So you want a TDD revolution? Selling your organization on testing…

Audience: Practically anyone who touches a software project—from developers to project managers to quality managers to engineering managers to executives and more. If you're a developer, read the sections for non-developers for perspective on your colleagues' roles. If you're not a developer, read your section and any others for a semi-comprehensive picture of what Test-Driven Development can help you accomplish.

Disclaimer: If you skipped adolescence and young adulthood, we have some bad news for you. It turns out that you can't just force people to do what you want them to do (unless you're comfortable with jail time). There is no surefire strategy to bring the rest of your team or department or company or mad scientist lab to adopt the techniques we are demonstrating in this course. That said, from our experience with this material and helping clients and colleagues put it into practice, we still have for you a variety of resources, recommendations, perspectives, and approaches to consider in promoting these practices in your organization. Adopt and adapt as you see fit.

Do this. Don't do that.

The following are a mix of recommendations for first selling test-driven development to your colleagues and for then implementing test-first practices early on.

Don't:

- Promise test-driven / automated unit testing techniques as a magic elixir able to cure every ailment. It takes time to achieve effectiveness with testing, and testing cannot fix all your quality and process issues (though it can go a long way).
- Introduce test-first / unit testing requirements organization-wide all at once.
- Force test-driven techniques onto all your colleagues without transparency, discussion, experiment, and a game plan.
- Use testing to compensate for poor developers. Testing helps good developers to be better but allows poor developers to write even more bad code.
- Introduce testing for the first time on a project under a tight deadline.
- Be too intimidated. A little testing accomplished while you and your organization are yet green can still deliver good value relatively quickly.
- Separate a project team into those who only write the unit tests and those who only write the code under test. They are largely inseparable. The process of writing tests and writing production code are flip sides of the same developer coin.

- Allow testing to be a scapegoat for a failed or failing project. Project failures invariably involve multiple factors. A technique such as what we're teaching in this course is well suited to preventing released bugs and promoting good design. It's highly unlikely, in and of itself, to be the root cause of a failed/ing project.
- Fall into the trap of attempting 100% code coverage. 100% coverage sings a siren song that can lead to ruin (see our "A Word of Caution" in *Code Assurance: Dynamic Verification, Code Coverage, and Static Analysis, Oh My!* for a full explanation).
- Replace code reviews or other good practices with testing.
- Use test failures to shame other developers. Test failures found now are bugs that don't get released later. When tests fail, they did their job of helping you and your colleagues do yours.

Do:

- Present evidence to support your claims of the benefits of test-first techniques (see Resources section).
- Work with test-first and automated testing before talking to your organization. Your own firsthand experience and initiative will be compelling.
- Appeal to your colleagues' recognition of their capacity to introduce bugs into their code to make your case for automated unit testing.
- Present benefits of test-driven techniques in terms of difficult-to-quantify but still positive experiences:
    - Satisfaction in better design.
    - Delight in unforeseen benefits testing often delivers down the road.[‡]
    - Far greater productivity when maintaining unfamiliar code that has an accompanying test suite.
    - Confidence in code releases (sleeping well at night before big releases).
    - Reducing the burdens of documentation (or lack thereof) through an executable form of documentation.
    - More reliable estimates. Estimation techniques are an advanced topic, but in short, measuring progress in terms of tested code provides a stable benchmark for completion and can then act as a feedback signal into the squishy process of forming estimates.
- Remember that like all aspects of programming, proficiency in testing will increase with practice, experience, and finding solutions to challenges as they arise. Learning to write quality code test-first is not unlike learning to program in a new language. It's about that much of a mental shift. That said, the value in testing begins almost immediately even if you are green and feel that you're fumbling about.
- Treat testing as one (important) practice among many necessary for a high functioning software development organization. Testing is a big leap forward and can be a foundation to build on with other practices, but it is not everything you need to save a sinking ship.

