Risk Management in Software and Hardware Development

Christopher Alexander

chris@aglx.consulting

Abstract

Quality in software and hardware development is increasingly proving a critical, if not decisive factor in delivering value, ensuring customer satisfaction, and increasing business longevity. Yet many technology companies lack a basic vocabulary and framework with which they can communicate, respond to, and mitigate the risks inherent in complex knowledge work. Especially in organizations employing agile practices, risks are introduced through development practices unfamiliar with risk management fundamentals, the results of which impact business in the form of escaped defects or errors (bugs) which slow internal production and affect end-customer utility and value.

Applying lessons learned from years of military service using Operational Risk Management (Office of the Chief of Naval Operations, 2010), we have developed a scalable framework providing a model for companies, teams, and individuals to apply to their risk management systems. Our approach focuses on prevention and detection while also addressing mitigation and response. We provide methods to continuously improve the system across multiple levels, increasing overall robustness, resiliency, and responsiveness.

Consideration is also applied to ways in which Risk Management can be used to improve not only the quality of the work itself, but also the processes and systems in which the work is being done, as well. In this way, Risk Management practices become active contributors to the continuous improvement cycle and can be used to drive change not only within software and hardware development, but across teams, departments, and workplaces to improve engineering and organizational practices.

Biography

Chris Alexander is a former Naval Officer who flew and instructed in the F-14 Tomcat and served abroad as a foreign policy desk officer for several European countries. He is also a web developer, agile coach, Scrum Master, and the co-founder of AGLX Consulting, where he co-developed High-Performance Teaming – a training methodology focused on teaching individuals and teams the social, interactive skills necessary to help them achieve high-performance. He currently works as an agile coach at Qumulo, Inc. in Seattle, Washington.

Copies may not be made or distributed for commercial use

1 Introduction

Risk Management has traditionally existed as a specialized domain of practice within leadership and project management circles. As a silo of concern, many other areas of business consider risk management to be “someone else’s problem,” if considered at all.

Yet in complex knowledge work such as software and hardware development, considerable risk exists at multiple levels throughout the system, and those most capable of detecting and preventing risks from becoming actualized have little to no part to play in risk prevention, analysis, and mitigation. What these individuals, teams, and the companies employing them lack is not a specific skill or ability available to only a select few, but rather a fundamental framework with which they can conceptualize and conduct risk management.

Given a central reference enabled by a common vocabulary and shared mental model (Mathieu and others, 2000), individuals and teams throughout an organization are transformed from passive, risk-unaware bystanders to active participants in risk detection, prevention, and mitigation. Additionally, they are able to assume active, intervening roles in improving the systems and strategies available to the company to manage and eliminate risk.

That last point – that Risk Management can be employed across co-located or distributed teams, departments, and entire enterprise organizations as a catalyst for organizational change – is the crux of this paper. Rather than relegating quality to a silo within a team, division, or department, individuals and teams can use risk management practices to improve the entire system within which they operate, across team and organizational boundaries.

2 Operational Risk Management

Risk Management in Software and Hardware Development is based on the application of Operational Risk Management (ORM) to companies developing software and hardware. Operational Risk Management is the name of the formalized process of risk management matured by the military and derived from routine human practices and habits.

Every day we awaken, review factors of our environment such as weather and traffic, and make decisions about how we plan to commute and what clothing to wear. We unconsciously make trade-offs every day of our lives between the things want to do (our goals), and what will be required to accomplish them, often weighted against the expected benefit or cost involved with our endeavor.

2.1 The Climber

As a practical example, imagine a rock climber. Rock climbing is a relatively risky endeavor. Yet statistically speaking a climber is much more likely to be injured or killed while driving to or from a crag or climbing gym than while actually climbing. When the climber awakens on a weekend morning, they review the weather, the anticipated climbing site, judge their personal desire to be climbing that day, and decide whether or not to face potential accidents, inclement weather, wild animals, etc., just to go rock climbing.

The risk and reward evaluative process designed to reach a decision about whether or not to undertake the activity in consideration is a risk management system. In the case of the climber, they are deciding how much risk they’re willing to accept in pursuing their goal of rock climbing.

Typically, for most of us, the impact and severity of risk in our daily activities is relatively low. In decent weather, when well rested, in a well-maintained car, the climber would accept the risks involved with driving to a climbing site. The training and climbing instruction they’ve undergone, combined with years of experience rock climbing, lowers their risk of being involved in an accident while climbing.

