Software Quality Assurance

Module 8

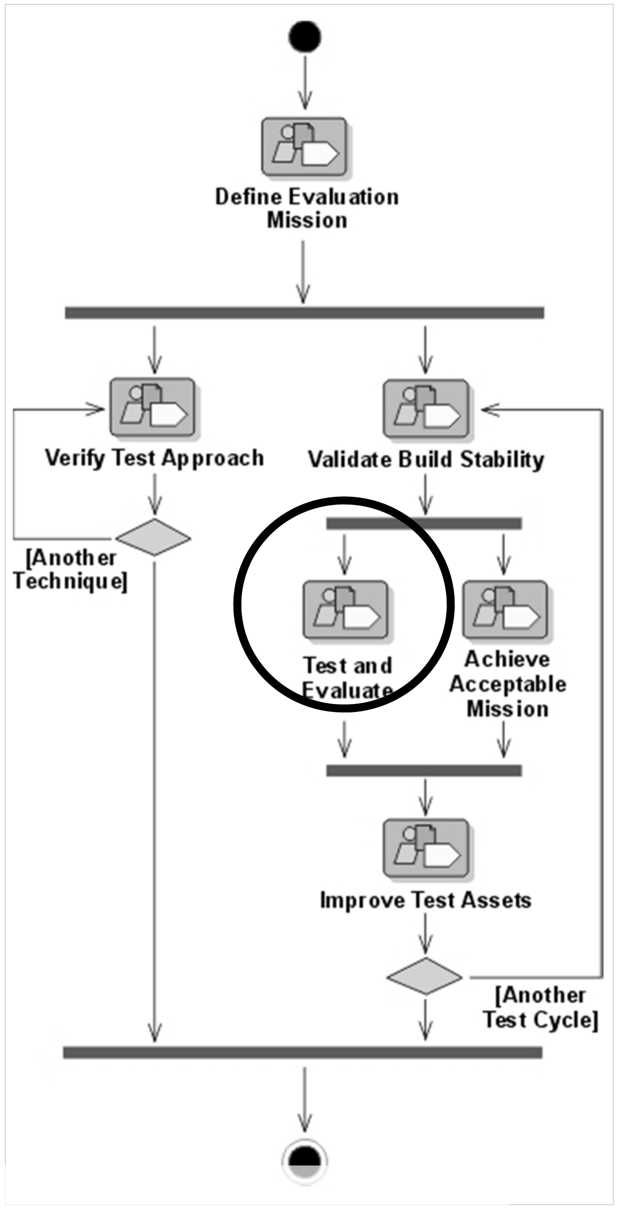
Analyze Test Failures

Module 8 Objectives

- This continues the workflow: ***Test & Evaluate***
- Module 7 focused on ***Test***
- This Module 8 focuses ***Evaluate***

