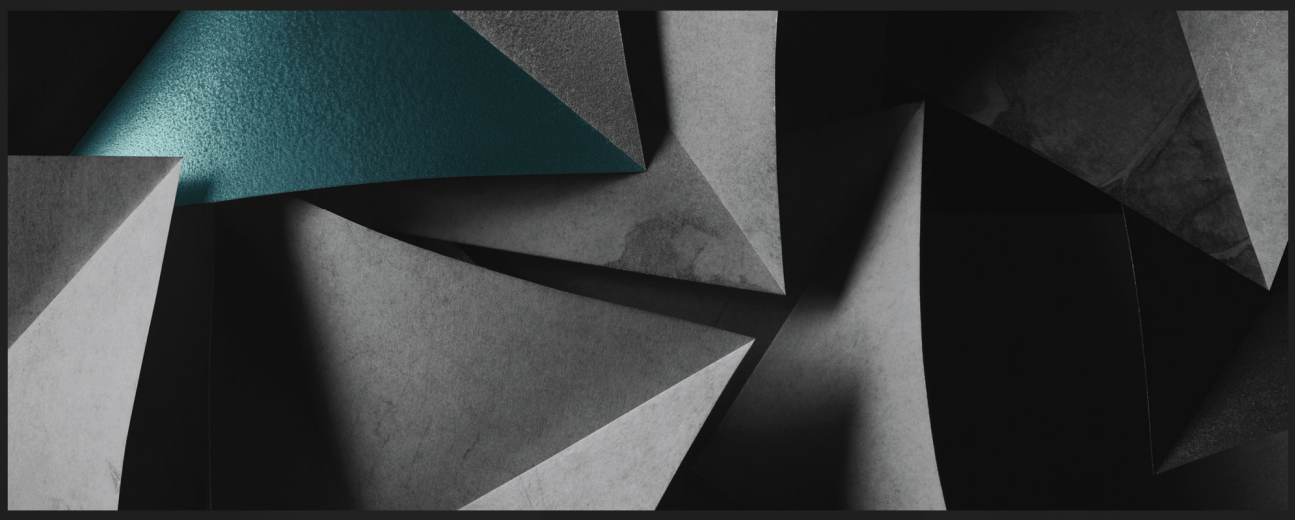


1ST EDITION

Technical Program Manager's Handbook

Empowering managers to efficiently manage technical projects and build a successful career path

JOSHUA ALAN TETER

Foreword by Ben Tobin
Career and Leadership Coach, Ben Tobin Coaching
Former Software Development Manager, Amazon

Technical Program Manager's Handbook

Empowering managers to efficiently manage technical projects and build a successful career path

Joshua Alan Teter

‹packt›

BIRMINGHAM—MUMBAI

Technical Program Manager's Handbook

Group Product Manager: Gebin George
Publishing Product Manager: Kunal Sawant
Senior Editor: Rounak Kulkarni
Technical Editor: Jubit Pincy
Copy Editor: Safis Editing
Project Coordinator: Deeksha Thakkar
Proofreader: Safis Editing
Indexer: Hemangini Bari
Production Designer: Shyam Sundar Korumilli
Developer Relations Marketing Executive: Sonakshi Bubbar
Business Development Executive: Debadrita Chatterjee

First published: December 2022

Production reference: 1251122

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80461-355-9

www.packt.com

To my partner, Courtney, for encouraging and supporting me on this journey. Without her, this book and the career that formed it would not have been possible.

– Joshua Alan Teter

Foreword

For the past 8 years, I have had the absolute pleasure to count Josh Teter among my most valued and respected friends and colleagues. Josh is a leader not only in terms of the work of being a **Technical Program Manager** (**TPM**) but also, more importantly, he is a champion and thought leader for what it means to be a TPM. He shaped my understanding of what a TPM is and does, and what it means to do this job well. In this book, *The Technical Program Manager's Handbook*, Josh will share his unique and valuable experience and philosophy on the special and crucial role of a TPM in today's technical industries. His journey through this industry has given him a rare perspective on the needs of technical companies and how TPMs enable successful programs by providing a critical bridge between the many other roles and disciplines in the field. In our shared years at Amazon, I have been impressed by Josh's growth in and mastery of this unique and challenging role.

In this book, he sheds light on a role that is still in the process of defining itself. He shares the ways in which it shares skills with adjacent roles and illustrates the unique skills and functions of the TPM role. Most importantly, he teaches fundamental skills for doing the job well and avoiding classic pitfalls.

In reading *The Technical Program Manager's Handbook*, you will learn about the fundamentals, techniques, and best practices for driving clarity and planning, understanding, and communicating the status, progress, and challenges in technical projects and programs. You'll understand how to apply this to the programs and products you will manage in your own role, and how to make a positive impact on the teams you work with. By the end, you'll have a clear understanding of the balance and depth of a TPM's technical role as well as their role as leader and driver.

You will see how a master TPM guides major projects and enterprise-scale programs toward success while optimizing the skills and responsibilities of everyone they work with.

As you read through this book, Josh walks you through his years of hard-won experience as a world-class principal TPM. As a natural teacher, he has created a path that is engaging and informative for you to explore. In his examples and anecdotes, you'll see the reality of theory versus practice, and understand the purpose behind everything a TPM does. By the end, you'll be ready to continue your development as you apply what you've learned to this ever-changing field.

Take this along with you and join the ranks of the TPMs making a big impact throughout the technical fields.

Ben Tobin

Career and Leadership Coach, Ben Tobin Coaching
Former Software Development Manager, Amazon

Contributors

About the author

Joshua Alan Teter is a principal technical program manager at Amazon. He has been in the tech industry since 2007, starting out as a software developer at Weatherford Laboratories. His passion for program management started at Hewlett Packard in 2012 where he joined as a technical lead and oversaw a team of developers. In 2013, Joshua moved to Amazon to join the Tax Services group and helped the team grow from 35 to 280 people. In the years since joining, he has received his Project Management Professional certification and run projects to help open new Amazon marketplaces, launch new products and services, and most of all, lead innovation in terms of tax compliance with high-profile legislation, such as acting as a marketplace facilitator in the United States and in line with Brexit, along with related legislation on EU imports.

I want to thank my wife for being with me on this journey and as a soundboard for the ideas that went into this book. I also want to thank Ben and Sandra for their insight and guidance – this book would literally not be the same without their input. And to the team at Packt for their help and support throughout the process.

About the reviewers

Sandra Boudreau retired from Amazon after 1 year as an SDM and 5 years as a TPM following her 15-year career as a software developer, mostly in the military defense industry. Her career highlights included launching Amazon's tax calculation solution in Brazil and launching Amazon Prime membership in India, as well as developing solutions for the Canadian military as a lead software developer on the trials and demonstrations team at General Dynamics Canada. She is now living in her dream house on the shore of a lake in Alberta, Canada and just starting to put a dent in her Steam backlog.

Ben Tobin spent 20 years in the software industry wearing many hats, most recently as a software development manager at Amazon. His path through the tech industry included working as the director of a small tech company and then dipping into the world of a DevOps engineer before returning to software engineering again and discovering management. In the process, he has interviewed hundreds of candidates and considers developing good engineers into great ones among his greatest accomplishments. He is now a career coach specializing in tech industry careers, engineers, career transition, and ADHD as the founder of Ben Tobin Coaching.

Angel Martinez has experience managing projects across 4 industries over 12 years. He is currently a manager and TPM at Google, focusing on data center networking deployments. He received his BS in mechanical engineering from Worcester Polytechnic Institute. Angel began his career as an associate in a leadership development program working on consumer products. He then transitioned into the transportation and product-building industries. Outside of work, he enjoys spending time with his wife and three children and, with any remaining time, playing indie board games and doing woodworking.

Nirbhay Anand has worked in the role of TPM and completed a master's in computer science and an MBA, with 16 years of industry experience in software product development. He has developed software in different domains, such as investment banking, manufacturing, supply chain, power forecasting, and railroad contract management. Currently, he is associated with CloudMoyo, a leading cloud and analytics partner for Microsoft. CloudMoyo brings together powerful BI capabilities using the Azure data platform to transform complex data into business insights.

Table of Contents

Part 2: Fundamentals of Program Management

3

Introduction to Program Management 31

4

Driving Toward Clarity 43

5

Plan Management 55

6

Risk Management 79

7

Stakeholder Management 93

8

Managing a Program 109

9

Career Paths 121

Part 3: Technical Toolset

10

The Technical Toolset 131

11

Code Development Expectations 141

12

System Design and Architecture Landscape 157

13

Enhancing Management Using Your Technical Toolset 175

Index 185

Other Books You May Enjoy 188

Preface

The role of a **Technical Program Manager** *(***TPM***) has been around inside and outside of the tech industry for quite a while; yet somehow there is still quite a sense of mystery around what the role is and why it is* beneficial, let alone how someone can succeed in being a TPM. This book looks to correct that by diving into what it means to be a TPM, where the role came from, and where it is headed. You'll get a look into how the TPM works and develops their career in the Big 5 – Amazon, Apple, Alphabet (Google), Meta (Facebook), and Microsoft.

I've been at Amazon for a little over 9 years now and I remember that when I first interviewed, I had a hard time remembering what TPM even stood for, let alone what they did. In my onsite interviews, I asked what the job role was and what the day-to-day was like. 9 years later and I'm asked those same questions by interviewees at least once a week. I attend conferences discussing what it means to be a TPM and have written papers on what it means to be a TPM within my own organization because as you'll see in this book, it depends on where you are as to what the role entails. However, no matter what, there are foundational principles that are followed across the industry that will set you on the right path and help you when you get stuck in a rut without a way forward.

Let's get you ready to be a successful TPM!

Who this book is for

This book is meant for TPMs at every stage of their career, including those that are considering transitioning into the role. To get the most out of this book, there is an expectation that the reader will have some basic knowledge of project management. I tend to lean into the **Project Management Professional** (**PMP**) lingo and style but the book does not follow a specific methodology, as I don't believe a single methodology can be adequately applied to this role!

The book will cover some basic programming topics, although very little code is used except for illustrative purposes in *Chapter 11*, *Code Development Expectations*. Most concepts are explored using figures and text, as that fits the audience of the book the best.

To read the book, there's no expectation of a specific technical proficiency, although as you will discover in *Part 3* of the book, there is an expectation that you'll have that if you want to be a TPM. This book will guide you through the technical skills that are prerequisites for most TPMs.

What this book covers

Chapter 1, *Fundamentals of a Technical Program Manager*, is an introduction to what a TPM is and where the role originated.

Chapter 2, *Pillars of a Technical Program Manager*, sets out the three pillars of a TPM: project management, program management, and the technical toolset.

Chapter 3, *Introduction to Program Management*, covers the key management areas that will be covered throughout the book: plan management, risk management, and stakeholder management. It also introduces a case study that will be used for all examples throughout the book.

Chapter 4, *Driving Toward Clarity*, elaborates on the recurring trait that defines everything a TPM does: being clarity-driven.

Chapter 5, *Plan Management*, dives deeper into the plan management best practices and goes over scenarios that are common in the tech industry.

Chapter 6, *Risk Management*, explores the risk management best practices and goes over scenarios that are common in the tech industry.

Chapter 7, *Stakeholder Management*, discusses the stakeholder management best practices and goes over scenarios that are common in the tech industry.

Chapter 8, *Managing a Program*, explains the differences between managing a program and a project and how program management builds on top of project management.

Chapter 9, *Career Paths*, examines the career paths available for a TPM using interviews and job data from across the Big 5 tech companies.

Chapter 10, *The Technical Toolset*, is all about the three fundamental tools in a TPM's technical toolset: programming fundamentals, system design, and architectural design.

Chapter 11, *Code Development Expectations*, is an outline of the programming fundamentals that a TPM is expected to understand and draw upon.

Chapter 12, *System Design and Architectural Landscape*, clarifies the system and architectural design patterns and principles that are useful to a TPM.

Chapter 13, *Enhancing Management Using Your Technical Toolset*, covers the technical toolset and dives deeper into how and where in a TPM's day-to-day work it can be used to enhance their career.

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: `https://packt.link/ytFtY`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This method has two parameters, a string named `message` and an object of the `MemberInfo` type called `miTo`."

