Tactics for Implementing Test Automation for Legacy Code

DevOps Enterprise Forum

IT REVOLUTION

Tactics for Implementing Test Automation for Legacy Code

DevOps Enterprise Forum

IT Revolution
Portland, Oregon

Tactics for Implementing Test Automation for Legacy Code

IT Revolution
Portland, Oregon
info@itrevolution.net

For quantity purchases by corporations, associations, and others, please contact the publisher at **orders@itrevolution.net**.

Cover design by Brian David Smith and Mammoth Collective.
Interior layout generated by O'Reilly Atlas.

Table of Contents

Preface

The DevOps Enterprise Forum, facilitated by IT Revolution, brought together 50 technology leaders and thinkers in the DevOps Enterprise community for three days in Portland, Oregon in May 2015, with the goal of organizing in to task teams and creating written guidance on the best-known methods for overcoming the top obstacles in the DevOps Enterprise community.

We tackled the five key areas identified by the community at the 2014 DevOps Enterprise Summit:

- Better strategies and tactics for creating automated tests for legacy applications
- Addressing culture and leadership aspects during transforming
- Top approaches to organizational design, roles and responsibilities
- Information security and compliance practices
- Identifying metrics to best improve performance with DevOps initiatives.

For three days, we broke into groups based on each of the key areas and set to work, choosing teams, sometimes switching between teams, collaborating, sharing, arguing… and writing.

After the Forum concluded, the groups spent the next six months working together to complete and refine the work they started together.

Our goal was to generate content in the form of white papers, articles, or other resources in time to share that guidance with the technology community during the DevOps Enterprise Summit 2015 in October.

We are proud to share the outcomes of the hard work, dedication, and collaboration of this amazing group of people.

—Gene Kim
October 2015

Introduction

How can I overcome the challenges of implementing test automation for legacy code?

Many organizations are adopting DevOps patterns and practices, and are enjoying the benefits that come from that adoption. More speed. Higher quality. Better value. However, those organizations often get stymied when dealing with their large legacy codebases, especially when trying to apply test automation. They struggle with where to start with test automation and how to justify the effort, often feeling overwhelmed by the sheer amount of work involved. Because of these struggles, many give up before even trying—consequently missing out on the many benefits that test automation can provide.

Purpose, Audience, and Structure

This document addresses how to meet and overcome the challenges associated with test automation for legacy code. Below, we look at the type of company that may have a need for test automation, along with the typical organizational structure found there. We walk through an approach for justifying test automation within your organization, providing pillars for that justification, objections that are commonly raised, and tactics for overcoming those objections.

Our intended audience is anyone who wants to apply test automation to their legacy code, but is running into internal roadblocks, such as:

- Management or company buy-in,
- Creating space in the schedule, and
- Budget constraints.

By creating this document, we hope that more people will tackle the task of test automation for their legacy code, will be successful in their efforts to implement or increase the use of test automation, and ultimately enjoy the benefits associated with test automation.

Our goal is to cover the basics you'll need to start the test automation journey for your legacy code, and help you engage those around you. However, this document does not cover every tactic, tool, or technique you will likely need.

Nor does this document prescribe any specific “right way” to approach test automation for legacy code. We intend this document as a starting point, and expect that you will tailor the approach for your particular organization. Furthermore, while this document mentions specific tools and technologies, we’re not endorsing any one over another. There are many good tools and technologies available and the “right one” is the one that’s right for you and your company’s current situation.

We describe a fictitious organization and the environment in which the legacy code exists. Our intent is to create a realistic, relatable context. We outline a way for an individual in such an organization to justify test automation for legacy code, and describe the approach for overcoming common objections and challenges to that justification. Of course, while many organizations are similar, no two are the same. You’ll likely need to adapt the approach to fit your context.

1 Introducing Fictitious Company with a Large Legacy Codebase

There are many organizations that started writing software years ago and continued to enhance and maintain that software over time. If you're reading this, you probably work at such a company now. One consistent aspect of such years-old legacy code is that it usually doesn't have any automated tests associated with it.

In this section, we describe a fictitious organization and the context in which that legacy code exists. While the organization and context may not exactly match yours, hopefully it will feel similar enough such that you are able to relate to the circumstances.

Code and Environment at Fictitious Company

You're on a team at Fictitious Company, responsible for an application with multiple millions of lines of code developed over many years. The application code is in a version control system and you've automated parts of the build process using your continuous integration server. The code has no automated tests of any kind associated with it. Many of the developers who built the early versions of the application are no longer with the organization.

The application is on a six-month release cycle. The feature backlog is prioritized and full for the next two years. It's likely a customer will have to wait a year or more even for a high-priority feature request to make it into a release. In some cases, customers have been told that features they were promised in a release would be delivered in a future release because of schedule slippage.

Each six-month release of the application adds about 100,000 lines of new code to the codebase. The Development team is reluctant to delete any unused code because they're afraid of unintentionally breaking other parts of the application.

The Development team follows an Agile process (Scrum) and works relatively well with the Product Management group (which everyone in IT calls "the business"). The Operations team hasn't yet adopted Agile methods. The Development team often, but not always, hits their sprint goals. They usually feel rushed at the end of the four-month development phase to get all of the planned feature functionality built before going into testing with the Quality Assurance team. Sometimes they finish building the planned feature functionality afterthey've handed off the code to the Quality Assurance team to test.

The Quality Assurance team is responsible for testing the application and performs all testing manually. The testing phase is scheduled for two months at the end of the six-month release cycle. However, the Quality Assurance team frequently complains they don't have enough time to adequately test the application. They cite reasons such as: priority "fast track" changes, upstream development activities that took longer than expected, and Product Management's unwillingness to move release dates promised to customers.

The Quality Assurance team conducts regression testing for every release. As the codebase grows, their regression testing is taking longer and longer. At the same time, more bugs are also making their way into production, often requiring post-release patches to fix them. Because of the size of the codebase and the compressed schedule for post-release patches, the patches often introduce as many bugs as they fix.

In general, everyone across departments and teams agrees that the pace with which they can add new feature functionality is slowing and the quality is decreasing. At this point, the status quo is not sustainable.

Organizational Structure

Your organization is like many: it is split into various functional silos. Your organization's CIO has established and documented a software development process which defines all of the responsibilities, artifacts, interactions, reviews, and handoffs for the various groups. The groups usually make decisions based on their own perspective and goals rather than looking across the entire value stream. The CIO does require some basic reporting for IT, including cost and schedule performance for projects, performance against operational service level agreements, and some high-level quality metrics for reported bugs and trouble tickets.