- Look for new opportunities to apply a test-first mindset to your work. Allow a testing mindset to seep into nearly all aspects of your development. Why not test-first hardware design? These are challenging to do test-first, but you're smart. You can find a process whose benefits outweigh the costs.
- Maintain an appropriate paranoia about defects and issues lurking in your code despite testing. Even an expertly crafted and extensive test suite does not guarantee a bug-free release.
- Adopt a shared ownership mindset. All your code is everyone's responsibility regardless of who wrote it (this *can* be practiced apart from establishing an individual developer's proficiency and contribution to the organization). If tests break, fix the code and/or tests—regardless of who is responsible.

[‡] *Real examples of the unforeseen benefits of test-first design with which the authors have firsthand knowledge:*

1. *Development with a test-first approach of a color measurement instrument that included a custom on-device GUI created as a byproduct a channel through which screen grabs were easily generated and transferred off the device. The interfaces and design driven by testing allowed the facilities used for testing screen buffers, etc. to be readily combined into a simple screen grab tool. The ability to access screen grabs was essential to writing operator manuals, performing quality checks, and generating marketing materials but was never expressly specified in product planning.*
2. *Test-first development of a network-operated user interface for a massive piece of production line equipment led to two unforeseen benefits and highly valuable but unspecified features:*
    a. *Network protocol handling facilities and tests for the UI were repurposed as a verification tool for the production line equipment itself.*
    b. *Test-first development yielded a protocol handler driven as easily by network traffic as by a file-based I/O stream. The sales staff saw an opportunity in this. Ultimately, the UI was bundled up on a laptop and driven by simulation files for sales demos of the production line equipment nearly a year before the equipment itself could be delivered.*

Resources

We have prepared for you an excellent resource so you may talk intelligently and articulately to your colleagues about the benefits and tradeoffs of the style of testing we are demonstrating in this course. Please see our other document: ***Test Driven Development: Is It Worth It?*** In it we link to a variety of articles and studies written by **both** proponents and detractors that paint a picture of automated unit testing from the standpoint of business practice and overall team productivity.

The detailed document mentioned above specifically addresses the practice of Test Driven Development in an automation-supported workflow. However, we have mentioned here and elsewhere that testing can yield more than high quality, reliable code. Since the audience for this present document is bigger than only developers, you may wish to read up on the following as they address successful budgeting, estimation, product release schedules, and client relationship management—all enabled and empowered by good testing practice:

- **Feature Driven Development (FDD)** In brief, FDD arranges development work according to the deliverables most important to your client: features. FDD eschews foundation-first development—where a working product is the last step after building layer upon layer—in favor of delivering end-to-end working software as soon as possible. The incrementally delivered product is feature incomplete but the most important features are completed first. Work continues in adding features with successive iterations. In some cases, you can begin generating revenue before a product is feature complete. In fact, you and your client may discover that less important features can be cancelled thus saving further time and money. Because of the realities of hardware development and delivery, FDD will fit your particular embedded software project along a spectrum from impossible to practical. Use your noggin to evaluate for and adapt to your circumstances. You may achieve real gains with FDD even if for only a portion of a project.
    - Delivering Real Business Value using FDD
    - Feature-Driven Development on Wikipedia (See References & External links)

- **Optional Scope Contracts** Optional Scope Contracts [pdf] are an alternative way to arranging the business of building a software-based product for a client. It is complementary to FDD and aligns a development team's resources (progress) with the client's resources (money). The traditional contract for software-based products tends to pit developers and a client as adversaries. In the traditional fixed-bid approach the development team pads estimates and is motivated to cut corners to maintain margins. Conversely, the client is motivated to over-estimate and over-specify their product to compensate for legitimate ignorance (it's oftentimes incredibly challenging to know just what you need before having built anything—especially for complex needs and in complicated workflows). Again, this idea may not be ideally suited for your particular project and client circumstances, but perhaps it will give you new ideas or better perspective on how to manage scope and work with your clients.