A block of code is set as follows:

```
html, body, #map {
 height: 100%;
 margin: 0;
 padding: 0
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
MessageType = MessageInformation.MessageTypes.Text,
To = miTo.ComputerName,
Text = message
});
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Select **System info** from the **Administration** panel."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Technical Program Manager's Handbook*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804613559`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: What is a Technical Program Manager?

The **Technical Program Manager** (**TPM**) role has been part of the technical industry since the beginning but is still shrouded in mystery. This part aims to set the foundation of where the TPM role came from, what it is, and why it is an important and in-demand position in the industry.

This section contains the following chapters:

- *Chapter 1*, *Fundamentals of a Technical Program Manager*
- *Chapter 2*, *Pillars of a Technical Program Manager*

1

Fundamentals of a Technical Program Manager

The role of a **program manager**, in some form, has been around for as long as humans have organized to accomplish a goal, and the **Technical Program Manager** (**TPM**) naturally followed as a result. The TPM plays a powerful role in any technical project or program and has carved its way into the tech industry culture as a mainstay position right alongside software and hardware developers, development managers, and product managers. Even with its ubiquitous role in the industry, the question of what a TPM is and how can we be effective practitioners of this kind is still asked on a daily basis. This book aims to correct that.

In this chapter, we'll start by discussing how the TPM role became what it is today. We'll do this by exploring the roots of the TPM, the generalized program manager role, and the skills and traits that we share. We'll round this out by exploring the basic requirements that are specific to our specialization – the systems development life cycle.

With the fundamentals under our belt, we'll explore the specific attributes that help a TPM thrive at their job. With a better understanding of the TPM, we can widen our perspective to look at the roles adjacent to the TPM to see how we complement one another and how we can fill in the gaps that our team needs us to fill.

Lastly, we'll look into the industry to get a grasp of how the TPM role is defined holistically by exploring job postings, as well as interviews I conducted with fellow TPMs from various companies.

In this chapter, we will explore the fundamentals of the TPM with the following:

- Understanding the modern TPM
- Learning the fundamentals
- Exploring what makes a TPM thrive
- Comparing adjacent job families
- Exploring functional competencies across the industry

Understanding the modern TPM

In the 1967 book, *The Technical Program Manager's Guide to Survival*, by Melvin Silverman, the author defines a program as an organization created to accomplish a specific goal. This organization was a group within a company that existed for this sole purpose and was to be dissolved once the goal was realized. You can see where the computer definition of a program gets its origins – a bit of organized code that executes to accomplish a task! Once the task is complete, the program would terminate.

I found Mr. Silverman's book while attempting to uncover the origins of the TPM role. What I found is similar to the evolution of the word *program*. As it turns out, Silverman's book was one of the first books that used the term *technical program manager*, though it only shows up on the title page – the rest of the book just talks about program managers. Elsewhere in the 1970s and 1980s, the term pops up in various United States government papers, listing someone as the TPM for a given project at NASA, the Department of Energy, and the Department of Defense, to name a few. This had me perplexed, as I couldn't see any role or definition that was recognizable as those of today. So, since Mr. Silverman defined program, I found the definition of *technical* in the Oxford English Dictionary:

technical (adjective). 1. relating to a particular subject, art, or craft, or its techniques, and 2. of, involving, or concerned with applied and industrial sciences.

What we commonly refer to as a TPM —where technical denotes a background in computer science—is actually just one of many instances in which the term technical denotes using a specialization.

As far as the technology industry is concerned, I identified the use of the term TPM at least as far back as 1993, though I suspect it has been in use in the industry as long as the industry has existed given its prevalence in other industries from the 1960s onward.

Old title, new meaning

While researching the origins of the term TPM, I utilized Google's **Ngram Viewer**, which indexes word usage in books and government publications by year between 1800 and 2019. Using the Ngram Viewer results as a starting point, I researched dictionary definitions, half-century-old books, and US government publications from NASA, and found that the TPM title has been around for a while. However, as I'm sure many readers are thinking, it feels as though it's a very recent addition to the workforce. I remember when I was first approached to interview at Amazon for a TPM role, I was confused as to what it was. I asked, and sure enough, it was roughly what I was doing professionally but the company I was at simply didn't use that term. In fact, very few companies seemed to be using the title in 2013 – let alone the 1990s!

Figure 1.1 shows the Google Books Ngram Viewer results for "Technical Program Manager" from 1955 through 2019 in the *English (2019)* dataset with smoothing set to 3. This graph was generated via `http://books.google.com/ngrams` with these settings:

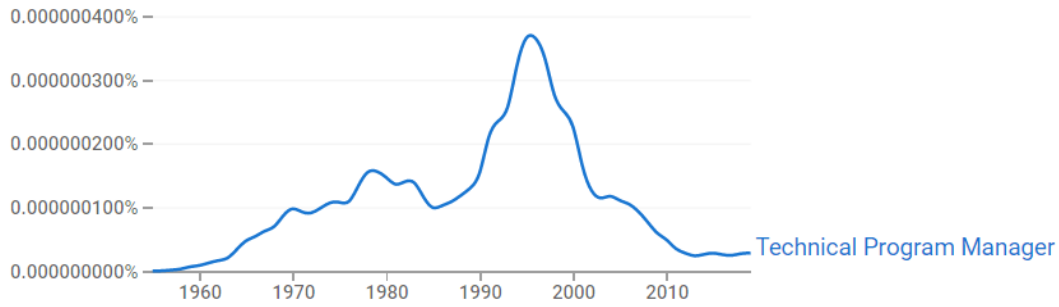


Figure 1.1 – Google Ngram Viewer results of the occurrence of the term "Technical Program Manager" from 1955 to 2019 with a smoothing of 3

Figure 1.1 shows that there is a very large uptick to the highest vertex for the term TPM in the year 1995 – the early days of the World Wide Web and the mad dash of startups rushing towards the year 2000. With these technology companies sprouting up, the need for specialized program management arose – people with a background in and knowledge of the systems being developed so they could be better facilitators and drivers of these new programs and websites. As is the case in the technology industry, trends that start within the few companies at the top slowly make their way down through the rest of the industry until they become common. In some cases, this trend is still working its way down in the industry, as some companies are still not fully aware of the position and its benefits. I believe this explains the lag in the term being seen in publications and more commonly used in the industry.

Now, here we are today with a title used to denote a specialized form of program management being wholly taken over by the tech industry to mean a program manager with a background in computer science or engineering – thus, an old title and a new meaning.

We've explored a bit about where the title of TPM originated outside of the tech industry and its transformation into a specific type of specialized program manager. Next, we'll review the fundamentals of a TPM.

Learning the fundamentals

Throughout this book, we'll discuss many concepts that are core to any program manager, as well as some that are more specific to the TPM specialization. Let's briefly discuss some of these terms so that we have a shared foundation to build upon.

Let's tackle some of the key management areas that project and program managers have in common. As we'll discuss more in *Chapter 2*, these are shared across all program manager roles, including specialized roles such as the TPM.

Project planning is where we work through requirements, resourcing, and constraints and develop an action plan. This makes up the backbone of our work – all the other management areas build from this or feed into it and it is paramount to a successful project. In *Chapter 5*, we will go into further detail on this.

Once you have a project plan, you will analyze the roadmap and identify any *risks* that could arise. These can be related to tight scheduling, resourcing constraints, project dependencies, or scope concerns. Depending on the risk, you may amend your project plan to help mitigate it (such as *swarming* – or increasing resources on a particular task to get it done quicker – to alleviate scheduling concerns).

Throughout these stages, you will be engaging with your stakeholders to provide insight. Requirements often come from one or more of the stakeholders and they may identify risks or mitigation strategies for reducing risks. You'll also develop a strategy and cadence for regular communication with your stakeholders called a **communication plan**.

Figure 1.2 illustrates the key management areas we'll discuss and how they influence each other:

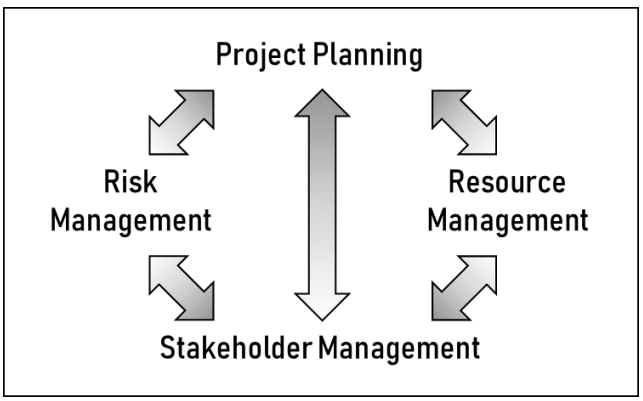


Figure 1.2 – Key management areas

In the preceding diagram, we can see that both project planning and stakeholder management have central roles during the life of your project. Risk analysis and strategies feed into the initial project plan as well as act as continuous feedback. As a risk arises, the schedule may need to be adjusted. The same is true for resource management – if you lose or gain resources, your project plan will need to be adjusted. The available resourcing also plays heavily into your initial timelines. Though some organizations resource based on an optimal plan, in that they will determine the quickest most efficient path to completion and resource according to this need, most tech companies provide resourcing based on prioritization of the project and the schedule adjusts based on what is available. If the project is deemed to be a high priority, more resourcing may be given to hit a specific date, and conversely, may be given less resourcing if there is higher priority elsewhere.

Each of these management areas also feeds directly into stakeholder management – especially around standard communication routines. Any changes in the schedule, resourcing, or risk realizations should be immediately communicated by the TPM to the appropriate groups of people based on the communication plan.

Now that we've covered the program management fundamentals, we'll move on to concepts that are aligned more closely with the technical specialization aspect of the role.

The Systems Development Life Cycle

The **Systems Development Life Cycle** (**SDLC**), sometimes written as Software instead of Systems, is fundamental for a TPM to understand, as it is central to both software and hardware development. As it is a cycle, by nature, it is iterative. *Figure 1.3* illustrates the cycle, starting at the top with requirements analysis and following clockwise to come back around to this initial step:

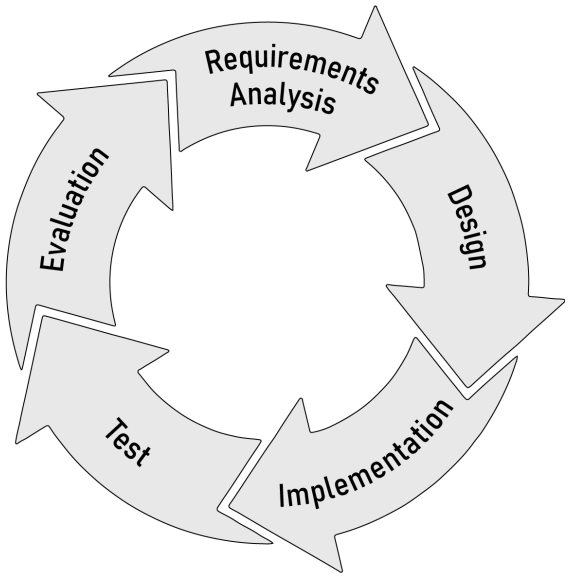


Figure 1.3 – The SDLC

The number of phases in the SDLC can vary depending on the situation. This configuration incorporates what I see as the main phases that need to be involved for the cycle to be successful. Starting at the top, the flow is as follows.

At the beginning of a project, the SDLC starts with the **requirements analysis** phase. These may already be well established or are newly discovered or added requirements that need to be broken down. Once these requirements are better understood, the **design** phase is started, which takes the requirements and builds a design that meets the requirements. The design then drives the **implementation** of the actual product, which then leads to *testing*. The final phase is **evaluation** or iteration. This step involves looking at the product and looking for improvements. These improvements may also come from feedback from stakeholders and customers. Once they are identified, you begin requirements analysis once more and the cycle continues. Though this looks to be a waterfall approach, where all steps of a phase such as requirements gathering are completed before moving on to the next phase, this cycle can happen as often as needed. A single requirement may go through this entire cycle while another is being clarified and may proceed to design much later. To that end, the cycle is utilized in waterfall, agile, and hybrid environments.

Many other pieces can contribute to the technical fundamentals, which we will cover in *Part 3* in more detail. Those skills will vary from company to company, as well as from team to team within a company, as the needs of that team can vary. Keeping that in mind, the SDLC is a fundamental understanding across all variations of being a TPM.

Now that we've covered the fundamentals, let's take a look at what skills or traits make a TPM the most successful at their role.

Exploring what makes a TPM thrive

This topic comes up in every interview I've conducted for the TPM position. So, I've put some thought into everything that a TPM does. The items I'm going to discuss are not qualities specific to a company, team, or variation on the TPM role but are transitive from one position to the next.

Driving to get things done

First and foremost, for a TPM to thrive, they need to focus on pushing forwards and getting things done. It sounds a bit cliché as though you are reading it from a job description, but it is resoundingly true. Our innate drive to solve roadblocks, build the plan, mitigate risk, and drive towards deadlines are key to a project's success. We don't want to lose momentum because the second we do, we risk losing all progress and having to start over. This may be because, during the time the project is blocked, the individuals working on it may move on or lose context as priorities change, necessitating more time to ramp back up. It is in our best interests to resolve issues quickly because keeping engagement, focus, and motivation facilitates an immediate and more dedicated response.

This is a trait that isn't always present in adjacent roles to the TPM, such as the development manager and the product manager. I believe this is because the primary function of those roles is different from the TPM role. For instance, a development manager's main focus is their people, and a product manager's focus is their product. For the TPM, our focus is the project or program – this is what our drive and perspective hone in on. A blocker in a project blocks our main purpose, so it is extremely important to us. This same blocker may be overshadowed by a performance review or by the bigger picture of the product roadmap in the other roles.

Driving towards clarity

The second trait that makes us thrive is when we take that drive, and we drive towards clarity. This trend to clarify can come at any phase or stage of a project or program. It's easiest to see during the requirements and project planning phases, as clarifying is a main attribute of the planning phase – clarifying requirements, clarifying scope, clarifying the communication strategies, important stakeholders, and so on.

However, driving towards clarity doesn't stop there. Every roadblock we encounter requires us to add clarity. First, we ask about the problem, the people or systems involved, any proposed solutions, and paths to escalation. Then, we take all of this data and define the problem and solution. We define the problem to make clear that our solution is fixing the right thing. Think of it as asserting your acceptance criteria for a development task.

One way in which we drive clarity plays off of the last critical trait to make a TPM thrive, the communication bridge. We'll talk a lot about communication throughout this book, but this is a special form of communication that takes our technical background into account.

Communication bridges

Since we have a technical background, not only can we talk with our developers, but we can also understand them! This is critical for our ability to understand the tasks it will take to complete a project and to push back on estimates or provide feedback. This skill also allows us to explain the work of our developers to other stakeholders. We can take highly technical information and transform it into language that a VP of marketing will easily understand.

This ability to translate from technical to non-technical works in reverse as well. In fact, we rely heavily on this early on in a project during requirement analysis. We take business requirements and translate them into functional specifications for our development team. We understand the business and their needs, and if required, we understand their knowledge domain. For instance, if our business team are accountants, then we have to understand accounting – what they do, what pain points they have, and what jargon they use. Knowing the domain isn't always required to land a job, but the ability to learn the domain on-the-job is key to being a successful TPM.

By understanding our business and their needs, and our development team, we effectively bridge the communication gap between business and technology. *From a career growth perspective, this soft skill is by far the most important skill to perfect.* Due to this, it is also the skill that shows up most in the growth category for career progression. To improve this skill, ensure you practice **active listening** by staying engaged with the speaker, affirming your understanding through body language, and following up with questions. This is a skill that I am still perfecting myself. I grew up in an environment where it was common to interrupt or talk over others during particularly engaging conversations. Though this can be fine among friends when talking about trivial matters, this is very disruptive in a work environment – especially given that at work, we are often trying to solve a problem. When you interrupt, you aren't taking in what the other person is saying. Instead, you are looking for a space in the conversation to insert your own thoughts and looking for this opportunity prevents us from fully taking in what the other person is saying. Active listening will ensure that you truly understand the point of view of the other person. Often, this information can build on your understanding of their problem. Follow-up questions can fill in the gaps in your understanding to paint a more complete picture of the situation. Over time, you'll understand your business teams and their domains, which, in turn, helps you become a more effective communicator. The better you understand the pains of your business team, the better you can relay that to your development team in your functional specifications and conversations about scope. The more they understand, the better the outcome of the software and the more successful the project will be.

We now understand what makes a TPM thrive at their job, as well as the fundamentals to accomplish that job. Next, let's compare the roles and responsibilities of a TPM to other roles that are often found adjacent to it.

Comparing adjacent job families

I love Venn diagrams because they show similarities and differences in a very concise and easy-to-follow way. Needless to say, you'll see a few in this book.

A common question in my profession as a TPM is: what is the value of a TPM in a software team? The question is understandable because there is a lot of overlap between our job and the jobs of the roles most often adjacent to us in a software or hardware team.

We'll start by exploring *Figure 1.4*, which shows the intersection of the roles most similar to the TPM, the PM, and the **Product Manager-Technical** (**PM-T**). Just as the TPM is a program manager that specializes in technology, the PM-T is a product manager that specializes in technology. The term PM-T may not be used at every company (just as TPM isn't used everywhere) but the particulars of the job are the same:

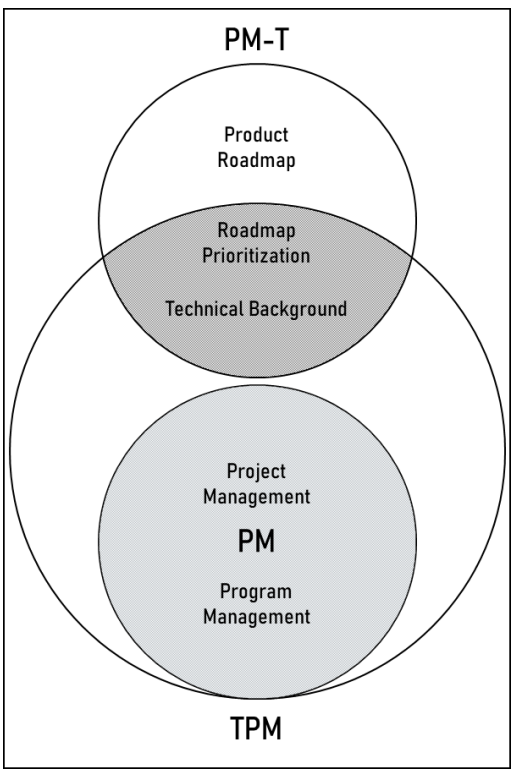


Figure 1.4 – A Venn diagram showing the PM, TPM, and PM-T

The similarities between a PM, TPM, and PM-T are greater than their differences. The TPM, as a specialized version of the PM, shares the same skill sets and adds technical depth to these. The PM-T shares the same technical depth as the TPM, as well as with regards to organizing and prioritizing a roadmap, but specializes in the product roadmap instead of projects or programs.

In *Figure 1.5*, we'll take a look at roles that are quite common to see next to each other, the TPM, **Software Development Manager** (**SDM**), sometimes called a **Software Engineering Manager** (**SEM**), and the PM-T:

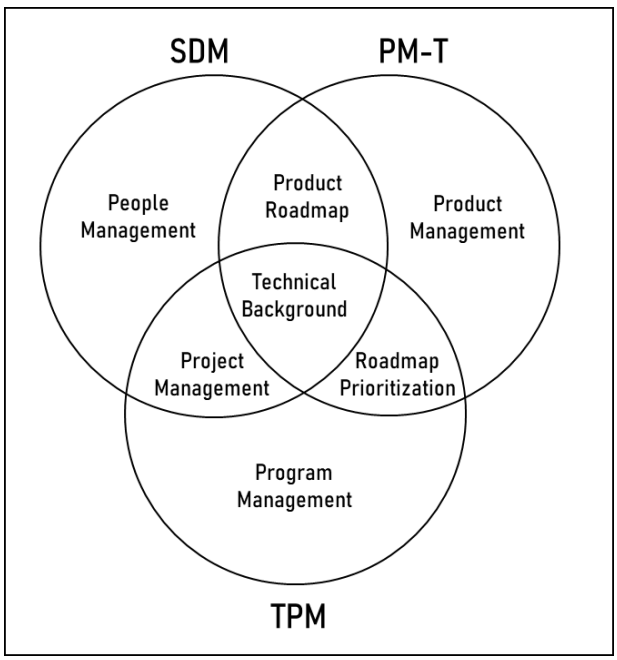


Figure 1.5 – A Venn diagram showing the TPM, SDM, and PM-T

Although this diagram is a simplification of all of these roles, it does represent the typical alignments for these roles. The TPM shares a technical depth with both the SDM and PM-T and project management with the SDM. Most SDMs will run projects that are related to their domain or services, though they aren't expected to be large or too complex from a project management perspective. To this end, they aren't expected to be able to handle an entire program that lies completely within a PM or TPM's realm of expertise. The SDM and PM-T can both create a product roadmap, and this is often a gap the SDM will fill when a PM-T is absent. Lastly, the PM-T specializes in product management and shares prioritization skills with the TPM. Simply put, at a pinch, these roles can often cover gaps in the team but given each role has unique skill sets – this would ideally only be short-term. For example, a PM-T isn't necessarily needed for a small team with a single service, but as the team grows and the product becomes complex or begins to serve many diverse types of clients, a PM-T should really be brought on board. The same applies to a TPM – once the number of stakeholders becomes too large and complex, it begins to fall out of the expected comfort zone for a SDM and a TPM should be hired.

Wearing many hats

At this point, you can see that the TPM role has many functions. The Venn diagrams show that we overlap a lot with many adjacent roles. This unique overlap allows our skill sets to merge and fill in any gaps depending on the needs of our team. Over the course of my own career, I have found myself filling in the gaps of missing product managers, process managers, and even people managers by helping guide the developers in my projects.

This aspect of the job makes it extremely difficult to define the exact nature of what we do. The short answer is that we do it all. Our job is to ensure our programs and projects are successful and that may require different skills depending on the context of what is going on.

Even though this book talks about specific skills, they are not all in equal measure and can even change from team to team within a company! The needs of the team are often unique to what it is trying to solve or how mature the team is. The reason I have filled in so many gaps over the years is that the team I have been on has grown 10 times in size while I've been here. This means that at various times, the skill gaps or needs in the team at that moment have changed. In general, the more mature the team, the more defined your role in that team will be.

Now that we understand the functional aspects of the TPM role, and how that impacts how the job may manifest itself based on our team needs, let's see how consistent this holistic view is across the industry.

Exploring functional competencies across the industry

I've worked for most of my time as a TPM at the same company – as such, I was concerned about an unconscious bias of what a TPM is or should be. To combat this, I sought outside perspectives from as many high-profile companies as I could – Amazon, Google, Meta, Microsoft, and Apple. To help confirm the interviews I had, and to fill in gaps where interviews weren't possible, I combed through the job boards of these companies to see what each was looking for in a TPM.

I consolidated the interviews and job descriptions into a matrix in *Table 1.1* that we'll dive a bit deeper into. If something popped out to me as different in a particular interview or job post, I'll call out what it is and why it's interesting:

Normalized Level	Company	Qualifications	Education
Entry	Apple	SDLC PM	CS or Comparable
	Google	Align across multiple teams PM/SDLC	CS or Comparable
	Microsoft	PM/SDLC Influence without authority Biz Intel	CS or Comparable Equivalent Work Experience (EWE)
Industry	Amazon	PM/SDLC Remove ambiguity Thought leadership	CS or Comparable EWE
	Apple	Leading a team Established PM/SDLC Communication Strategy and Program Delivery	BS or MS EWE
	Google	E2E Delivery System Design Data Analysis	CS or Comparable
	Meta	PM/SDLC Works with other TPMs	CS or Comparable EWE
	Microsoft	Exp. Writing code Defines program goals PM/SDLC	CS or Comparable EWE
Principal	Amazon	PM/SDLC Remove ambiguity Thought leadership	CS or Comparable EWE
	Microsoft	Proven PM Strong technical proficiency Excellent communication	BS/MS in CS or Comparable

Table 1.1 – A functional comparison of job roles across the tech industry

As you can see in the table, across the industry, there are more similarities than differences in the TPM role. This was a bit of a shock for me, to be honest. We've all heard of specific cultural stereotypes for Google, Meta, Microsoft, Amazon, and Apple – however, when it comes down to their expectations for the functional side of a TPM, they're close enough to call them equivalent.

The expected education, regardless of level, was pretty consistent with some sort of technical degree or **equivalent work experience** (**EWE**) being required. For qualifications, this varies somewhat based on the level, but the pattern is similar. At the entry level, the focus is on fundamentals such as the SDLC and basic project management. For industry hires, it's the core of program management and stellar communication, and then the principal level is focused on impact and proven results.

Insights into the TPM life from interviews

One thing that this matrix doesn't cover extensively is style, which I was able to learn more about from the interviews I conducted. So, let's talk a little about those things that stood out here.

Of the companies I held interviews for, Meta (formerly called Facebook) was the youngest, having formed in the mid-2000s. Due to this, their standards for project management are, as the interviewer put it, based on a "*do what is needed*" mentality. This is by no means bad, as many companies use this strategy to great success. There wasn't much from the job boards that clues us in on this but the focus on the SDLC does seem to agree with the *bottom-up* approach to management that the interviewee referenced, where the drive is from the engineering teams.

At the other end of the tenure spectrum is Microsoft. I think a lot of people think about Microsoft's own *Project* software and then lump it in as a highly regimented, PMP-style organization. As it turns out, this isn't true! Though I am sure some people use Microsoft Project there, it isn't standardized. The interviewee I talked to said many of the individuals they've worked with use Excel! Also, due to their tenure in the industry, they pre-dated the *TPM* job role and therefore mainly use the PM role title. Some organizations are starting to use the TPM role, both as seen on the job boards and also confirmed in the interview – the TPM title, and the matching compensation and requirements, are starting to be used in place of the PM title in areas that require technical expertise.

I've heard many people suggest that the TPM role originated at Amazon and that does make some sense. Back in 2013 when I interviewed at Amazon, I had to ask what the TPM role involved, as I had never heard of it! A quick internet search didn't turn up any results either. But through some discussions, and simply looking at the timelines, you can see that the TPM role pre-dates Amazon. I wouldn't be surprised if it was one of the first to widely use the job role based on conversations internally and externally, but it is hard to say for sure. Amazon is also a "*do what is needed*" organization, where the TPM role can vary quite a bit from team to team based on the gaps and needs of the team.

An interesting takeaway in my discussions about Google is that they are heavily product focused. A good portion – up to 30% – of the TPM's schedule is dedicated to working with product managers on the product roadmap and backlog grooming based on the roadmap. The job descriptions hint at this as well when they discuss that Google is product-focused and how cross-team communication with engineers and product managers is a must.

Lastly, we have Apple. As it turns out, Apple doesn't allow interviews with current employees about Apple life, so for this iteration, we only have the job descriptions. However, one interesting takeaway is that they are the only company (of those that I researched) that lists having a **Project Management Professional** (**PMP**) certification as a desirable (as in nice to have, but not required) skill to have.

A quick look into the main TPM career levels

Now that we've discussed different job families, let's dive a little closer at the TPM role itself across the life of your career. *Table 1.2* looks at the different levels of a TPM – entry, industry, and principal.

Level	Company	Focus	Years of Experience
Entry	Apple	SDLC/PM Fundamentals	None
	Google	SDLC/PM Fundamentals	0 to 1
	Microsoft	SDLC/PM Fundamentals Influence w/o authority	None
Industry	Apple	Strategy and Program Deliver	5 to 10
	Amazon	SDLC/PM Fundamentals Remove ambiguity	3 to 8
	Google	End-to-End Delivery System Design	2 to 6
	Meta	SDLC/PM Fundamentals Cross-team collaboration	4 to 7
	Microsoft	Program definition SDLC/PM proven record	6 to 10
Principal	Amazon	Program delivery Identify ambiguous problems	Over 8
	Microsoft	Proven PM/SDLC Strong technical proficiency	8 +

Table 1.2 – The TPM progression through three job levels

These terms are not standardized, but they still line up well with the various levels that I've encountered in my research for the book. The entry level means being 0 to 3 years out of college, and the scope and expectations are focused more on the fundamentals of the job. Industry level means 3 or more years of experience in the industry with no hard upper cap. This is the most common row with the largest growth curve. You go from a relatively new TPM with a basic understanding of the role and expand into mastering your domain. This is often the highest level of your career as a TPM, as most either stay at this level or transition to other job families. The last level that is common among these top companies is the principal level. Often this is the highest level achievable as a TPM across the industry, which puts you at the same level as directors or senior managers. However, this means the expected scope and impact are also at the same level as those directors, making this level more difficult to achieve – though certainly not impossible!

Now that we know what the TPM role looks like across the industry as well as the various levels, let's see how consistent this holistic view is across the industry.

Summary

In this chapter, we discussed the fundamentals of being a TPM. We started by explaining where the TPM overlaps with the generalized PM and program management and touched on the bare fundamentals for the technical aspect of our specialization. Then, we examined the TPM role specifically to see what makes us thrive and found that drive, clarity, and being a communication bridge set us apart from our colleagues. Next, we expanded our view to look more closely into how our specialties overlap with the job families around us and across the industry.

We'll continue to build on these fundamentals as the book progresses and as we dive deeper into these topics to get a solid understanding of our role, how to be the most successful at the job, and how to progress your career the way that fits your skills and desires best.

Next, in *Chapter 2*, we'll discuss the building blocks, or pillars, of the TPM role. We'll build on the fundamentals we tackled in this chapter and dive deeper into the program management and project management strategies that help make us successful. Lastly, we'll cover the technical nature of our role and how it feeds into the program management skills and techniques. Just as with the foundations found in this chapter, the pillars we discuss in *Chapter 2* will be built upon and referenced throughout the book.

2

Pillars of a Technical Program Manager

Throughout most of my professional career, I have been a **technical program manager** (**TPM**) – or a proxy to it. During this time, I have worked alongside other TPMs on large-scale projects, mentored junior TPMs on their journey, and I have interviewed a large number of TPMs from all corners of the industry. Over the years, I have adopted a considerable perspective on what a TPM is and what makes them successful.

Without getting into complex sentence diagraming, let's take a look at the job title itself, Technical Program Manager. The title is comprised of two parts, the noun, *program manager*, and the adjective, *technical*. From this, it's easy to see that there are at least two foundational pillars to a TPM. The *program management* and the *technical skills*. In my view, program management consists of two separate pillars since program management includes project management. You cannot have the skills to run a program without also having the skills to run a project. However, the reverse is not true – you can certainly have project management skills and not have the skills to run a program. So, that's our third, and final, foundational pillar.

In this chapter, we're going to dive a bit deeper into each of these foundational pillars by exploring the following:

- Understanding project management
- Diving into program management
- Exploring the technical toolset

Understanding project management

Before diving into the first pillar, let's take a quick look at how the pillars support each other, as illustrated in *Figure 2.1*:

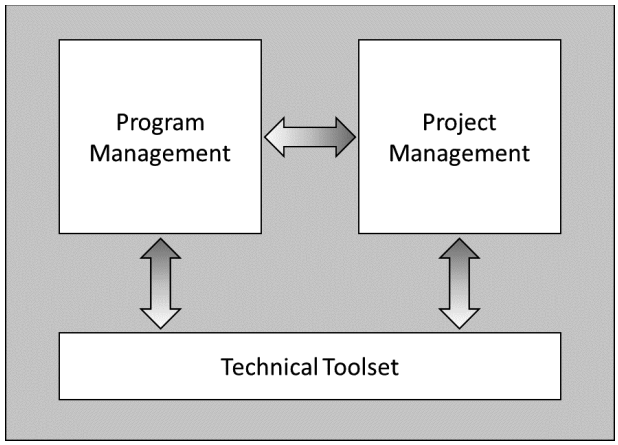


Figure 2.1 – The pillars of a TPM

Program management and project management are directly related to one another and complement each other within every program manager role. The program management phases and tactics build on top of the project management phases and tactics. And since a program involves multiple projects, they often share direct feedback, as risks at the program level can manifest at the project level and vice versa. The technical toolset supports both the program and project management pillars, as it provides the technical acumen to refine and more effectively utilize those skill sets.

To see how the three pillars relate to various managerial roles, we can look at *Figure 2.2*. The first two pillars are shared with other program and project manager roles. Nonetheless, they are an important part of our foundational skills:

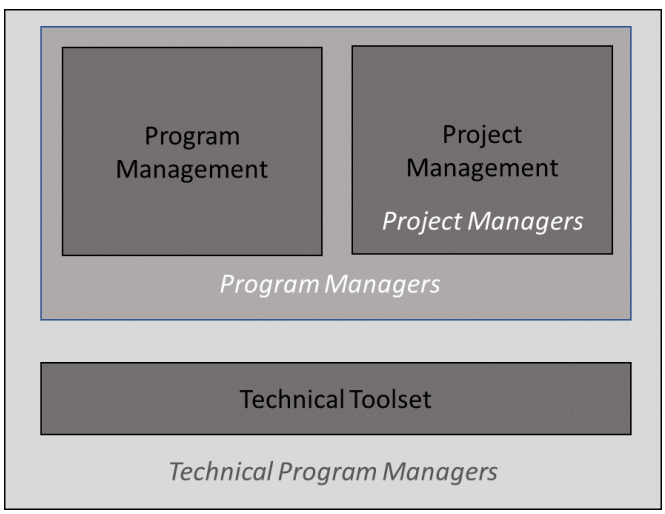


Figure 2.2 – The functional overlap between the role of program managers and TPMs

Though we call ourselves program managers, a lot of what we do day to day, especially earlier on in our careers, is related to project work. So, we'll start there.

Most of what we do is measured through our ability to deliver results, either directly or indirectly. Did we reach the milestone on time? Did we effectively unblock the low-level design quandary of our developer? Was the stakeholder looped in to help drive a discussion to the right conclusion? These are all different ways that we move a project forward, drive towards clarity, and ultimately deliver our goals.

In the project management life cycle, there are many opportunities to deliver results – not just within the project itself. At every stage, we are tested on our ability to take what is ambiguous, add definition, and refine it. We analyze risks and stop problems before they become problems.

In later chapters, we'll dive deeper into the major stages of a project to see how a TPM utilizes their technical skills to effectively manage a project. For now, we'll go through the general steps we take as project managers to successfully deliver our projects.

Exploring the typical project management tactics

We'll go through a quick overview of the tactics used in a few of the high-impact areas of a project. These are the areas that utilize the TPM's skills the most. They will be explored throughout the book in various levels of detail and with various perspectives on the tools and work they involve:

- Project planning
- Resource management
- Stakeholder management
- Risk management
- Communication

Let's take a look at each of these in more detail.

Project planning

Starting with **project planning**, a project manager takes the given requirements and builds out a breakdown of the work structure and a work list, which then develop into a project timeline with dependencies, durations, and planned estimates. In many project management tools, these are simply a single artifact along with the resultant Gantt chart, though traditionally, this involves many different steps. In this regard, this seems to be a lot of time and effort, but a seasoned and well-prepared practitioner can get through this quickly. The amount of time and effort also varies depending on the size and complexity of the project or program.

During planning as well as throughout the project, we manipulate the **project management triangle** to get the outcome we need. We often see it depicted as an equilateral triangle (as in, all sides and angles are equal) that represents the trade-offs between the project scope, timelines, and costs – this is done to illustrate that to remain equal, any change to one side of the triangle necessitates a change to one or both of the other sides. For instance, if you lose people, the timelines grow, or the scope has to shrink to make up for the loss. Our day-to-day project negotiations are always tied back to the triangle and the trade-offs required to maintain the quality of the deliverable.

Figure 2.3 illustrates the project management triangle where scope, resources, and time are bound together:

Figure 2.3 – The project management triangle

As you can see, any change to a side length – which represents the number of resources, time, or scope available to the project – will require an adjustment to one or both of the other sides to maintain equal (or balanced) sides. Without this balance, you cannot ensure the quality of the project or program goal. We'll cover this in more detail in *Chapter 5*.

The resulting project plan represents the vision of the project and is the basis for all the other stages that we'll cover. This is the metaphorical blueprint that TPMs follow. The ability to craft and maintain a project plan may not seem super significant, but I am often reminded of the fact that non-project managers have a hard time reading and interpreting a project plan. This is *our* language. Without a plan, the project is doomed to failure, leading to reactive actions instead of proactive delivery. It can reveal poor planning or estimations, a project trending yellow as time slips, or a risk rising in probability. It is a living document guiding our path and the decisions needed to reach our goals.

Resource management

Resource management also pulls directly from the plan, as it lays out the type of resourcing needed at each phase of the project. Resources can include developers, **user experience** (**UX**) designers, software testers, vendors, contractors, and anyone else that is contributing to a task in the project plan. Conversely, this can also include hardware procurement, software installation, or other tangible needs of the project that are not people related.

Although this sounds straightforward, a project can very quickly stall if the right people and assets are not available when needed. How much of a problem this will be for you depends on how your company approaches resourcing and prioritization – it could be a constant struggle or a negligible one.

Stakeholder management

Stakeholder management, though separate from project planning, uses the plan to help inform the frequency and style of communication. In many companies, this style includes written status reports with varying levels of detail depending on the need of the audience, and meetings to convey the current status (which may or may not include a written report of the meeting).

A large number of teams may indicate a need for multiple types of stakeholder engagement – vendors likely require different status mechanisms to convey the needs or desires of the group than internal teams do. Each of these aspects plays into how the stakeholder communication plan comes together. Being consistent and reliable here instills confidence in your executive sponsors and leadership that the project is headed in the right direction.

The frequency can vary based on the number and type of stakeholders as well as the complexity of the project. For instance, a plan with tight timelines, or a lot of dependencies, may warrant a more frequent communication cycle to help ensure the project stays on track by surfacing problems more quickly to the relevant groups.

Risk management

Risk management, or the identification of risks and the strategies that are planned to handle them once realized, is one of the most powerful cycles in managing a project. It touches all other aspects of the project that we've talked through because a risk – or opportunity – can occur anywhere! To that end, every management cycle also has risk identification and assessment as well as tracking embedded into the work being done. A good project manager never stops thinking about risks and opportunities. We see risks from a mile away and hone in on them when they become relevant. We don't let them alter our course, as we have already planned a course correction.

Communication

The key to all of these aspects of project management is **communication**. Without it, none of these cycles carry weight or value – you share your intended plan with stakeholders, you work with your teams for resourcing, and you discuss and analyze every risk you can think of. This is also true when working with your development team – you listen to them, understand their needs and concerns, and act upon them. In short, communication amplifies cohesion and is a vessel for transparency.

A well-written status report conveys urgency where needed and demands action. It calls attention to problem areas and aligns everyone on the next steps. It lays out all the risks so that when an issue does arise, it isn't a surprise. It instills confidence in your ability to deliver the project in all the right people.

Now that we've discussed project management and the typical tactics that we use while executing the project, we can take what we've learned and scale it up to program management.

Diving into program management

As you progress in your career as a TPM, your scope and impact will increase. One way in which your scope will increase is through running programs. Instead of just the day-to-day operation of one project, you will manage or oversee multiple projects with a shared goal.

What is a program?

As discussed in *Chapter 1*, a program is a combination of multiple projects, often spanning multiple years, that achieves a common goal. Each project may have its own goal, but it will also contribute toward the goal of the program.

Though seemingly straightforward, let's take a look at it from the perspective of a large, multiple-month project, instead of a program. The first thing you would do is break the project up into smaller deliverables, or milestones. Each of these should deliver a feature or functionality that, when put together with all the other milestones, achieves the goal(s) of the project.

A program is very similar to this concept, but usually on a much larger scale. Instead of a multi-month single project, the end goal may take multiple years. Each of the *milestones* may be different enough to warrant different personnel or strategies. In that sense, each of these becomes a project in itself (with its own milestones) that spans the program timeline. They all come together to deliver the goal of the program.

To better demonstrate this concept, let's take a look at a visual representation of a program and the projects that comprise its body of work:

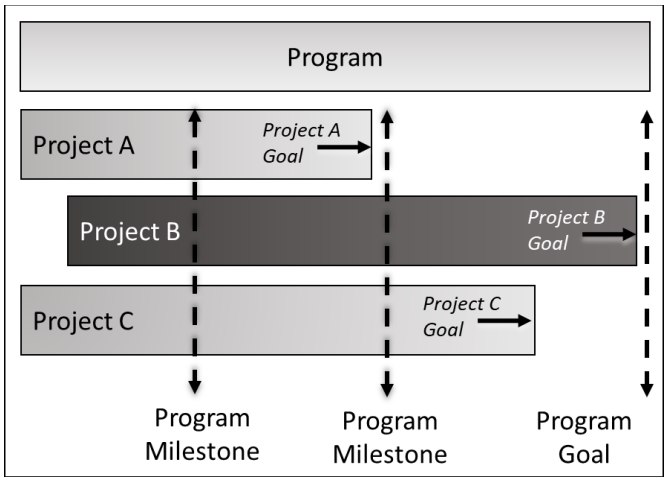


Figure 2.4 – A project versus a program

In *Figure 2.4*, a program has three planned projects, **A**, **B**, and **C**. All combined, the projects span the life of the program. Each project has its own end goal and its own internal milestones (which are not pictured). Notice that just as with tasks within a single project, not all the projects within a program start at the same time and can vary for the same reasons – whether resource constraints or internal and external dependencies. For this program, there are also program-wide milestones being tracked that align at different stages across the projects – project **A**'s end goal and the second program milestone are the same. A program-wide milestone can help provide cohesion and context across teams, driving better alignment for the program.

Now that we know what a program is, let's explore how we use our skills to drive and manage a program through its various phases.

Typical program management tactics

As a program manager, technical or otherwise, the approach toward managing a program is similar. The stages are the same as they are for a project, just on a larger scale, which leads to changes in the skill sets and approaches at each stage.

Project planning goes from determining a single project plan for a single set of requirements to multiple sets of requirements and plans all contributing to the same end goal. The coordination and effort involved at this level increase as the number of projects increases.

Unlike a typical project where the planning stage happens once, the planning stage of a program iterates as projects open and close. Though a singular plan is created in the beginning, multi-year programs will revisit the planning stage often, as projects come into better resolution as they get closer – similar to how planning the estimates for a task in the work breakdown structure is more refined once the breakdown of the task is understood better.

With multiple projects, there are often multiple leaders or drivers, each maintaining their own plan schedule that ties into the program plan schedule. This, again, is similar to a project plan but with a larger scope. A project's plan is overseen by the TPM with input from the people doing the work (developers, system developers, and so on) as far as estimates for the tasks are concerned. Further up, the program is overseen by a TPM with input from other TPMs, PMs, and SDMs with estimates and timelines for their respective projects.

Stakeholder management is similar to project planning in terms of the increase in scale from project to program level. Where a project may move quickly and warrant a bi-weekly status update, a program may have a status update monthly or quarterly. The goal here is to let stakeholders know that the program is moving in the right direction. This could be that each project is delivering the right thing at the right time, or that a problem exists, and the right measures are being taken to correct it. In this scenario, the high-risk items from each project are showcased in a report, as opposed to listing all the risks when sending a status at the project level.

By now, you can follow the general pattern of project versus program – scope and communication complexities both increase at the program level. The same can be said for risk management as well. A program tracks the risks for each project within its scope as well as the risks present at the program level. These risks follow the same scope pattern; think of inter-project dependencies for a program instead of inter-task dependencies for a project. Depending on the type of organization, funding may be of greater concern within a multi-year program compared to a multi-month project. **Projectized organizations**, or companies that fund and tie resources specifically to a project, may treat the program as a singularly funded entity for its life span. However, in the tech industry, funding tends to follow a yearly cycle more often. Within a given year, the funds are often fixed to the project, but priorities are re-evaluated yearly, and projects may be re-prioritized or de-funded as other needs arise.

The communication of risks, as alluded to in the stakeholder discussion, will often focus on high-risk items that are a threat to the overall program and leave the project risks to the project level. A good analogy here – and true in general for all aspects of program versus project – is the satellite imagery of a city. From 17 kilometers above the ground, the city shape as a whole is well defined due to the density differences between it and the surrounding area, but spotting the medical district as compared to the core of the city's downtown is likely hard from this perspective. Zooming in closer to street level (around 4 kilometers), these distinct districts come into focus, yet the city-wide view is now gone. Next, zoom all the way in to the street level and you can discern one office building from the next, and one medical wing from another. Each of these views has its purpose but to see the full city, you have to lose this specific focus on a singular building or district.

Let's look at a representation of a program-level status update as well as a status update from a project within that program to better understand how they connect:

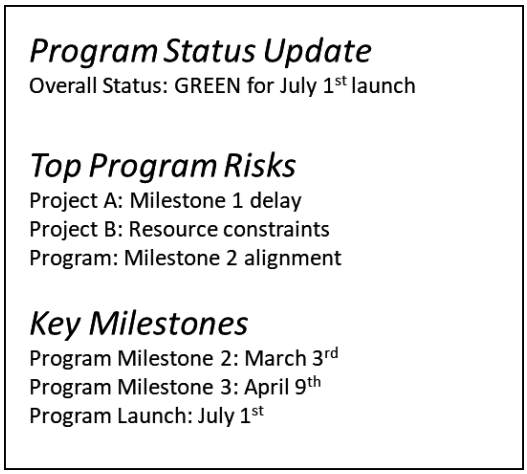


Figure 2.5 – A program status update

Figure 2.5 shows the type of details the audience of a program status update might want to see at a high level. The concerns are at the program level and only items that impact the program's **critical path**, or the sequence of dependent tasks that amounts to the longest time (and represents the minimum time in which a project can finish), are brought to light.

Figure 2.6 – A status update for Project A

Figure 2.6 shows a zoomed-in view of **Project A**'s status update. This goes deeper into the project to the development tracks, as those have more meaning at this level of granularity. A slip in a story's delivery can delay the next story from starting, but may not impact the milestone, therefore having an impact at this level more than it would at the program level.

Now that we've covered both program and project management and their tactics, which are shared across all program manager roles, let's discuss how your technical background provides you with the toolset to amplify your existing program and project management skills. This is the key to what sets a TPM apart from a generalized project manager.

Exploring the technical toolset

There's a reason why the *T* comes before the *PM*. The technical nature of the position is what differentiates a TPM from a generalist program manager. The area of focus is so deep within technology teams and technology products that specialized knowledge is needed to properly manage projects and programs. To be clear, not all companies actually use the term *TPM* and continue to use the generalized *PM* moniker instead. Make no mistake, though – positions that involve dealing with technical teams definitely require a technical background.

As a TPM, you need to have some foundational knowledge in the technology space. We'll go over the particulars across the industry throughout *Part 3* of this book, but in general, that background should afford you some intermediary knowledge in software or hardware engineering. Although some jobs may cite a specific type of degree (for example, computer science), related degrees and experience suffice in many cases. At the end of the day, if your technical toolset allows you to navigate technical teams, technical requirements, system diagrams, and planning, then you will succeed.

Earlier in this chapter, we went over the general skills and tools that a generalist program manager utilizes. These tools are great to ensure the proper delivery of a program or project across a wide range of disciplines. Now, we've learned that a technical background is key to the success of a TPM. Next, we'll learn why this technical toolset is so effective.

Discovering the effectiveness of your technical toolset

While studying for my **Project Management Professional** (**PMP**) exam, the course instructor stated that the skills I would learn would allow me to walk into any job that requires a PM and perform the job. If that were true, why is there a need for a specialized PM in technical situations? *Trial and error!*

The tech industry hasn't always had a TPM role – and as stated earlier, not all companies use the term, even today. However, over the last few decades, it's been shown that a program manager who understands the jargon, complexities, estimations, and designs of the software and hardware world leads to a more productive and successful program manager. The more successful the PM, the more successful the program, and thus, the TPM is brought in!

This doesn't just apply to the tech industry. You certainly could take your PMP certification and walk onto a construction site management office and apply for a program or project manager position and get the job. You would have the skills, as you would understand the PM lingo and processes. You could effectively handle change management, stakeholders, and multiple dependencies. However, you'd struggle because you don't understand construction at the level needed to be most effective. You might

be able to manage a plan and reason out a good course of action, but it will take you longer to notice risks, mitigate issues, and estimate properly because you don't have the knowledge and experience for this. Over time, you'd learn patterns and standard behaviors that would accelerate your performance but every nuance within an unknown situation would slow you down.

Experience in the field you are managing is invaluable to your success.

If you have developed software in the past, you will have a good understanding of the software estimates that your developers give you and can challenge the estimates when needed. Your understanding of architectural landscapes and system designs can help you see how multiple projects fit with one another – how one project's design may cause risk through interference with another, or allow you to realize a positive risk, where two projects in a program share design elements and can thus shave development time off one or both projects.

Here's a real-world example that might be closer to home for you from my professional software development days, where experience in your field matters. My first job as a developer was at a geochemical laboratory writing software applications to interface gas chromatographs and **Total Organic Carbon** (**TOC**) analyzers, among others, into a central lab system for analysis and reporting. My first degree was in geology, and I programmed as a hobby, so it was a natural fit to use my university knowledge to help me in my software development. At the same time, I was fortunate enough to have a desk right on the other side of the lab wall. I could see my applications in action, run tests, and be called upon by lab technicians that had a problem with the patch I had just pushed to their machines.

I had meaningful interactions with the lab technicians and actually understood their problems because I understood their domain: oil. Sure, I had to learn the analysis side and the nuances specific to the tests and situations in the lab, but my foundational knowledge set me up for success in that role. Near-instantaneous feedback imprinted the concept of customer focus into me. It also beat the concept of customer obsession through near-instantaneous feedback into me.

Later in my career, when I switched to a TPM role, I started on a team that specialized in updating services to be compliant with import and export laws. I knew nothing about the compliance domain but spent two years with the team learning the services, as well as the intricacies of import and export legislation worldwide. When I switched companies, I was able to pick up the business domain of my new team – tax law – very quickly due to the legislation knowledge I had built up previously. Just as in the geochemical lab, where I used my geology background to help my application development, the tax and legislation knowledge helped me contribute more quickly and more effectively to the tax projects. Finding gaps in the requirements or requirements that needed clarification was easier, as I understood the context of the requirements to start with – I talked with developers about what changes were needed, and how they would allow them to work faster and with more confidence that they were building the right solution.

All of this is to say that a proper foundational knowledge, or toolset, can accelerate your productivity and general efficiency at your job. Could I have written those applications without a geology degree? Yes. Was I faster, more focused, and able to provide a better customer experience because of my background knowledge? Absolutely.

These tools enhance your existing PM skillset and allow you to apply situationally specific intentions to make all aspects of execution smoother and quicker.

Summary

In this chapter, we learned about the three pillars of a TPM: project management, program management, and a solid technical foundation. We learned about project management as a common ground for all program and project managers and touched on the aspects of project management where the PM adds the most value.

We learned about program management being an expansion of the project management foundation and how the tools and phases involved are the same as at the project level but with a larger scope and more complex communication. We discussed how communication is key to all of these exercises.

Lastly, we talked about what specialization can do and how a specific focus can amplify your existing program management skills to deliver with greater ease and proficiency.

In *Chapter 3*, we'll continue to explore program management in more depth. We'll talk about the specific methodologies that can be used in the various program and project phases. We'll also look closer at the intersection of project and program management.

Part 2: Fundamentals of Program Management

A large part of the TPM role is program and project management. However, the tech industry does not fit the mold for a standard **Project Management Professional** (**PMP**) style approach. It's dynamic, fast-paced, and views the project management triangle in a different light than is seen in other industries. This part will explore the approach I see used the most in the industry. We'll end this part by exploring the various career paths that are available for a TPM.

This section contains the following chapters:

- *Chapter 3, Introduction to Program Management*
- *Chapter 4, Driving toward Clarity*
- *Chapter 5, Plan Management*
- *Chapter 6, Risk Management*
- *Chapter 7, Stakeholder Management*
- *Chapter 8, Managing a Program*
- *Chapter 9, Career Paths*

3

Introduction to Program Management

In this chapter, we'll take what was introduced in *Chapter 1* and *Chapter 2* and explore it in more depth. As the book progresses, we'll go deeper into each of the pillars and the traits that define them. To ensure we are not spending time on context building with new examples each time we dive a bit deeper, I'm going to set up the context of this chapter with the introduction of the Mercury program. In this chapter, I'll define the goals of the program along with the projects that comprise it as a good analog to starting a brand-new program and related projects. Going forward in each chapter, I'll use this program to examine each area or concept, in detail, as it makes sense in the book.

In this chapter, we'll explore program management through the following topics:

- Introducing the Mercury program
- Examining the program-project intersection
- Exploring the management areas

Introducing the Mercury program

Back in 2009, in my days as a software developer, I was working at a company that didn't utilize an internal messaging system. Phones weren't smart yet, but messaging clients were abundant. However, they all required a central server to be configured in order to manage the messaging network. This wasn't something that I had access to do, nor was it something the company was willing to supply, so I thought about the idea of a distributed, or **peer-to-peer** (**P2P**), messaging system. So, I set out to write one myself, and some of us on the software team used the system for a year or so. It was a fun and distracting side project that we can build upon here as a thought experiment. I called the application *Mercury*, which is the Roman name for the messenger to the gods.

Let's take a look at a typical P2P network diagram in *Figure 3.1* and explore the benefits of a P2P network over a centralized network:

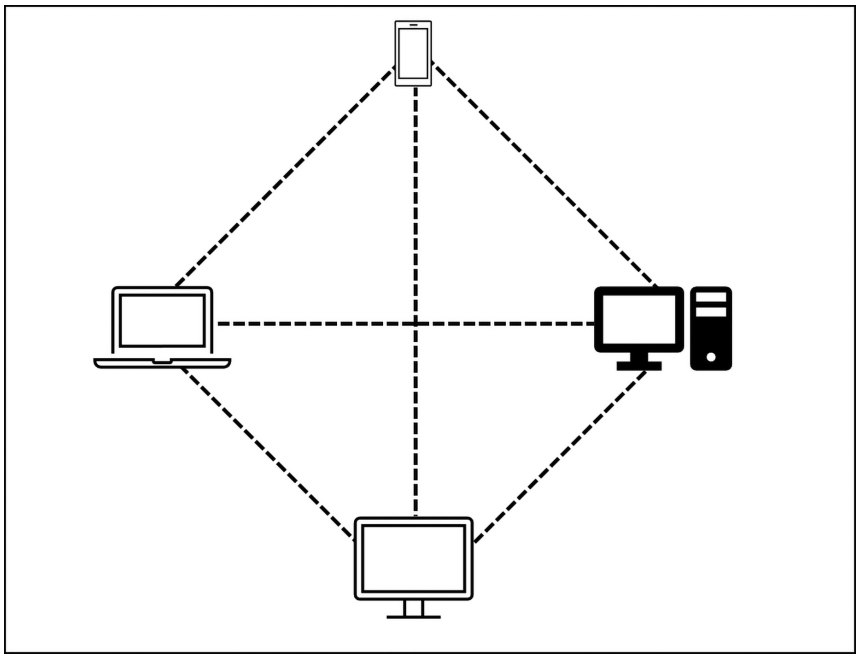


Figure 3.1 – A simple P2P network diagram

In a P2P network, the devices on the network talk directly with one another. This is opposed to talking via a central server. The central server can allow for the storing of network and session data, so only a single copy is needed. However, it means that without the central server, the network can't function. In a P2P network, each device on the network maintains its own list of connected devices and can relay information if needed. In this model, if one device goes offline, the network can still function, though there can be an increase in overall discovery traffic for activities such as network discovery. Additionally, the P2P network has the added advantage of no central overhead of maintaining a server for the application.

Mercury program scope

A program must have at least one achievable goal and, usually, consists of more than one project to realize them. Although I only built a simple application for Windows, that won't suffice for this book. So, let's suppose that we wanted it to be available in more places than just the Windows desktop.

