Contents

Part I

Front Matter

Chapter 1

Preface - JDD Briefly

Journey-Driven Development (JDD) is how we create software systems for clients at VMLY&R. The phrase, 'for clients,' is important to us as it reminds us that our clients judge our work not only by how it runs but also by examining its construction critically as it is developed for hire.

We wrote JDD for ourselves, the developers, analysts, testers, architects, and operators of the software we create for our clients. We share it as we believe the lessons learned are useful to software teams generally. You don't have to code to benefit from JDD but the subject matter is technical.

We work at an agency, VMLY&R. Sometimes we say advertising agency and sometimes we say marketing agency. We think of ourselves as agents of change and transformation.

Our software is delivered quickly. Our clients are demanding. We often guide the software through the initial launch and sometimes through years of updates, and our clients own the software we create and operate it long-term.

Goals:

1. Make high quality software fast.
2. Make it easy to learn our process.
3. Make it easy to modify our process.

We discovered JDD and we teach it to our clients as it is easy to adopt in parts or as a holistic transformation. We've applied JDD for large-scale transformation efforts in automotive, telecommunications, travel, finance, transportation, and construction.

JDD begins with API Forward, which simplifies complex systems integration. API Forward is event-driven, focusing on the events arising from the user journey. We map these events through lightweight orchestration via a forward facing facade. We implement the facade in a messaging system. That keeps the

complexity of transactions away from our experience platform. See API Forward topic for details.

For the experience platform, JDD uses a technique we call Content Rocket. Content Rocket also begins with the user journey, specifically the audiences and moments discovered by our Content Systems practice. We map these to reusable content components that we can combine in different ways to both allow for specialization and maximize reuse. We've mapped Content Rocket to all of the major experience platforms. See Content Rocket topic for details.

None of this is completely foreign. We are informed and influenced by the conversations in our software development community. These conversations happen at conferences, in chat rooms, on blogs, and in books like this one. Here are some of our influences:

1. Agile (Rasmusson 2014) generally and eXtreme Programming (Beck and Gamma 2000) in particular
2. Software Development as Craft (Cunningham 2005)
3. DevOps (Kim et al. 2016)
4. Automation
5. Responsibility Driven Design (RDD)
6. Domain Driven Design (DDD)
7. Behavior Driven Development (BDD)
8. Test Driven Development (TDD)

Perhaps the first written version of Journey-Driven Development began as a one-page brief when a team that had been high-performing began to stumble. The memo served as a starting point for many other teams to define their process.

What follows is one fork of that document:

1.1 Start

The team begins when a technical lead and a project manager estimate the effort and staff the team.

- PM holds a dev kickoff. Each team member reads the scope, plan, and aligns their task with the larger ask.
- Everyone knows who the tech lead is.
- PM and tech lead are careful when they have to move resources mid-project and account for ramp time and communication overhead.
- PM and tech lead ensure team and individual accountability for hitting dates and budget. Be creative and escalate as needed.
- The tech lead validates dev team progress, setting estimates (and resetting remaining work as needed).

1.2 Execute

The team uses a project page or wiki to ensure up-to-date versions of these key artifacts are available to the entire team.

- Scope
- Project plan (or backlog), release plan, test plan, and defect list
- Requirements
- Wireframes, creative, and an Executable Style Guide.

Leader portion of weekly One-on-one Meeting references one or more of these documents (particularly the project plan).

The entire team:

- Attends stand-up (mandatory for dev team and PM).
- Refers to the above in their daily stand up using measurement language (e.g., "Yesterday I reviewed 3 requirements and I plan to finish 80% of 1 today").
- Ensures the plan is realistic based on the estimated backlog and team velocity .

1.3 Automate

- Automate builds
- Automate tests - you don't need to test everything, but complex areas should have regression tests.
- Automate deployment

1.4 Review

Review regularly. Pairing is faster than working alone and improves initial quality. Regular pairing gives us a structure for regular and frequent code review. Review before commit: All code has at least a cursory peer review before commit. This arises naturally during paring, but can happen otherwise.

Ensure that VML solutions architects conduct an Architectural Review at the beginning of every project and again just before launch. Long projects may warrant additional review.

1.5 Ensure

- Quality is the responsibility of everyone on the team.

Key points

- Before QA, developers verify in approved browsers in the development environment.

- Developers list incomplete or broken items from key documents in QA request.
- Requirements changes appear with screenshots in the test plan.

1.6 Demonstrate

Demonstrate your running system often (at least every two weeks). To the final client is better but an internal demo is better than no demo. At the end of the demo, revalidate the plan and budget. PM and Tech Lead are mandatory.