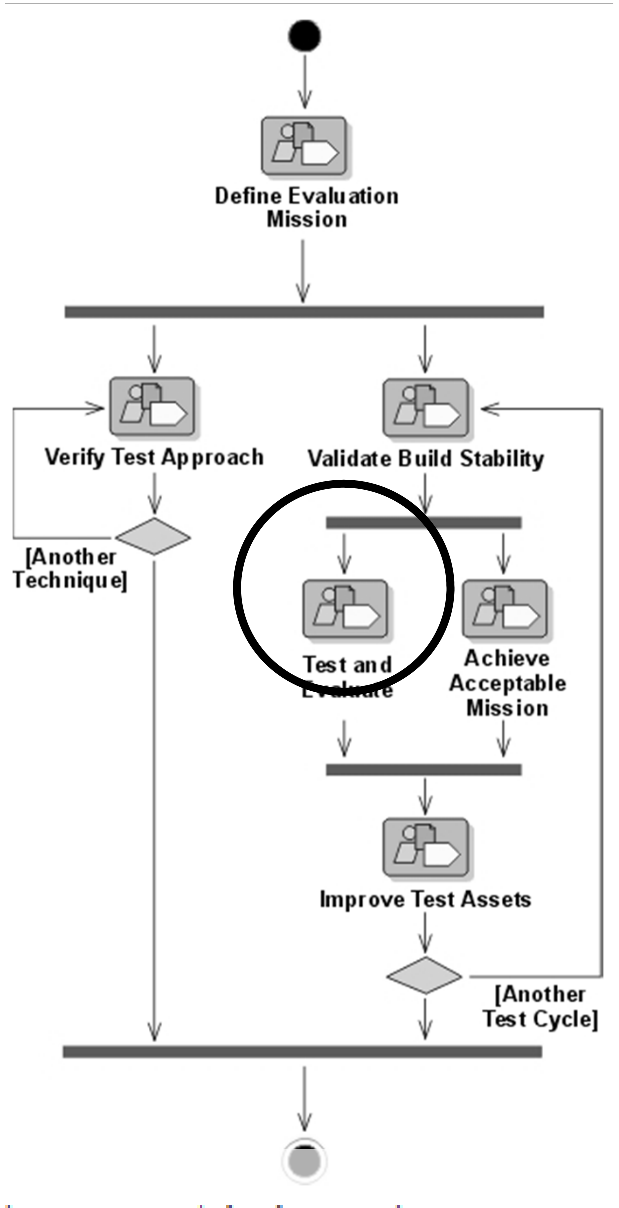
Review: Where We've Been

- In the last module, we covered part of the *Test and Evaluate* workflow detail.
- We focused on what *techniques* to use to implement appropriate tests



Test and Evaluate – Part Two: Evaluate

- In this module, we look at the *Evaluate* part of this work with the questions:
  - How will you evaluate your test results?
  - How will you report your findings?



Test and Evaluate – Part Two: Evaluate

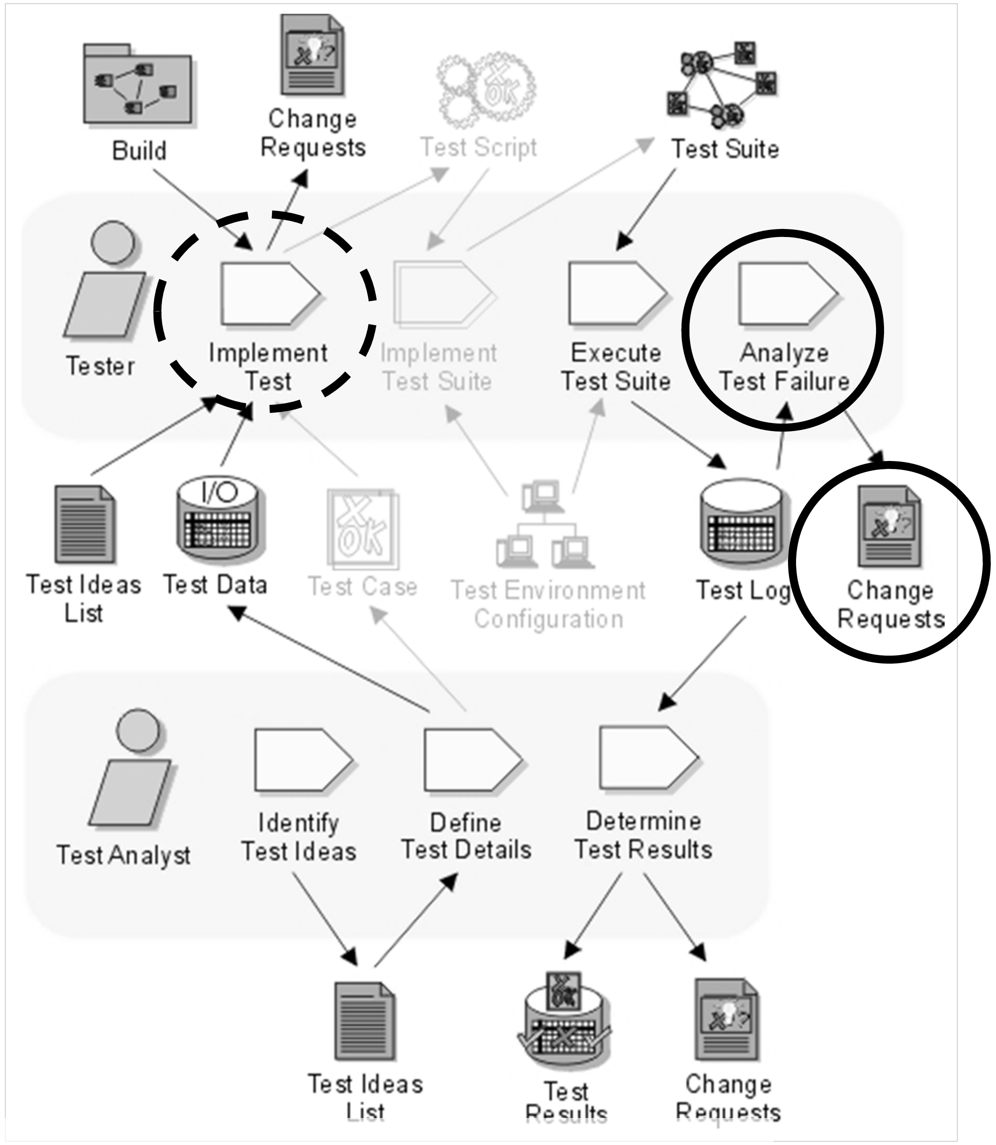
For each test cycle, this work is focused mainly on:

- •Providing ongoing evaluation and assessment of the Target Test Items
- •Recording the appropriate information necessary to diagnose and resolve any identified Issues
- •Achieving suitable breadth and depth in the test and evaluation work
- •Providing feedback on the most likely areas of potential quality risk

Typically enacted once per test cycle, this work involves performing the core tactical work of the test and evaluation effort: — namely the implementation, execution and evaluation of specific tests and the corresponding reporting of specific incidents that are encountered.

Test and Evaluate – Part Two: Evaluate

- This module focuses on the activity *Analyze Test Failures*
- In the last module, we covered *Implement Test* using different *techniques*
- We will spend a lot of time on *Change Requests*, as they are the product of this activity



Review: Definition of Quality

- In Module 2, we looked at definitions of Quality
  - Quality as fitness for use, determined by both customer satisfiers and dissatisfiers

    -- Dr. Joseph M. Juran
  - Quality as value to some person

    -- Gerald M. Weinberg
- Here we look at Analyzing Test Results
  - A bug report is a report that some aspect of the product appears to unnecessarily reduce its value
  - Testers (and others) may criticize design decisions -- the code may not be defective, the product is just unnecessarily clumsy
  - Effective communication is a key part of this activity.

Some Preliminary Definitions

- In this module, the following terms are used:
  - ***Change request***: any report of an incident, defect or potential enhancement, that is intended as a request for a change to the software under development
  - ***Defect report***: a change request reporting a (suspected) defect or error in the product
  - ***Bug***: some aspect of the product under test, that in the eyes a stakeholder, unnecessarily reduces its value; possibly a suspected defect
- In this module, no specific review process for change requests is assumed

Module 8 Agenda

- Advocate repairing the important problems
- Investigate problems effectively
- Write good change requests

Module 8 Agenda

- **Advocate repairing the important problems**
- Investigate problems effectively
- Write good change requests

Championing Your Defect Reports

A defect report is a tool that you use to convince someone to allocate time and energy to fix a bug.

Championing Your Defect Reports

- This leads to two important points.
- 1. **Visibility**. Defect reports are most testers' primary work product. Your defect reports are the only thing that many managers know about you.
  - *You get a reputation based on what you write.* This is what people outside of the test group will most notice and most remember of your work.
- 2. **Effectiveness**. The best tester isn't the one who finds the most bugs or who embarrasses the most developers. *The best tester is the one who gets the most bugs fixed.*

Discussion 8.1: What happens to your defect report?

