RBCS
TIME TESTED. TESTING IMPROVED.

How to Build Quality Applications

Testing is an excellent means to build confidence in the quality of software before it's deployed in a data center or released to customers. It's good to have confidence before you turn an application loose on the users, but why wait until the end of the project? The most efficient form of quality assurance is building software the right way, right from the start. What can software testing, software quality, and software engineering professionals do, starting with the first day of the project, to deliver quality applications?

The first step in building a quality application is to know what you need to build. An amazingly large number of projects get underway without clarity amongst the project stakeholders about what the requirements are. According to Capers Jones' studies, as many as 45% of defects are introduced in specifications. One working definition for quality is "fitness for use." If we're unclear on the intended uses, how can we build something that is fit for use? Not only do we need some specification of the requirements—whether formal or informal—but we should also conduct a thorough project stakeholder review of this specification to look for defects and to build consensus and understanding.

Another important early step is properly organizing the project. The overall approach to application development is the *software development lifecycle model* (SDLC). There are four main varieties of SDLC in common use today:

1. *Sequential* (also called *waterfall* or *V-model*): In this approach, the team proceeds through a sequence of phases, starting with requirements, then design, then implementation, and then multiple levels of testing. This model works best when you can specify requirements that will change very little if at all over the course of the project. It also works best when you can plan the project with great accuracy, which typically means it's similar to a project the team's done before.
2. *Iterative* (also called *incremental* or *evolutionary*): In this approach, the high-level requirements are grouped together into iterations (or increments), often based on technical risk, business importance, or both. The system is then designed, built, and tested group-by-group. This model works well if you need to deliver the most important features by a rigid deadline, but can accept some features

arriving later. This model can tolerate some change in the plan (often due to uncertainty or change in requirements) and still deliver the key features on time, which is not true of the sequential models.

3. *Agile* (such as *Scrum* and *XP*): In Agile approaches, each iteration is compressed to a short as two weeks. Documentation is minimized and change is expected from one iteration to the next, and within each iteration. Various rules help prevent devolution into churn and chaos. This model works when applied with discipline, and its emphasis on accommodating change allows it to produce results even in rapidly-evolving situations.
4. *Code-and-fix*: This approach is actually the absence of an approach. It involves starting the development of the application without any requirements, without a clear plan, without anything but a deadline, in many cases. This model can only work for the simplest, shortest, and least-risky of development projects.

Now, the first three of these models exhibit significant variation in practice. You should feel free to intelligently tailor the model to your specific needs, but beware of violating certain aspects of the model that enable other features of the model.

With the project properly organized and the requirements clearly understood (whether for the whole project or only for this iteration), design and coding can start. Of course, coding presents not only the opportunity to create great new features, but also the risk that the programmer will create great big bugs. To mitigate this risk, there are three things every programmer should do with every piece of code she writes:

1. *Unit testing:* The programmer should test every line of code, every branch, every condition, and every loop. Higher levels of testing such as system test often touch half (or less) of the code, and any untested code is a potential hiding place for bugs. New tools, both commercial and freeware, make the job of unit testing much easier than it was in the past.
2. *Static analysis:* Even code that passes unit tests can still contain latent defects, maintainability problems, and security vulnerabilities. Static analysis can cheaply and quickly find bugs that would take hours to find and remove during higher levels of testing. The programmer now has a wide variety of tools available to help with this task, too.
3. *Code review:* Once a given unit of code is written, tested, and analyzed, having a walkthrough or technical review of the code among the programming team is a great way to catch most of the remaining bugs and to ensure good understanding of how the program works across the entire team. Studies at Motorola show that as few as three experienced programmers (including the author), following a rigorous inspection process, can find as many as 90% of remaining bugs.

We can be very confident indeed in each unit of code if programmers go through these three steps prior to checking their code into the source code repository.

Even with high quality units, there remains the risk of integration bugs. Integration bugs occur when two or more interoperating units don't communicate, share data, or transfer control properly. To help mitigate integration risk, the project team can use continuous integration. This involves checking in code as it's finished, compiling and building that code together, and running automated tests against the code to check for integration bugs. As with unit testing and static analysis, a variety of tools exist to help with this process now.

When we deliver quality applications—applications that are fit for use—we get to enjoy positive outcomes such as satisfied users and customers, improved reputation, more revenue or resources, and greater job satisfaction. In this article, we've seen that the pathway to delivering quality and enjoying those outcomes starts on the first day of the project and continues to the very end. Good requirements. Proper organization. Quality-focused programming. Continuous integration. And, once the application is ready, we can go through formal system, system integration, and user acceptance testing. If you've followed the steps outlined in this article, you'll be amazed at how smoothly those tests go, and how quickly and confidently you can put a quality application into your data center.

Bio

With a quarter-century of experience, Rex Black is President of RBCS (www.rbcs-us.com), a leader in software, hardware, and systems testing. For over fifteen years, RBCS has delivered consulting, outsourcing and training services to clients ranging from Fortune 20 companies to start-ups. Rex is also the immediate past President of the International Software Testing Qualifications Board and the American Software Testing Qualifications Board. Rex has published six books which have sold over 50,000 copies, including Japanese, Chinese, Indian, Hebrew, and Russian editions. He has written over thirty articles, presented hundreds of papers, workshops, and seminars, and given about fifty speeches at conferences and events around the world. Rex may be reached at rex_black@rbcs-us.com.