The scenario described above is the essence of risk management in an everyday life. In a business environment, the primary focus of risk management is analyzing and reducing or mitigating business risk. Business risk comes in a variety of shapes and sizes, and can affect individuals, teams, projects, programs, or the security, safety, and integrity of business data, and in many instances can threaten the survival of the business itself.

Despite the generally accepted importance or risk analysis and mitigation in business, many technology companies, departments, and teams view risk as a *business* problem requiring a *business* solution.

With the advent of agile frameworks and methodologies, and particularly in the rise of DevOps, risk can no longer be accepted as a purely business problem, addressed by business people.

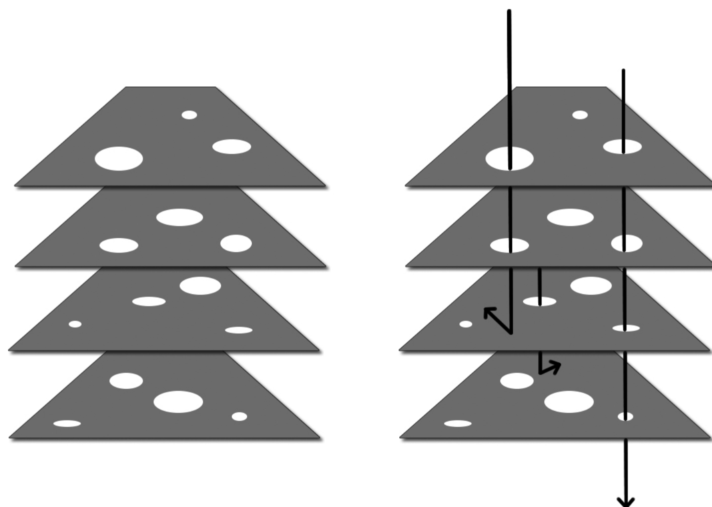
Indeed, in many cases, risks not analyzed or mitigated by software and hardware development teams are indeed passed on to the business, but in an opaque, informal, and latent manner. When individuals and teams fail to appropriately analyze, plan for, and react to risks, those risks endure and can impact business operations. Such impacts are generally unnecessary, as mitigations often could have been applied to these risks well before their effects became realized.

3 The Swiss Cheese Model and Defense-in-Depth

Any risk management system can be seen as a series of layers designed to employ a variety of means to stop errors from progressing further through or escaping from the system. This is known as error trapping. Each layer represents one slice of a larger system. The ability to trap errors with different layers in a system is known across industries as defense-in-depth.

Defense-in-depth reflects the simple idea that rather than employing a single, catch-all solution for trapping all errors and eliminating all risk, a layered approach which employs both latent and active methods for trapping errors at different points throughout the system will be far more effective in both detecting and preventing errors from escaping into production environments.

These layers are often envisioned as “slices” of Swiss cheese (Reason, 1990), with each slice representing a different part of the larger system of protection. As an error progresses through holes in those protective layers, it should eventually be caught by (unable to pass through) one of the layers. It is therefore only when “all the holes line up” that a bug “escapes” into production.



There are two basic classifications of traps (or layers) in any system: latent and active. Latent traps are those characteristics and features of existing systems and tools. Objective-C, for example, contains many built-in features for handing memory allocation, designed to trap many different types of potential memory errors which coding in traditional C occasionally produced. These memory management features exist already within the language and we use them almost literally without ever worrying about it.

Active traps, on the other hand, are active and purposeful measures taken to guard against errors in a proactive way. Running a set of manufacturing tests against a newly assembled product or running acceptance tests for a newly developed feature are examples of active traps. They are purposeful steps designed to find errors.

In daily life, latent traps are things such as the tires on your car or the surface of the road. All-weather, mud and snow tires are risk mitigation steps intended to help guard against the potential for slipping on wet or snowy roads and having an accident. Active traps, conversely, are things such as checking the weather, putting on safety gear, wearing a helmet, or deciding to mitigate the risks present by not driving in bad weather to begin with.

Latent traps in software and hardware development may be things such as the original (legacy) code base, the chassis fabrication line, development language(s), and system architecture and design. Active traps might include development practices like Test-Driven Development or Acceptance-Test-Driven Development, automated test suites, pair programming, code reviews, manufacturing tests, or acceptance verification tests.

