Strategies for adopting Test Driven Development in Operations

RANJIB DEY, PagerDuty

Application of Test Driven Development in Operations is new and there are limited resources. But when adopted strategically, it can help reduce the tactical workload and enable greater automation, while bridging the gap between development and operations domain specific differences.

1. INTRODUCTION

Test Driven Development has been popular in mainstream software development for more than a decade now. Though there are fair amount of learning resources available on Test Driven Development for application developers in various languages and technology stacks, domain specific resources are scarce.

IT Operations, which historically has been interrupt driven, is undergoing a major transformation due to cloud adoption and DevOps movement. Both of these factors influence heavy automation. Thus, increasingly IT operations is embracing codified representation of workflows and best practices. With this comes the concern of introducing changes against systems which are intended to provide stable and predictable ground for software deployments. In this experience report I have explored the benefits of Test Driven Development in introducing tactical as well as high impact strategic changes in operations, assuming an inherent bias towards automation for reproducible and predictable workflows.

2. BACKGROUND

Modern IT Operations involves routine maintenance and upkeep of an organization's internal infrastructure alongside supporting constant business innovation, in the form of new services, better tooling etc. Unlike software development teams, which strive for agility towards feature development and deployment, IT operations teams aim for an orthogonal objective of providing a stable system with known operational capacity, this historically creates deep division among the tools and processes that dominate IT operations versus application development teams.

While the depth and breadth of responsibilities of an IT operations team vary from organization to organization, almost all of them involve managing fleet of servers, providing mission critical services (e.g. identity management, inventory management etc), performing on-call duties (like outages) etc. Most of this introduces a varied amount of tactical workload. Vast majority of software and tools used by the operations teams cater specifically to these tactical works. As organization grows, individual operations teams build up their own automation suite in response to this growing tactical work. In rest of this paper we'll explore Test Driven Development in context of building and maintaining such automation suites under the premise of a basic strategy which aims to ease adoption, without disrupting system stability, but at the cost of initial correctness or completeness. Which means, the aim of the proposed methodology is to incrementally build up a bug free, maintainable operations code base, by iterating through a series intermediate phases where the code base or automation suite is always stable for production but may not be as sound or rigorous as prescribed by contemporary TDD approaches.

3. TDD ADOPTION STRATEGIES

Examples of using tests to guide software design and quality can be found as early as late 1950s, during early days of digital computing. Embracing the same philosophy and bringing it into day-to-day software development in the form of TDD was first popularized by Kent Beck through the SUnit framework, followed by the authoritative TDD book and through eXtreme Programming(XP). Modern TDD has evolved from its original inception to more complex concoction of different styles (like ATDD aka Acceptance Test Driven Development ,

Author's address: Ranjib Dey, 1075 Foster City Blvd, Foster City, CA; email: ranjib@pagerduty.com

BDD Behavior Driven Development). They differ on the definition of unit amount work, but underlying principles of introducing features alongside corresponding tests remain same.

For most of the operations team, daily automation work involves assembling more than one tool and code base. Thus overall testability of an entire workflow is a function of the component that is least tested. For the rest of the document I'll be discussing strategies for applying a combination of various TDD approaches in different stages and areas of operations workflows so that the net benefit of TDD is realized gradually and holistically, without making any particular component or system as primary target for TDD.

3.1 Invest on learning system specific to TDD

Learning resources are pivotal for success of any new practices, development methodologies. Team members will require both offline reading resources as well as just-in-time, practical tips and problem specific solutions.

3.1.1 Build up domain learning specific resources

A reading list on TDD and its applications in operations. There are several online resource available on this topic[10]. From theoretical papers on specific operations tools to many resources on TDD[9] are now available. Always prefer the resources that explore TDD concepts and principles using a tool that the current teams familiar with. It is also helpful to maintain code examples or snippets for elucidating or annotating some of these learning resources. Blogs and public wikis are also becoming a hotbed for such information. Building up an internal learning resource that enlists TDD related concepts in context of existing tools help reduce the learning curve for newcomers.

3.1.2 OpenSource communities and projects are best places to learn

A vast majority of popular operations tools are open source. Most of these tools internally follow test driven development. Encourage teams to use those projects for learning how TDD helps maintaining large code bases. Most of them are concrete examples of TDD in practice. It is also easy to find mentors and help in those broader communities. Infrastructure codebases for some of the linux distributions (e.g. fedora) are also developed openly, they provide a more holistic view of an IT organization than individual tool specific repositories. Hence they are also invaluable learning resources. At PagerDuty we actively participate in Chef and several other OpenSource project development. We have also Open sourced several of our internal projects which helps us received feedback from a wider community.

