Модуль 2. Объектно-ориентированное программирование

Тема 2.3. Приемы тестирования и отладки на примерах со строками

2 часа

Оглавление

2.3. Приемы тестирования и отладки на примерах со строками

2.3.1. Введение

Программы без ошибок можно написать двумя способами, но работает только третий.
Алан Перлис, американский учёный в области информатики

Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.

Брайан Керниган, создатель языка Си

Программы всегда содержали ошибки, содержат и будут их содержать. Как правило, пос-ледняя найденная ошибка на самом деле является предпос-ледней.

Отладка — это процесс определения и устранения причин ошибок. Чтобы понять, где возникла ошибка, приходится:

- узнавать текущие значения переменных;
- выяснять, по какому пути выполнялась программа.

Отладка включает поиск дефекта и его исправление, причем поиск дефекта и его понимание обычно составляют 90% работы. В некоторых проектах процесс отладки занимает до 50% общего времени разработки.

Существуют две взаимодополняющие технологии отладки.

1. Отладочный вывод и логирование (от англ. log — журнал событий, протокол).
2. Текущее состояние программы логируется с помощью расположенных в критических точках программы операторов вывода.
3. Использование отладчиков.

Отладчики позволяют пошагового выполнять программы оператор за оператором и отслеживать состояние переменных.

2.3.2. Строки

Прежде чем мы дальше рассмотрим приемы отладки рассмотрим подробнее тип данных String, которым мы уже не раз использовали ранее. Как уже понятно, String - это класс, а не примитивный тип данных.

Для создания строки можно использовать следующие конструкции:

```
String имя_строки="Hello World!"; //рекомендуемая форма
String имя_строки= new String("Hello World");
```

Например:

```
String hello1 = "Здравствуйте!";
String hello2 = new String("Здравствуйте!");
```

В результате выполнения приведенных строк кода в памяти создаются объекты класса String, каждый из которых хранит символы строки "Здравствуйте". Ссылки на объекты сохраняются в переменных hello1 и hello2.

В Java строки относятся к **неизменяемым объектам (англ. immutable)** - объектам, состояние которых не может быть изменено после создания. Т.е **символы строковой переменной после ее создания поменять нельзя!**

При работе с объектами String, если требуется получить другую строку, используя символы имеющейся строки, нужно создать новую строку. При необходимости можно присвоить прежней переменной сылку на новую строку в памяти, но поменять строку в уже существующем объекте String нельзя.

При необходимости менять символы именно имеющейся строки — используйте класс StringBuffer.

В Java неизменяемые объекты получили широкое распространение. Они позволяют обеспечить безопасность при многопоточном выполнении программ, что характерно для приложений под Android.

Основные методы и операции, которые можно применять к строковым переменным приведены в таблице:

Операция	Описание
`+`	Операция конкатенации — две строки сливаются в одну. К символам первого аргумента (того, что стоит до "плюса") приписываются справа символы второго аргумента (того, что стоит после "плюса"). Если при этом один из аргументов — не строка, а другой тип данных, он преобразуется в строковое представление.

`boolean equals(Object object)`	Метод сравнения двух строк — той строки, к которой применяется и строки, указанной в качестве параметра.
	Например:
	`String s1 = new String("Ваня учится в IT ШКОЛЕ SAMSUNG");`
	`String s2 = new String("Ваня учится в IT ШКОЛЕ SAMSUNG");`
	`System.out.println(s1.equals(s2));`
	`System.out.println(s1 == s2);`
	
	На экран будет выведено:
	`true`
	`false`
	Метод `equals` позволяет сравнивать содержимое строк, что не позволяет сделать операция ==.

`int length()`	Вычисляет длину строки в символах. Для предыдущего примера результат s1.length() будет равен 28.

`char charAt(int index)`	Выдает символ, находящийся в строке на позиции номер index. Номера символов считаются с нуля (как в массивах). Например, s1.charAt(3) будет равен 'я'.
`int compareTo(String anotherString)`	Сравнивает две строки по буквам с учетом регистра и языка. Возвращает целое число, показывающее, является ли эта строка больше (результат> 0), равный (результат = 0), или менее (результат <0) аргумент.