3.1 A Fractal, Self-Similar Model

The risk management models which follow can be used by individuals, teams, and organizations, with adjustments and adaptations as necessary, to inform the way the entire company holistically thinks about and manages risk. The benefit of a self-similar, fractal system is that it builds cohesion, understanding, and alignment through the application of a shared mental model. Shared mental models have been instrumental in aligning organizations and teams, and are critical in improving quality and performance in any domain.

At individual levels, and to a lesser extent team levels, this model may be extremely abstract and much less formalized. However understanding its structure, components, and functioning enables the abstraction of key concepts for use in informal, yet still useful ways

3.2 Separation of Concerns in Defense-in-Depth Layers

To better focus on dealing with the most appropriate work at the appropriate time (in reflection of the agile principle of simplicity) in responding to error trapping, bug detection, triage, and risk mitigation, we separate our risk management concerns into the following areas:

Software & hardware development: focus on trapping errors

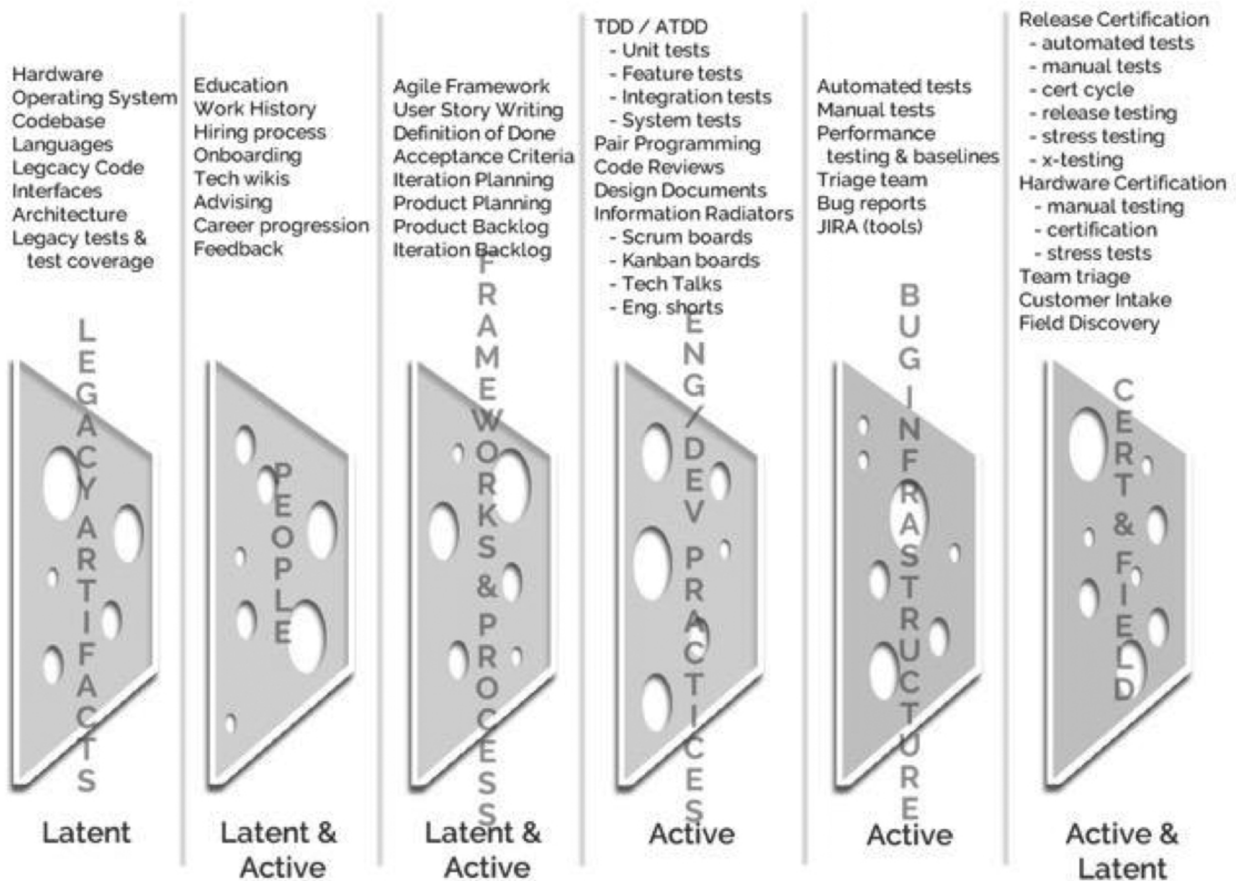
> **Prevention** - the practices, procedures, and techniques we undertake to help ensure we do not release bugs into our codebase or introduce defects during assembly and build.
> **Detection** - the methods available to us as individuals, teams, departments, and a company to find and deal with errors or bugs (which includes reporting and tracking).