The Product Management group is responsible for identifying and prioritizing the requirements for the organization's products. The Product Owners are in this group, but work with a shared pool of business analysts in another part of the Product Management group to develop detailed requirements and user stories for the individual product backlogs. Everyone in the organization refers to Product Management as "the business" because they define the product roadmaps, control the lion’s share of the organization’s budget, and have primary accountability for the success of the products. The Sales and Marketing teams generally take their direction from the Product Management group. Sales is responsible for managing individual customer relationships and passing along feature requests from customers to Product Management.

The IT organization has three main groups: Development, Quality Assurance, and Operations. Development has all of the developers and architects; that is, everybody who designs and writes code. Quality Assurance has the testers and testing analysts. The testers perform the testing. The testing analysts determine what needs to be tested and how it should be tested, and also write the test scripts for the testers to follow. Operations is responsible for managing all computing infrastructure and networking, as well as “keeping the lights on” for all the customer-facing and internal applications running in production. Operations is also responsible for deploying new application releases into production, with Development taking a supporting role in the deployments when needed. IT also has a small Security group that works with Development and Operations to secure the software and infrastructure.

Performance and Results of the Current Application

The current version of Fictitious Company’s application is certainly not perfect, but it’s not awful either. Most customers renew their contracts, although anecdotal evidence suggests it’s not because they love the application; it’s more out of necessity and avoidance of “switching costs.”

Recent releases have been particularly problematic. One release caused a significant outage, triggering a massive spike in customer calls to the help desk. Customers impacted by the outage received credits to their account in an attempt to improve their satisfaction and retain them as customers. Another release had several critical bugs in it, requiring a quick post-release patch that fixed some of the problems, but also introduced new ones. It took multiple patches to finally produce a stable application. Customer satisfaction metrics for these two releases were noticeably lower than prior releases.

In general, customer satisfaction metrics are definitely trending down. However, the organization is still hitting all its business goals and all the members of the organization’s senior leadership team are still getting their bonuses, so there’s no sense of urgency to make any changes to how the organization works.

Making Changes: Introducing Test Automation?

The decreasing quality of the application is creating a lot of unplanned work and schedule pressure, resulting in stress, longer days, and decreasing employee morale. You want to improve the quality of the application, thereby increasing customer satisfaction.

Based on your understanding of continuous delivery and DevOps, you firmly believe that your organization could benefit from adopting some of those practices, especially test automation. However, other people in the organization—including your management and many of your peers—aren't as convinced.

You feel your organization needs to make some changes to retain customers, retain employees, and meet business goals. You're certain that test automation is one of those needed changes. But how can you persuade the decision makers that it will pay off? And what approach should the company take to implement test automation for such a large legacy codebase?

You realize that it's not realistic to spend the time and money automating all the needed unit tests for all of this legacy code. You believe the answer is to find the right balance of automating tests while still adding the new feature functionality that Product Management wants.

Building a Case: Justification 2

You need a solid approach for justifying test automation to others within your organization. In this section, we identify some qualitative "pillars" as key concepts in your justification. We also give you a framework for evaluating the expected value of a test automation effort, along with a case study of how that framework applied in real life.

Qualitative Pillars

The qualitative pillars are important conceptual points in your justification—logical arguments to support why test automation is a better approach than your company's status quo. We describe a number of these qualitative pillars that you can use in combination with quantitative data to demonstrate test automation's value.

Pillar #1: You need stable and repeatable build and integration.

Repeatable builds are the foundation for automated testing. Although it is possible to invest in automated testing without this pillar, we don't recommend scaling an automated test investment without having repeatable build and integration processes in place. Without these practices, the benefits of fast feedback from automated testing are greatly diminished. If you cannot assemble the code and assets to run tests, what good is the test investment?

Additionally, we have found that lack of repeatable builds results in long feedback loops and can exacerbate the creation of brittle automated tests. We recommend investing in build practices so that you can at least produce a reliable build on demand, or with daily frequency, to validate minimum build quality. This should be a first step in any serious automated test investment.

Pillar #2: Manual testing doesn't scale.

As the scope of the application grows over time, the regression testing need gets larger and larger. Each time you add to the code base, you have to execute all the regression testing again and again. That means that you have to either sacrifice time (e.g., tests take longer), cost (by employing more testers), or quality (by skipping some tests) as the application grows in size.

Pillar #3: Delay and rework robs valuable feature development time.
As rework increases, the amount of time left to build features diminishes. Every rework cycle diverts the team from creating new value; instead, they are busy fighting quality issues. Complexity growth in the software further exacerbates this as it continues to take longer to find and fix any quality issues.

Pillar #4: The primary goal with test automation is to drive down cycle time to understand quality and working software.
Test automation unlocks the ability to greatly reduce cycle time and make quality visible and tangible. With an automated test suite, stakeholders no longer have to wait for elongated regression cycles to understand the quality in the system. The test automation becomes the rapid and visible indicator of this quality. Successful test automation also depends on: Continuous Integration and Deployment, Environment Setup & Congruency, Test Data Setup/Management and System Coupling. These areas will require dedicated focus if you want test automation to provide large scale benefit.

Pillar #5: Optimizing team output across both Dev and QA with high quality is a key goal.
Simply optimizing development time at the cost of quality and rework is not optimal. In environments with separate QA groups, it is vital to bring developers and QA together early and often to integrate test automation into the development process. Developers and testers need to collaborate continuously to embed testing design and implementation as early as possible, rather than just "waterfalling" test automation after development. Otherwise, you will find you have brittle and unsustainable test processes.

Pillar #6: Making the cost of quality visible to gain alignment on the value of test automation.
Low quality costs the organization in time and resources. In our example below, we show how you can leverage simple activity-based accounting to illustrate the inefficiencies brought on by lack of quality.

Metrics

Your management will want to see a return on investment (ROI) for the resources they are being asked to put towards test automation. The table below identifies the important metrics in the justification process, along with the importance of each metric, how automated testing improves the metrics, and an approach for measuring it.