- When you submit a defect report, what happens to it?
- What steps does it go through?
  - Who reads it? Do they see the whole thing or only a line in a summary report?
  - Who else reads it?
  - Who’s your primary audience?
  - Are there other audiences?
  - What’s your desired outcome?
  - How effective are you in practice?

Motivating the Reader: Analyzing the Impact

- What is it about a problem that makes the readers want to correct it?
- Here are a few ideas:
  - It looks really bad.
  - It will affect lots of people.
  - It looks like an interesting puzzle and intrigues the developer.
  - You’ve said that you want the defect fixed and the other stakeholders trust your judgment.

Overcoming Objections: Think About Your Audience

- What is it about a report that makes team members not want to spend time (and managers not want to allocate time) to make the change?
  - It will take a lot of work to make the change or fix the problem.
  - Appears to be unrealistic (e.g. “corner case”)
  - The developer can’t replicate the defect.
  - The developer doesn’t understand the report.
  - A fix will introduce too much risk into the code.
  - The developer doesn’t like / trust you (or the customer who is complaining about the bug).

Module 8 Agenda

- Advocate repairing the important problems
- **Investigate problems effectively**
- Write good change requests

Analyzing Failures with Follow-Up Testing

- Follow-up testing:
  - Show defect is more serious than it first seems.
- When you run a test and find a failure, you’re looking at a symptom, not at the underlying fault. You may or may not have found the best example of a failure that can be caused by the underlying fault.
- Therefore you should do follow-up work to try to prove that a defect:
  - is more serious than it first appears.
  - is more general than it first appears.

Exercise 8.2: Fault, Critical Condition, and Failure

- Here’s a defective program:

```
INPUT A
INPUT B
PRINT A/B
```

(Assume that “INPUT A” is a command that will accept only numbers as inputs.)

- What is the fault?
- What is the critical condition?
- What will we see as the failure?

Analyzing Severity: Follow-Up Testing

- When a coding error causes a failure:
  - The program is in a state the developer probably didn’t intend or expect
  - There may be impossible (unexpected) data values
- Keep testing to find full impact of the fault.
- Try four types of follow-up testing:
  - Vary your behavior (change the conditions by changing what you do)
  - Vary the options and settings of the program (change the conditions by changing something about the program under test).
  - Vary the software and hardware environment
  - Vary the data used by the program

Follow-Up: Vary Your Behavior

- Keep using the program after you see the problem.
- Bring it to the failure case again (and again).
  - If the program fails when you do X, then do X many times. Is there a cumulative impact?
  - Try things that are related to the task that failed.
  - Try things that are related to the failure.
  - Try entering the numbers more quickly or changing the speed of your activity in some other way.
  - Try the usual exploratory testing techniques.

Follow-Up: Vary Options and Settings

- When you follow up by varying options and settings, you treat the steps to achieve the failure as a given. Follow them without changing them.
- Try to reproduce the bug when the program is in a different state:
  - Use a different database.
  - Change the values of persistent variables.
  - Change how the program uses memory.
  - Change anything that looks like it might be relevant that allows you to change as an option.

Follow-Up: Vary the Configuration

- A bug might show a more serious failure if you run the program with less memory, a higher resolution printer, more (or fewer) device interrupts coming in, etc. For example,
  - If there is anything involving timing, use a really slow (or very fast) computer, link, modem or printer, etc..
  - If there is a video problem, try other resolutions on the video card. Try displaying MUCH more (less) complex images.
- Note that we are not:
  - Checking standard configurations
  - Looking for all circumstances that produce the bug.
- What we’re asking is whether there is a particular configuration that will show the bug more spectacularly.

Analyzing Generality: Configurations

- Defects that don’t reproduce on someone else’s machine are less credible.
- A report of a configuration dependent bug will be much more credible if it identifies the configuration dependence directly.
  - This sets the reader’s expectations correctly from the start.