Production environments: focus on risk analysis and mitigation

> **Risk Analysis** - the steps required to analyze the severity and impact of an error or bug.
> **Risk Decision-making** - the process for ensuring decisions about risk avoidance, acceptance, or mitigation are made at appropriate levels with full transparency.

Continuous Improvement in every case

> **Improvement** - the process of improving workflows and practices through shared knowledge and experience supported by learning in order to improve development and production practices and further build resiliency and robustness. This step uses causal analysis to help close the metaphorical holes in our Swiss cheese model.



Prevention and Detection deals with those layers, systems, and practices designed to either prevent or find bugs in a system prior to their release into a production environment. Once a bug has been discovered in a production environment, the follow-on processes of *Risk Analysis* and *Risk Decision-making* occur. In either case, *Improvement* completes the loop in the continuous improvement cycle by identifying root causes and lessons learned, which are used to feed improvements back into the system, increasing its robustness and resiliency.

4 Error Causality, Detection & Prevention

Errors occurring during software and hardware development can be classified into two broad categories: skill-based errors, and knowledge-based errors.

4.1 Skill-based errors.

Skill-based errors are those errors which emerge through the application of previously acquired knowledge and experience. They are differentiated from knowledge-based errors in that they arise not from a lack of knowing what to do, but instead from either misapplication of, or an overall failure to apply, what is known.

There are two types of skill-based errors: errors of commission, and errors of omission.

Errors of commission are the mis-application of a previously learned behavior (knowledge or experience). Returning to the climbing metaphor which opened this paper, if the climber tied the rope into their harness using an incorrect knot, they would be committing an error of commission. They know they need a knot,

know which knot to use, and know how to tie the correct knot. They simply did not accomplish tying the knot correctly. In software development, one example of an error of commission might be an engineer providing the wrong variable to a function call, as in:

```
var x = 1;      // variable to call
var y = false;     // variable not to call

public function callVariable(x) {
   return x;
}

// whole bunch of code… code… code…
callVariable(y); // should have provided "x" but gave "y" instead
```

Errors of omission are the total failure to apply previously learned behaviors (knowledge and/or experience). Returning to our climbing metaphor, an experienced climber *not* tying the rope into their harness before beginning a climb is an example of an error of omission (and yes, this actually does happen). In software development, an example of an error of omission would be an engineer forgetting to provide a variable to a function call (or possibly forgetting to add the function call at all), as in:

```
var x = 1;         // variable to call
var y = false;        // variable not to call

public function callVariable(x) {
   return x;
}

// whole bunch of code… code… code…
callVariable();   // should have provided "x" but left empty
```

4.2 Knowledge-based errors

Knowledge-based errors, in contrast to skill-based errors, arise from the failure to know the correct behavior to apply (if any). An example of a knowledge-based error would be a developer checking in code but not having written or run any tests for that code. If the developer is new and has never been indoctrinated on the requirements for code check-in as including unit, integration, and system tests, this is an error caused by a lack of knowledge (as opposed to omission, where the developer had been informed of the need to build and run the tests, but failed to do so).

4.3 Preventing and Detecting Errors

Prevention comprises the layers, systems, tools, and processes employed to trap bugs and stop them from escaping development environments and entering into build, staging, testing, certification, or eventually production environments. In envisioning our Swiss cheese model, we need to understand that the layers include both latent and active types of error traps, and are designed to mitigate against certain types of errors.

The following are examples of some of the methods, tools, and processes available to aid in *preventing* bugs.