- **Estimation & Tracking** Estimating a software product is difficult. We humans tend to be quite poor at estimating how much time a given task will require. This inability is only made worse by complicated, ambiguous projects. Budgeting a project is intimately tied to how you estimate and track it. Agile estimation and tracking techniques embrace the unknown to manage it rather than attempt to specify it away. Good estimates and

reliable tracking require a clear definition of “done.” “Done-ness” can be established by arranging a project around features declared to be done when tests pass.

- Planning and Estimating work for an Agile team
- What is the best way to measure progress on an Agile project?
- Burndown Charts
- How Do You Estimate On An Agile Project? [e-Book]

How to think about adopting testing if you are…

The following subsections present a variety of perspectives and adoption recommendations available to you loosely aligned with job titles and levels of responsibility. Some of these recommendations will naturally overlap as you may wear multiple hats within your software development organization. It’s also the case that you will likely need the buy-in of one or more of the roles listed below that you do not personally fill. Each section that follows is written directly addressing the individual in the named position. Feel free to allow your Project Manager, Product Manager, Quality Manager, Business Owner, etc. to read their corresponding section below (or any part of this document for that matter).

No matter what your role or viewpoint within your organization, keep this in mind (repeating one of the *Do* points above): Learning to write quality code test-first is something akin to learning to program in a new language and shifting your organization to the tools and environment of that language. It’s about that much of a mental shift and that much effort. This is a process and not just a switch to be flipped. That said, the value in testing begins almost immediately. So do not be too intimidated by the preceding statements. A little testing accomplished while you and your organization are yet green can still deliver good value relatively quickly.

A Lone Developer

If you’re a solo developer, there’s few roadblocks preventing you from incorporating test-first / automated unit testing into your personal workflow. Since properly unit tested code is parallel to and separate from production code, you can keep to yourself your unit tests managing them independently from the production code you produce for your organization. That is, even if unit testing is not an official part of your organization’s process, there’s little preventing you from maintaining your own test suite for production code you produce. How you organize this and deal with your specific code repositories is outside the scope of this discussion. But you’re a smart cookie so you’ll figure it out.

Consider suggesting unit testing proficiency as a personal skills development goal to your superiors. If your organization officially supports continuing education and skills development, the techniques we present here can easily fall within such a framework.

Having direct experience with the testing techniques we show you in this course along with actual tests of your organization's production code and personal tales of its effectiveness will go a long ways toward influencing your colleagues and decision makers on officially adopting testing. If at all possible, get yourself some direct experience with these techniques and tools before attempting to influence the rest of your organization.

A Team Lead

Becoming proficient in the sort of testing techniques taught in this course is no small feat. Even having completed this course will not yield proficiency—only experience will bring that. Introducing these techniques to your team all at once could produce shock and lead to failure. Only try to do so all at once if you have the luxury of a clean cutover with strong organizational support, buy-in from your team, and a generous timeline. As such circumstances are probably unicorn-like at best, you will be better served in treating adoption of these techniques as a series of expanding experiments.

Our best advice is to give a small number of developers on your team the task of learning testing techniques **along with you** and implementing these practices in a single all-new area of a project. Ideally, you will want to choose your best and/or most influential developers (really good developers tend to naturally have an affinity for some form of test-driven development). Developers new to test-first techniques have the easiest time in all-new development. Give your developers time to acclimate with multiple exposures. Only once you and your co-conspirators feel confident in the workflow and *desire* to do it for all development should you then task them with mentoring others and implementing all aspects of a project with these techniques. Regularly conduct reviews on progress, helping to solve each other's problems and adapting the techniques to your organization's processes and tools. Encourage experiments in toolsmithing and provide articles, blog posts, and book excerpts on relevant topics to spur discussion (see our various resources included with this course).