Мы ранее определили две формы инициализации строк:

1. `String имя_строки="содержимое строки";`
2. `String имя_строки= new String("содержимое строки");`

Давайте перепишем предыдущий пример на метод `equals()`, заменив 2 форму на первую:

```
String s1 = "Ваня учится в IT ШКОЛЕ SAMSUNG";
String s2 = "Ваня учится в IT ШКОЛЕ SAMSUNG";
System.out.println(s1.equals(s2));
System.out.println(s1 == s2);
```

Удивительно, но мы получим другой результат!

```
true
true
```

В чем дело? Почему казалось бы эквивалентная замена привела к другому результату?

Ответ кроется в том, что в Java первый способ определяет создание строковых литералов (англ. String literal) с использованием Строкового пула, а во втором случае каждый раз создается новый объект String.

Поэтому всякий раз, когда создаются строковые литералы, в Строковом пуле проверяется, существует ли такой же строковый литерал. Если он существует, то новый образец для этой строки в Строковом Пуле не создается и переменная получает ссылку на уже существующий строковый литерал.

В нашем примере переменным присвоены одинаковые строковые литералы и поэтому переменная s2 получила ссылку на литерал, который был создан для s1. И в результате операция s1 == s2 вернула true.

В связи со всем выше сказанным, рекомендуется в программах на Java использовать первую форму создания строк через строковые литералы.

2.3.3. Отладочный вывод и логирование

Данный метод требует включения в программу дополнительного отладочного вывода в узловых точках. Узловыми считают точки алгоритма, в которых основные переменные программы меняют свои значения.

Например, отладочный вывод следует предусмотреть до и после завершения цикла изменения некоторого массива значений (если отладочный вывод предусмотреть в цикле, то будет выведено слишком много значений, в которых, как правило, сложно разбираться). При этом предполагается, что, выполнив анализ выведенных значений, программист уточнит момент, когда были получены неправильные значения, и сможет сделать вывод о причине ошибки.

Создадим новый проект. В созданном проекте открываем файл **MainActivity.java**. В Android для логирования есть специальный класс android.util.Log. Чтобы добавить логирование, вставим вызовы метода Log.d() до и после setContentView(). Код теперь должен выглядеть так:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);

	Log.d(LOG_TAG, "Creating view..");
	setContentView(R.layout.activity_main);
	Log.d(LOG_TAG, "View created!");
```

Android Studio *будет подсвечивать Log.d красным цветом, т.к. класс Log не импортирован. Для быстрого импорта наведите на Log нажмите Alt + Enter.*

Eclipse *также будет подсвечивать Log.d красным цветом. Для быстрого импорта нажмите Ctrl + Shift + O. Т.к. Eclipse знает несколько вариантов класса Log, он спросит какой из них импортировать. Выберите android.util.Log. Кроме этого Eclipse будет подсвечивать слово LOG_TAG. Наведите курсор мыши на LOG_TAG и в открывшемся окне выберите Create constant 'LOG_TAG'*

Android создаст нам константу LOG_TAG со значением null. Заменим null на "MainActivity".

```
public class MainActivity extends Activity {

	private static final String LOG_TAG = "MainActivity";

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);