The *program goal* will be to create a P2P messaging application and have a 90% user reach. As this is the goal of the program, the program will remain open until this goal has been fully achieved.

Mercury project structure

The number of projects in a program and the program roadmap are defined from the goals of the program. The goal was written in a way that has room for analysis and redirection. It states only that the application should have a 90% user reach, but not how to achieve that. This is where the project structure comes into play and how the projects can build upon each other to achieve the goal. For instance, 90% of the user base could, arguably, be achieved in a few different ways. You could target mobile-only **Operating Systems** (**OSs**) and focus on Android and iOS, as most people that have a computer also have a phone, and those **OSs** dominate the market. The user base is also not completely defined here, so shifting from a consumer user base to an enterprise user base could mean that only Windows and macOS are required to achieve that goal. Lastly, you could go all out and utilize all major systems: Windows, macOS, iOS, Android, and Linux. Unlike in 2009, when I wrote the Windows application, there are several cross-platform development environments that make sharing code easier. Work done for one platform can be reused in the others with less effort than having to rebuild from scratch. These ideas will be explored more in *Chapters 4* and *5*.

A program goal can be more specific if the needs are already well researched, but this sort of goal can also help you start to move forward while working through some of the unknowns. In a real-world example, I was starting a program that impacted the customer experience as well as adding new data to our backend models. During the program initiation and early high-level designs, we realized that the customer-facing changes were ambiguous and would take a lot of iteration to get right. The goal was written in a way that didn't make it clear that the customer-facing interfaces would need drastic updates, which caused stakeholders to stop and focus on the goal wording. We were faced with either pausing the execution of the program to fully understand this impact or moving forward with this risk to the goal. To not lose the momentum we had, and with the active engagement of all stakeholder teams, we refined the program goal to state that it would include relevant customer experience updates and narrowed the project goal for the year to reference only the backend changes. This allowed us to move forward with diving deeper into the right changes we needed to make, while simultaneously working on the well-understood backend changes. Put succinctly, *the goal is a guiding principle to ensure the direction of each project is aligned with the purpose of the program.*

However, for our case study, given the high number of opportunities for shared code, which reduces investment, and the user base impact of doing all major OSs, we'll explore a project breakdown that delivers across all of the OSs. For now, we'll draw the boundaries for the projects at the OS boundary, as shown in *Table 3.1*:

Project/program name	Description	Goal	How does it help the program?
Mercury program	Program for P2P messaging app	Build and deploy a P2P messaging app to 90% of the user base.	N/A
Android rollout project	Project for the P2P Android app	Deploy the app for the Android ecosystem.	Android represents 72% of the worldwide market share on mobile.
iOS rollout project	Project for the P2P iOS app	Deploy the app for the iOS ecosystem.	iOS represents 26% of the worldwide market share on mobile.
Linux rollout project	Project for the P2P Linux app	Deploy the app for the Linux ecosystem.	Linux represents 2.5% of the worldwide desktop market share.
macOS rollout project	Project for the P2P macOS app	Deploy the app for the macOS ecosystem.	macOS represents 15% of the worldwide desktop market share.
Windows rollout project	Project for the P2P Windows app	Deploy the app for the Windows ecosystem.	Windows represents 76% of the worldwide desktop market share.

Table 3.1 – The Mercury project structure

The preceding table lists the programs and program goals for easy reference when discussing the project goals. Each project has its own goal – to deploy the P2P app to a given ecosystem – while also contributing to the overall program goal of reaching 90% of the user base.

To ensure that the program is tackling the right OSs to reach 90% of the user base, I included market share statistics from *statcounter GlobalStats*, with a data range from June 2021 to June 2022. The numbers I use are rounded for readability and don't add up to 100% due to low-use OSs adding up to statistically relevant amounts. For more information, please visit `https://gs.statcounter.com/`.

Now that we have had an in-depth look at a use case program and the constituent projects, we'll continue with the introduction to program management by taking a look at the program-project intersection. We'll use this case study as we continue to explore the depths of being a TPM as a means to dive deeper into each area.

Examining the program-project intersection

In some companies, the TPM role is specifically listed as a project manager, not a program manager. For the majority of cases where it does refer to programs, the TPM still manages projects, too. This can lead to ambiguity as to the difference – or if there even is one.

In *Chapter 2*, we briefly discussed the relationship between a program and a project. A program is made up of projects and works toward the goals of the program while having its own goals.

Figure 3.2 illustrates the specific relationship between the Mercury program and the projects within it:

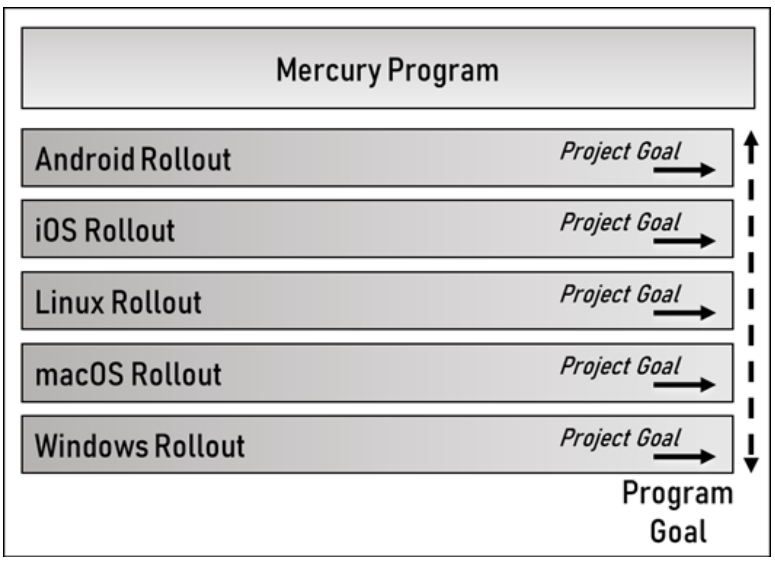


Figure 3.2 – The Mercury program roadmap

The Mercury program's goal is related to user reach, and the projects are broken down by OS. This figure shows how the projects fit into the roadmap overall. Since we chose to break the program down by OS, there are currently no inter-dependencies, so the relationship is straightforward. As we explore driving toward clarity, in *Chapter 4*, and project planning, in *Chapter 5*, the program roadmap will change.

In the Mercury program example, each project's goal directly contributes to the program goal, as any increase in user reach by an ecosystem deployment will increase the overall user reach and get the program closer to the goal of 90% reach.

Program management for Mercury will start and end with these five projects, as they will achieve the stated goal of the program. As such, the program status will discuss the project goals in cases where they impact the delivery dates. It will also surface any major roadblocks that each project encounters.

As the projects don't have any interdependencies, each project will only report on the work relevant to that project. They'll go into deeper details on development blockers, delays, and dependencies within the project.

With the way that the program and projects are currently laid out, a single status report might also suffice given how tightly related all projects are to the end goal of the program. At the end of the day, how you manage the program versus projects depends on many factors and can change from program to program.

We've introduced a case study and explored how the program and projects intersect with this example. We'll continue to build on this program plan and dive into how it is set up throughout the book. For now, we'll touch on the key management areas and use the Mercury program to explore a bit deeper.

Exploring the management areas

In *Chapter 1*, I introduced the key management areas: plan, risk, and stakeholder management. Using the Mercury program as an example, we'll examine each of these in a bit more depth and continue to build upon our use case study.

Project plan

Let's examine the Windows rollout project plan in a bit more detail. At this point in the book, we'll assume a few things:

- We have as much resourcing as we need (so, everything that can technically start at the beginning of the project will be scheduled to start)
- No resource timing constraints exist
- Estimates are uniform across all platforms
- All predecessors are considered **finish-to-start** (meaning the predecessor must be finished before the task can start)
- There are no inter-project dependencies
- In other words, a *textbook* program example!

Given these assumptions, *Table 3.2* lists a simplified project plan for the *Windows project*:

ID	Milestone	Predecessor	Effort (weeks)
1	P2P Subsystem Ready		8
2	User Interface Ready	1	16
3	**End-to-End (E2E)**	2	4

Table 3.2 – A simplified view of the Windows project plan

In this example, there are a few simple dependencies shown: the user interface milestone requires that the subsystem is completed, and testing requires both preceding milestones to be complete. Now, this is extremely simplified and many seasoned TPMs might be screaming right now at this optimistic project timeline! Don't worry! As we explore each management area in greater detail, the project and program timelines will be expanded on to highlight those management areas. In the end, we'll have a more complete plan that demonstrates the strengths the TPM brings to program management.

In *Table 3.3*, we'll expand on the Windows project plan, apply the same milestones to each OS, and include some tasks for inter-OS testing:

ID	Project / Program	Milestone	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug
1	Windows	P2P Subsystem Ready								
2	Windows	UI Ready								
3	Windows	E2E Testing Complete								
4	macOS	P2P Subsystem Ready								
5	macOS	UI Ready								
6	macOS	E2E Testing Complete								
7	Linux	P2P Subsystem Ready								
8	Linux	UI Ready								
9	Linux	E2E Testing Complete								
10	Android	P2P Subsystem Ready								
11	Android	UI Ready								
12	Android	E2E Testing Complete								
13	iOS	P2P Subsystem Ready								
14	iOS	UI Ready								
15	iOS	E2E Testing Complete								
16	Mercury	Windows to macOS Integration Testing								
17	Mercury	Windows to Anroid Integration Testing								
18	Mercury	Windows to Linux Integration Testing								
19	Mercury	Windows to iOS Integration Testing								
20	Mercury	macOS to Linux Integration Testing								
21	Mercury	macOS to Android Integration Testing								
22	Mercury	macOS to iOS Integration Testing								
23	Mercury	Linux to Android Integration Testing								
24	Mercury	Linux to iOS Integration Testing								
25	Mercury	Android to iOS Integration Testing								

Table 3.3 – The program plan with the Gantt chart

In the Gantt chart, a month is represented by a single cell, and we assume that 4 weeks is equal to 1 month. Given our initial assumptions of uniform task efforts and unlimited resourcing, we can see that all of the projects (Windows, macOS, Linux, Android, and iOS) start at the same time in January. The inter-OS tests are categorized under the Mercury program as inter-OS testing falls outside the bounds of the individual projects but is required for the Mercury program goals. This could also be a sub-project depending on the organizational structure of your company.

Project and program risks

At this point, we have a full, albeit simple, program plan including plans for all projects. We'll continue with our current assumptions for consistency while talking about risks. When we go through the exercise of risk analysis, we have a few different inputs to help us identify and categorize them:

- Program and project plans
- Risk register
- Stakeholder input

Let's take a look at each of these in more detail.

Program and project plans

Filling out the program and project plans can help us visualize constraint risks. Taking this program plan as an example, we can see that a few risks stand out, even with our infinite resourcing assumption. Let's look at a closeup of the risks in *Table 3.4*:

ID	Project / Program	Milestone	Jul	Aug
16	Mercury	Windows to macOS Integration Testing		
17	Mercury	Windows to Anroid Integration Testing		
18	Mercury	Windows to Linux Integration Testing		
19	Mercury	Windows to iOS Integration Testing		
20	Mercury	macOS to Linux Integration Testing		
21	Mercury	macOS to Android Integration Testing		
22	Mercury	macOS to iOS Integration Testing		
23	Mercury	Linux to Android Integration Testing		
24	Mercury	Linux to iOS Integration Testing		
25	Mercury	Android to iOS Integration Testing		

Table 3.4 – A closeup of the Gantt chart, focusing on the integration testing

The first risk that stands out to me with our given assumptions is the 10 overlapping integration tests between the platforms. All 10 are running over the same two weeks. Our assumptions are that we have the resources to do this, so that isn't the risk here. The risk is that this parallel schedule assumes all testing is perfect and that nothing can go wrong. Every system is testing against every other system, and if even one of these integrations were to find a bug, let's say Windows to Linux, it means that every other Windows integration and Linux integration might potentially be blocked by the same issue. This can lead to significant delays.

Risk register

The risk register is a central storage for all risks that have been identified in previous projects completed in a company or organization. Often, they are categorized in some fashion so that they are searchable based on similar project traits. This could include working with a specific system or team, dealing with the same clients, having the same level of complexity, and more. Honestly, I haven't found many instances where this risk register is a true database, but that is the perfect scenario.

Instead of a central repository, the risk register often relies on **tribal knowledge**. This refers to the implicit – known but not easily conveyed – and explicit – able to be easily articulated – knowledge relevant to the organization that is not written down. In this case, you review a plan with your peers to seek out the known risks they can identify and also draw from your own experiences. Though if you do find yourself in a position to create an actual central repository, do it!

For the sake of demonstration, *Table 3.5* shows an example of a risk register. This company – which is presumably running the Mercury program – deals with many other cross-platform business systems:

ID	**Project/program**	**Risk**	**Strategy**
1	Windows To-Do App	Network testing failures	Acceptance – 3-week milestone slip
2	Linux To-Do App	Distribution regressions	Mitigation
3	Android Project Tracker	App approval delays	Acceptance – 2-week addition
4	iOS Project Tracker	App approval delays	Acceptance – 3-week addition

Table 3.5 – A risk register for the Mercury program company

As you can see, there are a number of risks that can happen in one project and translate to the next. The network testing issue for Windows might be present and need to be accounted for in the testing timeline. Based on the root cause, you might be able to mitigate the issue ahead of time, or just need to ensure you have extra time allotted for testing over the network. The app approvals in the Android and iOS stores are common and are likely to happen each time. Again, knowing the root cause could help mitigate this, though the approval steps are constantly changing, so a more prudent approach might be to pad publishing timelines to account for the inevitable back and forth.

Stakeholder input

This is similar to the tribal knowledge aspect of the risk register discussion. Often, stakeholders can offer a different perspective on a problem or perceived risk. They might even have a risk register to share based on similar projects that they have been involved in. This also ties back to communication (which will be a recurring theme in this book) and just how important it is to communicate early and often. This can lead to critical information that can prevent a risk from being realized and save you valuable time.

We've talked through the project plan and the risk plan, and we have even covered how they overlap, as demonstrated in this example of stakeholders contributing to the risk plan. Let's discuss the stakeholder plan itself in more detail.

Stakeholder plan

The stakeholder plan, which includes the communication plan, is another essential area of program management. It can make or break your project and your career. You can be excellent at delivering a project on time, but if your stakeholders are kept in the dark even where there aren't issues, it will come across poorly. People want to know what is going on.

Aside from just letting your stakeholders know about the status, knowing how to tell them is just as important. Your status report to a development team should read very differently than a status to a senior leader. They have different goals and perspectives regarding the project.

For the Mercury program, there will be several different statuses and, thus, a layered stakeholder engagement and communication plan. *Table 3.6* shows the different status reports across the Mercury program.

ID	Project / Program	Stakeholder	Development Report	Monthly Review	Quarterly Review
1	macOS	TPM Manager	Yes	Yes	Yes
2	Windows	Dev. Team Manager	Yes	Yes	No
3	Linux	Lead Engineer	Yes	No	No
4	Android	Android Division Lead	No	Yes	Yes
5	iOS	Director, Mobile Systems	No	Yes	Yes
6	Mercury	VP, Productivity	No	No	Yes

Table 3.6 – The stakeholder plan for the Mercury program

This table describes three distinct types of reports: development report, monthly report, and quarterly report. Not every company does this many types of reports, and not all projects or programs will warrant sending this many out. This is informational to get an idea of the types of reports you might encounter or wish to use yourself. Also, some companies don't do written reports at all – but instead, rely on review meetings. However, the same principles should apply.

The development report is a weekly or bi-weekly report on the development efforts. This will include how the development cycle is going, including things such as a burndown or progress chart and velocity, along with any major blockers the team or teams are facing. Using the satellite analogy from *Chapter 2*, this would be the street view, as in seeing the buildings right in front of you.

The monthly review, which is, sometimes, called a monthly business review, is directed toward stakeholders that are directly involved in the project or program (as in, they have resourcing involved), and upper management teams. The content is focused on the direction of the project or program including major milestones and their statuses. This will include any major issues or blockers that are directly impacting a major milestone but not the lower-level details such as standup and minor development blockers. This is equivalent to the mid-level satellite views – you can see the neighborhoods of the city but can no longer make out exact details on the buildings.

Lastly, the quarterly review, also known as the quarterly business review, is directed toward senior leadership and executive sponsors for the project or program. Here, the content closely matches that of the monthly review but with a quarterly cycle. Some details may be taken out if deemed too granular, as the concern at this level is about the overall health and the end goal's trajectory. The view here is at the city level, using our satellite analogy.

Chapter 7 will go into more depth on how you can decide which of these communication cycles to utilize, as well as take a deeper look into them.

Summary

In this chapter, we introduced a case study, the Mercury program, which will build a P2P messaging application across multiple ecosystems. We then used the case study to explore a simple program and project intersection discussing various ways to manage status updates. Lastly, we dug deeper into key management areas and refined the definition and roadmap of the Mercury program as a result.

In *Chapter 4*, we'll explore one of the most important characteristics of a successful TPM, *driving toward clarity*. This ability of a TPM to take an ambiguous situation and find or define clarity is what helps us solve problems, find opportunities to reduce risks, and deliver results.

4
Driving Toward Clarity

In this chapter, I'll introduce the last foundational skill – driving toward clarity. It is not a single step in the management process but is instead a skill that is applied to every process we drive and every decision we make. It is the reason why you will hear the TPM in the room ask *why* and *how*. It's why you see TPMs run meeting after meeting, working session after working session, and attending every standup that they can fit into their schedule. They are constantly taking the data they have and clarifying it so that it has no ambiguity left and is easier to understand and follow. This then feeds into getting consensus because, without clarity, it is hard to get past the first stage of getting everyone onto the same page, and we can't move on to driving consensus.

As this skill permeates everything we do as leaders, it is not a pillar but the mortar that transforms a pile of bricks into a wall. It is an integral part of who we are and the reason this topic gets its own chapter!

We'll learn how to drive toward clarity by exploring the following topics:

- Clarifying thoughts into plans
- Using clarity in key management areas
- Finding clarity in the technical landscape

Let's get started right away!

Clarifying thoughts into plans

Everyone drives toward clarity in some aspect of their life. It is the act of taking knowledge and applying it to a situation to make the path forward better understood, and to lead to the desired outcome.

As an example, a retail worker may look at a product layout diagram, along with the products they have on hand, and the actual shelving they have available, and do their best to match the diagram – all along with making minor adjustments or judgment calls. They are driving toward the end goal of a properly stocked display. In the same situation, the worker may just be told to *make a display out of the seasonal summer products*. In this case, the level of ambiguity is much higher than the first – there

is no diagram and no specific list of products, so the complexity and number of decisions are much higher. In both cases, the worker drives toward clarity to end up with a properly stocked display.

This is also true for a TPM and the roles that surround it. What makes a TPM shine is that they drive clarity in every aspect of their job. We question *everything* and *everyone* to ensure that the objective is as clear as it can be. In this respect, a TPM must be willing to question requirements from upper management just as deftly as from a leader on a peer team.

The most common area where this comes into play is in the problem statement, as in what is it that we are trying to solve? It is such a large part of our job that many role guidelines specifically mention the concept with varying degrees of complexity. *Figure 4.1* shows a common breakdown of the level of ambiguity a problem statement will have for a TPM based on their career level:

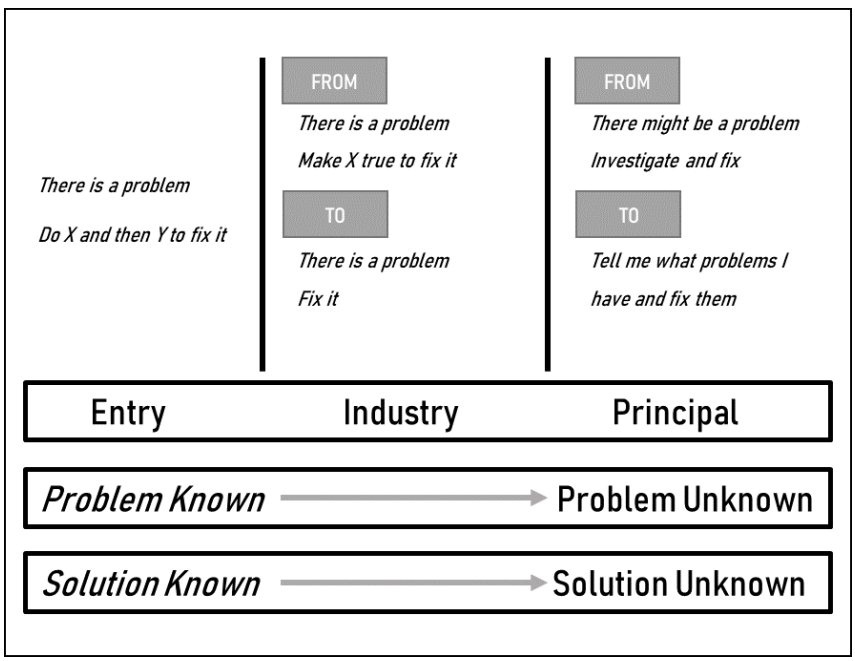


Figure 4.1 – Growth by ambiguity

In this figure, you can see that the entry-level position's problem statement and the solution are clearly defined – much like the first retail example. Both the problem and solution are laid out. There will be some minor ambiguities during execution you'll need to resolve, but it will be fairly straightforward. Moving on to the industry TPM, there's a little less direction. You might be told the general solution but not how to accomplish it. A senior TPM will simply be told to solve a problem but not how. Up to this point, the problem statement is known but the amount of ambiguity in how to solve the problem is what increases, the higher up in your career you go.

Beyond these levels, we get into high-impact positions where the ambiguity increases significantly. This can vary depending on your experience and the company but generally lands between being told that a problem may exist and to investigate and fix it, and to go *find* problems and fix them. In the first example, the problem statement is very ambiguous but, in the latter, the problem statement isn't ambiguous, it's non-existent! We'll touch on this again in *Chapter 9* and expand it as this directly correlates to scope as well as impact. *Figure 4.1* used elements of *Technical Program Managers by Ambiguity* from *The TPM Blog*. Visit `https://www.mariogerard.com/take-it-to-the-next-level-as-a-tpm/` for more information.

We now understand the basics of clarifying thoughts into plans and the impact this skill has on our success in a project as well as in our career (we'll cover this further in *Chapter 9* when we discuss career paths). Next, we'll see how this skill is utilized throughout the project.

Using clarity in key management areas

There are many ways in which driving toward clarity has an impact on project or program execution. We'll discuss the key management areas to demonstrate how seeking clarity is utilized.

Planning

Let's take the original problem statement that was the basis of our case study as an example for driving thought into plans. In the original iteration of the application on Windows, I had a problem I was trying to solve – I wanted to talk with my colleagues via a chat client but didn't have the resources or permission for a centralized server. I took the problem statement and looked for solutions that fit the need for no central server and landed on a P2P network. From this initial solution, I then dove into high-level designs, low-level designs, and then implementation. In every step here, there was some level of ambiguity that needed to be clarified to proceed. I didn't have stakeholders to convince, nor a development team, but the need for clarity was still there as I knew what I wanted – requirements – but not how to accomplish them. What was different then versus now was the scope and impact of ambiguity. In the beginning, the scope was limited to my colleagues on Windows machines, and there was no impact as it was simply a desired feature to have at the office. Therefore, the ambiguity was limited to how to accomplish the requirements. In the **Mercury** program that I'm using in the book, the ambiguity stretches across both the requirements and solution, and there are stakeholders involved who are impacted by the work.

Now let's expand this to the Mercury program that stands as the use case for this book. The problem statement is well understood, and the requirements are known at a high level: make a client for each operating system (based on our assumptions from *Chapter 3*). Many open questions differ from platform to platform. For instance, mobile platforms need more clarity on the certification process for their respective app stores to form a project plan and a high-level design. Similarly, all platforms need details on how to initiate a broadcast call over TCP/IP in order to finish high-level designs. This also helps determine the level of generalization the network communication layer component can be designed to work with across the platforms.

As the high-level designs for the broadcast call come into focus, further changes to the program plan, and each project plan, may be needed. To add to the complexity, each application will be developed by a team in the company that specializes in their respective operating system and may have additional requirements unique to their operating system, or the frameworks they use for development.

This iteration of the high-level design demonstrates how the planning cycles for the program and project can influence each other, and how driving clarity can feed right back into additional planning. Let's take a closer look at the cyclical nature between planning and seeking clarity in *Figure 4.2*:

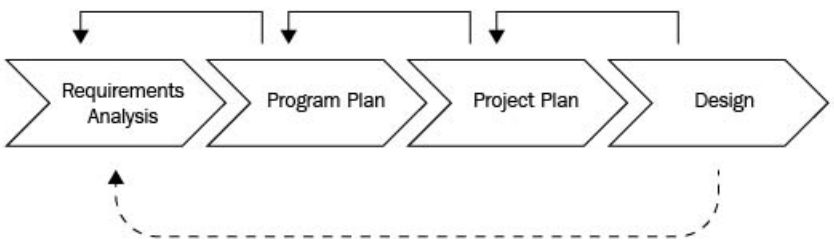


Figure 4.2 – Driving clarity in plan management

The preceding figure shows the interrelation between program and project planning, and how driving clarity plays a role in every step. As you work through the requirements for the program, you seek clarification on specifics in order to understand the impact and scope better. These clarified requirements then feed into your program plan, which, in turn, sets up the basis for each project plan. The program requirements that are covered by a specific project will make their way to the project requirements with traceability back to the program. Then, during the execution of both the program and project, as designs are finalized, updates to estimates and timelines are then fed back into the respective plans. The designs are also checked back against the requirements to ensure that the designs satisfy the requirement. Each of these cyclical paths is driving toward clarity to some degree.

What can happen when you don't clarify your plan enough?

We've talked a lot about the good things that can happen when your management practices are solid. However, it's worth talking about what can go wrong when clarity isn't sought out, or the level of clarity is insufficient.

The plan and the planning process are important for a successful project or program. Due to this, any lack of clarity in the requirements or elements of the plan can lead to long-lasting issues. The next three chapters will explore this concept in more depth, but let's walk through a few examples now.

Scope creep, or the addition of scope unexpectedly, can arise in a few different scenarios. We think of it mostly as additional requests that are added to the project after the scope is locked in, and this certainly does happen. However, it also occurs when requirements are not well understood. As requirements are fleshed out, they uncover complexities not seen at first, which will lead to additional tasks to meet

the requirements. This is due to a lack of driving sufficient clarity in the requirements, resulting in poor estimations and understanding. Conversely, scope creep can happen through **gold plating**, or the addition of features or enhancements not asked for in the requirements. This can happen during design, where the developer adds things to the design that they believe are needed, or feel are needed, and it can also happen at the request of the TPM.

Lastly, there are derived requirements where the TPM gives the requirements or functional specifications to the development team, who then adds additional requirements based on the needs of their service. This could be things such as scaling, latency, or code refactoring that are needed to make the required changes maintainable. As a TPM, the more you know about the roadmaps, architectural evolutions, and availability needs of the services you work with, the more you can anticipate these needs earlier on, so they are not added during development but added into the scope at the beginning.

All of these types of scope creep are different from iterative development in that the requirements are understood, but the priorities may shift around. Even in the cases where scope is added due to customer feedback, it merely shifts the priority of other items and isn't true scope creep, where scope is added but there is no additional time or resources to handle it.

Change management, which is the process used to handle a change in scope, resourcing, or timelines, is negatively impacted by scope creep, as it causes constant churn on the project plan. This then leads to delayed delivery of a milestone or the delay of the entire project. Though the nature of the project management triangle is that these elements are always tied to one another, a well-defined plan will reduce the friction between them.

In both of these examples, ambiguity in the requirements forces the late discovery of missed scoping during design and development. As such, ensuring that the project requirements are well understood and written in detail can mitigate these problems.

We've seen how driving toward clarity in project planning leads to a successful foundation for a project's execution, as well as some areas where a lack of clarity can increase the project scope or timeline. Next, let's see how the risk assessment process utilizes this same skill.

Risk assessment

For risk assessment, there are many ways to drive clarity. First and foremost is doing the exercise in the first place! I've seen plenty of projects that treat issues and risks synonymously – as in, an active issue is just a risk to the project, and that risks are an issue that causes timeline issues. However, a risk is something that might happen and impact the project timeline, scope, or resourcing, but hasn't happened yet. In a *textbook world, meaning in theory*, an issue is just a risk that has been realized – as in a risk that was already captured, and there is a strategy in place to deal with it. By knowing ahead of when something might happen, you have a better chance of avoiding it or at the very least being prepared to handle it.

We aren't in a textbook world though, and in the real world, many unforeseen problems can crop up (which should go into your risk register to reduce the chances of the same problem coming up again

in future projects). The risk register for your project should be abundantly full to account for any issues that you can foresee. Though issues that arise may not match the risk perfectly, the stated risk and strategy are usually adaptable to the *real-world* scenario that takes place.

As an example of a foreseeable risk that may or may not occur, I have worked with a team that has had trouble with consistent service deployments. There are multiple reasons for the inconsistency, from lengthy deployment timelines to consistent merge and regression issues. Owing to this, deployment delays are a standard risk for my projects when there is work on these services that have delays. Once I have my project plan and one of these services is identified in a task, I immediately add the risk to the project. The service could have a great month and not miss a single deployment, but that doesn't mean the risk is gone. This deployment risk isn't meant as a slight to the team that owns the service but is just a statement of fact and if I ignore that fact, I put the project in jeopardy.

When driving clarity in analyzing risks, it forms a cyclical relationship between risk assessment and planning. *Figure 4.3* shows this relationship:

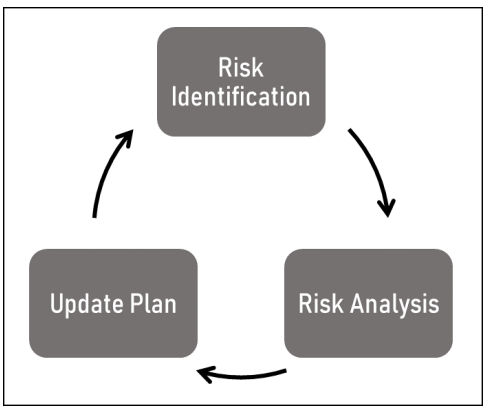


Figure 4.3 – Driving clarity in risk management

From our case study, we've already identified several risks in *Chapter 3*. The first pass at risk assessment is relatively surface-level; you find the risks that are obvious and capture them. These risks may help uncover more risks, which in turn find more and so on. Every pass adds clarity to the project by adding more detail to the project plan and risk strategies.

What can happen when you don't clarify your risks?

Risk assessment follows many of the perils of the project plan. As seen in *Chapter 1*, it feeds into the plan itself and the plan, in turn, feeds into the risk assessment. A misunderstood risk can create a false sense of security, by providing a risk strategy that may not fully address the problem.

For instance, a risk may be identified in the *Windows Rollout project* of the *Mercury program*, which states that there is a possibility of a network test failure that could add 3 weeks to the timeline (refer to *Table 3.5* in *Chapter 3*). However, if the type of network test failure isn't specified, the strategy of delaying by three weeks may not be accurate. It could also be that other risks, such as the **fast-tracking** – or running in parallel – of all end-to-end testing, could exacerbate the networking failures, and thereby cause even more delays than expected. The deeper the analysis and cross-checking of risks and projects, the better the outcome will be.

Risk assessment benefits directly from seeking clarity but also influences your project plan in return. Now let's examine the relationship between stakeholders and driving toward clarity.

Stakeholders and communication

At a first glance, working with stakeholders may not seem to have much need for driving clarity, but this is just as important as the more obvious methodologies we've already covered. We think of stakeholders as people or groups we need to communicate statuses to, discuss timelines and blockers with, and so on, and while this is true, they are also great sources of information throughout the project.

In order to get the most out of a stakeholder, you need to fully understand their role in the project or program. Are they merely an interested party, such as a VP of a department, or are they involved in the deliverables directly? If they are involved, what is their role in that involvement? A developer? A manager? A fellow TPM? These questions help determine who your subject matter experts are. A TPM is adept at seeing a problem at multiple levels and can connect a problem that one milestone is having with surrounding work or even other projects within their department.

Take the *Windows Rollout project* as an example. When getting ready for execution, the networking risk we discussed in *Chapter 3* may come up in the design. The list of stakeholders may reveal a subject matter expert in networking from within the *Windows Rollout project*, or even across the other projects within the *Mercury program*. If, for instance, the *Android Rollout project* has a dedicated networking team, they may be of some assistance given that three out of the four **TCP/IP** layers (**Transport, Internet, and Network** layers) are going to be largely common across different application layers, or in our case, operating systems. This could be a chance to combine network specialists across the projects, forming a co-op group to help solve common network issues and ensure consistent network solutions for the program. Without knowing who your stakeholders are and what their roles are, this type of opportunity to clarify networking requirements and put in place mechanisms to ease the design and implementation may be missed or take a lot longer to piece together than is necessary.

There are two avenues where you can drive clarity by communicating with your stakeholders. *Figure 4.4* shows the relationship between driving clarity and these avenues of communication:

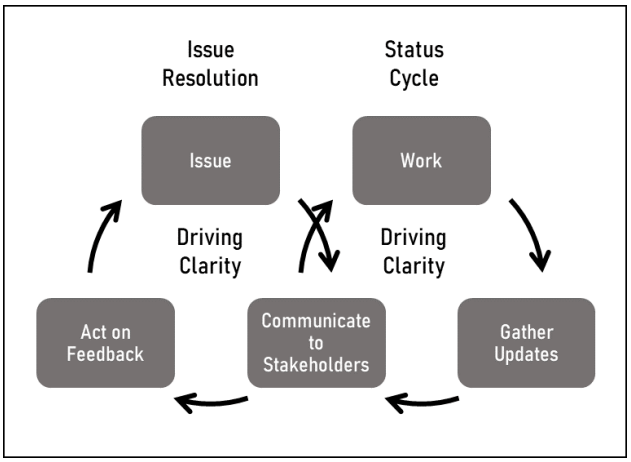


Figure 4.4 – Driving clarity using stakeholder communication

Issue resolution and status updates are the main avenues of utilizing stakeholder communication to drive clarity. In issue resolution, you are communicating in order to get additional input to help solve the issue. This is important as each stakeholder will have a different perspective on the issue and may be able to provide useful information to drive a solution.

Using your existing status cycle to communicate back to stakeholders keeps them up to date as to what is going on in the project or program, and who is working on what. This can help stakeholder teams prepare for milestone deliveries and reduce back-and-forth churn. In the next section, we'll explore how the lack of these cycles can cause problems.

Risks of excluding stakeholders in clarification strategies

Communication is key to our success. So is our ability to connect the problem with the right people to solve it, and to find out how problems may intersect with one another.

I have found over the years that keeping all stakeholders up to date with all open issues is advantageous for a few reasons. Firstly, it provides the highest level of transparency to everyone involved, thereby building trust. The more the stakeholders know what issues are coming up and how they are being addressed, the higher confidence they'll have in the project. Secondly, they'll have information to help resolve open issues. This leads to two main examples in which a lack of communication with stakeholders can cause even more problems.

When it comes to earning trust, it often comes down to the perception that stakeholders have of your ability to deliver on the goals. Your best avenue to control that perception is through status communications. Work with your stakeholders to draft the right message based on your audience. I have had long meetings discussing how a status report needed to show all of the issues, to ensure that the highest-priority issues were clear, and to ensure that the project didn't come across as in worse shape due to the number of issues.

Aside from the content, it also needs to be consistent. If your communication is promised to come out weekly, and you miss a status or are a day late here and a few days there, this leads to the perception that you are disorganized and have a hard time following through. Even if you have a legitimate reason for missing or delaying – such as waiting on a resolution to an issue to report the most up-to-date status – there's a good chance it won't be seen that way. This is so fundamental to what we do as TPMs that consistent communication can come up in performance reviews and promotion documents. It's a measurable artifact that demonstrates our ability to be an effective communicator – you can literally see a gap in a status, or that dates don't line up to when they were promised to. It can also lead to micromanaging from your stakeholders!

If stakeholders don't believe in your ability to manage the project, they may start to challenge your decisions. *Chapter 7* will go into more detail on the dos and don'ts of status reporting.

From a problem-solving perspective, not involving your stakeholders can lead to some negative impacts, all involving losing time. Let's examine two scenarios.

Avoiding wasted effort with clear communication

First up, I want to examine a use case of wasted effort. We'll use the *Mercury program* as the stage to discuss this. At a minimum, we have five projects going on in the program at once. Most of this work is contained within their respective operating system environments. However, there are examples of shared code from both the lower network layers, as well as a shared API library for the service itself. In these cases, each project team will be contributing to these shared libraries. As issues arise and problems are investigated, you may end up with multiple groups trying to solve the same problem, or simply working on the same code base at the same time. Without proper communication between teams, this can be missed. Worst-case scenario, you end up with code merges that remove, alter, or override behaviors in unexpected ways. Or you simply end up with time wasted by having two teams working on the same problem. Proper communication can ensure the overlap is noticed, the plan is updated, and the situation is avoided or corrected as quickly as possible.

A good analogy for this is *too many cooks in the kitchen*. If you've ever seen competitive cooking shows, you'll see multiple chefs in the same shared space and they're always yelling out something along the lines of *coming through, hot plate!* You've also seen what happens when they don't do that: plates collide and dishes are ruined. Communication can be the difference between winning and losing!

Reducing churn during issue resolution

The other consequence of poor communication when trying to find clarity is similar to the first example in that it results in wasting time. For instance, there's a large, mature product that recently involved sending a new set of specifications down via the technical writer. The tester and developer are both testing the software to see whether it fits the specifications. However, it turns out that there are three requirements that logically cannot coexist – any two of them can but not all three. The developer is doing integration testing and finds that one of the requirements breaks, so they fix it, which breaks the third requirement. Now the tester is running their tests, sees an issue, and reports a bug. This

is fixed but it breaks requirement two. This goes on and on, as the technical writer, developer, and tester are not communicating. A TPM gets involved due to the delay, gets everyone into a room, and within an hour sees that there is a problem with the requirements, and works with the technical writer to decide which of the three to move forward with. Had these stakeholders communicated about the issues they were facing, this may have come to light much quicker (or, simply had a TPM been involved from the start).

Often in these cases, the missed opportunities are not intentional – you are simply in the weeds trying to unblock your team – but to be an effective practitioner we need to always ensure stakeholders are in the know. It can save you a lot of time and unnecessary work.

As you can see, a lot can go wrong when stakeholders aren't involved in your quest to clarify. Conversely, a lot can go well when handled correctly. As a TPM, your job is to be a force multiplier; you get the right people in the right place so that a problem can be solved quickly and efficiently. And when you don't know the right people, or the right place, communicate with your stakeholders to find answers quicker!

We've explored how you are always driving toward clarity in every step of a project and looked at examples of how seeking clarity benefits the project and what happens when you don't dive deeper into a requirement or problem to define it. Now, let's look at how this may differ in a technical setting.

Finding clarity in the technical landscape

As a TPM, you are often aware of many programs and projects that impact your team or organization. You work with other TPMs, other teams, and leaders to deliver your goals. This gives you both a deep knowledge of where you directly run your projects, as well as breadth in the technical landscape surrounding you.

In a technical setting, we use this depth and breadth to our advantage as we have the ability to look at each problem from a different vantage point, all the way from the street level to the full cityscape. These varying vantage points allow us to connect the dots across projects and programs. Let's look at a few examples of seeking clarity and connecting the dots in a technical setting.

Let's explore a technical breadth example. As is often the case, members working on the implementation of a project are highly focused on the project goals. While this is desirable, as it ensures a solid solution to the problem at hand, they might miss how these changes may impact other initiatives. As a TPM, you might be in a position to help drive the technical direction of your team. This involves knowing where your team and technology are headed, as well as the tenets that drive them. This knowledge will help drive your day-to-day direction and your design decisions, by helping you connect problems and solutions across project and program boundaries.

In a recent example, my organization received a request from a client team to look into the behavior of a field we return in our service response. It was behaving incorrectly for an edge case, and they wanted us to fix it as they relied on that field. Our on-call developer was looking into this issue when

I came across the ticket. Though it is always a good idea to fix edge case issues in your API responses, for this particular field, the team had decided that, in the long term, they wanted to remove this field from our response. I engaged with the client that submitted the ticket and talked with them about what they were trying to solve. I took this back to our team to help ensure that our long-term vision was not missing a use case, while also providing a different way for the client to achieve their desired behavior without relying on that field. Our on-call developer didn't know or wasn't thinking about the long term, they were in triage and trying to solve issues, so this connection was initially missed.

By recognizing that fixing this field went against our long-term goal of removing the field entirely, I was able to steer the client away from using it. We found a solution that aligned with the long-term architecture and thereby removed the throw-away work they would have done in the next year to move away from the deprecated field. Inspecting broadly can help ensure you are influencing not just your project but your organization and driving everyone toward your shared goals. At the same time, we need to ensure we are diving deeply as well. Looking back at project planning, one key handoff is when a task is taken from the project and given to the development team to design and implement. This handoff is a key area to ensure you drive as much clarity as possible. You do this by providing all of the context needed to do the task upfront, and then by asking questions during design and estimation. Often, our project plan estimates are high-level in the beginning and as requirements go through design, we get a more confident estimate that we can then update our timelines with.

However, this isn't a static process as the capabilities and depth of understanding can vary from developer to developer. So, I always clarify with the developer their estimations starting out, by simply asking how confident they are in the estimate, as well as what elements were included in the estimate (design, code review, testing, and/or deployment), as many new developers will just include the code writing itself. I then follow up by probing the design to see what sort of responses I'm getting. This can help inform me of what a buffer might look like for the developer's estimates for the given task. In these scenarios, it's good to remember that a developer's job is to estimate based on ideal conditions. Though some developers can, it is your job as a TPM to look at how other factors, such as the following, may impact the developer's ability to deliver the task:

- Are there deployment issues or blockers on the calendar?
- What is the experience level of the developer for the area the task impacts?
- Are there any unknowns or new technologies for this task that may take extra time?
- Are there other projects ongoing that may impact the changes being made?

You will need to go deep into the project and understand how these tasks are connected, the technologies behind them, and the people involved in order to bring the highest clarity to the project plan and ensure the success of the developer in their task.

Summary

In this chapter, we discussed in greater detail how a TPM is constantly driving toward clarity. From project initiation, all the way through to project closing, the TPM is clarifying requirements, assumptions, issues, plans, and estimates to continually steer the project in the right direction.

We looked at how everyone takes thoughts and clarifies them into plans, from building seasonal displays at a retail store, to varying levels of ambiguity in the problem a TPM needs to solve. We also touched on how the level of ambiguity increases the further in your career you get.

We explored how driving clarity is used in all key management areas and tied it together with specific scenarios that you can encounter in a technology space.

In *Chapter 5*, we'll use the Mercury program and driving toward clarity to build out a more comprehensive project plan as we dive deeper into project and program planning.

5
Plan Management

In this chapter, we'll start delving deeper into the key management areas I discussed in *Chapter 1*, starting with plan management. We'll build on the *Mercury program* example to inspect all stages of creating a plan both at the project and program levels.

We'll dive into plan management by exploring the following:

- Driving clarity from requirements to planned work
- Planning resources in a technology landscape
- Exploring the differences between a project and a program

Let's start planning!

Driving clarity from requirements to planned work

In *Chapter 4*, we discussed the importance of driving clarity in all aspects of a project or program. Out of all of the areas where driving clarity is needed, plan management is the most important.

When writing a project plan, we might find ourselves asking why. *Why are we doing this?* I ask myself this every time I'm writing one, even though it is one of the aspects of project management I enjoy the most. The answer to why we do it is simple: *it enables us to be a force multiplier*. We know the work that needs to be done and so does everyone else. The team spends less time on determining *what* and *when* and instead focuses on how to achieve the tasks and goals. Driving clarity in requirements early on takes less time than correcting the issue later in execution as this can lead to bad estimations due to poor understanding, which results in longer timelines.

The first step in driving clarity in your requirements and beginning your project plan is choosing the right tool or making full use of the tools you have available. Let's explore some of the project management tools available.

Project management tools

The project plan is one of the most crucial artifacts for your project. This includes many aspects, which we'll cover in this chapter, such as the task list, resourcing, and predecessors that combine to produce a **Gantt chart** of the project timeline. This is a lot of work to do and there are several tools out there that can help you do it.

Through my interviews with industry leaders, I confirmed something I suspected was true all along – there is no standard tool in project management. It often comes down to personal preference combined with the constraints placed by your company, usually related to security concerns arising from your company data being stored on cloud servers.

Nonetheless, I do have a list of tools and feature comparisons to help drive home the variability in the marketplace that exists today. I can say, though, that the one and only tool that was used at every company that I interviewed for this book was **Microsoft Excel**.

I must confess that I also use Excel from time to time, for instance, for the plans in this book! Sometimes keeping it simple and in a well-known format is the best way to go. You lose a lot of functionality that you will find in dedicated project management tooling, but you are also free from their preconceived notions of what project management must be.

Taking a step up in functionality, **Microsoft Project** is ubiquitous in the industry – at least in name. It is certainly the standard in regard to features that other tools emulate and is a good introduction to purpose-built tools for project management. However, it doesn't span multiple projects easily, so for program or portfolio management, tools geared toward portfolio management are best.

I use a portfolio management tool in my day-to-day program management so I can easily track multiple projects and build a program roadmap. I often start in Excel as the resource constraints on how project tasks are planned in most tools do not fit well with how my team handles resourcing. I'll get into this a bit more later in the chapter.

In *Table 5.1*, I've listed some tools, though certainly not all, that cover portfolio and program management and some high-level features for comparison. Again, it comes down to personal preference and company need but this may help you in making decisions as well as knowing the tools that are out there and in use across the industry.

Tool	Project Management	Portfolio Management	Resource Management	Stakeholder Management
MS Project	☑		☑	☑
Smartsheet	☑	☑	☑	☑
Clarizen	☑	☑	☑	☑
Asana	☑	☑	☑	☑
MS Excel				

Table 5.1 – Comparing program management tools

The project and portfolio management industry has matured a lot in the last five years and now includes quite a few options that are strictly online and require no local setup. From the areas that we are looking at, most of these tools will provide the basic needs to track a project down to the task level and tie it to resourcing. Stakeholder management for most tools is relegated to an entry list and dashboards to share information with stakeholders.

The two outliers on this list, **MS Project**, and **MS Excel**, are added because of their ubiquity. MS Project handles project management well but doesn't offer portfolio management to the degree you see in other tools in the field. It does have good portability to other tools due to its maturity, so it can be a good starting point. MS Excel offers literally no specialized functionality for project management – it is a blank canvas. You can start working and add new fields (columns) and entries (rows) to the upper limit of the software itself. To this end, there is no barrier to entry or learning curve and most people know how to use it. A word of caution: to be effective, it takes a lot of work to add formulas, lookups, and straight-up manual manipulation.

> **Note**
> MS Excel has no built-in functionality but is widely used and thereby included in the comparison table.

Tools are great, especially when they are set up and customized to fit your needs, but this can take time. Time isn't always on our side and sometimes we just need to get moving. Let's explore some things you can do when you need to be quick in planning your project.

Diving deep into the project plan

We're going to explore the step-by-step process of getting a project plan built. This is similar to learning math in the way that you start out writing down every step even when the step is obvious. As you get better, steps are done in your head, or multiple steps are performed at the same time. For now, we'll show all of the steps. In real life, I usually combine the first two and the next two together. The steps we'll cover are the following:

- Requirements gathering and refinement
- Building use cases from the requirements
- Task breakdown with estimates
- Assembling the project plan

Let us look at each of these in detail.

Requirements gathering and refinement

This is a crucial step where a TPM drives clarity into the requirements. As this is the first step toward building a plan, everything builds from this foundation, so clarity is key.

Table 5.2 shows the high-level requirements for the *Mercury program's Windows Rollout Project*.

ID	Requirement
1	Create a **peer-to-peer** (**P2P**) messaging system
2	System must allow sending text messages to other peers on the system
3	Standard UX elements from other messaging apps should be available
4	All messages sent and received for a user must be accessible to the user within the app until the user explicitly deletes a message

Table 5.2 – Initial requirements for the Mercury program's Windows Rollout Project

The requirements are pretty straightforward here but are high-level. Reading through these, there are quite a few gaps and generalizations. Let's take a look at them one by one.

Requirement 1 doesn't mention the type of **P2P** system. Specifically, it doesn't mention that no central servers should exist. This is crucial as some P2P systems do have relay servers.

Requirement 2 doesn't specify anything about the nature of the *text* of the message. The meaning of the text can vary depending on the situation and should be clarified. For instance, rich text supports formatting, whereas plain text does not.

Requirement 3 is vague in that each messaging application has its own look and feel and feature set. There is no common reference here, so assumptions will be made by whoever is doing the design and their assumptions may not match that of the business team. The desired UX elements should be clearly called out.

Requirement 4 is a compound requirement; it is asking for multiple features in the same line. Though this can be okay, it makes traceability more difficult, so this should be broken up into multiple requirements.

Table 5.3 takes the feedback we just discussed and expands on the requirements.

ID	Requirements
1.0	**A P2P messaging system, with no servers, should be created**
2.0	**The system must allow sending text messages to other users on the network/system**
2.1	The text message should support all Unicode characters, including emojis
2.2	The text should support rich text formatting (bold, italic, underline, font type, and size)
3.0	**Standard UX elements seen in other messaging apps should be available**
3.1	Address book of saved contacts
3.1.1	Add to address book
3.1.2	Remove from address book
3.1.3	Load address book entirely or a single address
3.1.3	Export address book entirely or a single address
3.2	User profile
3.2.1	User profile image
3.2.2	User alias should be changeable
3.2.3	Short bio section including description/bio, company, title
3.3	Presence indicator
3.3.1	Configurable statuses
3.3.2	Configurable locations
3.4	Access control
3.4.1	User should be able to accept contact requests
3.4.2	User should be able to block contact requests
4.0	**All messages sent and received should be visible to the user**
4.1	Exception for when a message is deleted by the user
4.2	Messages should contain a status indicator of {sent, received, read}

Table 5.3 – Clarified requirements

The new requirements utilized an outline numbering system, so the original four requirements keep their ordinals and additional requirements are listed below them. The bolded lines are the original requirements. A messaging system is bound to have a lot of requirements and this list is meant to just be illustrative of some of the expansions that could be made.

For *requirement 1*, I added the clarification that no central server should exist as a standard P2P network is de-centralized. This is enough for the moment to ensure that the right type of P2P network is being developed.

For *requirement 2*, two additional requirements were added to provide context that the text should support all Unicode characters as well as rich text formatting. The specific formatting required is also listed to clear any doubt.

For *requirement 3*, we added the specific features we want to include: an editable user profile, an address book, presence indicators, and access control.

Lastly, *requirement 4* was broken down into components for traceability and a status feature was added.

Every person reading this book has a different perspective while reading and thus may find requirements or features that they would add, and possibly some expansion on those I've included here as well. This will always be the case and I suggest that requirements analysis is not done in a closed room and that multiple stakeholders are involved, at least at some point, as they may see a gap that you do not.

Building use cases from the requirements