Analyzing Failure Conditions

- Things that will make readers resist spending their time on the defect report:
  - Unrealistic (e.g. “corner case”)
  - The developer can’t replicate the defect.
  - Strange and complex set of steps required to induce the failure.
  - Not enough information to know what steps are required, and it will take a lot of work to figure them out.
  - The developer doesn’t understand the report.

Uncorner the Corner Case

- Readers sometimes dismiss tests as
  - ***unrealistic***, because they have no importance in real use
  - ***corner cases***, because they appear to combine extreme and unlikely values.
- *But once we find the defect, we don’t have to stick with extreme value tests or unusual conditions.*

Analyzing Non-Reproducible Errors

- Always report non-reproducible errors. If you report them well, developers can often figure out the underlying problem.
- Describe the failure as precisely as possible.
  - If you can identify a display or a message well enough, the developer can often identify a specific point in the code that the failure had to pass through.

Analyzing Non-Reproducible Errors

- The fact that a defect is not reproducible is data.
  - To reproduce a failure, you must put the program through the same relevant conditions as before.
  - If you can’t reproduce the failure, you are missing some of the conditions.
  - You can’t pay attention to all possible conditions, but some conditions are sometimes relevant.
- Keep a list of conditions involved in hard-to-reproduce failures.
  - When a failure seems irreproducible, look at your list. Could any of these conditions be the critical one? If so, vary your tests on that basis and you might reproduce the failure.
  - If a failure you couldn’t reproduce gets fixed, ask the developer what you have to do to see the defect. Identify the condition you missed. Write it in your notebook.

Analyzing Non-Reproducible Errors: Delayed Effects

- Some problems have delayed effects, such as:
  - A memory leak might not show up until after you cut and paste 20 times.
  - Stack corruption might not turn into a stack overflow until you do the same task many times.
  - A wild pointer might not have an easily observable effect until hours after it was mis-set.
- If you suspect that you have time-delayed failures, use tools to record a long time series of events:
  - Videotape, trace programs, capture programs, debuggers, debug-loggers, or memory meters

Module 8 Agenda

- Advocate repairing the important problems
- Investigate problems effectively
- **Write good change requests**

Writing the Defect Report: Make It Clear

- Your defects may be ignored or dismissed because, as written, they are
  - Too hard to understand or
  - They look too complicated to bother with.
- If you improve the reporting, more of the defects you report will be fixed.

Writing the Report: Keep it Simple

- If you see two failures, write two reports.
- Combining failures on one report creates problems:
  - The summary description is typically vague. You use words like “fails” or “doesn’t work” instead of describing the failure more vividly. This weakens the impact of the summary.
  - The detailed report is typically longer and more complex. Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and therefore more intimidating.
  - You’ll often see one bug get fixed but not the other.
  - When you report related problems on separate reports, it is a courtesy to cross-reference them.

Writing the Defect Report

- When the software fails, fill out a Problem Report form
- The headline is the most important part
  - It’s all many people see
- In the well written report:
  - Explain how to reproduce the problem.
  - Analyze the error -- describe it in a minimum number of steps.
  - Include every step needed to reproduce the problem.
  - Put each step in a separate paragraph; number the steps.
  - Make the report easy to understand.
  - Keep your tone friendly, neutral and non-antagonistic.
  - Keep it simple: one bug per report.
  - If a sample test file is essential to reproducing a problem, reference it and attach the file.
  - To the extent you have time, describe what events are and are not relevant and how results varied across tests.

The Defect Report: How to Reproduce the Problem

- First, describe the problem. What’s the bug? Don’t rely on the summary to do this -- some reports will print this field without the summary.
- Next, go through the steps that you use to recreate this bug.
  - Start from a known place (e.g. boot the program)
  - Then describe each step until you hit the bug.
  - NUMBER THE STEPS. Take it one step at a time.
  - If anything interesting happens on the way, describe it. You are giving directions to a bug. Especially in long reports, people need landmarks.