A Project Manager

You are not a developer, but you are integral to software development. You have a key role in influencing your development team. Let's discuss why you want your developers to use test-first techniques and why your developers want you to want them to use these techniques.

"Done" is a surprisingly unspecific and malleable concept that causes much trouble for software development. If you've ever used the phrase "done done" in a project meeting, well, you just made our point for us. Using tests to measure done is a good way to squeeze out ambiguity. Consider a part of your code done only when it is accompanied by an appropriately comprehensive test suite and all those tests pass (including some form of acceptance test). Measuring done this way keeps everybody honest and tends to align your estimates of progress with reality. With such an alignment, you then have the opportunity to manage scope in a timely and fine-grained manner. In fact, your test suite can form a backbone of sorts to address more

of your project than only quality. Testing is a nice complement to Feature-Driven Development and Optional Scope Contracts (see Resources section), two powerful ways to manage scope, cost, and delivery of software.

Software product development too often conforms to the flawed Waterfall method of project planning (either intentionally or unintentionally) where comprehensive testing is held for the end of development. This is bad news. Invariably, this sort of testing will be squeezed leading to insufficient testing and a flawed release (that is expensive to fix or recall). By incorporating testing all along the way, this phenomenon can be nearly eliminated.

Without a suite of automated unit tests, it's so so easy to accumulate technical debt in development—cutting corners, intending to clean things up later, etc. Like financial debts, these decisions accrue compound interest. At some unfortunate and inconvenient time in the future, the bill will come due. Testing of the sort we teach in this course is good practice to prevent technical debt in the first place.

A Project Manager is too often hamstrung in effecting meaningful changes to project scope and expectations when schedule trouble arises. In general, because of a lack of accurate, detailed information on progress, projects are discovered to be over budget or over schedule or both far too late in the process in order to do much about it. Further, developers are often motivated to offer overly optimistic evaluations of progress for a Project Manager's assessments and reports. Consequently, it's in your best interest to advocate and support the testing techniques of this course in your development teams. Properly implemented, you can measure progress reliably, make small adjustments to the project all along the way, and have much greater confidence in the end product. These techniques empower you to do your job well, build your reputation as a reliable and effective Project Manager, and to have a reasonably accurate and actionable picture of project health and deliverable progress well before significant delivery challenges arise.

A Product Manager

Take a look at our perspective for Project Managers. Our thinking on the intersection of your responsibilities with the opportunity for the style of testing we teach in this course is similar. Also take a look at the examples of unforeseen benefits arising from good testing practice listed in a lengthy footnote concluding the *Do This / Don't Do That* section.

Test-driven development of code gives you confidence in the product delivered. And particularly when combined with a Feature-Driven Development approach (see Resources), testing affords you greater ability to perform your job. You can deliver features to customers before the product is complete and incorporate their feedback early on or even begin charging partial fees for a functional but limited product. Organizing software work around features and a dependable definition of their done-ness (via passing tests) allows you to manage your product at a finer

grain, react to feedback and changing market conditions earlier, and experiment with greater confidence than other approaches may allow.

That is to say, like Project Managers, you want your developers to use test-first techniques and your developers want you to want them to use these techniques.

A Quality Manager

Like a Project Manager or Product Manager, since you do not write production code, you will be in one of two positions. You may be using your influence in quality management to advocate that your development colleagues adopt these testing techniques. Alternatively, you may be asked to reconcile yourself to working with developers using these techniques.

Now it may be the case that the idea of developers performing testing gives you the willies. You may fear that their underdeveloped abilities to perform testing will lead to ruin. Or, you may fear for your job if others are executing quality management tasks. Relax! Test-first / automated unit testing approaches implemented by developers are your best friend!

Developer testing as taught in this course (even with full system testing—see document *Test Term Potpourri* for our definition of system testing) accomplishes two basic goals:

1. Eliminates many silly low-level bugs that can be maddening to find at higher levels such as through end product testing.
2. Basic verification of feature presence.

In short, developer driven testing leaves for a human quality tester the most high value quality issues that are simultaneously the most difficult to test under automation: responsiveness, reliability, repeatability, security, resilience, etc. Hunting down an off-by-one error in the bowels of code from way up at a user interface is no fun. Stress testing a system with crazy input combinations and generally out thinking the developers with destructive testing is lots of fun.

Eliminating dumb bugs before they even reach you helps you, the developers, and your users. As such, the techniques in this course have a very good chance of allowing you to perform your job exceptionally well and look like a rock star in the process. Code developed in a test-first fashion can also create opportunities for unplanned facilities that allow you better access to the product to be stress-tested and verified (take a look at the examples of unforeseen benefits arising from good testing practice listed in a lengthy footnote concluding the *Do This / Don't Do That* section.)

An Engineering Manager

First take a look at the recommendations for a Team Lead. Our advice to you here is largely an extension of that thinking.

Start small. Task a willing team with learning the techniques (such as taking our course—hint, hint) and give them one or more components of a project as an experiment thereafter. Work closely with them to establish a meaningful review structure and metrics to build a shared understanding of progress, benefits, and integration into your existing organizational processes. As their confidence and proficiency improves, task them with more significant parts of projects or entire projects. Then strategically redistribute the team as mentors to other teams and grow exposure to the techniques with a timetable and project structure that works for you and your people. At an appropriate tipping point of support and exposure to the practices, transition from test-first development adoption as experiment to officially adopted development process.

To yield success in this transition, you must be prepared to arrange organizational support. This could include negotiating rescoped project deliverables, reorganized team structures, redefined yearly objectives for skills development, etc.

An Executive / Company Owner

Good test practice can form a backbone of sorts to address more of your product development efforts than only quality. Among other practices and ideas, testing is a nice complement to Feature-Driven Development and Optional Scope Contracts (see Resources section), two powerful ways to manage scope, cost, and delivery of software. That is to say, testing proficiency within your software development group can easily lead to a host of other benefits to your business processes and bottom line. A full discussion of those approaches and benefits is beyond the scope of this document (dig into the Resources section), but we mention this to help shape your perspective on the immense value of testing of the sort we demonstrate. Also take a look at the examples of unforeseen benefits arising from good testing practice listed in a lengthy footnote concluding the *Do This / Don't Do That* section.

An executive-led or owner-led transition to a test-first software product development organization builds upon our recommendations to an Engineering Manager (and, in turn, our recommendations to a Team Lead). Our advice to you is simply to take the same approach in your engineering department as we advocate to Engineering Managers with one important difference. Rather than negotiate organizational support bottom-up as an Engineering Manager would do, engage the other departments under your management to create top-down support. That support will require slight shifts in priorities, timetables, and expectations to give organizational headroom sufficient to incrementally experiment with and then adopt these techniques. Read the other role sections above for a sense of what this may entail. Most importantly, talk at length with your development team and surrounding structures to build buy-in and work strategically towards a test-first mindset and processes.

Summary and Parting Thoughts

Implementing and adopting test-first software development techniques can benefit your entire organization. The approaches we advocate are primarily concerned with improving overall software quality leading to lower costs and lower stress. But, secondarily, these same approaches can also empower your organization to better manage estimates, budgets, and customer relationships as well as lead to other unforeseeable benefits.

Adopting these practices is a demanding mental shift. But the bigger challenges are in developing proficiency and gathering the organizational support needed to develop and spread that proficiency. Testing can benefit everyone connected to your project, but they will surely need convincing. Evidence—studies and discussion we've linked to, this document, and your own firsthand experience with these techniques—is your friend in making the case on why and how to introduce this approach into your organization's software product development process.

A key idea of the Agile mindset is to be, well, agile. Nothing we've said here is fixed and absolute. Use your keen intellect and experienced judgment to adapt what we've put forward here to your own circumstances.

Good luck!