		Log.d(LOG_TAG, "Creating view..");
		setContentView(R.layout.activity_main);
		Log.d(LOG_TAG, "View created!");
```

Наш код выведет информацию о подгрузке макета для текущей Activity. Для просмотра этой информации нужно запустить проект и открыть вкладку с LogCat.

На вкладке LogCat видны наши сообщения, а также куча других сообщений, которые создаются различными модулями и программами. В данном примере по столбцу Time можно сделать вывод, сколько милисекунд заняла подгрузка макета.

Чтобы открыть вкладку с LogCat
В Android Studio *в нижней части окна выберите* ***Android*** *и откройте вкладку* ***LogCat***

В Eclipse *нужно выбрать в пункте меню* ***Eclipse Window ⇒Show View ⇒Other*** *и в папке Android выбрать LogCat*

Класс Log разбивает сообщения по категориям в зависимости от важности. Для этого используются специальные методы, которые легко запомнить по первым буквам, указывающим на категорию:

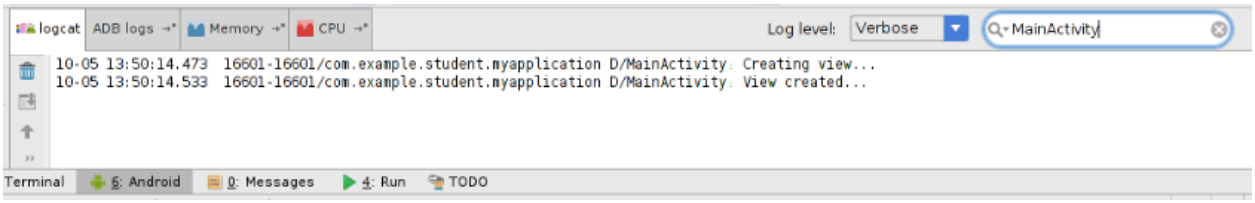
- Log.e() - ошибки (error)
- Log.w() - предупреждения (warning)
- Log.i() - информация (info)
- Log.d() - отладка (debug)
- Log.v() - подробности (verbose)
- Log.wtf() - очень серьезная ошибка! (What a Terrible Failure!, работает начиная с Android 2.2)

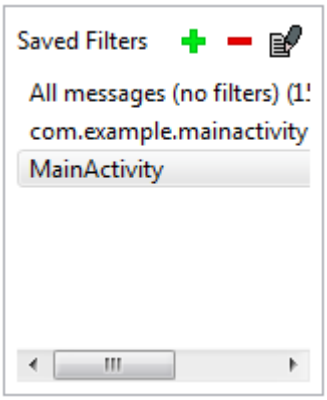
В первом параметре метода Log.d используется строка, называемая тегом. В качестве тега обычно задают имя класса, название библиотеки или название приложения. Обычно принято объявлять глобальную статическую строковую переменную и уже в любом месте вашей программы вы вызываете нужный метод записи в Log с этим тегом:

```
Log.d(LOG_TAG, "Сообщение для записи в журнал");
```

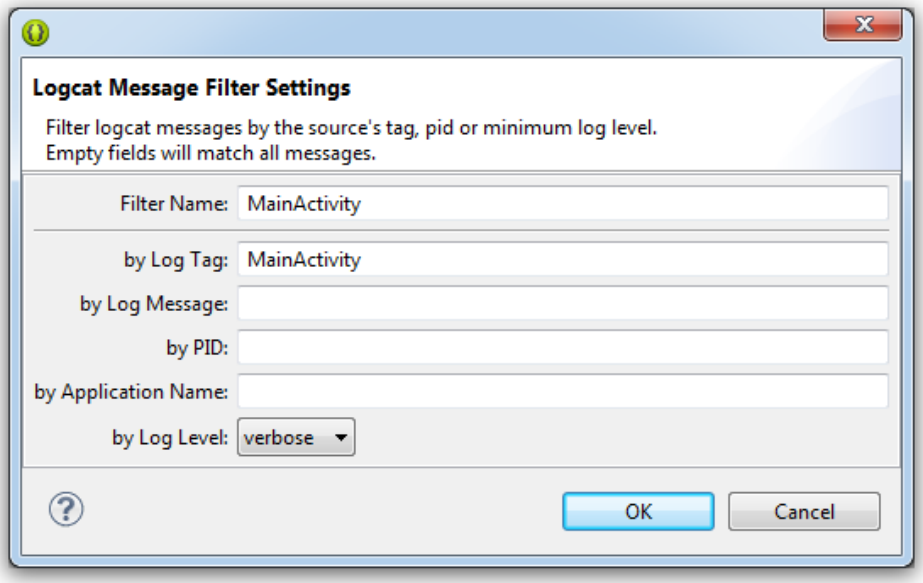
Данный тег мы можем применить для фильтрации сообщения в LogCat. Для этого нужно добавить фильтр по тегу.

В Android Studio нажмем в поле поиска и введем наш тег “Main_Activity”





В Eclipse в окне для фильтров лога нажимаем “+” и создаем фильтр MainActivity по тегу.



Теперь при выборе нашего фильтра будут отображаться только сообщения с тегом “MainActivity”.

Также мы можем отображать сообщения по уровням: VERBOSE, DEBUG, INFO, WARN, ERROR и ASSERT. Если выбрать уровень сообщений ERROR, то будут выводиться сообщения сгенерированные с уровнем ERROR и ASSERT. Если выбрать VERBOSE, то будут выводиться все сообщения.

Как правило, в серьезных приложениях в режиме тестирования постоянно логируется информация об обращении к сторонним сервисам API (application programming interface), обращении к базе данных, при возникновении нестандартных ситуаций и т.д. При выкладывании приложения в маркеты рекомендуется отключать всю отладочную информацию.

Задание 2.3.1

Добавьте логирование в другие методы класса, в частности onCreateOptionsMenu и onOptionsItemSelected и посмотрите когда происходят вызовы этих методов.

Существенный минус данного метода отладки - для получения информации нужно заранее в тексте программы проставить вызовы логирования с соответстующими данными. Но существует и другой метод отладки, позволяющий отлаживать программу на лету - использование отладчика.

2.3.4. Использование отладчика

С помощью встроенного отладчика Android Studio мы можем отлаживать программы на лету. Предположим мы написали программу, которая генерирует двумерный массив и переводит его в строку. Этот проект выложен в материалах и называется 2.3.LogCatExample (импортировать как Java проект).

