Quick Summary

1. There are Lots of Ways of Running Software Projects

There are lots of different ways to look at a project in-flight. For example, metrics such as "number of open tickets", "story points", "code coverage" or "release cadence" give us a numerical feel for how things are going and what needs to happen next. We also judge the health of projects by the practices used on them, such as Continuous Integration, Unit Testing or Pair Programming.

Software methodologies, then, are collections of tools and practices: "Agile", "Waterfall", "Lean" or "Phased Delivery" all prescribe different approaches to running a project, and are opinionated about the way they think projects should be done and the tools that should be used.

None of these is necessarily more "right" than another- they are suitable on different projects at different times.

A key question then is: **how do we select the right tools for the job?**

2. We can Look at Projects in Terms of Risks

One way to examine the project in-flight is by looking at the risks it faces.

Commonly, tools such as RAID logs[1] and RAG status[2] reporting are used. These techniques should be familiar to project managers and developers everywhere.

However, the Risk-First view is that we can go much further: that each item of work being done on the project is to manage a particular risk. Risk isn't something that just appears in a report, it actually drives *everything we do*.

For example:

[1] `https://www.projectmanager.com/blog/raid-log-use-one`

[2] `https://pmtips.net/blog-new/what-does-rag-status-mean`

- A story about improving the user login screen can be seen as reducing *the risk of users not signing up*.
- A task about improving the health indicators could be seen as mitigating *the risk of the application failing and no-one reacting to it*.
- Even a task as basic as implementing a new function in the application is mitigating *the risk that users are dissatisfied and go elsewhere*.

One assertion of Risk-First is that **every action you take on a project is to manage a risk.**

3. We Can Break Down Risks on a Project Methodically

Although risk is usually complicated and messy, other industries have found value in breaking down the types of risks that affect them and addressing them individually.

For example:

- In manufacturing, *tolerances* allow for calculating the likelihood of defects in production.
- In finance, projects and teams are structured around monitoring risks like *credit risk*, *market risk* and *liquidity risk*.
- *Insurance* is founded on identifying particular risks and providing financial safety-nets for when they occur, such as death, injury, accident and so on.

Software risks are difficult to quantify, and mostly, the effort involved in doing so *exactly* would outweigh the benefit. Nevertheless, there is value in spending time building *classifications of risk for software*. That's what Risk-First does: it describes a set of *risk patterns* we see every day on software projects.

With this in place, we can:

- Talk about the types of risks we face on our projects, using an appropriate language.
- Anticipate Hidden Risks that we hadn't considered before.
- Weigh the risks against each other, and decide which order to tackle them.

4. We can Analyse Tools and Techniques in Terms of how they Manage Risk

If we accept the assertion above that *all* the actions we take on a project are about mitigating risks, then it stands to reason that the tools and techniques available to us on a project are there for mitigating different types of risks.

For example:

- If we do a Code Review, we are partly trying to minimise the risks of bugs slipping through into production, and also manage the Key-Man Risk of knowledge not being widely-enough shared.
- If we write Unit Tests, we're addressing the risk of bugs going to production, but we're also mitigating against the risk of *regression*, and future changes breaking our existing functionality.
- If we enter into a contract with a supplier, we are mitigating the risk of the supplier vanishing and leaving us exposed. With the contract in place, we have legal recourse against this risk.

From the above examples, it's clear that **different tools are appropriate for managing different types of risks.**

5. Different Methodologies are for Different Risk Profiles

In the same way that our tools and techniques are appropriate to dealing with different risks, the same is true of the methodologies we use on our projects. We can use a Risk-First approach to examine the different methodologies, and see which risks they address.

For example:

- **Agile** methodologies prioritise the risk that requirements capture is complicated, error-prone and that requirements change easily.
- **Waterfall** takes the view that development effort is an expensive risk, and that we should build plans up-front to avoid re-work.
- **Lean** takes the view that risk lies in incomplete work and wasted work, and aims to minimise that.

Although many developers have a methodology-of-choice, the argument here is that there are tradeoffs with all of these choices.

"Methodologies are like *bicycles*, rather than *religions*. Rather than simply *believing*, we can take them apart and see how they work. "

6. We can Drive Development With a Risk-First Perspective

We have described a model of risk within software projects, looking something like this:

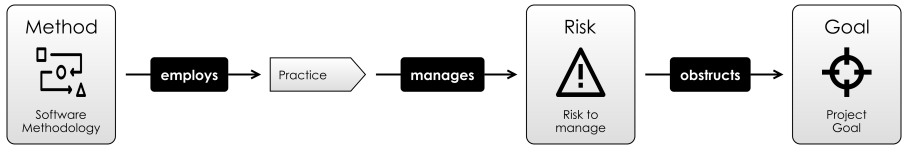

Figure 1: Methodologies, Risks, Practices

How do we take this further?

