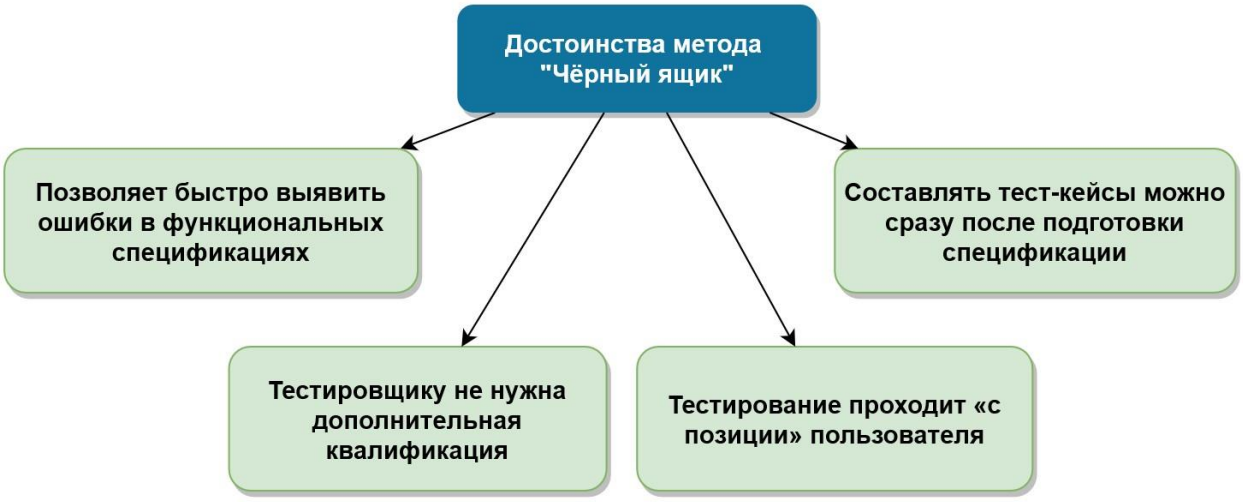


Рисунок 2. Достоинства метода "чёрный ящик"

Метод имитирует поведение пользователя, у которого нет никаких знаний о внутреннем устройстве программы. Методом «черного ящика» проводятся следующие виды тестирования:

