CREATING QUALITY NEWTON APPLICATIONS

Peter Murray, Newton 2.0 Quality Lead

INTRODUCTION

Quality makes a difference. It makes a difference to your customers and ultimately to your bottomline. But, it goes beyond just making sure you find and fix bugs. That's a given. You can't test quality into your application. Quality is meeting or exceeding customer expectations in all facets of the user experience. Aspects such as simplicity, robustness, and utility are critical to your success.

Whether you're a large or small developer, a strong quality assurance program can add a lot of value to your product. Besides giving you the confidence to ship a well-tested product, SQA can act as the focal point for customer feedback upon which to build increasingly customer-driven products. Qualifying software on the Newton is not much different from doing so on other platforms. There are three basic building blocks which comprise a good Newton SQA program:

- an understanding of software development
- a foundation of general SQA practices
- knowing the specifics of testing on the Newton platform

SOFTWARE DEVELOPMENT

Your product development should be planned and you should execute to that plan. Marketing should define the target customers and gather and relate their requirements. Engineering should respond with a description of how those requirements will be met by the software, both at the human interface and low level. Based on the marketing and engineering requirements, SQA develops a test plan defining how testing will be done to converge on a high quality product. All of the teams must agree on what the end product will look like, who it's for, and what problems it solves. Never get caught up in the details of the project so much that you lose sight of the big picture.

Software development is an iterative process. What does it mean for your software to reach the alpha, beta, final, and golden master milestones? What metrics do you choose to measure your progress and by what milestone criteria do you judge the status of the project? These questions are largely dependent upon the goals of the project. Choose metrics and criteria with care. Minimalism is better than weighing your project down with bureaucracy.

A well planned product development process provides the guidelines for getting the work done, but software development is intrinsically dynamic. Make sure your day to day decisions still reflect the project goals and customer requirements.

GENERAL SQA PRACTICES
The following concepts underlie any good SQA program.

• *Prioritized Test Coverage*
Since you can't test everything, prioritized testing is key.

- What absolutely has to work from the customer's perspective to make your product successful? Concentrate more of your resources on those features and less on others based on their relative importance.

- Coverage should be partially driven from development engineering.
  + Ask the development engineers about the areas of their code and hacks that concern them the most.
  + Look for interdependencies and assumptions? These might be bug rich areas.
  + Test any error checking. Simulate the errors and validate the error handling.
  + Development and quality engineers should collaborate on creative ways to break the code.

- Weaknesses might be uncovered in code or design reviews.

- Inexperienced programmers might require more test coverage, especially in the system specific features.

• *Shared Knowledge*
As much as possible, development and quality engineers should share a common understanding of the features of the software and the underlying code paths.

• *Scientific Method to Isolate Bugs*
The best quality and development engineers have internalized the formal scientific method and subconsciously use it when debugging. "The real purpose of the scientific method is to make sure Nature hasn't misled you into thinking you know something you don't actually know."[1]

1) statement of the problem
2) hypotheses as to the cause of the problem
3) test cases to test each hypothesis
4) predicted results of the test cases
5) observed results of the test cases
6) conclusions from the results of the test cases

---

[1]Pirsig, Robert. Zen and the Art of Motorcycle Maintenance. 1974. Page 94.

• *Managed Risk*

Most decisions related to qualifying and shipping the product entail varying degrees of risk. The challenge lies in correctly analyzing the risk and taking steps to minimize it. Risk analysis should be undertaken seriously and any judgments should be based on facts not conjecture. Always try to think of creative ways to minimize the risk. Here are some examples:

- Code changes

  If there are code changes late into the project or near critical milestones there are some obvious ways to minimize the risk of the change:

  + the change should be code reviewed by another development engineer
  + the developer making the change should write a release note about the change, its impact on other areas of the system, and important tests to run.
  + SQA should verify and test the change and any related areas

- Bugs

  Since you can't test everything or use the product in every way that a real customer might, you won't find every bug, and you'll almost certainly ship with known bugs. Make sure you prioritize the bugs to fix based upon their importance to the customer. If a bug happens in only obscure cases it may not be worth fixing. Fix the bugs and usability problems that will affect the majority of users, or which might cause bad press. Sometimes you'll guess wrong. Keep track of the bugs you defer and see if real customers report them, then you escalate them to fix in the next version.

• *Customer Feedback*

Improving product development is based on a feedback loop. There are three main programs for incorporating customer feedback at different parts of the product cycle: user testing, beta testing, and (uh oh) customer support.

- User Testing

  Bringing in target users to give feedback on your product early in the development cycle can give you data about the intuitiveness of the design and validate whether your feature set is the right one. SQA engineers can also improve their tests by watching how real customers use the product. Fresh perspectives are always valuable, because the development team becomes too close to the product.

- Beta Testing

  Software used in real world situations often provides some of the best feedback and bugs. During the 2.0 project most people on the Newton team carried around flash ROM MessagePads with the latest software. A lot of the good bug reports came from people using them in meetings.

That was the difference between 1.x and 2.0 -- you can actually take notes in meetings.

- Customer Support
Product development doesn't end when you ship your product. Customer support quantifies customer reaction, and based upon call frequency you can prioritize bugs to fix in future revisions. Feature requests provide a source of good ideas from people actually using the product. The development of Newton 2.0 was largely driven from customer requests and usability problems, along with inspired design work from Capps et al.