To provide context to the requirements and make it easier to test and validate, use cases are created that may span multiple requirements but tell a story on the behavior of the system. Each use case will trace back to the requirements that it will satisfy to ensure that all requirements are met.

Table 5.4 shows a typical set of use cases for the requirements we've reviewed.

ID	Requirement IDs	Use Case
1	1.0	As an admin, have no centralized setup or maintenance
2	1.0	As a user, install and use without a central server
3	3.2	As a user, create a user profile with a picture and an alias
4	3.1	As a user, add or remove a contact to my contact list
5	3.4	As a user, block and accept a contact request
6	2	As a user, send messages to a contact using rich text
7	4	As a user, see new messages sent to me
8	4	As a user, see all messages both sent and received
9	4.1	As a user, delete a message
10	3.3	As a user, set presence information

Table 5.4 – Use cases

The requirements are relatively simple for this application so the use cases will closely follow the requirements one-to-one, though in some more complex projects a single use case may cover multiple requirements. A use case should tell a story about something a user should be able to do. You often see them written out as a story, such as this:

As a user/admin/seller/customer, I can perform some action.

This provides context as to what you are solving and creates a test plan item as well! Also, for use cases that trace back to a top-level requirement, such as in *requirement 2*, the use case covers all sub-requirements from that requirement as well.

Task breakdown with estimates

Now, we will move on to the start of the project plan itself. We will take the requirements and use cases we have defined, and break them down into tasks. *Table 5.5* shows this in detail.

ID	Use Case ID	Task	Estimate (Weeks)
1	1, 2	**Create P2P network**	**12**
2	1, 2	Design network	4
3	1, 2	Implement	6
4	6	**Text message send/receive API**	**8**
5	6	Set up API for request/response	4
6	7, 8	Use an Ack tag to track message status	2
7	6	Ensure Unicode support in API payload	1
8	5	Text message to new contact initiates contact request	1
9	4	**Address book**	**16**
10	4	Add/remove API using message protocol/API	4
11	4	Import/export of the address book	4
12	4	Import/export of an address entry	4
13	-	Support search/discovery for members on network	4
14	3	**User profile object**	**3**
15	3	CRUD	1
16	3	Alias CRUD	1
17	3	Bio text CRUD	1
18	10	**Presence object**	**2.5**
19	10	Status key-value pair	1
20	10	Location key-value pair	1
21	10	Support full Unicode including emojis in values	0.5
22	5	**Access control**	**1**
23	5	Accept or deny contact request	1
24	4	**Message library**	**2.5**
25	8	Maintain message list for both sent and received	2
26	9	Allow deletion of sent/received message from list	0.5

Table 5.5 – From use cases to tasks

Each of these tasks expands upon one or more of the use cases and provides an estimate in number of weeks. As you can see, many of these are still high-level *design* and *implementation* tasks. The level to which you break it down depends on the expectations in your team as well as your ability for a given use case or high-level task to be broken down further. For the presence object, I suggest the data type to use as a key-value pair, but in the user profile object, I merely specify that a **Create, Read, Update, Delete** (**CRUD**) model should be used but not how or any specific behaviors.

At this stage, before moving on to the project plan, I will add buffers for the estimates and then review the task list and estimates with my project team. The team should all agree to what they are agreeing to accomplish and roughly the effort that it will take. If you don't have the information about the estimates (ambiguity and confidence), now is when you work with your team to get those gaps filled in.

Assembling the project plan

The project plan is a living document that goes through multiple phases during the life of the project. In the beginning, it is strictly a plan – it marks the planned path to get to the end. However, as the project starts, it will start capturing real-time data in the form of actuals: actual start date, actual completion date, as well as actual resourcing. For now, we'll focus on the starting point of the project plan as this form of the plan is what leads to the rest.

There are a few key pieces of data that you need to gather for any project. Aside from the standard items of the task description, duration, and predecessors, you'll want to capture the requirements that it satisfies. Though I showed this as a separate chart – and it can be a separate exercise – I have found that having a central sheet to see all of the important data is best and reduces the churn of cross-referencing, which can lead to mistakes.

Depending on your company and how you handle resourcing, I have found that capturing the swarming capability of a given task is very beneficial. **Swarming** is the act of having multiple people work on the same task at the same time. This can be in the form of overlapping sub-tasks such as quality assurance and actual implementation or splitting the development up among multiple developers. The swarming count is the maximum number of *cooks in the kitchen* that the task can handle but doesn't represent the actual number of people you might assign if everything is going okay. I have found that having this available ahead of time speeds up my ability to react to changes where I can speed up a task that is in danger of missing a milestone or that impacts other work from being able to start on time.

Last but not least, **start date** and **end date** round out the remainder of your basic plan. Your situation may compel you to add more than this, but these are what I have found are a good starting point for every project. In fact, I use a template for the fields I use to ensure that I don't forget!

Table 5.6 shows the requirements and tasks we've covered in this chapter as a project plan with all of the fields we have discussed.

ID	ReqId	Task	Duration	Swarm #	Predecessors	Start Date	End Date
1	1	**Create P2P network**	**16**				
2	1	Design network	6	1		2-Jan-23	6-Feb-23
3	1	Implement	10	2	2fs	13-Feb-23	24-Mar-23
4	1.3	Create a networkId to tie a message to a given network instance	3	1	2fs	27-Mar-23	14-Apr-23
5	2	**Text message send/receive API**	**13**				
6	2	Set up API for request/response	6	2		2-Jan-23	20-Jan-23
7	4.2	Use an Ack tag to track message status	3	1	6fs	23-Jan-23	17-Feb-23
8	2.1	Ensure Unicode support in API payload	2	1	6fs	23-Jan-23	3-Feb-23
9	3.4.1	New text message to new contact initiates contact request	2	1	6fs	6-Feb-23	17-Feb-23

Table 5.6 – Excerpt of Windows Rollout Project Plan

The plan is relatively straightforward, but we'll discuss a few points to illustrate how these fields are utilized to build out a project plan. First up, the duration here may look different from the duration first shown because I applied buffers to these numbers. In short, no estimate is perfect, and buffering protects your plan from reality. We'll discuss this in more detail in the next section. For the swarming number, most of these tasks are straightforward and very singular in nature and adding additional people likely won't work out, so most are given a value of *1*. A few that are implementation-focused received a *2* as a lot of development can be paired up so long as more than one code package is being touched. In a real example, your developers and software development managers should be able to assist you in determining the maximum amount of swarm you can handle. If you know the systems well enough, don't hesitate to use your own judgment.

For the start and end dates, I used the swarm here as an actual resourcing count, so the duration is split by the resource count. A duration of four weeks but a swarming count of *2* would end up at two calendar weeks. I'll usually capture the planned resourcing as well, but for brevity, I used the swarm field for both.

The last field to discuss is the predecessor. Though this is a default column in all project management tools, we'll discuss it as a refresher. There are a few different types of predecessors you can have. The

most common is the **finish-start** relationship, shown as **fs**, where the task number listed must finish before the given task can start. **Start-start** (**ss**), **start-finish** (**sf**), and **finish-finish** (**ff**) all exist as well and follow the same paradigm. I use finish-start the most – which is likely why it is the default relationship in most tools – though I've found it very useful to include lags. For instance, the implementation of a design doesn't always have to wait until the design is completely finished. There may be aspects of the design, or a key point such as an API definition being ready, where implementation can begin, and you fast-track the implementation. Additionally, you can break the task apart and have the API definition be a separate task and list it as an ff, with the rest of the design as an fs. Just like when swarming, I like to list these areas out ahead of time as well – it doesn't mean I will fast-track, but I know that I can, and the sooner I know, the easier it will be to quickly adapt to changes. Catching these optimizations in the plan that are based on how to fast-track and how a particular story might be broken down into smaller deliverables is a key way in which a TPM brings value to the planning process.

Now that we've discussed the four main steps to take when building a plan, let's see how we might be able to speed this up somewhat.

When planning has to be quick

Even when you have the time, using time-saving tricks can help you focus on the tasks that have no real shortcuts or just need more attention. Planning can be repetitive from project to project, especially when working with the same teams on domain areas as the themes of the type of work, the people, and resourcing available will be the same, and over time, knowing how well the people work together can help with estimations. When moving fast, removing repetitive work is the highest-value place to focus your energy. Here are the areas with highly repeatable effort that we can reduce the time spent on:

- Repeatable high-level estimates
- Management checklists
- Project plan templates
- Buffering

Let's explore each of these.

Repeatable high-level estimates

In many environments, there are repeatable tasks that continually come up across projects. These often relate to configurations and data modeling. Keep a list of these tasks, their descriptions, and the typical effort estimate. I typically use a range for the estimate to give some room for quick tweaking. If you feel a specific instance of a task is more complicated, just use the upper range; if standard or simpler, use the lower ranges.

When building out your project plan, consult this list to easily knock out the repeatable items and move on to where your attention is needed. As a bonus, this list is also a good place to pull from

for software improvements as repeatable work is the best to automate as it tends to have a high and guaranteed **return on investment** (**ROI**).

Management checklists

Just as some tasks are repeatable, there is a lot when it comes to managing a project that is repeated from project to project. Generically speaking, each key management area and process in the project management life cycle has repeated actions: reviewing requirements, building use cases and then a task list, analyzing risks, and building a stakeholder communication plan. The list can be very long as there is a lot to project management, and it is also fungible to a specific scenario. Certain tasks may be reduced to a static list – your stakeholders, for instance, might be the same in a small company or team and may not change from project to project.

As you work on your projects, look at the items you are repeating the most and start creating a list of these items along with standard operating procedures for them. When you are in a hurry on the next project, run down the checklist.

Project plan templates

Many of the portfolio management tools out there offer the ability to create and deploy templates for various aspects of project management. These templates are great for when a particular aspect of the project is standardized for your company or team. They can also take parts of your management checklist and delegate them to others as the template with needed or pre-populated information is already there.

For a project plan, tasks such as requirements gathering and verification, creating functional specifications, and launch plans are tasks that are present for every project. In software development, tasks such as code deployment, integration testing, and end-to-end testing are also always present. These may be added at the end, or possibly per feature or milestone depending on how your software landscape operates.

Buffers

Part of your project plan is adding in task estimates. Aside from the repeatable work I mentioned previously, estimates typically come from subject matter experts, usually developers. As a TPM, you may also be in the position to estimate these items yourself. This depends on your company and team needs as well as your own abilities.

In either case, you will have a set of estimates for each task; however, that is not often added directly into a plan. One of the responsibilities of a PM is to analyze an estimate and project plan and apply buffers. Nothing ever goes exactly according to plan, and it is our job to account for that. To aid in this, I often apply a matrix when determining the right level of buffer. This might take some trial and error to come up with as each person estimates differently, and each company has its own overhead that feeds into standard buffers. *Table 5.7* is an example of what this might look like.

Level of Ambiguity in Task	Confidence of Estimate Accuracy	Team Overhead	Buffer
Medium	Low	10%	35%
High	Low	10%	40%
Low	Medium	10%	25%
Medium	Medium	10%	30%
High	Medium	10%	35%
Low	High	10%	20%
Medium	High	10%	25%

Table 5.7 – Estimation buffer matrix

In this example, the team has a modest overhead of 10%. This accounts for standing meetings, recurring training, and other team activities that take a set amount of time. Each team should have its own overhead and should be revisited often to ensure it reflects the current realities in the team.

The level of ambiguity and the confidence of the estimate work together as a scorecard to determine the buffer. In increments of 5%, high ambiguity adds 15% with low at 5%, and the reverse for confidence with high confidence at 5% and low at 15%. This added to the team overhead gets you the percent increase for each estimate. Notice that none of these is 0% because estimates and real life never agree! There is always a buffer for the estimates you have even if you came up with the estimate. Adding buffers, though intuitive and most of us do this, is not the textbook approach to estimations via the critical path or PERT methodologies. This method is closer to the critical chain method. I've added a resource in the *Further reading* section of this chapter.

Note

Table 5.7 makes adding buffers formulaic or straightforward. However, as was alluded to, there is nuance involved here. I've spent the better part of my career refining the buffers I use, and I always scrutinize my percentages when there are changes to the team or organization or I'm simply starting a new complex project.

We've taken a detailed look into what it takes to drive clarity into requirements in order to produce a working project plan. Now that we have a plan, let's look closer at defining the milestones and feature list for the project.

Defining milestones and the feature list of a plan

Milestones and feature lists are different measurements that a TPM has against the health of their project. Milestones are often pre-determined and in some organizations are exactly the same for every project. These are goals during the project cycle that the project is moving toward. A few common milestones in a software development project would be design completion, implementation completion, user acceptance testing completion, and launch, as examples. For any given software project, these milestones would all exist to some degree and make it easy to compare project health across a program.

Somewhat in contrast to this, a feature list is specific to a project. As it is called, it's a list of features that are being built. From an agile methodology, this could be synonymous with the sprint demo at the end of a sprint where the tenet for each sprint is to deliver value. However, it can be separate as well, but in either case, it is an effective way to share when a specific feature will be available to your stakeholders and makes communicating dependent work easier by grouping tasks together that deliver a feature. These features usually correlate to the bolded top-level stories or tasks in my project plan. For instance, the presence object for location and status is a feature that can be delivered alone and be usable, as is the ability to send a text message. As you can see in *Table 5.6*, the top-level tasks, or features, also correspond to use cases and this is often the case.

Table 5.7 represents the feature list from the project plan.

Feature	**Start Date**	**End Date**
Create P2P network	2-Jan-23	14-Apr-23
Text message send/receive API	2-Jan-23	17-Feb-23
Address book	17-Apr-23	9-Jun-23
User profile object	20-Feb-23	3-Mar-23
Presence object	27-Feb-23	13-Mar-23
Access control	20-Mar-23	24-Mar-23
Message library	20-Mar-23	7-Apr-23

Table 5.8 – Feature list

Though the information is available in the project plan, it is often useful to list it separately as these are key metrics to follow the health of a project. As I covered in *Chapter 1*, different stakeholders require different levels of information and most stakeholders don't want to sift through a full project plan to discern what is happening, so this provides a concise snapshot of the project's trajectory.

You should now have a better understanding of the different ways TPMs drive clarity while executing plan management. We used the Mercury program to build out from requirements through to a project plan. We also discussed some tricks to lighten the work during this phase. Next, we're going to discuss resourcing and how it may look different in a technology company compared to other business domains.

Planning resources in a technology landscape

Every industry has its own unique challenges to project and program management and the tech industry is no different. There are two main aspects to resourcing that are consistent across the tech industry and are worth exploring in more depth: prioritization and team overhead.

Prioritization

From my experience, and through the interviews I conducted for this book, tech companies don't follow a **projectized resourcing model** where a project is formed, resources are assigned, and the project keeps the same funding through out the life of the project. Instead, these large companies utilize **capacity-constrained prioritization**, where they perform multiple prioritization exercises throughout the year to align their existing capacity to the most strategic projects. The frequency of these exercises varies from company to company, and in some cases, team to team, but can happen as often as monthly, quarterly, yearly, or some combination of these. This cyclical prioritization relationship is demonstrated in *Figure 5.1*.

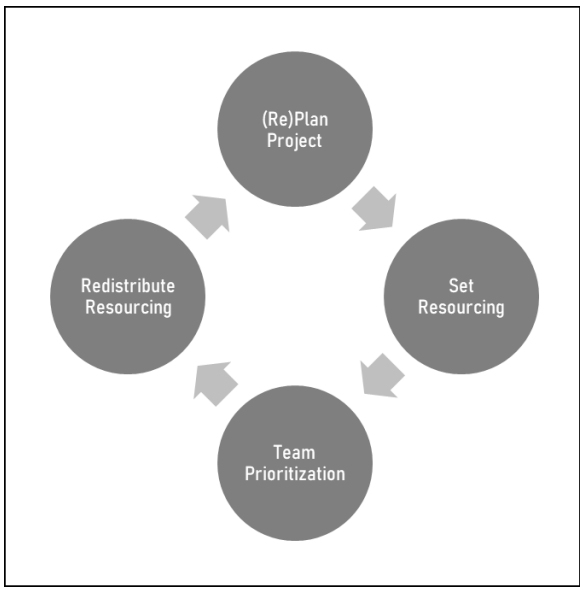


Figure 5.1 – Cyclical prioritization

The preceding diagram shows the relationship between the project plan and the capacity-constrained prioritization process used in tech companies. This is essentially a specialized version of the **Plan, Do, Check, Act** (**PDCA**) process. Once a project is planned and resourcing is set, a team prioritization cycle can then change the number of resources for your project either up or down. Either way, this leads to re-planning based on the new numbers you have. This is why I front-load a lot of pre-planning work such as which tasks can handle more resourcing, or swarming, adding in a buffer, and fine-tuning the predecessors so I know quickly how tight a finish-to-start relationship actually is.

Team overhead

We briefly discussed overhead when talking about shortcuts to use while project planning. In the example, I used a 10% overhead buffer. In reality, I've seen anywhere from 10% to 40% overhead on various teams I've worked with. Let's talk about a few of the major components of overhead that you'll want to keep track of.

On-call rotations

Many software development teams have on-call rotations where a software developer will listen to ticket queues for problems. If a severe problem occurs, a developer may be asked to look at the ticket at any time of the day or night. Most on-call rotations are a week in duration where the daily duration may be all 24 hours or split up into shifts to cover the full day or the working day depending on the criticality. During the on-call, the developer is often not available for project work.

Depending on the size of your project and the number of teams you are working with, on-call rotations can be a constant source of resourcing loss that can be much worse than other sources of loss such as vacation. A vacation is usually known well in advance and impacts a single person's tasks. On the other hand, on-call jumps from person to person, it can impact multiple resources in the same week, and developers may not remember they have an on-call rotation coming up (it's not nearly as exciting as a vacation, so I don't blame them).

To help combat this, the buffer time for your project should look at the teams involved and their rotation schedules to ensure you have enough slack to cover the intermittent stoppages in work. Check with the SDMs and the SDEs to understand how heavy ticket queues are as some teams rarely have major issues and an SDE who is on-call may be able to have some project time.

A quick note about on-call and overhead is that I have heard the argument that on-call is separate from overhead and should be tracked separately. To me, this depends on the team(s) you are working with. I usually start with a generalized buffer for overhead that includes on-call and may add explicit on-call items into my project as I know more about the rotation schedule.

Training and team-related work

Like many other industries, technology evolves constantly. To keep up, developers, as well as SDMs, TPMs, and others, attend training to learn about new advances or to brush up on known skills. These are a bit harder to track as it requires constant vigilance on the schedules of your developers and can be as painful as on-call. When a popular course such as **Design Patterns** comes along, a large number of developers will take the course and can all be gone at once.

Most companies have a schedule of classes that you can use to ask about availability with your team during sprint planning if running agile or some other planning forum. It's also good to ask SDMs about training quotas for their team and see whether they know the current trend in attendance to help you determine a good buffer.

Team-related work and offsite meetings have a similar impact on the project as training does. Many tech companies have started a trend of hackathon weeks, availability weeks, or improvement weeks. These all are geared toward allowing developers to work on projects that directly impact their services' maintainability or add some quality-of-life features. During these workshops, the entire team may not be available. The same methodologies apply here in terms of a buffer and consultation on the frequency of these events.

Company- or organization-wide outages

In larger tech companies, there are often company-wide events that can restrict code deployments. The events may be internal, geared toward reliability checks or security, or they may be externally motivated such as a sales event or a press conference for a new product. Though these are not team-by-team or resource-by-resource, they do need to be considered in the overall project plan.

Team meetings

Most teams have some number of standing meetings that range from operations to team-building exercises. This number is often set and the SDMs should have a good idea of what the impact here is. As a note, when discussing overhead, it isn't too unlikely that this bucket is the only bucket people think about, so be sure to probe when discussing an overhead number so that it covers all of these topics.

Project versus non-project hours

All of these components add up to subtract from the project hours available to a specific team. As each team may be different, there may be different degrees of overhead that you will be tracking in the project. *Figure 5.2* demonstrates the breakdown of project and non-project hours.

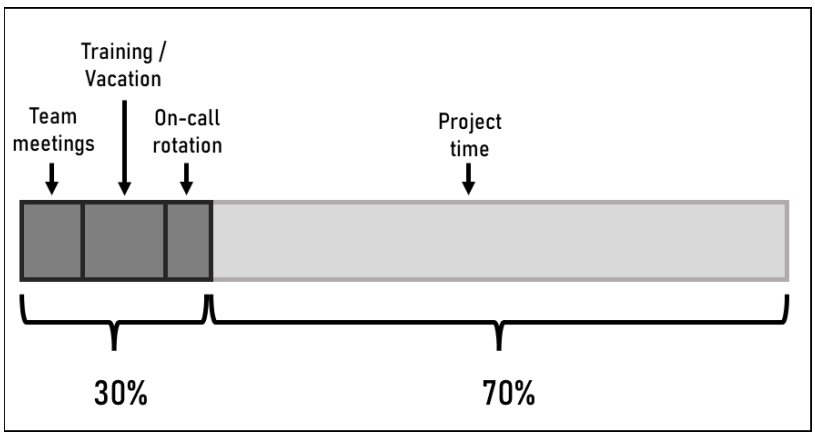


Figure 5.2 – Showing overhead, non-project, and project hours to determine available hours

In this scenario, the team has an overhead of 30%, leaving 70% of the team members' time to work on project tasks. The 30% is inclusive of the three sections discussed in the previous section: team meetings, training and vacation, and on-call rotations. The 70% is likely to fluctuate month to month based on actual team meetings, whether or not a particular person was in the on-call rotation for the month, as well as vacations and training. The *law of large numbers* states that with a large enough dataset, the results will be close to the expected outcome. This means that the overhead percentage of a team will equal the expected amount over the course of a project. When this doesn't happen, it's usually because the expected overhead is wishful thinking – ensure that you and your team are being honest when determining this number; it may just uncover a need for process improvement.

Resource overhead

The overhead mentioned here is illustrative and will change from team to team. When I discuss overhead it references the type of overhead that there is some control over. If absolutely needed, I can request that an SDM take an SDE out of the on-call rotation or pause training for a limited time.

Now let's move on to discuss the tools that you will use for resource planning in a technology landscape.

Tooling for resource planning

Just as with project planning, most tools on the market cover resourcing in the same way: non-swarming and projectized. For some industries, this practice is fine and follows how they approach resourcing. However, as discussed in this chapter, tech companies tend to use capacity planning that may shift due to priorities over the life of the project.

The lack of swarming capabilities means that a task timeline won't automatically reduce when you add additional resources, it will actually reduce the time each resource works on the task and keep the duration the same. Some tools allow you to change this behavior, but it isn't consistent. This means any time you need to swarm – or even want to know what the impact of swarming would be – you need to do it manually either through changing durations or forcing the hours spent to reduce the calendar time.

In prioritization meetings, I am often approached by management asking what dates I could meet with *x*, *y*, or *z* resources. So, instead of me dictating the number of people I need, I help weigh the number of resources against priority based on the best ROI I can provide. In these circumstances, I often export to Excel in a Gantt chart style and manually move tasks around to see the outcome.

The other issue, somewhat alluded to, is that the resourcing provided and the optimized resourcing for a project rarely match. This leads to issues where tasks with no written predecessors will all start at the same time, regardless of the number of people assigned. Some tools, such as MS Project, will warn you that a resource is overbooked but requires step-by-step approval to shift timelines. This is still far better than other tools that completely ignore, or hide under a budget sub-system, the fact that

resources are overbooked. Not all tools are like this either, but I am yet to find a tool that can handle all of these issues well, so you are left with manual work in at least one of these scenarios. For this, I'll often use a forced constraint for the start date or create a fake predecessor based on resource availability.

When planning has to be quick

When planning a project, ensure that you do not forget about overhead. Either ahead of a project, or through multiple projects with multiple teams, build up tables of standard items per team such as vacation trends, on-call rotations, team outings, or improvement weeks, as well as training.

If the tables are still taking longer than you have, then a quick chat with SDMs on their understanding of their teams' overhead can suffice early on in a project. Usually, a 10-minute conversation with an SDM to talk through the overhead list from this chapter and fill in values is enough time for a rough number. As it is an estimate in and of itself, don't forget to add a buffer until you have enough data to make the numbers more concise.

With regards to vacations, these often vary from person to person, some patterns may exist within your team, especially depending on the type of services they run. Local holidays often play into when people will take time off, such as Christmas, or a simple 3-day weekend that is padded to 4 or 5 days. Watch for these patterns and ensure that they are considered when planning.

Now that we know a little bit more about resourcing in a technology landscape, let's apply this to the project plan for the Mercury program in *Table 5.9*.

Task	Duration	Swarm #	Resourcing	Start Date	End Date
Create P2P network	16				
Design network	6	1	Arun	2-Jan-23	6-Feb-23
Implement	10	2	Arun, Bex	13-Feb-23	24-Mar-23
Create network Id	3	1	Arun	27-Mar-23	14-Apr-23
Arun on-call	1	1	Arun	20-Feb-23	24-Feb-23
Bex on-call	1	1	Bex	6-Mar-23	10-Mar-23

Table 5.9 – Updated partial plan with resourcing

For the sake of formatting and space, I reduced the fields that are visible as well as the number of tasks to be enough to understand the concepts we are discussing. As you can see in *Table 5.9*, Arun and Bex both have on-call rotations during the project. They both fall during the implementation of the P2P network between 2/14/23 and 3/28/23. Notice that the duration is 10 weeks and with two resources this should last five weeks on the calendar, but it actually takes six weeks. This accounts for the week each that both Arun and Bex are unavailable.

As I did here, the on-call is a separate task on the project so that it is easy to track. However, some tools allow you to enter calendar time off in the tool itself, and this can automatically shift work during the time off. Though a bit of extra work, I prefer the approach of adding tasks because it makes the gap in the Gantt chart much more explicable. With resourcing, you have to cross-reference other sections of the tool and cannot easily convey this via a plan without manual work.

We've discussed project planning in great detail and applied the knowledge to the Mercury program example. We then saw how resourcing is different in a technology landscape. Last, we'll explore how planning differs between a project and a program.

Exploring the differences between a project and a program

Many of the tools and processes are the same between a project and a program. One major difference is scope. In this case, there is the program scope, which has its own set of requirements in the form of the program goals. These requirements are relayed down to the projects that impact them. Though I've been referencing the *Windows Rollout Project*, these requirements could easily have been for any of the other platforms.

When starting the program, you need to refine the goals in the same way you refine the project requirements. In *Chapter 3*, we stated the Mercury program's goal is to have a P2P messaging application with a 90% user reach. This is an okay goal with enough wiggle room to assume success, but from a requirement standpoint, it's too vague. What is user reach? How do we measure 90% of that?

There are also technical issues with this statement. Let's say that the user base means all users of a given platform. The P2P network would span across the world, and thus across physical and logical networks. Many devices that have access to the internet do not have a public-facing IP address meaning there is no way for a direct connection from one device to another when they are on different networks. A relay would be needed to – at the very least – route the call between the devices. This can be done in a way to make the traffic not readable by the relay and for it to be a dumb router, which would satisfy most **information security** (**InfoSec**) teams at large tech firms. However, this loosely goes against the requirement of no centralized system.

Even if we talk about a private installation of this program for an enterprise client, their network is likely large enough for internal network segmentation that also requires a relay. If the relays were also installed on the local network, they would need to be maintained and essentially centralized. The relays could be public and maintained by the Mercury program corporation, but this would force traffic out of the network to travel across network segments.

As you can see, just by probing a bit into two words in the goal, we uncovered several potential issues that the program will need to address before the projects can start their work.

Let's look a bit closer at some of the aspects of plan management that may require some different strategies.

Tooling

The tools used for program versus project planning are largely the same, so long as the tool can handle portfolio management. Having multiple projects under a program or portfolio umbrella allows unique reporting as well as cross-project dependency tracking. Some tools will automatically update a cross-project dependency once a slip occurs while others require an explicit refresh to the cross-project dependency.

The automatic tying of multiple projects can act as a forcing function so that everyone is aware of the dependency as well as any change to it.

Planning

As a TPM running a program, one of your responsibilities will be to find opportunities across project boundaries to optimize time, effort, and scope. This is done by evaluating the technical requirements and possible solutions to optimize the path forward, which can lead to a better architecture by ensuring designs across projects align and are consistent where feasible.

For this program, there are multiple platforms that will have their own instance of the Mercury messaging app. When you have the same app across multiple platforms, there are opportunities for shared code. Some operating systems will more easily be able to share code than others.

When I wrote the original Windows application, Microsoft's .NET was around Framework 3.5 and the **Common Language Runtime** (**CLR**) for non-Windows systems was not robust. This would have meant that only code written in C – a language shared by all of these platforms – would be shareable. Arguably, the pain of writing in such a low-level language would outweigh the benefits.

In recent years, there have been frameworks that more capably span across operating systems. With the proliferation of smartphones and smart TVs, the desire to write once and deploy in multiple platforms has led to better tools to facilitate this.

Mono Project is one such framework. It uses the open source version of the .NET framework and has CLRs for all of the platforms that we are targeting in this program. The amount of code sharing possible will depend on the level of system coverage each CLR has on its respective operating system as well as the number of idiosyncratic definitions of network protocols and other lower-level APIs as these can cause divergencies in implementations.

From a planning perspective, these questions may become initial work at the program level to determine the right path forward in terms of frameworks to use and the target amount of code sharing that balances the difficulties in coordinating and writing with the reduced scope per-project. This is yet another example of where a *technical* program manager brings value to the program by evaluating technical avenues to increase the efficiency of the plan.

Knowing when to define a program

As discussed earlier in this section, scope is the major differentiator between a program and a project. *Chapter 1* also discussed that a program often has a longer timeframe than a project. When your company's projects are around half a year to a year in length, a program may be multiple years long. If projects are a few months long, a program may only last up to a year. In any case, though, they are relative to one another.

When you are faced with a set of requirements or goals and are trying to decide whether to spin up a project or a program, there are a few litmus tests you can do to see whether a program is the right fit:

- Do you have multiple goals to achieve?
- Is the timeline fixed or based on achieving the goals?
- How many stakeholders are involved?

If you have multiple goals to achieve, then multiple projects, one or more per goal, would ensure clarity of purpose for each project. In this case, a program to manage these projects would make the most sense. To be clear, by multiple goals, I'm not referring to features or milestones but end goals.

If the goal you are trying to achieve has a deadline, this is often a reason for a focused project to drive toward the deadline. If you have multiple goals, and one is time-bound, a program with a project specifically for the time-bounded goal may be a good fit.

Lastly, if your goal spans multiple departments or organizations within your company, all driving toward the goal or goals, a program with projects per organization may be the most effective way to manage the large number of stakeholders. Each organization may have a succinct deliverable that contributes toward a goal.

Figure 5.3 takes a close look at the company organization structure and how the Mercury program and the projects within relate.

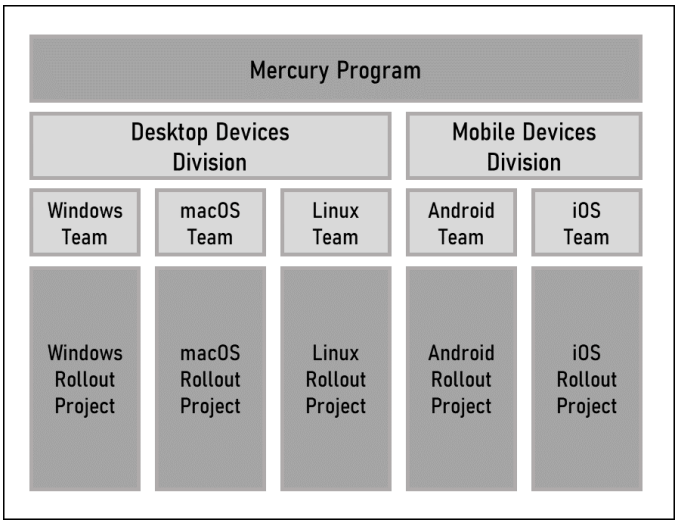


Figure 5.3 – Project and program boundaries

The company responsible for the Mercury application divides devices between desktop and mobile. Within these divisions, each operating system has its own team with its own roadmap and priorities. The Mercury program spans both divisions and each operating system team is responsible for a project within the Mercury program to deliver Mercury on their OS. This can be useful to map out because at first glance, the number of stakeholders in this example isn't too large, but the way they are organized and the large vertical slice of the program goal that they each own lends this to be treated as a program with projects as opposed to a single project.

As the questions alluded to, a program will often span multiple organizations and goals. It utilizes the Mercury program's company structure to show how the program spans multiple organizations with each organization having a dedicated project.

An answer to one of these questions in favor of a program does not mean that a program is absolutely required but may lean your answer in that direction. For instance, a regulatory compliance project in a company may need to span operations and accounting organizations but with a deadline and concrete deliverable, a project is likely the best way forward. Conversely, the Mercury program has a single stated goal but spans multiple organizations each with a concrete deliverable of an app that contributes to that goal and with no constraint on the timeline. Though this could be treated as a project, the complexities across the organization and distinct deliverables lend themselves to be treated as a program.

At the end of the day, these questions and answers are inputs to help you decide what the most effective and easy way to manage your goals will be for you.

Summary

In this chapter, we discussed plan management in greater detail. We drove toward clarity by refining requirements into use cases, tasks, and then a basic project plan. Asking questions during each step ensured that each artifact could be traced back to the requirements.

We covered how a tech firm can add unique challenges to plan management through capacity-constrained prioritization causing mid-project changes in resourcing based on priority shifts. We also discussed the components of team overhead in a tech team including on-call rotations that service-based teams utilize for service health and stability.

We started discussing the various tools that are available to program managers for both projects and programs and each key management area. Lastly, we discussed how planning differs between a project and a program, which is tied to scope, and that defining a program comes down to ease of management of the goals you are delivering.

In *Chapter 6*, we'll continue the deep-dive discussions and focus on risk management. We'll cover in more detail what risk management is, how driving clarity works in risk analysis, and discuss the unique challenges in technology that contribute to risks you will likely see in your project. We'll also build on the Windows Rollout Project plan that we've developed in this chapter and run a risk analysis.

Further reading

- Dr. Goldratt, Eliyahu. *Critical Chain* (North River Press, 1997). This book describes the critical chain methodology that I have discussed under the *Buffers* heading in this chapter. The method is more intuitive to the way we work and give us a tangible way to handle the unknowns by assigning buffers based on complexity and ambiguity.

6
Risk Management

In this chapter, we'll discuss the risk management process and the type of risks that a TPM may deal with in a technology landscape. We'll dive into the risk assessment process in greater detail and see how to successfully catalog project and program risks and methods to reduce or remove their impact.

We'll dive into risk management by covering the following topics:

- Driving clarity in risk assessment
- Managing risks in a technology landscape
- Exploring the differences between a project and a program

Driving clarity in risk assessment

During the introduction to driving clarity in *Chapter 4*, we discussed the cyclical relationship between project planning and risk assessment. Now, we'll drive clarity in terms of what risk assessment is and how we continue to manage risk throughout the project's life cycle.

Risk assessment starts as soon as you have enough information to start assessing, which can be early on in the project. If you start with a project in a domain that you know very well, a single paragraph about the goal of the project may be enough to start analyzing risks. In other cases, where the domain or project itself is too vague, it may take a full requirements document to start analyzing, which is in itself a risk!

Regardless of when the process starts, the steps are the same. Part of the process was outlined in *Figure 4.3* in *Chapter 4*, but the full process has two additional steps that *Figure 6.1* outlines:

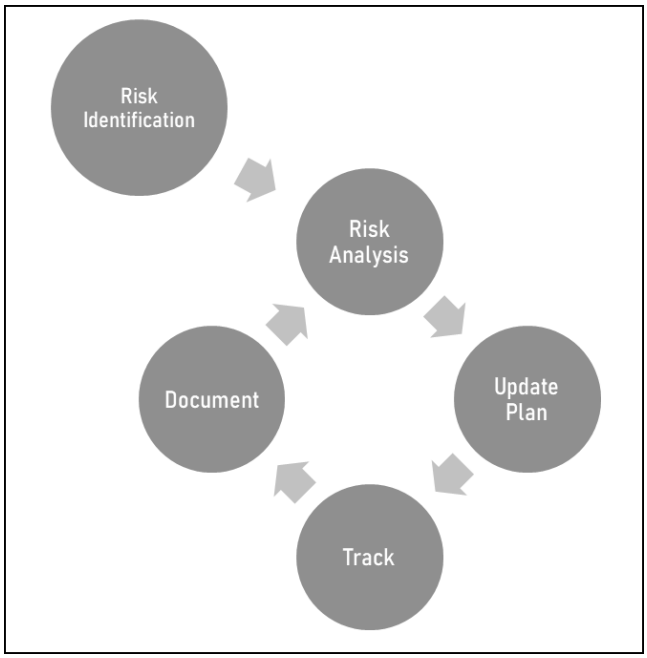


Figure 6.1 – Risk assessment process

The track and document steps were added to the existing three steps that were covered in *Chapter 4* – that is, risk identification, risk analysis, and update plan – to round out the process. Each of these steps is part of the assessment cycle, which is a continuous process throughout the project. We'll discuss each of the five steps here.

Risk identification

In *Chapter 3*, we briefly discussed methods to identify risks. These included using a company risk register that contains all past risks from previously completed projects to find risks that fit scenarios in the current project. As an example of a previous project's risk being useful, a project that had risks related to networking communication would be of interest to the Mercury program to ensure the same risks are accounted for.

In the absence of a risk register, or in combination with it, working with your stakeholders and project team to discuss potential risks is a good approach as well. The more experience you have to draw upon, the easier it is to identify risks and solve complex issues. One of my favorite examples of this is the introduction sequence to the show *MythBusters*, where the narrator announces that Jamie Hyneman and Adam Savage have 30 years of combined experience. In this case, both Jaime and Adam had careers up to this point, so their combined work and perspectives are equivalent to a single person's 30-year portfolio. Together, they were able to see the same problem from different perspectives and find ways to move forward. The show often showcased Jamie and Adam pointing out problems with each

other's designs via their perspectives and experiences. In this type of scenario, **tribal knowledge**, also called **institutional memory**, can be extremely powerful as it allows for quicker execution within the organization. Tribal knowledge is made up of both documented or documentable knowledge, known as **explicit knowledge**, as well as **implicit knowledge**, which is a person's experience and hunches that can only be conveyed through hands-on discussion and instruction. It's usually knowledge that is well understood within the organization and acts like muscle memory, meaning less time searching for documentation for a process or policy.

Risk analysis

Searching through registers and discussing with stakeholders and colleagues can lead to a list of risks that need to be analyzed. As with most processes, this can happen organically while having the discussions or it may happen separately. This analysis is where you determine the level of impact the risk would have on the project should it occur, as well as the probability of the risk happening. Both of these are somewhat subjective and will vary from situation to situation. The same risk that was present in an older project may not have the same level of impact or probability on a new project. For instance, let's say the risk was related to code deployment delays due to a congested service pipeline. As time passed, the reasons the congestion occurred may have improved or worsened. Also, there may be more or fewer code changes for that pipeline than in the older project. Both of these will change the impact the risk has, as well as the probability of it occurring.

During analysis, you might decide to add project tasks to dive deeper into the risk to understand it more. For instance, if you have a risk of using a new framework that your team doesn't know about, you might add a deep dive task to do a quick proof of concept or find training and get everyone signed up for it, or both.

Some risks may have inherent metrics that you can use to track impact and probability. In the pipeline example, there could be deployment counts and failure rates. Combine this with the number of tasks requiring code changes in that pipeline and you can reliably apply a formula to estimate risk impact and probability.

Other risks will not have metrics or will be indirect correlations, such as developer attrition rates. You can apply an average attrition rate to assess the likelihood of losing a developer, but statistics are about long-term data trends and do not predict immediate behavior.

Updating the plan

In *Chapter 3*, I listed the plan step as updating the project plan. Though true for that example, you can't update your project plan unless you have also planned what you intend to do about the risk itself! The risk strategy is often referred to as risk mitigation, but mitigation is only one of four different strategies. I've heard some call the risk strategy risk optimization as you are optimizing the strategy for the risk, both good and bad. Each of the strategies has different names, depending on the system you use; these are the names that I use:

- **Avoidance**: This is when you create a plan to avoid the risk. For instance, if there is a risk involving a faulty piece of equipment, or a vendor with a poor reputation, then avoiding the use of that equipment or vendor is your strategy to avoid the risk. This type of risk is likely one that comes up more often than you would initially think but you often organically deal with the risk and then dismiss it. In these cases, the risk may never make it to your risk log or subsequent company register, but the risk was real and avoided. In some specialties, such as security threat modeling, these are cataloged and still listed in the project risk log, even if they are completely avoided from the start!
- **Mitigation**: This is when you allow the risk to happen – or it just does – and you minimize the impact that the risk has. For instance, let's say that a risk can lead to a 4-week delay in a milestone. You might be able to mitigate the risk by **crashing** (also called **swarming**) the remaining tasks with additional resources to reduce the delay to 2 weeks. The result of the realized risk causes you to introduce a countermeasure to soften the impact.
- **Transference**: This is where you transfer the risk to another group. This may sound cruel, but it might make the most sense, depending on the risk. If there is a risk of delays in development due to a team not knowing a framework well, contracting a group that knows that framework may be a better course of action. This option may not always be available, depending on the company and their willingness to contract out or the expertise existing elsewhere.
- **Acceptance**: This is an option when there is no way to avoid, mitigate, or transfer the risk. Sometimes, the project will just need to take the hit and adjust timelines to compensate. In some cases, there may be ways to avoid, mitigate, or transfer, but the cost to do so is greater than the impact the risk will have on the project. Sometimes, *failure is the best course of action*.

Risk strategy selection

Each risk can have multiple strategies. In the example given for mitigation, you can also have an avoidance strategy where you re-plan your tasks in a way to add a buffer to timelines. In such cases, the strategy you choose depends on the current state of the risk. If you attempted to avoid it but the avoidance didn't work, for example, then the mitigation strategy would need to be used. If the mitigation can't happen because resourcing wasn't available to crash when needed, then acceptance may be the only course. So, it is important to look at each risk and see all the different strategies that you might be able to employ and how they may fall back on one another.

The next step in the risk assessment process is tracking.

Risk tracking

Once you have identified, analyzed, and planned for risks, you need to track them. Risk tracking is cyclical and can lead to re-analyzing, identifying, and planning on the fly as circumstances change in the project. Their impact may decrease as the project gets closer to completion, or the probability may go up as various factors play out – such as attrition causing reduced resourcing.

Also, risks may come to a point where they are avoided or can no longer impact the project. New risks may also come up as the project evolves. For instance, a security flaw could be published that requires internal systems to be updated immediately. This could result in a temporary standstill of all work in a particular system or package while the vulnerability is being addressed. Though some generic buffer can (and should) be added to unknowns such as this, the risk was still new during the life of the project and needs to be added, analyzed, and tracked.

Documenting the progress

As you are tracking, any updates should be included in your communication plan. Changes to risks should be clearly stated so that there are no unexpected changes in status due to a newly impending risk being realized.

Any changes to strategies, closed risks, or newly added risks should be well documented in the project risk log, status updates, and in the company risk register upon project closure. Documentation is how you build institutional memory that others can rely on.

Now that we know the risk assessment process in more detail, let's discuss the tools used in risk management.

Tools and methodologies

For a company risk register, some portfolio management tools will provide this functionality. In the absence of these, using whatever centralized documentation tools you have available to capture risks for future reference is useful.

Any project management software has some ability to manage risks at the project level, usually in the form of a risk scorecard and table entry. The scorecard helps you identify the **risk score** or severity of a risk on a fixed scale. The higher the risk, the closer it must be tracked as it will likely cause some level of issue. This is part of the typical *PMP Risk Log* system and is the most common for general use.

There are many different scales in use for risk scores. *Table 6.1* shows the scale and scorecard I prefer to use:

Probability	**Impact**	**Risk Score**
Low (1)	Low (1)	Low (2)
Medium (2)	Low (1)	Low (3)
High (3)	Low (1)	Medium (4)
Low (1)	Medium (2)	Low (3)
Medium (2)	Medium (2)	Medium (4)
High (3)	Medium (2)	Medium (5)

Probability	Impact	Risk Score
Low (1)	High (3)	Medium (4)
Medium (2)	High (3)	Medium (5)
High (3)	High (3)	High (6)

Table 6.1 – Risk scorecard

I use the same three-tier scale across both impact and probability, as well as the resulting score. I have found that a four- or five-tiered scale is too nuanced and can lead to nitpicking the analysis of the probability and impact. The difference between *High* and *Very High* is just too subjective. In contrast, the jump from *Low* to *Medium*, or *Medium* to *High*, is a bit more abrupt when your scale only allows for three levels, and the clarity of whether something is one of two things is easier to classify than whether something is *Medium*, *High*, or *Very High*. A similar strategy is often used in agile sprint planning when the team is collectively estimating the tasks in their backlog. Instead of a linear scale of 1 through 10, the first few Fibonacci numbers are used (1, 2, 3, 5, 8, and 13). This is to remove some debate of what a 3 versus a 4 would be and to add in some critical thought when you need to jump from 8 to 13 as you'll need to have a reason to make that jump.

I've included the numbers next to the tiered values to help illustrate that the resulting risk score is merely the sum of the probability and impact ranks. This also ensures you are being honest in your scoring by not tweaking the score based on any bias.

The risk score correlates to the amount of attention and energy a particular risk may warrant from you. If the risk score is low, then either the impact or probability is low, or both, and not much energy is needed to keep track of it. However, for a high-risk score, the impact is either high or it is almost inevitable that the risk will be realized, meaning that you will need to be ready to act upon your strategies and likely need to watch the risk closely.

In *Table 6.2*, I've included a few high-level, first-pass risks associated with the *Mercury* program:

ID	Risk	Probability	Impact	Strategy
1	Cross-platform tooling issues	High	High	Acceptance: Shift timelines to account for delays Mitigation: Training and crashing
2	Tight testing timeline	Medium	Medium	Acceptance: Shift timelines to account for delays Mitigation: Shift timelines to allow for buffer
3	App store approval delays	Low	Low	Acceptance: Shift timelines to account for delays

Table 6.2 – Risk log for the Mercury program

In this risk log, I do not include the actual risk score as it is derivative of the probability and impact. It also saves space for this illustration. However, depending on your stakeholders and status audience, including the calculated risk score for clarity's sake may be helpful.

For the first two risks, I included both an acceptance and mitigation strategy. Though both wouldn't necessarily be utilized, depending on the situation when the risk is realized, a different approach may be appropriate.

For the first risk, mitigation can also work as an avoidance tactic. With this risk being identified early enough in the exercise, the best course of action may be building training and crashing into the schedule at the beginning to hopefully avoid the issues of a new platform. However, if not possible at first, these strategies can be used to mitigate the impact it has on the schedule in-flight.

For the second risk, the mitigation and acceptance are also the same – it depends on whether you account for it early or late. This isn't always the case but was appropriate for this risk. From a planning perspective, ensuring testing has enough room for iteration is the best course of action if you have the space in your timeline to make it work.

The last risk is low-impact and low-probability, so it is a good candidate for acceptance. A quick search of iOS app store approval rates shows that they sit at 60% approval within 24 hours and the outlier approval approaches 5 days. This is a small enough impact and probability that the shift in timelines is acceptable. This could be avoided by adding enough buffer to account for the edge cases on approvals as well.

When risk assessment needs to be quick

When timelines are tight, there are methods to use to ensure that risk assessment is not skipped over entirely. Though there are likely to be gaps in analysis when rushed – as the risk in and of itself should be cataloged – capturing some is better than running the project completely blind to what may come at you down the road.

The risk register, if it exists, is a real time saver if it is searchable using multiple criteria. The more you can filter the results, the more relevant they will be to your specific needs. The probability and impact would still need to be determined but the identification can be largely copied and pasted if need be. Then, a quick iteration to determine the risk score is needed. Often, the strategies to solve a particular risk can be reused if the risks are specific enough to the situation. Again, peculiarities in the exact project may require some tweaking but it's better than working from scratch.

I've found that in cases where a fully indexed risk register isn't available, even risks that are grouped into larger thematic buckets can be a good shortcut. Think of this as adding labels or tags to each item, such as *networking* or *cross-platform coordination*. Even project types can be useful categories. For instance, if you have an ecosystem with multiple data flows, based on the type of transaction taking place, each transaction type or use case could have its own set of risks that are common within that use case. The same refinements would apply here to the probability and impact, as well as strategies.

Lastly, doing a working session with stakeholders to talk through requirements as a single pass can provide a lot of value in a short amount of time. Build in time during the session to identify, analyze, and plan the risk strategies; then, you can update the overall project plan for cases where avoidance was the best strategy to follow.

Driving clarity in risk management is your responsibility as a TPM, and like most aspects of project management, it involves the help of others. In the case of risk identification and analysis, the more people that can contribute to your success, the better.

So far, we've discussed some aspects of risk management that you will see, some of which exist regardless of the industry you are in. Next, let's look specifically at the type of risks you will encounter in the tech industry and use these scenarios to drive further clarity into the *Mercury* program.

Managing risks in a technology landscape

There are various aspects of software and hardware development that are common in the industry but less so in other professions. These relate to the development process itself. Let's take a look at a typical **software development life cycle** (**SDLC**) and discuss the common risks that arise at each stage:

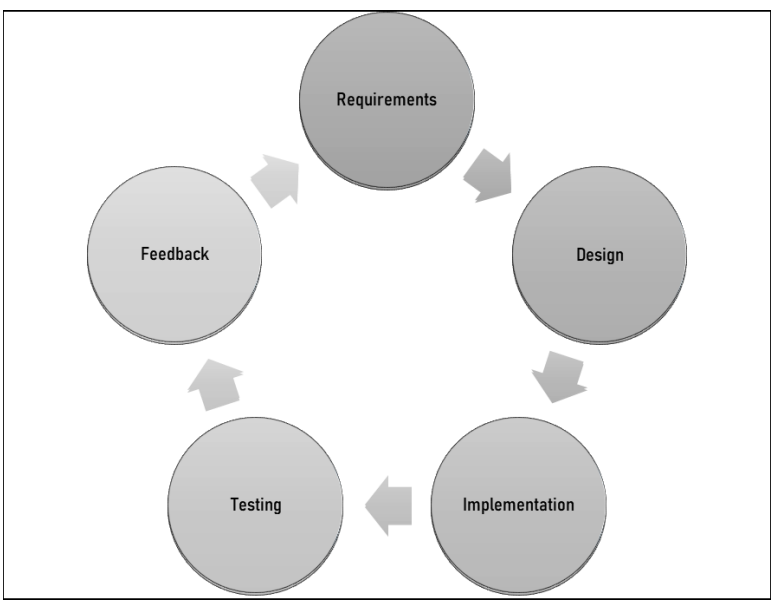


Figure 6.2 – SDLC

The **SDLC** specifies the series of steps or processes involved in software development. This cycle can be adapted to any style of development, including waterfall and agile methodologies. The steps may vary or happen in mini-cycles within the larger cycle, but the steps are still present. Let's explore each one.

We've already talked about the *requirements phase* in various forms in this book. In general, the focus of the requirements phase is to reduce the overall risk by bringing clarity to the requirements. The clearer the requirements, the less room there will be for scope creep in low-level designs and their implementation. In that sense, the main risk in this step is not driving clarity.

The *design phase* is where the high-level and low-level designs are drafted based on the requirements. This process can uncover technical issues with the requirements that were not apparent at first. Aspects that are specific to the technologies being used in a service, or how that service interacts with other services, will come out here in the API definitions, architecture strategies, and release strategy. For **distributed systems**, also known as **Software-as-a-Service** (**SaaS**) architectures, aspects such as latency or availability can play a role in changing or adding requirements and increasing development time as systems often have specific latency targets and availability goals. If your organization is in this type of environment, this risk should always be listed per your company's practices. While you work on getting estimates for work in the requirements phase, you can ask specifically about hot topics such as latency to ensure they are considered in the estimates given. If there is a set process involved at your company, adding a buffer to account for it may also be an option.

The *implementation phase* is where the code or hardware is built based on the designs. Developer overhead should be considered in this phase, but this is where a lot of that buffer will be taken up. So, if the team is new, or you are new, and the buffers are newly minted, this could be a risk to watch out for. Track your buffer and constantly re-base your plan as the buffer is taken up. Dates may need to be adjusted or milestones crashed if the buffer is insufficient. Two specific risks may crop up here that the design process may not have caught.