```
public class MainActivity extends Activity {
	private static final String LOG_TAG = "MainActivity";
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_main);
		Log.v(LOG_TAG, "Matrix: \n" + matrixToString(createMatrix(5, 5)));
	}
	public int[][] createMatrix(int n, int m) {
		int[][] array = new int[n][m];
		for (int i = 0; i < array.length; i++) {
			for (int j = 0; j < array[i].length; j++) {
				array[i][j] = (int) (Math.random() * 10);
			}
		}
		return array;
	}
	public String matrixToString(int[][] array) {
		String result = "";
		for (int i = 0; i < array.length; i++) {
			for (int j = 0; j < array[i].length; j++) {
				result += array[i][j] + " ";
```

```
                    }
                    result += "\n";
                }
                return result;
        }
}
```

В методе onCreate установим точку останова (breakpoint). Для этого нужно поместить курсор на нужную строчку метода и щёлкнуть левой кнопкой мыши слева от номера строки.

Для работы в режиме отладки удобно пользоваться панелью управления отладкой. На ней расположены кнопки:

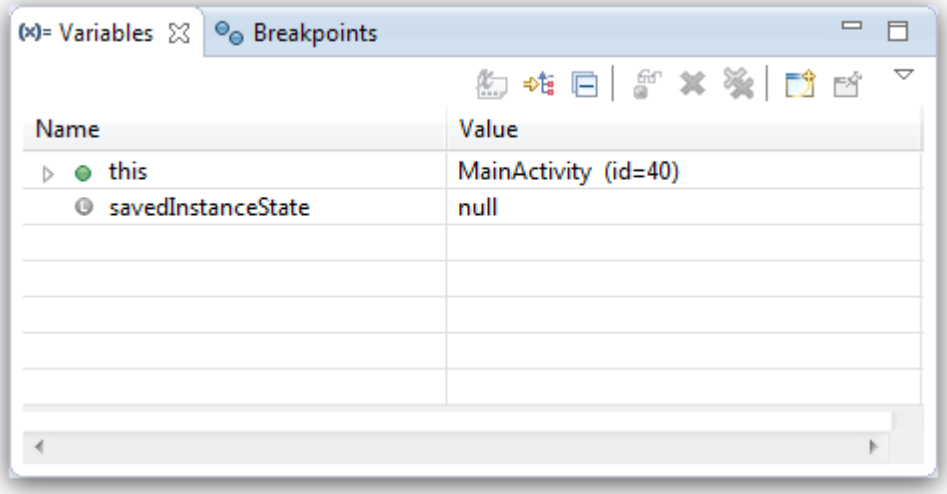
Android Studio	Eclipse	Наименование	Описание
F9	F8	Resume	Продолжить выполнение программы
		Suspend	Приостановка
Ctrl+F2		Terminate	Прекращение режима отладки
		Disconnect	Прекращение отладки
F7	F5	Step Into	Для захода внутрь метода
F8	F6	Step Over	Для перехода к следующей строки
Shift + F8	F7	Step Return	Для выхода из текущего метода

На строке кода, где установлен breakpoint, появится кружок. Теперь запустим выполнение в режиме отладки. Для этого нужно нажать на кнопку с жучком, либо выбрать из меню Run ⇒Debug ‘app’, либо нажать на клавиатуре Shift + F9 (в Eclipse F11). После запуска приложения Android Studio автоматически откроет вид отладки. Если этого не произошло, то выберите в меню View⇒ Tool Window ⇒Debug.

Перед вами откроется много разных окон, но мы остановимся на самых важных из них: окно с кодом программы и окно Variables.

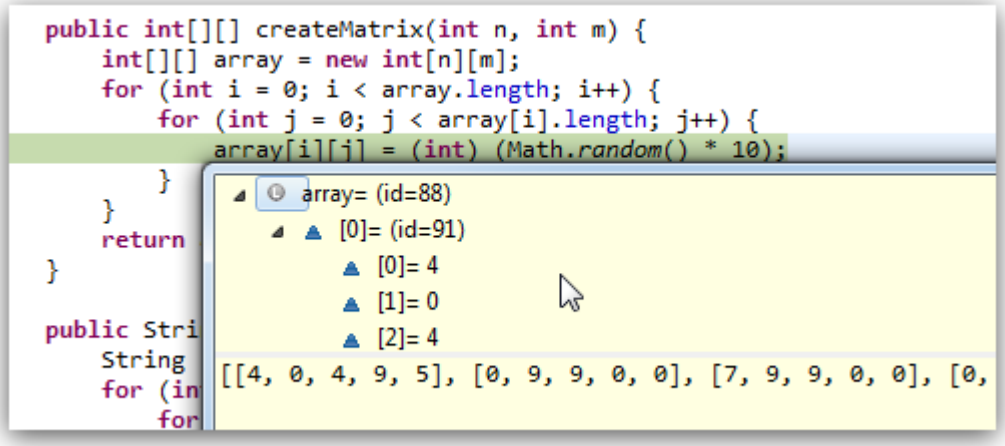
В окне кода одна из строчек подсвечена синим цветом. Так выделена текущая строчка, на которой приостановилось выполнение программы.

В окне Variables показаны переменные текущей области видимости:



Теперь добавим точки останова в начало методов `createMatrix` и `matrixToString` и запустим отладку (Resume Program). Выполнение программы остановится на строке с красной точкой в методе `createMatrix`. В окне Variables мы видим значение параметров m и n.