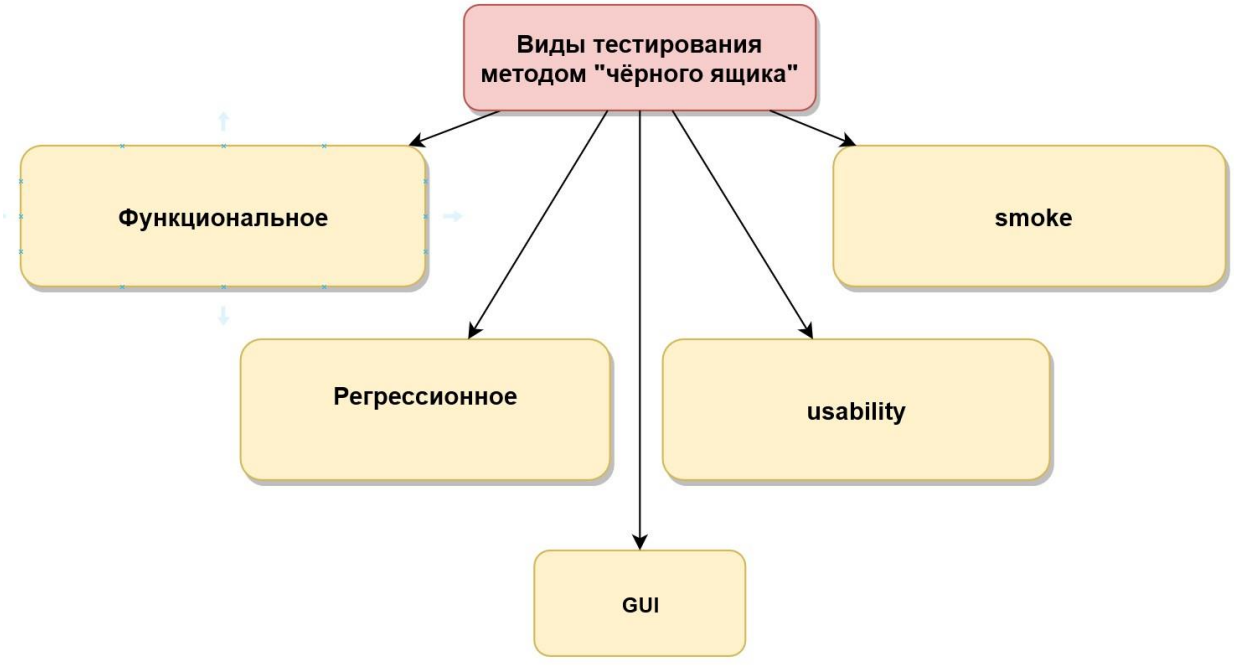


Рисунок 3. Виды тестирования методом "чёрного ящика"

К сожалению, использование этого метода далеко не всегда является достаточным при тестировании, так как существует высокая вероятность пропуска ошибки. Рассмотрим пример из практики.

Мы занимались тестированием формы регистрации и оплаты для VPN-провайдера. При регистрации клиенту предлагался на выбор набор тарифных планов и дополнительных услуг. После выбора и оплаты регистрация завершалась, и клиент попадал в свой личный кабинет. Мы проверили эту процедуру вдоль и поперек: все работало так, как нужно, ровно до того чудесного дня, когда было принято решение ввести новый промоплан для привлечения клиентов. Первая акция такого рода прошла весьма успешно: при регистрации по промоплану клиенту начислялся бонус на счет, и давался бесплатный доступ на 30 дней к одному дружественному сервису.

Вторая промо-акция отличалась тем, что клиенту при регистрации предлагался на выбор один из трех дружественных сервисов для бесплатного доступа. И тут что-то пошло не так: всем новым клиентам отправлялся доступ только к дружественному сервису из первой акции. Мы получили волну возмущения в саппорте и отток клиентов.
Ошибка заключалась в том, что первая промо-акция была учтена в базе под названием promo_1, а вторая – под promo_12, promo_13 и promo_14, но при этом в базу все записывалось под именем promo_1. Данные промо-акции не внесли в спецификацию, поэтому тест-кейсы не были составлены для новых акций. У тестировщиков не было доступа в базу, и они не могли проверить правильность записи о тарифном плане.

3.1.2. Проверка «белого ящика»

У этого метода существует несколько названий («стеклянный ящик», «открытый ящик» и др.), но чаще всего его все-таки именуют методом «белого ящика». Проверка «белого ящика» – это метод тестирования программного обеспечения, который предполагает, что внутренняя структура, устройство и реализация системы известны тестировщику.

Тестирование в «белом ящике» включает в себя несколько типов тестирования, применяемых для оценки удобства использования приложения, блока кода или конкретного программного пакета:

