Product Management Notes

Kurt Rhee

A free guide, or reference, for software developers or domain experts new to the field of product management.

October 16, 2024

To my wife Hannah,

Thank you for listening to me grapple with these concepts over the years, and everything else that you do.

Contents

1. Terms 5
2. Introduction 5
   2.1. How to Read this Book 7
3. Feature Design and Definition 7
   3.1. Descriptive vs. Prescriptive 9
   3.2. Falling Back to Manual 9
   3.3. Blooms Taxonomy 10
   3.4. Domain Driven Design 11
   3.5. Overfitting 12
   3.6. Happy and Sad Paths 12
4. Experimentation 12
   4.1. Experiments for Different Stages of the Product Life Cycle 13
      4.1.1. Finding Problem-Solution Fit 13
      4.1.2. Finding Product-Market Fit 13
      4.1.3. At Scale 14
5. Feature Level Prioritization 14
   5.1. On Frameworks 16
   5.2. Drag 17
   5.3. Features vs. Infrastructure 17
   5.4. A taxonomy of technical debt 18
6. Ceremonies 19
   6.1. On Backlog Grooming 20
      6.1.1. Grooming Meeting Tips 21
      6.1.2. On Estimates 21
      6.1.3. Story Points 22
   6.2. On Sprint Planning 23
      6.2.1. Sprint Planning Meeting Tips 23
   6.3. On Sprint Review 23
      6.3.1. Sprint Review Meeting Tips 23
   6.4. On Retrospectives 24
      6.4.1. Retrospective Frameworks 24
7. Product Strategy 24
   7.1. Product Strategy Frameworks 26
      7.1.1. Having Any Strategy 26
      7.1.2. Picking Up Pennies in Front of A Steam Roller 26
      7.1.3. Commoditize Your Complement 27
      7.1.4. Blue Ocean 27
      7.1.5. Establishing a Beachhead 27
      7.1.6. Pyrrhic Victory 28
      7.1.7. Enshittification 28
      7.1.8. Service Software Sprial 28
   7.2. On Product Value to Investors 29
8. Conclusion 29

8.1. Afterwards 30
Bibliography 31
Index of Figures 33

1. Terms

- **API**: Applications Programming Interface
- **CI/CD**: Continuous Integration / Continuous Deployment
- **DX**: Developer Experience
- **Feature Factory**: A team culture where features are prioritized without any consideration for technical debt or underlying software architecture.
- **PRD**: Product Requirement Document
- **SaaS**: Software as a Service
- **TAM**: Total Addressable Market
- **UI**: User Interface
- **UX**: User Experience

2. Introduction

It is true, there are already many books out there written for new product managers. Many of them talk about the transition from user experience (UX) designer to product manager, but few discuss the unique challenges of transitioning from a software engineering or subject matter expert role. Many of these books also focus on company level strategy, and less on the day to day feature level development and prioritization that will comprise most of your workday as a new product manager. I know that when I first transitioned, I felt lost, unable to find solid footing in an environment that had suddenly become murky and liquid all around me. This book will not solve that problem, not entirely. Unlike software engineering, or scientific domains, where you can stand on top of computer science or physics, there is no definitive science of product management. There are no basic tenets or building blocks which you can assume to be true, only loose frameworks and recommendations from people who have been in similar positions in the past.

If you are reading this, it is likely that you have been a talented individual contributor, with technical expertise in software engineering or a specific domain, or perhaps both. Like a violinist in an orchestra, your mastery of your craft made you an indispensable member of the team. But you've been asked to step into a new role - that of a conductor, and like a violinist who has spent most of their career in strings and hasn't mastered the art of percussion, woodwinds, brass, conducting, etc. it is likely that you haven't mastered the art of sales, user experience design, user research, business strategy, customer success, cyber-security etc. Perhaps you are the once in a generation musical polymath that can master every instrument in the ensemble and pull them together into a grand and genius magnum opus, but more likely you will have to triage your time and expertise like the rest of us. You will have to humbly make decisions that affect the work of the people around you. Decisions that will have ramifications that you cannot possibly fully understand on your own. Out of respect and empathy for them, you must become conversationally fluent in their disciplines as quickly as possible and also trust in their expertise.

This transition can feel daunting. The competencies that are rewarded in technical fields such as logical thinking, complex mathematical reasoning, and domain expertise are still rewarded in product management, but they are balanced against a different set of skills such as empathy, marketing, finance, communication, prioritization, etc. that may not be complement to the skills that a newly minted product person has developed over their career thus far. Product management is a generalist role. Taking the product oriented career path necessitates that we spread our time over a wider area. Technical product managers must let go, in part, of some of the skills that they are most proud of, the same ones that got them to where they are in the first place. In exchange, they get a much broader view of how an enterprise operates, the chance to positively affect the working lives of their colleagues, and the opportunity to shape products that can have a significant impact on customers and the business as a whole.

In a vector analogy of a software business, software developers represent the magnitude of the vector and product managers represent the direction. This book aims to create a framework of thought about product strategy from the perspective of a software developer or subject matter expert, but it makes no prescriptions on how things should be done. There are no 3

steps to riches in this book, only a few footholds here and there throughout the vast ocean of uncertainty that is product management.

The book starts narrowly with the definition of a single feature, and continually expands outwards to the prioritization of multiple features, to product level strategy. A great product manager will be able to navigate upwards mentally from the tactical to the strategic and back again, depending on their audience and the greatest need of the hour.

2.1. How to Read this Book

You could think of this document as a very long winded set of book recommendations. In each section, there are an unordered series of notes on a wide variety of product management topics. Each note is a small kernel of a much wider idea, and references have been provided to those source materials. The book favors breadth over depth deliberately. Curious readers should use each kernel as a jumping off point towards the fully recommended reading material in the accompanying citations for more detail.

3. Feature Design and Definition

> In theory, product management is a masterfully played game of chess. In practice, product management feels like a hundred simultaneous games of checkers - Matt LeMay [1]