To mitigate against Skill-based errors in bug prevention:

Code base and architecture [latent]
Automated test coverage [active]
Manual test coverage [active]
Unit, feature, integration, system, and story tests [active]
TDD / ATDD practices [active]

Code reviews [active]
Pair Programming [active]
Performance testing [active]
Software development framework / methodology (ie, Scrum, Kanban, DevOps, etc.) [latent]

To mitigate against Knowledge-based errors in bug prevention:

Employee education & background [latent]
Recruiting and hiring practices [active]
Employee Onboarding [active]
Performance feedback & professional development [active]
Design documents [active]
Definition of Done [active]
User Story Acceptance Criteria [active]
Code reviews [active]
Pair Programming [active]
Information Radiators [latent]

Detection is the term for the ways in which we find bugs, hopefully in the development environment but this phase also includes continuous integration, testing, and/or certification stages as well. The primary focus of detection methods is to ensure no bugs escape into production. As such, the entire software integration, deployment, and testing (or certification) system itself may be considered one, large, active layer of error trapping. In fact, in many enterprise companies, the certification team is literally the *last line of defense*.

The following are examples of some of the methods, tools, and processes available to aid in *detecting* bugs:

To mitigate against Skill-based errors in detecting bugs:

Automated test coverage [active]
Manual test coverage [active]
Unit, feature, integration, system, and story tests [active]
TDD / ATDD practices [active]
Release integration / certification testing [active]
Performance testing [active]
User Story Acceptance Criteria [active]
User Story “Done” Criteria [active]
Bug tracking software [active]
Triage reports [active]

To mitigate against Knowledge-based errors in detecting bugs:

Employee education & background [latent]
Hiring discipline and practices [active]
Employee Onboarding [active]
Continuous Improvement activities [active]

5 Risk Analysis

Should existing preventative measures fail to stop a bug from escaping our development environment and entering into a production system, an analysis of the level of risk needs to be explicitly completed. This analysis is almost always accomplished, but in an implicit way. The analysis of the level of risk derives from two areas (note that the severity and probability metrics introduced below are in no way meant to be proscriptive, but rather to serve as examples from which to build or deviate).

5.1 Risk Severity

Risk Severity is the degree of impact the bug can be expected to have to the data, operations, or functionality of affected entities (the company, vendors, clients, customers, etc.).

Blocker	A bug the effects of which are so bad, or a feature that is so important, that we would not ship the next release or additional units until it is remedied. Could also signify a bug that is currently blocking customer workflows, or one that is blocking our own internal submits/builds, etc.
Critical	A bug that needs to be resolved ASAP, but for which we would not hold a release or execute a special release. Depending on its probability, a team might need to be interrupted during their work to remedy the error.
Major	Best judgment should be used to determine the priority of this issue against other work in a Sprint. It needs to be resolved, but is not a "drop everything and fix this" issue. If a bug sits in major for too long (more than a Sprint or a few weeks), it should be upgraded to critical or downgraded to Minor.
Minor	A bug that is known, but that has been *explicitly* deprioritized. Often, these bugs should be either addressed as part of a Sprint or downgraded to Trivial.
Trivial	We should consider closing this. At best these should be put into a backlog for tracking and remedy as work capacity becomes available.

5.2 Risk Probability

Risk Probability is the anticipated percentage of all customers potentially affected by the bug who will likely experience it (for example: always, only if they have a power outage, most likely not, etc.).

Definite	100% *- issue will occur in every case*
Probable	60-99% *- issue will occur in most cases*
Possible	30-60% *- coin-flip; issue may or may not occur*
Unlikely	2-30% *- issue will occur in less than 50% of cases*
Won't	1% *- occurrence of the issue will be exceptionally rare*

5.3 Risk Assessment Code

Given Risk Severity and Probability, the risk can be assessed according to a risk assessment matrix (example below), and assigned the corollary Risk Assessment Code (RAC).

The Risk Assessment Codes are a significant factor in Risk decision-making. Based on your organization's culture and sensitivity to risk, which may need to account for things such as regulatory environment, compliance and reporting, sensitivity of customer data, reputation, etc., your decision-making on risk may need to be significantly altered from the examples here.

Risk Assessment Matrix		Probability				
		Definite	*Probable*	*Possible*	*Unlikely*	*Won't*
Severity	*Blocker*	**1**	**1**	**1**	**2**	**3**
	Critical	**1**	**1**	**2**	**2**	**3**
	Major	**2**	**2**	**2**	**3**	**4**
	Minor	**3**	**3**	**3**	**4**	**5**
	Trivial	**3**	**4**	**4**	**5**	**5**
Risk Assessment Codes **1** - Strategic **2** - Significant **3** - Moderate **4** - Low **5** - Negligible						

The example here is intended to inform discussion and act as a guide, not to prescribe levels or constrain a company's need to adopt and adapt scales more responsive to its needs.