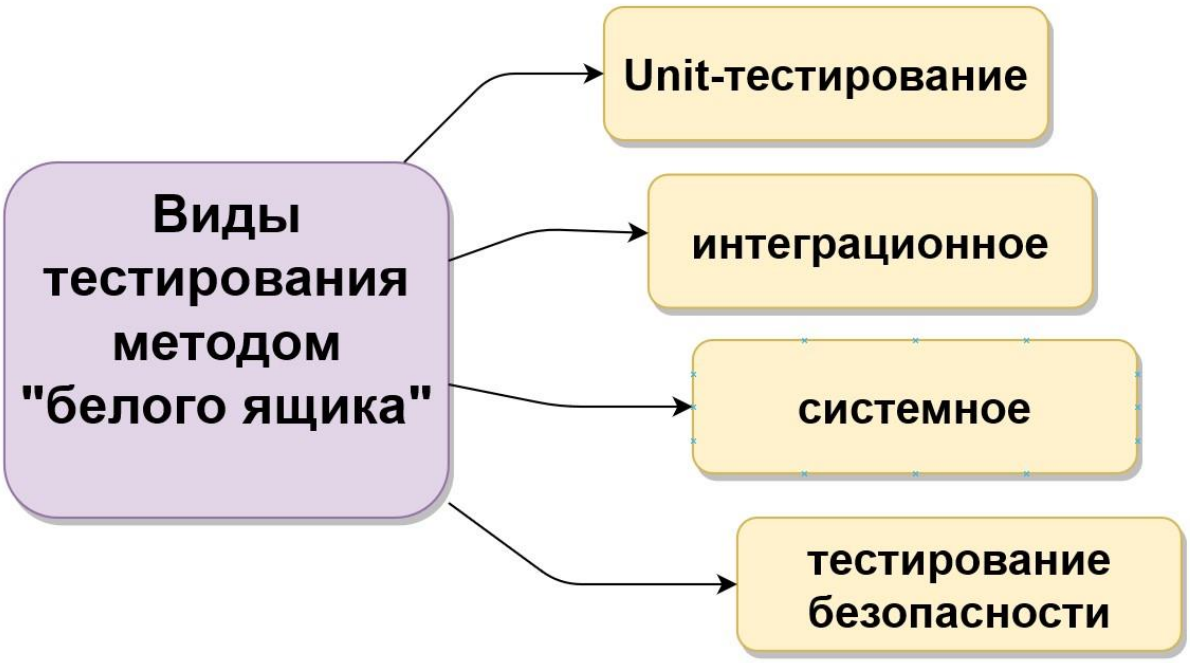


Рисунок 4. Виды тестирования методом "белого ящика"

Как правило, таким видом тестирования на проектах занимаются сами программисты, ведь для использования этого метода тестировщик должен обладать достаточно высокой квалификацией.

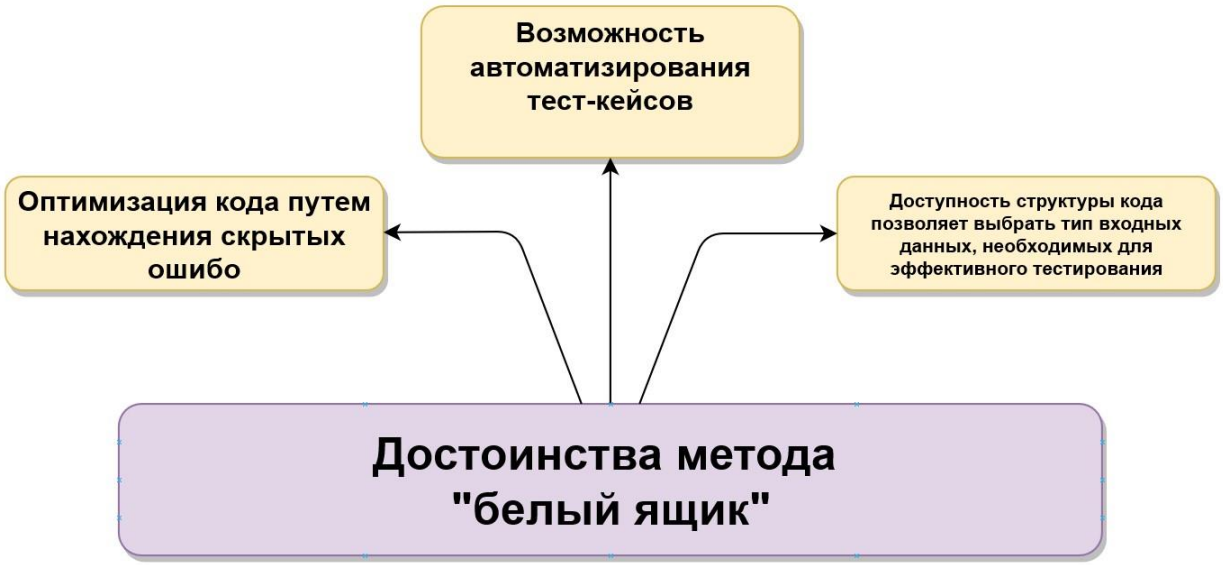


Рисунок 5. Достоинства метода "белый ящик"

Степень сложности тестирования методом «белого ящика» зависит от сложности вашего приложения/сервиса и от количества функций, которые оно выполняет.