А теперь нажимая на Step Over будем выполнять программу шаг за шагом. Заметим, что в окне Variables появилась переменная array. По мере выполнения массив наполняется элементами. Если навести курсор на массив мы увидим значения его элементов.



Если нажать на Step Out выполнение программы остановится после выхода из текущего метода, т.е. к тому методу, который вызвал данный метод. В нашем случае выполнение программы вернулось в метод onCreate. Нажимаем на Resume Program и выполнение переходит в метод `matrixToString`. Здесь также можно пошагово выполнять команды, чтобы посмотреть как наполняется строковая переменная result.

Данный инструмент отладки отлично подходит для просмотра значений переменных и просмотра пути выполнения программы. Можно на ходу добавлять, убирать точки останова, задавать условия останова. Но данный метод не может полностью заменить метод вывода отладочной информации. Чаще всего в обычных ситуациях разработчики логируют важную информацию, а при возникновении необходимости отладки используют инструменты отладки.

2.3.6. Использование утверждений (assertions)

Утверждения (англ. Assertion, они же ассерты) — это код, используемый, как правило, только во время разработки, с помощью которого программа проверяет правильность своего выполнения.

Проверка утверждений во время выполнения программы предполагает выполнение произвольных (и, возможно, длительных) вычислений, которые могут серьёзно повлиять на производительность приложения. Одна из самых привлекательных особенностей механизма утверждений — это возможность отключения проверки утверждений в промышленной версии приложения. Таким образом, утверждения это простой и удобный механизм для поиска ошибок во время разработки, который не оказывает никакого влияния на работу готового продукта. Если утверждение ложное, то программа прекращает выполнение с выводом стека вызова, по которому можно определить в каком месте программы произошла ошибка..

По умолчанию в Java ассерты (утверждения) отключены. Чтобы их включить, нужно запускать JVM со специальным параметром. Для включения ассертов в локальной программе нужно зайти в **Run Configurations ⇒Arguments ⇒VM Arguments** и прописать там “-ea”.

Пример утверждения, проверяющего параметры функции:

```
// Считает факториал числа n.
// Число n должно лежать в пределах от 0 до 10 включительно.
int factorial(int n) {
	// Факториал отрицательного числа не считается
	assert n >= 0;