Deploying the code is often not factored into the effort estimates from developers. I often talk about the concept of a task being *done, done, done*. In that, I mean that a task, or bit of code, is not truly done until the code is written (first *done*), it is reviewed and approved (second *done*), and finally has made its way to the production fleet and been verified (the final *done*). Knowing your **time to production**, or the amount of time post-code completion that exists until the code reaches production, is a good metric to ensure you are adding enough of a buffer. Also, any temporary stresses on the deployment pipelines should be accounted for. As there are always *unknown unknowns*, tracking this as a risk is always prudent.

During implementation, you will also need to perform any security certifications that need to take place. Every company's **Information Security** (**InfoSec**) team will have different requirements, depending on the risk aversion of their organization and the nature of the data they work with. These certifications can take up a significant amount of time, depending on what data your service needs to handle, store, and vend. A new service typically takes longer than re-certifying an existing service. The timing of these activities should be known in the company and can be referenced. Regardless of how prepared you are for this process, it is an unknown that is out of your direct control – the *InfoSec* analysis may find an issue your team didn't foresee and delay the process further than planned.

The *testing step* is where a lot of the *unknown unknowns* cause issues. These can be problems that only robust testing was able to find, usually in edge cases for your software or service. The level of test

framework maturity you work with will determine the level of risk involved here. If you are testing a self-contained software package, then high-test coverage can suffice to greatly reduce the chance of undetected defects. In larger, distributed systems, there may be software states or transactions that are so vast in number as to make full test coverage impossible. Also, when adding new features, there are more chances for unknowns since what you are adding is new. These circumstances all add together to determine the level of risk you may face during the testing step.

The *feedback step*, also known as the evolution or iteration step, is where you plan the next iteration of your product or service based on feedback from users and stakeholders. There are no inherent risks to this stage, though it should include risk analysis while prioritizing which feedback to act upon in the next cycle. This would be a form of high-level analysis; for instance, the feedback may be a good idea but still somewhat vague. This can cause scope creep during the requirements and design phases if the requirement isn't sufficiently clear enough to transform into requirements.

Cross-step risks also exist within the tech industry. These impact your resource availability due to factors outside of the project. Depending on the company and team that you are working with, on-call rotations can impact developer availability. Though this should have been accounted for during planning, things such as sickness or impromptu time off can shift the on-call schedules and cause a developer to be on-call more than expected to cover for others.

Another cross-step risk that occurs is also related to security: mandatory software updates. This can come in the form of mundane version updates away from versions that are leaving their support cycle. Though these upgrades are well published, a project may run into a use case where it can't move forward without the upgrade and need to take time to do so. Operational teams can help reduce this burden, but they aren't always caught and can be a risk to consider.

Another version of a cross-step security risk comes when an industry-wide vulnerability has been discovered that requires immediate attention. In December 2021, the *log4j* exploit, referred to as *log4shell*, caused industry-wide panic as firms rushed to patch all *log4j* instances in their services. Naturally, this was an *all-hands-on-deck* situation that diverted a lot of development hours into deploying the patch. There's no arguing, no negotiation of resourcing or payback – projects are just impacted at that moment. These are hard to predict and calling out a risk like this is akin to *crying wolf*. However, it is good to know that these can and do happen. A good amount of buffer can alleviate the impact if everything else is going well in the project. The good news here, if any, is that most stakeholders are understanding of delays caused by industry-wide catastrophes.

Technical risks in the Mercury program

Each project within the *Mercury* program will have technical risks, as will the program itself. We've already discussed the idea of creating a *P2P* subsystem that is shared across the different platforms. As this is cross-project work that creates project dependencies for every project, this would be considered a program-level risk. A separate project may be spun up to handle the sub-system, but the risk merely shifts from the program to that project.

To introduce the subsystem, you can use a cross-platform development environment to organize the shared code and make it easier to utilize in each **operating system** (**OS**). This platform was called out in *Table 6.1* as *Risk ID 1 – Cross-platform tooling issues*. *Table 6.3* provides clarity on this risk by breaking it up into multiple risks:

ID	Risk	Probability	Impact	Strategy
1	Cross-platform tooling issues	High	High	Acceptance: Shift timelines to account for delays Mitigation: Training and crashing
1.1	New **Integrated Desktop Environment** (**IDE**)	High	High	Mitigation: Training on a new IDE
1.2	New coding language for some teams	High	High	Mitigation: Training on a new language
1.3	Cross-team collaboration	High	Medium	Mitigation: Daily stand-ups, co-location, or chat rooms

Table 6.3 – Cross-platform IDE risk analysis

Three additional risks were added in the same way you would break down a user story into smaller sub-tasks. I used outline notation to denote that they are related and expand upon the original risk. This list is not exhaustive of the risks that could occur cross-platform but is used to illustrate diving deeper into a risk to add clarity.

Two of the three risks relate to dealing with new technologies, both the IDE as well as possibly a new language. For instance, if the iOS developers are used to using the **Swift** programming language, they may have a learning curve to switch to a *MonoDevelop*-supported language such as C#. This task may be easier for an Android developer who is used to using Java as both **Java** and **C#** are C-based languages and Swift is not. So, the risk to each project may have a different impact and probability.

The last risk is a risk that occurs anytime you have cross-team or cross-project collaboration as this introduces additional communication. The world today is arguably much better at collaborating across different groups (whether that's different teams, companies, or locations) since the pandemic forced new habits. The risk still exists, especially when collaborating with a team you are unfamiliar with, as there is always a **storming phase** where the different groups are working out how to best communicate with each other.

So far, we've learned about the risk assessment process and the cyclical nature of assessing and planning for risks, as well as some unique risks faced in the tech industry. Next, we'll explore how risk management differs between the project and program levels.

Exploring the differences between a project and a program

Risk management, as a process, does not vary between a project and a program. Instead of thinking about cross-task risks at the project level, you must think about cross-project risks at the program level. Risk registers, risk logs, and scorecards are utilized in both cases.

As for the tools, each one provides risk management forms and templates at the project level. Tracking risks at the program level may require some workarounds in popular tools. Creating a program project within the program to track these concerns is one such way. Transferring risks down to the projects can also work, though this will include some duplication. Depending on the level of cross-project communication, each project team knowing about every risk that impacts them is a way to reduce instances of risks being missed.

Assessment

When planning the project composition of a program, one aspect to look at is how the project structures can create risk. Cross-project dependencies require a higher amount of coordination than task dependencies within a project.

Figure 6.3 shows program milestones with dependencies between them. We'll look at how this division of work can create efficiencies as well as risks:

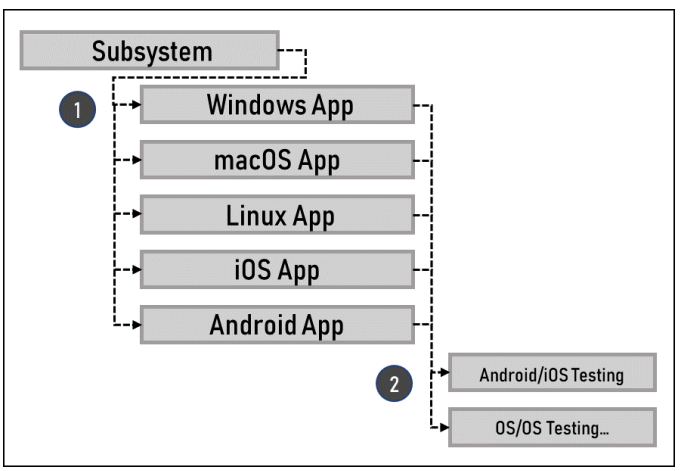


Figure 6.3 – Cross-project risks

In this example, the subsystem for the *P2P* network is a separate milestone and is tracked outside of the OS projects. By doing this, we remove duplicative effort across all five OS projects, but we

introduce the same number of cross-project dependencies on that subsystem, as indicated by label **1** in *Figure 6.3*. If the subsystem is delayed, it will delay all five subsequent projects.

In label **2**, we can see another dependency that exists between all five OS projects. As each OS needs to test interoperability with all other OSes, this can lead to multiple cross-project delays if testing goes wrong in one OS project. For instance, if the *Windows Rollout Project* is delayed, then all testing with Windows is delayed, which can delay all other projects as well. Some shifting of tests can reduce the impact of the delay by testing the integrations that are readily assuming they weren't all being run in parallel, or Windows could be completely separated in scope to allow the other platforms to move forward and launch Windows at a later date if more time were needed.

As you can see, just as we would look for and analyze cross-task risks at the project level, you'd do the same at the program level. Each of these risks may be tracked at the project level but should also be monitored at the program level.

Summary

In this chapter, we learned how to drive clarity in risk management through the steps of the risk assessment process. We discussed the different risk strategies available and how tracking risks is a constant process where the strategy that's utilized to address risk may change over time. Then, we discussed some key risk categories that show up in the tech industry. We also dove deeper into the *Mercury* program by driving clarity in terms of the identified risks. Lastly, we looked at how risk management differs in scope from the program to project levels and used the *Mercury* program to illustrate how a program decision can both remove and create risk based on the project composition of the program.

In *Chapter 7*, we'll close our deep dive into key management areas by focusing on stakeholder management. We'll learn how to drive clarity with stakeholders through various stakeholder processes such as communication plans and status reporting. We'll look at what makes stakeholder management unique in the tech industry and the challenges a TPM will face.

7

Stakeholder Management

In this chapter, we'll discuss the stakeholder management process from determining who your stakeholders are, to building a comprehensive communication plan. We'll discuss what is different about stakeholder management in the tech industry and explore how the communication plan differs from program to project.

We'll dive into stakeholder management in the following sections:

- Driving clarity in stakeholder management
- Managing stakeholders in a technology landscape
- Exploring the differences between a project and a program

Let's get started!

Driving clarity in stakeholder management

Stakeholder management is the art of managing expectations. Each stakeholder has a different needed set of goals for the project, and their point of view will vary from others. The communication plan helps you draft ways to communicate that span the different needs of your stakeholders.

There are two aspects to setting up a good communication plan: defining the types of communication you need and discovering who your stakeholders are. We'll cover the different types of communication first, as this is usually done only once at the company or team level and is reused from project to project, with only minor modifications. The list of stakeholders will vary for every project, depending on who the stakeholders are, and may require minor tweaks to the communication types.

Table 7.1 lists the different communication types that are common across the industry along with examples of recurrences, owners, and distribution methods:

Type	Goal	Recurrence	Owner	Distribution
Stand-up	Day-to-day collaboration and unblocking progress	Daily	SDE Lead	Sprint board, in-person updates, or email progress updates
Status Update	Milestone-level project status	Weekly (*Every Tuesday*)	Project TPM	Email
Monthly Business Review (**MBR**)	Leadership-level program status with key insights relevant to leadership	Monthly (*3rd Wednesday of every month*)	TPM Lead	Meeting and Email
Quarterly Business Review (**QBR**)	Senior leadership-level program status	Quarterly (*3rd Wednesday of the 1st month per quarter*)	PM-T Lead	Meeting and Email

Table 7.1 – Example communication plan

The *names* of the communication types listed here may not be the same at every company, but the concepts exist. As called out in *Chapter 1*, there are some instances where the written status report isn't used at all; in those cases, the distribution method may not include sending an email, but the intent of the communication is largely the same. We'll cover each communication type in more detail in the upcoming sections.

Stand-up

The stand-up type is used in agile, and even non-agile, settings where regular syncs are needed – often with software developers. The exact protocols used depend on the methodology being practiced, so I won't go into detail on the specifics. However, in all cases, the takeaway for the TPM is knowing what the developers are actively engaged in and getting an up-to-date look at any blockers that they may have. Regardless of the presence of a *scrum master* or *development manager*, the TPM's role here is to facilitate unblocking the development team, either directly or indirectly, by getting the right people involved.

The stand-up can also help you craft the status updates as you have the most current task progress as well as projected work for some amount of time (sprint, week, or other, depending on development methodology). Depending on team culture or the number of projects you are driving, you may not attend the stand-up, but an update should be produced in some form, either through updating a sprint board, or a written status summary from the developers.

Status update

Most projects will have some form of a status update, either in a written format or via a status meeting. The purpose here is to give a health check on the project at the moment in time with some projection as to the overall timeline health. The focus is on current and upcoming milestones, blockers, and risks with a high-risk score (in other words, risks that have some chance of being realized in the near future). To this end, this type of report tends to be more technical and is meant for the stakeholders that are actively working on the project itself in order to keep them apprised of what is going on around them.

The ownership of the status update is on the TPM driving the project and will include input from all stakeholders (developers, managers, PM-Ts, and so on) that are actively engaged in the project.

Monthly business review (MBR)

The MBR is a status report whose audience is the leadership. As such, the focus isn't on the day-to-day work or issues but on the longer-term progress of the project, along with risks, issues, and milestones that are of higher impact on the project goal.

Beyond these basics, some MBRs contain learnings, highlights, lowlights, and other thought-leader types of information. Because of this, the format of an MBR tends to be company-specific, and sometimes leadership-specific, in the format and information that is expected to be included.

In all cases though, the intent is to update leadership on the longer-term and broader health of the project.

The distribution of an MBR is often in a meeting and may include an email depending on the format of the report that is used in the meeting. Some companies thrive on PowerPoint or similar slide-based tools to ingest this type of information that wouldn't be conducive to an email, while others utilize a written report that can be used in a meeting as well as sent out.

The MBR is usually owned by the TPM, although co-ownership with a development manager, PM, or PM-T is also a possibility. This depends on the company culture and what type of resources the project has. Your company may prefer the PM-T to own this, but if one isn't assigned to the project, it might fall back to the TPM to fill in. Filling in organizational gaps is part of what a TPM does best!

Quarterly business review (QBR)

The QBR builds on top of the MBR with a target audience of senior leadership. Not every project will have a need for a QBR, as it depends on the level of visibility the project has at that level in the company. If there is interest at the senior leadership level, a dedicated communication forum is needed to convey details with the right context.

The QBR combines all MBRs in the quarter (or the previous quarter, depending on timing) together into a single report with an emphasis on the end goals and major steps, victories, and misses in the time since the last QBR.

Communication timing

You may notice that each of these communication types builds upon the previous. The status report builds on the stand-ups, the MBR builds on the status report, and the QBR builds on the MBRs. As such, the timing of each is spaced out to allow for the collation of information into the next communication level up. In the example laid out in *Table 7.1*, the status reports are sent out on Tuesdays and the MBR is on the third Wednesday. So that third week allows for the weekly status report to be sent and then added into the MBR before it takes place. Since the QBR is a combination of previous MBRs, I run those on the same day of the month, and every third MBR will instead be a QBR.

This is just one way to handle timing and it will vary depending on project needs, and the behavior and expectations of your stakeholders. So, this will vary from project to project. However, I try to not send status updates on Fridays or Mondays. On Fridays, people will miss or gloss over the status before heading out for the long weekend, and in many countries, holidays land on a Monday, meaning you may not be in the office. Sending on Monday would mean that you would frequently have to send it on the next business day. Even if you stay on top of communication and let the stakeholders know, it will still feel inconsistent, so better to avoid it. These are personal guidelines that I feel are useful, but I have come across teams that require a specific day for status reports to be sent, so you'll need to take team expectations into account.

Now that we've defined some potential communication channels, we need to decide which stakeholders will participate in which communication channel. To do this, we need to determine who the stakeholders are.

Defining your stakeholders

Finding your stakeholders and defining a communication plan is often depicted as a one-time, early-on process. However, like most processes in project and program management, it's cyclical. Depending on the size of the project or company, doing this all upfront may be achievable in a single pass; however, as the project size and company size increase, this becomes something that you revisit often as new stakeholders are discovered. Here are a few ways you can go about finding out who your stakeholders are.

Requirements gathering and refinement is an opportunity to not only understand your requirements better but to determine who the requirements impact or involve, such as the owner of a service.

Talking with the stakeholders you've already identified will give you a different perspective on the requirements and impact. They might know of clients from their services that are impacted by the changes they must make that you aren't aware of. Anyone impacted by the project, no matter how far down the chain or seemingly tangential they may seem, is a stakeholder.

During *project execution*, especially during low-level designs, details may come to light that highlight a stakeholder that you weren't aware of. New developers may also come on and off the project, requiring updates to your list.

The more you drive clarity in your requirements, designs, and risks, the more clarity you'll simultaneously drive in who your stakeholders are.

Stakeholder list

In *Table 7.2*, I've drafted a partial stakeholder list for the *Mercury* program:

Name	Alias	Department	Project	Role	Comm Type
Josh Teter	jteter	Mercury	All	TPM	N/A
Arun Ardibeddi	aardibeddi	Windows Team	Windows Rollout	SDM	MBR
Danielle Wednesday	dwednesday	Windows Team	Windows Rollout	SDE Lead	Stand-up
Bob Belkan	bbelkan	Windows Team	Windows Rollout	SDE	Stand-up
Vicky Preston	vpreston	Desktop Devices Division	Windows, macOS, Linux	VP	QBR
Cassette Santoro	csantoro	Windows Team	Windows Rollout	TPM	MBR
Artem Danyluk	adanyluk	Linux Team	Subsystem	TPM	MBR

Table 7.2 – Example stakeholder list for the Mercury program

This stakeholder list gives me enough information to know who the stakeholders are and what role they play in the project. For my projects, I create stakeholder email distribution lists for various communication needs and use this stakeholder table to ensure the lists stay updated.

This table is a snapshot of the program-level and Windows rollout project-level stakeholders. For each stakeholder, I list which communication type they will be a part of. The table lists each project as well as the program so that each project TPM can see which stakeholders are important for them as well as how they may overlap with other projects.

For instance, *Vicky Preston* is the VP in the desktop devices division, and her involvement is spread across the three desktop projects. So, the Windows TPM, *Cassette Santoro*, knows that *Vicky Preston* would be interested in any risks in the Windows project that may also impact those other desktop operating systems. The point is that your stakeholders can vary depending on the situation, sometimes by level (VPs may not care about a task blocker due to permission issues), but also on the context of the problem at hand and how far-reaching the problem may be.

As people come and go from the project, requirements are refined, and designs are finalized, you will keep this table up to date to ensure the right people are informed and consulted throughout the project.

Roles and responsibilities

Now that we have our stakeholders, the last component of the communication plan is the roles and responsibilities. Understanding expectations from every stakeholder in the project is important to ensure work gets done smoothly. It also helps inform natural escalation paths if a task is delayed or otherwise in trouble. *Table 7.3* lists a portion of an example roles and responsibilities chart. Let's look at it in more detail:

Step ID	Name	TPM	SDM	PM-T	Lead SDE	SDE	Bus.
1	Requirements refinement	A	C	C	C	C	R
2	Project planning	R and A	C	C	C	C	C
3	High-level design	C	C	C	A	R	I
4	Low-level design	C	C	C	A	R	I
5	Sprint planning	R or A	R or A	I	C	C	I
6	Daily stand-ups	C	C	I	A	R	I
7	Status report	R and A	C	C	C	I	I
8	MBR	R	C	A	C	(I)	I

Table 7.3 – RACI chart

There are many different styles of a roles and responsibilities chart, and like the communication plan chart, this is often static as it is referring to the role of the stakeholder, not the actual stakeholder. The style of chart used in this table is also referred to as a **RACI** (pronounced *race-y*) chart. It's an acronym for the roles available: **responsible**, **accountable**, **consult**, and **inform**.

The *responsible* role is where the person or group is responsible for the step or activity. It does not mean that they are the sole contributors or owners but are the ones responsible for its delivery. For instance, the TPM may be responsible for delivering the MBR (*Step 8*) but will receive input from the rest of the project team.

The *accountable* role is similar to the responsible role, but the accountable party is who ultimately deals with the consequences of something not getting done. In this example chart, the *Lead SDE* is accountable for the daily stand-ups – a sort of scrum master role. They are where questions go if a stand-up does not occur or if there are questions or concerns. Another example to differentiate responsible and accountable is that the TPM is accountable for meeting a milestone, but the developers are responsible for doing the development to get to the milestone.

Tip on varying responsibilities

There can be cases where a single person is both responsible and accountable for a single deliverable. As *Table 7.3* shows, the TPM is often both responsible and accountable for the project plan as well as the status report.

In other cases, the responsibility may shift depending on the project size or team dynamics. In cases where a service team runs its own sprint, the SDM may be accountable for the deliverables and the TPM is responsible for requesting the right work gets done. In some cases, the TPM may be running the sprint and is thus both responsible and accountable.

The *consult* role identifies the different roles that should be involved in a step but are not accountable or responsible for its completion. For example, developers are consulted during sprint planning

because they help determine task estimation and capacity, but they do not own the sprint plan itself. A similar role, called *participate*, can take the place of consult or even leave both to differentiate a participating SME versus a non-SME.

The *inform* role is for all parties that only need to know of the outcome of a particular activity. In the example here, the business teams are informed about the designs, sprints, and work being done. The mechanism for informing can be the status report or meetings and does not need to be dedicated to each task.

If you have a large number of stakeholders that span many different disciplines and aspects of the project, an additional role of *optional* can be added, or using parenthesis around *Inform* or *Consult* to indicate *optional*, as you can see in *Step 8* for the SDE. This is in lieu of leaving the cell in the chart blank. It allows for a stakeholder to be optional for a particular activity where their active participation is not required but they may choose to participate if it makes sense in the particular situation.

Exploring the dos and don'ts for status reports

There are many types of communication that exist, but the most common and influential is the project status report. This may be in a written form, in a meeting, or via a slide deck, but the purpose and content should largely be the same across these formats. I'll be using a written format as it is common and the easiest to work with in a book, but the concepts are translatable to whichever format you or your company use.

First up, here's a quick inventory of what items should be included in a status report:

- **Executive summary**: This is a high-level summary of the progress that is easy to digest and clearly states the trajectory of the project. The traffic-light status is included here, as well as the items contributing to the current traffic-light color the project is on.
- **Project milestones**: There are two schools of thought on milestones. They are either a list of predefined phases or steps that every project goes through, such as implementation, testing, deployment, launch, and post-launch, or are used as shorthand for the deliverables of the project. I use the former as it provides good metrics that can be used across all projects (such as how often a project's deployment takes longer than expected) and separate this from the feature deliverables.
- **Feature deliverables**: This is a list of deliverables, usually synonymous with the top-level tasks (also called **epic tasks**) in a task list. This lets the stakeholders know when to expect a specific feature or group of features and varies with every project.
- **Open and recently resolved issues**: This is a list of issues that are currently being tracked. These are separate from the known tasks and the known risks but will include realized risks as they are currently being dealt with. I usually have a separate table where I move resolved items once they have been shown during a single status report as being resolved to reduce the size of individual tables.

- **Risk log**: This is a log of the risks as discussed in *Chapter 6*. If the number of risks is large, I usually just include the high-risk items in the status report with a link to the full risk log.
- **Contact information**: In case of issues or concerns, a reader of the status needs to know who to contact.
- **Project description**: Not everyone that is a stakeholder may remember what the project is about, especially in large organizations and email distribution lists, so including a blurb on the project helps set the stage.
- **A glossary of terms**: This won't make or break a status report, but if it is included, you will receive praise. Tech companies are large, and the number of **three-letter acronyms** (**TLAs**) that are in use can quickly become daunting. Just as not everyone will know the project, not everyone knows all of the terms being used. Be kind and don't assume.

Now that we know the basic elements that should go into a status report, here's a list of things to consider that turn an okay status report into a great one:

- Every action must have an owner and date
- Define your traffic light and stick to it
- A non-green status must have a path to green
- Keep important details above the fold
- Format and grammar matter

Let's go through each of these in detail.

Every action must have an owner and date

The number one rule for any status is that for every action, an owner and date must be identified. This helps ensure the right people are accountable for actions and that the stakeholders know when to expect a resolution. This conveys that you, as the TPM, are in control of the situation and are proactively engaged in all issues.

In some cases, an action item may not have a date at the time of a status report; in these cases, a date should still be supplied as a **date for a date** (**DFD**). This lets people know when to expect a concrete plan of action.

Define your traffic light and stick to it

A traffic light, in this case, is a convention of using red, yellow, and green to denote different conditions of your project. It's easy to understand as the meaning of an actual traffic light closely matches how it is used in project management. That being said, every company is different, and even within a company, the thoughts behind how each should be used can be different. With different cultures as well as people originally from different companies involved, preconceived notions can cause confusion. For

this reason, I define my usage of the traffic light status to make it clear why a project is a particular status. Let's look at the definitions I use.

Green

The project is meeting current milestones and is on track to deliver *on schedule*. Some companies may need to include *on budget* in this status. So long as the currently agreed upon resourcing and scoping can get the project to the scheduled date, I consider it green.

Yellow

There are active issues that may impact the ability to deliver on schedule. It is not definite that the date will be missed at this stage, but it requires changes to get back on track.

For a *yellow* status, there is ambiguity still – it is a cautionary status and lets your stakeholders know that you are on top of the situation and being proactive to keep the project from going *red*.

As a note, the issues here should rarely be new or unexpected. The project risk log should include all risks that may become realized. Ideally, the *yellow* status will be short-lived as risks have defined strategies to deal with them.

Life does happen, and things come up, but keep this ideal state in mind so that when unexpected issues come up, you can have a retrospective to see whether they could have been expected, and then add them to your risk register.

Red

A *red* status means there is no current way to meet the scheduled delivery with the current resourcing and defined work. This is the status that I've seen the most contention in its definition in my career. If a project, left unchanged, cannot meet the due date, it is *red*. This does not mean that the date is impossible to meet but there is *no ambiguity* at this point that something must change.

Contentious topic – When to change the status color

As a TPM, my preference is always transparency on status, even when it makes you or another stakeholder look bad. It's better to be honest and upfront than to sow distrust by reporting a **watermelon status** (*green on the outside, but red on the inside*)!

There are also cases where an event occurs, such as a risk being realized, where it may not be perfectly clear as to whether a status should change. I try and use the status definitions as my guiding principle: if a risk was successfully mitigated and didn't impact your ability to deliver, then the status can stay green (or whatever status you are on, as it does not shift the status).

Sometimes a problem occurs and is resolved in-between status reports, but a stakeholder may wish to change the status to flag that an issue happened. This can get very political and may require some negotiation but if you use transparency as your goal, you will find a path that works for your circumstance.

A non-green status must have a path to green

Both *yellow* and *red* statuses must have a **path to green** (**PTG**). This is the set of actions that need to take place in order to get a project back on track. Just as with every other action, these require an owner and a date.

For a *yellow* status, the PTG is often well understood with no ambiguity, which is why the project is *yellow* and not *red*. However, there are cases where the issue's impact isn't well understood and requires some investigation to determine the impact and the right path forward. In these cases, yellow may be more appropriate because it isn't yet clear whether there is a path forward. Also, DFD may be required instead of a date because the action is pending investigation. It may be tempting to put the date of the investigation's conclusion, but the date should reference when the status will shift to green, which isn't known.

For a *red* status, the PTG is often exploratory. Most paths look to understand the impact so that the right action can be determined, first focusing on scope and resourcing. You might look at adding more resources to reduce calendar time, you might de-scope or reprioritize requirements to different milestones or phases of a project. Worst case, moving the due date may be the only way to get a project on track. What this looks like depends on your company and the ability for the project's due date to change. External drivers like a customer promise or regulation may make that option not feasible.

Keep important details above the fold

Above the fold is a concept in media that refers to information that is immediately available to the reader. The term comes from the printed newspaper where the page is folded in half on the newsstand. This made the top half of the front page visible at a glance, so this was reserved for the most important headline. The practice is still in use in newspapers today and has moved on to digital media where the content available without needing to scroll is considered to be *above the fold*.

The reasoning behind the delineation is that the content above the fold is the only content that is guaranteed to reach your audience. The concept is often applied to executive leadership at a company where the general consensus is that leadership will read the content above the fold and then stop reading. Even in this case, there is no full guarantee that your stakeholders will receive the information, or read it, but this puts the most critical information at the top and increases the chance of it being seen.

This means that your most important information – project status, PTG, and summary of status – must always be first and foremost in a status. Senior leadership may not need to know all of the details of how something is being accomplished, but they need to know that the project team is being proactive and has the situation under control.

Once you are below the fold, the information becomes more granular, with updates on the sprint schedule, project burndown, and risk log. This is where the team leads and development managers will get the most value in knowing what the day-to-day is like in the project.

At the bottom, you place information that is static or near static, such as the project description, project contact information, and the communication plan. The communication plan will technically change every status, as it conveys when the next status will be sent and may mention related communication dates for project MBRs, QBRs, and a stand-up schedule. Even so, I usually have this at the bottom as it is expected information and not pertinent to the status of the project. However, for critical projects, I often repeat the next status date above the fold as well.

Format and grammar matter

A consistent format that is easy to follow and understand is important. The more time that the reader has to spend in understanding what you are trying to say, the less they may spend reading in the future and just send out emails or meetings to discuss the status. When working with new stakeholders, impromptu meetings may occur as they don't understand your writing style or the full context of the project. They may simply have a different style or way to consume information. *Listen to your stakeholders* and work to find ways to format your status to make your stakeholders comfortable with how the status is presented – it will save a lot of time during moments of high-stress movement in the project to not have to go over misunderstandings because the status format didn't make sense. In the same vein, grammar matters – to an extent. I'm not suggesting that perfect Honors-English-level grammar is required, but clear and concise sentences and correct word choice can also improve comprehension among stakeholders.

Pro tip – Format ahead of time

Once I have built my project plan and found my stakeholders, I spend some time working on the format of my communications. In the same way that you line up communication schedules to have an easy flow of information, you can format your status reports to make it easier to refresh things such as milestone tables and current issues and risks. Spending time upfront to find a format that reduces day-to-day work can quickly add up in times of high churn in a project and allow you to put your effort towards more critical work.

The following is an example status report, broken up into two pictures: *Figure 7.1* with information that is *above the fold*, and *Figure 7.2* with information that is *below the fold*:

Windows Rollout Project Status: Feb-21-22

Next Status: *Feb-28-22*

Executive Summary

Status: **Yellow** for *Jun-5-22 launch*

Summary: A delay in the Text Message API definition as caused a day-for-day slip of starting the User Profile Object work. With project buffer and the early stage of the project, this is expected to be recoverable.

Path to Green: Cassette Santoro to work with Danielle's SDM to move her upcoming on-call rotation to later in the year to make up lost time. This will add buffer back into the project lost by the delay. **ETA Feb-23-22.**

Risk Log

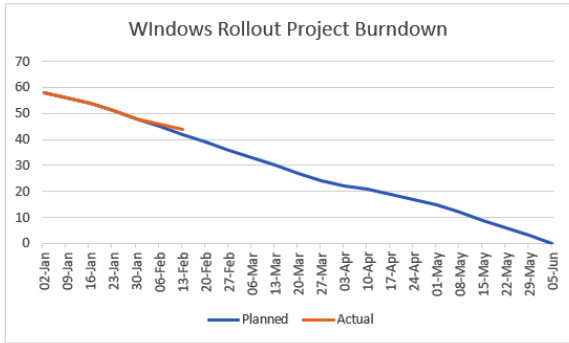
Id	Risk	Probability	Impact	Strategy
1	Cross-platform tooling issues	High	High	**Acceptance:** Shift timelines to account for delays **Mitigation:** training and crashing
1.1	New Integrated Desktop Environment (IDE)	High	High	**Mitigation:** training on new IDE
1.2	New Coding Language for some teams	High	High	**Mitigation:** training on new language
1.3	Cross-team collaboration	High	Medium	**Mitigation:** daily stand ups, co-location, or chat rooms

Figure 7.1 – Status report above the fold

The information above the fold is the most pertinent to understanding the state of the project and no additional information is needed. This includes the status color and target date for the status, the summary of where the project is at, and the PTG. Since the project isn't green, the status conveys why the status is yellow and the PTG with an owner – Cassette Santoro as the project TPM – and the date, February 23rd, 2022. The risk log doesn't always have to be above the fold, but in this case, it conveys high-risk items that can be relevant for upcoming status reports that shift back to yellow based on these risks. If other more important things are going on in the project, they'll take priority here.

Figure 7.2 shows the remainder of the status report and represents the data below the fold:

Project Burndown



Project Contacts

TPM: Cassette Santoro (@csantoro)
SDM: Arun Ardibeddi (@aardibeddi)
SDE Lead: Danielle Wednesday (@dwednesday)

Communication Schedule

Status Archive: Link

Next Status: Feb-28-22

Next MBR: Mar-16-22

Next QBR: Apr-20-22

Figure 7.2 – Status report below the fold

In this example, I chose a project burndown to convey the week-by-week status of the project. The dip in *actual versus planned* here coincides with the delay called out in this report. Over time, it will act as a visual reminder of the delays incurred. A milestone or feature list table can also be useful here. The **Project Contacts** section is static in this case and gives context as to who to reach out to in the project. Lastly, the communication schedule lists the next weekly status, the next MBR, and the next QBR. These dates all follow the dates laid out in the communication plan in *Table 7.1*.

We've explored the artifacts used in stakeholder management and how and when to use them, and explored the content of a good status report. Next, we'll see how managing stakeholders in a tech world can offer up unique challenges.

Managing stakeholders in a technology landscape

Most of how you manage stakeholders is not unique – all disciplines need a stakeholder list, a communication plan, and roles and responsibilities. However, there are some aspects to working in a technology company that can add some additional constraints to how we move forward. We'll take a look at three aspects of how managing stakeholders can be different while working in a tech environment:

- Communication systems

- Tooling
- Technical and non-technical stakeholders

Communication systems

Working for a technology company means you get to work with cutting-edge software; however, this can also be a burden. During the pandemic, many companies scrambled to update and add additional ways to communicate while workers were remote. Companies have since transitioned into a seemingly permanent hybrid model of working from home and the office, and it's hovering around the 50/50 ratio – at least in the US.

With all of these rapid changes from fully in-person to fully remote, and now hybrid, the technology landscape of available ways to communicate has grown with a myriad of messaging systems such as Slack, Microsoft Teams, Amazon Chime, and Zoom, as well as other productivity tools such as Quip, OneNote, and Evernote. In some cases, a single company may be using multiple systems that accomplish the same or overlapping purposes. This may mean that not everyone is at the same level as to what forms of communication work for them or their team. So, adding to the list of stakeholders, their roles, and which communication types are appropriate for them, you may find that you need to track what systems are available for each stakeholder.

If your company follows a standard here, then this can be a one-time exercise, but if you are like many tech companies right now, knowing which tools to use can save a lot of churn as you move from phase to phase in the project.

Tooling

When using tools to manage a project or program, you'll want to look into how a given tool can help in distributing the various communication plans that you need. Do this as early on in the project as you can to ensure its benefits help you as early as possible. Some tools offer built-in dashboards that can be used to build a real-time status of a project showing upcoming milestones, feature dates, overall status, and more. These can be great tools by reducing the need to craft a status, but it does mean that status is immediately and always available. If something is going wrong in a project, being able to control how the update is consumed by stakeholders can be a valuable asset to help manage expectations and keep panic and escalations at bay.

If you do use a tool though, you'll also need to verify permissions to ensure that everyone you expect to have access does, in fact, have access. On a related note, you'll also want to check default permissions in the cases where there are non-stakeholders that shouldn't have access to ensure they cannot get in.

Technical versus non-technical stakeholders

In the tech world, you are building technology, but you are also in a business that needs to make money. As such, you will run into plenty of non-technical teams that are your stakeholders. In most

cases, there is a business team that is driving the work being done. This highlights your superpower of the communication bridge between these worlds. If you have a lot of non-technical stakeholders, an improvised communication plan may be needed to ensure you have a format that will fit their understanding.

To do this well, you need to know your audience while drafting a particular communication. The issues you need to go over might be highly technical, but they also need to be distilled to a level of simplification so that non-technical people will be able to grasp the issue at hand. I have found that the exercise of re-articulating a problem for a different audience can bring more clarity to the problem as it forces you to look at it from a different perspective.

Exploring the differences between a project and a program

For stakeholder management and communication plans, as with most key areas of management, the difference between managing a project and a program is about scale. The number of stakeholders increases and the number of concurrent communication plans that are being utilized increases. Instead of a single stand-up, you may have at least one per project, the same goes for status reports and possibly MBRs, depending on the complexity of the project.

This isn't to say that you, as the TPM running the program, are directly responsible for each of these communications, but you are accountable to them. If a project is falling behind on statuses, or the TPM isn't meeting with stakeholders enough to keep them in the loop, as the program manager, you are accountable to keep the project TPM on track. Luckily, there are some additional tools you can use to help keep the program's stakeholders happy and communications flowing freely.

Scheduling for natural accountability

In *Table 7.1*, I stepped out the communication schedule to allow lower-level communications, such as a weekly status report, to flow into the MBR meeting. It reduces the time running around to get up-to-date information. The same concept can be used at a program level; the level of status you care about will just start out at a higher level. Ensure each project's MBR is published or takes place prior to the MBR for the program. If the project doesn't need that level of visibility, then just ensure the weekly status reports are published on a day of the week that is earlier than the program MBR to give you adequate time to collate information. This may lead to a case where an issue's status has changed between the status report and the MBR, in which case, updating the appropriate stakeholders out-of-band is important to ensure that there are no unexpected updates.

The same can be done at the program level, as the program's status report will include details from each project, and ensuring the project status reports are sent out prior to the program will ensure a natural flow of information.

Leadership syncs

In a project, there are multiple ways you get information from daily stand-ups with your development team, to project management syncs including development managers, PM-Ts, and key stakeholders. At the program level, you still need avenues of information, but the information you need will usually be at a higher level. As you will have multiple project teams to work with, a leadership sync – akin to a scrum of scrums, where the scrum masters from various scrums teams meet to discuss each scrum's sprint – is a concise way to get project information passed on. Bringing each project TPM into a combined forum not only keeps you informed but gives each TPM a chance to understand what else is going on in the program.

The sync should flow very similarly to a sprint stand-up. Going round robin from TPM to TPM to get a quick status on the project including any blockers or concerns is all that is needed. No doubt, if there were major issues, you would already be aware prior to this sync, so it's really to ensure everyone is aware of the high-level progress being made as well as to ensure all cross-team impacts are well understood by everyone.

Projects tend to use the weekly status report and MBRs as their main ways to communicate with a wider audience. With a program, the QBR will become more prevalent as the impact of a program will be high enough to warrant meeting with senior leadership. This may mean that of all of the communication plans I listed in this chapter, all of them will be utilized when working on a program.

Summary

In this chapter, we covered the artifacts used in stakeholder management such as the communication plan, stakeholder list, and roles and responsibilities. We looked at how the different communication types allow you to convey information in a specific way based on the needs of your stakeholders. We also discussed how you can discover who your stakeholders are.

We learned what elements go into making a good status report and how clear and concise language and status definitions reduce churn during the project when trying to convey a status.

We discussed the types of challenges you can face in the tech world with a highly diverse set of communication tools, along with the varying levels of technically minded stakeholders you need to communicate with.

Lastly, we explored how stakeholder management differs at the program level and how to utilize leadership syncs to collate information across multiple projects.

In *Chapter 8*, we'll wrap up the section on program and project management core principles by discussing how to effectively manage a program. We'll discuss when and how to define a program and how to track the program across all open projects.

8

Managing a Program

In this chapter, we'll discuss in more depth how to manage a program. As discussed earlier in this book, program management builds on the foundations of project management by utilizing the same key management areas. However, there are some differences due to the larger scope and broader impact that a program has compared to a project within it, so both of these concepts play a substantial role in defining the differences in management techniques.

We'll explore how to manage a program by doing the following:

- Driving clarity at the program level
- Deciding when to build a program
- Tracking a program

Let's begin!

Driving clarity at the program level

In order to drive clarity at the program level, we need to understand what sets a program apart from a project: scope and impact. Let's look at what each of these means in the context of program management:

- **Scope**: The scope of a program or project is the set of requirements that will be met by the goals. It's easy to dismiss this as being the same as the full set of requirements that are given, but that isn't always the case. Going back to the project management triangle in *Chapter 2*, in *Figure 2.3*, both time and resources can impact the scope. It is a negotiation process between the TPM and the stakeholders as to what the right balance is going to be. Other factors, such as technical feasibility, may also hinder the ability to meet a requirement. This is especially true in cases where those setting the requirements are not technically inclined; they may ask for a feature that sounds feasible but is not something that can be achieved.
- **Impact**: This refers to the set of stakeholders and systems that are affected by achieving the goals of the program or project. Every service, application, or device that the scope will involve

is affected by the changes. Going beyond that, any client of those same systems is also affected and must be considered when making the changes. These clients may be internal as well as external, such as users.

Figure 8.1 uses the *Mercury* program to illustrate where the scope and impact exist.

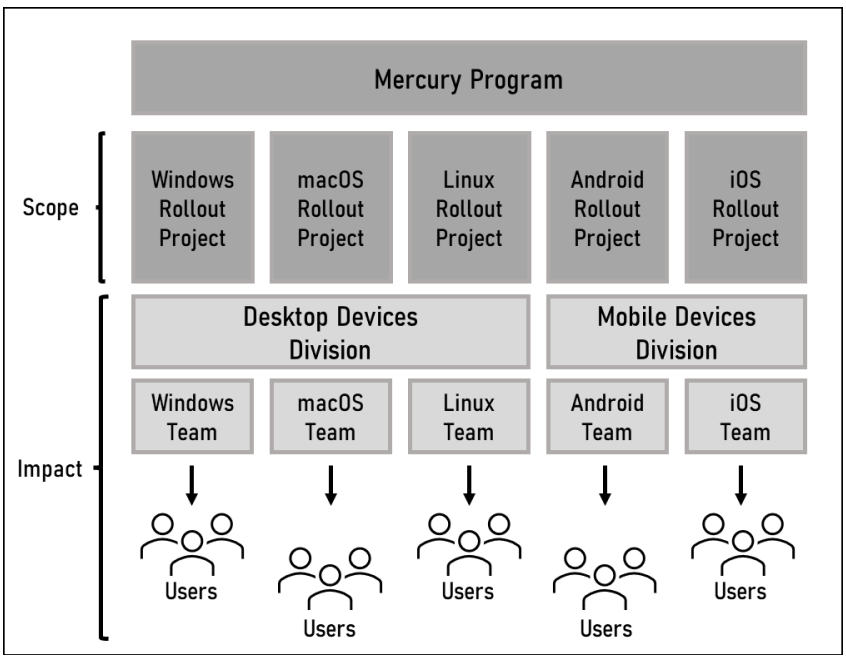


Figure 8.1 – Scope versus impact

The scope is the combined work of the projects within the program; this is essentially the five operating system applications. For impact, this includes both the internal teams as well as external users. For example, in the Windows rollout project, the stakeholders are the desktop devices division management, the Windows team itself, as well as the expected user base for the Windows client. The mobile devices team is impacted by the Android and iOS rollout projects, but this does not affect the Windows team or Windows users, so there is no impact on those groups. Though Windows users might also have smartphones, and thus also care about iOS or Android releases, the Windows and smartphone releases won't intersect – they are by and large users in multiple groups. In contrast, the *Mercury* program is impacted by every stakeholder from every project.

In some organizations, the impact may also be measured by a dollar amount. You may hear colleagues referring to themselves as managing a $2 million program or portfolio, though arguably, the dollar amount has a direct relation to the number of users and stakeholders impacted as well.

Understanding the difference between scope and impact will help us define proper boundaries in both programs and projects.

Defining boundaries

The boundaries of a project or program are simply its goals and, by extension, the requirements that it will deliver. For a standalone project or program, these are just the goals themselves. For projects within a program, however, it's a little more nuanced. Defining the boundaries of a project or program can help ensure scope creep is noticed and that the designs that fit the requirements stay within the bounds of the project, and help you see which projects should be part of a particular program. Without a clear boundary, a program can stray from its purpose.

To understand how to define the boundaries within a program, let's first talk about how we set the boundaries within a project. At this level, it's fairly straightforward – we define milestones and features. As discussed in *Chapter 5*, *Plan Management*, milestones are often the same across projects and break the project up into phases, such as design completion and user acceptance testing completion. The features are where some work is done as they are specific to the requirements. Looking at the Windows rollout project feature list in *Chapter 5*, in *Table 5.8*, you can see that there are several self-contained deliverables, such as the user profile object and presence object. Each of these features groups together a set of requirements into a cohesive deliverable.

This same methodology is used at the program level, where a feature is represented by a project instead. Looking at the *Mercury* program as a whole, the goal is to make the *Mercury* application available to 90% of users across all platforms. As discussed in *Chapter 3*, *Introduction to Program Management*, an easy way to break this apart is by drawing boundaries around each operating system. Each project would then have a well-defined boundary within a single operating system but would also line up well with the stakeholder boundaries of the Mercury company.

From a feature perspective, there is an additional boundary that can be created that further restricts the platform scopes that we introduced by carving out the P2P subsystem as a separate project. Again, this has a well-defined boundary of delivering the P2P functionality in a consumable library by the operating system teams. In this way, each project is a feature, albeit a large one.

Now that we've learned how to drive clarity at the program level using the scope and impact to define boundaries, we'll use this knowledge to discuss when and how to define a program.

Deciding when to build a program

We know what a program and project are, but this has all been with the assumption of a pre-existing program. As a TPM, one of your jobs will be deciding when a program needs to be created – when it is appropriate to the needs of your organization to do so.

Knowing when to create a project is straightforward – you receive requirements to deliver a new application or service, a new feature, and so on, and you create a project to deliver on those requirements.

However, knowing when a program should exist is a bit more nuanced and depends on the situation. There are two avenues in which a program is created: from the beginning when requirements are given and during project executions where a need arises. Let's explore both scenarios.

Building from the start

Deciding to form a management program around a set of requirements is how most people perceive program formation as it fits the form that project creation takes and thus makes sense. To make this decision, you need to first clarify the requirements as this process will require analyzing the requirements closely to define appropriate boundaries.

Chapter 5, *Plan Management*, went in depth into driving clarity in requirements, so I won't repeat much here in that regard. The biggest difference between driving clarity in requirements for an existing project compared to before project and program formation is that your clarity helps you categorize the requirements. You are looking for patterns or deliverable chunks of requirements – much like determining features in a project.

When examining requirements, you are looking for scope and impact cues that may serve as logical delineations between multiple projects to deliver on the requirements you have. *Once a set of requirements is broken out into more than one project, a program is needed to ensure the full requirements are met.*

Using the *Mercury* program and project list as an example, *Figure 8.2* illustrates a typical project and program boundary.

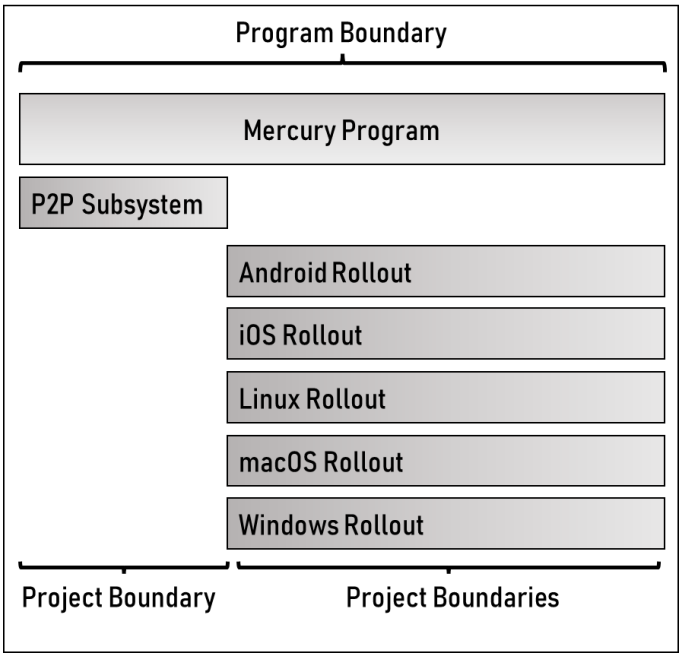


Figure 8.2 – Program versus project boundary

The *Mercury* program goal requires that the P2P application is available to 90% of the user base. In *Chapter 3*, *Introduction to Program Management*, we examined the ways of achieving this and concluded that making the application available on all five operating systems was the best approach. Looking at the requirements with this in mind, a natural delineation could be to carve the requirements apart by the operating system. Though these could have been treated as features in a single project, the stakeholders were different for each operating system, and thus the impact on these stakeholders rarely overlapped. The impact cues are what help us decide that creating multiple projects is the right way forward. As there is more than one project, a program is also needed to ensure that these projects keep the program goal in focus, as well as facilitating cross-project collaboration.

Had we decided to just deliver to the desktop operating systems, as an example, then a single project with multiple feature deliveries may have been appropriate as the stakeholders all roll up within the same desktop devices division within the *Mercury* company. So, the combination of what to deliver to meet the program goal requirements and the organizational structure of the company helps form the decision to build a program. Each company has its own organizational structure, so whether or not a program makes sense for you depends on your company as well as the requirements you are working with. The only hard and fast rule comes down to having multiple projects to get the requirements met.

Though building a program from the beginning is natural, it is not the only way a program may need to be formed. Let's explore what types of scenarios will lead to creating a program mid-execution.

Constructing a program mid-execution

When I say mid-execution, I'm referring to any point after one or more projects are in-flight and a program is created to encompass them. *Chapter 4*, *Driving toward Clarity*, introduced the perpetual skill of driving clarity, and this goes beyond the requirements and issues at hand. TPMs are adept at looking at the bigger picture and piecing together when one project may impact another. It is our job to ensure the health of our projects and programs across our organization.

In this constant state of analyzing, a pattern much like the one you look for when given new requirements may emerge. As an example, my current team is working on several projects that impact customer experience. Each project came to life separately out of different needs and different stakeholders; however, looking at the requirements for each project, a common end goal could be discerned relating to the ultimate end state for the customer experience. I did an exercise where I combined the projects into a program to see how they would fit, and this led to realizing that one of our unfunded projects fit into the program to close it out and therefore had more value than initially thought. This exercise ended with creating the program and funding the last project.