Вернемся к нашему примеру. На входе мы имеем название подписки, на выходе – информацию по ней. Обычно список подписок хранится в базе данных, подписки могут добавляться в произвольные моменты времени. Black-box

тестирование просто не сможет обеспечить стопроцентное покрытие, ведь с точки зрения этого метода набор тестов устареет в момент добавления новой подписки в базу данных. В данном случае white-box тестирование имеет неоспоримое преимущество в виде прямого доступа к информации из базы данных. Наш набор тестов может загрузить список всех имеющихся подписок из базы данных и проверить, выдает ли контроллер в backend-е информацию о подписке для всех элементов списка.

3.1.3. Проверка «серого ящика»

Проверка «серого ящика» – это метод тестирования программного продукта или приложения с частичным знанием его внутреннего устройства. Для выполнения тестирования «серого ящика» нет необходимости в доступе тестировщика к исходному коду. Тесты пишутся на основе знания алгоритма, архитектуры, внутренних состояний или других высокоуровневых описаний поведения программы.

Рисунок 4. Виды тестирования методом "серого ящика"

Достоинства метода:

- Тестирование серого ящика включает в себя плюсы тестирования «черного» и «белого». Другими словами, тестировщик смотрит на объект тестирования с позиции «черного» ящика, но при этом проводит анализ на основе тех данных, что он знает о системе.
- Тестировщик может проектировать и использовать более сложные сценарии тестирования.
- Тестировщик работает совместно с разработчиком, что позволяет на начальном этапе убрать избыточные тест-кейсы. Это сокращает время

функционального и нефункционального тестирования и положительно влияет на общее качество продукта.
- Предоставляет разработчику достаточно времени для исправления дефектов.

Недостатки метода:

- Возможность анализа кода и тестового покрытия ограничена, так как доступ к исходному коду отсутствует.
- Тесты могут быть избыточными в том случае, когда разработчик также проверяет свой код Unit-тестами.
- Нельзя протестировать все возможные потоки ввода и вывода, поскольку на это требуется слишком много времени.

В нашем примере у каждого клиента мог быть набор дополнительных функций (capabilities):

- «can_vpn» – клиент мог подключиться к VPN;
- «can_double_vpn» – клиент получал возможность подключиться к VPN, используя функцию DoubleVPN;
- «can_port_forward» – клиент имел дополнительный порт для входящих подключений на стороне сервера;
- «can_promo1» – клиент имел доступ к дружественному сервису.

Для удобства проверки разработчики предусмотрели возможность тестировщикам читать набор разрешенных функций из таблицы capabilities для каждого клиента. Тестировщики ставили тарифный план (подписку) и проверяли правильность изменения флагов в этой таблице. Без использования методики «серого ящика» проверка возможности для клиента совершить VPN-соединение в сочетании с дополнительными функциями потребовала бы гораздо больших затрат времени и труда.

Заключение

Из представленной информации можно понять, что метод «серого ящика» помогает в следующих случаях:

- когда нет возможности использовать «белый ящик»;
- когда необходимо более полное покрытие по сравнению с «черным ящиком».

Используя этот метод, тестировщики получают доступ к проектной документации и могут подготовить и создать более точные и полные тест-кейсы и сценарии тестирования. Наибольшая эффективность применения

«серого ящика» достигается при тестировании web-приложений, web-сервисов, безопасности, GUI, а также для функционального тестирования.

Список используемой литературы

1. Особенности тестирования «серого ящика» [Сайт] – Режим доступа:https://quality-lab.ru/key-principles-of-gray-box-testing/
2. Першиков В. И., Савинков В. М. Толковый словарь по информатике. – М.: Финансы и статистика, 1991. – 543 с.
3. Weinberg G. M. The Psychology of Computer Programming. New York, Van Nostrand Reinhold, 1971.

Виды тестирования ПО [Сайт] – Режим доступа: http://wiki.rosalab.ru/ru/index.php/Виды_тестирования_ПО