	// Если n превысит 12, то это может привести к целочисленному
	// переполнению результата.
	assert n <= 12;

	int result = 1;
	for (int i = 2; i <= n; i++) {
		result *= i;
	}

	return result;
}

// мы 'забыли' об ограничениях функции factorial()
// и пытаемся вычислить факториалы чисел от 0 до 99.
//
// проверка внутри factorial() любезно напомнит
// нам о своих ограничениях, так что мы сможем
// быстро выявить и исправить этот баг.
//
// если бы эта проверка отсутствовала,
// то баг мог бы долго оставаться незамеченным,
// периодически давая о себе знать целочисленными
// переполнениями и некорректным поведением программы.
for (int i = 0; i < 100; ++i) {
	a[i] = factorial(i);
}
```

Другой пример утверждения, проверяющего результат функции:

```
int factorial(int n) {
	int result = 1;

	for (int i = 2; i <= n; ++i) {
		result *= i;
		// С первого взгляда эта проверка никогда не сработает -
		// факториал должен быть всегда положительным числом.
		// Но как только n превысит допустимый предел,
		// произойдет целочисленное переполнение. В этом случае
		// переменная result может принять отрицательное
		// либо нулевое значение.
		//
		// После срабатывания этой проверки мы быстро
		// локализуем баг и поймем, что либо нужно ограничивать
		// значение n, либо использовать целочисленную
		// арифметику с произвольной точностью.
		assert (result > 0);
	}

	return result;
}
```

Еще один пример утверждений (проект называется LinesAssertSample, импортировать как Java проект):

```
import java.io.PrintStream;
import java.util.Scanner;

public class CrossLines {

	public static Scanner in = new Scanner(System.in);
	public static PrintStream out = System.out;

	public static void main(String[] args) {
		double k1, b1, k2, b2;
		k1 = in.nextDouble();
		b1 = in.nextDouble();
		k2 = in.nextDouble();
		b2 = in.nextDouble();
		double[] cross = crossLines(k1, b1, k2, b2);
		if (cross != null) {
			out.println("Пересекаются в (" + cross[0] + ", " + cross[1] +
")");
		} else {
			out.println("Не пересекаются");
		}
	}

	// Функция пересекает две прямые на плоскости,
	// заданных уравнением y = kx + b.
	// Если прямые не пересекаются или совпадают,
	// то возвращается null.
	public static double[] crossLines(double k1, double b1, double k2, double
b2) {
		double dk = k1 - k2;
		// если они параллельны, то возвращаем null
		if (dk == 0.0) {
			return null;
		}
```

```
            // формула получена из системы уравнений
            double x = (b2 - b1) / dk;
            double y = k1 * x + b1;
            // Проверка того, что полученная точка находится
            // на обоих прямых (проверка инвариантов).
            // Чтобы проверка работала необходимо добавить в
            // Run Configurations ⇒ Arguments ⇒ VM Arguments
            // строку -ea.
            assert onLine(k1, b1, x, y);
            assert onLine(k2, b2, x, y);
            return new double[]{x, y};
      }

      // Функция проверяет, что точка лежит на прямой,
      // заданной уравнением y = kx + b.
      private static boolean onLine(double k, double b, double x, double y) {
            return y == k * x + b;
      }
}
```

Если в данном коде поменять формулу вычисления точки пересечения на ложную, то получим:

```
Exception in thread "main" java.lang.AssertionError
      at CrossLines.crossLines(CrossLines.java:37)
      at CrossLines.main(CrossLines.java:16)
```

Хочется подчеркнуть, что т.к. assert'ы могут быть удалены на этапе компиляции либо во время исполнения программы, они не должны менять поведение программы. Если в результате удаления утверждения поведение программы может измениться, то это явный признак неправильного его использования. Таким образом, внутри assert'а нельзя вызывать функции, изменяющие состояние программы либо внешнего окружения программы. Иначе поведение программы при разработке со включенными ассертами и поведение готового продукта будут различаться.

Более подробно об утверждениях можно почитать в статье “Assert. Что это такое и с чем его едят?”:

http://dev.by/blogs/main/assert-chto-eto-takoe-i-s-chem-ego-edyat

О том, каким образом отключаются утверждения в готовых приложениях можно почитать на блоге компании Oracle:

https://blogs.oracle.com/vmrobot/entry/проверка_утверждений_assertions_и_условная

Резюмируя, хочется подчеркнуть, что ассерты нужны для программистов, для более раннего нахождения грубых ошибок в коде, для того, чтобы обратить внимание других программистов на обязательные предусловия/постусловия. Для ожидаемых ошибок (например: отсутствие файла, отсутствие сети, ошибка при установлении соединения, отсутствие определенного датчика на устройстве и т.д.) используется механизм исключений, который будет рассмотрен в следующих разделах курса.

2.3.7. Модульное тестирование

С усложнением программного проекта растет и потенциальное количество ошибок в нем. При этом увеличивается не только количество строк кода проекта, но и, к примеру, появляются новые разработчики, которые заменяют старых или работают совместно (каждый разработчик разрабатывает свою часть проекта). Для добавления нового функционала зачастую приходится переделывать старый код. Ошибки неизбежны.

В связи с всем выше сказанным в промышленной разработке программ используют модульное тестирование.

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 появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок. Цель модульного тестирования — исключить из поиска ошибок отдельные части программы путем их автоматической проверки на заранее написанных тестах.

Итак, из примера 2.3.4. у нас есть класс MainActivity. Также можете загрузить этот проект, он в материалах под именем 2.3.Testing. Давайте создадим Unit-тест для него. Для этого в Android Studio добавляем через gradle библиотеки junit 4.12. В Project выбираем app ⇒ Gradle Scripts ⇒ build.gradle. В dependencies добавляем testCompile 'junit:junit:4.12'