The Defect Report: How to Reproduce the Problem

- Describe the erroneous behavior and, if necessary, explain what should have happened. (Why is this a bug? Be clear.)
- List the environmental variables (config, etc.) that are not covered elsewhere in the bug tracking form.
- If you expect the reader to have any trouble reproducing the bug (special circumstances are required), be clear about them.
- It is essential keep the description focused.
- The first part of the description should be the shortest step-by-step statement of how to get to the problem.

Writing the Report: Eliminate Unnecessary Steps

- It’s not always obvious what steps can be dropped
  - Look for critical steps -- Sometimes the first symptoms of an error are subtle.
- When shortening the list, look for:
  - Error messages
  - Surprising timing
  - Changes in sounds
  - Display oddities
  - Devices surprisingly in use
  - Debug messages
- Try to eliminate almost everything except the critical steps

Writing the Report: The Headline

- The most important part of the report
- Only thing to appear in summary listings
  - Often truncated
- Executives will often read detail *only* if the headline is compelling

Writing the Report: The Problem Report Form (1)

- A typical form includes many of the following fields
  - Problem report number: must be unique
  - Reported by: original reporter’s name
  - Date reported: date of initial report
  - Program (or component) name: the item under test
  - Release number: like Release 2.0
  - Version (build) identifier: like version C or 20000802a
  - Configuration(s): on which the bug was found
  - Can reproduce: yes / no / sometimes / unknown.
  - Severity: assigned by tester.
  - Priority: assigned by developer/project manager

Writing the Report: The Problem Report Form (2)

- A typical form includes many of the following fields
  - Report type: e.g. coding error, design issue, documentation mismatch, suggestion, query
  - Customer impact: predict actual customer reaction (such as support cost or sales impact)
  - Headline problem summary: 1-line summary of the problem
  - Key words: use these for searching later, anyone can add to key words at any time
  - Problem description / how to reproduce it: step by step reproduction description

Writing the Report: The Problem Report Form (3)

- A typical form includes many of the following fields
  - Suggested fix: leave it blank unless you have something useful to say
  - Assigned to: typically used by project manager to identify who has responsibility for working on the problem
  - Status: Tester fills this in. Open / closed
  - Resolution: What was done to the bug. The project manager usually owns this field.
  - Resolution version: build identifier
  - Resolved by: developer, project manager, tester (if withdrawn by tester), etc.

Writing the Report: The Problem Report Form (4)

- A typical form includes many of the following fields
  - **Resolution tested by:** originating tester, or a tester if originator was a non-tester
  - **Change history:** date-stamped list of all changes to the record, including name and fields changed.
  - **Comments:** free-form, arbitrarily long field, typically accepts comments from anyone on the project.
  - **Closing comments:** (why a deferral is OK, or how it was fixed for example) go here.

Writing the Report

Exercise 8.3:
Defect Reporting

Improving Defect Reports By Review and Editing

- Some groups distinguish:
  - ***Submitting*** a defect report from
  - ***Opening*** the change request to fix the defect
- A senior tester reviews submitted defects before marking them as open and assigning to the developer.
  - If there are problems, the editor takes the bug back to the original reporter.
- This editor might review:
  - All defects.
  - All defects in the editor’s area.
  - All of the buddy’s defects.
  - But don’t overburden the editors. Reviewing takes time.

Exercise 8.4: Editing Bugs--Practice at Home

- Go through your bug database and find some bugs that look interesting
  - Do an initial review of them
  - Replicate them
  - Revise the descriptions to make them clearer and more useful.
- Assignment:
  - Give two improved bugs to a co-worker
  - Review two improved bugs from a co-worker
  - Compare notes

Optional Review Exercise 8.5: Change Requests

- Describe the purpose of a change request
- What makes a change request effective?
  - In analysis?
  - In presentation?
- How do you approach nonreproducible defects?
- What did you learn from trying it?