designwest Embedded Systems Conference 2013, San Jose CA
Tutorial # ESC-111, Monday, April 22, 2013, 1:30 PM - 5:30 PM

“Practical and Fun Lessons on Testing during Software Development”

by Dave Nadler
Nadler & Associates – Consultants
www.nadler.com (Dave.Nadler@Nadler.com)

Overview

The cost to fix a bug increases exponentially as the distance from coding to fix increases. By reducing this distance, integrating testing tightly into development saves time and money. Yet ask an embedded developer about test-driven development or using test tools during development, and you'll often hear "it's too expensive", "we don't have time for that", or "it takes too long". When you hear any of these answers, make sure to ask if the developer regularly delivers on-time, on-budget, and with low bug-count...

Pulling testing forward into the development process reduces the cost of development to the first release (in addition to reducing life-cycle cost). Integrating testing into development is often a culture change; thus to be successful requires:

- Recognizing patterns of development difficulties easily remedied by integrating testing,
- Explanation and communication so the entire team internalizes the benefits to everyone, and
- Use of simple, appropriate, cost-effective tools and techniques.

This training course walks through real-life examples where lack of early testing caused serious trouble, and how integrating testing tightly into development got things on track. Working real-life examples will help you quickly recognize and explain patterns that cause trouble, and gives you tools to avoid repeating these common mistakes. The paper below summarizes the major points conveyed in the class: it highlights cost issues, defines “integrated testing”, discusses cultural problems, and reviews techniques for effective integrated testing.

Why Costs Increase Exponentially with Time from Coding to Fix

When a bug is fixed immediately after coding, normally the fix is implemented by the original developer, using the testing environment and tools used for development of the code containing the bug. This testing may be completely ad-hoc, stuffing variables in a debugger or worse, but at least the code and test are current in the developer's mind. Important reasons bug-fix costs escalate as time elapses include:

- The test environment may not be easily available (especially if it was ad-hoc or requires a special hardware setup).
- The code is no longer current in the developer's mind.
- The original developer is often unavailable, adding learning time for a new developer.
- The development team and/or tooling may no longer be available.
- Details about design decisions, code structure, and pitfalls may no longer be available.
- Ongoing development during the delay period means the problem may be hard to reproduce, and the fix may need to be applied to multiple source branches (ie last release and current development head).

Copyright © Dave Nadler 2013, redistributed by UBM with author's permission.

- If the product is released, customer support costs are added.
- For some products, a recall may be required for updates.

What is "Integrated Testing" ?

"Integrated Testing" means:

- Testing is highly automated and repeatable.
- Testing is tightly integrated into software development such that developers can run tests easily and often <u>during</u> the development process.
- Most testing is run in a simulated target environment on a workstation, and does <u>not</u> require using a target embedded platform.
- Ideally, most tests are created prior to system design, and certainly prior to coding.

Tests span unit tests to system tests, and always include system "smoke tests" (described below). As new problems are discovered, additional tests are added to ensure a fix and prevent regressions. Ideally, test runs are integrated into continuous build tools so regressions are caught immediately.

Because software is developed and tested primarily on workstations, software development can commence before target hardware is available, and development can continue without all developers requiring target hardware and associated tools.

"Integrated Testing" does <u>not</u> mean no separate QA or testing engineers ! It does mean that QA works tightly with engineering to ensure they have the test cases required, such as helping to define Test-Driven Development (TDD) test cases during detail design. Critically, developers work closely with QA to ensure that the code is readily testable on workstations as well as the target – software testability (discussed below). Integrated Testing mitigates problems with an isolated QA department and a "throw it over the wall to the QA ghetto every once and a while" mentality.

Why Costs To First Release are Lower with Integrated Testing

Without integrated testing, the delay between writing code and testing it adds cost as summarized above. Techniques to reduce this time (and to reduce bugs and cost) include:

- TDD. Tests are usually specified/implemented by the primary developer, but may be assisted by QA (a QA engineer may do a better job of identifying boundary and exception conditions to test).
- Code review. On average, the cost of code review (including fixing problems found during review) is less than the cost that would be incurred by finding and fixing these problems later, due to the cost of repair increasing over time. Performing code reviews thus <u>reduces</u> costs, and brings the additional benefits of improved code quality/maintainability and additional developers familiar with the code.
- Scripted tests, often using record-playback tools (discussed in detail below).