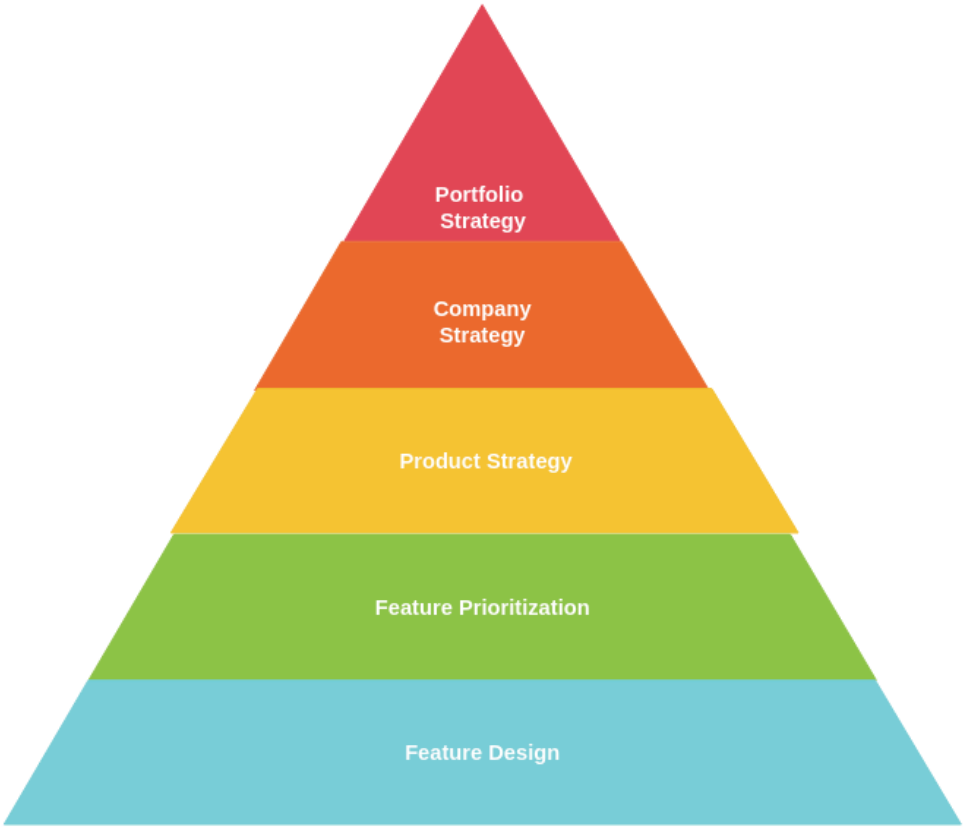

Figure 1: Product Management Hierarchy of Needs

Feature design is the bread and butter task of the product management job responsibility and you could say that it is the bottom level task on the product's hierarchy of needs. It is also the closest you will get to returning to the execution and implementation work of an individual contributor. The task generally consists of writing "cards" or "tickets" or "product requirement documents" (PRD) depending on the organization. Either way, feature design is the layer at which abstract concepts like strategy meet the practical realities of written code. At the end of the day, the company strategy could be perfect, but if the product and engineering team cannot properly execute, the product will still fail.

One of the things that distinguishes product management from software development is that there are so many ways to fail, and those failure modes aren't binary. They exist instead on a spectrum where doing too little or too much of something causes breakdowns in execution. In software development, you could estimate the difficulty of a task wrong, or choose the wrong algorithm, pick the wrong library or create bugs. In product management you could write an

overly verbose PRD and have developers miss the nuanced details or write one that is too short and have the development team floundering for information. You could add too much scope and not be first to market, or too little scope and squander your big marketing opportunity with a product launch that falls flat. You could spend too much time polishing the UX on a feature that nobody uses or spend too much time in user research rather than execution. You could be too prescriptive about the solution and piss off your software development team, or not prescriptive enough and not meet the needs of your users. There are no golden truths in product management because the answers to these trade-off questions all depending on your company's unique context. The right balance on each of these spectrums depends on the team, the history, the industry, the competition, etc.

The following sections are a non-exhaustive list of different things to consider when designing features and writing the corresponding requirements documentation that may help new product managers walk the many thin lines toward successful product execution.

3.1. Descriptive vs. Prescriptive

> "If you define the problem correctly, you almost have the solution." - Steve Jobs

The right design will be prescriptive of the solution in some cases and descriptive of the problem in others. How far you go in either direction depends on the company culture and also depends on the individual developers you are working with. Some developers don't want to learn about the context at all and just want a list of tasks to check off. Other developers want to fully understand the problem before helping you craft the solution.

In either case it is best to lean towards the descriptive end of the spectrum unless the task is very small, or you have very specific reason to believe someone would prefer for you to design prescriptively. A general contractor who habitually prescribes to their electrician how to rewire the main panel is going to have a hard time retaining good electricians.

This trade-off becomes simpler if your team has a strong engineering manager. In this case, you can be entirely descriptive of the problem and the feature requirements and let the engineering manager define the solution.

3.2. Falling Back to Manual

> "One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man." – Elbert Hubbard

Do not attempt to automate what cannot be completely automated. Imagine developing an application that automates your travel plans. The application finds the lowest prices within your travel dates, arranges pickup and drop-off from the airport, automatically checks you in, updates your calendar, and offsets your carbon footprint all with one click of a button. Only, the app doesn't know where on the plane you want to sit. Maybe the application decides in

all of its wisdom that it is most important to save money and gives you and your partner two middle seats at the back of the plane when a small upgrade fee would have let you sit together.

People would hate this application.

The practice of feature design, like all practices, benefits from a healthy dose of humility. Not all things can be fully automated no matter how brilliant the product and engineering teams. Designing an escape hatch or two for users to manually control what is necessary can make or break product adoption.

In more serious applications, like those that automate industrial processes, especially with a human in the loop, there is a risk that automation reduces the effectiveness of the human operator. Those situations where the human in the loop is required, also tend to be the most critical.

> As a result of increasing levels of automation, the human operator can become more of a passive monitor, which tends to make them less capable of handling situations where human intervention is suddenly necessary.
>
> — Ironies of Automation by Lisanne Bainbridge [2]

3.3. Blooms Taxonomy

> "Tell me and I forget. Teach me and I remember. Involve me and I learn." - Benjamin Franklin

Bloom's Taxonomy [3] is an educational framework that categorizes learning objectives into levels of complexity and specificity. It was originally created by Benjamin Bloom and his colleagues in 1956 and later revised in 2001. The taxonomy is often represented as a pyramid with six levels from bottom to top:

- **Remembering**: Recalling facts and basic concepts
- **Understanding**: Explaining ideas or concepts
- **Applying**: Using information in new situations
- **Analyzing**: Drawing connections among ideas
- **Evaluating**: Justifying a stand or decision
- **Creating**: Producing new or original work

These levels progress from lower-order thinking skills to higher-order thinking skills. Educators use this taxonomy to design learning objectives, activities, and assessments that target different cognitive levels, helping students develop a range of thinking skills. It can also be applied to product design.

For example, if a user can't remember how to get to your documentation, then it doesn't matter that all of the information that they need has been carefully crafted and written down for them. Likewise, it doesn't matter if the application is super powerful and can be applied in a wide variety of use cases if the user can't understand how to use it.

3.4. Domain Driven Design

> "If everyone spoke the same language, we'd have half as many arguments and twice as many conversations." - Jarod Kintz

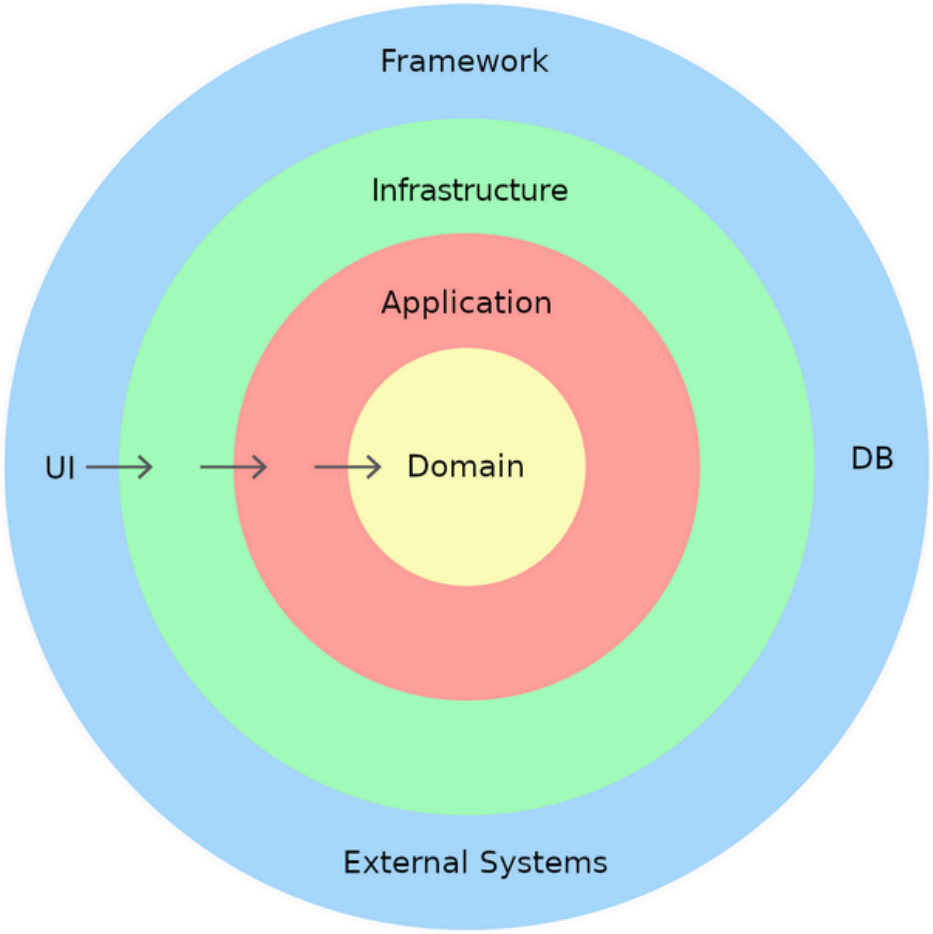

Figure 2: Domain Driven Design Diagram [4]

Domain driven design [5] is a design pattern in software engineering where the core business logic layer of the application is specifically architected to use the same language as the business domain. This allows software engineers, product managers, and subject matter experts to collaborate more closely in the one layer of the application stack where collaboration makes the most difference. In other layers of the application software engineers can be a bit more lax in their choice of terminology.

If you have a collaborative engineering manager and subject matter expert, encouraging the use of the domain driven design pattern can increase the velocity of the team since everyone starts off speaking the same language.

3.5. Overfitting

> “The road to hell is paved with good intentions” - Proverb

Overfitting [6] isn’t limited to statistics; it occurs in feature development as well. A common mistake is to add functionality based solely on customer feedback without considering the context. This can lead to a product that is overly complex and inflexible.

Consider this scenario: A project manager, who believes they are being customer centric, takes the customers asks literally. The customers are asking for additional functionality in the UI, specifically asking for additional fields. Initially, customers are satisfied that their immediate pain point is being fixed. However, as the product becomes more complex, those same customers begin to churn, preferring simpler, more usable tools.

There are many examples of tools becoming more and more complicated over time and losing market share to their simpler less feature-full competitors, for example Yahoo’s over-complicated search page vs. Google’s entirely minimalistic one, or MySpace’s complicated profile page customization features vs. Facebook’s standardized profile pages.

3.6. Happy and Sad Paths

> “I am prepared for the worst, but hope for the best” - Benjamin Disraeli

Good feature design considers not only what happens when everything goes right (the happy path), it also considers what should happen if anything goes wrong. Error messages, error screens, redirections, can all improve the overall UX of the application. Anyone who has read about entropy (or written software) knows that there are more disordered states than ordered ones. Considering only what happens for the user when things work correctly leaves out the majority of possibilities.

4. Experimentation

> "The first principle is that you must not fool yourself and you are the easiest person to fool." - Richard Feynman

The validation of product ideas may be one of the tasks that feels most foreign to software developers or domain experts. For software developers, it might seem strange to spend time just talking to users without having any "productive" artifact to show at the end of the conversation. For domain experts, it might seem counter-intuitive to sit back and learn from a user who is likely more novice than they are. Experimentation is inherently a humbling activity, an admission that we could do better with the help of others relative to going it alone.

There is an infinitude of options and branching paths when creating products. Some of these branching paths lead to product success, most of them lead to failure. One way that product roles add value to a team is to prune some of these paths before any development works gets done. In this way the product person serves as the team's forward scout in order to inform the team of potential opportunities and pitfalls in the validation before the more expensive execution phase even begins.

4.1. Experiments for Different Stages of the Product Life Cycle

VMware's product management playbook [7] discusses three different regions of the product life cycle where different experiments may apply.

4.1.1. Finding Problem-Solution Fit

- **Customer interviews:** Just sitting down to talk to potential customers may tell you what their problems are, whats standing in their way, how to segment them, which segments might be early adopters.
- **Prototypes**:
  - Can help with validation efforts, both in terms of market viability and in terms of technical feasibility. One technique, called a "walking skeleton test" involves building out a tiny sub-system to perform one end-to-end function to help validate technical feasibility.
  - Prototypes are a double-edged sword. Only in enlightened organizations will prototype code be thrown away as intended. The incentive to just iterate on something that doesn't have a strong foundation is too strong for many organizations to avoid. Prototypes should be built with this context in mind. If your organization is disciplined with prototypes, they can be a great tool to learn quickly about a given problem or technology. If your organization is not disciplined, prototypes should be built to eventually iterate up to a production grade tool. This approach lends you insights more slowly, but is less dangerous.

4.1.2. Finding Product-Market Fit

- **Wizard of Oz test**: There is a joke that A.I. stand for Actually Interns. Used poorly, these tests which replace a complex technical solution with manual human intervention (without the end user knowing) may be interpreted as just another marketing ploy to dupe investors.

For example, Amazon Go's "Just Walk Out" cashier-less grocery store concept was criticized for using an army of people in India to manually review cart data. On the other hand, Wizard of Oz tests can let you know ahead of time whether or not people are willing to buy your product without having to go through the expense of building one out completely.

- **Concierge test**: Some companies run a services business on the side. If customers are willing to pay for human services to get their tasks done, then they certainly will be willing to pay a lower price for your automated solution. In this way the services business acts as a validation tool and an income generator. A bona-fide Wizard of Oz test with no smoke and mirrors.
- **Smoke test**: A smoke test is a website that describes the product's value proposition and asks customers to sign up for the product before it is available. Kickstarter is a great example of a platform which shows the power of smoke testing in real life.

4.1.3. At Scale

- **A/B test**: Presents two versions of a product or feature to see which one performs best. Works best for large user bases, might not be useful in a business to business context with small numbers of users.

5. Feature Level Prioritization

> "The difference between successful people and really successful people is that really successful people say no to almost everything." – Warren Buffett

Product managers can create work items all day, but eventually there will be more work items in the backlog than can be completed in one sprint by the software development team. If your company had an infinite amount of money, they wouldn't need product managers. They would just hire an army of software engineers and UX designers to build every single one of those features. In the real world, companies don't have infinite money and each company and team operates under a set of constraints. Like a linear programming problem, the art of product management lies in first defining the feasibility space and then finding the best position solution within the range of possibilities.

In this regard, it may be more useful to be exposed to as many different considerations as possible when making prioritization decisions, and then determine how to balance each consideration against every other consideration in an $O(n^2)$ fashion. This could be represented as a graph with 5 nodes where each node is connected to every other node in the system. Below is a graph with 5 nodes, in practice you will likely have to consider many more prioritization factors than 5, and this is one of the reasons why product management is still mostly done by humans in a world where human advantage against computers grows smaller every day.

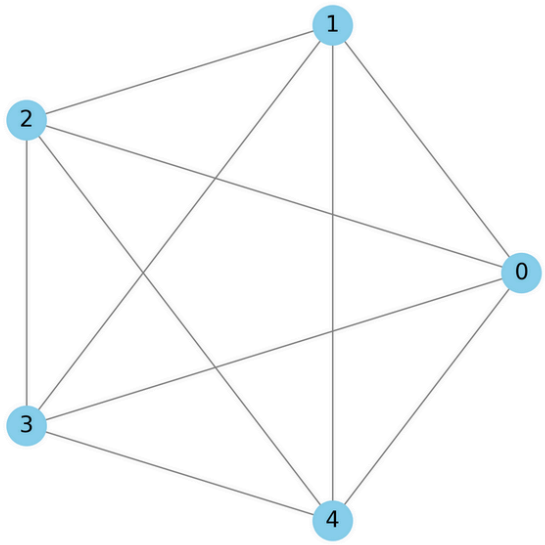

Figure 3: A complete graph with 5 nodes.

At the highest level, product managers should non-exhaustively think about whether the feature is desired by users, if it is feasible for the engineering team to accomplish and what impact the feature will have on the business. Many other lower level considerations are included in the following sections.

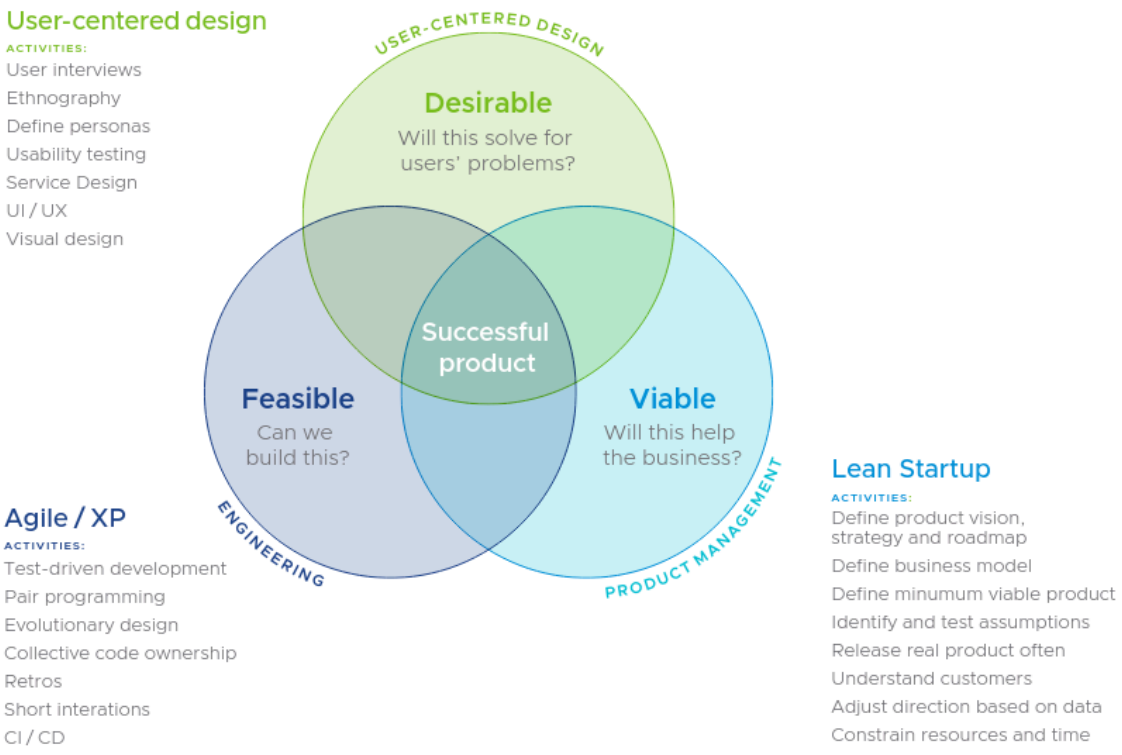

Figure 4: From VMware Tanzu Labs Product Management Playbook [7]

5.1. On Frameworks

> "Rules are for the obedience of fools and the guidance of wise men." - Douglas Bader

There are numerous product management frameworks that exist to help product managers prioritize work. For example the R.I.C.E. framework encourages product managers to prioritize based off the following criteria:

- **Reach**: Features that will reach more customers are more important.
- **Impact**: Features that will have a more positive impact are more important.
- **Certainty**: Features that have been requested by the most number of customers are important.
- **Effort**: The lowest effort features are more important.

The framework does not take into account how each of these four factors should be optimized against one another and is therefore only useful in a concrete sense when a given feature only differs from another feature by one factor. It also doesn't take into account many of the additional considerations one should think about when making prioritization decisions.

Newly minted product owners from a technical background may be tempted to created a numeric prioritization score for each of their features based off of the R.I.C.E framework, but they will soon find that numbers are inadequate and give a false sense of security that

prioritization is being done correctly. Product management is more biology than physics, and crude models break down at this level of complexity.

A framework can be useful as a jumping off point, but good product management requires decision making to be a part of a wider philosophy which actually takes into account the various nuances of the team, the company, and the market as a whole.

Instead of applying frameworks blindly to prioritization problems, it may be more useful to learn the underlying ideas that each framework suggests are worth considering.

5.2. Drag

> For every action there is an equal and opposite reaction - Isaac Newton

Some features have a higher maintenance burden than others which we will call "drag". For example, interconnecting with an external API will require maintenance if that API changes. Using an immature library may cause breaking changes. Any time these events occur, it will add unplanned drag to your product team. This drag has a high likelihood of disrupting your roadmap since the company which maintains the external API has their own roadmap to deliver on.

Any additional lines of code will come with some amount drag. Some features have asymmetric amounts of drag which will disproportionately affect smaller teams and will slow down new feature development, much like technical debt. Product managers should think carefully about how much drag exists in their product and determine if additional drag is worth the projected benefit of the proposed new feature.

5.3. Features vs. Infrastructure

> "Technical debt is like entropy: it's always increasing in your project, and you have to work hard to keep it under control." - Martin Fowler

If you come from the domain, say physical engineering or another technical field, you may be familiar with the term project manager. You may have worked on projects yourself. But products, especially software as a service (SaaS) products, are entirely different from projects because projects end. Products on the other hand have lifetimes that go on for decades. Every feature you add comes with a small amount of imperceptible burden. In a project, this burden doesn't matter since the project will end soon and the next project begins with a fresh slate. In product, the burden can become deadly.

In teams without a strong engineering manager to balance out the product manager, the culture of the team can devolve into a feature factory. The balancing act between working on features and working on infrastructure is task dependant, but the following few points should help to give an indication of whether your product team is more feature oriented or infrastructure oriented.

Starting from +8, subtract one from your "product speedometer score" if you do any of the following things. If you don't do them, add one instead.

- One step builds
- A continuous integration / continuous deliver (CI/CD) pipeline
- Zero Defect Culture (generally, you fix bugs before implementing new features)
- User Acceptance Testing
- Unit Testing
- Integration Testing
- Regression Testing
- Application frameworks are nearly up to date

Likewise, subtract one from your score if you are doing any of the following, add one if you are not:

- Rewriting a portion of your application from scratch
- Solving anticipated architecture problems instead of already existing ones

Adding up your score can give you a sense of your product/engineering culture. If you are at 0, you are likely balanced. If you are at +16 you are probably disregarding engineering best practices and if you are at −2 you might be over-engineering your software [8].

One way to pay down debt over time is to prioritize tackling debt along with features that add user value.

> For examples, if a story calls for adding a field to an existing form, you should consider also cleaning up the logic that delivers form validation errors.
>
> — VMware Tanzu Labs Product Manager Playbook [7]

5.4. A taxonomy of technical debt

> "If you think good architecture is expensive, try bad architecture." - Brian Foote and Joseph Yoder

One way to think about technical debt is to rank the debt item across three possible axes [9].

- **Cost**: How much time will it take to implement a fix and equally important, what is the risk of actually deploying that fix.
- **Impact**: How much does carrying this debt affect users, designers, developers, etc.
- **Contagion**: If the debt item is allowed to continue its' existence, how much will it spread?

Most people consider the cost to fix an item. Fewer will properly weigh the impact of the debt item. For example, in a feature factory team, the impact axis might be entirely ignored. Even fewer will think about contagion, which might be the most important axis of them all.

Ranking along the three axes allows us to create a taxonomy of technical debt items which can help product and engineering managers come to an understanding about which ones they want to solve. The following are a few categories of technical debt:

- **Local Debt**: Local debt is low on contagion, it is contained behind the wall of some interface, say a function block or an API. Developers may look into the black box and be horrified, but because it is contained, it can likely be left alone.
- **MacGyver Debt**: Two competing systems have been duct taped together. Generally the system which is easier for developers to use will win out. If you can tip the developer experience (DX) in favor of one system over the other, MacGyver debt will pay itself off over time.
- **Foundational Debt**: An assumption that lies at the heart of your system. These are the high cost, high impact and high contagion technical debt items that require a heroic effort from the team to fix. It can also be converted into MacGyver debt if the team is willing to maintain two systems at the same time.
- **Data Debt**: One of the three other types of debt has been allowed to live long enough to create a bunch of artifacts that sit on top of the tech debt item as their foundation. The cost then becomes very high since the risk of deploying the fix now includes a bunch of data items. Data debt can also be turned into MacGyver debt by creating two different data types with a conversion mechanism between the two.

6. Ceremonies

> "Individuals and interactions over processes and tools" - Agile Manifesto

In the years since the Agile Manifesto's creation, we've witnessed a proliferation of agile processes, frameworks, and tools. This irony is particularly evident in the widespread adoption of agile ceremonies – structured meetings that have become nearly ubiquitous in organizations claiming to embrace agile methodologies. Many have pointed out that the existence of these gatherings ironically contradict the spirit of valuing individuals and interactions above all else. The reality is that Agile ceremonies – such as daily stand-ups, sprint planning, sprint reviews, and retrospectives – have become the defacto standard for many organizations attempting to implement agile practices.

Product management can sometimes be criticized for proliferating unnecessary meetings, leading to decreased productivity and in some cases this criticism is well deserved. It is important to remember that each meeting has a cost, both in terms of the burdened salaries of the people involved in the meeting, but also the switching cost that each meeting entails. For those unfamiliar with Paul Graham's famous blog post on the topic of switching costs in software development, arguing for meeting blocks instead of dispersed meetings [10] may be informative.

> [In balanced teams] communication among team members tends to be informal, favoring spontaneous conversation over lengthy meetings
>
> — VMware Tanzu Labs Product Manager Playbook [7]

6.1. On Backlog Grooming

> Welcome to [backlog grooming] where everything is made up and the points don't matter. - Adapted from the T.V. show Whose Line Is It Anyways

Backlog grooming is the process of reviewing items in the product backlog with your development team. It involves:

- Clarifying user requirements
- Breaking down large items into smaller, manageable tasks
- Estimating effort for each item
- Removing outdated or unnecessary items
- Giving an effort score for each task

This activity helps keep the backlog organized and ensures that the team has a steady stream of work for upcoming sprints. It's typically done on some repeating basis.

6.1.1. Grooming Meeting Tips

A few tips for making backlog grooming a quick and painless experience for everyone:

- Have a set of items ready in a "priority backlog" queue before the meeting begins. Searching through a large backlog in the meeting while everyone waits for you is a waste of everyone's time. Better yet, have the priority backlog ready some time in advance of the meeting and allow the developers to see what you have on deck for them.
- Try to think of all of the repercussions of the feature addition before backlog grooming and add those detail to the work item before presenting it to the group.
- Only keep a groomed backlog that is 2 iterations (sprints) worth of work long. If your groomed backlog gets too large, people will forget all about what the tasks entail [7]. If your backlog is overfull, feel free to cancel the grooming meeting.
- Keep the un-groomed backlog short. You know that your backlog is too long if people consistently cannot find tasks and end up creating duplicate ones.
- Keep the both groomed and un-groomed backlogs searchable. Make sure you utilize tags, epics, and descriptive names so that people can find their items easily.
- Don't focus too much on the effort score as a specific number, just get a general idea of the level of effort. Estimates are not and can never be perfect.

6.1.2. On Estimates

> "The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time." - Tom Cargill

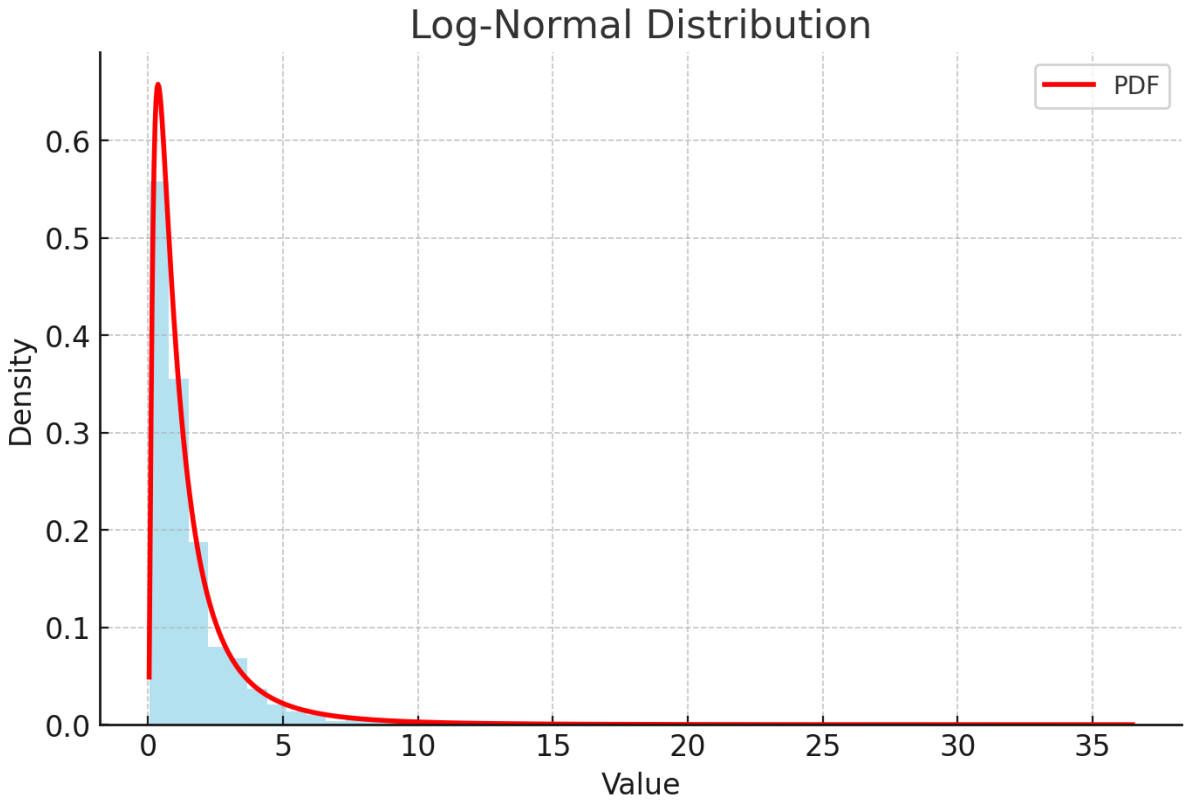

Figure 5: A log-normal distribution

A naive model for estimating how long a software development task will take is to model the estimate with a gaussian distribution. In a gaussian distribution, sometimes the actual task will take longer than the guess, and sometimes the actual task will get completed shorter than the guess. The likelihood of over-estimation is equal to the likelihood of under-estimation.

Unfortunately, software development is not best modeled with a gaussian distribution. A more fitting distribution would be a log-normal distribution [11]. In a log-normal distribution, the lower bound is bounded and the upper bound is un-bounded, much like a software development task. In software development, any task takes at least some time to complete and therefore, under-estimations are bounded on the low end of the distribution. In addition, many unknown-unknowns exist and therefore estimations are inherently unbounded on the high end.

If you approach estimates with the gaussian model in your head, you will be disappointed when some items inevitably slip past the sprint deadline. Plan instead for a log-normal distribution of task completion time and save yourself the heart-ache.

6.1.3. Story Points

Some teams will place “story points” in a Fibonacci sequence to work items, with lower points meaning the task is low difficulty and higher points meaning the task is of high difficulty. The number of story points does not matter. The purpose of the story points is to give you the product owner an idea if the task is easy, medium or hard. This will come in handy when doing sprint planning. If a task gets tagged with a very high story point score you may want to break it up into smaller tasks.

6.2. On Sprint Planning

> Continuous prioritization enables agility Agile/XP teams practice continuous prioritization. That is, although we do look a few weeks ahead to plan our work at a high level, we don't commit to a fixed sprint scope. That's because we might at any time learn about unexpected changes or new information that causes us to reevaluate our priorities. Continuous prioritization enables us to maximize our responsiveness to changing conditions and navigate complexity and uncertainty better than if we were to stick to a detailed, upfront plan. Developers will always pick up and work on the story at the very top of the backlog, so the order of the stories is very important. A story's position in the backlog communicates its priority—the most important stories are at the top of the backlog, whereas less important stories are near the bottom. It's your job as product manager to ensure the order of the backlog represents the latest priority.
>
> — VMware Tanzu Labs Product Manager Playbook [7]

6.2.1. Sprint Planning Meeting Tips

- Pre-plan the sprint. Don't make everyone wait for you while you try to decide what to prioritize.
- Let people leave. If you have planned someone to capacity, let them leave the meeting and regain some of their productive time.
- Know in advance what items are necessary to complete and which ones can have their schedules slide if need be. Some things have deadlines and other things don't. It is best to avoid creating a culture of immediacy because then everything becomes the same level of urgency.
- Differentiating high priority tasks from low priority ones also lets you be more flexible with the sprint planning meeting and people will stop arguing over insignificant details about how long a task will take since they know it is okay for some items to slip into the next sprint if need be.
- You can either control the timeline or the scope, but not both at the same time. If product management gives a due date for development, it is engineering's job to determine what can fit within the given timeline. If product management defines the scope of the project, then it is engineering's job to give a timeline for when that amount of work can get completed [12]. Not adhering to this rule is a surefire way to anger your execution team, or cause burnout.

6.3. On Sprint Review

Sprint review is your chance to keep your sales team and leadership team in the loop as to what is happening in the week to week happenings of the development team. There should be as few surprises in this meeting as possible. A team that is communicating fluidly and informally will likely cover most of the items that would otherwise need to be discussed in this meeting naturally.

6.3.1. Sprint Review Meeting Tips

- Reduce surprises.
- Use the meeting to create release notes or inform stakeholders (internal and external) that the item they requested has been completed.

6.4. On Retrospectives

> "Responding to change over following a plan" - Agile Manifesto

Among the various agile ceremonies, the retrospective stands out as the most crucial. These sessions are dedicated to reflecting on what worked well, what didn't, and how to improve. For the profit and loss (P&L) motivated, well done retrospectives have the possibility of reducing attrition, reducing unnecessary meeting time and allowing developers to do better work.

For those that might be driven by factors other than the bottom line, retrospectives are about culture. Not only do you get to talk about what might have gone wrong during the sprint, you get to talk about the cultural practices that allowed those things to occur. Documents like NASA's Elements of Engineering Excellence [13] showcase meta-cognition about the underlying factors that drive the failures that then show up at the surface level. What NASA understands is that culture is a form of automation. The engineering culture is the default mode in which people will approach and solve a problem. A more effective culture (one that writes tests and follows architectural patterns, etc.) will outperform a free for all culture where best practices are applied only on a case by case, person by person basis.

6.4.1. Retrospective Frameworks

The following are a set of three different frameworks for holding a retrospective meeting. Like any framework, they are merely guidelines. The true power of retrospectives lies not just in the ceremony, but cultivating a retrospective mindset in the team.

- **Start Stop Continue**
  - Start: Things that the team should start doing this upcoming sprint
  - Stop: Things that the team should stop doing this upcoming sprint
  - Continue: Things that the team started doing last sprint and would like to continue to do.
- **4L's**
  - Liked: What did team members enjoy or appreciate?
  - Learned: What new knowledge or skills did they gain?
  - Lacked: What was missing or could have been improved?
  - Longed For: What did they wish they had or hope for in the future?
- **Speed Boat**
  - The Speed Boat technique uses a visual metaphor of a boat to represent the team's progress and might be particularly useful in a team with feature factory culture.
  - The boat represents the team or project
  - Anchors represent things slowing the team down
  - Wind/propellers represent things helping the team move faster
  - Islands or a shore represent the team's goals

7. Product Strategy

> "All successful businesses are successful for the same reasons, all failed businesses have failed for different ones" – Leo Tolstoy (adapted)

There are many different force vectors that act on the success or failure of a product. Intuitively, it helps to have all force vectors pointing in the same direction. All else held equal, the product manager that has the best mental model [13] of these forces and how to influence them wins. A product manager that has an understanding of the market, categorical, company, product and cultural forces and how to act given those constraints has an advantage over one that does not have that same understanding. In the same way, product managers with experience in domain or software development roles have an advantages in understanding and influencing those particular forces, but may be disadvantaged elsewhere.

- **Market forces:** Government regulations, macro-cultural shifts, economic conditions. No product managers have control over market forces.
- **Category forces** [14]**:** The single largest predictor of product success. A stellar product in a dying category (DVD's, print newspapers, coal power, etc.) receives a much smaller multiple from investment companies than lesser known products in upcoming categories. At the C-suite, VP, and portfolio level, product managers can decide to get out of or enter different categories.
- **Company forces:** A stellar product in an expanding category during an economic boom can still fail if forces at the company level oppose the progress. A disfunctional leadership team, investment in other arms of the business (a failure to focus), an overwhelming debt burden, etc. can all serve to tank a otherwise good product. Some product managers will have influence but not control over forces at the company level.
- **Product forces:** These are the forces that new product managers have the most control over. Focus, user-centricity, problem-solution fit will all serve to buoy a potential product, but this force (like the others) exists within the context of all of the other forces.
- **Team Cultural forces:** A toxic team culture will serve to hinder the product in a host of different ways. Everyone has a say in team cultural forces. Of course, some people will have more say than others.
- **Execution forces:** How well does the team execute on the tasks that is given? How large is the execution team? Product managers may have some influence here, but most of this responsibility lies on the engineering team.

Ultimately, strategy is the art of deploying context to your advantage. The more context you have, the better the strategy that you can create. In 1597, a Korean admiral named Yi Sun-Shin fought a naval battle against a vastly numerically superior Japanese navy, he used his knowledge of the Korean coastline, current patterns and ship design to win against unthinkable odds [15]. Today, tiny startups best huge corporations with fractions of their budget and headcount all thanks to good strategy.

7.1. Product Strategy Frameworks

Competitive strategies are macro level strategies for thinking about the positioning of a product at the level of the business competition. If execution is what you do, strategy is what you deliberately do not. There are more than a few failure modes when it comes to product strategy. Not having a strategy leads the company to peanut butter their resources across a bunch of different initiatives, each one slightly more predisposed to fail due to lack of support. Having the wrong strategy allocates limited resources to something that will fail either way. Under-communicating the strategy leads to buy-in problems during execution. The underlying theme in most books about product strategy though is just to focus on something, anything. It seems that the main failure mode in industry is the peanut butter approach, and merely avoiding the pull to do everything can put you far ahead of the competition.

The following few sections are devoted to a non-exhaustive set of possible product strategies utilized by companies today and in the past. Discussions about portfolio level positioning are deliberately excluded.

7.1.1. Having Any Strategy

- **What is it**:
  - Reading the market tea leaves, crafting a strategy, getting buy-in from important stakeholders all take time. In the interim, it is important to have a working product strategy which you can use to get everyone moving in the same direction [16], even if that direction is just to clean up technical debt or fix bugs until a true strategy can emerge.
- **When to consider it:**
  - Any time there is a product strategy vacuum.
- **When not to consider it:**
  - When there is already a robust strategy in place.

7.1.2. Picking Up Pennies in Front of A Steam Roller

- **What is it:**
  - This is the default mode of operation for most companies without strong product leadership. It is also the default mode of operation for sales lead organizations. It involves going after a host of small easy win opportunities instead of one large ultimate goal [14]. For example, a company which makes small adjustments to their software to satisfy the wants of each marginal customer they come across instead of focusing on some over-arching mission is picking up pennies in front of the steam roller that is their focused competition.
  - Leading up to WWII, the French constructed a series of fortifications extending across their eastern border called the Maginot Line to deter Nazi aggression. The Nazi's instead focused their offensive into a small, less fortified area through the Ardennes forest and were able to break the Maginot Line without having to spread their attack over a large area [17].
- **When to consider it:**
  - If your product is end of life and you are attempting to squeeze as much value out of it as possible, downsizing your team and then picking up pennies in front of a steam roller is a viable strategy, one that is even favored by some private equity firms.
- **When not to consider it:**

- When the goal is to grow, not shrink.

7.1.3. Commoditize Your Complement

- **What is it:**
  - A business with a strong core product can take a loss or even make complementary products free in order to drive more revenue in the core product. For example, Google makes android free which drives more search on mobile to Google [18]. This strategy fits into a wider philosophy which appears in numerous product strategy recommendations [16], which is to compete where you are strong and retreat where you are weak.
- **When to consider it:**
  - Your company has a strong core product which is competing in a landscape where a competitor may have a stronger complementary product. The user needs both products in order to do their job. By making the complementary part of your product free, you take away market share from the competitor where it hurts them most and drive revenue to your core product instead.
  - Revenue is higher in your core product than it is in your complementary product.
- **When not to consider it:**
  - You do not have a strong core product to drive revenue to.

7.1.4. Blue Ocean

- **What is it:**
  - A red ocean is a business landscape where competition is fierce and direct. It is meant to illicit images of a shark infested waters where each shark competes with other sharks over the same set of fish. A blue ocean strategy [19] is one in which new waters are explored. For example, Cirque-de-Soleil moved into bluer waters when they decided to blend theatre and circus together, instead of competing in the traditional Barnum and Bailey circus act market.
  - Ultimately, a blue ocean strategy is about understanding your opponent's strategy as context for your own strategy. In a head to head fight, the numerically superior competitor wins, but it is not always necessary to participate in a head to head fight.
- **When to consider it:**
  - If your product is not well differentiated from the competition, it may be time to think about looking for bluer waters.
- **When not to consider it:**
  - Your product is backed by overwhelming financial resources and you can drown out the competition in the traditional red ocean by keeping prices low.

7.1.5. Establishing a Beachhead

- **What is it:**
  - Instead of trying to create the most compelling product offering right off the bat, you should find one customer segment of likely early adopters and focus on them. Once you have established a beachhead [20] with this group, you can then expand to take on more laggardly technology adopter groups.
- **When to consider it:**

- You are an underdog in a new space, whether than means that you are a startup or an an established company which is entering a new arena.
- **When not to consider it:**
  - You are the incumbent.

7.1.6. Pyrrhic Victory

- **What is it:**
  - Some companies will take a loss on a product until the competition folds, then they enjoy a monopoly once all of their competition has been starved out. China (and other nations of course) does this very effectively at a state level by subsidizing certain industries, like photovoltaic module manufacturing, or electric vehicle manufacturing. Many unsubsidized competitors in other nations cannot come close to competing and they fold.
  - The Soviet Union often employed pyrrhic tactics when retreating on the eastern front of WWII, burning and salting fields, destroying equipment, etc. so that they could not be employed by the advancing German forces. Eventually the Germans, far from their supply lines and fighting on multiple fronts, were defeated and driven out [21], [22].
- **When to consider it:**
  - You have a larger bank account or more resolve than your competition.
- **When not to consider it:**
  - You have a smaller bank account than your competition.

7.1.7. Enshittification

- **What is it:**
  - Gain a network effect by subsidizing users on your platform with investor money and being in general very user focused instead of profit focused. At some point the investors will need to be paid back. The platform can then be enshittified by becoming highly profit centric at the expense of users, who will stay because of the built up value in the network effects of your platform [23]. Many social media companies are examples of enshittification.
- **When to consider it:**
  - Network effects play a big role.
- **When not to consider it:**
  - Network effects don't matter.

7.1.8. Service Software Sprial

- **What is it:**
  - Your company maintains both a services arm and a software arm. The services arm creates income that can be fed into the software arm and the software arm can create tools which make the services arm more efficient and competitive. The services arm also serves as a pool of internal users of your product and can fill gaps that the product cannot serve.
- **When to consider it:**
  - Early stage companies generally benefit more from this arrangement.
- **When not to consider it:**
  - The services business can end up being a distraction for the leadership team which should focus on the more scalable product business.

7.2. On Product Value to Investors

In another set of forces that serve to balance one another, we can see how investors might place value on a product . Product managers may think about maximizing or minimizing qualities of their offering in order to attract investors.

- **Urgency**: How badly do people want or need this thing right now.
- **Total Addressable Market (TAM)**: If the product had 100% market share, how much revenue would be captured?
- **Cost of Customer Acquisition**: How much does it cost the business to generate one additional customer?
- **Cost of Value Delivery**: A manufacturing business which needs to build a factory before delivering value to customers may be more risky than a software business which requires no factory.
- **Uniqueness of Offer**: How easily can this be replicated? Does something like it already exist?
- **Up-sell Potential**: Can the offer lead to significantly higher offers down the road?
- **Evergreen Potential**: Business consulting requires ongoing work to get paid. A book can get produced once and then sold over and over as is.

8. Conclusion

Product is humbling. There is so much more each of us could learn about sales, about software engineering, about marketing, about product strategy, about when to just get out of the way of your engineering team and let them do their work. It is a hard job to do well, but a rewarding one. You get to influence the operations of the company at a user, culture, feature, and product level. There are so many ways to mess up, and more than a few ways to get it right.

In the end, product management is all about embracing fluidity. You are not a cog in the machine. You are the grease that makes the different parts of the whole work together. You are going to be asked about the exact wording of an error message in one conversation and in the next you are going to be asked about your opinions on strategy given possible valuation scenarios at your companies next series X raise. You are going to get your hands dirty. You will do tasks that are way below and way above your pay grade. Embrace all of it, because like they say, where there is muck there is brass [24].

8.1. Afterwards

If some of the notes in this notebook were interesting to you and you want to learn more, please take a look at the original publications in the bibliography. There is no way that this text can fully encompass so many great ideas in just the few pages that I have used here. If there is something that you think I am missing and would be valuable to add to this manuscript please feel free to send me an email at [MyName]@gmail.com.

If you feel that I am totally off base and none of this was useful for you, I am sure that you are right. Product management has been a different experience at every company that I have ever worked for, and I have only worked in one industry.

If, on the other hand, this booklet has been useful to you, feel free to share it, adapt it, cite it, plagiarize it, forget entirely about it, or feed it to a large language model. In other words, do with this whatever is expedient and useful to you. I am humbled and glad that something I did positively affected your day.

Bibliography

[1] M. LeMay and M. LeMay, *Product Management in Practice: A Practical, Tactical Guide for Your First Day and Every Day After*. O'Reilly Media, Incorporated, 2022. [Online]. Available: https://books.google.com/books?id=78rHzgEACAAJ°

[2] L. Bainbridge, "Ironies of automation," *Automatica*, vol. 19, no. 6, pp. 775–779, 1983, doi: https://doi.org/10.1016/0005-1098(83)90046-8°.

[3] B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwohl, *Taxonomy of educational objectives: The classification of educational goals. Handbook I: Cognitive domain*. New York: David McKay Company, 1956.

[4] HiBit, "Domain Driven Design: Layers." [Online]. Available: https://www.hibit.dev/posts/15/domain-driven-design-layers°

[5] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley Professional, 2003.

[6] K. Yien, "Overfitting." [Online]. Available: https://kevinyien.com/blog/overfitting.html°

[7] V. T. Labs, "VMware Tanzu Labs Product Manager Playbook."

[8] J. Spolsky, "The Joel Test: 12 Steps to Better Code." [Online]. Available: https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/°

[9] E. Lacher, B. Matthews, C. Leff, and S. Lane, "The Taxonomy of Tech Debt." [Online]. Available: https://technology.riotgames.com/news/taxonomy-tech-debt°

[10] P. Graham, "Maker's Schedule, Manager's Schedule." [Online]. Available: http://www.paulgraham.com/makersschedule.html°

[11] E. Bern, "Why software projects take longer than you think: a statistical model." Accessed: Oct. 03, 2024. [Online]. Available: https://erikbern.com/2019/04/15/why-software-projects-take-longer-than-you-think-a-statistical-model.html°

[12] P. M. Institute, *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*, 7th ed. Newtown Square, PA: Project Management Institute, 2021.

[13] T. T. Bales and M. F. Holguin, "Elements of Engineering Excellence," Washington, D.C., Jan. 2012.

[14] G. A. Moore, *Escape Velocity: Free Your Company's Future from the Pull of the Past*. New York, USA: HarperCollins, 2011.

[15] W. contributors, "Battle of Myeongnyang."

[16] B. Rubick, "Product Strategy." [Online]. Available: https://www.rubick.com/product-strategy/°

[17] Wikipedia contributors, "Maginot Line — Wikipedia, The Free Encyclopedia."

[18] J. Spolsky, "Strategy Letter V." Accessed: Oct. 16, 2024. [Online]. Available: https://www.joelonsoftware.com/2002/06/12/strategy-letter-v/°

[19] W. C. Kim and R. Mauborgne, *Blue Ocean Strategy: How to Create Uncontested Market Space and Make the Competition Irrelevant*, Expanded Edition. Boston, MA: Harvard Business Review Press, 2015.

[20] G. A. Moore, *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers*, 1st ed. New York: Harper Business, 1991.

[21] R. Overy, *Russia's War: A History of the Soviet Effort: 1941-1945*. New York, NY: Penguin Books, 1997.

[22] S. G. Fritz, *Ostkrieg: Hitler's War of Extermination in the East*. Lexington, KY: University Press of Kentucky, 2011. [Online]. Available: https://www.kentuckypress.com/9780813134160/ostkrieg/°

[23] C. Doctorow, "Potemkin AI: Behind the facade of artificial intelligence breakthroughs," *Pluralistic*, 2023, [Online]. Available: https://pluralistic.net/2023/01/21/potemkin-ai/#hey-guys°

[24] J. Spolsky, “Where There’s Muck, There’s Brass.” [Online]. Available: https://www.joelonsoftware.com/2007/12/06/where-theres-muck-theres-brass/°

Index of Figures

Figure 1: Product Management Hierarchy of Needs 8
Figure 2: Domain Driven Design Diagram [4] 11
Figure 3: A complete graph with 5 nodes. 15
Figure 4: From VMware Tanzu Labs Product Management Playbook [7] 16
Figure 5: A log-normal distribution 22