One idea explored is the *Risk Landscape*: Although the software team can't remove risk from their project, they can take actions that move them to a place in the Risk Landscape where the risks on the project are more favourable than where they started.

From there, we examine basic risk archetypes you will encounter on the software project, to build up a Taxonomy of Software Risk, and look at which specific tools you can use to mitigate each kind of risk.

Then, we look at different software practices, and how they manage various risks. Beyond this we examine the question: *how can a Risk-First approach inform the use of this practice?*

For example:

- If we are introducing a **Sign-Off** in our process, we have to balance the risks it *mitigates* (coordination of effort, quality control, information sharing) with the risks it *introduces* (delays and process bottlenecks).
- If we build in **Redundancy**, this mitigates the risk of a *single point of failure*, but introduces risks around *synchronizing data* and *communication* between the systems.

- If we introduce **Process**, this may make it easier to *coordinate as a team* and *measure performance* but may lead to bureaucracy, focusing on the wrong goals or over-rigid interfaces to those processes.

Risk-First aims to provide a framework in which we can *analyse these actions* and weigh up *accepting* versus *mitigating* risks.

Still interested? Then dive into reading the introduction.

Part I

Introduction

Part II

The Risk Landscape

CHAPTER 1

Operational Risk

> "The risk of loss resulting from inadequate or failed internal processes, people and systems or from external events."
>
> —Operational Risk, *Wikipedia*[1]

In this chapter we're going to start considering the realities of running software systems in the real world.

Here, we're going to set the scene by looking at what constitutes an Operational Risk, and then look at the related discipline of Operations Management. Following this background, we'll apply the Risk-First model and dive into the various mitigations for Operational Risk.

1.1 Operational Risks

When building software, it's tempting to take a very narrow view of the dependencies of a system, but Operational Risks are often caused by dependencies we *don't* consider - i.e. the **Operational Context** within which the system is operating. Here are some examples:

- **Staff Risks:**
  - Freak weather conditions affecting ability of staff to get to work, interrupting the development and support teams.
  - Reputational damage caused when staff are rude to the customers.
- **Reliability Risks:**

[1] https://en.wikipedia.org/wiki/Operational_risk#Definition

  - A data-centre going off-line, causing your customers to lose access.
  - A power cut causing backups to fail.
  - Not having enough desks for everyone to sit at.

- **Process Risks:**
  - Regulatory change, which means you have to adapt your business model.
  - Insufficient controls which means you don't notice when some transactions are failing, leaving you out-of-pocket.
  - Data loss because of bugs introduced during an untested release.

- **Software Dependency Risk:**
  - Hackers exploit weaknesses in a piece of 3rd party software, bringing your service down.

- **Agency Risk:**
  - Suppliers deciding to stop supplying you with something you need.
  - Workers going on strike.
  - Employees trying to steal from the company (bad actors).
  - Other crime, such as hackers stealing data.

.. basically, a long laundry-list of everything that can go wrong due to operating in "The Real World". Although we've spent a lot of time looking at the varieties of Dependency Risk on a software project, with Operational Risk we have to consider that these dependencies will fail in any number of unusual ways, and we can't be ready for all of them. Nevertheless, preparing for this comes under the umbrella of Operations Management.

1.2 Operations Management

If we are designing a software system to "live" in the real world, we have to be mindful of the Operational Context we're working in, and craft our software and processes accordingly. This view of the "wider" system is the discipline of Operations Management.

Figure 1.1 is a Risk-First interpretation of Slack *et al*'s model of Operations Management[2]. This model breaks down some of the key abstractions of the discipline: a **Transform Process** (the **Operation** itself) is embedded in the wider **Operational Context**, which supplies it with three key dependencies:

[2] `http://amzn.eu/d/b6ZjuMu`

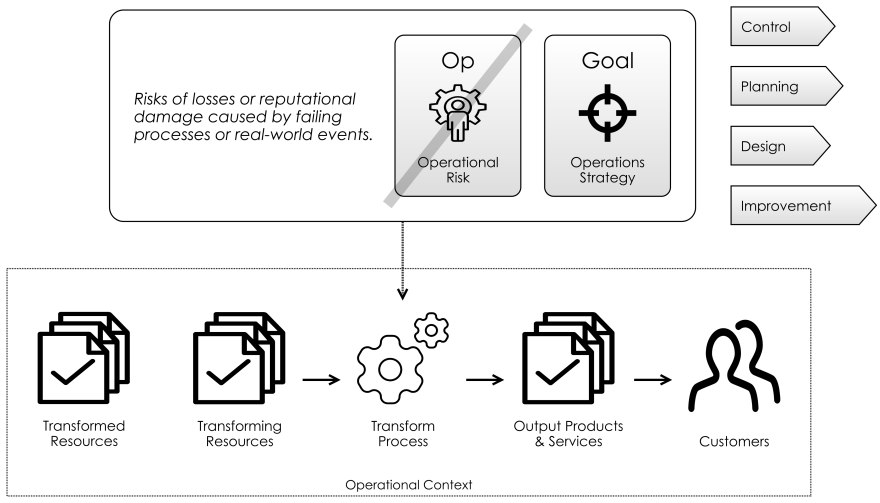

***Figure 1.1: A Risk-First Model of Operations Management, inspired by the work of Slack* et al.**

- **Resources**: Whether *transformed* resources (like electricity or information, say) or *transforming* resources (like staff or equipment).
- **Customers**: Which supply it with money in return for goods and services.
- **Operational Strategy**: The goals and objectives of the operation, informed by the reality of the environment it operates in.

We have looked at processes like the **Transform Process** in the chapter on Process Risk. The healthy functioning of this process is the domain of Operations Management, and (as per Slack *et al.*) this involves the following types of actions:

- **Control**: Ensuring that the Operation is working according to it's targets. This includes day-to-day quality control and monitoring of the Transform Process.
- **Planning**: This covers aspects such as capacity planning, forecasting and project planning. This is about making sure the transform process has targets to meet and the resources to meet them.
- **Design**: Ensuring that the design of the product and the transform process itself fulfils an **Operational Strategy**.

- **Improvement**: Improving the operation in response to changes in the **Environment** and the **Operational Strategy**, detecting failure and recovering from it.

Let's look at each of these actions in turn.

1.3 Control

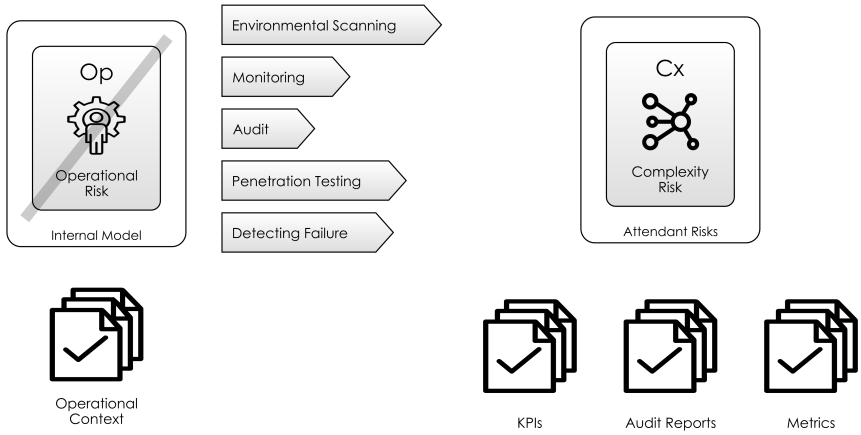

Figure 1.2: Control, Monitoring And Detection

Since humans and machines have different areas of expertise, and because Operational Risks are often novel, it's often not optimal to try and automate everything. A good operation will consist of a mix of human and machine actors, each playing to their strengths (see the table below).

Humans Are...	Machines Are...
Good at novel situations	Good at repetitive situations
Good at adaptation	Good at consistency
Expensive at scale	Cheap at scale
Reacting and Anticipating	Recording

The aim is to build a human-machine operational system that is *Homeostatic*[3]. This is the property of living things to try and maintain an equilibrium

[3] https://en.wikipedia.org/wiki/Homeostasis

(for example, body temperature or blood glucose levels), but also applies to systems at any scale. The key to homeostasis is to build systems with feedback loops, even though this leads to more complex systems overall. Figure 1.2 shows some of the actions involved in these kind of feedback loops.

As we saw in Map and Territory Risk, it's very easy to fool yourself, especially around Key Performance Indicators (KPIs)[4] and metrics. Large organisations have Audit[5] functions precisely to guard against their own internal failing processes and Agency Risk. Audits could be around software tools, processes, practices, quality and so on. Practices such as Continuous Improvement[6] and Total Quality Management[7] also figure here.

The Operational Context

There are plenty of Hidden Risks within the environment the operation exists within, and these change all the time in response to economic or political change. In order to manage a risk, you have to uncover it, so part of Operations Management is to look for trouble:

- **Environmental Scanning** is all about trying to determine which changes in the environment are going to impact your operation. Here, we are trying to determine the level of Dependency Risk we face for external dependencies, such as *suppliers*, *customers* and *markets*. Tools like PEST[8] are relevant here, as is
- **Penetration Testing[9]** is looking for security weaknesses within the operation. See OWASP[10] for examples.
- **Vulnerability Management[11]** is keeping up-to-date with vulnerabilities in Software Dependencies.

1.4 Planning

In order to *control* an operation, we need targets and plans to *control against*. For a system to run well, it needs to carefully manage unreliable dependencies, and ensure their safety and availability. In the example of the humans,

[4] https://en.wikipedia.org/wiki/Performance_indicator

[5] https://en.wikipedia.org/wiki/Audit

[6] https://en.wikipedia.org/wiki/Continual_improvement_process

[7] https://en.wikipedia.org/wiki/Total_quality_management

[8] https://en.wikipedia.org/wiki/PEST_analysis

[9] https://en.wikipedia.org/wiki/Penetration_test

[10] https://en.wikipedia.org/wiki/OWASP

[11] https://en.wikipedia.org/wiki/Vulnerability_management

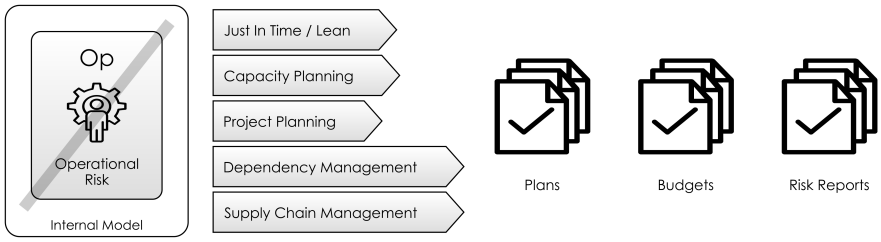

Figure 1.3: Forecasting and Planning Actions

say, it's the difference between Hunter-Gathering[12] (picking up food where we find it) and Agriculture[13] (controlling the environment and the resources to grown crops).

As Figure 1.3 shows, we can bring Planning to bear on dependency management, and this usually falls to the more human end of the operation.

1.5 Design

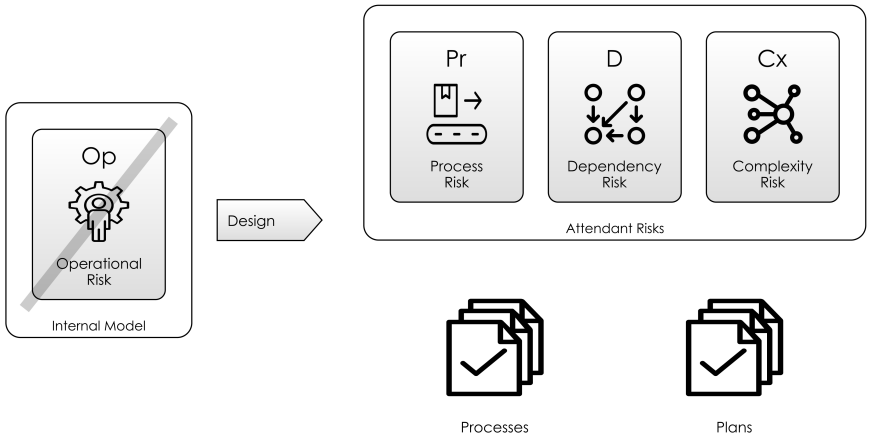

Figure 1.4: Design and Change Activities

[12] https://en.wikipedia.org/wiki/Hunter-gatherer

[13] https://en.wikipedia.org/wiki/Agriculture

Since our operation exists in a world of risks like Red Queen Risk and Feature Drift Risk, we would expect that the output of our Planning actions would result in changes to our operation.

While *planning* is a day-to-day operational feedback loop, *design* is a longer feedback loop changing not just the parameters of the operation, but the operation itself.

You might think that for an IT operation, tasks like Design belong within the Development function within an organisation. Often, this is the case. However separating Development from Operation implies Boundary Risk between these two functions. For example, the developers might employ different tools, equipment and processes to the operations team, resulting in a mismatch when software is delivered.

In recent years, the "DevOps" movement has brought this Boundary Risk into sharper focus. This specifically means:

- Using code to automate previously manual Operations functions, like monitoring and releasing.
- Involving Operations in the planning and design, so that the delivered software is optimised for the environment it runs in.

DevOps heavily borrows from the Agile movement, both of which will be covered in more detail in Part 4.

1.6 Improvement

Once exposed to the real world, no system is perfect: we will want to improve it over time. However, conversely, Operational Risk includes an element of Trust & Belief Risk: our *reputation* and the good will of our customers. This gives us pause: we don't want to destroy good will created for our software, this is very hard to rebuild.

So there is a tension between "you only get one chance to make a first impression" and "gilding the lily" (perfectionism). In the past I've seen this stated as:

> "Pressure to ship vs pressure to improve"

A Risk-First re-framing of this might be the balance between:

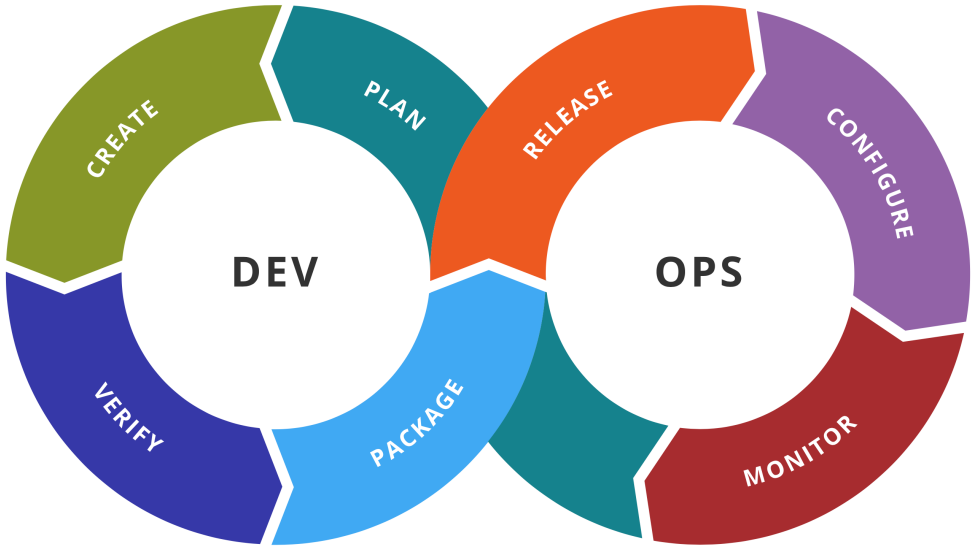

Figure 1.5: DevOps Activities: Development and Operations activities overlap one-another (Credit: Kharnagy, Wikipedia)

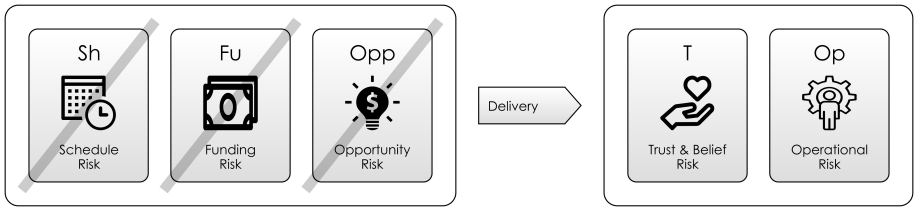

Figure 1.6: Balance of Risks from Delivering Software

- The perceived Trust & Belief Risk, Feature Risk and Operational Risk of going to production (pressure to improve).
- The perceived Scarcity Risks (such as funding, time available, etc) of staying in development (pressure to ship).

The "should we ship?" decision is therefore a complex one. In Meeting Reality, we discussed that it's better to do this "sooner, more frequently, in smaller chunks and with feedback". We can meet Operational Risk *on our own terms* by doing so:

Meet Reality...	Techniques
Sooner	Quality Control Processes, Limited Early-Access Programs, Beta Programs, Soft Launches, Business Continuity Testing
More Frequently	Continuous Delivery, Sprints
In Smaller Chunks	Modular Releases, Microservices, Feature Toggles, Trial Populations
With Feedback	User Communities, Support Groups, Monitoring, Logging, Analytics

1.7 End Of The Road

In a way, actions like **Design** and **Improvement** bring us right back to where we started from: identifying Dependency Risks, Feature Risks and Complexity Risks that hinders our operation, and mitigating them through actions like *software development*.

Our safari of risk is finally complete, it's time to look back and what we've seen in Staging and Classifying.

Part III

Tools & Practices