~ UNIT 5. Bug Tracking ~

Lesson 29

Bug Tracking System / Part 4

Trust, but verify.
- Ronald Reagan

BTS Attributes: Resolution

Resolution is a drop-down menu with the following list:

Reported
Assigned
Fix in progress
Fixed
Fix is verified
Verification failed
Cannot reproduce
Duplicate
Not a bug
3rd party bug
No longer applicable

Resolution is one of the most important BTS attributes. If Status is about global things like "was born," "died," and "got reincarnated," Resolution provides details like "graduated from college", "got married", "bought a condo" etc. Resolution describes the stages of a bug's life.

Question: What drives a bug from one life stage to another?
Answer: Certain activities geared towards **bug resolving**.

Question: Where can we find information about those "certain activities" and the associations between them and the concrete values for **Resolution**?
Answer: In the **Bug Tracking Procedure**.

Let's learn about each Resolution.

Because the majority of bugs are found in the software (not in the specs or other documentation), we'll be talking about situations where the person responsible for the bug fix is the **programmer**.

In our course each Resolution is illustrated by a bug filed into the bug tracking system of Test Portal.

You can see those bugs here: Test Portal>Bug Tracking>**Bug Vault**.

Resolution: Reported

This Resolution must be chosen when the person who files a bug doesn't know who will be fixing the bug. This is the situation when a tester assigns a bug to himself.

In case of sophisticated BTS software, we can program our **Bug Tracking Procedure** (**BTP**) into the BTS workflow.

For example, **depending on the situation, the required Resolution value would automatically be selected**.

For this course, we assume that the people who use the BTS voluntarily follow the BTP.

*See **Bug Vault>Bug #2***

Resolution: Assigned

This Resolution means that the programmer in *Assigned to* should investigate the bug.

Example 1: The person who files a bug knows who is going to fix it, so that person selects name of the programmer from *Assigned to* and selects the **Assigned** for Resolution.

*See **Bug Vault>Bug #3** – please note that this bug was invented to illustrate the point. ShareLane application doesn't have this bug.*

Example 2: The person filed a bug with **Reported** status, but changed **Reported** to **Assigned** (and put programmer's alias into *Assigned to*) once he found out who was going to fix the bug.

*See **Bug Vault>Bug #4** (pay attention to Change History)*

Resolution: Fix in Progress

The programmer selects this Resolution **as soon as he starts working on the bug fix.**

*See **Bug Vault>Bug #5** (pay attention to Change History)*

Resolution: Fixed

The programmer selects this resolution as soon as he has checked the bug fix into the CVS. Along with that, the programmer must select the same alias as in *Verifier* from the drop-down menu *Assigned to*.

*See **Bug Vault>Bug #6** (pay attention to Change History)*

Testers should remember that it takes time before the build script picks up the CVS content and populates it onto a particular environment. So, if a programmer changed the Resolution to **Fixed** at 8:00 am, and the next build starts at 9:00 am and takes 15 minutes to complete, then the bug fix will be available on the target environment no earlier than 9:15 am.

The moral of the story: **before you start regressing the bug, make sure that the build with the bug fix has already been pushed to the target environment and that the build was successful.**

In other words, check out 2 things before you begin bug fix verification:

- **Build status** (see example at Test Portal>Release Engineering>Build Status)

- **Application version on the target environment** - you can see it if you view the HTML source of any Web page on ShareLane.com:

<!-- application version 1.0-23/34 -->

Resolution: Fix is Verified

Remember that bug regression consists of two parts:

Part 1. Verification that the bug was really fixed
Part 2. Checking if the bug fix hasn't broken some other parts of software.

First of all, we just try to **reproduce the bug** following the instructions in the Description:

- **If the bug is NOT reproducible, then it was really fixed.**

- If bug IS reproducible, then it wasn't fixed, so we send it back to the developer with the Resolution **Verification failed** – and we DO NOT execute Part 2 of the bug fix verification.

If the bug is NOT reproducible, we move to Part 2. This is where it gets interesting. In the case of more or less complex software, it's sometimes extremely hard (even for a programmer) to confidently predict how a certain change to a certain piece of code will affect other parts of the software. So, the only way to do comprehensive regression testing is to perform 100% coverage of all possible scenarios (what's usually impossible!).

The standard way to execute Part 2 is to perform a basic end-to-end test of the feature that contained the bug. In special cases (for example, in case of Emergency Bug Fix) you can:

a. Ask the developer who fixed the bug what could have gotten messed up as a result of the bug fix and what he recommends that you check out in particular.
b. Next follow his instructions.
c. Then perform a basic end-to-end test of the feature that contained the bug.