Other integrated testing benefits that reduce cost to first release include:

- A project's initial code structure often needs refactoring. With integrated test, refactoring proceeds more quickly (no QA cycle delays), and thus costs less.
- Its easier to test more paths through the code in a structured test environment (e.g. exception paths), so bugs in non-primary paths can be caught earlier with lower cost.
- Because software is developed and tested primarily on workstations, software development can

Copyright © Dave Nadler 2013, redistributed by UBM with author's permission.

commence before target hardware is available, and development does not require all developers have access to complete target hardware and associated tools. This way, software can be completed sooner and with less risk.

Overcoming Cultural Objections

Some engineering organizations have cultural difficulties with integrated testing. For such organizations, adopting integrated testing can only happen when all parties internalize the benefits. In the class, real-world examples highlight common patterns of failure without integrated testing, and illustrate benefits from changing to integrated testing, but here we'll just summarize the key points. For engineering managers, key points to internalize include:

- **Costs, time, and risks to the first release are reduced**.
- The product we initially release will likely have fewer bugs.
- The code produced will be less expensive to maintain in future.
- Testability plus low bug and rework count must be an explicit objective for developers, and must contribute to their performance and salary reviews.

Key thoughts to help engineers internalize the benefits include:

- TDD provides real time feedback that what is being developed works, and code review catches problems early as well (instead of waiting weeks/months for QA cycle).
- I'd rather have a colleague find my bug than a customer. Codes review with colleagues, and working closely with QA to create testable code both increase the chance of shipping a high quality product that reflects well on engineering (and my next bonus and raise).
- As a professional engineer, I want to deliver code I would be happy to receive, which includes making it easily testable and maintainable.
- Integrated testing will allow me to make changes with less fear and fewer secondary bugs.

For QA engineers, key benefits are:

- More time is spent creating useful tests (as opposed to rerunning tests).
- Less time is spent reporting the same bugs when you can give an engineer a test showing a bug and the engineer can run the test to verify the bug-fix.
- Working tightly with engineering results in better, easy to use, tools for testing, resulting in better test coverage.

What is “Software Testability” for Embedded Software ?

Testable embedded software is easily testable in both a simulation environment as well as target hardware, both at the component level and especially the system level. This has two important implications:

- Almost all code builds for a workstation as well as the target hardware. This means hardware and target-platform dependencies are properly isolated, and target-specific facilities stubbed or mocked.
- **All** inputs are controlled in the simulation environment as well as potentially on the target hardware (with target software in test mode).

Inputs often not properly abstracted and encapsulated, and thus impeding testability, include:

- system time and date (for example GPS time, posix <time.h>, SYSDATE in SQL queries)

Copyright © Dave Nadler 2013, redistributed by UBM with author's permission.

- passage of time (control of elapsed time)
- event stream from user interfaces
- hardware interfaces and their event stream

Stubs (mocks) must provide easy methods for the simulation environment to control these inputs and step through time. Less obvious is that these same inputs can be similarly simulated for effective testing in the target environment, by bypassing hardware inputs (examples covered in the class include an aircraft collision-avoidance product).

Simulation testing focuses on general processing and explicitly ignores issues you must test on the target, especially resource consumption/contention and timing/performance.

Environment Simulation and Scripting Techniques

For complete systems or major subsystems, simple scripting can be an easy way to do testing.
A script typically is a simple ASCII text file, with one command per line, and no conditionals or branching. Commands typically include:

- input commands set an input parameter, simulate an external event, or set simulated time
- output commands dump the state of the system under test, for example a screen image or the value of important state variables
- comment commands for test documentation (may include commands to display test narrative in a GUI window)

Input == Output ! To support simple regression testing, log the generated output in-line with the input test script. Make sure all program output generated during a test run is preceded by a "comment" command indicating a program-generated output line. Discard program-generated comments immediately on input during testing, and the output log should exactly match the input. Now you can use a simple file compare to detect regression failures !

```
! A line starting with a "!" is a comment (echoed by simulator).
! Lines starting with ";" are program-generated comments which will
! be discarded on input to simulation.
d s ! Dump Screen command outputs LCD screen image with leading ";"
;    ILEC  SN10
; =Test  Software=
; =Not For Flight=
; Copyright @ 2013
;     Nadler &
;    Associates
;
;  Version 2.38x1
! Running this script through the simulator produces output == this input
```

Figure 1: Example test script where input == output

In the class, we discuss the example above (from an avionics product), and an example where scripting takes the form of NMEA sentences.