Strategic - the risk to the business is significant enough that its realization could threaten business operations, basic functioning, regulatory / compliance requirements, and/or professional reputation to the point that the basic survival of the business could be in jeopardy.
Significant - the risk poses considerable, but not life-threatening, challenges for the business. If left unchecked, these risks may elevate to strategic levels.
Moderate - the risk to business operations, continuity, and/or reputation is significant enough to warrant consideration against other business priorities and issues, but not significant enough to trigger higher responses.
Low - the risk to the business is not significant enough to warrant special consideration of the risk against other priorities. Issues should be dealt with in routine, predictable, and business-as-usual ways.
Negligible - the risk to the business is not significant enough to warrant further consideration.

5.4 The Risk Decision

The risk decision is the point at which a decision is made regarding actions to take or not take based on the risk. Typically, risk decisions take the form of:

Accept - accept the risk as it is and do not mitigate or take additional steps.
Delay - for less critical issues or dependencies, a decision about whether to accept or mitigate a risk may be delayed until additional information, research, or steps are completed.
Mitigate - establish a mitigation strategy and deal with the risk.

For risk mitigation, feasible courses of action (COAs) should be developed to assist in formulating the mitigation plan. These potential actions comprise the mitigation and or reaction plan. Specifically, given risk severity, probability, and the resulting RAC, the courses of action are the possible mitigate solutions for the risk. Examples include:

Pre-release

Apply software fix / patch
Code refactor
Code rewrite
Release without the code integrated (re-build)
Hold the release and await code fix
Cancel the release

In production

Add to normal backlog and prioritize with normal workflow
Pull / create a team to triage and fix
Swarm multiple teams on fix
Pull back / recall release
Release an additional fix as a micro-release

Mitigation strategies at the corporate level may also need to include deliverables from and outputs for multiple departments, such as marketing, PR, operations, IT, and legal. For serious issues a marketing and PR campaign may be required to mitigate impacts from a production error on the company's reputation. For legal and compliance related issues, self-reporting is often a swift path to avoiding investigations, penalties, and fines.

For all risk decisions, those decisions should be shared with stakeholders, recorded, and for those which remain active, tracked. There are many methods available for logging and tracking risk decisions, from simple spreadsheets to entire software platforms expressly designed to track and monitor risk status and decisions.

Decisions to delay risk mitigations are the most important to track, as they typically require future action, and at the speed most business move today, another risk exists in potentially losing track of risk-delayed decisions. Therefore a Risk Log or Review should be used to routinely review the status of pending risk decisions and reevaluate them, not to mention using simple calendar reminders or other "low-tech" methods to keep the delayed decision visible.

Risk changes constantly, and risks may significantly change in severity and probability as quickly as overnight. In reviewing risk decisions regularly, individuals, teams, and leaders are able to simultaneously ensure both that emerging risks are mitigated and that effort is not wasted unnecessarily.

6 Process Improvement

Once a bug has been discovered and risk analysis / decision-making has been completed, a retrospective-style analysis on the circumstances surrounding the development or engineering practices which failed to effectively trap the bug completes the cycle.

The purpose of the retrospective or debrief is not to assign blame or find fault, but rather to understand the cause of the failure to trap the bug, inspect the layers of the system, and determine if any additional tools, methods, procedures, or process changes could effectively improve collective practice and help to prevent future bugs.

6.1 Methodology

Review sequence of events that led to the anomaly / bug.
Determine **root cause**.
Map the root cause against the layer model.
Decide if there are remediation efforts or improvements which would be effective in supporting or restructuring the system to increase its effectiveness at error trapping.

Copies may not be made or distributed for commercial use

Implement any changes identified, sharing them publicly to ensure everyone understands the changes and the reasoning behind them.
Monitor the changes, adjusting as necessary.

6.1.1 *Review* sequence of events

With appropriate representatives from engineering teams, hardware, operations, customer support, etc., review the discovery path which found the bug. The point is to understand the processes used, which layers or guards worked, and which ones let the error pass through.

6.1.2 Determine *root cause* and analyze the optimum layers for improvement

What caused the bug? There are many enablers and contributing factors, but typically only one or two root causes. The root cause is one or a possible combination of Organization, Communication, Knowledge, Experience, Discipline, Teamwork, or Leadership.