Metric	Why is this metric important?	How does automated testing improve this metric?	How should I measure this metric?
Time to produce a viable build or Minimum Viable Build(MVB).	A viable build is the starting point for any automated test effort to provide value. If you can't build, you can't test.	Automation via continuous integration (CI) practices makes producing a build repeatable and something that can occur automatically or on demand.	Tracking the amount of time it takes to create a build, deploy and test minimum viability should be measured.
Defects exposed to production	Defects exposed to production cause customer impact as well as unplanned work for development and operations to support.	Automated testing increases coverage as well as drives down cycle time to understand quality.	Track tickets by Root Cause Product or Component to create a baseline that can be improved against.
Time spent on regression	Dedicated test regression time is an artifact of "waterfalling" system testing or component testing to the end of the development process. Excess time spent on regression indicates a lack of automated testing.	Good automated test coverage can make regression a non-event and something that can occur every day or continuously. Data and environment setup may also need to be addressed in order to make regression efficient.	Activity accounting can help you measure time spent on regression. Also look for large hardening windows or schedule buffers added to the end of development cycles.
System Test(API) Coverage %	System coverage via APIs tracks two things well: 1) Good architectural design to expose functionality to downstream systems, and 2) coverage of functionality in the system	Automated testing will allow you to rapidly cover more and more functional breadth of the system to reach higher levels of coverage.	The best way to measure API level System Test Coverage is via code coverage. Even if the systems starts at very low levels (5%), improving to just 50% represents a 10x improvement and can make dramatic improvements in quality.
Cycle time to run full test suite	The amount of time required to run a full test suite is important because it represents the time required to understand system quality. Shorter cycle times here also indicate greater system agility.	Good tests and automated test infrastructure help provide faster cycle times to understand quality.	Measure the time required to run the test suite. You should also measure the frequency with which you can run the suite. Ideally, a full suite can run in a few hours every day.
Feature time %	The amount of time in a value stream or team	Good automated testing decreases feedback	Measure the activity of the team via activity ac-

Metric	Why is this metric important?	How does automated testing improve this metric?	How should I measure this metric?
	spends on innovating and creating new features.	loops and reduces waste in downstream activities and rework. This increases the time that the team can continue innovating.	counting to determine how much time is spent creating new features.

Model

There have been several models developed that try to measure the cost of low quality in software. Many apply cost measurements to defects and the expense related to fix them. For our justification here, we focus on a simple Activity Accounting Model derived from Gary Gruver and Jez Humble's work at HP (see Jez Humble's and Gary Gruver's books in "Resources"). We mainly focus on maximizing feature development time by leveraging automation to improve quality and minimize unplanned re-work and decrease cycle time.

Feature development time can be thought of as the time spent creating new value. All other time — essentially waste — can be reduced by testing automation. Note that automating testing alone is often not enough; rather, continuous integration practices and environment automation are often required to get maximum value. Automated tests won't help if you have to spend days assembling environments manually.

To begin building an activity accounting model, first define the key activities that a team or set of teams carries out to build and prepare software for a release. This data can be obtained from timesheet data or by interviewing the team and tracking activities for several sprints or a release window. Note that for justification purposes, it's not necessary to enumerate every activity, or detail your time precisely to the hour. We find it best to break it out as a percentage of total activity as shown in the table of "Before Activity" (or, current state).

Before Activity	Before Time
Agile planning	5.00%
environment setup	10.00%
code builds	10.00%
production/downstream support	25.00%
manual testing (current features)	15.00%

Before Activity	Before Time
test regression (all features)	20.00%
feature development	15.00%
Total	100.00%
Feature Time	15.00%
Non-Feature Time	85.00%

Once you have current activity time gathered, you need to determine how to increase feature development time. In our example table, only 15% of the team's time is spent building new features. The effort expended to automate tests can be justified if you are able to increase that feature percentage, and decrease that non-feature percentage. This can be accomplished by removing other non-value added activities.

In this example, we will look to decrease production support and time spent running regression. The time saved in these activities can be re-directed towards feature development. Note that although this model is directionally sound, there are inevitably limits to getting perfect returns. For instance, reducing testing time may require new skills on the team to implement automation. New skills take time to develop, so achieving perfect ROI is unlikely.

We will carry this model into the case study. In the case study, we look to define target areas to decrease cycle time and improve the amount of time spent building features.

Case Study: CSG International's Test Automation Effort

CSG International's test automation effort on their Service Layer Platform (SLP) is a real-life example of justifying and applying automated tests to legacy systems. This effort required an investment in CI as a basis and then a significant investment in test automation using acceptance test-driven development (ATDD) practices and tooling.

CSG International: Services Layer Platform (SLP)

CSG International provides outsourced customer care and billing services for major cable customers in the United States. The CSG hosting environment is complex, with over 20 application stacks supported by over 40 teams.

Before Description

The CSG Service Layer Platform is a critical service infrastructure that exposes many upstream systems for direct consumption by clients. There are more than 1,200 individual integrations, supporting over a billion transactions per month for fifty million subscribers for key cable and satellite customers in the United States.

The SLP system suffered quality issues in platform code as well as defects leaked from upstream systems to client consumers. At one point in 2011, the teams were fielding almost one major incident per week. Not only was the client impact huge, but this was a large distraction for the team. Since the SLP layer supported customer engagement systems, it was critical to quickly triage and fix these issues. The fixes pulled vital resources (developers, testers and architects/leads) off of feature work to fix production issues. Many of the resolutions were lengthy and required pulling in additional teams to fix or understand desired behavior. Fixes to these incidents were risky and required re-running limited coverage across areas believed to be impacted by the fix.

CSG conducted testing on this platform using traditional test case management practices with pseudo manual efforts. The teams had simple test injection tools that would send sample XML messages into the platform, but this was hardly modern automated testing. Total test coverage was hard to determine because code coverage was not leveraged.

Actions Taken and Results

To fix these issues required investment in several areas beyond simply adding automated tests. The team took the following actions over a three-year period to improve quality on the platform:

1. Investment in stable build and CI.

Prior to taking action, builds were packaged in a haphazard manner and assembled via ad hoc scripts and emailing artifacts between development and operations teams. So, one of the first investment areas was to apply CI practices to the SLP. CSG had seen great success using CI for new platforms but had yet to apply these practices to established or legacy systems. Adding CI to the SLP drastically reduced cycle time and increased certainty that a build was viable.

2. Infrastructure simplification using the Strangler Pattern.

In addition to the poor testing infrastructure, CSG used a complex and arcane middleware to build the nearly 300 transactions exposed to clients. The complexity of the code, combined with lack of tests, prevented them from changing the system in a rapid and low-risk way. Given this, the teams decided to apply the Strangler Pattern to greatly simplify the operating environment and the application code. But how could the team proceed with this path when there was scarce documentation and low test coverage on the current API?

3. Automated Test via ATDD and the Strangler Pattern.

Teams at CSG had recently been trained in ATDD practices and were leveraging these techniques with great success on new efforts to produce code that was automated with acceptance testing as part of the development lifecycle. After seeing these results, the SLP team felt that ATDD practices would be an excellent approach to tease unknown requirements from legacy code to facilitate infrastructure changes in a low risk way. After training and several prototypes the SLP team proceeded on a large scale test automation effort by applying ATDD techniques to implement API test coverage across the legacy system for one area at a time. Once coverage had reached a satisfactory level, the transaction could be ported with near zero risk. The success here demonstrated that leveraging ATDD and modern techniques on legacy code can yield significant value in quality and also enable other concerns like infrastructure upgrades. This automated test approach made strangling off the old infrastructure possible.

