TestRail

Best of the TestRail Quality Hub

Technical Skills for Agile Testers

TestRail

Software testing is a demanding field. Do a quick search online for "software testing jobs," and you'll often find a daunting list of requirements. Of course, analytical, problem-solving, and communication skills are must-haves for software testers. But as organizations adopt agile practices and testing shifts left, these job requirements increasingly include technical skills. We've selected this series of articles from the TestRail Quality Hub to help testers enhance their agile practices and gain the technical skills necessary to thrive in a shift-left environment.

Contents

Technical Skills for Agile Testers

Integrating Testers into Modern Agile

Justin Rohrman, Excelon Development

My experience of agile when it initially went mainstream was that teams continued doing what they already had been doing, just over the course of two weeks instead of two months. Our developers still had a "write the code" part of the development cycle where they made a large amount of the feature changes locally before attempting to spin up a new build, and our test team still spent a majority of their time waiting around for that first build, and then again after rounds of testing when we needed bug fixes.

Teams I see today are at the logical conclusion of agile. Developers take very small feature changes and work on them until they are done — not ready to test, not ready for a demo; "done" as in ready to go to production. A development pair can take a feature request from Jira to production in days instead of weeks.

Where do testers fit into this new, fast-paced workflow?

Pairing, Plus One

The normal development workflow is a single person working on a single change. The developer takes a new card; disappears for some time, maybe coming out occasionally to ask a question; and then, in a few days or a week, has something to commit.

Pairing reduces the number of mistakes and misunderstandings one person can have on their own, as well as the number of handoffs that come with a process where people work on their own. So what if you add a third person to that? Having two developers "pair" with a test specialist further reduces errors and ensures quality.

The developers and test specialist start a change with a "Three Amigos" meeting to review what they are supposed to build and ask any clarifying questions. One of the first things I like to ask as a test specialist working with a developer pair is, "How do we test this thing?" That small question helps us find a place to start, and usually that is a test.

That test might be something at the unit level, something at the service layer or a cuke that runs in the browser, but the initial point is that it will fail when we run it. Sometimes that test is written by the developer, sometimes by the tester. Once the test is written a developer will write the product code that will satisfy the test. We go back and forth in that pattern until we get a gut feeling that we are in a good place to make a new build to explore the changes we have made so far.

One thing that occasionally comes up is a test will feel difficult to build. Either the process of building that test is taking too long, it requires a lot of data setup or the code needs significant refactoring to be possible. Usually at this point we will stop and ask whether we are testing the right thing at the right level in the technology stack. We review what we are trying to test and why, then we make a decision about how to move that test up or down the stack, or even to an exploratory activity, to have a more effective test.

If you were paying attention there, you might have noticed there are large swaths of time when the tester isn't "working."

The Lag

There is a lag in the process, even when a test specialist is blended into a developer pair. Let's take a closer look at the pairing flow I described above.

The flow starts with a "Three Amigos" meeting where we get refreshed on the change. The change is fairly well defined at this point, but we can still occasionally discover missing scope by asking questions. After this, the tester is active when talking about initial test design and jumpstarting the development flow.

Depending on the driver-switching cadence, which I see ranging from switching on every test to not switching through the entire change, the tester might be mostly idle for a while. During this time I ask questions about

programming language features, pointing out typos or the occasional syntax error and asking leading questions about where we are going with a feature. I generally don't see testers with hands on the keyboard until some sort of UI automation needs to be written.

At some point during the sprint, we have a feeling about where the product is in terms of usability, and we create and deploy a new container. When we are ready to explore, I might disengage from the developer pair while I explore the changes so far in a browser, but I still listen and occasionally watch via terminal to what is going on.

Despite being embedded in a developer pair and working in what is close to a continuous delivery process — namely, being release-ready once we create a pull request — there is still a notion of "testers' work" and "developers' work." The tester role in this equation is only minimally effective at the beginning and end of the process. The middle is mostly asking questions and trying to stay engaged in a process we are only minimally involved in.

So, how do we fix this?

The Technical Tester

In my experience, the tester in this scenario needs to develop some technical skill to become more effective. Let me explain with an example:

Instead of having two developers and one test specialist, let's say we have one developer and one test specialist. Rather than the tester asking a question about how we'll test at the beginning of the code change, the tester and developer have a conversation and the tester writes the test. Working on one terminal or IDE, the pair goes back and forth, with the test specialist writing tests or building test data and infrastructure and the developer writing code to satisfy that test.

Each person in the pair contributes to what the other is doing by helping through problems, pointing out mistakes to prevent tedious debugging later, or helping with course correction, but they also each work within their realm of expertise. This back-and-forth flow between two people keeps both parties engaged in the work and aware of what is happening.

When it is time to build and deploy a container, the test specialist will probably talk through the work they imagine needs to be done, then divide that work up between the pair so that both the tester and developer are doing exploration. The tester might take some of the more involved or complicated exploration work, just as the developer might take some of the more difficult programming tasks.

This pair structure of one developer and one test specialist, along with the driver-navigator switching cadence, keeps the power structure more level. There is less a notion of "testers' work" and "developers' work" and more the idea that they are working together on the same tasks to send something important to production in a reasonable amount of time.

Putting It into Practice

Getting to a smooth pairing flow when it isn't already built into the company culture can be a challenge — as can moving from a process with strong divisions between roles and handoffs at each break point in the development flow. If you are new to pairing, try doing it with two developers and a test specialist to get into the flow. Once you have that down, you can maximize efficiency by asking a high-performing test specialist if they want to get a little more involved in the pairing process.

Just as with agile, this process will need assessment and tweaking when you notice areas where the product could be better or your team could be more efficient. But also as with agile, when you encourage collaboration and let the workers direct themselves, you should end up with higher quality and happier team members.

Technical Skills for Agile Testers

3 Tips to Become a Technical Tester

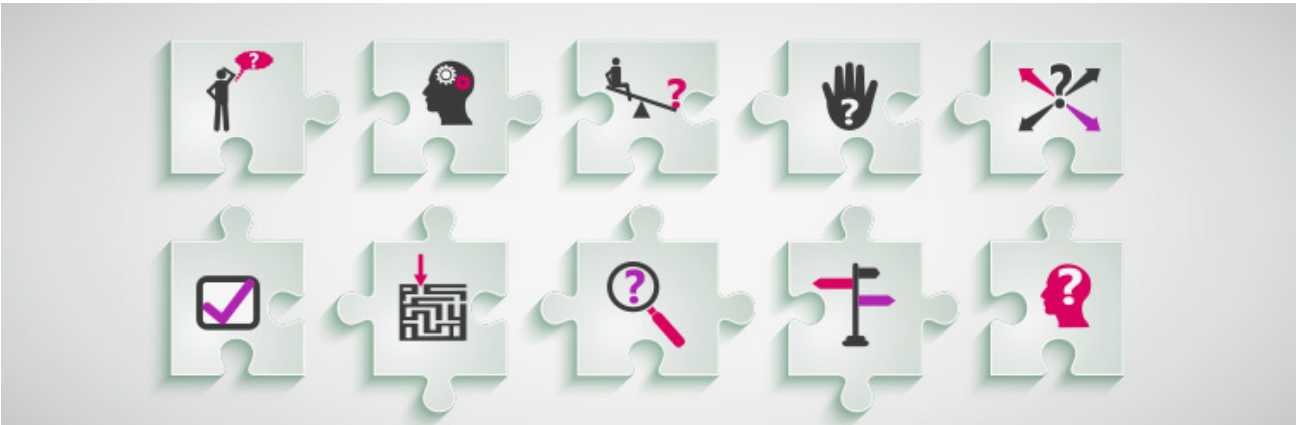
Justin Rohrman, Excelon Development

There seems to be demand for technical skill everywhere I go. Conferences that used to draw talks on how testing is influenced by the social sciences and philosophy are slowly getting stacked with talks on how to work as part of a developer pair. Companies that used to have test specialists to explore the product after development now want test specialists who can read code fluently and write small bits here and there.

Whether technical testers are the right solution to today's testing problems is an interesting question. But regardless of the answer, technical skill is a quality highly sought after in testers these days. Spending nights and weekends studying code is a privilege few can afford. I don't see many companies offering money for training, either.

So, how is a tester supposed to develop this new skill set? Here are three areas testers can focus on to grow their technical skills.

1 Find a Real Problem You Want to Solve



My first experiments with technical skills were driven by the fact that my co-workers were talking about automation—nothing more. We were working in a mostly waterfall environment, and like in all waterfall development flows, regression testing was taking too long. The solution that our manager eventually decided on was using UI automation to run tests every day so testers didn't have to at the end of the release.

In practice, what this means is we took a bunch of non-programmers and thrust them into a software development project, without much guidance or direction. This project didn't end up working out very well. Slow regression testing was a symptom of several

other problems that were upstream from testing work. UI automation is a solution, but maybe not the ideal one for the situation we were dealing with.

One actual problem we had was the amount of time it took from when a build started to when it was usable on a test environment. A few times each day a developer or test specialist would kick off a build in our continuous integration system. That involved making sure all the code that needed to be in the build was checked in, then clicking a button to get things started. The CI tool handled running unit tests, reporting and creating the installation file for our product. Our build took a couple of hours because tests were integration tests masquerading as unit tests. After the build completed, someone had to FTP the installation file to the right server, run any database migrations, run the installer, then pray to the gods of software that all the services started up correctly on the first shot. During that time, most of the team was "writing documentation"—a polite way of saying "waiting around for something to test."

I wanted to remove the amount of hand-holding involved in this process. My very first technical project was writing a batch script that polled the build system for a green build, moving the file to the right server, and then running a small smoke test built in WebDriver that took about five minutes to complete. Once the smoke tests were green, the Bash script would send out an email to the test team letting them know a new build was available. If the smoke test failed, the Bash script sent an email to a few developers to let them know something was wrong with basic functionality. This script took me several Google searches and a few days to write, but it potentially saved an hour or more of time dealing with deployment every day.

There is a lot of technical work like this that is peripheral to the work of testing software. Understanding continuous integration, source code repositories, containerization, build and deploy pipelines, shell scripting and even basic programming knowledge will make much of your testing work easier, faster, or better. If you want to develop technical skill, find a meaningful problem to solve.

2 Pair with Developers

Has anyone ever asked you why testing is taking so long? The reason is usually not the actual testing work. There are a lot of activities—talking with developers, environment maintenance, building test data, reading and research, and isolating and reporting problems—that all add time to the process. Each time you switch from designing and performing experiments on software to one of these other activities, you insert a small pause in your test process.

I like pairing with developers most of my day partly because it removes the need to pause while testing. The other part is because of the natural skill transfer that happens during long-term pairing.

A normal day looks something like this for me right now. My project team meets in the morning to take a look at the current work in flight, we pick pairs for the next day or two, and then we start working. Starting work means selecting a new code change and then having a Three Amigos meeting (dev, project manager and test specialist) where we review the change and talk about what it means to be done. After that, the developers and I figure out where to start working on this change by asking how we can test.

That question is more complicated than it sounds. We could build a unit test, a test against the view using a tool like Jasmine, or a test in WebDriver that runs through a behavior-driven development framework. Once we make a decision about the right layer to attack first, we have to talk about what to test and how we want to approach the problem. Sometimes the developers write the test code, or if I feel comfortable with it, I will ask to drive at that point. We move back and forth between test and production

code. If anyone on the team, including me, sees something they want to do differently, they can ask to drive and write a little code.

(If you are about to say this is developer-centric and probably shallow testing, OK. There is some truth there. But this flow creates much better quality code than any development process I have had the pleasure of working in.)

Once we are done implementing the code change, the three of us explore the product in the browser in a test environment. As the local test specialist, I take a much stronger role by driving most of the work while they ask questions and make observations.

I started this project completely unfamiliar with the technology stack, a few years out from my last programming project. Today, I am able to write small tests at a few layers in the product, read code fluently enough to see and fix problems, and contribute to code reviews. I am nowhere near close to being comfortable calling myself a programmer or tool builder at this point, but I am very comfortable with making useful additions to the development process while it is happening.

3 Don't Major in the Minors

One popular question I see on email and chat threads among software testers is, "what programming language should I learn?" Most of these people are either at a transitional point in their careers and think learning to write code will help them get a job, or they are wondering where they stand with their current employer.

Once this question is asked, there is a blast of answers with near-religious feelings on what the tester should be learning—or, more commonly, what they should not be learning. "Don't bother with Java, you'll end up typing too much." "Don't waste your time with JavaScript; it is trendy and a new library comes out every 10 minutes, so you'll never keep up."

Realistically, none of the very strong opinions matter at all at this time. The point of diving in right now is to get familiar with how programming works and how you can leverage that as a test specialist—and that goes for any new technical skill you want to delve into. Focus on what is important: how the technology works, how it will help you perform testing work, and why you want to use it.

Conclusion: Just Get Started

Becoming a technical tester can be intimidating for people who have not done much more than work with a browser. There are a lot of options, and everyone you ask will have strong opinions on where you should start and why.

Focusing on three basic steps—finding a real problem to solve, collaborating with someone on your team, and learning to ignore details you don't immediately need—will help you on the path toward becoming a more technical tester.

6 Technical Testing Skills That Aren‘t Automation

Justin Rohrman, Excelon Development

When testers hear the words "technical testing", our minds immediately jump to "automation." Testers are often unaware that many highly valuable technical testing skills have nothing to do with automation. It is invaluable for any tester to be able to skilfully use the different tools that help build software. There are hundreds of tools, free and paid, that can be in the tester's arsenal; here, we look at six of the most popular tools and skills that aren't automation.

1 Using Version Control

Version control systems, like Git and SVN, are ubiquitous among software developers, with good reason. Using version control means that software can be worked on, simultaneously, by multiple developers, without them stepping on each other's toes. If something goes wrong with a deploy, the software can easily be rolled back to the last working version until a fix is generated.

What does version control mean for testers? Testers who know a version control system like Git, can work with developer branches of code and start performing exploratory tests before that code is merged further upstream.

For example, if a developer has made a branch for a new search feature, a tester who knows Git can get notifications about the status of the developer's work. In some circumstances, testers who have sufficient test environments can pull a feature branch into a test environment before it is merged into master, and perform tests against the branch. This is a practical way to

"shift left", and give testing feedback earlier in the software development life. There are many blogs, videos, and resources to learn Git. One helpful, free resource is ProGit.

2 Viewing Log Files

Log files are stores of data waiting to be mined. Accessing logs can seem difficult at first, but there are several different types of log files, and many places from which to access them. Info logs, error logs, and debug logs are just some different types of logs that can be useful for testers. Info logs keep track of information such as different services starting and stopping on the product, or the processes that happen during that time. Error and debug logs have information including diagnostic messages, to help developers and testers diagnose potential issues. Some error logs may be accessed through the web console, while others will require access to internal systems where developer or production code is running.

3 Using the Browser Developer Console

An overlooked and underused tool in testing is the developer console, available in all web browsers. For testers who are developing UI automation tests, this tool is necessary to identify DOM elements. Sometimes, building a CSS expression for difficult-to-locate elements is necessary and developer tools make this possible. By investigating logs and adjusting JavaScript, testers can investigate bugs and provide detailed descriptions of the problems they see. There are many tutorials available that help beginners learn to use the console methods that are useful to conduct a deep investigation of a website.

4 Using Accessibility Testing Tools

Websites should, ideally, be accessible to everyone. Testers who work for regulated industries may be required to do some accessibility testing. Developers who do not face a disability, such as vision or hearing impairment, may not have accessibility concerns at the forefront of their mind. Fortunately, there are some great tools that are easy to implement to check for basic accessibility requirements. The W3C markup validation service is a fast and simple way to validate all the markup on a webpage to ensure that it meets basic W3C accessibility standards. In addition, Chrome offers some free browser tools, available on GitHub, to use in developing and checking websites for accessibility.

5 Using Virtual Machines

Virtual machines and containers are helpful tools to test multiple operating systems or multiple versions of software. Often, testers are required to verify that software will work for users with a wide range of operating systems and browsers, but having several computers with different operating systems may not be realistic for some companies. Additionally, most computers will only support one version of each browser, but some companies may support older browser versions. Virtual machines are often a good solution for companies that support older machines or browser versions. Virtual machine software like Virtual Box is free, and companies like Microsoft even offer VM images of their IE and Edge browsers for download and test.

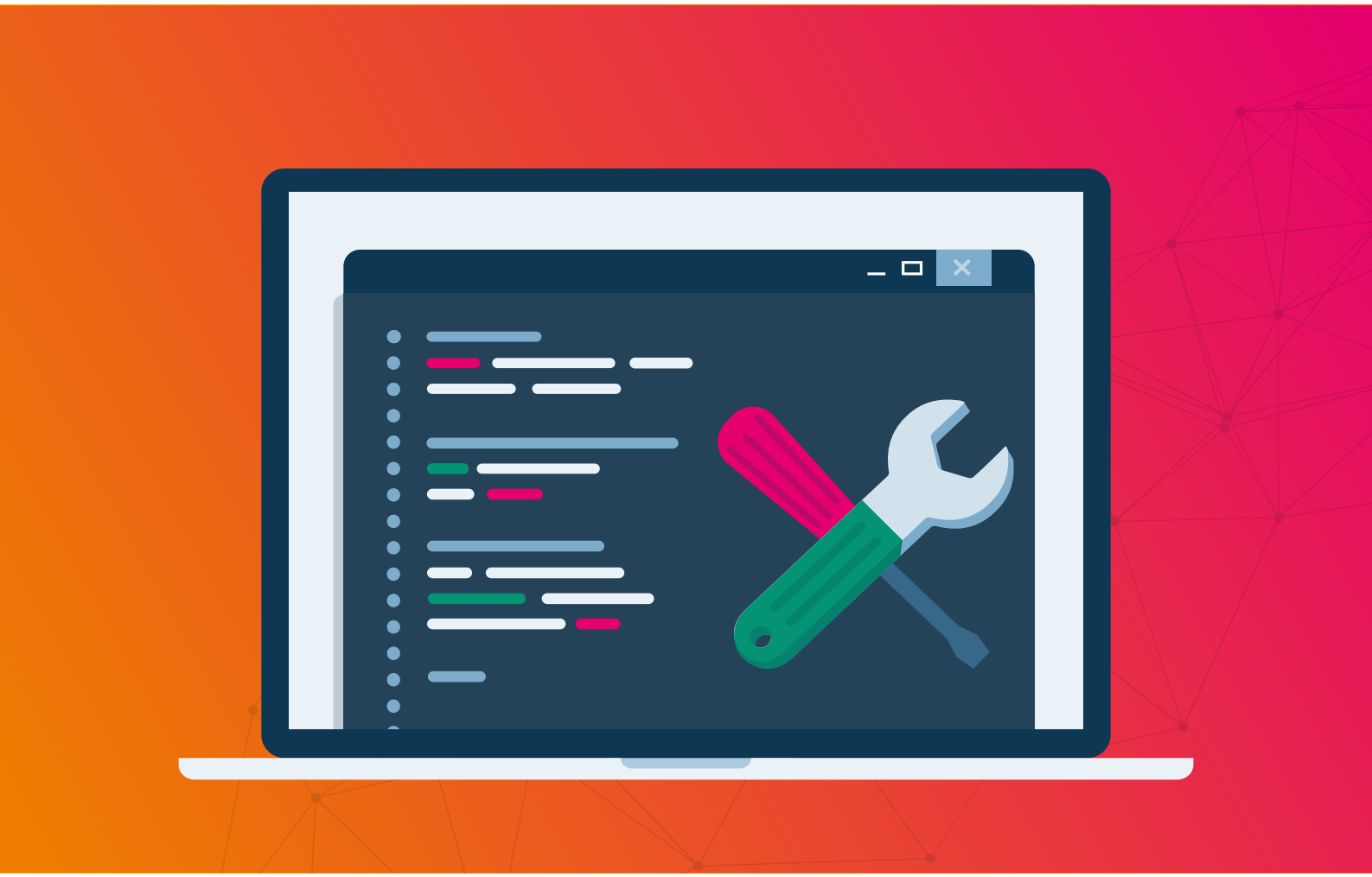
6 Applying Telemetry and Analytics

DevOps skills and tools are your window into the backend of the product. Often, the DevOps team has access to monitoring tools, and monitor performance, user activity, errors in production, and other behavioral changes in the website that might indicate security concerns. Some of these changes, for instance, include how much traffic is coming into the site at any given time. Very high spikes in traffic can indicate potential problems allowing a denial of service (DDoS) attack to occur.

How can you use this DevOps information in your tests? User activity monitoring can indicate what features of the website are updated, changed, or require more robust attention during regression. Error logs from production can be quickly traced back to issues which may require testing, and subsequent fixes. Security issues can be mitigated earlier in the testing process if some security tools are used. ZAP Proxy tool is a free tool provided and maintained by OWASP. With the tool, you can scan your site for security concerns, helping prevent these issues from reaching production.

Tools applied in appropriate situations enhance problem solving by providing more information, more efficiently, with less test effort. Testers who regularly employ technical skills to enhance their testing, can spend more test cycles investigating interesting problems, and less time doing shallow tasks. Check out the resources below to learn the skills highlighted in this article below:

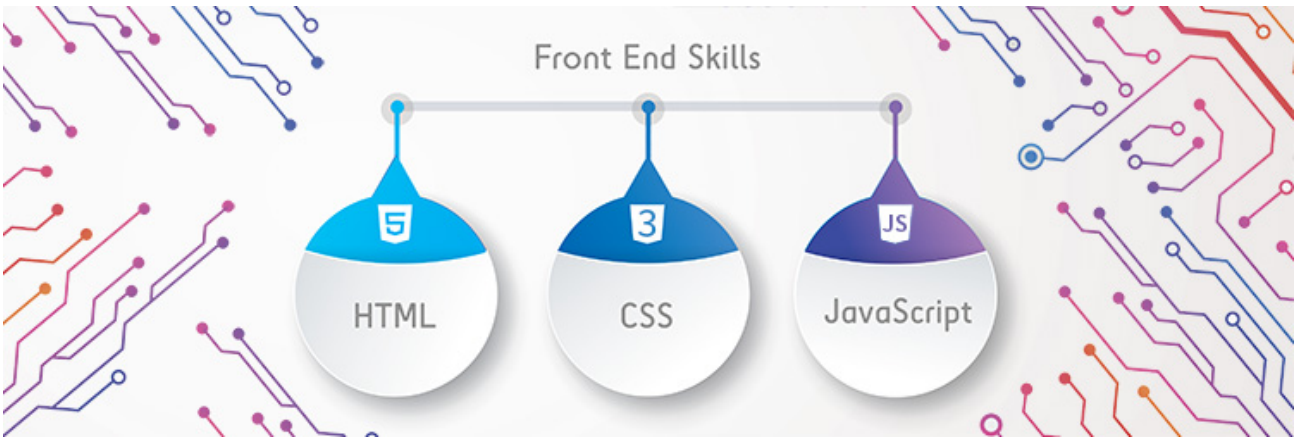
- ProGit
- Chrome Dev Tools
- Accessibility Testing Skills
- Microsoft Virtual Machines
- Virtual Box
- ZAProxy



Top Programming Skills for Testers

With literally thousands of programming languages and new technologies being created daily, figuring out what technical skills to learn can be overwhelming for testers. Fortunately, many web and mobile apps tend to work with a similar tech stack. This means that learning some core skills can help testers work in a variety of environments, large and small. Let's talk about the most useful skills for software testers, and where to find information and training.

"Do the thing you think you cannot do." - Eleanor Roosevelt

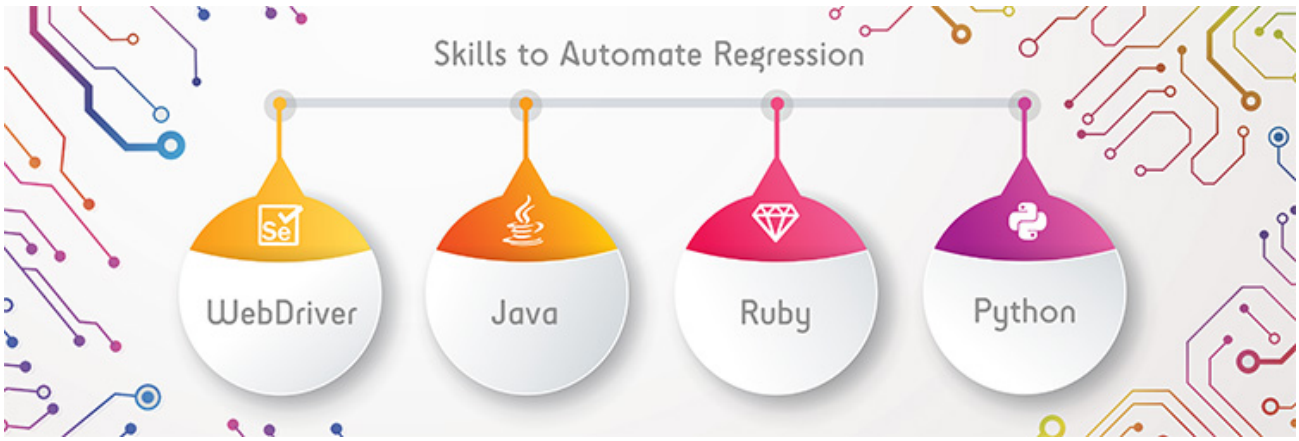


Front End Skills: HTML, CSS, JavaScript

Any tester wishing to know how to write automated UI tests will need to be familiar with HTML/CSS/JavaScript. Understanding these languages, and how they work together, enables a software tester to learn more about how a web application is built. Knowing HTML and CSS helps testers to interpret code from a browser console as they investigate a page. It is a good starting point as HTML is the foundation behind all web pages. It's used to add structure, text, images, and other types of media. CSS is the language used to style HTML content to create visually appealing web pages.

Knowing JavaScript is also helpful because it can be executed from within Selenium scripts, if needed.

Let's say you are a software tester and you are testing a form submission on a webpage. When you submit the form, you see an error. A tester who knows HTML/CSS/JS can open the development tools in their browser, select the console option, and repeat their actions to reproduce the error. In the console, they will be able to see the JavaScript error that is thrown, and then use this information to either further investigate the issue or make a thorough bug report to developers.



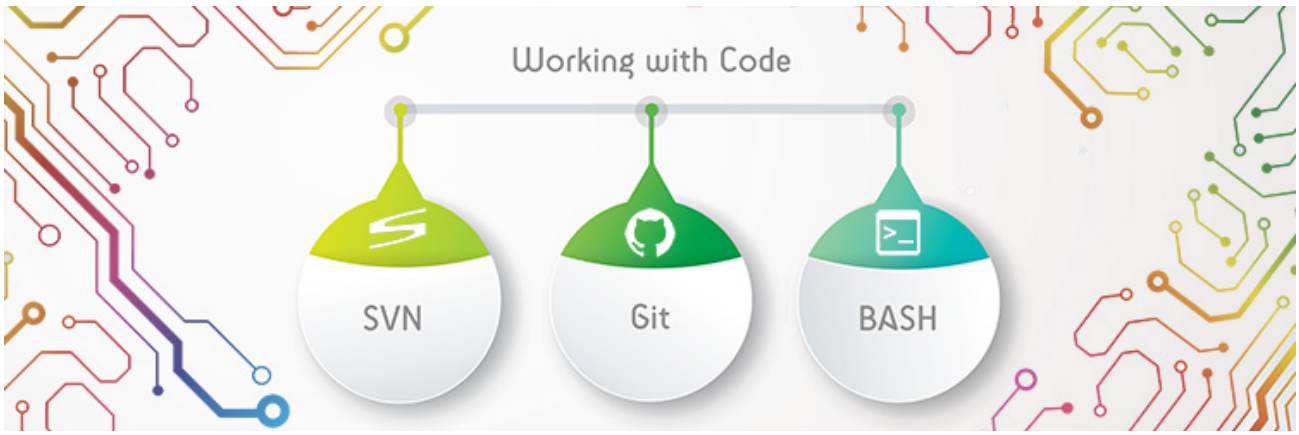
Skills to Automate Regression: WebDriver, Ruby, Java, Python

More often, testers are being asked to create automated UI tests for web applications. While there are many options, Selenium WebDriver (WebDriver) tends to be the most popular API for driving a browser. Ruby, Python, and Java are popular language choices for people wishing to work with Gherkin style syntax and WebDriver. Another reason these languages tend to be popular is that many web apps use them in other capacities, making it easier for the tester to get help from developers in creating and maintaining the test suite.

For example, a tester may be asked to design an end-to-end regression test of a website that includes logging into the site. Using HTML and CSS skills, the tester can open the browser console elements, locate the login box on the page, and inspect the element to find the id of the element. The tester can then use Ruby, Java, or Python to write commands for the WebDriver API. Some sample code in Python might look like this:

```
driver.get("https://www.website.com")
element = driver.find_element_by_id("login-id")
element.send_keys("yourname")
```

The above code goes to the website, locates the textbox that accepts the login, and then enters the login name into the text box. Learning the WebDriver API along with a language like Ruby, Java, or Python are key skills for building automated regression tests.



Working with Code: SVN, Git, and BASH

Sometimes, a tester is lucky enough to work in a large or mature company where they can develop their tests against an environment that just "exists". More typically, however, some environment setup is required. Also, testers who are writing UI automation will need to check in their own code. For maximum work independence and flexibility, testers should know a version control system also. Two popular version control systems are SVN and Git. Both SVN and Git can easily be learned online with free resources.

In addition, testers should get to know the command prompt/command line and some simple Bash Commands to move around both their computer and their code base. Bash Commands can help the tester navigate files easily while writing tests. Additionally, it is generally necessary to use the command prompt to run automated UI tests and investigate the results of failed scenarios or features.

For example, a tester might be starting a job at a company whose code is kept in GitHub. They can do a 'git clone' command to get the code onto the computer. Bash Commands would help the tester move files around, save changes to his test code, and push test changes up to Git for review. Some sample code might look like this:

```
git clone (gets a copy of the repository)
cd (bash command "cd" means "change directory" to your company's repository)
git checkout -b (makes a local copy of the code just for you)
git add . (save your changes)
git push origin/your_shiny_new_branch (sends your branch to the repository)
```

There are, of course, many tech skills beyond this that software testers can learn, and there are no rules about where to start. Different companies can and will employ different technologies. However, these top tech skills can help software testers build a core understanding of web technologies that they can use to branch out into more varied technologies.

Places to Learn Programming Skills Online for Free

The best way to start learning is with a quick web search for a specific programming language. For your convenience, several sites are listed below that were available at the time of writing. The team at TestRail is not responsible for the content or continued availability of third-party sites, and the listing here should not be taken as an endorsement.

HTML and CSS

Learn HTML and CSS – This course takes approximately 10 hours to complete.

JavaScript

Learn JavaScript – This course will teach you the most fundamental concepts in programming JavaScript. It takes approximately 10 hours to complete.

Computer Programming – Learn the basics, starting with Intro to programming. With the Khan Academy you can learn how to use the JavaScript language and the Processing.js library to create fun drawings and animations. There are also courses available that will enable you to combine HTML and JS for interactive webpages.

WebDriver

Online Selenium Tutorial for beginners in Java – Learn Selenium WebDriver automation step by step hands-on practical examples.

Ruby

Learn Ruby – In this course you can gain familiarity in Ruby around basic programming concepts, including variables, loops, control flow, and object-oriented programming. You will also get the opportunity to test your understanding in a final project which you'll build locally. The course takes approximately 9 hours.

Java

Learn Java – In this course you'll learn fundamental programming concepts, including object-oriented programming in Java. You will also get the opportunity to build 7 Java projects, like a basic calculator, to help you practice along the way. This course takes approximately 4 hours to complete.

Python

Learn Python – This course is a great introduction to both fundamental programming concepts and the Python programming language. It takes approximately 13 hours.

Learn Python the Hard Way – This is the 3rd Edition of Learn Python the Hard Way. You can visit the companion site to the book at http://learnpythonthehardway.org/ where you can purchase digital downloads and paper versions of the book. The free HTML version of the book is available at http://learnpythonthehardway.org/book/.

Command Line

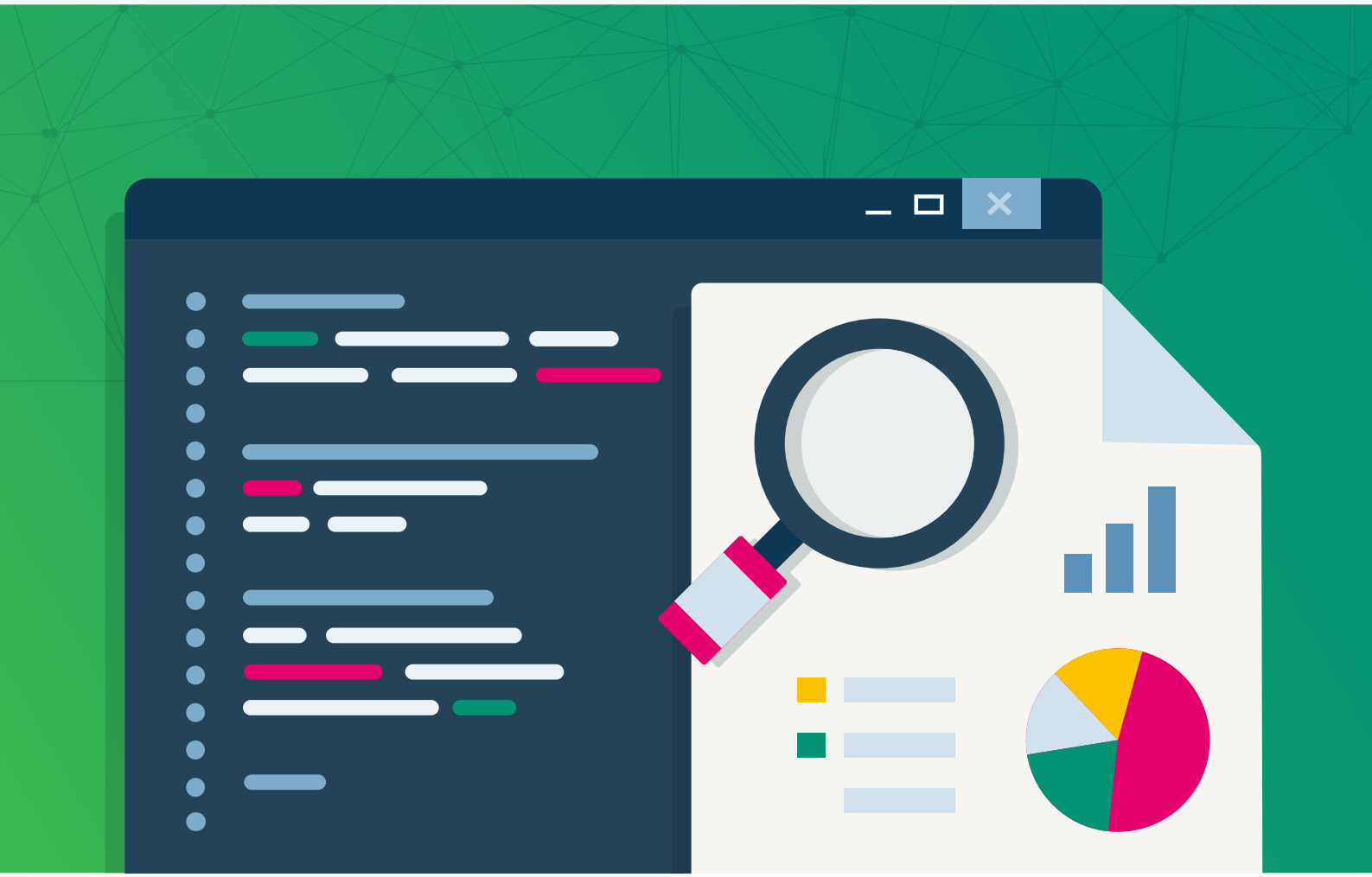
Learn the Command Line – Learning to use the Command Line will help you to discover all that your computer is capable of and accomplish a wider set of tasks more effectively and efficiently. This course takes approximately 3 hours.

SVN

Learn SVN – Apache Subversion which is often abbreviated as SVN, is a software versioning and revision control system distributed under an open source license.

Git

Learn Git – This course teaches you to save and manage different versions of code projects. It takes approximately 2 hours to complete.



Technical Skills for Agile Testers

Top Five Code Metrics to Help You Test Better

Understanding the health of the software's codebase can help you better focus your testing efforts. Static analysis metrics can help testers focus in on areas of the codebase to invest extra attention. Some testers may feel intimidated about code metrics, but understanding how metrics work is vastly different than understanding how to write code that solves hard problems like threading, concurrent data access, scalable web services, etc.

Software metrics are easily used without deep knowledge of how highly technical pieces of software are written. Moreover, metrics are a vital tool in helping educate testers what good software should look like. First off, some clarity around terms as I use them below:

Software metrics are measurements of various aspects of a software codebase. The metrics can be at a high level across the entire codebase or can focus on small blocks of code at the class, method, or even lower level. Such metrics are generally gathered by tools during static or dynamic analysis phases.

Dynamic analysis happens while the system is in operation; static analysis runs without the system being live and focuses on the structure of the code itself, versus the operation of the system. This article focuses on metrics gathered during static analysis.

Static analytics tools examine the code or system and measure specific concerns during the analysis. Over time the industry has figured out general guidelines around various metrics indicating whether or not that particular measurement aligns with generally accepted ranges. Every modern analysis tool will provide measurements and some reference as to standard accepted values for that metric.

(No, there aren't any "best practices" around such metrics, as there's no such thing as a "best practice." Metric results need to be interpreted in the context of the organization's overall practices and maturity.)

There are a great many metrics available for measuring software. This article focuses on the five I've found most useful over a number of years in the industry. They're listed below in reverse order.

5 Comments in Code

What It Is: Overuse of comments in code. Comments in code can be a controversial topic. Old-school development practices encouraged a huge amount of comments, and some in academia mandated a comment for every single statement. Every. Single. Statement. This led to practices such as:

```
int index = 0; //set index to 0
```

which does nothing other than litter code with meaningless information that can be easily missed when making updates to the code, eg:

```
int index = 1; //set index to 0
```

Such disconnects cause huge difficulties when someone goes back to read the code as part of a bug fix or feature enhancement. Is the code right? Was index supposed to be set to a value of "1"? Or was the comment right and index should have been set to zero?

Modern software practices prefer to focus on good naming practices in the code itself. Comments should be reserved for explaining a section with very difficult logic or domain-specific rules. Blocks of confusing, critical behavior should be extracted into well-named methods that make the intent clear. Short, concise, applicable comments should be limited to explaining the *why*, not the how.

For example, here's a completely contrived, non-functional example:

```
public boolean IsShippingTypeValid(Product product,
Order custOrder,
IShippingLookup lookup){
//Call lookup system to check shipping. Reject if
// too heavy or bulky for specific destination.
//See IShippingLookup for more details on rules.
lookup.IsDestinationValidForProduct(custOrder.Destination,
product);
}
```

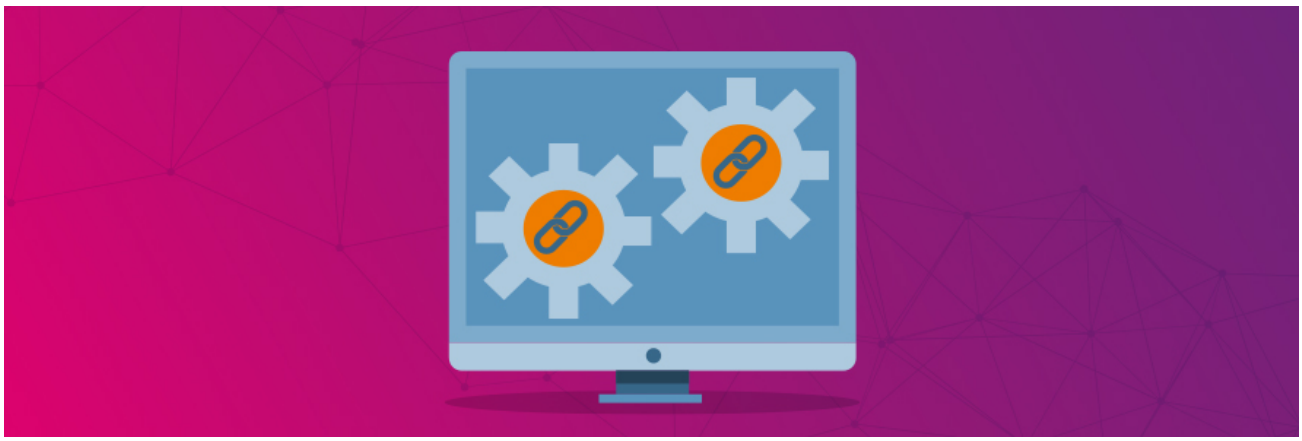
How to Use It: Look at the metric reports for classes and methods with a large number of comments. Open up those classes in your favorite editor and look for disconnects between what the code does and what the comments say. If there's confusion and disconnects then talk with a developer to see if you can find clarity. Look at any automated tests for clarity. Build up and execute test scenarios/exploratory charters to exercise that specific area of functionality.

4 Lines of Code

What it Is: Overly long classes, modules, methods, blocks. Huge classes, blocks, or methods can indicate an area of code that has far too many responsibilities and behavior. Long sections of code are generally very confusing, difficult to test, and very hard to maintain.

How to Use It: Look for long classes and methods. Are those blocks confusing and have mixed responsibilities such as creating data connections while building threads and updating the UI? See what you can discern from the code, and pair up with a developer to discuss risky areas of those large blocks. Again, build up and execute test sessions based on what you discover.

3 Coupling



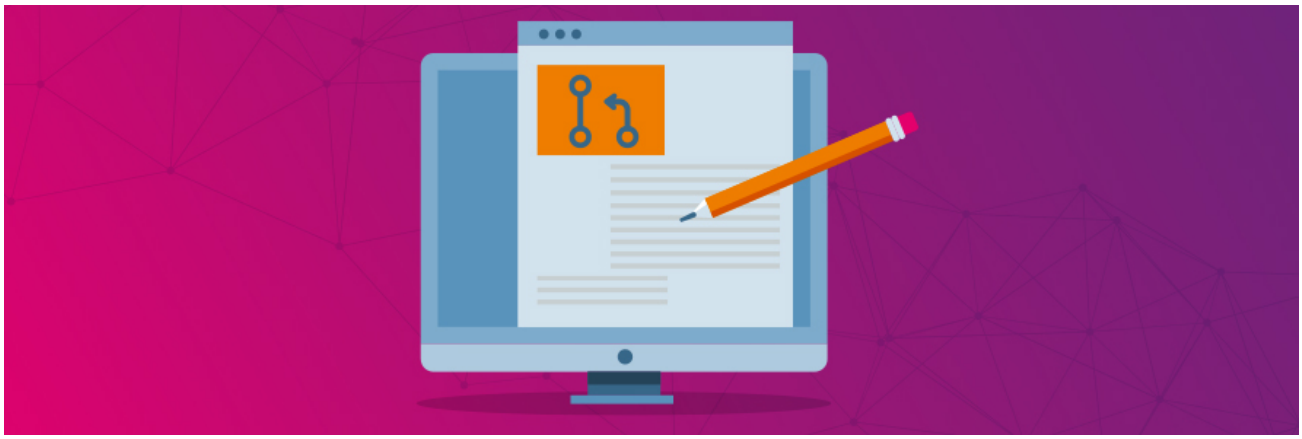
What It Is: Coupling is a two-way measurement. Coupling is how many components use the one you're looking at, and is also how many other components the one you're looking at relies upon. It's a measure of incoming and outgoing dependencies. ("Afferent" is incoming coupling and "Efferent" is outgoing.)

Components with lots of coupling are at risk in two different ways. If lots of other components depend on the one you're examining, then any change to the current component risks breaking those others.

Conversely, if the current component relies on lots of external components, then the risk of the current component breaking explodes since so many other components can impact the current one.

How to Use It: Look for high coupling in either direction. Create breaking changes inside external components and see how the current component reacts. Do the opposite to test outgoing components. Examine how well those areas are protected with good automated unit and integration testing. Shore that up as needed. Explore API testing of the various components to better understand quality issues.

2 Churn



What it Is: Churn is the amount of edits to a source code file. It is usually measured by the number of commits to source control for a specific file.

Files with a high churn indicate specific areas of the system that undergo a lot of updates. While there are exceptions, files frequently updated are normally an indicator that there are quality problems within that file. These quality issues could be straightforward: a difficult, brittle section of the system that requires lots of bugfixes. Quality problems could also be more subtle: a badly designed class or component that is very hard to modify when trying to extend or modify other areas of the system.

How to Use It: Look for areas with high churn. Spelunk into these classes, modules, packages, whatever. As with high coupling, ensure automated regression tests are solid. Build them out on a risk/value approach where needed. Work high churn areas hard via fuzz testing and other similar high-impact approaches.

1 Complexity

What it Is: Cyclomatic complexity is a count of the number of separate paths through a block of code. The more paths, the harder the code is to test and maintain. Nested IF/ELSE statements, multiple SWITCH/CASE blocks, and other similar constructs all quickly drive cyclomatic complexity into dangerous realms.

Cyclomatic complexity is my number one, go-to metric when trying to get a quick feel for the quality and health of a codebase. High, or even outrageous complexity numbers indicate many things about the team's maturity level, design thinking, and plain software engineering skills. None of those indicators are good, by the way.

High complexity crushes a team's ability to clearly express a block's intended behavior, and it makes re-reading that block a mind-numbing, error-prone death march. Moreover, studies have shown a positive correlation between high complexity and high defect rates.

How to Use It: Start top down with the most egregious offenders–blocks with complexity metrics well above normal. Spend some time and overlay churn metrics with the worst complexity sections. See if you can determine bug history (open and closed) for those same blocks. With that data in hand head off and break out testing efforts around the riskiest areas of that metaphorical Venn diagram.

Static Analysis Metrics: Learn Them. Use Them!

Static code analysis metrics can provide testers a wealth of information on the codebase's overall health. The metrics can help inform of solid areas which don't need much attention, and they certainly can help testers determine what areas to spend their valuable time on.

Technical Skills for Agile Testers

7 Pair Programming Tips for Beginners

Erik Dietrich, DaedTech LLC

When you start something new, there's always kind of a make or break period. A good first impression can win a convert for life. Conversely, a bad first impression can win a critic for life. This is true for just about any pursuit. But it's especially true for pair programming.

Why? Well, in the development community, pair programming has plenty of ardent supporters, but also plenty of passionate detractors. If, as a shop, you decide that the pros outweigh the cons, it's unlikely that everyone will share this opinion. Right from the get-go, you'll have ambivalent or reluctant adopters.

So for that reason, here are some pair programming tips specifically for beginners. By following them, you can avoid stumbles out of the gate that might knock you off track permanently.

1 Hygiene Matters a Lot When Pair Programming

Let's get the awkward one out of the way first. Hygiene matters, and that can always make for some wince-inducing conversations. You need to have these, though.

When you start pair programming, you're going to have folks sitting in closer proximity to one another than they're used to. And they'll do it for extended periods of time. So breath mints, showers, not reeking of cigarettes, deodorant — this stuff matters. Lay out some ground rules.

2 Ask Questions Rather Than Leading With Opinions

Many software developers have many opinions. These are often strong opinions. And the holder might often express them bluntly.

In a pair programming context, this can lead to acrimony in a hurry, threatening the enterprise in general. "That's a stupid thing to name a variable" or simply "that's wrong" don't start discussions — they start arguments. You want to avoid this.

One easy way to counteract this impulse is to favor questions instead of opinions. Don't tell someone the variable has a bad name. Instead, ask them why they named the variable what they did. When you ask for explanation first, you start a discussion. Now, that discussion may still lead to disagreements or debate, but it will start off on a constructive foot.

3 Resist the Impulse to Interrupt

The next tip is similar in the sense that it seeks to preempt interpersonal issues. Everyone working in pairs needs to resist the impulse to interrupt the other person, both when they're talking and when they're writing a line of code.

This is a hard one, particularly for people who are enthusiastic and gregarious. Some folks get animated and want to finish their conversation partners' sentences. Or maybe they see where the pair partner is going with a line of thought and start telling them what to name a variable or put a guard clause.

And some people react to this in kind, counter-interrupting and generally enjoying the exchange. Interrupters can be compatible with one another. But they can also completely shut down more reserved pair partners. These folks will go silent, take a backseat and nurse resentment. Establishing "don't interrupt" guidelines help to make sure that everyone is comfortable and heard, and not just the loudest, most excited people.

4 Don't Sprinkle in Solo Work

Now that everyone smells good, and isn't fighting and interrupting one another, we can move on to some logistical concerns. First up among these, don't sprinkle in solo work.

Let me set the scene. A pair of developers is working gamely on some feature when one gets an important personal call that she has to take. She gets up to take the call. While she's gone, her partner shrugs and says, "well, I guess I'll just keep going."

It's a little counterintuitive, but resist the impulse to do this. When you do that, it's jarring to the temporarily detained pair partner, for sure. She'll come back and feel less involved in the feature. But, beyond that, it can easily turn into a sort of "race" where you can get your way by outlasting the other person. If you're going to pair, then pair. When someone has to take a break, the other person can work on other things, like email or a different project.

5 Take Breaks

Speaking of taking breaks, take breaks. If you're used to working alone, you might have a tendency to think you're a lot more productive in uninterrupted fashion than you actually are. "Work for 3 hours straight? No problem! I do that all the time."

Except, you probably don't. You probably stop every time you get an email to see who it's from. You probably get up for more cups of coffee than you realize. And, let's be honest. You probably go on Facebook more than you want to admit.

When you're pairing, you don't do those things. So you wind up staring at the prospect of actual hours of non-stop work, which will exhaust you, particularly if you're not used to lots of collaboration. Get up and take a lot of breaks. The Pomodoro technique can fit well here.

6 Don't Let Passivity Slide

Let's say that you're paired up with someone. You're coding along, getting into a state of flow and feeling pretty productive. And then it occurs to you. It's as if you're working alone.

When this happens, don't just shrug and keep going. Stop and engage your pair partner. Ask him to take over at the keyboard and mouse, or, at the very least, ask him for his opinion. Ask him if he understands what you're doing or if there's anything he'd do differently.

Pair programming should be an ongoing conversation, reminiscent of a mashup of coding and code review. It shouldn't be silent and nobody should be passive. If you're doing that, then you've just got one person working and another person spacing out while watching.

7 Learn the Lingo and Techniques

Here is the most important tip: learn the lingo and techniques commonly used by folks that pair a lot. People have given names to popular pairing approaches, such as driver-navigator and ping-pong pairing.

Things get names when they have well-worn paths to value. So by learning what others have done and had luck with, you'll increase your chances for success. You can do a lot worse than modeling your approach after others who have used a technique effectively.

Technical Skills for Agile Testers

Leveraging Code Kata as a Tester

Jim Holmes, Pillar Technology

Modern software testers need to be comfortable with code. The days of testers not being able to open up a code editor and understand software fundamentals are thankfully vanishing. As modern testers, we need to embrace a new challenge.

This doesn't mean every tester needs to be proficient at writing multi-threaded, highly performant concurrent REST web services accessing storage through dependency injection framework managed data access and surfaced through the latest React Node Angular Javascript framework. To the contrary, despite me having made up the gobbledygook of the preceding sentence testers don't need to know the in-depth technical aspects of coding. Instead, we testers need to know enough to better focus our testing and better collaborate with the developers who do write that complex code.

Kata as a Practice: Creating an Instinctive Flow

Getting to that fundamental competency requires two things: some elementary knowledge, and lots of disciplined practice.

Martial artists have long used kata for developing combat techniques. As Wikipedia says, "The goal is to internalize the movements and techniques of a kata so they can be executed and adapted under different circumstances, without thought or hesitation. A novice's actions will look uneven and difficult, while a master's appear simple and smooth."

Repeatedly working through the small, intentional parts of kata lets students and masters develop muscle memory so that the more significant action becomes instinctive.

Early in its infancy, the software craftsmanship movement seized upon the idea of kata as a practice which could help developers in the same way: create an instinctive flow through the mechanical phases of a software problem, leaving students able to dedicate more critical thought to solving the harder domain aspect of the problem.

Early adopters in the movement looked to small, realistic problems that could be used in the same fashion as martial arts kata: repeat the same problem over and over, learning to flow through the exercise smoothly. As such, software kata around string calculators, bowling game scoring, and other similar practices were born.

These problems lend themselves to the same learning concepts as judo or karate kata. For example, internalizing the mechanics of following the arrange, act, assert pattern for automated tests enables developers to focus on the harder parts of the problem at hand. In the same line, careful attention to the Test Driven Development cycle of red, green, refactor builds the discipline to follow the same critical cycle naturally without thought.

Numerous software kata exist. Dave Thomas, the renowned author and software expert, created a blog series that walks through 21 kata. Robert C. "Uncle Bob" Martin, another well-known expert, wrote the Bowling Game kata years ago. Another great source is at katas.softwarecraftsmanship.org, where you can watch videos of developers working through kata.

So what can you as a tester expect to learn from working kata?

As mentioned above, testers need a basic comfort and level of proficiency in writing, reading, and understanding code. Kata practice is a terrific way to achieve this.

Master Your IDE, Basic Language Syntax and Necessary Libraries

First off, kata help you quickly master your Integrated Developer Environment (IDE). Regardless whether it's Visual Studio Code, Atom, Vim, or something else, kata give you a route to quickly memorize critical keyboard shortcuts and expansion templates. You'll learn how your IDE helps you automatically best format your written code to follow a language's idioms. Kata also give you a chance to learn useful window and pane layouts or configurations. Finally, you'll quickly master executing and interpreting automated tests within your IDE.

Secondly, kata will help you quickly master the basic language syntax and necessary libraries of the language and platform you're using. You'll soon grow comfortable with basic constructs of classes, methods, variables, and statements. Constant repetition will help you move past struggles of how to declare arrays of particular types or the initially confusing mechanics of interacting with collections.

Flow control via a language's if/then/else or do/while constructs can be intimidating and time-consuming to gain proficiency around. A good kata problem can help you move quickly through learning effective structuring and flow handling.

Additionally, good kata get you intimately familiar with popular test automation frameworks. You'll quickly master the mechanics of creating test methods, and the arrange/act/assert structure of most tests will become second nature to you. A good kata teaches you many different ways to leverage a test framework's comparison/equivalence capabilities.

As you get better and better at working through various kata, you'll find yourself developing different patterns of logic and thinking. This maturity is perhaps the pinnacle of kata practice, where you're so adept at the necessary forms that your mind is freed up to do what you as a tester need your brain to do: focus on understanding the problem's domains and intricacies, rather than struggling with guessing at statements.

Develop an Understanding of How Good Clean Code Appears

Finally, good kata give you something slightly more intangible, but greatly more important: an instinctual understanding of how well-written code appears. You may not understand the in-depth technical aspects of how a web service authenticates, checks user authorization, pulls data from an object relational mapper and presents it back through a browser via Javascript. You don't need to! What you can do is sit down with a developer and have a discussion about how the code looks complex due to its unclear naming, extremely long methods and classes, and convoluted, nested if/then/else statements.

Dedicate time each day to working through code kata. You'll find you quickly become proficient and comfortable with the code. You'll be a better tester for it, and you'll be a far better team member.

Technical Skills for Agile Testers

How Testers Can Be Truly Agile

Justin Rohrman, Excelon Development

Every company I have worked with in the past decade has claimed some amount of agility. These companies all practiced Scrum, had sprints, and did retrospectives after each sprint to talk about what was learned and what we could improve upon. All the signals and rituals of agile were firmly in place.

Despite all of that, testers usually spent a significant amount of time waiting around — they usually didn't have any work related to the sprint until the last few days. Development may have been able to respond to change, but testers were stuck in the dark ages.

How can testers be more adaptive to changes in software development?

Move Closer to Development

Handoffs happen when there are strong divisions between roles on a development team. The business analyst talks to the customers, the product owner takes that information and makes requirements, the developer takes those requirements and turns it into software, and then the tester finds where things may have gone wrong.

This is the stuff boring computer science books are made of. Each handoff is a point that adds time to the process — and a point where it can become harder to respond to change.

I've had much better experiences working closely with developers instead.

A couple of weeks ago I was working with a developer to change a key concept in our product. We have files that get scanned into the product that are categorized with a label at scan time. The list of labels had active and inactive labels. Our task was to

remove the concept of inactive labels. We ended up going down a route that wasn't very good. Midway through the change, we talked with another developer and decided to do something completely different.

I was able to change my testing approach in real time with the developer because I was pairing with him on the change and was part of the conversation when we decided to take another design strategy in the code. This new testing strategy included different unit tests, browser automation and some subtly different exploration.

In other projects, I found out about large changes in direction days later and had to amend my approach based on secondhand information.

Being there when the change happened made me far more adaptable to the change, which is the heart of agile.

Get Over a Fear of Technical Problems

Normally when I read about testers having a strong presence in an agile process, it's about their testing ideas. Testers attend design sessions and help to surface important ways the customer might use the product, they go to code reviews and ask about code coverage or what happens when a certain usage scenario occurs, or they generally talk with people and try to find software problems through conversation.

Conversation is powerful, but at some point you actually need to interact with the product. The question for testers working in agile is, what is the earliest point you can test software, not ideas and conversation? I don't think you can get very early in the

process without being comfortable with tooling and being able to string together a few lines of code.

Having some sort of API underneath the user interface is a popular way to develop software. This gives access to data manipulation (CRUD) and also conveniently offers a place to begin serious testing work before a user interface exists, or even to facilitate faster testing of more data permutations when the UI is in place.

One of my last full-time employers was a company that was building a marketing platform. They had been building this platform for about a year and had a single tester on staff until I joined. Their tester was overworked and not able to keep up with the changes that the team of seven developers was putting out each week.

The entire product was built on a REST API platform that wasn't very well tested or explored — scary. But it was also an opportunity. I spent my first few days building a proof-of-concept test and getting that running in continuous integration. After that, we isolated risk in the product and built tests in those areas.

We identified a need on the team, discovered a place where I could be useful as a tester, selected a tool and pivoted after each set of tests based on what was relevant at the time. This is another good example of responding to change and being agile.

Seek Improvement

When I look back at the past 10 years of testing, the majority of the advancements I see come from improvements in development. I regularly see developers doing things that result in fantastic first-time quality: pairing, test-driving, using build deploy pipelines

and containerization. There is also a strong culture of contributing to the craft by creating libraries for public usage and sharing what was learned.

This software is both more testable and harder to test at the same time. The easy-to-find bugs are often designed out of the code through developer testing. Most testers I work with, however, are still focused on the same problems we were working on a decade ago: how we can test effectively in reduced timelines, how we can display the value of our work, how we can build useful automation, how we can effectively report coverage, and why we missed that bug.

A few months ago, the project I am working on was hit with a hard and fast-approaching deadline. We had a large queue of work to complete and not enough staff to get it all done. The normal path for management in these situations is to require staff to work overtime until the deadline hits or the work is complete, whichever comes first.

We had a meeting with the entire development team to figure out how we could become more efficient. We came up with a long list of things, like reducing pairing when it makes sense, having testers float between projects, and having several hour-long blocks where there are no meetings.

Each day during our standup, we would review how our changes were affecting the process, what was working and what needed changing. We also spent some time during our sprint retros talking about how our changes were going and where we could go next. This sort of thing is a daily discussion now. The topic of how we could become more useful is always on the table, and we don't have to wait for a meeting to implement a new change.

Shift Your Culture

Nearly 20 years after the creation of the Agile Manifesto, we still see companies trying to "be agile" or buying into frameworks that allow them to stay as rigid as possible while pretending to respond to change. The successes I have seen are the companies or projects where there is a cultural desire to hunt down specific areas where a team could improve, try something new, make observations about what happened and then try something else. Testers can be involved in the process too, and not just on the receiving end of the development process.

What part of your job can you do better, more efficiently or more effectively? Try something new and see how it works. And keep doing that.

About TestRail

We build popular software testing tools for QA and development teams. Many of the world's best teams and thousands of testers and developers use our products to build rock-solid software every day. We are proud that TestRail – our web-based test management tool – has become one of the leading tools to help software teams improve their testing efforts.

Gurock Software was founded in 2004 and we now have offices in Frankfurt (our HQ), Dublin, Austin & Houston. Our world-wide distributed team focuses on building and supporting powerful tools with beautiful interfaces to help software teams around the world ship reliable software.

Gurock part of the Idera, Inc. family of testing tools, which includes Ranorex, Kiuwan, Travis CI. Idera, Inc. is the parent company of global B2B software productivity brands whose solutions enable technical users to do more with less, faster. Idera, Inc. brands span three divisions – Database Tools, Developer Tools, and Test Management Tools – with products that are evangelized by millions of community members and more than 50,000 customers worldwide, including some of the world's largest healthcare, financial services, retail, and technology companies.