```
dependencies {
 testCompile 'junit:junit:4.12'
 }
```

Далее нажимаем Sync Now.

Открываем "Build variants" слева и выбираем в колонке Test Artifacts Unit Tests.

Далее создаем новую директорию и создаем там класс MainActivityTest. И создаем тесты для методов MainActivity. Любой метод с аннотацией “@Test” в JUnit считается отдельным тестом. Рекомендуется называть такие методы со слова “test”. Кроме этого, все методы тестирования обязательно являются public void. Если внутри теста хотя бы один assert вызвет ошибку, то тест будет считаться заваленным. Если все гладко, то тест пройден.

```
public class MainActivityTest {

       int testArray[][] = {{1,2},{1,2}};

    @Test
    public void testCreateMatrix(){
        assertArrayEquals( MainActivity.createMatrix(1, 1), new int[1][1]);
       }
    @Test
    public void testMatrixToString(){
        assertNotEquals( MainActivity.matrixToString(testArray),"1 2 1 2");
       }
```

Запускаем проект.

Информацию о доступных методах JUnit и практических методиках написания модульных тестов можно почитать здесь: http://javaxblog.ru/article/java-junit-1/

Широко известна оценка распределения трудозатрат: на отладку разработчик в среднем тратит в 4 раза больше времени, чем на само написание кода. Поэтому вполне объяснимо, почему в современном мире программирования Unit тесты стали очень популярны. Даже появилась новая методика разработки программ - разработка через тестирование (англ. test-driven development, TDD).,

При TDD разработчик до написания кода пишет тесты, отражающие требования к модулю. И только после этого пишет код, который должен успешно проходить эти тесты.

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.

При внесении изменений в код, который хорошо "покрыт тестами", риск появления новых ошибок значительно ниже. Если новая функциональность приводит к ошибкам, тесты сразу же это покажут. При работе с кодом, на который нет тестов, ошибку можно обнаружить спустя значительное время, когда исправить программу будет намного сложнее и дороже.

2.3.8. Другие виды тестирования

Разработка программных проектов не ограничивается тестированием отдельных компонент системы. После модульного тестирования следует интеграционное.

Интеграционное тестирование - это тестирование не отдельных компонентов системы, а результата их взаимодействия между собой в какой-либо среде.

Упор делается именно на тестирование взаимодействия. Т.к. интеграционное тестирование производится после модульного, то все все проблемы, обнаруженные в процессе объединения модулей, скорее всего связаны с особенностями их взаимодействия.

Вначале описывается план тестирования, подготавливаются тестовые данные, создаются и исполняются тест-кейсы (пошаговые действия для тестирования определенного функционала системы). Найденные ошибки исправляют и снова запускают тестирование. Цикл повторяется до тех пор, пока взаимодействие всех компонент не будет работать без ошибок.

Для автоматизации интеграционного тестирования применяются системы непрерывной интеграции (англ. Continuous Integration System, CIS). CIS проводит мониторинг исходных кодов. Как только разработчики выкладывают обновление кода выполняются различные проверки и модульные тесты. Далее проект компилируется и проходит интеграционное тестирование. Выявленные ошибки включаются в отчет тестирования.

Таким образом, автоматические интеграционные тесты выполняются сразу же после внесения изменений, что позволяет обнаруживать и устранять ошибки в короткие сроки.

Следующий этап тестирования - **системное тестирование**, которое разделяется на 2 этапа.

- Альфа-тестирование — имитация реальной работы с системой разработчиками, либо реальная работа с системой ограниченным кругом потенциальных пользователей.

- Бета-тестирование — в некоторых случаях выполняется распространение предварительной версии для некоторой большей группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Например, среда разработки Android Studio, разрабатываемая компанией Google, и, быстро набирающей популярность, с мая 2013 года находилась в альфа-тестировании, а в июне 2014 года перешла в стадию бета-тестирования.