3.2 Target known components to explore TDD

Every new learning comes at a cost. For operations teams this means reduced capacity to deliver tactical work in the beginning. It also means writing a bit of extra code. And if the whole TDD is unpleasant for IT operations team, it will fail, hence strategize TDD introduction for low impacts.

3.2.1 Consider the smallest. most frequently changing scenarios as first target

Known problems are easier to work with. Given that the test framework and TDD principles will be new to the team members, it helps to keep the focus on TDD concepts. For operations, often time tactical work involves rolling out new systems about which the operator does not have a priori knowledge (like a new load balancer software). Such scenarios are less than ideal for new adopters. Effective use of TDD will likely involve introduction of tests in iterations, hence it will be easier to explore TDD with a component that undergoes frequent changes (like provisioning a server, or installing a package).

3.2.2 Use common vocabulary for operations codes. Establish patterns

Since operations involve multiple tools and individual tests depend on the corresponding tool, apply the generic 80-20 design rule. Write tests for workflows and tools that used 80% of the time by the team. Even though there are huge numbers of operations tools, vast majority of them can be categorized with a few basic patterns. For example configuration management and monitoring systems alone constitute the large part of automation suite. Similarly within the configuration management system, bulk of the codebase involves implementing few common patterns. Establishing a common vocabulary for these operation specific patterns helps in communication. Each of these common patterns can have their known boundary conditions which can be tested by common techniques. Identifying this patterns helps in accelerating development of test suites against them. At PagerDuty we use several design pattern specific to operations code base, such as external services memoization, composite resources, authoritative resources etc.

3.3 Design a layered testing strategy

Contemporary TDD resources explain TDD using unit testing only. In practice operations involved assembling disparate tools, and a combination of different testing methodologies is often time more effective than a single unit testing suite. Depending upon the composition of tools being used and problem domain, different styles of testing should be employed to cater differing requirements.

3.3.1 Unit testing for easy maintenance, better designs

Unit tests are the fast, granular, white box(tests are aware of or coupled with implementation) tests. They improve system design (complex parts are harder to test) by incentivizing decoupling and simplicity. They also provide fast and early feedback (with most external dependencies being mocked out). Unit test help most in maintenance of rapidly evolving code bases, where code contributions originate from different individuals at different subsystem. An example unit test candidate will be a chefspec-based test for a chef recipe for apache installation. At PagerDuty we use a comprehensive unit tests that executes several thousands of assertions in less than couple of minutes against our infrastructure code base. This helps us proof checking code contributions from external groups quickly.

3.3.2 Functional testing for component level behavior correctness

Functional tests, on the other hand are mostly black box (tests are not aware of or decoupled from implementation details), and time consuming. They are used to ensure overall correctness of behavior, to better understand end user requirement etc. Because the functional tests are time consuming they are often triggered periodically (unlike the unit test suite which runs after every code change). An example functional test will be a serverspec-based test for setting up apache and ensuring that port 80 is network bound. For example, at PagerDuty we use both container backed integration tests as well as Chef audit mode for functional testing.

3.3.3 Integration testing for behavior correctness for the whole system

Since most operations code results in target systems for application deployment, integration testing is pivotal. In most cases they'll involve combining both operations and application code bases as a unified solution and testing the resultant system as a whole. These are the most time consuming tests, and generally key performance indicators are codified and asserted via integration test suites. Because of their nature, integration tests can be triggered from a variety of sources (like changes in operations code base or application code base or while provisioning new infrastructure etc). While doing major architectural changes (like migrating from one persistence layer solution to another), integrations tests can effectively help test drive the whole migration process.

3.4 Introduce and embrace CI for operations

Continuous Integration is an established methodology of running tests outside developer machines, independently, upon every changes against a codebase. Existing CI systems can be easily introduced to operations codebases to ease TDD adoption.

3.4.1 Make failures affordable, build staging environments

Having a continuous integration system for operations code base makes failure visible. This alongside staging or test environment for operations codebases reduce the cost of failures. This does not remove the risk of individual change that is being introduced, but they split up the risk into multiple phases, and allow catching them in a sandbox environment. From any operations perspective often time it's difficult to create production sandbox environments, but an initial dissimilar sandbox environment can help building a more production like environment iteratively, in conjunction with the test suite and feature development.

3.4.2 Extend feedbacks from CI to main communication platform.

Different teams communicate through different mediums. Depending upon the organization this can be internal chat or voice application, or emails. Extend the feedback provided by continuous integration systems (like build failures, code coverage changes etc.) into these communication mediums. This increases the visibility of TDD process as it is being applied. It also helps consolidating feedbacks as individuals can initiate a communication in context of a test failure. At PagerDuty we use both Travis CI (a hosted CI service) and GoCD (an in house build-ci farm), and both of these system send build-related notification to our main communication medium which is hipchat.

4. Lessons learned

4.1 Treat operations as any other functional domain

Treating operation as any other functional domain (like mobile or finance) encourages reusing existing software development techniques and treating operations team as any other software development team, but with constant tactical workload. This allows the teams to use their testing strategy to counter tactical workload (like what failures can be mitigated to reduce most common tactical works). Maintenance work can be treated a tech debt, and systems involving high maintenance can be subjected to greater TDD to lower the recurring maintenance requirements.

4.2 It is fine to have poor design in the beginning

For most scenarios TDD in operations is fairly new. It is new for the folks who are maintaining the existing infrastructure, it is also new for the ecosystem (tools and community) as whole. It is important that the individuals who are adopting TDD feel they are productive and they stay motivated throughout the journey. This wont be the case if an upfront steep code quality requirement is imposed. Which often means time, codes having known smells will be introduced, and necessary time for addressing all those smells will not be available or the knowledge required to understand those issues will not be available. This is fine, it's best to take an iterative approach and improve things in passes.

4.3 Use tools that grows with the team instead of making a system blackbox

Tools affect a lot in TDD adoption. Tools that treat operations as any other codebase and use mainstream programming languages are easier to work with in contrast to tools that use custom domain specific languages or serializable data (like yaml) for describing operations, these are harder or limited to TDD as they require custom testing frameworks or high level composition can not be done (with serializable data formats like yaml and json) with them. On the other hand tools that are used using mainstream programming languages (like chef uses ruby, fabric uses python etc) grow with the team. Teams can learn and introduce TDD concepts from a larger community of developers using the same language for other domains.

4.4 Encourage systems thinking

The net effect of TDD is subtle in immediate future and profound in the long term. Understanding the big picture helps both in justifying the initial learning curve as well as devising a proper strategy. Depending upon individual teams need, they might choose different style of tests and associated tools. Understanding the whole business system (operations code along with application code) helps in prioritizing the appropriate testing strategies.

4.5 Adopt existing TDD techniques for operations

Since most existing books in TDD are targeted towards application developer, often time operations teams find the existing resources are out of context or inapplicable. It's not only the code, but also the corresponding assertion, that should reflect techniques used in in context of operations. For example a functional test involving zookeeper (a key-value store) can use telnet (a popular command line tool in operations for debugging network services) for assertions.

4. ACKNOWLEDGEMENTS

ThoughtWorks and all the amazing ThoughtWorkers for introducing agile & TDD. PagerDuty operations team, for embracing agile and giving the opportunity to apply TDD. Tim O'Connor for the review of this paper. Andrew Crump and Seth Vargo for ChefSpec.

REFERENCES

(1) D.D. McCracken. "Digital Computer Programming" 1957
(2) Kent Beck. "Test Driven Development: By example"
(3) Glenford J Myers. "Software Reliability: Principle & Practices"
(4) Andrew Shebanow. "A brief history of testing frameworks" http://shebanator.com/2007/08/21/a-brief-history-of-test-frameworks/
(5) Seth Vargo. "Test Driver Infrastructure" at 2013 Velocity Conf. https://speakerdeck.com/sethvargo/test-driven-infrastructure-with-chef
(6) ServerSpec - An integration testing frameworks for servers. http://serverspec.org/
(7) Ranjib Dey. "Operations tools patterns" . https://dzone.com/articles/infrastructure-tooling-patterns-list
(8) Fedora infrastructure repo. https://infrastructure.fedoraproject.org/cgit/ansible.git

(9) Stephen Nelson Smith. Test Driven Infrastructure with Chef, 2nd Ed
(10) Mark Burgees. Testable System Administration, ACM Queue,System Administration,Vol 9, Issue 1
(11) Chef testing at PagerDuty - https://www.pagerduty.com/blog/chef-testing-pagerduty/