Summary of Improvements at CSG

The following table highlights valuable metrics that can be used to measure quality in the system. These metrics cover both Continuous Delivery/Continuous Integration aspects such as "Time to produce a viable build," and traditional quality metrics like "Defects exposed to production" and "System Test Coverage."

Metric	**Before**	**After**	**Notes**
Time to produce a viable build or Minimum Viable Build(MVB).	48 hours	<30 minutes	CSG teams had deep experience in CI pipelines and were able to apply this expertise to the SLP effort. A full CI investment is not required to produce a MVB.
Defects exposed to production	49/year	2/year	The prior defect rates were caused by a variety of problems: brittle infrastructure, upstream system quality, environment congruency and SLP quality. As the teams drove automated test quality higher and cycle time down these other constraints needed to be corrected. Additionally, SLP quality drove up quality in upstream systems exposed via the service layer. In a perfect world, test coverage in upstream systems would always originate at the source. In our case, a combination of automated testing at the exposure layer(SLP) drove earlier detection of quality in the

Metric	Before	After	Notes
			source systems and drove up quality across many systems.
Time spent on regression	20% of release time(15 days)	5% of release time (4 days)	Regression time batched up large amounts of re-work for the entire team.
System Test(API) Coverage %	15%*	68%	*True code coverage prior to test automation was difficult to determine. This is an estimate.
Cycle time to run full test suite	15-20 days	2.5 hours	In the “before” case, a full test suite was defined as new feature testing plus regression time. In the “after”case, most regression was automated and run nightly. All new features were also added to this automated suite. Any errors during nightly regression were added to current backlog and mostly fixed in the current iteration.
Feature time %	15%	55%	Due to re-work, only a small amount of time on the team was being spent building new features. The rest of the time was spent on support and wasteful activities that could be automated.

CSG International: Service Layer Platform Metrics Before vs. After

There are several other key points to note:

1. Core team capacity did not increase.

During the SLP effort the team size dedicated to developing and supporting SLP did not change. The same size team implemented the automated testing, ported the application and also supported the legacy SLP. This was possible due to the efficiencies and feedback loop reduction yielded from CI and automated testing. The team became more efficient via these techniques and was able to take on more work. As noted in realities below the automation effort did impact both team composition and teams outside of SLP development.

2. Team satisfaction and confidence increased.

Prior to adding CI and automation, team satisfaction was low, as was the confidence level in completing changes. This was because there was little visibility into how changes would affect quality, and the incoming stream of high-stress and impacting incidents in production. After making this automation investment, teams were confident that changes made were low-risk as they would receive daily feedback of regressions to the system.

3. Operations partnership was critical to success.

The SLP development team built an excellent DevOps style relationship with the operations team that supported the platform. As quality increased and cycle time dropped, it became imperative to develop new ways of deploying and operationalizing the SLP infrastructure. Without operations involvement, the effort would not have been successful.

The activity accounting improvements as a result of this effort are documented here:

Before Activity	Before Time	After Activity	After Time
Agile planning	5.00%	Agile planning	5.00%
environment setup	10.00%	environment setup	5.00%
code builds	10.00%	code builds	0.00%
production/downstream support	25.00%	production/downstream support	10.00%
manual testing (current features)	15.00%	automated testing (current features)	20.00%
test regression (all features)	20.00%	test regression (all features)	5.00%
feature development	15.00%	feature development	55.00%
Total	100.00%		100.00%
Feature Time	15.00%		55.00%
Non-Feature Time	85.00%		45.00%
%Improvement			266.67%

Some key takeaways from CSG's turnaround:

1. The SLP improvements resulted from many areas.

Initial drivers for the SLP project were to improve quality and reliability of the platform, which required more than just adding automated tests. The improvement efforts began with CI and also included a re-engineering of brittle and complicated middleware. However, we want to encourage readers that automated testing of legacy applications does pay off and it does not require such an ambitious effort. Start first by implementing basic build repeatability to produce a repeatable MVB and then begin layering in your first automated tests.

2. Automated testing pays off, but requires perseverance.

The SLP team was aware that they needed to invest in CI and automated testing. Due to lack of quality and customer impact, the investment was critical in order for the platform to remain viable as a product. The first steps took a leap of faith to get going. Once the initiative started, the scope grew, and they were required more time than originally estimated, impacting resources across many teams including operations and environment management. It took significant courage and perseverance to keep teams and stakeholders reassured and engaged. But in the end, overall efficiency across all teams was improved.

3. Team skills and resources may require investment.

To move testers from simply designing and executing manual tests to actually automating the tests as part of the development process, the SLP team had to invest in training current employees, as well as hiring new resources that could provide automation expertise.

Objections 3

You are likely encountering objections to pursuing test automation for your legacy code; otherwise you would already have the test automation in place. While objections are expected, they can be overcome with time, work, conversation, and creativity. Below, we identify some of the most common objections to test automation, as well as insights into the source of the objections (i.e., what the person might be thinking or feeling). The good news? We also list multiple tactics to help you address each objection.

We've encountered a variety of objections to test automation. This document specifically addresses four of the most common.

- Objection 1: There's not enough time or money to implement test automation.
- Objection 2: Automated tests are too brittle and expensive to maintain.
- Objection 3: Testers don't have the skills to create automated tests.
- Objection 4: We've tried test automation in the past and it didn't work.

You can use as many tactics as appropriate to address these objections and tweak as necessary to fit your particular circumstances.

Objection 1: We don't have enough time or money to implement test automation.

This objection deals with the perceived negative schedule and budget impacts for the work associated with test automation. The objection is coming from a "zero-sum" mentality. You'll hear something like, "We don't have enough time or money to create automated tests. Our schedule is already more than full and we don't have the budget to bring any new people on."

Objection Source

The person raising the objection might be thinking or feeling some of the following:

- "We're already maxed out on the work we can do for the release. We're 100% utilized and have no slack."
- "We are at risk of schedule slippage. Schedule slips have happened before and we don't want it to happen again."
- "I look at how much (manual) testing we're already doing, and it's a lot of work. You're asking me to automate a lot of work. Which is a lot of work."
- "Automation takes a lot of time and the payoff is too far down the road."
- "I'd have to divert dev time to work on the automated tests. If I divert their attention from features, we won't get all the features done."
- "This is new work. New creates change. Change creates risk and chaos. We don't need any more of that right now."

Response

To overcome the objection of "not enough time or money," you need to accomplish three goals:

1. Create space in the schedule to do the test automation work.
2. Manage risk and shorten the "time to payoff" by limiting the scope of the initial test automation effort.
3. Address the value of test automation in the current context of the project.

Applicable Tactics

You should consider using the following tactics in your approach to addressing the objection.