Кроме тестирования по этапам, выделяют также классификацию по объектам тестирования:

- Юзабилити-тестирование.
- Тестирование удобства пользованием продукта.
- Тестирование безопасности.
- Оценка уязвимости программного обеспечения к различным атакам.
- Тестирование локализации.
- Проверка перевода пользовательского интерфейса, документации и сопутствующих файлов программного обеспечения на различных языках.
- Тестирование производительности.
- Проверка скорости работы системы под определённой нагрузкой. В тестировании производительности выделяют следующие направления:
    - Нагрузочное тестирование. Нагрузочное тестирование обычно проводится для того, чтобы оценить поведение приложения под заданной ожидаемой нагрузкой. Этой нагрузкой может быть, например, ожидаемое количество одновременно работающих пользователей приложения, совершающих заданное число действий за интервал времени.
    - Стресс-тестирование. Стресс-тестирование обычно используется для понимания пределов пропускной способности приложения. Этот тип тестирования проводится для определения надежности системы во время экстремальных нагрузок и отвечает на вопросы о достаточной производительности системы в случае, если текущая нагрузка сильно превысит ожидаемый максимум. Стресс-тестирование позволяет определить «узкие места» системы, которые вероятнее всего приведут к сбоям системы в пиковой ситуации. Проверка правильности работы программы осуществляется на большом количестве случайно сгенерированных данных. Мотивация данного тестирования - предпочтительнее быть готовым к обработке экстремальных условий системы, чем ожидать отказа системы, тем более когда стоимость отказа системы в экстремальных условиях может быть очень велика.

Для лучшей иллюстрации рассмотрим виды тестирования по аналогии с производством техники, например телефонов.

Завод закупает множество комплектующих - от винтиков до печатных плат. Очевидно, что качество готовой продукции напрямую зависит от любого из компонентов телефона. Поэтому контроль входного качества компонентов очень важен.

Все начинается с простейших тестов - веса и размера компонентов. Из партии деталей выбирают часть для тестов. Если это новый поставщик или деталь ранее не тестировалась, то проверка проходит самым тщательным образом - устойчивость к агрессивной среде, влажность, прочностные характеристики, рентген снимки на наличие скрытых дефектов и так далее. В результате тестирования фабрика решает будет ли она работать с даным поставщиком. Это все модульное тестирование.

После сборки телефона вставляется сим-карта и начинается проверка его работоспособности в целом: по заданным сценариям проверяются функции телефона. Это - интеграционное тестирование.

Аппараты могут проходить и более тщательную проверку:

- время работы телефона в различных режимах;
- деградация аккумулятора с количеством циклов разряда/заряда;
- работа процессора при максимальной нагрузке.

В тестовой лаборатории могут присутствовать помещения для проверки в условиях различных температур и влажности - имитация климатических зон.

Кроме этого, есть специальные камеры "старения" для получения реального эффекта старения за максимально короткий срок.

Механическая прочность экрана испытывается металлическими шариками, которые падают на него с определенной высоты. На других автоматах телефон поднимают на определенную высоту и роняют на металлическую поверхность.

Все описанное специальное оборудование применяется для нагрузочного и стресс-тестирования. Зачастую эти понятия считают синонимами, но это не верно. Нагрузочное тестирование - это, когда для проверки воссоздают предполагаемые нормальные условия эксплуатации. Когда же производится проверка в условиях сверхвысоких (выходящих за пределы обычного использования) нагрузок, то это уже будет стресс- тестирование.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателям IT ШКОЛЫ SAMSUNG Мансурову Руслану Маратовичу и Ахмадиеву Айнуру Альбертовичу