Please note that you can:

- Change the Resolution from **Fixed** to **Fix is verified** AND
- **Close the bug** (change its Status to **Closed**) as soon as Part 1 is finished and you have verified that the bug was really fixed.

Why bother about Part 2? We need Part 2 simply as a safety measure. Of course, in the case of Part 2 we're not talking about serious testing - we just do a quick check that usually takes several minutes.

*See **Bug Vault>Bug #7** (pay attention to Change History)*

Resolution: Verification Failed

The verifier changes the Resolution to **Verification failed** and *Assigned to* to the alias of the responsible developer if the bug is reproducible; that's if Part 1 of the bug fix verification failed.

*See **Bug Vault>Bug #8** (pay attention to Change History)*

Resolution: Cannot Reproduce

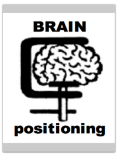
This unpleasant situation occurs if the developer tries to reproduce the bug assigned to him or her, and cannot do it. The bug is usually not reproducible for 1 of 3 reasons:

1. The tester didn't provide a comprehensive Description.
2. The tester's environment and the developer's environment are different.
3. There is no bug.

Let's look at these reasons one by one.

1. **The tester didn't provide a comprehensive Description.**

One of the main concepts about bugs is the idea that **if a bug exists, then it is reproducible**. **NEVER file a bug until you reproduce it at least once after you discovered it.** Read the following *Brain Positioning,* and remember it for the rest of your testing career!



Sometimes you can get really excited once you find a bug, especially a fat P1, and you can be tempted to run to the developer RIGHT AWAY to share your finding. **Don't do this!**

Whatever the problem is:

1. **Try to reproduce it.**
2. **Try to narrow it down - in other words, try to isolate the problem.**

For example*:* Let's assume that after you clicked the **Make Payment** button using *Google Chrome on Windows 8*, you got a **500 – Internal Server Error**. What you should do next is this:

1. **Try to reproduce it again**, and write down the exact parameters (book title/price/quantity/credit card info: type: number, cvv2, expiration date).

2. **Play with those parameters**; for example, try to reproduce the bug using a different credit card.

3. **Try to use a different browser/OS combo**; for example, try Firefox under Windows 8 or Google Chrome under Mac OS.

4. If a feature is on production and you found a bug in the test environment, try to reproduce the bug on production.

After your **4-step investigation** is done, file a bug report right away, or go talk to developer (and file the bug RIGHT AFTER that).

Again, **if a bug exists, then it's reproducible**.

Each bug is a result of a certain scenario which is:

1. **Actions**
2. **Data**
3. **Conditions.**

Usually people are more focused on providing info about actions and data rather than about conditions. Try to avoid that pitfall! **In many cases, it's the condition that makes all the difference.** In some cases, a condition can be very unusual or hard to grasp, but if you pay attention you'll find that condition. Please read the story below to better understand my point.

4 Scientists and 1 Flask

Once upon a time, there was a pharmaceutical laboratory where 4 scientists worked to find cures for illnesses. Scientist Leo N. invented a unique chemical substance that could serve as the basis of a new, powerful medicine. The problem was that the 3 other guys could not produce the substance, even if they methodically followed Leo's every step. Leo was happy to share all the information he possessed about the process, but the others had no luck - it seemed like Leo had some unexplainable ability. One evening, those 4 scientists with their PhDs in chemistry got together and decided that they were going to believe in miracles, but only after one last thing: during the preparation of the substance, Leo's EACH AND EVERY action must be captured on video and analyzed afterwards.

Following the plan, after the video was ready, the 4 colleagues got together and made a thorough analysis of EACH AND EVERY one of Leo's actions. After several hours of analysis and conducting tests they found out what was really happening: in the middle of the preparations, whoever was preparing the substance had to walk for 1 minute between the two laboratories, which were situated in different buildings. It was wintertime, BTW. Leo was a smoker, so before going outside **he would put the flask under his coat to free his hands for a cigarette and matches**. Therefore, the substance in the flask wasn't exposed to the cold like it was with the 3 other scientists who didn't smoke and who simply ran between the buildings with the flask in their hands! So the magical condition that made all the difference was: **Don't expose the flask to the cold!**

The moral of the story here is that, in some cases, even a little, hard-to-notice nuance makes all the difference.

Let's get back to our testing. In many cases, a bug is not reproducible not because the tester didn't know the scenarios/conditions, but **because he just didn't write a comprehensive Description.**

Please remember that even if you get excited about finding some buggy area and you want to continue hunting for bugs instead of filing bugs, it's your direct responsibility to:

- File bugs
- Make sure that others can understand your bugs and **be able to reproduce them**.

It **really bad** when a developer must put off his coding, spend time trying to reproduce a bug, and is not able to do it because the tester forgot to include a little detail right in the middle of **Steps to reproduce**. You must avoid situations like that!