- Tactic 1: Create a small number of automated **smoke tests** to run for each QA build.
- Tactic 2: Hold a weekend hackathon to create the automated tests.
- Tactic 3: Ask for two days for a small team of developers to create the automated tests.
- Tactic 4: Provide justification for the value of the automated tests.
- Tactic 5: Create **information radiators** and reports to increase visibility and transparency.
- Tactic 11: Integrate the automated tests into the build pipeline.

- Tactic 12: Start with creating integration and component tests.
- Tactic 13: Focus the first automated tests on areas of the codebase that produce the most bugs.
- Tactic 14: Automate existing tests the QA group is already running manually.
- Tactic 16: Create failing unit tests to reproduce bugs.

Objection 2: Automated tests are too brittle and expensive to maintain.

This objection comes from the perception that: 1) automated tests often break as the codebase changes, and 2) automated tests take too much time to maintain relative to the value the automated tests provide. It's true that certain types of automated tests are more brittle than others, particularly those higher up in the **testing pyramid**. Therefore, if the team invests too heavily in the more brittle types of tests, they may spend an excessive amount of time just keeping the tests up-to-date and running. They won't get the increased velocity and quality a more balanced testing approach might bring. You'll hear something like, "Given the changes we're making to the codebase, the automated tests would break too often and we'd spend too much time updating them to get them to pass."

Objection Source

The person raising the objection might be thinking or feeling some of the following:

- "The cost of maintaining the automated tests isn't worth the value we'd get from them."
- "We don't even have enough time to create the automated tests, let alone maintain them on an ongoing basis."
- "We're making lots of changes to the codebase. Those changes will break a lot of the tests, and we'll be further slowed down trying to figure out whether there's an actual bug or the test is just out of date."

Response

To overcome the objection that automated tests are too brittle and expensive to maintain, you need to accomplish three goals:

1. Explain the different types of automated tests, their purpose, and the costs/benefits of each (i.e., **the testing pyramid**)

2. Increase the relative value of the automated tests.
3. Decrease the relative cost of creating and maintaining the tests.

Applicable Tactics

You should consider using the following tactics in your approach to addressing the objection.

- Tactic 1: Create a small number of automated **smoke tests** to run for each QA build.
- Tactic 4: Provide justification for the value of the automated tests.
- Tactic 8: Provide education to the team on how to create and maintain automated tests.
- Tactic 9: Use tools that record manual actions by the tester to generate scripts.
- Tactic 10: Create templates and frameworks to make it easier to create and maintain automated tests.
- Tactic 11: Integrate the automated tests into the build pipeline.
- Tactic 12: Start with creating integration and component tests.
- Tactic 13: Focus the first automated tests on areas of the codebase that produce the most bugs.
- Tactic 14: Automate existing tests the QA group is already running manually.
- Tactic 16: Create failing unit tests to reproduce bugs.

Objection 3: Testers don't have the skills to create automated tests.

This objection comes from the traditional view that a tester's primary function is to execute manual test scripts (e.g., click this, type that) and log the results in a tracking system. A smaller number of test analysts (different from the testers executing the manual test scripts) usually write the manual test scripts for the testers to execute. The traditional view also holds that individuals involved in testing do not have the same level of technical expertise as developers; that is, coding is generally outside their skill set. You'll hear something like, "Automated tests require coding. Our testers don't have that skillset."

Objection Source

The person raising the objection might be thinking or feeling some of the following:

- “Our testers don’t have the coding background or expertise needed to create automated tests.”
- “Testers are testers for a reason. They didn’t want to code or didn’t have the aptitude to code. If they did, they would’ve been developers.”
- “Coding is a complicated task requiring lots of expertise to do it right.”

Response

To overcome the objection that testers don’t have the skills to create automated tests, you need to accomplish three goals:

1. Create an accurate picture of the coding involved in automated tests and the skills required to do it.
2. Get the decision-makers to focus first on “what” needs to be done to create automated tests, rather than “who” will do it.
3. Break down the notion that “developers don’t do testing and testers don’t do coding.”

Applicable Tactics

You should consider using the following tactics in your approach to addressing the objection.

- Tactic 6: Pair a developer and a testing analyst together to create automated tests.
- Tactic 7: Identify those doing testing who also have coding skills or the ability to learn them.
- Tactic 8: Provide education to the team on how to create and maintain automated tests.
- Tactic 9: Use tools that record manual actions by the tester to generate scripts.
- Tactic 10: Create templates and frameworks to make it easier to create and maintain automated tests.
- Tactic 15: Change the culture around testing and builds.

Objection 4: We’ve tried test automation in the past and it didn’t work.

This objection stems from previous efforts to implement or increase test automation that didn’t deliver the expected results of increased velocity and improved quality. Of course, the shortfall in testing automation results your company experienced in the past could be attributed to a variety of reasons.

- The people who created the tests were not the same people responsible for maintaining them over time as the code changed. There was no accountability for keeping the tests valid. For example, the initial creation of automated tests may have fallen to an offshore/outsourced group while new development tasks stayed with the in-house IT shop. No one was responsible for maintaining the tests once they had been created.
- The company culture valued meeting the schedule more than they valued quality. As a result, they may have ignored failing tests in order to hit a milestone date, rather than fixing the root issue that made the test fail in the first place.
- No time was built into the schedule to create and maintain automated tests. The codebase “evolved away” from the tests, resulting in new code without automated tests and old tests that became irrelevant and ignored as code changed.
- The test automation effort didn’t focus on the right kind of automated tests from the outset. For example, the emphasis may have been on creating many low-level unit tests requiring significant effort, or perhaps there may have been too many brittle acceptance tests that needed to be changed frequently.
- No one created a job on the continuous integration server to integrate the automated tests into the pipeline. Nothing and no one ran the automated tests regularly or frequently so they went unused and became inconsistent with the codebase.

Objection Source

The person raising the objection might be thinking or feeling some of the following:

- “We’ve been down this road before without success. Why will this time be any different?”
- “We spent a lot of time and money the last time and got no value for it.”
- “We still have remnants (like old automated test code) from the last test automation effort that aren’t being used. Why would we do more when we’re not even using what we already have?”
- “Even if we start well with the automated tests, we won’t be able to maintain them. We lack the discipline to keep the tests valid.”

Response

To overcome the objection that any new effort to implement test automation won't deliver the expected benefits any more than past efforts did, you need to accomplish three goals:

1. Identify the important differences between the last test automation effort and this one that will improve the chances of success this time.
2. Establish a realistic approach for keeping the automated tests up-to-date as the code changes.
3. Address the accountability for automated test maintenance; that is, who will maintain the tests and why they should care.

Applicable Tactics

You should consider using the following tactics in your approach to addressing the objection.

- Tactic 5: Create **information radiators** and reports to increase visibility and transparency.
- Tactic 11: Integrate the automated tests into the build pipeline.
- Tactic 12: Start with creating integration and component tests.
- Tactic 13: Focus the first automated tests on areas of the codebase that produce the most bugs.
- Tactic 14: Automate existing tests the QA group is already running manually.
- Tactic 15: Change the culture around testing and builds.
- Tactic 16: Create failing unit tests to reproduce bugs.

Tactic Details 4

As you can see, your strategy for implementing test automation and overcoming objections can include a variety of tactics. The tactics that will work best for you depend on your particular circumstances and context. Again, there is no "one size fits all" approach to this kind of change—what works great in one setting may be completely ineffective in another. As the change agent for your organization, you (and other supporters) should assemble these tactics and others into a plan that works for you and your organization.

For reference, the table below shows which tactics you should consider when responding to each objection

The full list of tactics is:

- **Tactic 1:** Create a small number of automated **smoke tests**.
- **Tactic 2:** Hold a weekend hackathon to create the automated tests.
- **Tactic 3:** Ask for two days for a small team of developers to create the automated tests.
- **Tactic 4:** Provide justification for the value of the automated tests.
- **Tactic 5:** Create **information radiators** and reports to increase visibility and transparency.
- **Tactic 6:** Pair a developer and a testing analyst together to create automated tests.
- **Tactic 7:** Identify those doing testing who also have coding skills or the ability to learn them.
- **Tactic 8:** Provide education to the team on how to create and maintain automated tests.
- **Tactic 9:** Use tools that record manual actions by the tester to generate scripts.
- **Tactic 10:** Create templates and frameworks to make it easier to create and maintain automated tests.
- **Tactic 11:** Integrate the automated tests into the build pipeline.

- **Tactic 12:** Start with creating integration and component tests.
- **Tactic 13:** Focus the first automated tests on areas of the codebase that produce the most bugs.
- **Tactic 14:** Automate existing tests the QA group is already running manually.
- **Tactic 15:** Change the culture around testing and builds.
- **Tactic 16:** Create failing unit tests to reproduce bugs.

	Objection 1: "We don't have enough time or money to implement test automation."	**Objection 2: "Automated tests are too brittle and expensive to maintain."**	**Objection 3: "Testers don't have the skills to create automated tests."**	**Objection 4: "We've tried test automation before and it didn't work."**
Tactic 1: Create a small number of automated **smoke tests**.	X	X		
Tactic 2: Hold a weekend hackathon to create the automated tests.	X			
Tactic 3: Ask for two days for a small team of developers to create the automated tests.	X			
Tactic 4: Provide justification for the value of the automated tests.	X	X		
Tactic 5: Create **information radiators** and reports to increase visibility and transparency.	X			X
Tactic 6: Pair a developer and a testing analyst together to create automated tests.			X	
Tactic 7: Identify those doing testing who also have coding skills or the ability to learn them.			X	
Tactic 8: Provide education to the team on how to create and maintain automated tests.		X	X	
Tactic 9: Use tools that record manual actions by the tester to generate scripts.		X	X	

	Objection 1: "We don't have enough time or money to implement test automation."	**Objection 2: "Automated tests are too brittle and expensive to maintain."**	**Objection 3: "Testers don't have the skills to create automated tests."**	**Objection 4: "We've tried test automation before and it didn't work."**
Tactic 10: Create templates and frameworks to make it easier to create and maintain automated tests.		X	X	
Tactic 11: Integrate the automated tests into the build pipeline.	X	X		X
Tactic 12: Start with creating integration and component tests.	X	X		X
Tactic 13: Focus the first automated tests on areas of the codebase that produce the most bugs.	X	X		X
Tactic 14: Automate existing tests the QA group is already running manually.	X	X		X
Tactic 15: Change the culture around testing and builds.			X	X
Tactic 16: Create failing unit tests to reproduce bugs.	X	X		X

Below, we delve further into each tactic that you might employ.

Tactic 1: Create a small number of automated **smoke tests**.

- A "small number" might be 5–10. Keeping the number of tests small limits the amount of work to create the tests, as well as the downstream work to maintain the tests.
- These tests should exercise the basic functionality of the system—not complicated edge cases. This will make the tests less susceptible to change.
- These tests should be simple and easy to maintain.

- You're creating a small number of good, valuable tests—don't try to take on too much, which would overwhelm your developers, your schedule, and your budget.
- Good candidates for these tests would be the starting points for many of the other testing scenarios—like "log in" or "go to the home page."
- These tests should run at a minimum right after doing a deploy to the QA environment and before announcing to the QA group the new build is ready for testing.
    - A better approach would be to run the tests after each code commit or on a schedule through a job on the continuous integration server (see Tactic 11).

Tactic 2: Hold a weekend hackathon to create the automated tests.

- Holding the event on a weekend limits the impact on the "normal" work schedule.
- Make the event fun.
- Provide comp time or another benefit.
- Position the event as (among other things) a learning opportunity for attendees.
- Invite developers and others who might be interested in automated testing (e.g., analysts, testers, ops) to promote inclusivity.
- Use this tactic if your organizational culture is receptive to "giving your own time."
- You should also use the hackathon to create a job on the continuous integration server to run the automated tests (see Tactic 11).
- Ask for some time in the normal schedule following the hackathon to maintain the tests.

Tactic 3: Ask for two days for a small team of developers to create the automated tests.

- You're time-bounding the effort and limiting the number of developers involved to minimize the impact on the organization's schedule.
- The rest of the developers can continue to focus on feature development.
- You should also have the team create a job on the continuous integration server to run the automated tests after each build (see Tactic 11).
- Use this tactic instead of or in addition to the hackathon to create space in the schedule.

Tactic 4: Provide justification for the value of the automated tests.

- The organization is already losing time due to idle time and context switching from unplanned work resulting from broken builds.
- You're also spending significant time doing regression testing late in the lifecycle.
- You can expect to start saving time as soon as the automated tests are in place, so the ROI is immediate.
- Using automated tests will save time in regression testing.
    - Testers will need to do less manual testing for areas covered by the automated tests.
    - Testers will find fewer bugs in the areas covered by the automated tests, which will result in less unplanned work for the team overall (i.e., developers fixing, testers re-testing).
- When using automated smoke tests for QA builds (see Tactic 1):
    - A broken build delivered to QA costs us two days for both Dev and QA to handle break/fix.
- One broken build caught before it goes to QA saves two days against the schedule and helps us avoid "heroic efforts" (e.g., late nights, weekends) to stay on schedule.

Tactic 5: Create **information radiators** and reports to increase visibility and transparency.

- The information radiators and reports will show which tests you have, when they run, and the results of those runs (e.g., pass, fail).
- A report will also show how much of the code is covered by automated tests, which will increase confidence in the quality of that code.
- A secondary benefit of the test automation and reports is they also increase auditability since the results of the test runs are artifacts themselves.
- The team will get timely notifications when a test fails so they can fix the issue quickly. The sooner you know about a problem, the easier it is to fix, the fewer downstream impacts you have (e.g., sending a broken build to QA), and the less unplanned work you will have.
- Using information radiators accessible to the whole team will increase the team's awareness of testing and overall code quality.
- You'll get more visibility into the quality of the code so you'll feel more in control.

- The information radiators and reports could help you justify the test automation effort by measuring both defect density and mean time to resolution (MTTR) for automated versus manually tested code.

Tactic 6: Pair a developer and a testing analyst together to create automated tests.

- Dev and Test employees together have complementary skills enabling them to create and maintain the tests.
- The testing analyst brings the skills of what needs to be tested and how it should be tested.
- The developer brings the skills to write code to automate the tests.
- These two skillsets complement each other.
- Initially pair senior developers with senior testing analysts to improve the quality and value of the automated tests. This will also help you learn more about what works and what doesn't in your environment. You can introduce less senior staff into the process later.

Tactic 7: Identify those doing testing who also have coding skills or the ability to learn them.

- Some of those performing testing do have the skills based on their background, experience, and interests.
- Some would be eager to learn new skills to advance their career.
- Creating automated tests doesn't require the same level of expertise as designing and building business functionality.
- Creating the tests is separate from and requires different skills than creating the test harness. Creating the test harness generally requires more technical expertise than creating automated tests that work within the harness.

Tactic 8: Provide education to the team on how to create and maintain automated tests.

- There are many options for the education: classroom-style training, online training and tutorials, after-hours events, resources lists, “lunch and learn/brown bag” sessions, workshops.
- The education will raise overall awareness of automated testing and signal to the team that automated testing is important since you are allocating time to learn about it.

- Different options could be more effective for some people at learning and more cost effective for the organization.
- Ensure the education is applied on the job as soon after (or even during) the training as possible.

Tactic 9: Use tools that record manual actions by the tester to generate scripts.

- Some tools (e.g., **Selenium**) provide a “record” feature that captures the manual actions of a tester for future automated “playback.”
- These tools provide a starting point for code that can be edited later, rather than having to create code from scratch.
- Using the tool and maintaining the generated code doesn’t require experienced developers, although it does require someone to be comfortable with the tool and code in general.

Tactic 10: Create templates and frameworks to make it easier to create and maintain automated tests.

- Templates and frameworks reduce the amount of effort needed to create automated tests since they provide reusable code.
- Templates and frameworks make the creation of automated tests more accessible to people with less technical experience because they abstract away some of the complexities of creating the tests and integrating them into the test harness.
- Because templates and frameworks make it easier to create and maintain automated tests, you’ll reduce the dependence on key development resources and increase the number of people capable of creating and maintaining automated tests.

Tactic 11: Integrate the automated tests into the build pipeline.

- The integration could be accomplished through a job on the continuous integration server.
- The job should run after each build.
- You should configure the job so that if an automated tests fails, it breaks the build.
- The tests will run more frequently and therefore will be more valuable.
- Running the tests automatically as part of the build pipeline will shorten the feedback loop between code creation and test run, which will make it easier for developers to fix problems and reduce unplanned work.

Tactic 12: Start with creating integration and component tests.

- Automated integration and components tests are less susceptible to change than brittle acceptance/user journey tests, which makes them easier to maintain.
- Automated integration and component tests deliver more value quicker than unit tests. You can test larger parts of the system with relatively fewer tests.
- Automated integration and components tests help stabilize what is often the most fragile parts of the system—component-component interfaces and boundaries with external systems.

Tactic 13: Focus the first automated tests on areas of the codebase that produce the most bugs.

- You can determine what areas of the codebase produce the most bugs by analyzing bug reports and trouble tickets.
- Automated tests used to stabilize fragile parts of the system are more valuable since the tests will catch more problems earlier and reduce more unplanned work.

Tactic 14: Automate existing tests the QA group is already running manually.

- You're automating tests already deemed valuable. (Why else would QA be running them?)
- By aligning the automated tests to valuable work already being done, the test automation isn't an academic exercise or a "crusade."
- After creating the automated tests, you can compare the results from the automated tests to the results from the manual testing. If the manual testing is no longer catching bugs for areas of the codebase covered by the automated tests, you could make the case to eliminate that manual testing, which would save time and money without sacrificing quality.

Tactic 15: Change the culture around testing and builds.

- You're establishing clear accountability with developers for keeping the tests up-to-date and passing.
- The developers will maintain their code and the associated automated tests—not a separate group. This promotes accountability and increases efficiency because the tests and the code are maintained together.
- Include automated tests as part of code reviews.

- Establish cultural team norms around testing and builds.
    - As a team, you're committed to keeping builds "green" (i.e., all automated tests pass).
    - As a team, you're committed to fixing a broken build immediately and fixing a broken build takes priority over other work.
    - As a team, you're committed to not "ignoring" any automated tests (i.e., commenting them out, removing them from the test harness, ignoring a failing test result) to keep the build "green." (Note: One organization went so far as to say if you were found to have commented out code or ignored a test result to get a build to pass, you would be fired.)
- Bringing the pain of broken builds forward will encourage positive changes, such as decreasing build time and fixing non-deterministic (or "flaky") tests.

Tactic 16: Create failing unit tests to reproduce bugs.

- A developer responsible for fixing a bug first writes a failing unit test to reproduce the bug. Then the developer works on the code until the unit test passes.
- A failing unit test promotes more clarity and understanding of a bug between developers and testers.
- This approach reduces the possibility of regression—which saves time and money for both developers and testers.
- Writing one unit test to reproduce a bug is a small effort and won't adversely impact the schedule or budget. In fact, it should help the schedule by reducing rework.
- Unit tests are less brittle than other types of tests (e.g., integration, UI, acceptance).
- You should create a job on the continuous integration server to run unit tests on each code commit. If a unit tests fails, it should break the build.
- Set the expectation with developers that their work is not done until all the tests pass.

The Ask 5

So, you've concluded that adding test automation to your legacy codebase will produce many benefits for your team and your customers. You have business justification. You're also armed with reasonable tactics you can use to overcome any objections to test automation. Now it's time to make "the ask."

"The ask" is when you go to the person (or people) that holds the positional and decision-making authority within your organization to make these changes happen. It could be your manager, the Dev lead, the Product Owner, a project manager—anyone whose buy-in can help set the test automation implementation in motion.

You (the "change agent") should go to the "decision-maker" with your proposal, and ask for the resources you need to pursue test automation. "The ask" could be for time, money, people, flexibility, or freedom. You should keep the following principles and concepts in mind when making "the ask."

1. Focus on the business reasons for doing test automation.

Who will benefit from this effort? How will they benefit? How much will they benefit? Keep your language focused on the business—rather than technical correctness or opinion-based arguments.