Copyright © Dave Nadler 2013, redistributed by UBM with author's permission.

Record-Playback Tools

Record and playback tools are incredibly helpful. If you log inputs in the format of simulation commands, this simple log of inputs creates a test. Logging to create test scripts can be effective both in simulation and target environments. This is especially helpful when:

- You need to retrofit testing into an existing system
- You are trying to prevent regressions during modifications
- Tests have not already been constructed for the code you must update
- You are working on a system with major subsystems daisy-chained together, and you want to be able to use output from one subsystem to generate test cases for the next in the chain and ease system integration
- Where developers are reticent to develop tests first, record-playback in simulation can be used to generate tests; beats stuffing variables into a debugger and watching the results !

Examples covered in the class include an aircraft collision avoidance system, an aircraft flight computer, and an automated toll collection system.

Unit Test Tools

Unit tests are especially applicable for algorithmic modules which lend themselves to TDD. They are only part of the picture; you still need system tests. A few tips:

- For module-level tests, keep test code in same source file, for example TDD main compiled with `#if` symbol.
- Try to minimize the number of artifacts (separate test input files etc.), and keep them <u>all</u> in source control.
- embUnit for simple “C” tests, or choose from many other available simple frameworks.
- Don't be frightened of C++ unit test frameworks !

What is a “Smoke Test” ?

A smoke test is a broad but shallow sanity test intended to ensure basic functionality of all major subsystems. Developers should run smoke tests plus detailed tests for the subsystems they are working on before checking in source code. Better yet, if everything is automated, run the entire test suite !

When faced with updates to legacy code, start by developing a smoke test...

But, We Can't Boil The Ocean !

You can't test everything. Deciding what tests are needed is a bit of an art-form best guided by experience, but here are some tips on what's important:

- <u>Always</u> write unit tests for algorithms (math functions and the like) – TDD is ideal here.
- Concentrate testing efforts on areas with the most churn, whether from bugs, specification changes, refactoring, or other issues.
- Always create system smoke test(s) with broad coverage to catch regressions early.
- Don't spend all your testing resources in one corner; make sure to look broadly across the application.

Copyright © Dave Nadler 2013, redistributed by UBM with author's permission.

- Avoid test duplication to minimize future work when required behavior changes.
- Automate as much as possible so that tests are continually run, and ensure that test failures are immediately addressed. Unless you do this diligently, you'll end up with a pile of failed tests that are ignored ("test rot"). Then when you really need them, you'll have a big cost to fix them.
- Try to ensure tests are insensitive to minor formatting and display changes.

Recommended Reading

Test Driven Development for Embedded C, by James W. Grenning (well-written TDD discussion)
Working Effectively with Legacy Code, by Michael Feathers (great tips on testing in general)
Code Complete, by Steve McConnell (great general coding tips, light on testing)
Making Software: What Really Works, and Why We Believe It, edited by Andy Oram and Greg Wilson

Author Biography

Dave Nadler has decades of consulting experience building products, helping improve engineering practice, and getting challenged projects back on track. Dave has designed or contributed to avionics products now flying around the globe. His career has also included leading software development, systems, and networks for a major international financial market data provider (a team of ~250), and working as a principal engineer developing a couple of automated electronic testers. Dave worked on his first embedded system back when dinosaurs roamed the earth in 1978 and is still at it...

Contact: Dave.Nadler@Nadler.com
Nadler & Associates Web Site

Dave Nadler in his Antares 20E electric-powered glider

Copyright © Dave Nadler 2013, redistributed by UBM with author's permission.