Do your best to avoid "merchandize returns" that come with the Resolution Cannot reproduce!

2. The tester's environment and the developer's environments are different.

Again, it's all about conditions. There are situations where some kind of underlying item is different in the dev environment (for example, **billy**.sharelane.com) than in the test environment (for example, **main**.sharelane.com).

For example, the dev environment could have a more recent version of the Python interpreter, so Billy could use the Python libraries that exist in his environment but are missing on main.sharelane.com. The tester sees a bug and files it, only to get a **Cannot reproduce** Resolution, which leads to a waste of time going back and forth with Billy trying to understand what's really going on!

3. There is no bug.

Here's a potential situation: The DB on main.sharelane.com is down for maintenance, and the tester doesn't know about it. He sees the bug, tries to reproduce it (with success), and then files it. The programmer tries to reproduce the bug **when the DB outage is over**, but the bug doesn't exist anymore. So the programmer sends the bug back with the **Cannot reproduce** Resolution.

A similar unpleasant situation might happen when the testing is done while a push of a new build is under way.

ALWAYS check the build status page before starting testing!

In case of a **Cannot reproduce** Resolution, whoever selects that Resolution should also assign the bug back to the person who filed it - it's the person from the *Submitted by* field.

What happens if a tester receives a bug back? He should either close it OR provide more details and assign it back to the developer with an **Assigned** Resolution.

*See **Bug Vault>Bug #9** (pay attention to Change History)*

Resolution: Duplicate

This Resolution is selected if another bug was already filed for the same issue. In software companies, the BTS may contain thousands of open and closed bugs, and sometimes it's not physically possible to review each of them to avoid filing a duplicate. The best way to avoid duplicates is to

- Do comprehensive search in BTS (BEFORE filing a bug) and
- Keep close look at all new bugs filed for the functional area or component you are responsible for. Example of the functional area is **Checkout**.

A professional BTS usually allows you to modify the settings for your BTS account and sends you an email if any new bug is filed or a filed bug meets certain criteria, for example, when value of BTS attribute **Component** equals **Checkout**.

That way you'll know what was already filed.

On the other hand, there can be not-so-obvious situations. For example, **two or more filed bugs can be the result of the same root cause**.

Let's look at Bug #4 in the Bug Vault: "**shopping_cart.py: 2% discount if user buys 12-19 books inclusively**".

The WRONG way to do it is to file 8 bugs:

1. "*shopping_cart.py: 2% discount if user buys **12** books*".
2. "*shopping_cart.py: 2% discount if user buys **13** books*".
3. "*shopping_cart.py: 2% discount if user buys **14** books*".

…

8. "*shopping_cart.py: 2% discount if user buys **19** books* ".

The root cause is the following statement in **shopping_cart.py**:

```
if q >= 12 and q <= 49:
    discount = 2
```

So, one bug is enough to cover all 8 cases.

Black box testers don't look into the code, and **situations like this do take place**. So when the developer returns your bug with a **Duplicate** Resolution, explaining (in *Comments*) that your bug was caused by the same code as another bug, don't take it personally – that happens, and it's okay.

Once a bug is marked as **Duplicate**, the person making that change must also put the original (previously found) bug ID in *Comments* and assign the bug back to the person who submitted it.

*See **Bug Vault>Bug #10** (pay attention to Change History)*

Resolution: Not a Bug

Startups, especially in the early stages, are usually a mess.

In some cases, it's really hard to find out the **correct** expected result, and our old friends: common sense and life experience might be different from what the PM tried to communicate during his talk with the developer some time ago.

In some cases the PM communicates a product change to the developer and forgets to:

- update the spec and
- mention that product change to the tester.

In some cases, the tester is correct that it's a bug, but the developer thinks that it's a feature and sends the bug back to the tester.

So be prepared to answer the question: "**Why is that a bug?**" If you were wrong about it, but you had reasons for believing that it WAS a bug, then there is no problem.

If you are not sure and it's a good time to ask ("Hey Linda, is this a bug or not?"), then ask. If nobody is available, then file a bug report. It's better to file a bug (if you have reason to believe that it *is* a bug) and later find out that it's not a bug, than be scared of the **Not a bug** Resolution and ignore the real problem.

In any case, when you receive your bug back with the **Not a bug** Resolution, read the Comments section and try to understand what the developer tried to communicate. If you still think that you are right, go talk to the developer, and then:

- If you both agree that it is a bug, send it back to the developer (*Assigned to*: developer's alias; Resolution: **Assigned**) with your explanations in *Comments*.

- If you both agree that it's not a bug, just close it.

Sometimes when a spec is poorly written, the tester and developer cannot come to a consensus about what the PM meant in the first place. In that case, reassign the bug to the PM and let him decide whether it's a bug or not. In a situation like this, we also recommend filing a separate bug against the spec.

