Давайте посмотрим попристальней на такое знакомое и (до боли?) родное слово БАГ. Происходит оно от английского слова Bug, означающего «насекомое». Есть еще много сторонних значений, в частности английское выражение «to go bugs» — сойти с ума, что легко коррелируется со вполне русским «тараканы в голове завелись». Также вспоминаются и «жучки на линии» (тоже, кстати, по-английски – bugs). И опять мы пришли к насекомым.  
  
Конечно, более распространена история о том, что в 1946 году разработка компьютера Марк-2 (Mark-II) были приостановлена из-за сбоя его функционирования, вызванного попаданием мотылька между контактов. Трупик мотылька был извлечен и приклеен к отчету липкой лентой с комментарием «First actual case of bug being found.» («Первый реальный случай нахождения жучка»). Как нетрудно догадаться, примерно оттуда же «растут уши» и слова «дебаггер» (debugger) – буквально «избавитель от жучков».

**Определение дефекта**

**Дефект** – это несоответствие между ожидаемым результатом (ОР) и фактическим результатом (ФР). Разберем это на примере:

|  |  |  |
| --- | --- | --- |
| **Действия** | **Ожидаемый результат** | **Фактический результат** |
| Ввести в ячейку выражение «=2+2\*2» (без кавычек) и нажать ENTER | 6 | 8 БАГ!!!! |

(Это, кстати, реальный баг старого Microsoft Excel – он не учитывал приоритета математических операций, по которому умножение имеет высший приоритет по сравнению со сложением) (<http://habrahabr.ru/blogs/testing/104952/>)

Поведение системы должно быть описано в документации, т.е. в требованиях. Когда требования нечеткие, неполные, содержат несоответствия, то это является причиной появления дефектов.

Еще определения:

Дефект - проблема, которая является причиной неправильной работы приложения или неверно возвращаемых ею результатов.

**Terms for Software Failures**

В различных компаниях применяются различные определения терминов, которые описывают проблемы в ПО. Вот некоторые из них:

|  |  |
| --- | --- |
| Defect | Variance |
| Fault | Failure |
| Problem | Inconsistency |
| Error | Feature |
| Incident | Bug |
| Anomaly | Issue |

Так, например, **fault, failure, defect** подразумевают такие условия, которые могут быть действительно небезопасными для программы. Эти слова также могут означать "It's his fault that the software failed."

**Anomaly, incident, and variance** не несут строго негативный характер, и часто обозначают не полный отказ приложения, а лишь какую-то неисправность.

**Problem, error, and bug** наверное являются наиболее часто используемыми.

Определение бага:

* The software does (does not do) something that the product specification says it should not (should) do.

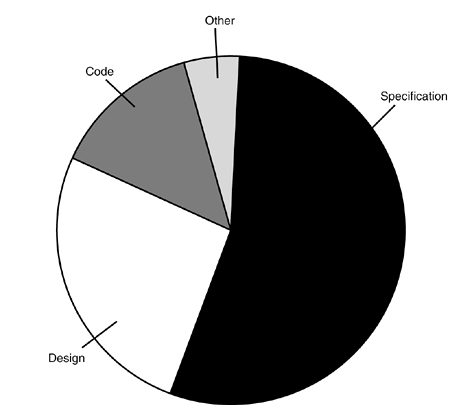
Например:

1. В спецификации указано, что приложение не должно сохранять форму с незаполненными полями. Но оно позволяет.
2. В спецификации указано, что приложение должно выводить сообщение с просьбой заполнить пустые поля. Вместо этого, форма закрывается, сообщение не появляется.

* The software is difficult to understand, hard to use, slow, or the software in tester's eyes will be viewed by the end user as just plain not right.

Программа сложна для понимания. Тяжела в использовании, медленна, тестировщик, смотря на эту программу глазами конечного пользователя видит, что она работает неправильно.

Рассмотрим причины, по которым возникают ошибки в программе. Вы можете быть удивлены, когда обнаружите, что большинство из них возникают не по причине ошибок в программировании. Множество исследований были проведены на очень маленьких проектах, но результаты получались одинаковыми. Причиной №1 были ошибки в спецификации.



Есть несколько причин, по которым спецификации являются наибольшим производителем багов. Вообще для многих программных продуктов спецификации вообще не написаны. Другой причиной может быть то, что спек неточный, постоянно меняется, или команда разработчиков не достаточно осведомлена о ней. Планирование ПО является исключительно важным. Если оно было выполнено некорректно, то это может стать причиной появления багов.

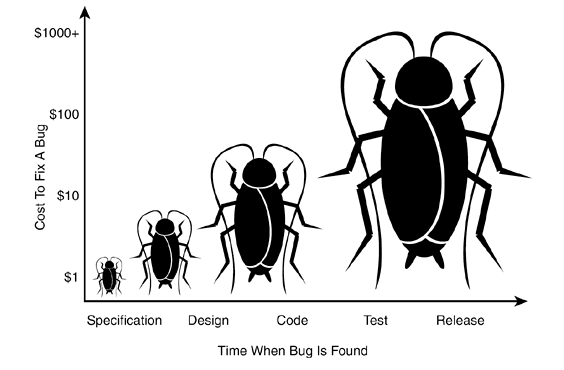
Следующим большим источником багов является дизайн – архитектура. Здесь программисты создают скелет приложения, продумывают его основные составные части. Это можно сравнить с созданием проекта здания архитектором. Если архитектура также будет выполнена в спешке, неточно, или программисты будут не осведомлены о ней, то могут появляться серьезные дефекты.

NOTE

There's an old saying, "If you can't say it, you can't do it." This applies perfectly to software development and testing.

Ошибки в кодинге возникают из-за сложности архитектуры ПО, плохой документации (особенно когда код часто обновляется), ограниченность временных ресурсов, или обычных ошибок в коде. Важно отметить, что многие дефекты, которые всплывают на поверхность, могут быть действительно отнесены к спецификации или ошибкам проектирования. Часто от программистов можно слышать, что «если бы мне кто-то сказал, то я бы не написал код именно так».

На пути создания ПО от идеи, через планирование, программирование и тестинг до его использования конечным пользователем, возможно нахождение багов на каждом этапе. На данной картинке изображен пример, как возрастает стоимость фикса багов со временем.



Стоимость растет логарифмически, она растет десятикратно с ростом времени. Баг, найденный и пофикшенный на ранних стадиях, когда спецификация только была написана, может еще ничего не стоить, или 1$, например. Та же ошибка, найденная во время кодинга или тестинга, может стоить от 10 до 100 $. Если же ее найдет пользователь, то стоимость может быть тысячи или миллиона долларов.

As an example of how this works, consider the Disney Lion King case discussed earlier. The root cause of the problem was that the software would not work on a very popular PC platform. If, in the early specification stage, someone had researched what PCs were popular and specified that the software needed to be designed and tested to work on those configurations, the cost of that effort would have been minimal. If that did not occur, a backup would have been for the software testers to collect samples of the popular PCs and verify the software on them. They would have found the bug, but it would have been more expensive to fix because the software would have to be debugged, fixed, and retested. The development team could have also sent out a preliminary version of the software to a small group of customers called a beta test. Those customers, chosen to represent the larger market, would have likely discovered the problem. As it turned out, however, the bug was completely missed until many thousands of CD-ROMs were created and purchased. Disney ended up paying for telephone customer support, product returns, replacement CD-ROMs, as well as another debug, fix, and test cycle. It is very easy to burn up your entire product's profit if serious bugs make it to the customer.

**Что делают тестировщики?**

Цель тестировщика – найти баги. Есть утверждения, что цель тестировщиков - убедиться в том, что ПО работает, но не найти баги. Но если вы будете тестировать вещи, которые должны работать, и делать тест кейсы таким образом, что они будут лишь проверять правильность работы и которые будут проходиться успешно, то вы пропустите вещи, которые не работают. Вы пропустите баги.

The goal of a software tester is to find bugs.

Если вы пропустите баги, то это выльется в бюджете вашей компании. Поэтому вы должны заботиться не только о том, чтобы найти баги, а еще о том. Чтобы сделать это как можно скорее.

The goal of a software tester is to find bugs and find them as early as possible.

Но и этого недостаточно. Помните определение бага, вы смотрите на ПО глазами пользователя, который впервые видит ваш продукт. Вы должны добиваться совершенства.

The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.

This final definition is very important.

**Итог.** Software testing - это очень важный этап в разработке ПО. Те приложения, которые пишутся сегодня, с их размерами и всей функциональностью, крайне необходимо, чтобы тестирование было качественным и эффективным.

**Defect reporting tips**

1. Необходимо просмотреть все дефекты в базе для того чтобы избежать дублирования дефектов.
2. Написать четкий и краткий Summary для заведенного дефекта.

Example

Bad summary: ‘Product does not work’

Good summary: ‘Crash on exit only after save’

1. Проверять правописание
2. Defect priority – “how important the defect is from business perspective”, determines the order in which defects get fixed

Defect severity - “how severe the defect is from software system’s perspective”, reflects the actual or expected impact on customer

**Types of defects**

**Software changes** – дефекты, которые созданы, чтобы отслеживать изменения функциональности и исправлять дефекты в разных версиях продукта (для каждой версии свой дефект). Эти дефекты отображают состояние, в котором находится проблема на текущий момент. Programming Manager или Programming Team Lead ассайнит таски, которые нужно пофиксить программистами. Работа включает в себя изменения в коде, перекомпиляцию и тестинг.

**Software** – дефект, когда функциональность продукта работает неправильно. (не по рекварментам) Необходимо изменять приложение (design, coding и integration testing)

**Test Case** – Тест Кейс (проверочный пошаговый сценарий) не соответствует рекварментам, или ТК нужно создать.

**Environment** – проблема с энварментом (он неправильно работает, или неправильно настроен), или это проблема SE. Корректировать конфигурацию продукта.

**External -3rd Party** – дефекты, которые не связаны с функциональностью разрабатываемого продукта. Найти воркараунд или изменять другой продукт.

**Hardware** – проблема в ПО связана с ошибками в hardware. Нужно изменять / исправлять hardware, за это отвечают SE.

**Installation** – была ошибка в инсталляции или процедуре инсталляции, или инсталляционный скрипт неисправен. Необходимо изменять инсталляционный скрипт или сборку, или еще раз переинсталлить продукт.

**Defect Type:**

* Business Logic
* Coredump
* Grammar
* Intended Use
* Interface
* Non Functional - дефект не влияющий на внешние проявления функциональности продукта (неверная / неточная документация, переделка внутренней реализации функциональности, т.е. рефакторинг), внешнее поведение программы остается такое же;

**Defect Statuses**

## Evaluate the problem

Once the defect is registered in bug tracking system, product specialist or programmer evaluates the defect in order to understand what needs to be done to resolve the problem.

## Schedule the problem fixing

As soon as the defect is evaluated, it can be moved to the following target statuses:

* Open – when a defect is added to the scope of certain version
* New – when the evaluation of the defect lacks some details, the defect can be sent back for re-evaluation
* Reviewed – when the problem was already fixed before.
* Closed – when no changes are required, (if the defect was created by misunderstanding)

## Fix the problem

* When the defect is in status “Open” it means that the defect will be fixed as soon as the development of target version is started.

## Review the changes

After changes are performed, they should be reviewed by another employee to make sure there’s no error and the solution chosen is the optimal one.

## Perform internal testing

After changes are reviewed, they should be tested by internal QA.

## Perform change testing

Change testing is performed by a special testing team, independently from programmers.

## Close the defect

As soon as defect is tested, Programming Manager or Product Specialist can close the defect.

## Resolve the problem found during testing

As soon as some problem is found during testing and the defect status is changed to “Rework”, there are 2 ways how the problem can be resolved:

* If the fix proves to be really incorrect, defect should be re-fixed (status is changed to “Open”);
* If the fix is still correct, re-testing is required (status is changed to “Fixed”).

**Баг-трекинговые системы**

**Defect tracking System** - a software application that is designed to help quality assurance and programmers keep track of reported software defects in their work.

**Defect Tracking Tool common tasks:**

Collect defect/request records

Assign defect/request record to the responsible person

Collect defect/request history

Find defect/request by defined condition

Filter defect/request list by defined condition

Close defect/request record

В большинстве современных компаний используются специальные системы для регистрации дефектов, или багов. В эти системы тестировщики заносят найденные ошибки, разработчики их исправляют, а менеджеры проектов смотрят статус продукта. Ведение дефектов в таких системах и называется баг-трекингом. Кстати, если ты читал книгу Канера, то явно знаком с подходом к баг-трекингу, который применялся десятки лет назад: печать багов на бумажках в трёх экземлярах с передачей этих листков разработчикам :) К счастью, эти времена давно прошли, и для баг-трекинга используются специальные системы, называемые баг-трекерами.

**Но зачем что-то в них регистрировать, почему нельзя устно сообщить о проблеме? Причин несколько:**

* Информация может забыться, потеряться, поэтому её важно хранить в одном месте
* Менеджеры проектов используют баг-трекер для анализа: "готов ли продукт?", "сколько задач ещё необходимо выполнить?" и "когда я могу обещать выпуск релиза руководству?". Для этого они считают баги, смотрят графики, анализируют сходимость дефектов, а сделать это по бумажкам не представляется возможным.
* Структурируя информацию мы можем отслеживать статус и не забывать про свои задачи, связанные с уже заведёнными дефектами. Да-да, просто завести их мало, их ещё поддерживать надо, но об этом ниже.