Organization - typically latent, organizational root causes include things like existing processes, tools, practices, habits, customs, culture, etc. These are factors of the company or organization as a whole.
Communication - a failure to convey necessary, important, or vital information to an individual or team who required it for the successful accomplishment of their work.
Knowledge - an individual, team, or organization did not possess the knowledge necessary to succeed. *This is the root cause for knowledge-based errors.*
Experience - an individual, team, or organization did not possess the experience necessary to accomplish a task (as opposed to the knowledge about what to do). *Experience is often a root cause in skill-based errors of omission*.
Discipline - an individual, team, or organization did not possess the discipline necessary to apply their knowledge and experience to solving a problem. *Discipline is often a root cause in skill-based errors of commission*.
Teamwork - individuals, possibly at multiple levels, failed to work together as a team. Additional root causes may be knowledge, experience, communication, or discipline.
Leadership - less often seen at smaller organizations, a Leadership failure is typically a root cause when a leader and/or manager has not effectively communicated expectations or empowered team execution regarding those expectations. Leadership has, either directly or indirectly, hindered the individuals', teams', or organization's ability to deliver effectively.

6.1.3 *Map the root cause* to the layer(s) which should have trapped the error.

Given the root cause analysis, determine where in the system (which layer or layers) the bug / error should have or could have been trapped. The best location to identify is the one which most closely corresponds to the root cause of the bug and which occurs earliest in the system or process flow.

The earlier an error can be prevented or caught, the less costly it is in terms of both time (to find, fix, and eliminate the bug) and effort (a bug in production requires more effort from more people than a developer discovering a bug while checking their own unit test).

While we should seek to apply fixes at the locations best suited for them, the earliest point at which a bug could have been caught and prevented will often be *the optimum place* to improve the system.

For example, if a bug was traced back to a team's discipline in writing and using tests (root cause: *discipline* and *experience*), then it would map to layers dealing with testing practices (TDD/ATDD), pair programming, acceptance criteria, User Story writing, definition of "Done," etc. Those layers where the team can most readily apply improvements and which will trap the error sooner, should be the focus for improvement efforts.

6.1.4 *Decide* on improvements to increase system effectiveness.

Based on the knowledge gained through analyzing and mapping the root cause, decisions are made on how to improve the effectiveness of the system at the layers identified. Using the testing example above, a team could decide that they need to adjust their definition of Done to include listing which tests a story has been tested against and their pass/fail conditions.

6.1.5 *Implement* the changes identified.

Once changes have been identified, those changes must be clearly communicated at appropriate levels so that everyone involved understands their individual and collective responsibilities in implementing those changes. This is also a good place to share lessons learned about the process and the improvements discovered so that everyone may benefit from the shared knowledge.

6.1.6 *Monitor* the changes and adjust as necessary.

In an agile environment, where teams are able to continuously learn and adapt, as new insights or options become available to either reinforce or replace existing aspects of a defense-in-depth system, the effectiveness of methods and layers should be evaluated against new ideas and improvements to continue building the optimum configuration for error trapping. The goal is to build and maintain a robust and resilient system.

7 Application of Risk Management in Planning

When undertaking backlog grooming, Sprint planning, or any type of work breakdown activity, we tend to estimate work items / User Stories in terms of the size and amount of work required to complete them, often including consideration of the degree of complexity (unknowns) inherent in the work. Some items are low in complexity and tend to be estimated lower, while others are higher in complexity and tend to be estimated somewhat larger.

Once the backlog of work has been broken down and estimated according to effort and complexity, the work is prioritized according to business value (which is typically, but not always, the value of the work to potential or actual customers). In many instances, this is where estimation and prioritization ceases and the most valuable (highest priority) items are pulled to kick-off work.

This is a mistake in most environments, however. The backlog needs to also be analyzed for risk, risks need to be estimated, and the backlog needs to be prioritized again to account for both risk and value.

For example, the physical assembly and testing of a new hardware platform may be a relatively low-value story or work item, however the risk of not being able to deliver the platform in a sufficient amount of time due to incomplete testing, coupled with the risk that the build and shipment of the platform’s physical chassis itself requires three months, may mean that this work item becomes the highest priority item, today.

Without adjusting our estimation and prioritization for risk, we may miss important dependencies, durations, challenges, opportunities, etc. which could be easily mitigated or capitalized upon if recognized early enough, helping to avert a potential disaster later in our product development or release.

7.1 Adjusting and prioritizing for Risk

With the product or work backlog now estimated and prioritized according to value, the team looks again at each item and evaluates the risk(s) inherent to it. Complexity is a risk, for example, present in many larger software development User Stories and relates directly to the amount of unknown (unknowable) work involved in completing the story.