2. Start small.

Use an incremental, near-term approach at the outset. Keep "the ask" within your sphere of control or influence as much as possible. The more control or influence you have to make things happen, the easier it will be to make them happen. And keeping the effort bounded in near-term will help with clarity and your ability to deliver value quickly. Once you've made good on your commitments, shown success, and delivered value, you'll have earned the right to ask for more.

3. You’re trying to break the **vicious cycle** that includes manual testing and get into the **virtuous cycle** that includes test automation.

The vicious cycle explains the chain reaction that happens when you have technical debt, in this case: no test automation. The virtuous cycle explains what will happen when you start reducing technical debt by pursuing test automation. These cycles should provide a good conceptual foundation for the discussion.

4. Balance the “gives” and “gets” between the team and the organization.

You’ll likely have to “give” something first (likely your time) if you want to make a change. However, you should attempt to maintain some semblance of balance over time of “gives” and “gets” between the organization and the team. If the team “gives” something (e.g., their personal time), the organization should “give” something, too (e.g., time during the workweek, budget, policy flexibility).

5. Don’t argue if someone raises a concern or objection.

Arguing doesn’t help. Rather, it builds walls and resentment, and makes people less willing to work with you. Instead, use empathy and a spirit of collaboration to address the underlying concerns causing the objection in the first place. You won’t successfully overcome every objection immediately. Be patient and keep the long-term goal in mind. The short-term goal is to create just a little space to do something small now that provides value and will advance you toward the long-term goal.

We hope that this document has provided guidance that helps you find success in implementing test automation for legacy code within your organization. Good luck!This is some text.

Resources A

We've provided some links to resources we've found useful in crafting approaches to test automation.

Books

- ***Working Effectively with Legacy Code*** by Michael Feathers, (Prentice Hall, 2004)
    - **Summary / Introduction**
- ***Experiences of Test Automation: Case Studies of Software Test Automation*** by Dorothy Graham and Mark Fewster, (Addison-Wesley Professional, 2012)
- ***More Agile Testing: Learning Journeys for the Whole Team*** by Janet Gregory and Lisa Crispin, (Addison-Wesley Professional, 2014)
- ***Clean Code: A Handbook of Agile Software Craftsmanship*** by Robert C. Martin, (Prentice Hall, 2008)
- ***Refactoring: Improving the Design of Existing Code*** by Martin Fowler, (Addison-Wesley Professional, 1999)
- ***ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*** by Markus Gartner, (Addison-Wesley Professional, 2012)
- ***Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*** by Jez Humble and David Farley, (Addison-Wesley Professional, 2010)
- ***Lean Enterprise: How High Performance Organizations Innovate at Scale*** by Jez Humble, Joanne Molesky, and Barry O'Reilly, (O'Reilly Media, 2014)
- ***A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*** by Gary Gruver, Mike Young, and Pat Fulgham, (Addison-Wesley Professional, 2012)

Online Resources

- **Test Pyramids** by Martin Fowler
- **Running an internal hackathon** by Randall Degges
- **Strangulation: The Pattern of Choice for Risk Mitigating, ROI-Maximizing Agilists When Rewriting Legacy Systems** by Joshua Gough
- **Can unit testing be successfully added into an existing production project? If so, how and is it worth it?** Stack Overflow

Authors and Contributors B

Authors

- Jeff Gallimore, Partner, Excella Consulting
- Steve Neely, Director of Software Engineering, Rally Software
- Terri Potts, Technical Director, Raytheon IIS Software
- Scott Prugh, Chief Architect, CSG International
- Tim Wilson, Solution Architect, IBM

Other Contributors

- William Hertling, HP
- Anders Wallgren, CTO, Electric Cloud
- Jeremy Van Haren, Director of Software Development, CSG International

Acknowledgments C

IT Revolution wishes to thank the following sponsors for making the DevOps Enterprise Forum possible.

ca technologies DEVOPS ENTERPRISE IBM puppet labs RALLY

We wish to thank all the participants of the 2015 DevOps Enterprise Forum

- Steve Barr, Executive Director, Operations at CSG International
- Ross Clanton, Senior Group Manager, Target
- Jason Cox, Director of Systems Engineering, The Walt Disney Company
- Dominica DeGrandis, Director, Learning & Development, LeanKit
- James DeLuccia, Director and Leader for Certification Services, EY Certify-Point
- Jason DuMars, Senior Director of Technical Operations, SendGrid
- Paul Duvall, Chairman and CTO, Stelligent, Author of Continuous Integration and DevOps in AWS

- Damon Edwards, Managing Partner DTO Solutions, Inc
- Nicole Forsgren, PhD, Director Organizational Performance and Analytics, Chef
- Jeff Gallimore, Partner, Excella Consulting
- Gary Gruver, President, Practical Large Scale Agile LLC
- Sam Guckenheimer, Product Owner, Microsoft
- Mirco Hering, DevOps Lead APAC, Accenture
- Christine Hudson, Solutions and Product Marketing, Rally Software
- Jez Humble, Owner, Jez Humble & Associates LLC
- Mustafa Kapadia, DevOps Service Line Leader, IBM
- Nigel Kersten, CTO, Puppet
- Gene Kim, Author and Researcher
- Courtney Kissler, Vice President of E-Commerce and Store Technologies, Nordstrom
- Dave Mangot, Director of Operations, Librato, Inc.
- Mark Michaelis, Chief Technical Architect, IntelliTect
- Heather Mickman, Senior Group Manager, Target
- Chivas Nambiar, Director DevOps Platform Engineering, Verizon
- Steve Neely, Director of Software Engineering, Rally Software
- Tapabrata "Topo" Pal, Product Manager, CapitalOne
- Eric Passmore, CTO MSN, Microsoft
- Mark Peterson, Sr. Director, Infrastructure Engineering & Operations, Nordstrom
- Scott Prugh, Chief Architect, CSG International
- Terri Potts, Technical Director, Raytheon IIS Software
- Walker Royce, Software Economist
- Jeremy Van Haren, Director of Software Development, CSG International
- Jeff Weber, Managing Director, Protiviti
- James Wickett, Sr. Engineer, Signal Sciences Corp
- John Willis, Director of Ecosystem Development, Docker
- Tim Wilson, Solution Architect, IBM
- Elizabeth Wittig, Field Solutions Engineer, Puppet
- Julie Yoo, Vice President, Information Security Compliance, Live Nation`

And we would also like to acknowledge the organizers, scribes, editors, and designers who lent their support and attention to make the event and these artifacts possible:

Alex Broderick-Forster, Alanna Brown, Robyn Crummer-Olson, William Hertling, Aly Hoffman, Todd Sattersten, and Brian David Smith.