При этом от того, насколько качественно мы заводим дефекты, во многом зависит, будут ли они исправлены и когда. **Что же такое качество заведения дефекта?**

* В первую очередь, это **точность дефекта**. Чем точнее мы можем определить причины неработоспособности, тем лучше. "Иногда здесь что-то не работает" - это плохое описание дефекта. Когда именно не работает? При каких условиях? Обобщая влияющие факторы, мы проводим **генерализацию** дефекта, а сужая набор факторов, мы **локализуем** дефект. К примеру, мы тестируем калькулятор, и 2+2 не работает. Двоечник сразу заведёт ошибку, а джедай начнёт анализировать: а другие операции? А другие числа? А другие последовательности, окружения, условия? Чем точнее дефект, тем проще разработчику его исправить, а менеджеру проекта понять его влияние.
* Во-вторых, это добавление любой **полезной информации**. Логи, краш-дампы, скриншоты. Не жалуйтесь, что это долго, иногда одна минута на добавление полезных файлов сэкономит несколько часов вашему проекту!
* В-третьих, это **оформление дефекта**. Понятно ли всё описано? Иногда то, что понятно вам, абсолютно непонятно другим людям. Занося дефект, вы его уже знаете, и вам кажется, что всё ясно. Возьмите за привычку перед отправкой дефекта прочитать его глазами "человека со стороны". Точно ли всё понятно?

Конечно, помимо вышеперечисленных пунктов, в каждой компании есть свои собственные правила: как формулировать заголовок, какие поля заполнять и т.д. Но перечисленные выше пункты являются общими и всегда очень важны для любой команды.

Ну хорошо, представим, что мы заводим качественные дефекты. Этого достаточно?

К сожалению, нет. Еще нужно**:**

* **Правильно указывать критичность дефекта**. А это зачастую невозможно, если вы плохо знаете заказчиков, клиентов, бизнес-сторону вашего продукта. Если РМ или аналитик регулярно понижает или повышает критичность вашим дефектам, значит, вам экстренно не хватает знаний о бизнес-области и бизнес-логике вашего приложения! А если это так, то у вас и тестировать продукт правильно не получится. Вывод? Узнаём, исследуем, спрашиваем аналитиков, РМ'а, внедренцев.
* **Проверять исправление дефектов**. Каждый раз, когда разработчик помечает дефект как исправленный, в этом ещё надо убедиться. Причём, помимо проверки самого дефекта, посмотрите немного "вокруг": вдруг при починке было сломано что-то "рядом"? Обычно, так оно и происходит.
* **Регулярно узнавать мнение о дефектах у коллег** (разработчиков и РМа): устраивают ли их заводимые вами дефекты? Всё ли понятно? Если кто-то на что-то жалуется - выясняйте, как это можно улучшить. Только непрерывно совершенствуя своё мастерство вы можете начать заводить действительно отличные дефекты.

**Open source BT Systems: (**<http://www.thegeekstuff.com/2010/08/bug-tracking-system/>**)**

### Bugzilla - написан на Perl, и работает на различных базах данных, включая MySQL и Oracle. Широко используется различными крупными open source проектами.

Bugzilla Features:

* Time tracking
* Private attachment and commenting
* Flexible reporting and charting (via email).
* Adding custom fields and workflows.
* [Email Notifications Controlled By User Preferences](http://www.bugzilla.org/features/#email)
* [Automatic Duplicate Bug Detection](http://www.bugzilla.org/features/#autodup)
* [File/Modify Bugs By Email](http://www.bugzilla.org/features/#email-in)
* [Request System](http://www.bugzilla.org/features/#rs)
* **Mantis**. Написана на PHP, and works on various databases including MySQL, MS SQL, PostgreSQL.

Mantis Features:

* Web based, support any platform that runs PHP
* Source code integration
* Time tracking
* Issue relationship graph
* Custom fields and workflow
* Anonymous access

- **Trac.** Написан на Python. Также предоставляет wiki, интеграцию с подверсиями. Очень простой web интерфейс. Также предоставляет функциональности для проджект-менеджмента, включая milestone tracking.

- **Readmine.** Написана на Ruby. Кроме бег трекинга предоставляет полный комплект инструментов для проджект-менеджмента.

Features:

* Project management including
* Project Wiki
* Time Tracking
* Flexible issue tracking system
* Multiple database support