Key Points

- Conduct the demo from a QA or Stage environment (not a development environment)
- Make sure QA has completed a round of testing prior to the demo.
- Do a practice run to see what is working well and what is still in progress.
- Invite AM, PM, & Creative to the practice run so they understand what the client is going to see.
- If you have significant issues, reschedule

1.7 Scale

We have more than 50 team leads skilled in this method which means we can quickly divide and conquer to bring timelines in and tackle emergent opportunities as they arise. We've thought a lot about how to create and empower teams so that both experimentation and collective learning grow as your team expands.

Bringing all of this together has enabled DevOps through JDD which has led to dramatic improvements in delivery. At one client, we moved them from 10 releases per year to more than 10 releases per day in production using canary releases with a dramatic increase in quality: we had a 10 month stretch of production releases with no severity 1 or 2 defects reported (a company best).

1.8 Reinvent

Our process is organic and ever-changing; created, modified, and reborn by hundreds of 5-9 person teams over decades delivering software systems.

We talk about this process internally in meetings (and hallways) as well as externally at conferences. In 2019, we had the idea of naming and versioning our process. From that came the idea of open-sourcing the process.

This book is the result. We know this book is on its own journey. We hope to continue to modify it after publication. We hope that you use and modify it as well.

Chapter 2

Dedication

To the individuals on our teams defining and redefining Journey-Driven Development with each delivery and retrospective. Though many have contributed topics to this book through numerous commits and pull-requests, even more have contributed ideas.

Chapter 3

Epigraph

'When I use a word,' Humpty Dumpty said in rather a scornful tone, 'it means just what I choose it to mean—neither more nor less.' – (Carroll 1897)

Chapter 4

Errata

We declared a 1.0 a while back (or was it 0.1). There is still so much to do. Adding this page to hold a changelog of sorts for future releases.

Chapter 5

Acknowledgements

The Pro Git book inspired us to use Pandoc for this.

Discovered after we started that Daniele Dellafiore wrote about something similar she called User-Journey Driven Development in August of 2015.

Thanks to Billie Thompson (a.k.a., PurpleBooth) for providing the starter contributing.md template, which we took from Good-CONTRIBUTING.md-template.md.

Many people contributed content to this book. A comprehensive list can be derived from the git history of our project, but what follows are some specific authors and sections (apologies for any omissions):

1. Chandler, Preston Agile at VMLY&R
2. Heryer, John: Journey Driven API Design
3. Martin, Jennifer: Maturing the Practice
4. Sykes, Jeff: DevOps

Part II

Process

On some level, this whole book is about process. We added a specific section on process because we believe that methodology concerns itself first with people, process, and tools. Through revisions, we extracted architecture from the tools list and techniques as a sort of smaller scale process. I'm struggling here to to avoid calling a technique a micro-process, but it seems clear to us that what remains here after removing techniques is a sort of macro-process, or process in the large.

Specifically, we think of Journey-Driven Development as a descendant of Rebecca Wirfs-Brock's Responsibility-Driven Design

. While we were not present at the signing of the Agile Manifesto, we find that manifesto a useful check to see if we are on the right path.

The DevOps journey at VMLY&R began in 2013 as development and operations teams searched for better ways to support the rapid pace of work in an agency. The scripts and manual processes that we had used were not sustainable or scalable enough to deliver as development efforts switched from Waterfall to Agile methodologies. That effort started with destroying the silos that prevented teams from working together. Our collaboration produced automation that enabled teams to build, unit test and deploy code rapidly with greater confidence. As that automation matured, incremental improvements added additional testing from our Quality teams to implement stricter quality gates into application deployment pipelines.

Because DevOps is more about removing silos, we treat automation, scripts and pipelines as tools, not the product of DevOps. We are committed to measuring, refining and improving those tools to improve our application delivery process. We constantly ask the questions: "how might we improve this process?" "What did we learn from this experiment?" "Why did this failure occur?" Asking these questions is the key to learning and continually improving.

When focusing on customer journeys, firms realize that agility and experimentation are key to unlocking value and testing insights. In 2001, the Agile Manifesto articulated a new vision of software development aimed at unlocking value faster. This value creation focused squarely on removing the barriers between business and development in order to deliver software that meets customer needs. Traditional application delivery and deployment techniques seek to improve system stability by slowing the rate of change. By controlling, documenting, and testing changes, organizations attempted to mitigate risk. The competing aims of Agile and preserving system stability lead many to disillusionment with Agile.

The promise of Agile development practices is hampered or lost when traditional by following traditional application deployment and delivery techniques. Unlocking the promise of Agile and enabling the kind of experimentation necessary to learn what customers want requires increasing the rate of change not slowing changes down.

DevOps seeks to tear down the barriers between those responsible for developing

and operating software. As barriers fall, operational processes must evolve in order to deliver speed and stability simultaneously. Efforts that do not require both speed and stability as equal goods are sure to deliver neither.

Chapter 6

DevOps

DevOps is a set of cultural, architectural, and technical practices for teams developing and operating the software that enables customer journeys and delivers value. DevOps applies principles from improvement of physical manufacturing to the IT value stream. DevOps depends on a culture that breaks down silos between the work of development, operations, quality and security. Architecturally, DevOps includes decisions to organize infrastructure and applications for agility and resiliency. The technical practices include techniques for automation of application delivery and deployment, infrastructure as code, and automated testing.

6.1 Manufacturing and IT Value Stream

Physical manufacturing turns raw materials into finished goods for sale. The raw materials for software are business hypothesis resulting from insights into customer journeys. The IT value stream describes the process to convert business hypotheses into working software. DevOps uses techniques from the Lean and Toyota Kata movements to optimize the IT value stream the way those techniques optimized physical manufacturing. Gene Kim, author of *The Phoenix Project* describes the "three ways" essential for all DevOps practices.

The Three Ways:

- First Way: Flow
- Second Way: Feedback
- Third Way: Continual Learning and Experimentation

DevOps optimizes flow in the IT value stream to minimize lead time. Lead time is the amount of time it takes for a feature to move from being an idea to being available to a customer. Optimizing flow depends on reducing the size of work

moving through the value stream (i.e. reducing batch size) and building quality into the process.

Feedback loops are essential to improving flow. Feedback loops allow teams to prevent problems from recurring or detect problems more quickly to enable recovery. Without Feedback teams cannot learn or improve.

Building off of Flow and Feedback loops, teams experiment with improving flow and use feedback loops to learn if an experiment is successful or not. As these experiments yield success and failures, the team learns how to optimize or change processes to increase speed and safety. Running many smaller experiments helps teams manage risk and build resiliency into systems.

6.2 Cultural Practices for DevOps

The "three ways" mentioned above depend on a shared purpose and trust. Teams need to make promises and then deliver on those promises in order to build trust. That purpose and trust helps drive the change necessary to find new ways of working. Trust is vital because new ways of working creates fear around losing jobs or losing importance.

As teams share information minimizing tribal knowledge and silos, the value people bring shifts. Value shifts from undocumented knowledge of how to manually complete a task to the ability to work together to automate repeatable tasks. As manual tasks are automated, teams need to take opportunities to learn and improve as toil and the tyranny of the immediate lift. Technology's continual evolution and ever increasing pace of change mean that teams need feedback loops as bottlenecks shift or move. Improvements in one area may cause unintended side effects elsewhere.

Emphasis on culture, as opposed to tools, rightly moves the focus for DevOps efforts from something one purchases to a new ways of working. While tools are an important part of any culture, those tools apart from the culture may create problems. For example, a centrally managed continuous integration/delivery tool, may just create a new silo instead of empowering teams and fostering collaboration. Anti-patterns in behavior reoccur or new challenges surface as the problems teams tasked with developing and operating software change. Teams working to build and sustain positive culture realize their work is never complete.

6.2.1 Making Work Visible

Work that teams do to operate and develop software needs to be visible to management and project leadership. When team members do work that originates from email, chat or other avenues that is not centrally tracked the opportunity to learn and search for patterns in work evaporates. Too often teams operating software take and complete tasks without recording that work occurred. That

failure leads to questions about the efficacy of teams and how they are spending their time.

While ticketing systems may at times indicate silos in organizations, work that cannot be seen by team leads and managers is work that cannot be managed. A priority must be a honest accounting of the effort necessary to develop and operate any software solution. The exact means of that accounting may vary based on the team size and structure; however, failing to make work visible will obscure patterns that may help identify bottlenecks and opportunities for optimization.

6.2.2 Distributed Decision Making

As teams make work visible and find opportunity, those teams need to be empowered to make changes. Trust, built on promises made and delivered, should be matched with goals and direction described in terms of intent and purpose. When communication around intent and purpose is not clear enough or when teams make mistakes, those mistakes should be seen as opportunities to learn rather than something to be punished. Teams afraid of punishment will hide reasons for failures preventing improvements that might prevent future failures. Celebrating and creating a culture of learning must include embracing errors and failures.

For example, deployment of broken code should lead to an investigation of how the failure occurred and how it might have been avoided. A culture of blame might ask, "how did a person make this mistake?" A culture of learning would want to understand how a choice seemed rational and how processes might be modified to prevent a bad outcome. Removing decision making power actually masks problems and obscures the improvements necessary to improve process safety.

6.2.3 Sharing Responsibility to Create Change

Change occurs most rapidly when the decisions of a team impact all members of a team. If code that is brittle or fails regularly impacts the team operating a system with little to impact on the team developing the system, then there will be less incentive for change. Teams where the work of developing and operating software share both the pain and success that comes from a system. Sharing responsibility prompts necessary discussions around how non-functional requirements are met and delivered.

6.3 Architectural Practices for DevOps

We need to recognize that architectural decisions are also a shared discipline. Optimal solutions require teams with multiple disciplines and different experiences. What is obvious to one architect as a serious risk, may look completely

non-consequential to others.

6.4 Technical Practices for DevOps

Often DevOps is associated with the technical practices or tools used for technical practices. Adopting DevOps technical practices in isolation may lead to some positive outcomes; however, the greatest impacts come when these technical practices are combined with the architectural and cultural DevOps practices.

6.4.1 Automated Deployments

Many DevOps discussions begin with Continuous Integration (CI) and Continuous Delivery (CD).
CI includes practices of constantly integrating branches back into main branch. Short-lived branches reduce the batch sizes of changes coming back into the main branch and reduce the risk of conflicts when merging branches or pull requests.

6.4.2 Automated Testing

6.4.3 Infrastructure as Code (IAC)

6.4.4 Instrumentation

Instrument code to observe behaviors coming out of a system.

6.5 Metrics for DevOps

1. Lead Time
2. Deployment Frequency
3. Mean Time to Recover (MTTR)
4. Change fail percentage

See also:

1. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*, Gene Kim, Jez Humble, Patric Debois & John Willis
2. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*, Gene Kim, Keven Behr, and George Spafford

Chapter 7

Journey of the Process

Part III

Architecture

Chapter 8

API Forward

API Forward is a pattern for evaluating a technical architecture. Why forward:

- forward from behind the CMS
- forward in front of systems of record
- forward in time (like API First) before reference implementation

8.1 Forward From Behind the CMS

Content Management Systems (CMS) dominate the technical thinking of marketing teams. We often get feedback when we show diagrams that don't have the CMS or digital asset management platform (DAM) at the center, coordinating activity.

This makes sense for sites that are purely informational. As sites begin to offer transactions, several patterns emerge:

1. Serverless. Use other people's services (and servers). Your clients already use other services; you can too. Prefer API Forward for custom services.
2. CMS-hosted services. Host your custom services alongside your CMS' existing "headless" content services. Convenient in the near term. Prefer API Forward as scale or complexity increase.
3. API Forward or something similar with a different name. You may derive API Forward from first principles. Related patterns like Service Facade, Facade and Front Controller share some structure and consequences. Consider API Forward as a framework to evaluate your solution's fit for modern marketing.

In short, consider API Forward as you move from content-only websites. Specifically as you add commerce, other transactions, or even as you add things like chatbots as conversational state and transactional state are analogous.

8.2 Forward of Systems of Record

When logically organizing architectural components for modern marketing, several patterns emerge:

1. CMS at the center
2. Stranger Danger
3. API Forward

8.3 Forward in Time

When developing an API, several patterns emerge:

1. Implementation First (a.k.a Reference Implementation). The implementation is the specification. API Forward recommends coding against a specification, which sounds more like...
2. API First (a.k.a. Contract First, Design By Contract). The implementation must meet the specification. API Forward leverages the technical approach API First in that we sketch a front-end UI design to translate the end-user's needs to an API endpoint.
3. API Forward or something similar. Methods like Responsibility-driven design (RDD) and Domain-Driven design (in particular the idea of Ubiquitous Language), may yield an API similar to one aligned with API Forward. Use API Forward as an iterative approach aligned with modern marketing.

When: Adding transactions. Also consider in conversational commerce, as conversations are transactional as well.

8.4 Uses Customer Language

API Forward is journey-driven. That is, it uses the the language of the customer, not the language of enterprise components; you likely have existing API endpoints that may not be written in customer language. We do not expect to rewrite them. Instead, API Forward tends to lead to a layer of orchestration aggregating and sequencing lower-level APIs. API Forward lets you leverage existing and future enterprise investment.

8.5 As a Pattern

8.5.1 Situation

8.5.2 Consequences

8.5.3 Examples

8.6 Next Steps

8.6.1 Accelerators

8.7 API Forward and Microservices

Microservices.io provides a good baseline for understanding what we mean by a microservice:

1. Highly maintainable and testable
2. Loosely coupled
3. Independently deployable
4. Organized around business capabilities
5. Owned by a small team

Most modern discussions of microservices include a few more expectations, such as:

1. Microservices execute and participate in a service mesh
2. There is a consistent and predictable contract across services, at a macro-level
3. Are easily composable with other services
4. Fine grained, distributed traceability
5. Graceful fault tolerance and resiliency

Unpacking the service mesh a bit more, we find these additional expectations:

1. Dynamic registration/discovery
2. Independent and precise auto-scaling
3. Health check telemetry
4. Polyglot support

Chapter 9

Content Rocket

For brands, personalization at scale is the marketing holy grail. But when people try to create these experiences, it is easy to get stuck in silos: attempting to deliver a personalized experience using disconnected technology. Personalization can become more about internal turn wars than anticipating customer needs.

Content Rocket is an end-to-end framework that puts the customer at the heart of our efforts and illustrates how we deliver personalized experiences at scale.

9.1 Data

Everything starts with a customer, which is to say, everything starts with customer data. Start by gathering qualitative and quantitative insights using both first and third party data to better understand our customers, their needs, and their journey.

9.2 Strategy

In a content process named Content Rocket, you might be surprised that we also consider experience strategy in concert with content strategy to uncover the best use of content in context of the journey. Experience can be a fundamental unlock for putting content into this context.

1. Experience Strategy
2. Content Strategy

9.2.1 Experience Strategy

1. Personas
2. Jobs Theory

3. KPIs
4. Value-centered design
5. Think-Make

9.2.2 Content Strategy

1. Editorial
2. Product
3. Campaign
4. Offer
5. Utility

9.3 Asset Creation

1. Audience story
2. Core assets
3. Copywriting
4. Proofing
5. Editing
6. Translation
7. Legal review
8. Tagging

9.4 Activation

Primary tasks:

1. Identify
2. Assemble
3. Serve

Platforms:

1. Digital Asset Management
2. Content Management
3. Customer Relationship Management
4. Customer Data Platform / Data Management Platforms
5. Dynamic Creative Optimization
6. Artificial Intelligence / Machine Learning

9.5 Journey Audience Map

9.6 Evolution

9.6.1 Reuse

9.6.2 Refine

9.6.3 Reevaluate

Part IV

Techniques

Chapter 10

Journey-Driven API Design

Journey-Driven API Design is a contract-first approach to API creation. Contract-first means the implementation must meet a specification. The result is customer value alignment with technology through a well documented and designed API that services multi-channel client interactions.

Journey-Driven API design consists of three primary phases of analysis:

- Domain
- System
- Persistence

The activities in each phase are ordered and should be worked continuously and iteratively throughout a project. Each step should have a design artifact for documentation, typically a technical diagram. Document significant architectural decisions as an Architectural decision Record.

10.1 Document the decisions

A technical decision may be "architecturally significant" due to its potential for long-lasting impact. Context is lost as projects evolve, and it is vital to capture the architectural decisions made. Preserve essential aspects through the design process with Architecture Decision Records.

10.1.1 Architecture Decision Record

Capture architecture decisions with an Architecture Decision Record. The format consists of a minimal set of information needed to understand: factors that influence a decision specific items decided anticipated outcomes of the decision

10.2 Resource Analysis

Resource Analysis provides a technical architect the inputs required to bridge technology and customer value. These activities focus on understanding the problem domain and requirements to guide API design.

10.2.1 1. Gather Functional Requirements

Understand the problem domain by working with business stakeholders and product owners to learn how customers will interact with the system. The Functional Requirements Document (FRD), User Stories, Statements of Work, and even UI/UX inputs can help inform the API design. Common scenarios included customers interacting with a web or mobile application supported by APIs.

10.2.2 2. Gather Non Functional Requirements

Non-functional requirements (NFR) are the constraints on operational behavior. Information such as response time requirements, performance budgets, accessibility requirements, legal and regulatory compliance constraints are inputs into a technical system design. These constraining attributes need to be realized early and designed into the architectural foundation of the system. Work with business and product owners to define non-functional requirements.

10.2.3 3. Create a Domain Model (Conceptual ER Model)

A domain model is a living document, updated as requirements evolve. It establishes a ubiquitous language with domain experts and identifies fundamental entity types, behaviors, and relationships. Start a domain model by extracting key nouns and verbs from user stories or requirements. The domain model is a prerequisite to creating the API Contract.

10.2.4 4. Design an API Contract

Create the API contract first, so the implementation meets the specification. The OpenAPI spec is an Interface Description Language (IDL) to design and document RESTful APIs. The definition of an API with an IDL provides clear direction and acceptance criteria to API developers and consumers by way of a well-defined contract. The document should be tracked in source control with the code and updated as the API evolves.

10.3 System Analysis

System Analysis is valuable to technical and non-technical audiences. Analysis with diagrams captures the current and future state of systems, including their relationships at several abstraction levels. Start with the highest level that

delivers value by identifying who is using which system. Zoom into individual technical systems to visualize the deployable containers that represent application or data. Finally, capture an understanding of how application functionality is related and encapsulated down to the source code representation.

Describing how the API will fit into a new or existing system can be difficult. The C4 Model diagraming framework helps technical and non-technical audiences navigate and communicate complex software systems. The following steps describe the parts of defining and documenting a technical system using C4.

10.3.0.1 1. Create a System Context Diagram

A system context diagram provides a starting point, showing how the software system in scope fits into the world around it. This diagram describes the type of software system, who uses it, and how it fits into an existing environment.

10.3.0.2 2. Create a Container Diagram

A container diagram zooms into the software system in scope, showing the high-level technical building blocks (containers) and how they interact. This diagram shows the high-level technology decisions of the architecture and how it distributes responsibility. Additionally, it illustrates the relationships that containers have to each other.

10.3.0.3 3. Create a Component Diagram

A component diagram represents the pieces into an individual container. It demonstrates the high-level structure of an application by describing the responsibilities of components and how they interact.

10.3.0.4 4. Create a Code Diagram

UML Class diagrams and package diagrams describe how to implement a component as code. There is value in proactively creating UML Class and Package diagrams to communicate important software design concepts and patterns.

10.3.0.5 5. Document Sequences and Interactions

Capture interactions between components. Sequence diagrams visualize the message flow as a communication tool to describe complex interactions over time. Documenting advanced interaction paths clarifies processes to non-technical audiences and informs acceptance criteria used by development teams.

10.4 Persistence Analysis

If the API needs persistence, create a logical and physical data model. Use the domain model or conceptual model from the Resource Analysis phase as an

input.

10.4.0.1 1. Create Entity Relationship Diagram (Logical Model)

Expand upon the domain model and create an (ERD)[] as a logical representation of the data entity properties and relationships.

10.4.0.2 2. Create Database Schemas (Physical Model)

Build on the ERD created and database schema. If an RDBMS is needed, a schema is required. Design a schema for systems using non-relational data storage.

10.4.0.3 3. Create a caching strategy

Data access patterns and available technologies should drive the definition of a caching strategy. The performance-focused NFRs should be an input to the caching strategy. Identify available cache solutions, such as Akamai and Redis, to address data at different architecture levels. Describe a caching strategy by detailing use-case scenarios around data access, caching, and eviction policies.

Chapter 11

Maturing the Practice

Maturing a practice is a battle against decay and entropy. You require an infusion of energy and purpose; maturation never happens by default. High performing development teams hold a commitment to maturation because the quality of the software depends on it.

Evolving a team starts with a decision from somewhere in the hierarchy. Identifying a path for affecting those changes can be as challenging setting the direction.

Approach the process with a combination of directive and suggestive measures to allow the practice to fulfill software quality expectations with fully invested developers who embrace team values and demonstrate ownership of results.

11.1 Directive vs Suggestive Approaches

The *directive* approach hinges on the authority to enforce a mandate and is effective when the team has little option beyond compliance. For example:

- Edict 1: PR's must have two reviewers before being merged
- Enforcement: Configure github project to require 2 approvals before merging
- Result: All merge PR's have received 2 approvals

Mission Accomplished. When the letter of the law is all that is required and when the decision maker has the authority to compel compliance, this is a simple, direct path.

The vulnerability of the directive approach surfaces when the decision maker is looking to change internalized values.

- Edict 2: Developers resolve linting violations before merging a PR

- Enforcement: Configure github project to block merging if there are linting violations beyond a reasonable threshold
- Result: Some developers resolve the violations; other developers raise the violation threshold

Mission Failed. At this point, enforcement becomes a bit more harsh and possibly personal. It risks affecting the cohesiveness of the team. What is needed here is for team members to embrace the value of minimizing linting violations and to own the practice of resolving them immediately. Suggestive tactics are needed to fill the gap.

The *suggestive* approach works like harvesting a crop. There is soil to prepare, as well as seeds to plant. Then, with purposeful cultivation, there will be a crop to harvest: At an abstract level, it looks like:

- prepare the soil -> build trust and establish credibility,
- plant seeds -> sell the value of proposed evolution, and
- cultivate the crop -> capitalize on timely opportunities.

A suggestive approach takes time and patience. Additionally, there is a synergistic benefit of the rich soil of trust and credibility being available for sowing other seeds. In reality, most attempts to mature a practice will benefit from a combination of both approaches.

11.2 Example: Increase Unit Testing Adoption

You want to increase adoption of unit testing. First consider evolving your internal practice where you have authority to set a directive and then consider evolving your client's practice where you might not be able to set a directive.

11.2.1 Evolve Our Practice

How might we evolve our practice to include more unit testing. Consider three approaches:

1. Strictly directive
2. Strictly suggestive
3. Suggestive with directive

Strictly Directive: Issue an edict: Reject all PR's without unit tests.

- Many developers feel resentment at being forced into a practice they will argue is not valuable
- A few developers of a certain mindset accept the challenge and become good testers
- Others get really good at finding ways around testing
- Still others leave the team

This brute force way produces some testers, but it very likely poisons the environment for future changes and results in practices that evade the mandate without fulfilling it.

Strictly Suggestive: make team members feel safe, convince the team of the value of testing, remove obstacles to testing

- Probably fewer feelings of resentment
- More developers, but not all, become good testers
- Fewer developers, but not none, find ways around testing
- Likely none will leave

Because resistant team members avoid adopting the practice, this is approach is not successful.

Suggestive with Directive: use the suggestive approach along with rejecting PR's without unit tests

- Dramatically less resentment
- Dramatically more good unit testers
- Very limited avoidance or attrition

Eventually, we adopt the practice fully.

11.2.2 Evolve Our Client's Practice to Include Unit Testing

The twist here is that we lack direct authority to use the directive approach within a client organization. We begin with suggestive strategies to influence developers to embrace unit testing and to influence client leadership to eventually include directive strategies.

Turns out suggestive fluency is a super power. A confluence of micro actions point to the overall objective. Every context requires its own implementation details. Here are some ideas to start with.

1. Prepare the soil
2. Plant the seeds
3. Cultivate: Capitalize on timely opportunities

11.2.2.1 Prepare the Soil

We have 3 goals when preparing the soil:

1. Build trust
2. Establish credibility
3. Avoid harming the environment

11.2.2.1.1 Build Trust Make it abundantly clear that our goal is to make the client's life easier and project better

Demonstrate an urgency to relieve client devs' pressure points

Work on personal connections with all client devs

11.2.2.1.2 Establish Credibility Walk the walk. Include unit tests in all PR's, with exceptionally rare exceptions

Drop everything if someone wants to pair. Drop everything faster if someone wants to pair on testing<

Demonstrate a willingness to do the heavy lifting of establishing a practice of testing

Communicate personal story of learning to unit

11.2.2.1.3 Avoid Harming Environment Do nothing that could possibly communicate an attitude of superiority

Never diminish the objections to the challenges of testing

Readily acknowledge the learning curve for testing is an investment

Avoid sounding righteous about testing. A good technique is to "blame" VMLYR leadership for the requirement to test: "We're graded on testing our code"

11.2.2.2 Plant the Seeds

We have 3 goals when planting the seeds:

1. Testing is beneficial for developers
2. Developers Own the Code Quality/Success
3. Unit Testing is inevitable

11.2.2.2.1 Testing is beneficial for developers Identify and respond to the primary objections to testing by facilitating group discussions

11.2.2.2.2 Developers Own the Code Quality/Success Make a concerted effort to replace 3rd person pronouns with 1st person pronouns:

- THEY should -> WE should
- The Business wants -> WE want
- QA's bugs -> OUR bugs

Move from "we are victims of this mess" to "we are mastering this domain"

11.2.2.2.3 Unit Testing is inevitable Regularly make statements that assume the change will be implemented:

"When we have better test coverage, we won't have so many bug fixes regressing"

"When we have better test coverage, we'll be able to make these changes easier"

"Your job is going to be so much easier when our coverage improves"

11.2.2.3 Cultivate: Capitalize on Timely Opportunities

We have four opportunities to capitalize on when cultivating:

1. Retros
2. Light Sprints
3. New Hires
4. PR Reviews

11.2.2.3.1 Retros Since retros produce action items, this is a great place to talk about anything related to lack of testing.

Celebrate improvements in testing.

11.2.2.3.2 Light Sprints Identify tech debt that could promote unit testing. Get tickets created, and lobby for that work to be included in sprints.

11.2.2.3.3 New Hires Develop allies by onboarding new client hires and incorporating pairing and testing into their orientation.

11.2.2.3.4 PR Reviews In the comments of PR's without tests, provide code snippets for adding tests or offer to pair for writing tests to cover the work.

Withhold PR approval if there are no tests. Important to read the room here and time this well. Under certain conditions, this could be perceived as aggressive and do more harm than good.

Part V

Pattern Languages

Part VI

People

Chapter 12

Journey of the Software Professional

Chapter 13

Manager Tools Adoption

Chapter 14

Evolution of Teamwork

Chapter 15

People Matter

See also:

1. Peopleware: Productive Projects and Teams
2. Multipliers Book
3. Radical Candor: Be a Kickass Boss | The Book

Part VII

Tools

Part VIII

Future Topics

Chapter 16

Agile Release Planning

Answers the question: How do you estimate project costs, client pricing and delivery timelines? This is a favored method for coming up with an agile release plan. We have used this method successsfully on a number of projects. We adapted this approach from training by Gregory Smith of the Agile Alliance. Once we have a story backlog, we move through a series of steps to create a release plan that maps out the stories into sprints. The plan addresses how long the project is estimated to take, and how many points per sprint the team can take on. We take the anticipated staffing and development cadence along with the timeline and derive project costs.

STEP 1: ESTABLISH THE TARGET VELOCITY

First we take the stories and lay them all out on a big table. We start with wild guess groupings. No scale or points are disclosed to the team as a basis for evaluating relativity.

The point here is to establish relativity to one another in size groupings, without indicating precision. For example, some may use a scale of sm, med, lg, xl, xxl . But we need to steer clear of qualifying a medium as "exactly twice as big as a small," or a large is "three times a small," etc. We seek rough sizing relative to one another. Med is simply more complex than a small, large more complex than med. To emphasize relativity and remove any precision in the scale (and have a little fun), VML favors "fruit sizing" using a planning poker deck consisting of grape, orange, coconut, pineapple, watermelon.

1.1 CALIBRATE STORIES

After assigning relative sizes to stories, the team goes through an exercise in calibration of the sizes. They group by size and look across the group for consistency. Are all grapes really grapes, or are there some oranges hiding among grapes? We perform boundary analysis — look for the biggest small and the smallest medium and evaluate that the stories are in the right size group.

1.2 EVALUATE DISTRIBUTION

Another thing we do is plotthe distribution, or the number of stories in each category to evaluate the distribution of stories. Sometimes if the team lacks enough criteria to scale stories or the scale itself is not understood, a trend may arise where "everything's an orange" which is problematic, and shows the team needs to talk more or the scale may need to be addressed. If everything were XL that would be a similar problem too.

1.3 RUN SCENARIOS OF SIZINGS IN SPRINTS

Next the team goes through an exercise of looking at each category and estimating the number of stories per sprint (# per sprint) they can fit in. Assuming a two-week sprint, the team grabs as many smalls as they think they can fit in a sprint. Cycle through this three or four times for smalls, then move on to mediums and do the same. Repeat for the rest of the sizes and record the findings 18 (scenarios a, b, c, d).

1.4 ESTABLISH TOTAL SCOPE

We pick a scale to assign numerical weight values to the categories. A scale based on the Fibonacci sequence is used most often (grape = 2, strawberry = 3, orange = 5, pineapple = 8, watermelon = 13). The first step toward total scope is to calculate the value per fruit category by multiplying the number of stories by the corresponding numerical weighting. Total the values across the row and this value becomes your total scope.

Chapter 17

Code Review

17.1 Pull Request

17.2 Pairing as Review

17.3 Pride and Shame

17.4 Like, Wish, Wonder

Chapter 18

Turn Watermelons Into Orangeberries

On a recent transformation project, we used Fruit Sizing with Planning Poker decks.

A watermelon is a big story, effectively a marker to remind you to break it into smaller pieces. That sounds like work, slicing the watermelon into bit sized chunks. But a technology optimist sees a chance for transformation.

As is often the case, I draw my inspiration from Kent Beck, his eXtreme Programming process, specifically his admonition to DoTheSimplestThingThatCouldPossiblyWork. Question your assumptions.

Back to the work...

For the purposes of Agile Release Planning, our teams estimate their cumulative scope distance across 6 iterations is around 600 points (+/- 30%). That's just a sizing estimate but it helps product owners set priority. In this case, the team is looking at a 1200 point wish list. Time to make some cuts.

Fortunately, they had 10 watermelons (around 500 points).

While hanging out in one of the API rooms when a product owner came looking to talk with a pod lead about a watermelon. One goal is to reduce calls to the customer care centers. A big cause of calls is bill shock. Any change in the bill can trigger a call. Also, it turns out some people a more likely than others to call.

At first blush, sounds like a watermelon. But that depends on how you slice it:

1. Amount of change rule. Easy. Compare bill A to bill B and see if it exceeds a tolerance. If it isn't big, don't send.

2. Likely to call rule. Hard. Doing this well requires that we create a predictive model of customer behavior. Classic big data problem.
3. Do not contact rule. Easy. Company had this exposed as an API. Reuse.
4. Actually sending a notification. Easy. Company already had an API for contact. Reuse.

So, the size is all in that second slice. We spent some time imagining lots of ways how you could determine who would call. Undefined things sound hard. The product owner proposed something simple, "What if I can make it easy for the initial Minimum Viable Product (MVP) release? What if it was just a text file?"

The pod lead understood where these things come back to bite you, "How often would it need to change? How will we manage the changes to the file?"

"Assume it doesn't change for the MVP scope."

"Is there any PII data in the list?"

"We could use the this internal account number, that's not PII."

"Well, then it's easy."

"How easy. Grape, strawberry, or orange?"

We settled on orangeberry.

Part IX

Back Matter

Chapter 19

Bibliography

Beck, Kent, and Erich Gamma. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

Carroll, Lewis. 1897. *Through the Looking Glass: And What Alice Found There*. Henry Altemus.

Cunningham, Ward. 2005. "Software Development As Craft." *Portland Pattern Repository*. http://wiki.c2.com/?SoftwareDevelopmentAsCraft.

Kim, Gene, Patrick Debois, John Willis, and Jez Humble. 2016. *The DevOps Handbook: How to Create World-Class Speed, Reliability, and Security in Technology Organizations*. IT Revolution Press.

Rasmusson, Jonathan. 2014. *The Agile Samurai: How Agile Masters Deliver Great Software*. Pragmatic Bookshelf.