See an example of ping-pong with "Not a bug" here:

*See **Bug Vault>Bug #11** (pay attention to Change History)*

Resolution: 3rd Party Bug

The code in the software company cannot rely exclusively on in-house software – which is a software code written by company programmers. We use databases, compilers, interpreters, and Web servers that have been designed and developed by others.

In many cases, we also integrate our software with the software of our partners, for example, with credit card processors. So, here is the situation: you file a bug, the bug programmer returns this bug to you with a **3rd party bug** Resolution and the comment that the bug is not in the software that he wrote, but instead in the software written by others. Here we might encounter two situations:

1. **We CANNOT influence "others" to fix their software** – for example, if the bug is in the Python interpreter, we cannot call Python's father, Guido van Rossum and ask him to fix the bug before our release goes out. So, the only way to fix the bug is to find some programming **workaround**. So, if a programmer returns your bug with a **3rd party bug** Resolution, it doesn't solve the problem, because whether he likes it or not, help is not coming, and we have to find a solution ourselves. In a case like this, talk to the programmer and re-assign the bug back to him.

*See **Bug Vault>Bug #12** (pay attention to Change History) - please note that this bug was invented to illustrate the point. ShareLane application doesn't have this bug.*

2. **We CAN influence "others" to fix their software**.

For example, in **checkout.py** we have the function **get_ccp_result()** (you can see source code here: Test Portal>Application>Source code>checkout.py).

This function connects to the payment processor to process credit card payments (*of course, in case of ShareLane there is no real processor – it's all training software*). That payment processor software belongs to another company, which is our service provider (also called **vendor**). If there is a bug in the vendor software, then we can call them and ask them to fix the problem ASAP. As a rule, the tester usually doesn't make this contact. As a rule, a contact like this is the responsibility of the project manager (**PjM**). *BTW, in startups PMs are often also PjMs.*

So, if a developer returns a bug to you with a **3rd party bug** Resolution in this case, you should reassign the bug to the PM and that person would be the bug owner that is responsible for resolving the situation.

*See **Bug Vault>Bug #13** (pay attention to Change History)*

Resolution: No Longer Applicable

This is usually selected for bugs found in features that have been deprecated.

Deprecated feature is a feature that is still available to users but no longer officially supported.

*See **Bug Vault>Bug #14** (pay attention to Change History)*

Lesson Recap

Bug Resolutions include:

1. **Reported**: A bug was filed, but the developer to fix it has not been assigned.

2. **Assigned**: Assigned developer must start bug investigation.

3. **Fix in progress**: The developer is fixing the bug.

4. **Fixed**: The bug was fixed, but the bug fix hasn't been verified yet.

5. **Fix is verified**: The bug fix has been verified.

6. **Verification failed**: The bug fix verification failed - in other words, bug is reproducible after the bug fix.

7. **Cannot reproduce**: The developer cannot reproduce the bug.

8. **Duplicate**: The bug is a duplicate of another bug.

9. **Not a bug**: The bug is not considered to represent a problem (which is a deviation of actual from expected).

10. **3rd party bug**: The bug is in 3rd party software.

11. **No longer applicable**: The bug doesn't have any meaning anymore.

Homework

Match bug Summaries to bug Resolutions:

Bug Summaries:

1. "add_to_cart.py: cannot add book using Internet Explorer v.3.0" (IE 3.0 was released in 1996)
2. "add_to_cart.py: bug in Python 2.7 interpreter"
3. "add_to_cart.py: only 1 book can be added to Shopping Cart"
4. "add_to_cart.py: bad logic"

Bug Resolutions:

1. ***Reported***
2. ***Cannot reproduce***
3. ***3rd party bug***
4. ***No longer applicable***

Quiz

Question 1: Reported means that it's not clear yet who is going to fix that bug.

☐ True

☐ False

--

Question 2: Assigned means that the person should fix the bug or find a person who will fix the bug.

☐ True

☐ False

--

Question 3: Fix in Progress means that programmer could reproduce the bug.

☐ True

☐ False

--

Question 4: Fixed means that the bug fix is in version control system and will be available in the next build.

☐ True

☐ False

--

Question 5: Bug fix verification assumes a two part process.

☐ True

☐ False

--

Question 6: Verification Failed means that the bug is reproducible.

☐ True

☐ False

--

Question 7: Cannot Reproduce means that there is no bug.

☐ True

☐ False

--

Question 8: Not a Bug means that there is no bug.

☐ True

☐ False

--

Question 9: 3rd party bugs are not fixable.

☐ True

☐ False

--

Question 10: No Longer Applicable is often applied to deprecated features.

- ☐ True
- ☐ False