Other types of risk might be dependencies (either internal or external), critical components, availability of hardware/infrastructure/etc., outcomes of experiments, etc. Such risks need be evaluated for their degree of impact and likelihood of occurrence, just as business risk analysis was described previously.

Again, as an example, an external dependency (vendor) on a critical component (chassis) which will entirely block future work must be mitigated by prioritizing the work item appropriately in order to ensure lead time is met. If we are at the start of an estimated four month project and we have a two month wait time for a vendor delivery of a critical component, we need to prioritize this work item now.

Having reviewed backlog or work items, we are now aware of risks inherent in the backlog, and can reprioritize it again, adjusting the balance of work to account for *both* value *and* risk. This new prioritization also has the added benefit of potentially revealing risks which we cannot mitigate or overcome through prioritization of work alone - we may need help from sources external to the team or we may need to add additional risk-mitigating work to the backlog / work inventory.

These discussions regarding how to think about and prioritize risk *may* seem pedantic and overly process-focused, even intuitive, but if such is the case then you are among the blessed minority of teams who have a good methodology for risk management. The depressing reality is that the overwhelming majority of teams have no framework, process, tool, or skillset to help them analyze and manage risk in any explicit or useful way.

Through a mindful analysis of potential risks and a purposeful strategy to manage them, teams can improve their chances of successful outcomes as they build and deliver products and features, in any domain.

8 Summary

Individuals, teams, and companies engaged in software and hardware development have typically regarded risk in indirect, assumptive ways. In increasingly competitive environments in which quality and development time can prove to be critical market differentiators, companies can no longer afford to view risk as a project- or management-level factor. Risk management needs to be understood and employed at every level within an organization to achieve effectiveness.

Risk Management is an organic activity which we undertake every day of our lives in some form. As such, leveraging our innate ability to analyze risk and make risk-based decisions, our strength in building robust systems lies in extricating our intrinsic risk management knowledge and formalizing it into a purposeful and conscious system.

In this paper we have provided a framework for individuals, teams, and organizations to conceptualize, analyze, and evaluate risk, based on a common vocabulary and using shared mental models. These structures (common vocabulary and shared mental models) have been proven to be instrumental in ensuring alignment and effectiveness in teams and organizations (DeChurch and Mesmer-Magnus, 2010; Mathieu and others, 2000).

Understanding how a company's processes and practices form a Defense-in-Depth system for containing errors enables the system to be purposefully improved over time based on lessons learned from root cause analysis of defect or error occurrences. The improvement of multiple layers of the system creates a more robust and resilient system, in turn increasing the capacity for higher levels of quality and productivity.

Every organization operates some form of this Defense-in-Depth system, which can be conceptualized as layers of Swiss cheese. Understanding that the majority of preventable errors fall into one of several categories enables teams and individuals to root cause those errors and apply improvements to the system, optimally in multiple places, thus increasing the organization's robustness and resilience.

Copies may not be made or distributed for commercial use

In cases of escaped defects, risk decisions must occasionally be taken and should be based on a clear understanding of the risk's severity and probability of occurrence. In these instances, a risk decision log should be created, updated, and reviewed.

Individuals, teams, and companies should not seek to apply this framework in a prescriptive manner, but rather adapt them to fit the unique circumstances and requirements of their situation.

Copies may not be made or distributed for commercial use

References

Office of the Chief of Naval Operations. “OPNAV Instruction 3500.39C, Operational Risk Management, (dated 02 Jul 2010).” United States Navy. http://www.public.navy.mil/airfor/nalo/Documents/SAFETY/OPNAVINST%203500.39C%20OPERATIONAL%20RISK%20MANEGEMENT.pdf (accessed July 12, 2016).

Reason, James. 1990. "The Contribution of Latent Human Failures to the Breakdown of Complex Systems." *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences 327 (1241)*: 475–84.

DeChurch, Leslie A. and Jessica R. Mesmer-Magnus. 2010. “Measuring Shared Team Mental Models: A Meta-Analysis.” *Group Dynamics: Theory, Research, and Practice 14 (1):* 1-14.

Mathieu, John E and others. 2000. “The Influence of Shared Mental Models on Team Process and Performance.” *Journal of Applied Psychology 85 (2):* 273-283.

Copies may not be made or distributed for commercial use