Without the program, the projects would have been successful on their own, but the program brings cohesion and purpose to the group of projects and helps uncover value that wasn't obvious on the surface. This will lead to a better outcome in aggregate than each project could deliver on its own.

Figure 8.3 illustrates the construction of a program from existing projects.

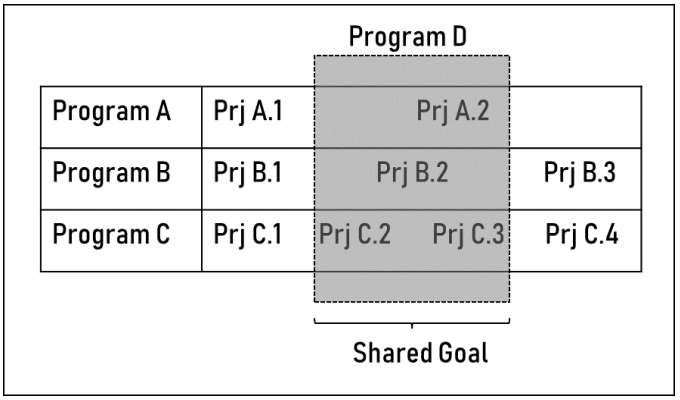


Figure 8.3 – Defining an in situ program

In this figure, multiple in-flight projects, each part of their own program, are combined into a program to align their goals to a shared desired end state. In this case, **Program D** is the new program created to encapsulate projects **A.2**, **B.2**, **C.2**, and **C.3**. The program is temporary and will end prior to **Program B** and **Program C** concluding.

This brings up an interesting concept of a project existing within multiple programs. If project management tools are to be believed, this should not be allowed, but there is value in allowing overlap as the overlap highlights different aspects of that project's goals and helps ensure the wider organization is moving together in the same direction while ensuring singular goals are also achieved. In a sense, this overlap is natural and feeds into the TPM's ability to see both the forest and the trees, and not get stuck in one or the other.

Now that we've driven clarity at the program level and learned when to create a program, we'll look at the different aspects of tracking a program.

Tracking a program

A program has the same key management areas as a project and each one has additional aspects to consider given the larger scope and impact. We'll touch on each of these key management areas:

- Program planning
- Risk management
- Stakeholder management

Program planning

Regardless of how the program came to be, the key requirements in the planning stage remain the same. The reason that a program exists is to facilitate multiple projects in achieving a common set of goals. The most important step for a program is to define a common set of goals.

If the program is being created from the beginning and no project has started, this is a straightforward task as the requirements as a whole will tell you what the end goals will be. Each project would then take a subset of these requirements to form its own goals. In the case of the Mercury program, the requirements stated that the Mercury application needed to be available to 90% of the user base. This became the goal of the program, and the projects took the subset of requirements needed for their given operating system.

In the case where the program is being formed from projects that are in flight, this exercise can be a bit more abstract. This is because it is easy to assign items into known categories and yet hard to determine what categories should be given those same items.

Let's pretend that the *Mercury* program doesn't exist, and instead, each operating system team is independently creating a P2P messaging app. Each application will look very different in terms of the user interface as each operating system has different conventions to adhere to. Looking at each one individually, it's harder to see where a cross-platform P2P subsystem could be utilized because you are looking at the application as an isolated idea. However, if the potential opportunity for de-duplication of effort and assured cross-platform capability is seen, then a program goal can be created from this opportunity.

It may be that the program goal is written in a way to make this preferred synergy part of the goal and states that a common network layer component is part of the end state. The result of looking at the program as a combination of existing projects compared to a goal split into projects can change the nature of the goal to ensure alignment and understanding, though the end state will be the same.

Now that we've planned the program, we need to look at how to manage risk.

Risk management

In *Chapter 6*, *Risk Management*, we looked in depth at risk management practices and touched on how risk assessment and management at the program level is about a difference in perspective. The program will focus on cross-project dependencies and the risks involved, whereas the project will focus on cross-task dependencies as well as risks associated with a given task itself.

At the program level, we need to examine how a project's deliverables may overlap with another project's deliverables. These intersections will always include some risk. *Chapter 6*, in *Figure 6.3*, talked specifically about the risk of project delays due to the subsystem component that was required for each operating system project to complete. Aside from scheduling risks, there may be inconsistencies in APIs and behaviors of those APIs from the operating system to the operating system. The intersection of the subsystem with all five operating system projects represents a risk that the subsystem API capabilities

don't align with the needs of a particular operating system. Any change to the API contract can lead to issues in the other operating systems, which could lead to a cyclical risk given the high number of touchpoints. This is certainly solvable as this isn't the first application to have a shared API across multiple systems, but the risks are still present and need to be paid attention to. For cross-project dependencies such as these, the TPM running the program would own the risk and ensure that the risk strategies are followed within the impacted projects.

Some risks that exist at the project level may be important enough to be tracked at the program level as well. In these cases, the owner may still be at the project level, but the visibility of the risk is elevated to the program level and usually a larger set of stakeholders. Ensuring the risk strategies are followed still falls on the project TPM but will require scrutiny from the program TPM due to the high-profile nature of the risk. As an example, the Windows rollout project may have a risk that impacts their ability to enter into cross-platform testing on time. Though this is solidly a project risk based on task slippage, it has the potential to impact the other projects' ability to finish cross-platform testing. As such, the program TPM will report on the risk to the program stakeholders – which includes all operating system teams and divisional management. The Windows rollout project TPM, *Cassette*, would own the risk, with the program TPM watching closely and providing help as needed.

Sharing risks from the project to the program level and aligning program goals across projects require strong communication strategies. This additional overhead of a program coordinating across projects adds risks related to dependency delays and longer design times to allow for syncing across teams. To reduce the impact of these risks, let's look at stakeholder management to see how we can provide a strong communication foundation for the program.

Stakeholder management

The role of the program TPM is to set the projects up for success. They are there to ensure that each project TPM understands the role of their project in relation to the program. As a project TPM, they should already understand the goal of their project but may not realize how it fits into the larger picture of the program. The program TPM also needs to ensure that the stakeholders are aware of all of the projects and the interdependencies that may concern them. To accomplish these tasks, there are multiple ways to engage your stakeholders throughout the program to offer proper guidance. We'll explore each in the following subsections.

Kickoff

Once a program is created, either at the beginning along with the associated projects, or in flight, a **kickoff** meeting is needed. A kickoff meeting is where all stakeholders come together to discuss the program and its goals, as well as the role each stakeholder will play in the management of the program. Just like at a project kickoff, this is where the communication plan is shared with the program team, including expected deliverables used in the communication plan.

The kickoff sets the stage to ensure that everyone is aware of the end goal of the program and not just their respective project goals. Each project lead gets an idea of how their project may impact other

projects in the program and each stakeholder has a chance to raise concerns and point out potential new stakeholders.

Leadership syncs

In *Chapter 7*, *Stakeholder Management*, we discussed the concept of leadership syncs to help coordinate across the projects. Not all programs will necessarily require the use of leadership syncs, but in cases where cross-project risks are involved, this is a valuable line of communication to get up-to-date information on the risks the program is tracking.

The leadership sync should flow very similarly to a sprint stand-up. This means going **round-robin** from TPM to TPM to get a quick status on each project, including any blockers or concerns. This continues the theme of the kickoff by not only informing you as the program TPM of the current status but also ensuring all project TPMs are aware of what each other is doing. If a new major issue is surfaced, address it after the round-robin is complete so those not impacted may leave. This concept, known as the **parking lot**, is also used in sprint stand-ups to ensure a quick and efficient use of everyone's time.

Roles and responsibilities

As covered in *Chapter 7*, every project should have its own roles and responsibilities chart. In most cases, it is the same for every project. This is also true at the program level. *Table 8.1* is a RACI chart for the Mercury program:

Step ID	Name	Program TPM	PM-T	Project TPM	Business
1	Program Planning	A(/R)	C	C	(R)
2	Project Status Report	A	C	R	C
3	Program Quarterly Business Review (QBR)	R/A	C	C	I

Table 8.1 – Program-level roles and responsibilities

The roles here are usually less than at the project level as there isn't as much day-to-day work involved to track. However, it is still important for everyone to know their role in the management of the program. The program TPM is accountable for the program plan and depending on the involvement of the business, the responsibility may be with them, or on the TPM as well. This also outlines the project status reports and lists the project TPMs as responsible. Since a program exists, the accountability is with the program TPM. Lastly, the program's QBR is owned by the program TPM and the project TPMs are consulted. Though this may seem obvious, it sets up the communication strategy for the program.

Communication strategies

Projects tend to use the weekly status report and **Montly Business Reviwes** (**MBR**) as their main ways to communicate with a wider audience. With a program, the QBR will become more prevalent

as the impact of a program will be high enough to warrant meeting with senior leadership. This may mean that of all of the communication plans listed in *Chapter 7*, all of them will be utilized when working on a program.

As the program QBR builds on the project status reports, ensuring that every status report is on time and that the schedule of reports is set up properly falls on the program TPM. *Figure 8.4* illustrates a reporting timeline for the *Mercury* program that allows for timely feedback to the program TPM.

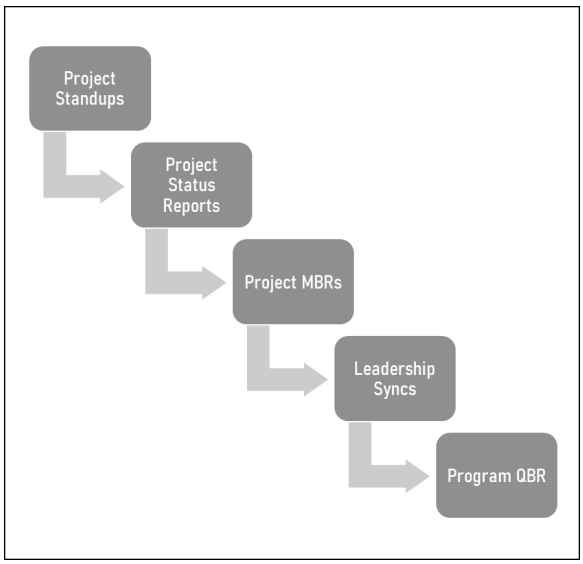


Figure 8.4 – Aligning communication

As mentioned in *Chapter 7*, *Stakeholder Management*, the key to effortless communication is setting up a schedule that naturally feeds the right information into the right reports at the right time. Here, all project weekly reports are due prior to the QBR so that the QBR can have the most up-to-date information to draw from. Any clarifications needed can come about during leadership syncs or directly if urgent enough.

We've discussed the ways in which we can engage with our stakeholders, and how to pass along information. Next, let's dive into when the program TPM may need to dive a bit deeper into a problem.

The art of intervention

The program TPM is a force multiplier to the project TPMs, just as the project TPMs are a force multiplier to the project team. In most cases, project-level issues are delegated to the project TPM and the program TPM is informed or consulted if necessary. However, cases may arise where more involvement from the program TPM is necessary.

The program TPM is accountable for everything in the program, and this is especially true for project risks that impact the program. If a risk is close to being realized, and thus has a high-risk score, or a risk has already been realized and risk strategies are underway, closer involvement of the program TPM is warranted. This is to facilitate faster communication and not a means to take control of the project TPM. Attending project-level stand-ups to understand the day-to-day or even hour-by-hour concerns will reduce communication times and allow the program TPM to coordinate cross-projects faster if the need arises.

To facilitate quick intervention, having the project sync on your calendar will allow you to easily drop in when needed and reduce the churn of getting the invite while an issue is at hand.

Intervention should rarely be a case of taking control, but rather offering a lending hand to resolve the issue more rapidly and help identify cross-project impacts and ensure good communication with all impacted stakeholders. If cross-project communication is not required, or the issue isn't being reported at the program level, intervention is not required. That is the job of the project TPM.

Summary

In this chapter, we learned that scope is the set of requirements, and the impact is the group of people affected by the scope. We learned how to drive clarity at the program level by discerning the scope and impact of the requirements.

We then discussed how to use the scope and impact to define boundaries around the requirements. The boundaries can make up both the boundaries of a project and the deliverables within that project. This brings shape to the program.

After that, we learned that a program should be created when sufficient impact and scope dictate that multiple projects are needed to complete the goals. We also discussed how seeing patterns and goals across existing projects may align and warrant the creation of a program to track them as a whole.

Lastly, we discussed various aspects of tracking a program through program planning, risk management, and stakeholder management. Each of these key management areas builds on the management styles used at the project level but with increased scope and impact, necessitating some change in strategies. This includes planning the composition of the program as well as the program goals and deciding which risks need to be tracked at the program level. We also dove into the various communication techniques used in managing stakeholders, such as a kickoff meeting and leadership syncs, all used to facilitate the success of the projects in the program.

In *Chapter 9*, we'll focus on the career paths available to a TPM that we first introduced in *Chapter 1*. We'll cover a story of both a traditional career path and an unconventional career path from fellow TPMs in the tech industry.

We'll then dive into more specifics on the two main branches of career paths for TPMs: the individual contributor path and the people manager path. Both will include insights from across the industry about where the paths can lead you as well as the types of challenges you may face.

9
Career Paths

In this chapter, we'll look into the paths you may take to become a **technical program manager** (**TPM**), followed by a closer view of the different career paths that are available to a TPM. As the tech industry is large, the paths described here represent a generalized view of these career paths based on the interviews I have conducted, as well as the job descriptions available to me.

We'll also get a close look at two professional TPM journeys that showcase the different paths you can take, which will reinforce that no two journeys are the same, nor are they ever straight paths.

We'll explore the available career paths by doing the following:

- Examining the career paths of a TPM
- Exploring the **individual contributor** (**IC**) path
- Exploring the people manager path

Let's dive right in!

Examining the career paths of a TPM

The career paths of a TPM are as varied as the paths to becoming a TPM in the first place. Each person has their own story and path they follow, and the timing from step to step is also just as variable as the number of TPMs. I've combined job postings, interview notes, and personal stories of TPMs at different stages of their career to get to the perspectives in this book. What the data only hints at, however, is where we will start. *How do you become a TPM?*

The path to becoming a TPM

The TPM role is highly technical in most instances across the tech industry, as well as in industries needing technical expertise. A key component to a successful start as a TPM is having a strong technical foundation. This is traditionally in the form of a technical degree, such as **computer science** (**CS**), **information technology** (**IT**), or **informatics**, with CS being the most traditional. Regardless of the name of the degree, the knowledge you carry from the degree and your experiences is what matters

both in the interview process as well as the job itself. We'll cover the technical competencies most often used on the job in *Chapters 11*, *Code Development Expectations,* and *Chapter 12*, *System Design and Architectural Landscape*.

I've seen TPMs start out their careers in various roles such as **software development engineer** (**SDE** or **SE**), **development operations engineer** (**DevOps**), and **business intelligence engineer** (**BIE**). On occasion, a TPM will start their career as a TPM out of college, though this is becoming a rare occurrence as many companies are removing entry-level roles for TPMs, opting instead for people switching to TPM after obtaining *real-world* experience. This experience helps grow your depth of knowledge by working directly on solving real problems. Depending on how far you go down the pre-TPM path, you'll begin to work on your breadth of knowledge with system and architecture designs and possibly strategic product planning.

Some TPMs start out via an SDM role or other people management role, which usually stems from a software development role or another IC role. In these cases, they may see a product, project, program, or situation that suits them as they switch back to an IC TPM role. In other cases, they found that they prefer honing their IC skills instead of people management skills.

The paths of a TPM

Once you are a TPM, you'll begin to progress in your career, strengthening your core competencies and stretching both the depth and breadth of your knowledge. Each company defines its own career paths, which can vary depending on size and available opportunities. Within the tech industry, there is a general trend toward two distinct paths you can take to further your career – the IC and the people manager. Both are equally valid and have their own unique traits that may fit your style and aspirations.

Figure 9.1 explores some of the traits that are both shared and distinct between an IC and a people manager:

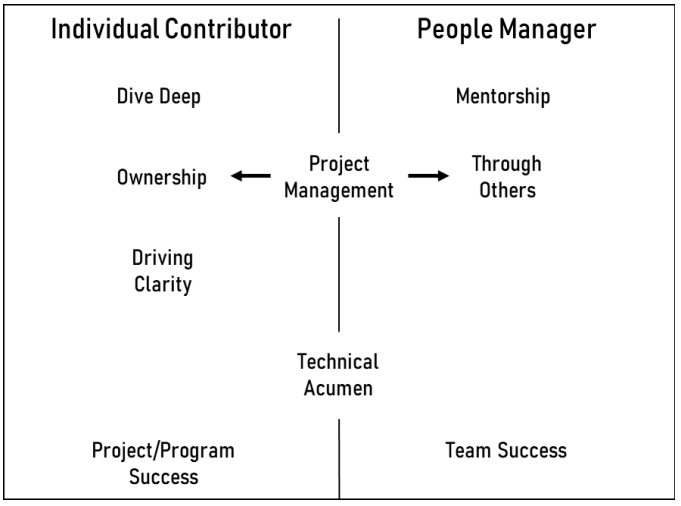


Figure 9.1 – Traits of an IC versus a people manager

The IC role focuses heavily on the direct delivery of value and impact. The skills that you have grown to manage a project and program, navigate stakeholders, and dive deeply into issues are the main skills in your toolset. In general, your focus is on delivering the project or program that extends to ensuring your organization is on the right long-term path for project and program success.

Your focus as a people manager is on the success of your team. You mentor your direct reports and grow their skill sets by sharing the knowledge you gained as an IC. They grow as an IC and your entire team benefits. In this way, you are focused on the success of your team and by extension the programs and projects that they deliver.

There are some focal points of your role that are shared between IC and people management; however, the emphasis on that focal point may be different. For instance, as a TPM, project and program management are a central focus to you. As an IC, this comes out via individual ownership of a project or program. You thrive in owning and executing programs and projects and dive deeply into every aspect of the project. As a people manager, you work through others by guiding them so that they can be successful in delivering the program or project instead of delivering it by yourself.

Technical acumen is also shared across both of these different career paths, which makes sense given that they both share the same technical foundation. The depth and breadth of your technical knowledge may differ based on your chosen path as people managers will focus on a higher level of understanding of the systems their team owns or works in. As an SDM, this may also go deep into your own service but not necessarily the services surrounding you. As an IC, your technical acumen will depend on the needs of your projects and programs – you may dive deeply as an embedded TPM for a specific team, or you may have a broad understanding of the systems around you to better execute cross-functional programs. As your career develops and you go higher up, both the breadth and depth of your understanding will increase.

For people management, there are generally two paths available – a software development manager and a manager of other TPMs. Both paths are not always available depending on company philosophy, but these changes can also be accomplished between different companies and represent the two people manager paths most often taken by TPMs.

Figure 9.2 shows the career paths available for a TPM that chooses the people manager progression, and compares them to the IC path:

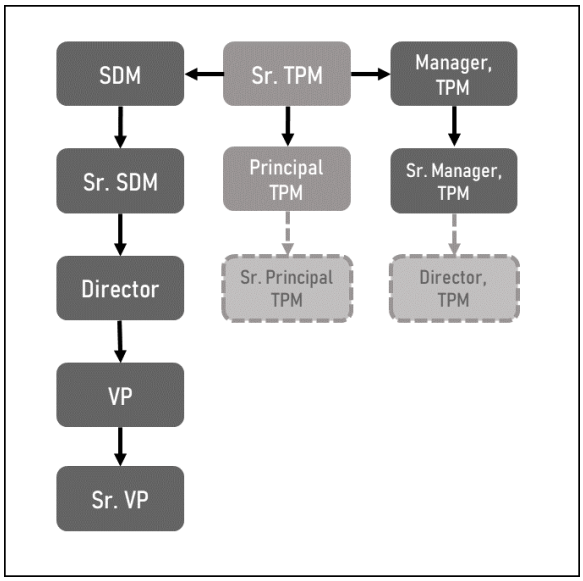


Figure 9.2 – TPM career paths

The center column represents the IC path, moving downwards toward the higher levels. The senior principal TPM position is listed here as it does exist in some companies, and I expect this trend to continue, but it isn't mainstream as of now. The columns on either side represent the two people manager paths mentioned: SDM and TPM manager. As you can see, the typical first change from IC to people management is lateral – meaning you stay at the same level, just a different job family (for SDM) or classification (TPM manager).

The SDM path is where the potential to move upwards to senior VP is the most prevalent (though that doesn't mean it's easy, just that it's a path well taken!). Of course, this can go beyond senior VP, but this is a good stopping point for illustrative purposes. Each level increases your team and influence (thus your impact).

The TPM manager path is often a shorter upward path as compared to the SDM route. Just like the senior principal TPM level, the director, TPM position is available in some organizations though not widespread. However, it is worth noting that once a people manager, moving from a TPM manager to SDM is a simpler move than from IC to people management.

Now that we've seen how IC and people management paths differ, let's look at each path in more detail.

Exploring the IC path

The IC role is similar to that of other IC roles such as software developer. As it suggests, an IC will contribute directly to the success of the company through deliverables, whether that's code for an SDE or delivering a project for a TPM. This role works as part of a team in the sense that they work

with other ICs to deliver results. Unlike an SDE role though, a TPM will rarely work with other TPMs from their own team. Instead, when working with other TPMs it's in a cross-organizational fashion.

Given that the IC role leans solely on the deliverables of the individual, the IC path tops off at the principal level in most companies. This is equivalent to a senior manager or director, depending on the company's verticality. The reason for this largely comes down to the ability of a single person to have a large enough impact to warrant the level. Though TPMs are force multipliers, the people we work with are still within different organizations and their successes are not directly our own. The deliverables are also measured at the scope of a single program, and 0though this is large, it is not as large as the deliverables under a director or VP since they are measured by the entire work of their respective organization. This makes influencing without authority much harder at higher levels.

That's not to say that a higher level isn't achievable in some cases. For instance, Amazon recently merged the career path of the TPM and the software developer starting at the L8 level, now called the Senior Principal in Tech. This allows a TPM to achieve up to L10 as an IC which is a Distinguished Engineer in Tech (there is no L9 at Amazon). These roles show that at this level, your influence is geared towards the strategic goals of the company and the starting path is no longer relevant as you aren't writing code or delivering on projects per se. Of the Big Five I interviewed, this level for the IC path was only seen at Amazon; however, this will likely be a trend we see moving forward that more companies to pick up.

Figure 9.3 denotes the career path of the IC TPM:

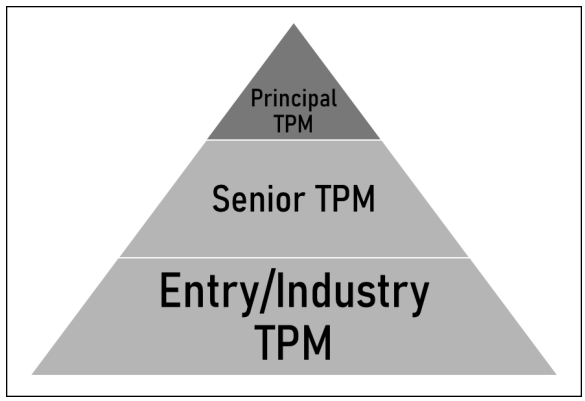


Figure 9.3 – IC career path

The IC career path is shown as a pyramid to denote not only the upward path but also the relative number that each of these positions holds within a company. This helps illustrate the complications of moving upward, as well as the diverging paths that lead to a smaller number of positions the further up you go.

The industry TPM position is largely considered a career position meaning that if you are happy at that level, you are not required to continue climbing. The senior position is also a career position, but

it is also a crossroads for most TPMs as there is a split path at this point in your career. In the words of *The Clash*, "should I stay, or should I go?" You can choose the IC path and either stay a senior TPM, push upward to the principal, or you can pivot to the people manager path. As mentioned in *Chapter 1*, switching to people management is a popular shift at this level. This can be either through switching over to a software development management position or into a TPM manager role.

All of these paths are equally valid and which path is right for you depends on what you are looking for out of your career. Let's look at the path of a current principal TPM to see why IC was ultimately the right choice for him.

Oscar Jaimes is a principal TPM at Amazon. His journey took him through both IC roles as well as people management roles. He received a degree in CS and worked as a software developer for six years. Toward the end of that time, he went back and got a master's degree in business administration to help him move into project management, where he stayed for three years. From here, Oscar found opportunities where he would lead people and products, even being the chief technology officer for the Digital Business Unit at Latam Multinational. These positions were all strategic in nature and he helped build businesses and ideas from the ground up. He expanded his leadership up to thirty people as the director of innovation and business development – again, concentrating on growing a product from idea to launch. Though his path changed often, the main driving focus was the product and seeing it through. This is what led him to being a TPM at Amazon, launching multiple global products on his way to principal TPM in 2021.

Now that we have a good understanding of the IC path, let's move on to the people manager career path and see what opportunities are available.

Exploring the people manager path

For TPMs, the people manager path goes in two directions for most companies in tech: SDM or TPM manager. Both of these paths switch focus from individual contributions to the company's success and instead focus on your ability to bring success through others. By growing the careers of the people that report to you, their contributions grow, and the collective contributions of your team grow. The difference in the paths is the composition of your team as well as your mobility upward. Each company is different, but this is true for the *Big Five* companies that I interviewed for this book. As you will see in an example later, companies outside of the *Big Five,* or really any of the top tech companies, have their own progression route that may not differentiate between these two people manager paths.

The journey of a people manager can bounce between both SDM and TPM manager. Let's take a look at the journey of a people manager that held both of these roles.

Faheem Khan is a senior manager TPM at Amazon. Throughout his career, he has been both an IC as well as a people manager in various forms. He started out ambitiously co-founding a start-up, Planetsoft, in India and found quick success with requests to build out enterprise solutions. He realized he needed to learn how enterprise software was created before committing to do so himself, so he got a job at a tech firm as a software tester.

When finding an issue, Faheem was driven to get the right people together to solve the problem. This tendency promoted him to test lead, where he moved to Seattle to work with Microsoft. He first shifted into people management as a Software Test Manager to build out the test framework used by Expedia. He enjoyed working with the users and stakeholders while building the framework. Faheem switched to the TPM job family to learn about business while still being close to technology at Expedia. He grew into a principal TPM, director, TPM, and then director of technology and managed cross-functional teams. He bounced back and forth between people management and program management. This back and forth led him to think about what he was best at and what he enjoyed doing. He decided that he was a leader of people and enjoyed project management in the technology space. Knowing what he wanted, he found a TPM management position at Amazon.

Faheem notes that whenever his career needed him to dive deep, he would take up an IC role so he could focus, learn, and hone new skills. Once those skills were present and he was ready to share and drive others, he would find a people manager role. This is an incredible insight and embodies both the breadth and depth of knowledge that is expected out of a TPM.

From both Oscar's and Faheem's journeys we've seen the hallmarks of what makes a great TPM. They exhibit a drive to see a project through to completion, a willingness to learn and help those around them grow, and a desire to dig deep. Even with these common traits, different paths were taken for each, and both held IC and manager roles, and each ended up preferring one path over the other in the end. Faheem's insight that the type of role that was best suited for him depended on what he was looking to achieve rings true for all of us.

Summary

In this chapter, we examined the different paths that can lead to a TPM career, focusing on roles such as an SDE, DevOps, and even an SDM. We discussed the motivations for the switch to TPM as well as the paths that are often available once you are a TPM.

We saw that the IC role allows you to exercise your project and program skillsets and directly contribute to the success of your company. We followed the path of a principal TPM as he went from IC roles in product and program management, to people roles as a director and development manager.

We then explored the people manager role and how you focus on the growth of others by imparting your IC experience to achieve success for your team.

We also followed two different TPM professionals through their career journeys to see how their choices led them to different paths. Most importantly, we got a glimpse of why they made the decisions they made to help us better understand what career path may be best for us.

In *Chapter 10*, I'll introduce you to the technical toolset that is fundamental for a TPM. I'll discuss why a technical background is necessary, as well as which areas of your technical acumen are most utilized in the TPM role.

Part 3: Technical Toolset

In this part, we will go into greater detail on the intersection between a technical background and project management. We'll discuss the technical skills most often utilized in the TPM role and see how these skills enhance your existing management skills to be the most efficient practitioner you can be.

This section contains the following chapters:

- *Chapter 10, The Technical Toolset*
- *Chapter 11, Code Development Expectations*
- *Chapter 12, System Design and Architecture Landscape*
- *Chapter 13, Enhancing Management Using Your Technical Toolset*

10
The Technical Toolset

In this chapter, I'll make the case for the technical toolset. Introduced in *Chapter 2*, *Pillars of a Technical Program Manager*, the technical toolset is a foundational pillar for a TPM. It is the connecting glue between the other pillars, program and project management, and is the foundational pillar that sets a TPM apart from a generalist PM.

We'll continue by discussing the various tools in the technical toolset and how they help you not only excel at your job but also work across job families as the need arises.

We'll explore the technical toolset through the following topics:

- Examining the need for a technical background
- Defining the technical toolset

Let's get started!

Examining the need for a technical background

In discussing the origins of the TPM role, we looked at the role of the PM and how gaps in knowledge could hinder the execution of a project or program. This book has focused largely on the tech industry and the common usage of the word *technical* to refer to information technology. However, if you look at the word as a synonym for *specialized*, then the need for a *specialized* background might be a bit clearer.

Looking back at the pillars of the TPM from *Chapter 2*, program and project management are two-thirds of the foundation of a TPM. This is because these skills transcend each individual project and program management position that you may hold. They are the most fundamental needs to succeed. However, to truly thrive as a TPM, your specialty focus as a technically minded practitioner requires a fundamental understanding of technology. This is what sets you apart and allows you to be more successful than a generalist PM in this role.

TPM specializations

The tech industry as a whole is still getting used to the concept of a TPM as a needed role. However, some companies are going a step further and making it more specialized within the technical spectrum. This makes sense given that the reason for the TPM role is to provide specialized knowledge of information technology to the PM role. As the market for TPMs becomes saturated, a more nuanced delineation is needed when a simple technical background will not suffice. When you need a TPM specializing in **Application Security** (**AppSec**), then the title becomes *TPM – AppSec*. If you need a TPM that can help you move entire ecosystems to the cloud, you may need a *Solutions Architect – TPM*.

> **Becoming a TPM**
>
> The career path of a TPM was covered in *Chapter 9*, *Career Paths*, but it's worth noting here that while the discussion on needing a technical background sounds like the path to a TPM is as a PM that expands their technical knowledge, the path is most often a person with a technical background (for example, a software developer, systems developer, or another similar role) that expands their PM skill set.
>
> This is why this book provides a lot of chapters on PM skills that are relevant to a TPM as this is the area of growth for most people coming into the TPM field.

This trend of specialization is similar outside of the tech industry where TPMs are needed. A medical company may need a TPM with specific medical knowledge or a specific software package or device. One such job where this is needed is Program Manager (Medical Device). Though listed as a PM, the role requires an engineering background with knowledge of working with feature launches of medical devices. If the need is large enough, a new title may be created. *Figure 10.1* illustrates a few different TPM specializations found on the *Amazon* job board and how they are related:

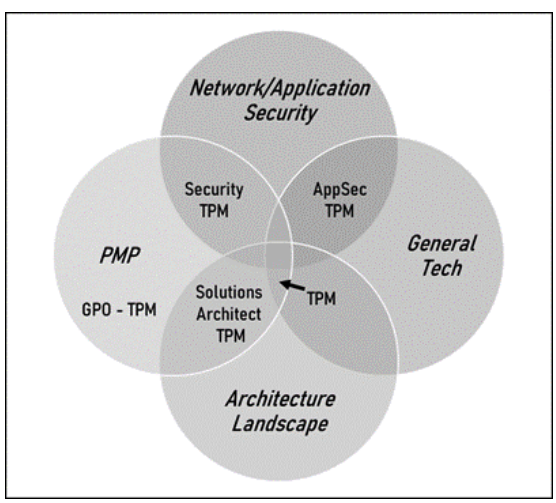


Figure 10.1 – Specialized PM overlaps

All of these TPM positions have a basic requirement of project and program management, as well as some technical background. The type of technical background needed is a large differentiator that denotes the nuances between some of these specialties. AppSec and Security TPMs share similar security-related skills. The solutions architect needs to understand the **full stack** – or the services used from the frontend to the backend of a cloud system.

Some of the specializations are not geared toward the technical background but instead toward specialized tools used in that role. The **Global Process Owner** (**GPO**) role focuses on change management, acceleration, and process improvement skills such as *Lean* and *Kaizen*. The Security TPM focuses on the Risk Management Framework, which is a specific type of risk analysis life cycle based on the life cycle described in *Chapter 6*.

This categorization is not exhaustive of what is out there, nor are the differences between them the same as every company has slightly different needs. However, this illustrates the varying degrees to which a technical background is required to perform the TPM role.

Technical proficiencies used daily

In each of these roles, a generalist TPM can ramp up and perform the job as they have the pillars needed to be a successful TPM. This ramp-up is often costly in terms of both time and resources and isn't always a fit for the needs of the organization. So, just as a PM could learn the technical foundation needed for a specific purpose or role, a TPM could do the same.

As we'll explore more in the next three chapters, your technical skills are widely used at every stage of a project or program. We've discussed some of the roles and responsibilities that a TPM has in various aspects of project management. *Table 10.1* provides a list of standard roles and responsibilities that a TPM may have where some level of technical acumen is needed:

Key Management Area	Step	Role
Planning	Refine requirements	Accountable
	Create functional specification	Accountable/Responsible
	Sprint planning	Responsible/Consult
	Review designs	Consult
Stakeholder Management	Draft communication plan	Accountable/Responsible
	Daily stand-ups	Consult
	Status report/meeting	Accountable
	Monthly business review	Responsible
Risk Management	Risk analysis	Accountable
	Risk monitoring	Accountable/Responsible
	Issue resolution	Accountable

Table 10.1 – Roles and responsibilities of a TPM

Throughout the life of the project or program, the TPM is constantly drawing on their technical toolset to deliver key artifacts, resolve issues, drive clarity, and ultimately deliver on the goal.

During planning, the TPM relies on their technical background to clarify ambiguity in requirements and to ensure they can transform them into a **functional specification** (**functional spec**), which is a document mapping business requirements to functionality (APIs, user actions, and features). In turn, the functional spec relies on the TPM's technical background and system knowledge to trace the system functionality back to the requirements. TPMs provide input during sprint planning as to which items to pick up. This draws on the project plan and technical dependencies between tasks that the TPM has identified. Lastly, design reviews are where the TPM uses their understanding of system design as well as the surrounding architectural landscape to influence the right design choices based on the needs of the project, team, and company.

Stakeholder management is focused on your ability to bridge the gap between business and technical concepts. As the communication bridge between the technical and business worlds, you draft the communications plan and ensure you have the right type of communications in place to bridge the technical gaps in understanding. As such, status reports, meetings, and business reviews all require you to tailor your status and narrative to the technical proficiency of your audience.

Let's look at a single scenario and see how it would need to be handled differently, depending on your audience. Let's say you have a design that has been delayed because the service interface contract between two services cannot be agreed upon. The sending team wants to treat missing or empty data as null, whereas the receiving team has a technical limitation where their service cannot handle a null value. The receiving team wants the values to be sent as empty strings, or to create default values for every parameter, and the sending team is challenging why the receiving service cannot handle a fundamental concept like null values.

If this scenario came up in a project that was building out a platform feature that was not externally facing, your stakeholders are likely all technically minded, and your status reports can stay technical. You would list out exactly what the issue was: a service contract disagreement, and the options being explored – setting default values or updating the receiving service to understand a null value. The audience would understand what was going on, as well as what the next steps are.

However, if your project is to deliver a customer-facing feature or product, you likely have non-technical stakeholders, so the MBR and status report need to be written in a way that they will understand this issue. In this case, it may be best to state that a design sign-off has been delayed due to a mismatching of requirements and what the newly expected ETA for the sign-off will be. A link to a more detailed rendition of the problem can be added but the details are left at a level non-technical teams would likely understand: the requirements or implementation steps are not clear, and the team is seeking clarity to close it out.

On the opposite end of translating a technical issue to a business stakeholder, you will be translating business needs into technical requirements or clarifications for your developers during daily stand-ups or sprint planning meetings. A common occurrence for this could be in the interpretation of the business requirements as the business teams may have their own jargon that can cause confusion on the developer side. As an example, I work with tax compliance legislation, where requirements take multiple steps to get to the development team as they are essentially translated from legislation into system requirements and then into functional specifications. On the extreme end, you have the business as the tax legislation that is written. Legislation is often very heavy in legal terminology, which can be hard to translate into system requirements. For instance, one law may refer to the tax owed on a customer's purchase. This may seem straightforward for a brick-and-mortar system but can be complicated when talking about e-commerce systems, where a purchase may result in multiple boxes being shipped out. Understanding the difference between what the law says a purchase is versus what that means for the e-commerce ecosystem is something I would work out as the TPM when writing the functional specification. Helping the development team understand that a purchase has a different meaning in the business than it does in the technical workflows helps remove built-in assumptions from the developers as to what work is being done.

In risk management, the analysis will draw on your deep understanding of the technical systems you are working with to identify and classify risks. The more you know and understand about the frameworks, network protocols, latency constraints, and availability needs of your system, the more problems you will be able to foresee.

Now that we know why the technical toolset is needed, let's explore how the toolset can help you help your team by helping others in their role.

Using your technical toolset to wear many hats

Wearing many hats means leaning into one technical skill over another to fill in gaps in roles based on need. As such, your technical toolset should be well-rounded in preparation for that need.

In *Chapter 1*, we discussed the need for a TPM to *wear many hats*, or to take on part of a role that is normally adjacent to you but that is missing. If your organization has a product but doesn't have a product manager, you might find yourself prioritizing a product roadmap. Similarly, some TPMs take on the role of a scrum master when one isn't present.

Each of these roles leans on different technical skills and will require some level of proficiency to *wear the hat* effectively. *Figure 10.2* illustrates the interconnection your technical toolset provides to allow you to step into varying roles:

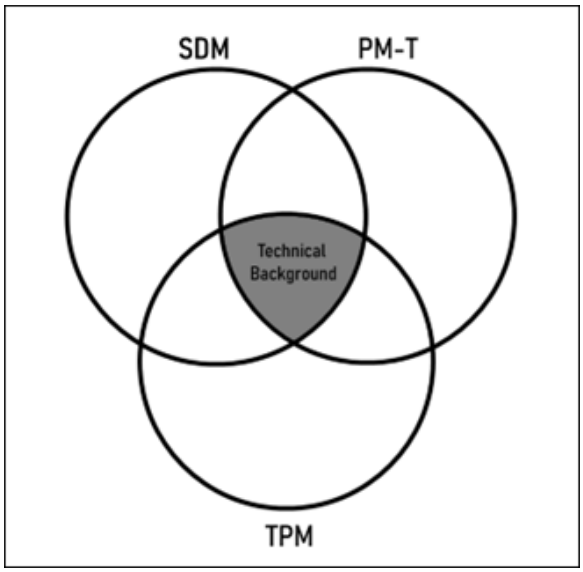


Figure 10.2 – Technical overlap across job families

This Venn diagram highlights that, across the adjacent job families of PM-T and SDM, all three positions rely on a technical background to succeed. As seen in *Chapter 1*, *Figure 1.5*, each of these has some overlap outside of the technical background that also helps you wear the hat of a PM-T or SDM. However, without this technical background, the other skills would not be enough for you to step into these other roles to take on service-level projects for the SDM or roadmap prioritization efforts of a PM-T.

Now that we've made the case for needing a technical background and how the technical toolset will help you in your role, let's look into what the skills are in the toolset.

Defining the technical toolset

As we discovered in the previous section, the technical foundation that you have can vary widely, depending on what your role requires.

Each of these tools is an accelerator to your role when your proficiency matches the need of your team. In cases where it doesn't match, you will need to lean on a specialist that does have that proficiency, such as a specific programming language. This will add time to the project as additional communication is needed to get the same results you would get if you were proficient in the right technical skill.

In this section, we'll discuss three areas that are most widely used across the general and specialist TPM roles. We'll touch on the level of code proficiency that may be needed, system design, and architectural landscape.

Code proficiency

As a TPM, you will rarely be asked to write code. However, this doesn't mean you shouldn't be proficient. As a TPM, you will be working alongside software developers in most roles as they are the main resources working on technical projects. This means that your work breakdown structure will largely be high-level tasks, or stories, aimed toward your development team to implement. Your estimates, daily stand-ups, and a large portion of issues and risks will be related to code or the software development life cycle.

To determine accurate estimates, vet timelines for tasks, and even help with designs, you'll need to have a basic understanding of reading and writing code. As with most tech jobs that require you to write code, the language you know isn't as important as knowing one. This is because understanding a programming language is more about the capabilities, shortcomings, and problem-solving related to that language. Learning the syntax of another language and ramping up on its weaknesses and strengths is much easier once you know at least one language. This is why many tech companies allow their developers to interview in any mainstream language. This applies to the TPM role as well.

There is one caveat to this, however. There are different types of programming languages and knowing a language of one type doesn't immediately translate to a language of a different type. The most common types in modern systems are object-oriented languages such as *Java*, *C#*, and *Objective-C*, and functional languages such as *Scala* and *Haskell*. *Python* can be used to write in both styles and is popular in technical classes. Knowing an object-oriented language does not guarantee an immediate understanding of a functional language because how they behave is fundamentally different. The same is true for the reverse as well. However, knowing one language is still easier than learning from scratch!

Chapter 11 will go into more specifics about the level of code proficiency expected, along with some examples.

System design

System design involves mapping the components and data flow within a system or service. It is also the most commonly exercised technical proficiency for a TPM. This is because most TPMs are associated with a service team or group of service teams, so knowing how the service components connect, what their APIs do, and which other services it interfaces with is required to perform your job. You cannot effectively write a functional spec without understanding the system that the functional spec is referencing.

In most cases, your day-to-day experience with system design will be concerned with writing functional specs and participating in design reviews. This means you will mainly be learning existing systems, so understanding how to dive deep into a system is an important skill to hone. You need to learn what APIs are available to clients of the system, what arguments they take, and what information they vend. I find that to truly understand an API, it's best to also understand why it exists in the first place. This goes deeper than the answer that it exists to vend certain data. You need to ask why the API was created in the first place instead of extending an existing API and what service boundaries that the API is working within.

As the bridge between the business team and the technical team, it is important to understand the *why* and *how* of your systems as much as the *why* and *how* of your business. The reason why an API exists – which is often business-related – can help drive clarity to software designs when the developer proposes multiple options based on technical optimizations or needs. You can rule out proposals based on the business context of the requirements of the API or the needs of the business. Acting as the bridge is where our most valuable contributions come from.

Chapter 11 will also cover system design in more detail and will provide an example of system design, as well as some expectations in interviews.

Architecture landscape

An **architecture landscape** is a map of application and service interconnectivity at an organization or enterprise level. It's similar to a system design but has a wider scope. Whether or not you will need this depends on your organizational structure and how closely you work with systems outside of your team. This also correlates with your TPM level: the higher your level, the wider your scope and impact become, which will necessitate understanding periphery systems in your team. Just as a junior developer would only focus on creating classes and methods for an existing system, an entry-level TPM would focus more on the immediate systems they interact with and may not think much about the overall architectural landscape.

Given the broader scope of architecture landscapes compared to a system design, the level of depth you'll be expected to understand is much less. After all, these systems are not your immediate concern, so knowing their internal data flows is not nearly as important as understanding the input and output of the systems.

When mapping out an architecture landscape, you may focus on different aspects of system relationships, such as system dependencies or data flow. Each of these aspects brings a different perspective of the landscape to light and may aid in answering different questions. For instance, if you need to get access to a piece of data, knowing where that data originates and which systems it currently flows through can help you define and design a solution that gets the data to where you need it. However, knowing the system dependencies can help you determine the impact an API change may have on downstream services so that you can start a campaign to work with potentially impacted clients to navigate the change.

Understanding your architecture landscape is another area where the TPM acting as the bridge between the technical team and business teams is important. Similar to knowing the ins and outs of a system, knowing how your system connects to other systems in the company ecosystem can help you notice patterns or places of concern that others closer to the project cannot see. Understanding how downstream systems might react to a new piece of data or even a new value for an existing field can speed up the design process and prevent undue issues from coming up during integration and testing. Aside from knowing how systems will behave with new data, you also understand which systems are impacted by your requirements and can include them in your initial project plan.

Summary

In this chapter, we looked into the need for a technical background to be successful as a TPM. As a specialized PM, the TPM brings their technical toolset to the table to manage a technical project more effectively. We saw how the trend of a specialized PM is going deeper as the TPM position itself is being refined into hyper-focused areas of expertise, such as AppSec – TPM.

Then, we examined the common technical background across adjacent job families and how this allows the TPM to fill in gaps in personnel on their project or team more effectively than PM skills alone.

Lastly, we looked into the tools in the toolset that are most used across the industry to get a sense of how those tools are used.

In *Chapter 11*, we'll dive deeper into some of the technical skills introduced in this chapter. We'll discuss code use and proficiency for the TPM role and look at some of the high-level concepts you should understand.

We'll also discuss system design. As this is a highly used skill and it is often brought up in interviews, we'll focus on good and bad design practices while using the *Mercury* project as a basis for the examples.

11
Code Development Expectations

In this chapter, we'll cover code development expectations for a TPM. Though programming isn't a core competency that you should expect to see come up in an interview, it is foundational to project and program success. If we don't understand what our team is doing, and if we can't relate, push back, and advise, then we will constantly be in the dark.

Trust is a very important part of being a leader. You need to be able to trust that your team knows what they are doing and are making the right decisions, but you also need to be able to verify progress and be there for your team when they need your input. Without understanding the basics of programming, it will be hard to achieve the right balance of trust and verification. With that in mind, we'll go over some fundamental code concepts that are most relevant as a TPM. These concepts are not exhaustive but will provide a good foundation for additional knowledge related to programming. Along with system design and architectural landscape design in *Chapter 12*, these topics are important to any TPM position in the tech industry. Not every TPM has these foundational skills, but the best TPMs I know in the industry do.

We'll explore these code development expectations through the following topics:

- Understanding code development expectations
- Exploring programming language basics
- Diving into data structures
- Learning design patterns

Let's begin!

Understanding code development expectations

TPMs are surrounded by software development teams and interact with developers, support engineers, and development managers on a daily basis. They are involved in technical discussions around requirements, release management, and feature and system designs. So, in the midst of all of this software talk, let's explore what level of code proficiency is expected in the TPM role.

No code writing required!

As a TPM, your focus will not be on writing code yourself, but on getting it written through others. Of all of the TPMs I know and have worked with, only two—including myself—have written code as part of their role. I've heard of TPMs writing code in start-ups, and this aligns with our need to wear many hats because the number of people involved in a start-up is comparatively small and the need to step up has a higher chance of occurring. In larger companies, roles are usually tightly defined, and the overlap and opportunity have a smaller chance of occurring.

The type of code I've written as part of my role was only to help me perform my job and was not related to project deliverables. In one case, I was doing some data analysis (wearing the hat of a data analyst as we did not have one on the team) and needed to take service logs and extract client information from each call to get a list of clients. I started by just using simple text manipulation in a text editor but saw an opportunity to write a script to get the job done much faster, so I wrote it. A fellow TPM wrote a script to pull and format data from a tool into a status report, reducing the time to draft the weekly status report. This wasn't strictly needed to do the job, and no one saw the code, but it made our jobs easier.

Though not required, writing this fit my style of management, and I also anyway enjoy writing code. I say this to illustrate that though some TPMs may write code, it's by no means a requirement of the job.

That being said, you will be working directly with software developers, and to be an effective leader and communicator, it's best that you understand the basics of at least one programming language. We'll start out by looking at basic concepts of programming.

Exploring programming language basics

Most companies lean heavily toward a specific language or set of languages, depending on what the code is used for. Server-side applications in enterprise settings tend to prefer Java, whereas **machine learning** (**ML**) applications perform fairly well in functional languages or hybrid languages such as Python. Whichever language you find yourself closest to, you should learn the basics of that language. If the team you work with is heavily using a functional language and you only know **object-oriented** (**OO**) languages, use this as an opportunity to bridge the gap and learn the fundamentals of functional programming. There are books on both language types referenced in the *Further reading* section of this chapter to get you started.

As a brief recap, **OO programming** (**OOP**) is a language paradigm where the application is defined by a series of objects that can interact with each other. An object consists of data fields and methods that act upon the data fields. Most OOP languages also support events, which are methods that are automatically invoked based on the state of the application. As an example, in a standard Windows dialog box, the **OK** and **Cancel** buttons on the dialog box are both objects. Both buttons have an event to handle when a user clicks on the button. The buttons also have data, such as the text on the button, pointers to the `OnClick` event, and even font settings for the text. By clicking on the **Cancel** button, the button's `OnClick` event will interact with the dialog box—which is the parent object for the button—and have it close with a special state denoting that the dialog's request was canceled. The application that invoked the dialog box will then act upon the canceled state. Not all actions are driven by the user, often referred to as being event-driven, but the concept of objects interacting with each other is the central concept of OOP. The interactions are through a series of written-out procedures to manipulate the state—this makes OOP a type of **imperative programming**.

In contrast, **functional programming** utilizes **declarative programming**. In a declarative language, the code does not describe how to perform an action, only what the outcome needs to be. In the popular declarative language **SQL**, you are stating which data you need in the `SELECT` clause, which filter to apply to the data in the `WHERE` clause, and which tables to retrieve the data in the `FROM` clause. You are not stating how to retrieve the data (such as running a loop over the data and for each object, retrieving specific fields, then sorting the resulting list). The language will determine how to best retrieve the data on its own.

The declarative nature of functional programming puts emphasis on the **function**, which is a method that returns a value, instead of objects and their interactions with one another. Therefore, the data remains local to the individual functions, which reduces side effects by not allowing non-local data manipulation. The data returned by the function is all that has been manipulated.

Regardless of the language type, you should know enough about the language to read through the code and have a basic grasp of what it is doing. This is especially useful in organizations where the code base is accessible to you (not all companies have an open code base nor allow non-developers access). Being able to read a method and see what it does instead of finding a developer to explain it can be a valuable time-saver and will give you the chance to understand your services at a deeper level.

A good start to understanding a language, and your own services, is understanding the method, function, and API signatures. To do this, you need to understand the components of the signature as well as what that API is doing. The following code snippet takes a look at an example method from the original source code (in C#) for the Windows `Mercury` messaging app. We'll examine the method signature and discuss the basics of what the method is doing as an illustration of the level of depth that is useful in *Figure 11.1*:

```
public void SendTextMessage(string message, MemberInfo miTo)
{
    messenger.SendMessage(new MessageInformation
        {
            Member = myInfo,
            MessageType = MessageInformation.MessageTypes.Text,
            To = miTo.ComputerName,
            Text = message
        });
}
```

Figure 11.1 – Mercury code snippet

In this example, the method we are looking at is a method on the `Member` class that sends a text message to another client on the network. This is a method, as opposed to a function, as it is defined within a class and therefore has access to all information within the class. This tells us that the procedure does not return a value and is therefore a method. For a function, the keyword `void` would be an object or data type corresponding to the object or data type that the function returns. This method has two parameters: a `string` named `message` and an object of type `MemberInfo` called `miTo`.

Inside the method, the `SendMessage` method is invoked on an object named `messenger`. This method, in turn, is passed a single argument of a new object of type `MessageInformation`, which is instantiated inside the method.

There are several objects referenced within this method that are defined outside of this code snippet. However, some inferences can be made based on the names of these objects. For instance, we can guess the intent of the method is to send a text-based message from the current user to someone else on the network. Based on the method call to `SendMessage` on the `messenger` object, we can make an educated guess that the `messenger` object is responsible for sending the actual message. This is further backed up by the payload in the argument being information on the sender, client, and the actual message to be sent.

As you can see from this example, a full understanding of every construct in a language is not needed to build a good idea of what is going on. Following through from the method signature to what the method is doing can often build up enough context to understand what is going on. And if you have full access to the code, further investigation into these custom objects can solidify your confidence in what the method is doing and even where it is invoked from within the application. Though this is a method from within a class, this same exercise works for APIs of services and is extremely helpful in building an understanding of how the services and applications you work with interact with one another and the type of data they return.

Now that we have established a basic language literacy, we'll move on to some fundamental programming constructs that will likely surround you in your day-to-day dealings with developers and SDMs.

Diving into data structures

Though a TPM is not held to the same level of proficiency as a software developer, we should understand the basic concepts of programming. We'll cover the programming topics that come up the most in your day-to-day activities.

A *Data structures and algorithms* class was likely in your first or second year in college if you took a traditional route. As with most of the programming fundamentals, you won't be using this yourself in your day-to-day work. However, you can think of them as a strong foundation for the language that your development team will be used in most conversations you have with them.

I'll briefly go over a few of the more common data structures that I come across in design meetings, standups, and general work conversations. I encourage you to read through the books in the *Further reading* section on this topic for a more in-depth review. Even if you've taken the class and remember the concepts, it's always good to refresh your memory.

Space and time complexities

Before discussing data structures, it is helpful to understand the **key performance indicators** (**KPIs**) that we measure them by. In a computer, **random access memory** (**RAM**) is where data is stored that is in active use, such as variables in an application. As RAM is limited, measuring the amount of space data takes up in RAM is an important consideration as it is a limited resource. The other consideration is the amount of time it takes to perform an action such as searching, inserting, deleting, or accessing data. This is especially true for OOP, where data is often accessed inside of loops that iterate over large datasets. The amount of time it takes to perform an action once is then compounded by the number of times the loop is run and can add up very quickly to a considerable time sink if the wrong data structure is utilized for the task.

Both of these measurements use what is referred to as **big O** (*big "Oh"*) notation. These measurements are essentially categories used for reference to understand the performance of a data structure or method. For this context, the big O notation is used assuming asymptotic growth and uses *n* to denote the input that impacts the growth. Essentially, these math functions represent the curve, or behavior, that the space or time performance will start to match as *n* gets large enough. As an example, if the amount of time it takes to access a specific element correlates linearly to where it is in the data structure—for instance, the fifth element in a collection—the big O would be *O (n)*. As *n* increases, so does the time it takes, which is called linear time. However, if the amount of time it takes to access an element from a data structure is the same regardless of where in the data structure the element is, then the big O is *O (1)*, or in other words, constant time. As a TPM, knowing where the complexity categories come from isn't as important as knowing the relative costs associated with each big O. *Figure 11.2* shows each big O category on a curve of *operations* versus *elements*:

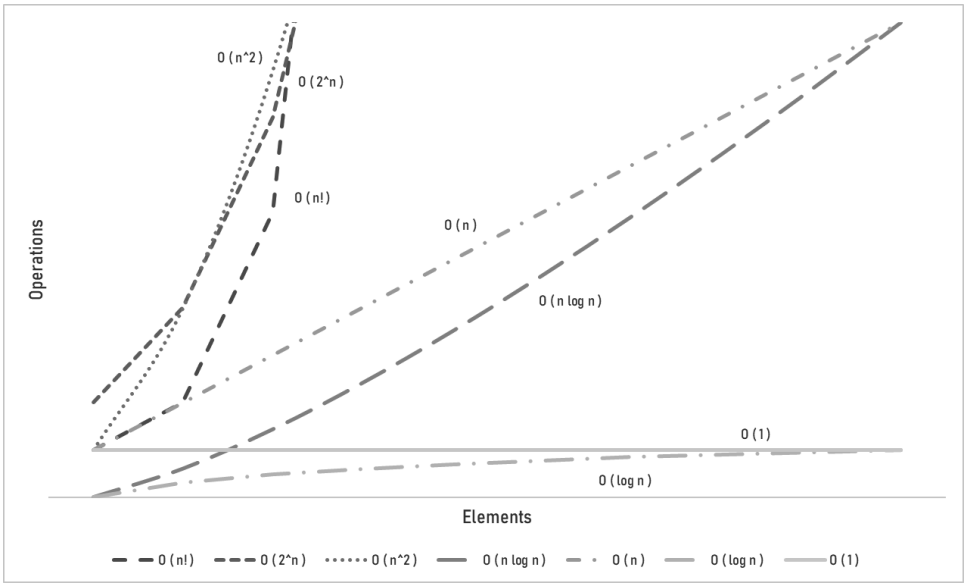


Figure 11.2 – Big O space and time complexity

As a TPM and not a software developer, this is one of the instances in which understanding the relative position of best to worst is more important than the how and why. In this diagram, we can see that a big *O* of *O (n!)* (*n factorial*) performs the worst once *n* gets large, and *O (log n)* and *O (1)* perform the best at near-flat curves. In the *Data structures* section, I'll refer to the big O if it is of particular interest. It's worth noting that these categories are models of the performance and won't necessarily match real-world performance, especially as *n* becomes much larger than expected, as can happen in large-scale tech industry applications.

Data structures

There are a large number of data structures that you will come across in programming, so much so that there are entire books on this topic alone. I've added a good book that covers this extensively in the *Further reading* section at the end of this chapter. For now, I'll summarize a few of the more common data structures that you'll come across as a TPM listening and contributing to design reviews, standups, and various hallway conversations. The goal here isn't to make you an expert, but to ensure you know enough to be comfortable in conversations and make informed decisions when appropriate.

Linear data structures

First up are **linear data structures**. These are a collection of objects that are accessed in a sequential fashion. In OOP, the most common way to act upon a large dataset is to loop through each piece of data and perform the same manipulation on each one. As such, linear data structures are some of the most common data structures you will encounter in OOP applications. We'll discuss the three most common next.

Arrays are a sequentially allocated collection of values or objects accessed via one or more indices. The array can have multiple dimensions, the more common being a two-dimensional array, which can be visually thought of as a table with rows and columns. Arrays can be multi-dimensional as well and behave like a one-dimensional array where the value is an array itself. Since an array is sequentially allocated in memory, accessing a particular index is constant—a *O (1)* complexity—since the index represents the distance in memory from the starting point of the array, so to get to the 50th element, you add 50 to the starting point in memory.

Lists are a collection of objects that are accessed in a linear fashion where you can traverse forward and sometimes backward. Many modern lists also support accessing via an index, essentially making them a one-dimensional array of objects with built-in methods to manipulate the list. As such, they behave similarly to arrays in terms of complexity when an index is involved. Without an index, accessing a specific element has a *O (n)* complexity, and is constant with an index. In many languages, such as *Java* (`ArrayList`) and *C#* (`List<T>`), lists are internally structured as arrays but have built-in methods for manipulation. In this regard, lists are often preferred over arrays unless specific optimizations regarding size are required.

Dictionaries are a collection of key-value pairs where each key can only appear once in the collection, meaning that dictionaries are hash tables or hash maps—depending on the usage of the dictionary.

Figure 11.3 compares these three linear data structures:

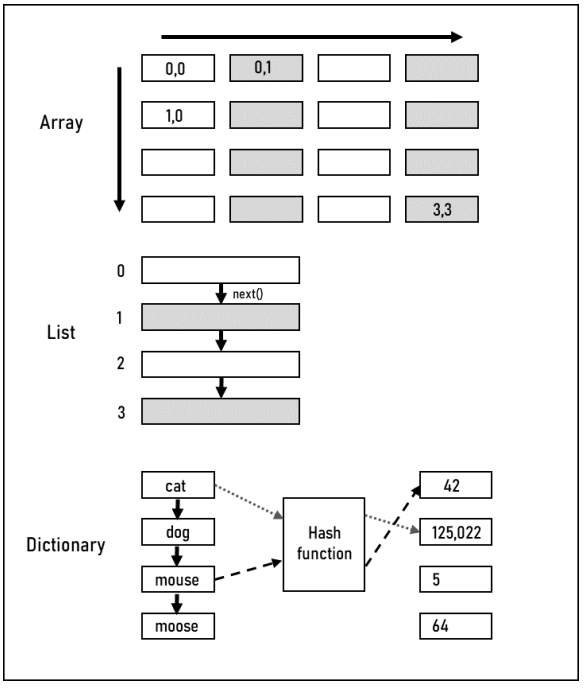


Figure 11.3 – Linear data structures

In this diagram, you can see that all of the data structures we've talked about so far are linear in that they have values that are accessed in a linear fashion, forward and backward, or directly via an index. Though a key is more abstract than a simple index, it is a direct accessor to a value, and many dictionaries allow you to enumerate through the collection using a `next` method. As these are all linear data structures, they have a constant *O (1)* complexity for accessing an element, but varying complexity for searching, adding, and deleting, depending on how each element is connected, or relates, to its neighbor. *Arrays* and generic *Lists* both have an *O (n)* cost for these actions as all items after the one you are adding or removing need to be reassigned to make room for the new element. If the list is sorted, this can be reduced to *O log(n)* by using a binary search. But if the list is a **Linked List** where there is no index, and every element is linked to the next element—and the previous element for **Doubly Linked List**—the operation is constant as only the references of the inserted item and previous and next elements need to be updated. The list doesn't need to be traversed in any manner.

Now that we've discussed some of the common linear data structures, we'll talk about non-linear data structures, including trees and maps.

Non-linear data structures

Trees are a collection of **nodes**, where each node can point to a collection of nodes called children where each child node can only be referenced once and the path between any two nodes is defined as an **edge**. This creates a hierarchical relationship to the data. There are many examples of trees in your day-to-day life, such as an organizational chart at work, or a phylogenetic tree that represents evolutionary relationships between organisms (this likely came up in a biology class!).

Graphs are similar to trees in that they are collections of nodes, and the path from one node to another is called an edge. Unlike a tree, however, a node can be referenced by more than one other node. Also, a node's relationship to another node can be *bi-directional*. A graph is the computer science equivalent of a map graph in mathematics. The best real-world example of a graph is a road map. If the intersection of two or more roads is a node, then it's easy to see where you can have a two-, three-, four-, and even five-way intersection. In this case, the roads are the edges that define the path between nodes, or intersections. In most types of trees and graphs, inserting, deleting, and searching all have the same complexity of *O log(n)*, or logarithmic time, with a worst case of *O (n)*.

Figure 11.4 illustrates a tree and a graph for comparison:

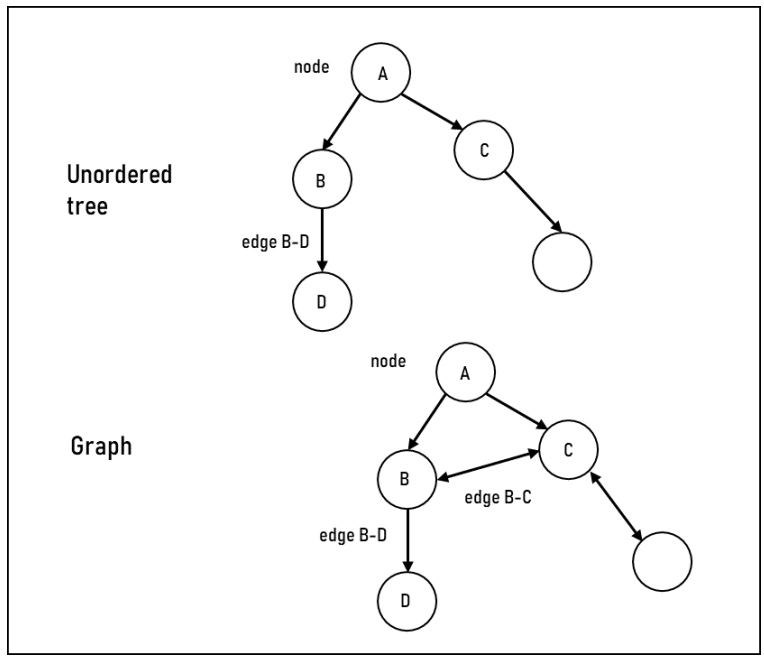


Figure 11.4 – Examples of non-linear tree and graph data structures

In this example, we have an unordered tree and a graph. The nodes are in the same locations to illustrate the similarities and differences. The nodes are represented as circles, with the path between nodes as the edges. Notice that in the unordered tree, the paths only go in a single direction and any one node is only referenced once. In the graph, the path between two nodes can be bi-directional, as seen on *edge B-C*. Also, *nodes B* and *C* are both referenced from two different nodes. In these ways, a graph allows for a more complex interrelationship between the nodes that a standard tree cannot provide.

These are all of the basic data structures that will be beneficial regardless of the discipline of the software teams that you will work alongside as a TPM in the tech industry. However, just as a TPM may be specialized, so can the development team, so additional data structures that are more closely related to the software being developed may be helpful. The *Further reading* section has an entry for data structures that goes over a larger list of useful structures. Use this as a starting point and dive deeper where needed.

Next, we'll move on to design patterns to wrap up our basic expectations in code knowledge.

Learning design patterns

Design patterns is a class that I see developers take often while on-the-job, mainly as a refresher as it is taught in college. It ensures a common ground of understanding, which is why I encourage TPMs to take the class, if available, as well. Here, we'll explore two groups of design patterns: creational and structural. There are more, but these are the two that I find the most useful for a TPM to have a good understanding of. To learn more, check out the *Further reading* section in this chapter.

Creational design patterns

Creational design patterns are related to the creation of objects. By creation, I'm referring to how to create an instance of an object. We'll discuss three of the more common ones next.

Builder pattern

The **builder pattern** separates the construction of an object from the specific composition of that object. As an example, we'll take the `Mercury` subsystem, where you might have two different styles of message you can send: rich text and simple text. A builder will allow you to specify a generic set of building methods that each object type will then provide its specific implementations of. The rich text builder would include additional steps in the method to handle rich text data such as text formatting, whereas the simple text builder would just need to deal with the text itself. In this way, the application can deal with a single builder object and, depending on which type is instantiated, the output can be specific to the needs of the object.

Figure 11.5 represents a simple builder pattern that is present in most OOP languages today:

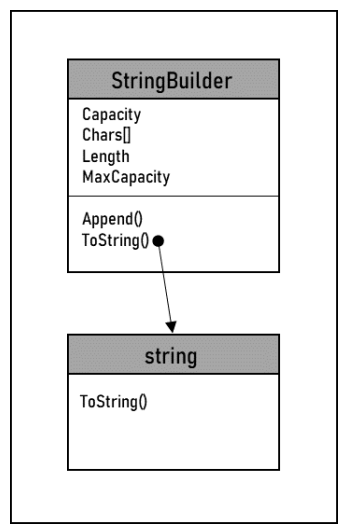


Figure 11.5 – Builder pattern

The `StringBuilder` class allows for the creation of a string from various complex data elements. The `Append()` method has several overrides that allow passing in every primitive data type to add to the end of the string. The creation method in this case is the `ToString()` method, which will take all elements appended to the builder and output a single string. Though seemingly simplistic, the `StringBuilder` class is significantly faster and takes less memory than simple string concatenation. This is because strings are immutable objects in most OOP languages, meaning that they cannot be modified. So, each addition of two strings creates an entirely new pointer and memory allocation. The `StringBuilder` class, on the other hand, stores all of the data in an array and only needs to update its memory allocation if the array runs out of space.

Simple factory pattern

A **simple factory** is where a class is used to create a specific type of object, usually through the use of an **enum**, or a group of constant values, to specify the type. Using the same message example as in the builder pattern, you can instead of a class that creates a message while passing in an enum to specify the type to create. *Figure 11.6* demonstrates the flow of a simple factory:

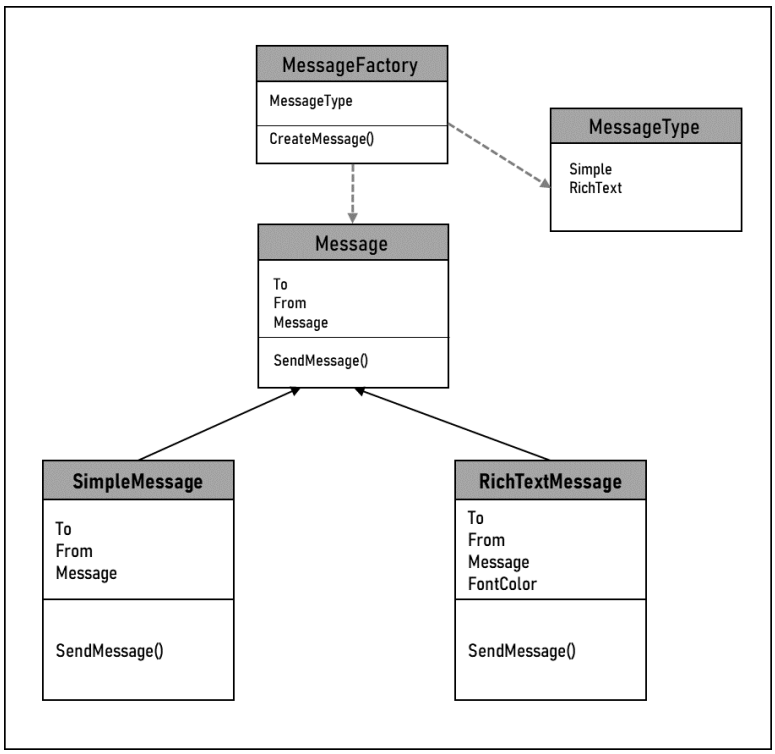


Figure 11.6 – Simple factory

The simple factory uses the `Message` class from the `Mercury` application as an example. The `MessageFactory` class takes in an enum to specify the type of message that you want to create and

creates the correct instance. Notice how the `RichTextMessage` class has an additional property that neither the `SimpleMessage` class nor the `Message` class has that has data on the font color. The number of properties and methods can vary greatly between the objects created within a single simple factory.

Singleton pattern

A **singleton** is a pattern that ensures that only a single instance of a class can exist globally. This is done through the use of a static object within the class. A static object can't be instantiated directly, and thus only one can exist. *Figure 11.7* illustrates the behavior of a singleton class:

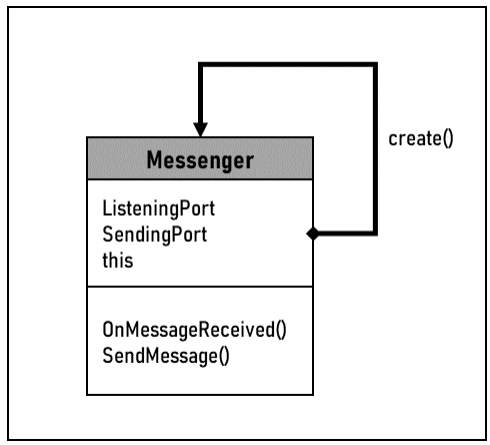


Figure 11.7 – Singleton

The `Messenger` class from the `Mercury` application is an example of a singleton class, as only a single instance of the `Messenger` class can exist. In this diagram, you can see that the class has a reference set to itself that is returned when asked for the object. If the object doesn't exist yet, an instance is created as referenced.

As you can see, the creational design patterns will be a common occurrence as they relate to the creation of objects. Knowing the patterns and their names will help you interact more smoothly with your development team by having a common understanding of basic concepts so that you can focus on the problem you are trying to solve instead of trying to understand what the code or a design is actually doing.

Let's now look at the other pattern that is often discussed in design reviews: structural design.

Structural design patterns

Whereas creational design patterns focus on object creation, structural design patterns take a step back and discuss how system components are related to one another. Though these do have in-code representations, and that is often what people look to for examples, I'll be using the design approach as these patterns will come up in the system designs and architectural landscapes that we'll cover in the next chapter.

Adapter pattern

The **adapter pattern** is used to convert between different object types, which often may serve similar purposes. From a code perspective, it's similar to an abstract class and classes that extend them. From a system perspective, think of a currency conversion system. Each bank has its own currency conversion system, and you may need to interface with multiple banks depending on the currency you are converting. Instead of custom integrations for each bank, you can use an adapter to take your common data model used by your internal systems and convert it to the specific need of the bank you are calling. *Figure 11.8* illustrates what this might look like:

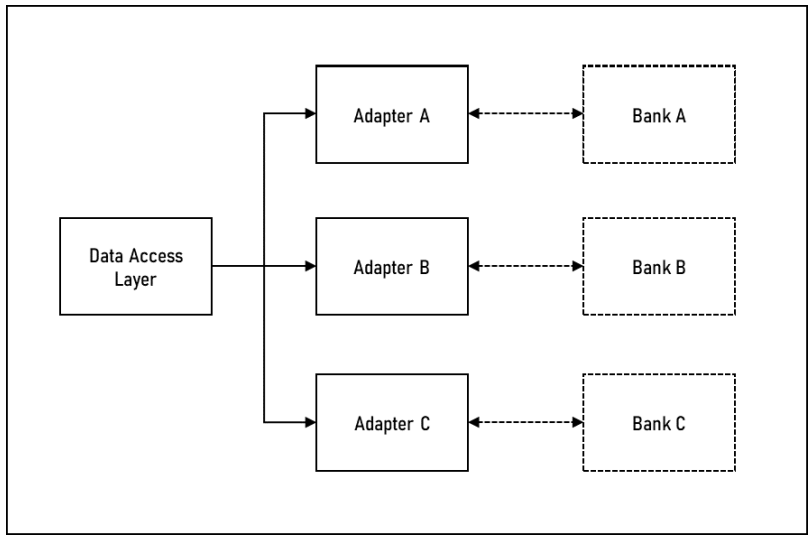


Figure 11.8 – Using adapters to connect with multiple currency converters

As you can see, the adapters act as a type of bridge between the internal systems and the needs of the external systems. Each adapter takes the internal data model and adapts it to a conversion request that is specific to the bank that it is calling. The data returned from each bank is then converted back into the internal data model. In this way, it encapsulates the details of those differences so that the rest of the system doesn't need to know about them. So, if *Bank A* were to change its API, only *Adapter A* would need to be updated and the rest of the system would not need to be updated.

Decorator pattern

The **decorator pattern** allows the addition of new data to an existing object without changing the underlying object. Using the adapter example of multiple bank interfaces, each bank will have a different response that has different data in it. The adapter will fit that data into the internal data model, but you may need to append additional data to this that is not related to the bank call. This might include adding additional context to the data model that isn't available at the time the conversion takes place or wasn't relevant to that call—such as full customer information. Though it shouldn't be sent to the bank, it may need to exist in the model so that a decorator can be invoked to add this information back in. *Figure 11.9* shows how a decorator pattern would work:

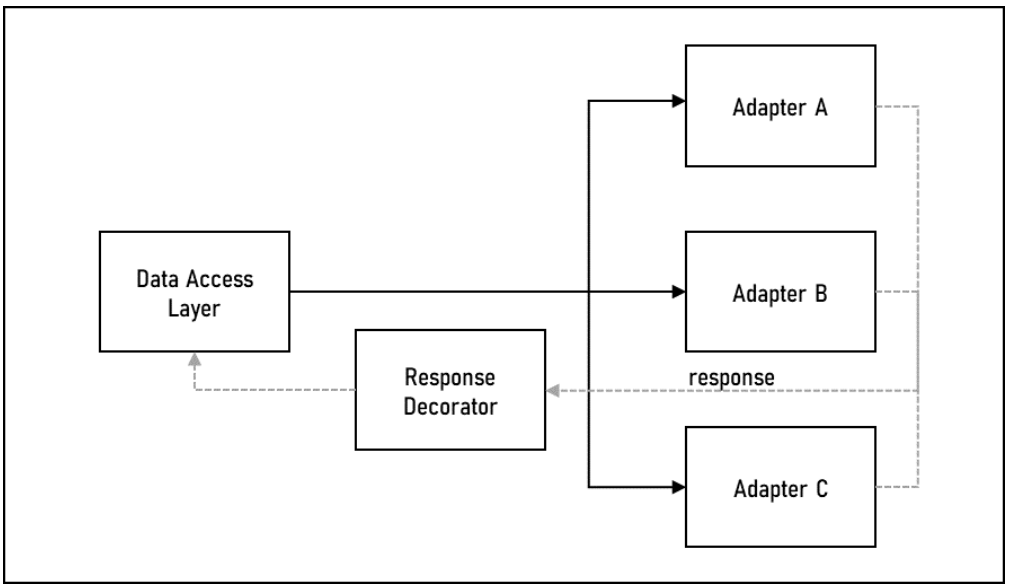


Figure 11.9 – Using a decorator to add data to a response

Using the adapter example from *Figure 11.8*, the response back to the data access layer takes a slightly different path through a response decorator. In practice, this may be multiple decorators or a single one that can add data, depending on the adapter that was invoked.

Façade pattern

Last but not least, the façade design pattern is used to simplify component interactions by reducing the number of systems that need to be interfaced with. *Figure 11.10* illustrates a system map before and after a façade is introduced:

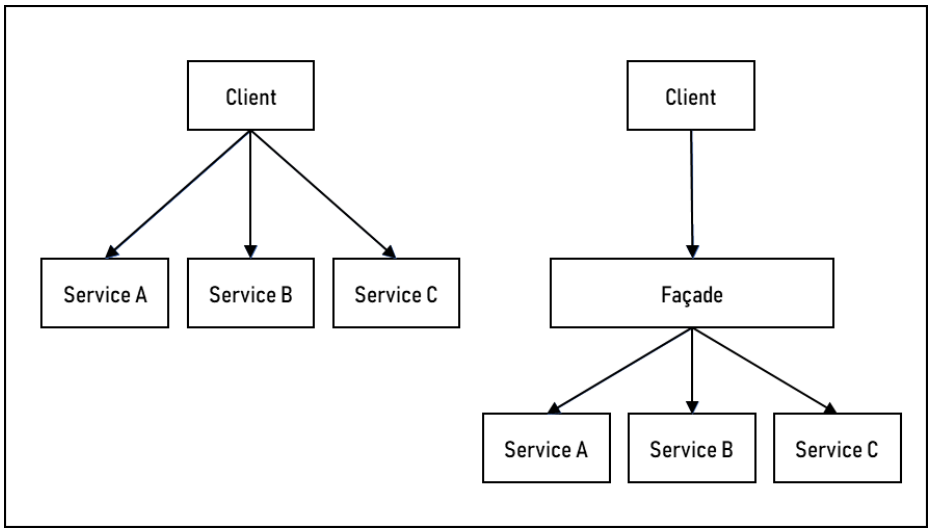


Figure 11.10 – Using a façade to simplify client interfaces

The façade on the right side of the figure reduces the number of connections between the components. It simplifies the landscape by reducing the amount of information each component (or service or client) needs to know about the landscape by delegating inter-system calls to a single component. This also has the added benefit of allowing underlying system changes and re-architecturing without impacting the calling clients.

Summary

Code development makes up a large portion of the deliverable work in a tech industry project or program. As a TPM, we are responsible for this work, though do not do this work ourselves. To ensure that we are successful in our role, we discussed the styles of programming that are most in use right now—OOP and functional programming. We learned that many languages allow for a mixture of the two styles in the same code base and that certain styles have benefits for particular tasks.

We learned about data structures that are commonly used and how their performance is measured using space and time complexities expressed through big O notation. We learned about simple list-based structures as well as more abstract structures, such as a graph that represents non-linear, unstructured, and complexly interrelated data.

Lastly, we discussed both creational and structural design patterns, which are foundational to feature and system designs. As such, these patterns will be useful for any design reviews you are a part of as a TPM.

In *Chapter 12*, we'll dive into system design and architectural landscapes. Both of these are technical tools that you will be relying upon a lot as a TPM. We'll explore both good and bad system design concepts using the `Mercury` application landscape to dive deeper.

Further reading

- *Learning Object-Oriented Programming*, by *Gaston C. Hillar*

  This is a great introductory book OOP, starting with a basic real-world understanding of objects and methods. If you are unfamiliar with OOP or want an in-depth refresher, this is a good place to start.

  `https://www.packtpub.com/product/learning-object-oriented-programming/9781785289637`

- *Mastering Functional Programming*, by *Anatolii Kmetiuk*

  This book uses both a traditional functional language, Scala, as well as an OOP language staple, Java, to teach the foundations of functional programming. It then goes beyond the basics to get you comfortable using functional programming concepts and styles in your day-to-day programming.

  `https://www.packtpub.com/product/mastering-functional-programming/9781788620796`

- *Hands-On Design Patterns with Java*, by *Dr. Edward Lavieri*

  This book gives you a real hands-on approach to learning a large number of design patterns using Java. All design patterns I covered are also covered here at a greater depth, making it a good next step to dive deeper.

  `https://www.packtpub.com/product/hands-on-design-patterns-with-java/9781789809770`

- *Everyday Data Structures*, by *William Smith*

  This book discusses data structures as well as algorithms, a cornerstone of computer science, in great depth. It uses hands-on programming in various OOP languages to explore each data structure and algorithm. All the data structures I discussed are in this book, and I encourage you to dive deeper using this book.

  `https://www.packtpub.com/product/everyday-data-structures/9781787121041`

12

System Design and Architecture Landscape

In this chapter, we'll cover two of the most important technical skills that a TPM possesses: **system design** and **architectural landscape design**. These are the levers we use to influence the technical direction of our organizations.

We start out in our career focusing on system designs in individual projects and influencing the right design for the requirements and services. As we grow, we start looking at the architectural landscape around our projects and programs to see patterns of opportunity and areas of risk. We start to influence the teams around us and the organization as a whole.

We'll explore system designs and the architectural landscape through the following:

- Learning about common system design patterns
- Seeing the forest and the trees
- Examining an architecture landscape

Let's dive in!

Learning about common system design patterns

As a TPM, you split your time between the high-level scope, which spans across multiple systems, and the weeds of a specific feature design. It's due to this breadth and depth that system design is one of the most important technical skills a TPM can have. It's important enough that it shows up in most interviews for the bigger tech companies. I'll cover the aspects of system design that you need to consider to ensure that your design is well thought out.

When we think about system design, we often conjure up a diagram of multiple services, each covering a single function or area of concern. However, system designs come in many different sizes and complexities. On a smaller scale, a feature design, such as a feature to add a new contact to your contact list in the Mercury messenger app, is its own system design. Somewhere in between these is a system design for an entire desktop or mobile application.

As a TPM, you need to be prepared to work with a design at any of these levels of complexity. Many designs will comprise more than one design pattern, especially for complex systems. The names and behaviors of these patterns may be used in the designs themselves and therefore need to be understood. Knowing the behaviors and key features of design patterns will also make evaluating them more effective. As such, I'll walk through some of the more predominant system design patterns (often referred to as architectural patterns) that are in use today.

Model-View-Presenter

The **Model-View-Presenter** (**MVP**) design pattern is a variant of the **Model-View-Controller** (**MVC**) pattern. Where MVC is popular in client-server applications, MVP is used often in desktop applications:

- The **model** is the data model of the application. All the data manipulation and storage happens in this layer.
- The **view** is the user interface at the top of the diagram. This displays all of the data in the application in ways that make sense to the user.
- The **presenter** sits in between the model and view and acts as the connecting layer between them. The user interacts with the presenter, usually through input fields in the view such as text boxes and buttons, which triggers events. The events live in the presenter and make changes in the model based on the event that occurred.

Let's try to understand this better with the help of an example:

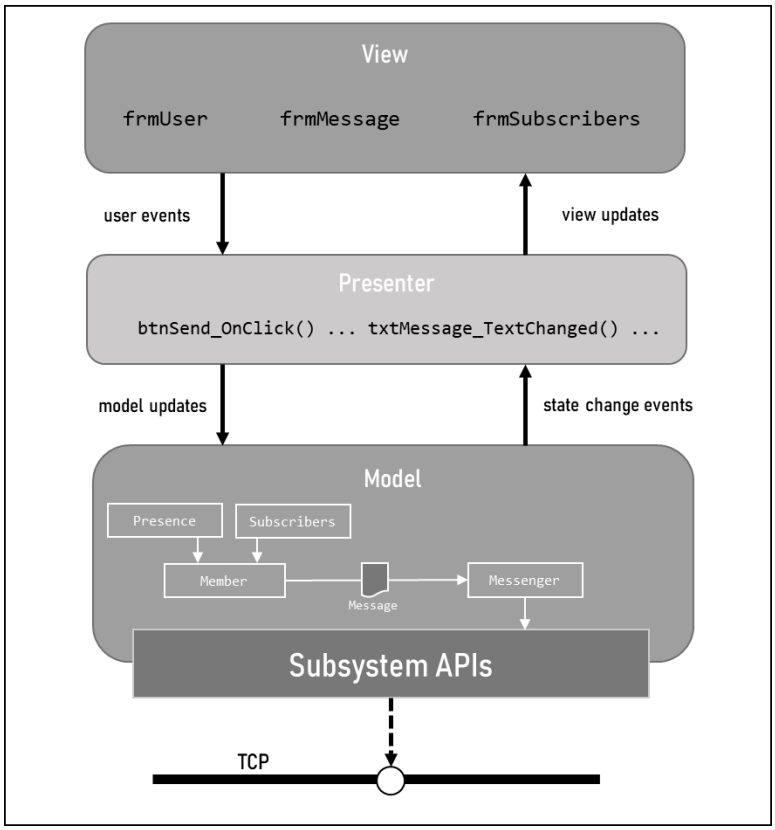


Figure 12.1 – Windows Mercury application system design

In this example, the model is broken down into three sub-components, the first of which are the business objects. These are the classes that house the data for the user (the member), their messages, their contacts, and their subscribers. The second is the messenger class that handles sending and receiving of messages across the network. Lastly is the subsystem that is shared across the operating systems that talks to the TCP network layer (via system APIs) to transport the messages.

The view displays the data model based on the context of the user actions from the presenter. For instance, the `PresenceInformation` class may be displayed as a colored dot beside the username of the client logged in, where the color represents a specific status. The user may also click a drop-down button to update their status, which would be relayed back to the model as an update to the `PresenceInformation` class via an event in the presenter on the `IndexChanged()` event for the dropdown.

Object-oriented architecture

Object-Oriented Architecture (**OOA**) is where you use objects to model real-world concepts. Though this seems straightforward due to **Object-Oriented Programming** (**OOP**) being widespread, it does not dictate what the objects themselves are and, in many cases, the objects created are bespoke to the problem space that the application is in. However, in many industries where the software is a byproduct of or in support of the main business, the software tends to emulate real-world functions that are digitized. Actions such as filling out forms give rise to form-based applications centered around this architectural concept.

Figure 12.2 shows how OOA applies to a form-based application.

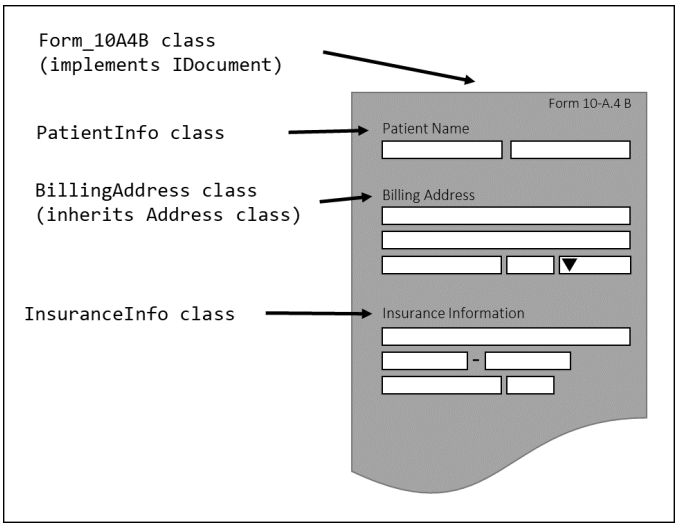


Figure 12.2 – Object-oriented architecture

The preceding figure shows a fictional US healthcare form on the right-hand side with typical information, such as the patient's name and billing address, as well as their insurance information. Each of these bits of information is represented by a class, and these are all combined into an object that represents the form. Grouping objects together and inheriting through abstract classes, such as a generic `Address` class, highlights **abstraction** and **inheritance**, tenets of OOP that OOA takes advantage of.

Domain-driven design architecture

Domain-Driven Design (**DDD**) is a concept where domain experts work alongside the development team to influence the data model and data flow of an application. This is used in domain-heavy scenarios such as healthcare and financial systems where specific procedures, nomenclature, and protocols need to be followed.

On the surface, this feels very similar to OOA since OOA models real-world objects. However, you can model real-world objects in some cases and then not in others. In the case of DDD, the entire design is based directly on the domain, and, in some instances, the user interface is a direct reflection of the underlying model. This type of stricter coherence is not the goal of OOA.

DDD does not require the use of an OOP language, but there are natural similarities. However, there are cases where a **Domain-Specific Language** (**DSL**) such as **SQL**, which isn't an OOP language, is used in conjunction with DDD for relational databases.

Event-driven architecture

Event-Driven Architecture (**EDA**) is where the application's data model is updated through user-triggered and system events, which often signal to change something about the data model state. As we saw in the *Model-View-Presenter* section, the presenter utilizes events that are triggered by actions taken by the user on the user interface (for example, clicking a button). As such, EDA is often associated with OOA and can also be used in conjunction with DDD.

P2P architecture

This book covered P2P in *Chapter 3* to some degree, but it is worth expanding upon here. There are two types of P2P networks, structured and unstructured.

A **structured network** has a strict topology that is adhered to as hosts leave and join the network. Every host has a list of nearby hosts, or neighbors, so that any one host can easily access another host. This type of network results in a lot of discovery traffic being broadcasted but allows you to find a peer or file quickly.

An **unstructured network** has no imposed structure and is resilient to changes in hosts dropping and adding to the network. However, finding a specific host or file is not guaranteed. This is a problem particularly when using P2P for file sharing, as rare files will be just as rare to successfully find. For a messaging app, the biggest hurdle to this type of structure would be finding your contacts on the network, if relying strictly on searching. A way to share direct-connect information between two parties would alleviate some of this pain.

Service-oriented architecture

Service-Oriented Architecture (**SOA**) is arguably the most-used architecture in the tech industry today. SOA is the basis for cloud-based architectures and offers a logical separation of concerns in larger organizations.

Figure 12.3 takes the Mercury desktop application from *Figure 12.1* and re-imagines it in an SOA setting. Let's take a look at how this changes the design and behavior of the system.

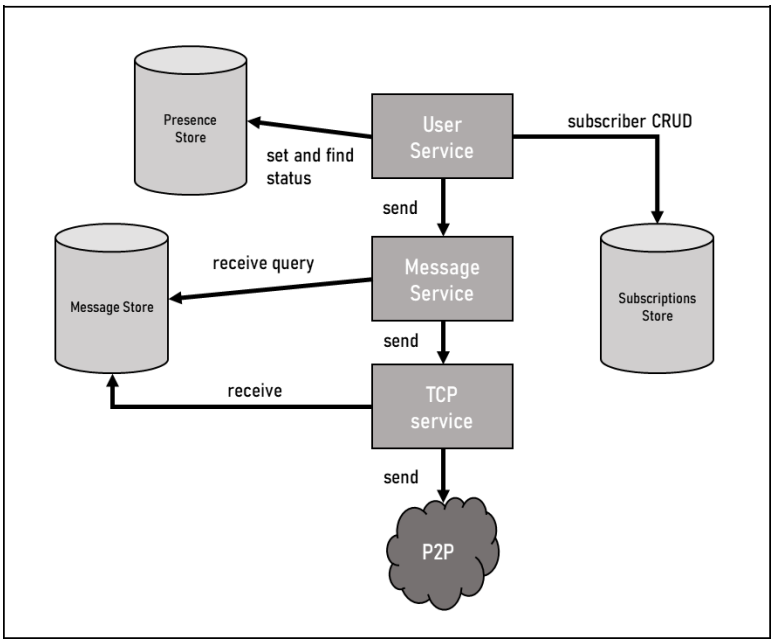


Figure 12.3 – Mercury re-imagined as an SOA

In the preceding figure, the three sections of the model from the MVP are now distinct services. **User Service** is essentially the user model objects and controls all aspects of the user. Now that what was a single desktop application is represented by multiple services and is decentralized, we need storage mechanisms to house what might have been in memory or serialized to the local disk in a desktop application. **User Service** connects to **Presence Store** as well as **Subscriptions Store**. **User Service** connects to **Message Service**, which is an analog for the `Message` class in *Figure 12.1*. It connects to **TCP Service**, which handles both sending and receiving messages. This is the most important difference between the desktop system design and the SOA design. Since this is a distributed system, a user isn't logged into a specific physical device and thus their location on the P2P network isn't static, so a user cannot reliably be reached to receive inbound messages. **TCP Service** circumvents this issue by storing incoming messages in a data store that a user can query for unretrieved incoming messages.

Although not a practical example given the requirements of the P2P messaging service, this does serve as a good illustration of where certain aspects of a design might change. The lack of a centralized system often leads to a high reliance on centralized or synced data stores to handle stateful data. On the other hand, having individual services allows for scaling to be handled on the pieces of the system that need it and not others. It also serves as a concrete mechanism to ensure the separation of concerns by forcing data contracts between any two services that interact with one another.

Client-server architecture

A client-server architecture is a classic design used to connect a user to a centralized service. This is the architecture used to browse the internet and to connect to a specific website. It is also used in mobile architectures where the mobile application (the client) connects to the application's backend on a centralized server to send and receive data.

Throughout the book, I've been referencing a program that builds a P2P-backed messaging system across multiple operating systems. A P2P system is relatively straightforward, so I'm going to expand the program requirements for the sake of a more representative system design. For this exercise, I'm going to add the requirement of a web-based interface for the messaging application. It will still be P2P-backed and the operating system-specific applications are still present, but this design will focus on the web interface.

Figure 12.4 showcases the system design of the web portal interfacing with the P2P network.

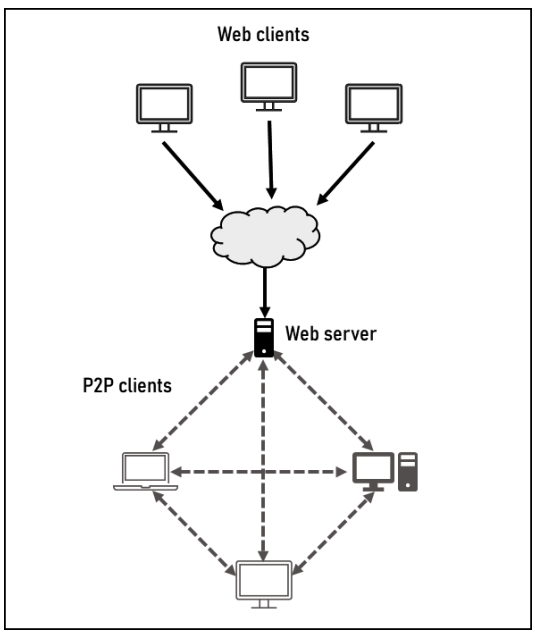


Figure 12.4 – Web browser Mercury system design

This system design is a hybrid between a traditional client-server pattern and a P2P pattern, with the biggest difference between this design and the one used in *Chapter 3* being the addition of **Web server**. This server is the gateway for the web-based interface. Once a user connects to the web portal, the backend behaves in the same manner as a single host on the network would because the server is just another host in the network. The difference is that this particular host – or fleet of hosts – maps to multiple users instead of a single user on a single host.

This design isn't perfect, as it is a first pass at integrating the two patterns. This is a good time to talk about some design considerations that can help when working with any design and then we'll take a second look at this example.

Design considerations

These design considerations will help ensure your designs convey a clear intention, which will reduce project churn and has the added bonus of helping you during an interview for system design. Pay close attention to these concepts:

- *Avoid vague design traits*: You want your system design to be clear, so make sure you are not generalizing aspects of the design so much that you lose your key understanding of the interactions between components. Ensure that pointers between components are directional when direction matters. For instance, a network topology doesn't need to specify directional flow, as data flows in both directions on a network. However, when a component is accessing a read-only datastore, the direction makes a difference and conveys the read-only nature through the data flow.
- *Don't omit key functionality*: Missing out on functionality is usually due to an expectation of familiarity where the missing piece is seen as obvious – or parts of the design are overly generalized, which blurs the functionality between two or more parts of the system.

Latency

As is the case in most companies in the tech industry, **latency**, or the amount of time for which a request and response are processed, is of the utmost concern when creating a system design. Long responses, or even perceived long responses, can turn users away from using a system. This could mean lost sales revenue through not placing orders, missed ad revenue, or a reduction in market share by driving users to the competition. For these reasons, you must ensure that your system design takes latency into account when it's applicable.

There are a few ways to combat latency that you can consider depending on how they fit with the needs of your system. If you have a lot of microservices, you can minimize network hops by co-locating services onto a single server. If you have a data store, ensure that the right data store is chosen based on the needs of the system. A system that focuses more on reads than writes, for instance, should have a data store that is optimized for reads that will reduce the overall latency of the system. The system design can show the specific database being used to highlight the high read or high write capabilities.

Though not easy to show on a system design, you can hide latency by increasing the amount of asynchronous data processing to reduce the overall time for the call to process. This is akin to swarming a project with multiple resources to bring the calendar time down.

Design patterns have a varying impact on the latency of the system, so the latency requirements both now and projected in the long term should be evaluated when selecting the right patterns to use. This does not mean you shouldn't use a pattern just because it introduces latency, but that measures

may need to be put in place to counter the latency introduced. These long-term considerations of the requirements and needs of the system are key contributions from a TPM during the design process.

Availability

Availability, or the ability of a system to respond to high volumes of requests without experiencing an outage due to no resources, is a key performance indicator that is tracked in every client-service architecture. As this architecture pattern is utilized to make thousands or millions of connections to a system, ensuring that the system can handle that load and not fail is critical to the success of the system. To mitigate availability outages, you need to increase the number of hosts that are ready to take requests. In order to ensure that the web client doesn't need to know how many hosts you have and where they are, you can utilize a load balancer service that takes all requests to the web client and distributes the requests across the fleet of servers. Most load balancers can detect server outages and switch off traffic automatically to a misbehaving host to reduce the number of bad responses as well. In some cases, a fleet of load balancers may also be needed. Equally, when increasing hosts, other factors such as concurrent database connections need to be adjusted as well and may factor into which type of database you use for your system.

Scalability