SPECIFICS OF NEWTON TESTING

Testing a Mac, Windows, or Newton application is all essentially the same. Take your software development and SQA background and overlay the technical details of the platform. When testing a Newton application, know how the Newton works and how it's architected. The best sources of information are the user 's manual, Newton Programmer's Guide (NPG), and Newton Human Interface Guidelines. Applications written for the Newton plug into a layer of system services. So, besides testing the features of your software in isolation, you need to test the areas where your application intersects with these system services.

Newton System Services

Assist	Find	Filing	Recognition	Dial Assist	Backup/ Restore	Ink	Keyboard
ListPicker	Locale	Sleep	1.x/2.0 Data Conversion	Styles	Clipboard	TextEdit	Help
Storage Cards	Undo/ Redo	Dictionaries	Backdrop	Landscape	Fonts	Sound	Mail
Printing	Faxing	Beaming	Alarms	Worksites/ Persona			

Some of the important things to test are:

• Make sure the recognition view flags are set for the right kind of input.
• Test data storage and filing on an unlocked and locked card and internally to validate the soup code.
• Erase your application from extras to test its removeScript and optional deletionScript. Make sure the application removes itself from various registries to avoid the "grip of death."
• Eject a card with application data on it. Make sure the screen is refreshed.
• Test the differences when your application modifies a built-in prototype.
• If your application supports landscape, make sure all the dialogs fit.

• Test your application's use of Find - especially if it does something different like highlight the entry.
• Make your application the backdrop and see if it behaves correctly.
• If you link in custom dictionaries, test them in the various fields.
• Make sure soup change and other notifications get intercepted by your application.
• Optimize your program's use of frame and/or system memory.
• Make sure your application doesn't create global functions unless absolutely necessary.
• Test with the latest version of the system update. (The easiest way to remove a system update is to remove all batteries and short the backup battery terminals with a penny).
• Make sure your app uses screen relative bounds to remain compatible with different screen sizes.
• Make sure the human interface of your application conforms to the guidelines.
• Delete your soup from the storage folder and see if your app handles it gracefully.
• If your application supports both 1.x and 2.0, test on both platforms.
• Test how your app reacts to its own data on a read-only 1.x and locked 2.0 formatted card in a 2.0 system.
• Use the Newton Keyboard with your app and make sure the tab order for views is correct.
• Test any application interfaces exported for public use.
• Test all peripherals you support.
• If your app runs on 1.x and 2.0 make sure it uses gestalt to test for the existence of Newton features.
• If your app modifies the behavior of a built-in app, then research existing applications and see if anyone else is doing what your doing and document any incompatibilities.
• Make sure your application uses documented APIs or DTS approved methods, for example, when accessing built-in application soup data. This will avoid incompatibilities in the future.

Applying the background and Newton specific information, the following five steps provide the framework for testing a Newton application through its development:

1) Create a very detailed, hierarchical feature list for the product based upon documentation, prototypes, and communication with development engineers. Every feature that a user can access should be represented, but don't go overboard on the details. Keep the feature list updated to reflect the current state of the product.

2) From that feature list create test cases. For each specific feature there will be one or more test cases. Your base functional test cases should only be testing

one thing. That way you can more easily isolate bugs and relate test cases to bugs. Be sure to go through the list of system services and create test cases for the way your application interacts with the rest of the Newton.

> *example*: A feature in the Newton 2.0 Time Zones application is the ability to delete cities. Some test cases to derive from this feature would be:
>
> a. delete a built-in ROM city
> b. delete a city added by the user
> c. delete a city that was specified as a worksite
> d. delete a city and tap undo
> e. delete a built-in ROM city then delete the Time Zones soup from the storage folder in extras

3) Identify a subset of your test cases as "quicklooks". These test the major areas of product functionality. When a new build of the software is handed off to SQA you start your structured testing by running through your quicklook test cases. It's more productive to discover that a major part of your application is broken in the first 30 minutes of your test cycle than two days into it.

4) Determine the most effective test cycle time for normal engineering builds and milestone builds. You won't want to run all your test cases for every build. It's more effective to run all the quicklooks and then concentrate testing on the changed areas and bug fixes documented in the build release notes. You will want to run all your test cases at milestones like alpha, beta, and final.

Be sure to devote some time in each test cycle to ad hoc testing. Use your understanding of how your application works and how customers might use its features. If you've ever observed a user test you're immediately struck by the fact that most users don't follow the "rules" -- even when using the "simplest of features". Towards the end of testing 2.0 we placed more importance on ad hoc testing by splitting the test team into two groups: Validators and Exterminators (for lack of better names). The Validators performed the systematic testing of all the product features by running through all of the developed test cases and the Exterminators were exclusively ad hoc testers with free reign across the whole system.

5) As the project progresses and the features of the software coalesce into stability, expand your testing into more stress and boundary conditions. The goal being to increase the robustness of the software in its ability to respond gracefully to unusual conditions. And since you can't test everything in every way that customers will use your product, this is one more thing you can do to minimize the risk of serious bugs in the field. A simple way to stress a

Newton application is to see how it handles a lot of data. For example, the average Newton user probably has 100 - 200 entries in the Names application, but in 2.0 we tested with 1500 and sometimes more.

CONCLUSION

There's no guarantee that you'll ship quality applications even if you follow all of this advice. The glue holding everything together is caring -- the pride in your work and the satisfaction of knowing you did your best. "...care and Quality are internal and external aspects of the same thing. A person who sees Quality and feels it as he works is a person who cares. A person who cares about what he sees and does is a person who's bound to have some characteristics of Quality."[2]

---

[2]Pirsig, Robert. Zen and the Art of Motorcycle Maintenance. 1974. Page 247.