Scalability, or the ability to quickly respond to a fluctuating call volume, is often discussed at the same time as availability, as they have similar solutions. Once the servers are behind a load balancer, servers can be added or removed without compromising client connections. In many systems, the ability to add and remove on-demand is key to scalability, as it can save on costs to reduce the fleet size in off-peak times and only have a large fleet when it is needed.

Now that we've discussed some common concerns with client-server architecture, let's take a look at the architecture diagram again.

Defendable choices

Above all of these individual design considerations and patterns, you'll want to ensure that the design you have created is defendable. There is more than one way to solve a problem and a system design is no exception to that rule. As such, you need to understand the choices you have made in the design and why you made them. I've found that in most interviews, so long as you understand the trade-offs you are making, the outcome is often favorable. You can always be taught new ways of solving problems, so the goal here is more to demonstrate that you can think critically and adjust as new data comes in that may impact your design considerations.

If you are already a TPM, then you will often review designs from your development team and can use these concepts to ensure that the design holds up to scrutiny. Aside from these high-level checks, you will also look at the designs and see how they hold up to the other projects and programs that are in flight and are there to ensure that other teams, services, and projects are considered in the design.

Now that we've discussed some design considerations, let's take a second look at the client-server architecture from *Figure 12.4* with some of these considerations applied to *Figure 12.5* here.

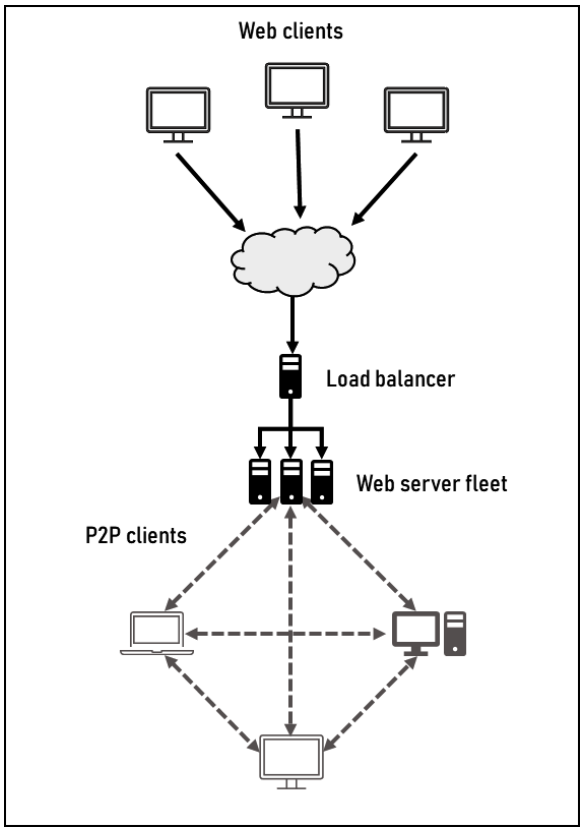


Figure 12.5 – System design with availability and scalability mitigations

As you can see, the network diagram now has a fleet of boxes for the web service, all backed by the added **Load balancer** to distribute the requests. As the Mercury application is a single local application, there are no latency concerns that need to be addressed at the system design level, although this will be revisited at the architectural landscape level. The availability and scalability have been addressed through the addition of a load balancer, which supports multiple web servers to ensure enough connections are available during peak demand. As with *Figure 12.3*, this design has a similar concern where a web client doesn't have a persistent connection to the P2P network, so inbound requests won't know which server to send the connection to. A similar database solution that stores incoming messages that the web client can query would be one solution to this.

Now that we've talked about all of the system design patterns and how to make a good and defensible design, let's expand our scope and learn to see the forest and the trees.

Seeing the forest and the trees

The architectural landscape is not often talked about as a standalone topic outside of specific instances such as migrating from on-premises to the cloud. In many ways, it's similar to system design in its intention, as well as patterns. In most cases, the design of a single system within a larger ecosystem will match the design patterns of the systems around it. If the trend at your company is to utilize SOA, then you will see SOA at every level. The biggest difference between the architectural landscape and system design is the scope that the design encompasses.

You can't see the forest for the trees is a proverb that was first published in 1546 in *The Proverbs of John Heywood*. The idea is that from the middle of a dense forest, you can see every tree that surrounds you in great detail from the trunk all the way to the crown of the tree – but from this vantage point, you literally cannot see the whole forest; you don't know how vast it is or the directions in which it flows, as the breadth and depth are lost. This saying has become so common it is now an idiom in the English language.

People often relate to this idiom when they are too close to a problem to see the bigger picture or have been too close for too long to see clearly. Think back to a time when a problem was frustrating you and you worked to solve it with no progress. Then, you stepped away from the problem and when you came back, the solution came to you quickly! It came because you changed your perspective by allowing different information to come into play. Simply put, you saw the forest.

> **Pro tip**
>
> I offer a slightly different version of this proverb as a directive to all TPMs: *You must see the forest as well as the trees*.

A TPM must have both a breadth and depth of knowledge. The depth (or the trees) refers to system designs and the breadth refers to the architectural landscape (or the forest). You can extend this analogy further and see where a system design and an architecture landscape are one and the same; when no other systems surround the one that the system design describes, then it is both the system design and the architectural landscape, like a lone tree on a hill. However, you can then zoom out from that tree far enough to see other trees and understand where it fits into the larger picture of the trees around it. That is to say, no modern system is in isolation, but the level at which you focus will depend on the needs of your role. It may be at the team level, organization or department level, or company level.

Important tools we use to see the bigger picture better (the forest) are program, product, and team roadmaps. These roadmaps encompass different components of an architectural landscape. Each one looks into the future deliverables of a group of systems. More importantly, they describe either directly or indirectly the intention of that group of systems, why they exist, and what they are striving to become. The intentions, as well as discrete deliverables, are what determine the picture of the architectural landscape both as it exists today and how it will change in the future. Some companies publish technology directives that give the general direction that the company's platforms are evolving toward, which can include the technologies that will be centralized.

A TPM will be expected to pick up on these cues from roadmaps and directives and point out when there's an issue or an opportunity between multiple projects or programs. A Principal TPM is expected to see these issues or opportunities without being prompted whereas a Senior TPM is expected to see these connections as they relate to the projects or programs they run.

On the other hand, we also need to recognize the individual trees. Your contribution to design reviews at the system level will lean on your knowledge of the architectural landscape as you set outside context against the proposed system design. In general, an SDE's focus is on the system design and its interactions within the context of their own software. Just like a TPM, the SDE's breadth also grows as they ascend the corporate ladder. Even so, the TPM's job of looking across projects and cross-organization is always required.

Figure 12.6 illustrates the varying areas of concern of a TPM, SDM, and SDE across an architectural landscape.

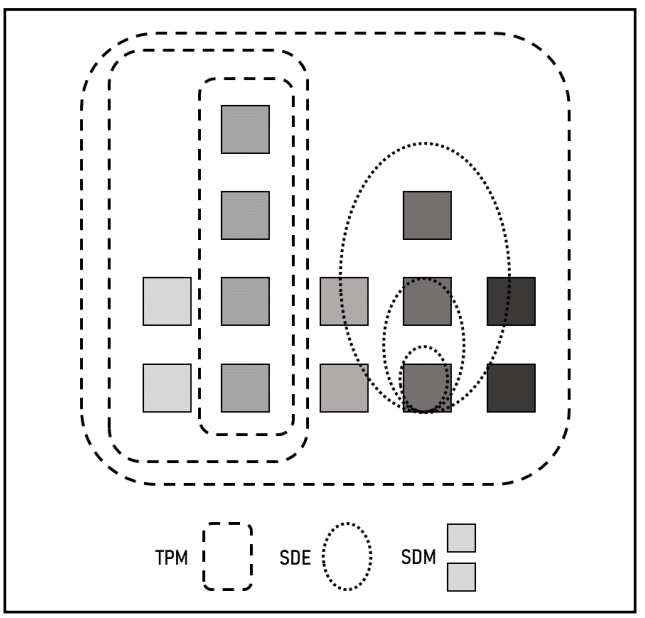


Figure 12.6 – Areas of concern across job families

In the preceding figure, each column of boxes represents the services owned by a single **SDM**. These services are the purview of our **SDM**, and they are expected to understand each one in its entirety, as well as how they interact with each other. It's often necessary for our **SDM** to know about the services outside of their team that have a relationship with one or more of their own services. In these instances, the **SDM** area of concern will look similar to the Senior **SDE** area of concern.

The circles represent the three main areas of concern for our **SDE**. The smallest inner circle is an entry-level **SDE** who is mainly concerned with a single service. An industry **SDE** will need to stretch

to related services, often under the same **SDM** that they report to. They know how these services work and how they interact with one another and are starting to learn about outside dependencies. The largest circle is the Senior **SDE** who needs to know about all services under their **SDM**, as well as the services that have a relationship with them from other teams.

The last group is our **TPM**, represented by the rounded squares in the figure. An entry-level **TPM** starts out with the same expectations as an industry **SDE** in that they need to know the complete architecture of their **SDM** (also known as an **embedded TPM**) or the services they are focused on. An industry **TPM** is expected to know about the dependent systems of their focus area as the initiatives going on in that space. Lastly, a Senior **TPM** is expected to understand the architecture of their organization – not necessarily to the same level of detail as with the services they focus on, but at least the high-level data flow and interactions.

This illustration is a common example of the areas of concern and how they overlap but is by no means the only breakdown. Highly specialized TPMs may focus more on the depth of their services or on certain aspects such as security. In some organizations, the Senior SDE's purview may be the same as a Senior TPM, especially for a seasoned SDE on the path to Principal. In cases where a TPM isn't present, an SDM's area of concern and influence may also need to expand to fill in the gap.

We've focused on seeing the trees through various system designs and how this aligns with your day-to-day role as a TPM. We also see how the areas of concern grow as your level grows. Next, we'll take a look at the forest by learning what an architectural landscape is and how it is different from a system design.

Examining an architecture landscape

To get a good understanding of what an architecture landscape is, we'll compare a system design with an architecture landscape. We'll follow this up with a look into the implementation of the *Mercury* messaging application on a corporate network.

There is more in common between a system design and an architectural landscape than not. The design patterns between the two are the same and are often referred to as architectural patterns. They also both describe the relationship between components of an ecosystem that share some relationship either in the data they process and handle or the function that they collectively perform.

Where they can differ is the scope and depth of the design. A system design is limited in scope, as it often covers a single feature or limited data flow between highly related systems. The design may dive into API definitions, as well as illustrate the data model and how it flows through the system.

An architectural diagram usually covers higher-level system interactions and multiple systems and features that share a common theme. APIs and data field flows are less relevant at the level of the architectural landscape.

To start, in *Figure 12.7*, we compare a system design to an architectural landscape.

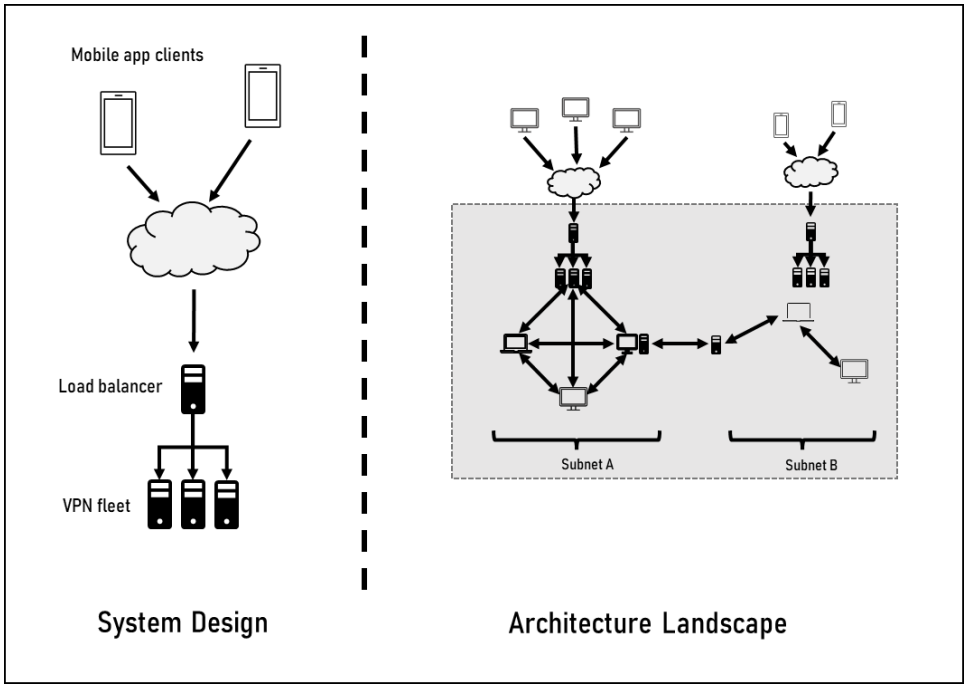


Figure 12.7 – System design versus architectural landscape

In the preceding figure, the left-hand side is the system design of a **Virtual Private Network** (**VPN**). A VPN provides a secure connection to a specific network as though your device were on that network and is often used in corporate settings to give employees access to the corporate network while working remotely. This design describes a single feature of gaining access to a network through a load-balanced VPN.

On the right-hand side is the architecture diagram for the *Mercury* application implemented in a corporate network. On the upper right-hand side of this picture, you can see the VPN system design within the architectural landscape. The landscape tells the story of the entire *Mercury* system, including all points of entry and network traversals. It does not detail the *Mercury* application as we saw in *Figure 12.1*, but it does include multiple independent system designs, including the web-based client system design from *Figure 12.5*.

Next, let's take a closer look at the architectural landscape including labels, and discuss the various components. *Figure 12.8* shows the Mercury corporate installation in greater detail.

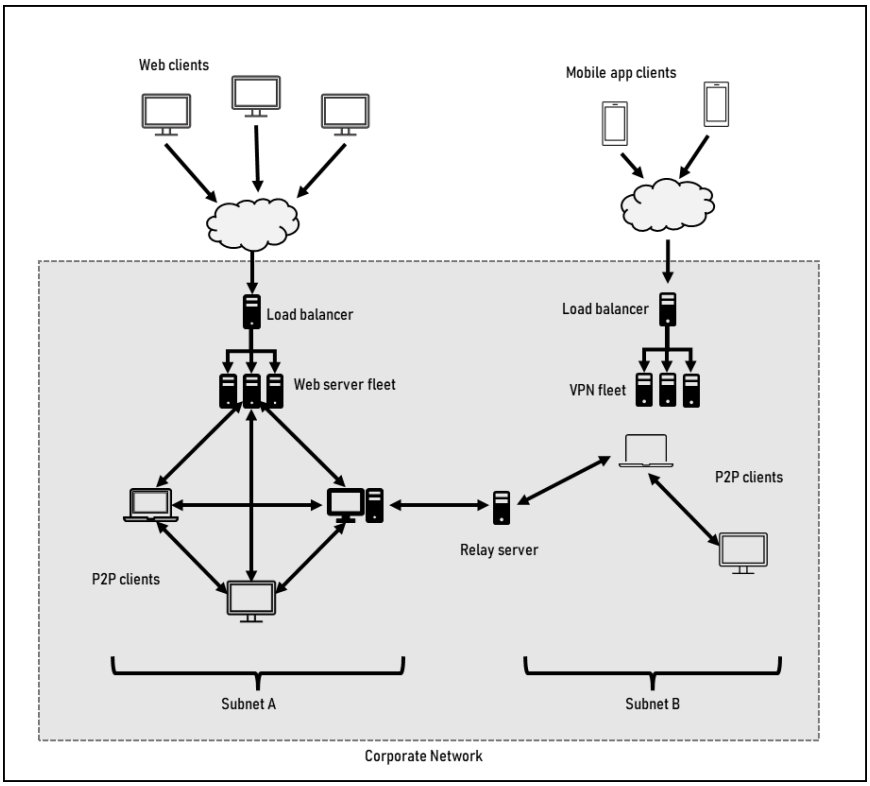


Figure 12.8 – Mercury corporate installation architectural landscape

We've seen various parts of this architectural landscape throughout the book, starting with the P2P network on the right-hand side. This is representative of an unstructured P2P network and is contained within the same network as well as on the same subnet – in this case, denoted as **Subnet A**. The web client that we added in this chapter to illustrate a client-server architecture is above the P2P network and connected to the network as **Web server fleet** where each member of the fleet is on the network.

New to this architecture landscape is the second subnet, **Relay server**, and **VPN fleet**. A large corporate network may need to be segregated into multiple **sub-networks** (**subnets**) to reduce network collisions by having fewer devices share the same physical wire for data transmission. In a traditional network, routers are used to transmit data across subnets. In a P2P network, all hosts have direct connections to other hosts, which does not include routers. To get around this limitation, the relay server that is part of the P2P network would need to receive a P2P packet and then relay it to the correct subnet to which it is also connected. Lastly, to get the *Mercury* application on mobile devices onto the corporate network, the design incorporates a VPN. Through the *Mercury* application on the device, it first establishes a connection to the P2P network via the VPN. In this sense, it behaves a lot like the web client in how it connects to and interacts with the P2P network.

To be clear, many of these additions to the architecture go against the original requirements of the *Mercury* program, specifically around not requiring any centralized setup or maintenance. On a hyper-local, single subnet network, none of these components are necessary and the simple P2P design would suffice. However, these are good additions to illustrate key design concepts and the overall complexity that can come up when working through requirements.

All of these system designs come together to paint a larger picture of how the *Mercury* messaging system could be implemented. As a TPM, I may be on the Windows team and thus largely focused on the internal system design of the application itself as seen in *Figure 12.1*. I might also be the lead TPM for the *Mercury* program and thus need to understand the full architectural landscape in order to effectively oversee all projects in the program.

Lastly, it's worth noting that both the architectural landscape and the system design utilized multiple design patterns to fit the needs of the ecosystem.

Summary

In this chapter, we explored the trees as well as the forest. Just as a forest is made up of trees and therefore trees and forests share a lot in common, so do system designs and architectural landscapes.

We learned about various design patterns that are used in both system designs and architectural landscapes. We discussed the elements of a good design, as well as a bad design. Above all, we discussed the importance of defensible choices, as there is always more than one way to design a system.

Finally, we dove into the differences between a system design and architectural landscape and how this relates to the areas of concern for a TPM throughout their career.

In *Chapter 13*, we'll close by discussing how to use your technical background to enhance your project and program management skills.

Further reading

- *Architectural Patterns*, by Pethuru Raj, et al.

  This book covers all of the system design patterns discussed in this chapter, as well as additional patterns. If this is an area of particular interest to you, this is a good place to start.

  `https://www.packtpub.com/product/architectural-patterns/9781787287495`

- *Solutions Architect's Handbook – Second Edition*, by *Saurabh Shrivastava, et al.*

  The work of a solutions architect is a popular field, as it focuses on moving from on-premises to the cloud. To do this, a full understanding of the current architecture is needed in order to determine the right solution for the cloud. As such, this offers a great view of understanding an entire architecture.

  `https://www.packtpub.com/product/solutions-architect-s-handbook/9781801816618`

- *Hands-On Design Patterns with Java*, by *Dr. Edward Lavieri*

  This book gives you a real hands-on approach to learning about a large number of design patterns using Java. All the design patterns I covered are also covered here in greater depth, making it a good next step to dive deeper.

  `https://www.packtpub.com/product/hands-on-design-patterns-with-java/9781789809770`

13

Enhancing Management Using Your Technical Toolset

In this chapter, I'll cover how the various tools in your technical toolset enhance your ability to manage projects and programs. I've discussed each of the key management areas in the first two parts of this book through various lenses and have touched on challenges you face in the technical space. Next, I talked about the technical toolset, which consists of code expectations, system design, and architecture landscape design. Now, we'll cover each of these intersections of the technical toolset and the key management areas and, lastly, explore how the toolset will also help you be a better leader in your organization.

We'll explore how to enhance management with the technical toolset by doing the following:

- Driving toward clarity
- Resolving conflicts
- Delivering through leadership

Let's dive in!

Driving toward clarity

A TPM is an extension of a program manager and, as such, it shares the trait of driving toward clarity. However, our technical toolset allows us to use specialized tools to tackle technical problems that would leave a non-technical PM struggling to process them. We speak the same language – quite literally – as the people who are delivering the project and can bridge the gap between technical jargon and the business. I've talked about driving clarity in various stages of a project and have discussed the technical tools TPMs use. Now, let's see how these two concepts help each other. I'll go through each key management area, from planning through risk management and stakeholders and communications.

Planning

During the planning stage, all three technical skills I've discussed are used. We'll go through each of the technical skills across three steps in the planning process: requirements analysis, project plan creation, and feature design. *Figure 13.1* shows the overlap between the planning process and the technical toolset and will aid in the discussion.

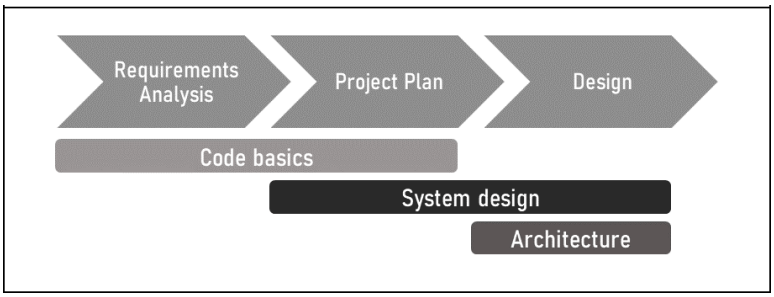


Figure 13.1 – Technical toolset during planning

Requirements analysis

The first pass through the requirements has you exercising your code knowledge to start to piece together the functional specification based on the requirements. You use the known system designs and architecture landscape to determine viable solutions and identify upfront risks, which we'll cover more in the next section.

Project plan

The more you know of the code base that your developers use and have an understanding of the programming language, the more you can begin to intuit the effort it would take to implement your solutions and start to lay out your project plan. For instance, in many applications, there are tasks that are performed often and follow a particular pattern. A good example is adding a new data type based on an existing abstract class and plumbing that data through multiple services. Knowing where the abstract class is located and where the data model is defined can help you piece together the various tasks for implementation, and then knowing the deployment timelines for the packages where the class and model are will give you an idea of the launch plan.

Strategic planning

In this example, I mention that creating a new class instance from an abstract class is a common scenario. If you notice, tasks are often repeated from project to project; this is a good area to dive deeper into and see if there is a way to automate or reduce overall time-to-production. Your next project might be a good place to add time to build out an automated solution. Watching for patterns is a key part of a TPM's long-term strategy planning.

Design

Once you have a plan drafted, you consult with your developers and SDMs to dive a bit deeper. At this stage, you are relying on your ability to bridge the gap from the business requirements to the technical requirements to help your team understand the scope and bring context to your proposals.

Risk management

Good risk management starts with good risk analysis and in order to analyze, you first need to understand what you are analyzing well enough to do so. That is to say, to properly analyze risk in the technology sector, you need to understand technology.

Figure 13.2 illustrates how the technical toolset allows for more effective risk management.

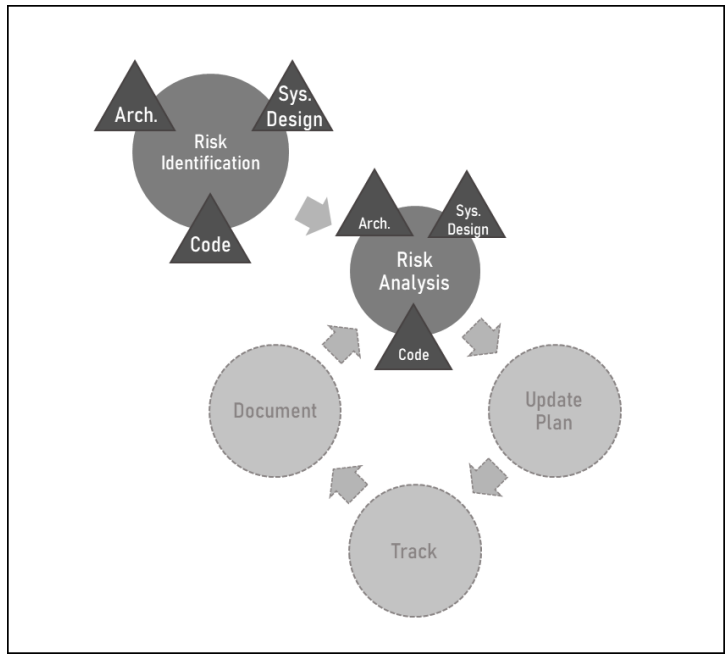


Figure 13.2 – Technical toolset in risk management

In the preceding figure, we are concentrating on the first two stages of risk management: **Risk Identification** and **Risk Analysis**. The last three in the cycle are mainly administrative and acting upon the work completed in the first two stages. As you can see, both of these stages include all three technical skills that we've discussed as represented by the triangles surrounding the stages. Let's discuss both stages in a little more detail.

Risk identification

Similar to how knowing the code base can help you create a plan, it can also help you identify risks. For instance, you might notice that a particular requirement will require a code refactor in a package that is currently being modified in another project, potentially causing a cross-project dependency and risking delivery timelines.

Knowing your code base and service dependencies can also help you identify risks in system designs. Introducing a new architectural pattern to an existing system or in a system that is new to the team is a risk to timelines as there are a lot of unknown unknowns when doing **greenfield development**.

Knowing your architectural landscape can help identify cross-organizational or even cross-team dependencies arising from a system design. These dependencies carry risks as you are reliant upon changes from another project, your changes are needed by another team, or it simply requires working closely with another project to ensure that no duplicative effort is done on either project. During design reviews, always ensure you are looking upstream and downstream of your system to ensure proper alignment in data flows, as well as at other projects that may be working in the same code packages concurrent with your project.

Risk analysis

During the analysis stage, you are determining which strategies can be planned for every risk that was identified. Each of the strategies involves varying degrees of technical acumen. Let's look at each one:

- The **avoidance** strategy may depend on code understanding and system design to know whether or not a risk can be avoided. As an example, if you know using a specific technology framework would add risk due to stability issues, you need to know whether alternatives to the framework offer more stability. It may be helpful to know whether other frameworks are used elsewhere in the company (architecture landscape) where using a known system may offer some synergy.
- The **mitigation** strategy would require similar knowledge as avoidance in that you would need to know whether specific mitigation is feasible. For instance, using additional resources to crash a task would only work if you knew the task could support additional resourcing. If the task involves a single code package, it may be hard to parallelize enough work for more than one person.
- The **transference** and **acceptance** strategies do not lean heavily on technical acumen as they are mainly project management trade-offs between time and scope. For transference, there may be some level of understanding of the technical requirement of the task and to be able to evaluate whether the party you need to transfer the task to is technically capable, though standard vendor evaluation procedures can help in these cases.

We've looked at how the technical toolset enhances both the planning and risk management areas, now let's explore stakeholder management.

Stakeholder management and communications

Our technical toolset naturally enhances our ability to work with stakeholders simply by the nature that many of our stakeholders are also technically minded. For those that are not, we act as a bridge between the technical details and clear understanding. Since we interact with stakeholders all throughout the project, we'll look at this from the perspective of the technical toolset and see how each skill offers ways to enhance our engagement.

Code basics

By understanding the code base that your teams work with, you can have open and unrestricted conversations with your developers. Your developers do not need to worry about translating the problem when talking with you and can just explain the issue in a way that is natural to them. This reduces the chances of misunderstanding and can speed up conversations by being able to speak on the same terms.

This isn't to say you won't have questions, but that the questions will be technical clarifications and deeper in nature. In fact, I encourage asking clarifying questions as this is key to driving toward clarity. Without knowledge of basic programming philosophies, the depth of the questions would be insufficient to clarify the problem to a point at which you can help.

System design

As a TPM, you contribute to project designs, from feature designs to full system designs that cover the project from end to end. Your technical background will help you navigate design reviews effectively. In these reviews, you tie the design back to the functional specification as well as the business requirements. I have often found discrepancies in designs where the business requirement wasn't fully understood, and it becomes clear that what seemed to be a clear understanding in a meeting translated to a misinterpretation. Your ability to read the system design and catch this early on during the design is a safeguard against late-breaking discoveries that take more time to fix and often lead to a project delay.

I recently had a discussion in a design review where two of the three design choices didn't meet the business requirements and I realized that the team didn't fully understand the background of why the project was being done. All the developers were new to the project and the months-long context building that happened before they came on board was lost to them. They saw the requirements text but didn't know why the requirements existed. So, I set aside time to go through the history of the project and what system limitations led to the need for the project. This context helped the developers understand why we were doing the project, which in turn helped them better understand their design choices and led to a better design that offered greater room for future scalability.

Architecture design

During a design discussion, the TPM is also responsible for ensuring the right people are in the room to make the most appropriate choice. To do so properly, you need to be able to speak about the design authoritatively and understand what the various stakeholders may need to get out of the proposal. In some cases, this may require looking beyond the system design you are reviewing and up to the architectural landscape. I have found that in large distributed systems, solving a data flow problem within a few local systems may not always scale to the entire ecosystem. This could lead to an ineffective design where the new data flow cannot be utilized upstream and downstream due to the client's – or even a client of the client – data handling. Looking at the larger picture, you may know of architecture evolutions that are in-flight and not present in the current architecture that should be examined to save future throw-away work. While this is true at the system design level as well, the SDEs and SDMs are more likely to be aware of these initiatives than of cross-organizational initiatives.

Now that we've discussed all three key management areas, let's see how your technical toolset helps you resolve conflicts in these same areas.

Resolving conflicts

Conflict resolution is one of the key methods by which we find a path forward as many issues are rooted in disagreement or misunderstanding between multiple parties. As such, any place where a TPM can run into an issue, conflict resolution can make an appearance.

Planning

Conflicts during the planning phase usually occur while analyzing requirements but can also occur while creating the project plan. We'll go through a few examples and see how your technical background can help resolve these conflicts.

During requirement analysis, disagreement on a requirement or an ambiguous requirement is common. As you work with your stakeholders, any clarification may introduce new complexities or insights and can steer the conversation organically. I have found that knowing my system behaviors and complexities helps during these conversations.

For instance, a recent project had a requirement to take a regional functionality and make it available globally for business expansion. I understood how the functionality was implemented and could use that to elaborate on specific inputs needed from the business team and was able to clarify ownership of data elements and expectations across multiple teams. Without knowing how the feature was implemented, this discussion would have come up much later during the design or implementation and could have led to delays by not having proper expectations about when and where data would be made available for the expansion.

Sniffing out implementation requirements

Depending on where your requirements are coming from, they will include some technical details and push for a specific implementation. Being able to recognize these scenarios will give you the chance to challenge the requirements and dig deeper to find what the underlying ask actually is. This is important because systems change over time and relying on a specific way to solve a problem can slow down the natural evolution of a system. Knowing what the actual need is allows the development team, along with the TPM, to find the best long-term way to meet the requirements.

Risk management

Risk management can involve conflicts in the identification and callout of risks as well as the risk strategies that are planned.

We discussed how your technical background helps identify risks in the planning section. Conflicts during identification are almost solely office politics in nature. As such, the skills needed here are mainly soft skills as people don't like a light shined on their team or service if it is called out as a risk either due to a cross-project dependency or code deployment difficulties. Leaning on transparency and ensuring language is as neutral as possible is the best course of action.

For risk strategies, there can be disagreement on whether avoidance, transference, mitigation, or acceptance should be planned for. In these instances, your technical background can be of help in resolving conflict. As an example, a project I was working on identified a risk where a new data field was being passed between services from different organizations. The risk was planning for the use case where the new field wasn't ready in time for system integration. The proposed mitigation was to pass a *null* value downstream so that the data field could at least be picked up and some work downstream could be unblocked. I knew that the downstream system could not handle null values in this context and that this mitigation would not work and so I challenged it. Instead, we mocked non-null data to unblock integration testing downstream. Though a straightforward example, my technical understanding of downstream systems allowed for a quick re-direction of the risk strategy to ensure we identified the right strategies that would actually achieve what was needed.

As an arbiter in many conflicts, a TPM's technical understanding of the problem at hand will aid in quicker arbitration. It also allows us to catch instances where the technical problem is not being conveyed properly across all parties – again acting as the technical bridge. This bridge can be between two technical teams and may rely on your domain and technical knowledge combined to successfully arbitrate between teams.

During a systems integration between two services, there was an error that was identified during testing. The error seemed to point toward one of the two services as the root cause though that team denied that the error was their problem. Upon investigation of the error code returned, I realized that there was a misinterpretation of a field as required by one of the teams. Essentially, one team assumed the

other team had made a field required, so they did the same, but it wasn't actually required. Knowing the two systems involved, and the requirements, I was able to catch the mistake and worked with the team to update their model to reflect that the field was not required in both services.

Stakeholder management and communication

Most conflict happens in stakeholder management as this encompasses most communication with the people involved in the project, from the developers to the team leaders. As it involves communication, it too is often focused on soft skills and tailoring your responses to the audience for clarity of thought and relevance. However, there are cases where your technical background plays a pivotal role in resolving conflicts with stakeholders. Let's take a look at a few examples.

During the execution of the project, you will be working with your developers on a daily basis in most cases and will attend stand-ups to help your team along and get an update on tasks. During stand-ups, blockers are often identified and talked about. It often falls on the TPM to act as a scrum master or at least help developers to unblock. Having a solid technical foundation will help you understand the risk quicker and allow you to act on it faster. As a TPM, there are times when the developers may look to you for technical guidance. Your insight into system behavior, client behavior, and code structure makes you a natural resource for trying to find the right solution to a problem. The TPM is also often the source of the functional specification that the developers reference during design and implementation, so quick consults with you can go a long way to quickly resolving roadblocks.

In the same way that your technical background helps you during stand-ups, it also helps when reviewing system designs. Part of your job is to challenge the system design when it doesn't fit the needs of the project and the needs of the organization. The other part is being an arbiter of conflict during system design reviews. As the TPM, you own the functional spec, you understand the requirements, and you know the architectural landscape, which gives you a unique perspective on the problem that the design is trying to solve.

A program is a good analogy of your role in a system design. In a program, multiple projects share a common set of goals, but each has its own set of goals that may seem unrelated if looking at the project on its own. As program managers, we ensure that the common goals are understood across all projects in the program and that every decision made in a project is done with full awareness of the implications at the program level. The same is true for a system design discussion where you bring knowledge and understanding of systems that are related and have an impact on the system design being reviewed and you ensure that the decisions being made on the design take the other systems, projects, and initiatives into account.

We've discussed many different ways in which your technical toolset enhances your ability to manage a project and resolve conflicts. Now we'll discuss how your technical foundation helps you deliver through leadership.

Delivering through leadership

As a TPM, you are a leader in your organization. You effect change through your programs and projects, but also as a contributor to the technical direction and strategies of your team. You identify problems but you also resolve them. Here are the ways in which your technical toolset helps you deliver through leadership.

As a leader, you will be expected to define the direction of your team. This expectation grows the higher up you go as a TPM. To do this, you are expected to find issues and define and drive solutions to fix them. This can be a difficult move for some who prefer to work within the bounds of a specific problem. However, defining a problem and solution within the confines of a project is similar to the confines of a team or organization. Just as a project has a goal, a team or organization has a charter. The most reliable way to find a problem worth fixing is to look at repeatable efforts across projects and recurring pain points during development. If the problem occurs often, the ROI is likely worth the effort to solve it. Use your knowledge of your system designs, architecture landscape, and even coding practices along with your backlog of project issues and risks, to help understand where problems may exist. The only difference between these is the scope in which you are looking to solve problems.

It isn't always your job to come up with the direction to go in but sometimes it is up to you to ensure that the chosen direction is being pursued. During project planning and design, you can reinforce the team or organizational direction by ensuring system designs and design patterns match the intended direction of the organization.

As an example, many companies have a technology directive that persuades the use of specific technologies or announces a shift from one technology to another. These are often forward-looking and gradual shifts, allowing teams time to make the switch. Knowing the expected technologies, if any, of your company allows you to be an arbiter of change. This can come up during project planning where you add additional time to allow for the migration from one technology to another for a single project, or even create a project to track a team-wide migration effort.

Summary

In this chapter, we discussed how your technical toolset can help enhance your project management by examining the key management areas of planning, risk management, and stakeholder management and communication. We looked at each of these in terms of driving clarity, resolving conflicts, and delivering through leadership.

Across all aspects of project management, a TPM's technical toolset allows for quicker, more efficient execution by providing foundational context and knowledge in the problem space. The toolset also enhances the TPM's leadership by effecting change in the technical direction of their organization.

A TPM is both a PM and a technical expert, and they are greater than the sum of their parts, as the old saying goes. It is the joining of the project and program management skills with technical acumen that acts as a force multiplier and allows the TPM to do their job better than a subject matter expert or a PM alone.

Index

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

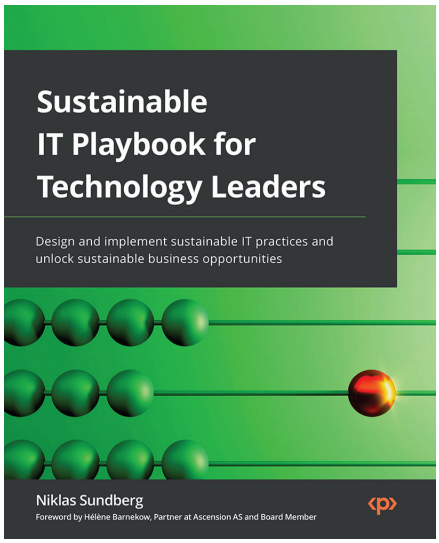
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

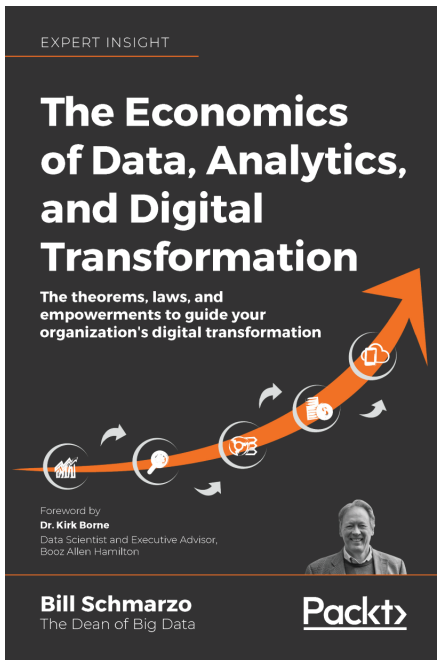


Sustainable IT Playbook for Technology Leaders

Niklas Sundberg

ISBN: 9781803230344

- Discover why IT is a major contributor to carbon emissions
- Explore the principles and key methods of sustainable IT practices
- Build a robust, sustainable IT strategy based on proven methods
- Optimize and rationalize your code to consume fewer resources
- Understand your energy consumption patterns
- Apply a circular approach to the IT hardware life cycle
- Establish your sustainable IT baseline
- Inspire and engage employees, customers, and stakeholders



The Economics of Data, Analytics, and Digital Transformation

Bill Schmarzo

ISBN: 9781800561410

- Train your organization to transition from being data-driven to being value-driven
- Navigate and master the big data business model maturity index
- Learn a methodology for determining the economic value of your data and analytics
- Understand how AI and machine learning can create analytics assets that appreciate in value the more that they are used
- Become aware of digital transformation misconceptions and pitfalls
- Create empowered and dynamic teams that fuel your organization's digital transformation

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Technical Program Manager's Handbook*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804613559`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly