How To Double Your R&D Effectiveness

Introduction

Digitalization is disrupting industry after industry and, as the saying goes, software is eating the world. More and more companies define their competitiveness by increasing amounts of software and the value created in products, solutions and services is driven by bits rather than atoms. As a consequence, the ability of companies to engineer software efficiently and effectively becomes a key differentiator. There are several reasons why software engineering, relative to material science and electrical engineering, is increasing in relevance. The main factors, for most companies, are threefold. First, software can be deployed and updated throughout the economic life of a product, solution or service. So, rather than having a system that is fixed and immutable from the point system design is finalized, software allows for continuous improvement. Second, software allows for a level of customization and personalization that is entirely unfeasible for anything atoms-based. The software in the system allows for dynamic adjustment of user interfaces, algorithmic behaviours, presence or absence of features, etc. As most mobile phone users have realized, although the hardware and mechanics of your phone are the same as everyone else's, it is the software that makes your phone unique and personalized. Finally, software allows for a transition from opinion-based to data-driven decision making. The same mechanisms that allow for software to be continuously deployed on your system also allow us to measure customer behaviour and system performance, allowing for a fast feedback loop to develop between R&D and the software deployed in the field.

With companies adding software based functionality to their products, the amount of R&D resources allocated to software started to go up as well. Many companies follow Moore's law in terms of the amount of software being put in the system if only for the simple reason that systems engineers are driven by bill of materials cost. As the cost of electronics is halved every 12 to 18 months it is viewed as "free" to double the computational power and memory space. This does mean, though, that computing resources grow exponentially and no self respecting product manager, architect or engineer will let all those resources go to waste and instead look for ways to add value to the system. As a consequence, the size of the software in many systems follows the hardware capability and doubles every five to ten years.

The consequence of this is that the amount of resources allocated to software has been going up in most companies and as with any exponential growth, at some point the limits of reality start to hit. As most companies express R&D as a percentage of revenue, the challenge is that as more and more resources are requested for maintaining, evolving and creating software, at some point it become unfeasible to meet these requirements. This is when R&D management, often far from experts on software, start to squeeze the resource requirements.

The main problem is that due to a lack of understanding of software, general and R&D management treats all software as being created equal. Hence, rather than focusing the software R&D resources on those software assets that are most important and limit resource allocation to those areas that are less important, the approach tends to be to just limit R&D resources across the board. This then leads to underinvestment in R&D and a negative effect on competitiveness. Strategic use of the R&D resources allocated to the strategic software components and subsystems becomes increasingly important with the increasing importance of the role of software, but companies fail to do so as they lack the tools to reason effectively and systematically about the allocation of resources to software.

This book provides one set of tools to address the aforementioned challenge of effectively and systematically reasoning about software assets, resource allocation, refactoring, platforms and engaging the ecosystem surrounding your organization. As the foundation for this set of tools we have developed the three layer product model (3LPM) as a tool and framework to reason about strategic use of software in large scale software engineering. The 3LPM, either conceptually or physically, organizes the software functionality into one of three types, i.e. commodity functionality, differentiating functionality or innovative and experimental functionality. By categorizing the functionality in a system, the 3LPM supports several types of decision making that allow for a much more strategic and focused approach to software R&D. Although we'll introduce the 3LPM in more detail later, the figure below shows the basic elements and structure.

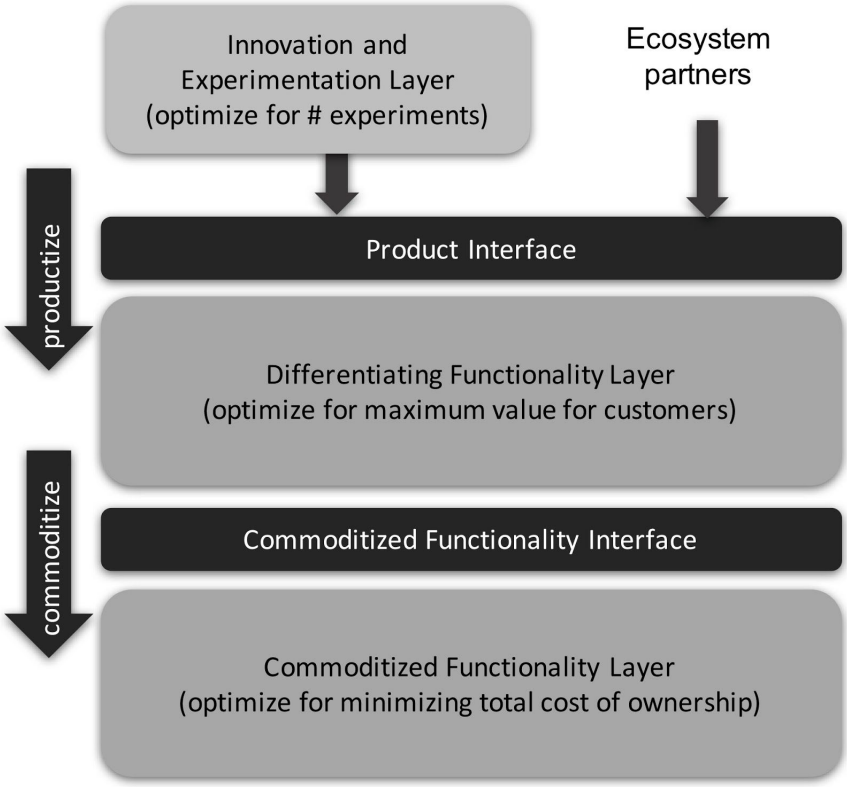

Figure X: The Three Layer Product Model

Although the 3LPM may easily look like a high-level architecture picture, it can be used for several use cases concerning resource allocation, architecture refactoring, software platforms and ecosystems. In this short book, we discuss the four primary use cases. These use cases also form the four main parts of the book:

1. **Strategic resource allocation**: In the first part, we focus on the key problem that we raised in the introduction: resource allocation into software R&D is often conducted without any understanding or awareness of the differences between different areas of functionality in the system. Allocating resources in an undifferentiated fashion easily results in the majority of resources being allocated to commodity functionality rather than innovation and differentiation. As shown in the figure above, our research shows that in typical companies, 80-90% of resources are allocated to commodity functionality. In the first part of the book, we start by categorizing the components in existing software as innovative, differentiating or commodity. Based on this, resource allocation can be controlled by allocating resources predominantly to innovation and differentiating functionality whereas only resources for bug fixing is allocated to commodity components. This facilitates limiting resources to commodity functionality and results in a much more strategic allocation of resources.
2. **Refactor software**: Although part I is concerned with strategic resource allocation, it does not affect the structure of existing software assets. The second part of the book is concerned with refactoring the software to align with the architectural structure proposed by the 3LPM. This provides much simpler management of software assets and allows for organizational alignment with the main software layers. In part II, we focus on assessing the current architecture, designing the desired architecture and planning and implementing the transformation.
3. **Towards platforms**: As the structure of the 3LPM suggests, there is a very natural transition from a single 3LPM for a system or product to a platform where the 3LPM is used to model where functionality is allocated and how it transitions. When a software asset (or multiple) are split into a platform and products on top of the platform, the interface and the process for moving functionality between products and platform need to be discussed. Of course, at this point it will also have become obvious that one can experience a cascading series of 3LPM where the commodity layer of one model, for instance a product, is aligned with the differentiation layer of the platform model that it is built on top of. This is the topic of part III.
4. **Engaging the ecosystem**: In the final part of the book, we discuss the use of the 3LPM to engage with the ecosystems around the company. As we'll discuss in part IV, the position we take is that every organization is involved in at least three ecosystems, organized around the 3LPM layers, i.e. an innovation, differentiation and commodity ecosystem. As companies are no islands, the ability to engage ecosystem partners, either in a directed or undirected fashion, for those parts where the company itself does not have a unique differentiation is critical. However, once again, companies often lack a systematic and effective model for deciding when and where to engage their ecosystems. In this context, we'll discuss the Three Layer Ecosystem Strategy Model (TeLESM) to provide strategic guidance on ecosystem engagement.

In the remainder of this introduction, we first introduce the three layer product model in more detail. Subsequently, we introduce WKI, the running case example that we'll use throughout the book.

The Three Layer Product Model Framework

In most organizations, the connection between the "business side" and the "engineering side" of the company is not as strong as it should be. This easily leads to a situation where both sides make decisions about product functionality, allocation of resources, prioritization of customer requests, etc. without a full understanding of the implications. This frequently causes major inefficiencies in the organization. For instance, R&D may engage on major refactoring efforts in areas of the architecture that are irrelevant from a business perspective. Or, the sales team may promise functionality to customers that results in severely negative implications for the product architecture. In the most unfortunate cases, this leads to a situation where the business side views R&D as incompetent and slow and R&D views the business side as naive and ignorant of the technical realities of their systems.

The main reason for discrepancies and inefficiencies between business and engineering is because of a lack of language and tools to facilitate effective communication, prioritization and agreement on key priorities. In response to this, as well as several other challenges, we have developed the Three Layer Product Model (3LPM) as a tool to facilitate common understanding across the company.

The 3LPM is graphically presented in figure X. The 3LPM starts with classifying all functionality into three main categories. The first category is the commodity functionality. This constitutes all functionality that customers typically expect in a product or system, but that is also offered by competing companies. Because of this, there is no differentiation or preference to be expected from customers. As there is no benefit to be gained from this type of functionality beyond the "tick the box", the focus should be to offer this functionality at the lowest possible cost. As this functionality typically has been part of the system for a long time (likely it was differentiating at some point in the past), R&D efforts concerning this functionality should focus on minimizing the total cost of ownership and attempts to add and perfect functionality should be discouraged. Most of the efforts should be concerned with simplifying the structure and replacing proprietary software components with open-source or commercial components. If possible, removing commoditized functionality from the system should be considered if this would be feasible from a customer expectations perspective.

The next layer is the differentiating functionality layer. This layer contains the functionality that makes customers select our products or systems over those from competitors. Depending on the industry, the functionality will be in this layer for a few weeks, months or years before it commoditizes and transitions to the bottom layer. While the functionality is driving the differentiation, though, the focus should be to invest such that the value delivered by this

functionality is maximized for customers. This is typically accomplished by enriching the functionality in various ways, such as adding variation points to allow for different configurations, supporting multiple paths through the use case and supporting exceptional cases well as well as deeply integrating the user experience of different chunks of functionality in a fashion that makes interacting with the system natural for the user. For systems that are less driven by user interaction but by standalone, autonomous behavior, increasing the value of differentiating functionality is accomplished by focusing on those aspects that drive value to the customer and these may be unexpected. For instance, in many industrial systems, increasing the amount of autonomy and decreasing the need for operational support by humans is an important factor as it drives down cost.

The top layer is the innovation and experimentation layer. This is where the company experiments with different innovative ideas in order to identify new future differentiation. Innovation pipelines such as the one in figure X focus on this layer of the model. Humans have an amazing capability to rationalize why certain ideas would be good or not good, but reality shows that we are not very good at predicting the success of innovative ideas. Consequently, around half of the features in a typical product are hardly, if ever, used. The purpose of the innovation and experimentation layer and the innovation funnel is to validate the relevance and potential value of innovative concepts through the measurement of customer behaviour rather than through customer interviews.

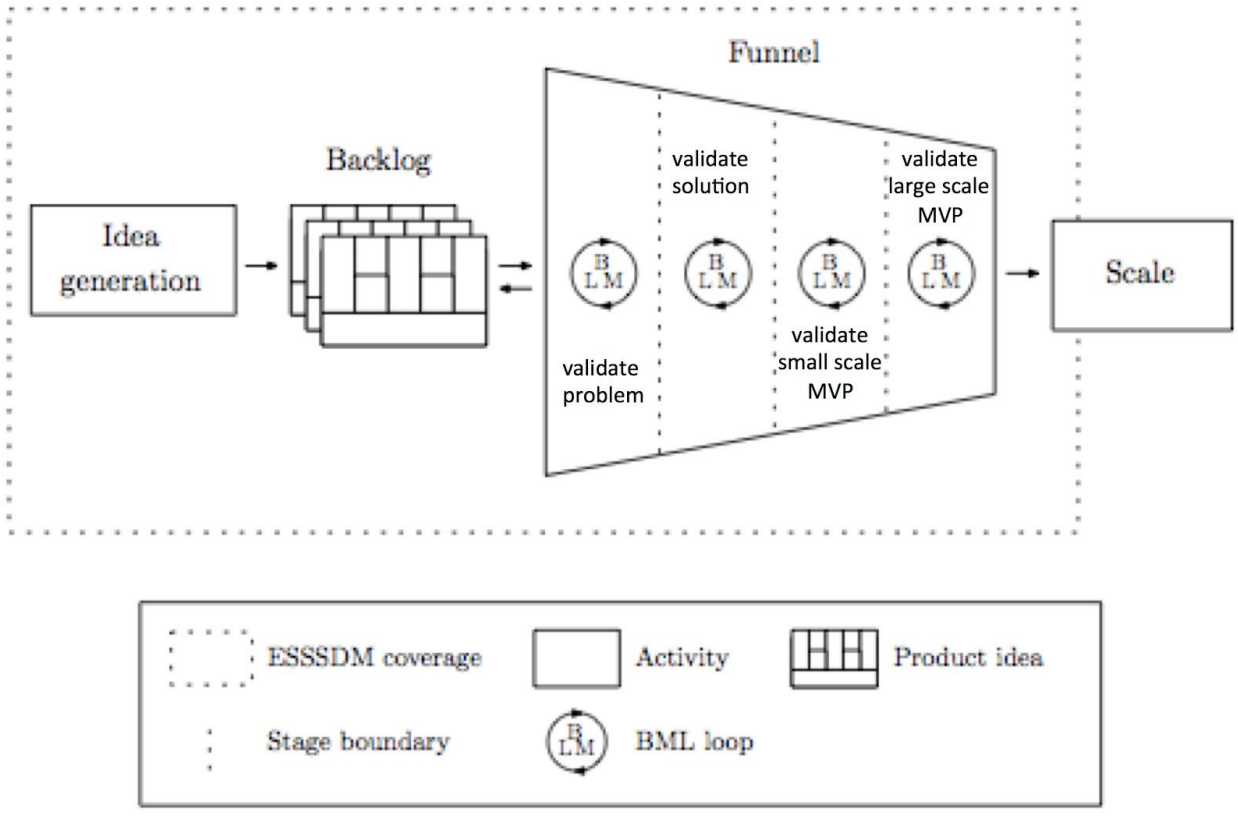

Figure X. Innovation Funnel [Bosch et al 13]

The goal of the innovation and experimentation layer is run as many experiments as possible against the lowest cost per experiment. As we are not good in predicting which concepts are the most valuable, we need to test many ideas and trust that we end up with a set of innovative concepts that resonates with customers and that drive future differentiation. The ability to rapidly test a concept requires us to build the realization of it as rapidly and easily as possible. Consequently, this requires that it can be built with as little integration in the existing product or system software as possible. Ideally, the idea can be realized on top of the product interface and only interact with the rest of the functionality through that interface. This significantly reduces the cost of implementation and simplifies the removal of the software associated with the innovative concept if it does not deliver the value that was expected.

Although the concept of these layers is intuitive and easy to understand, it is important to realize that there is a time dimension to the 3LPM as well. Functionality flows downward through these layers. Thus a promising innovation concept starts its life in the top layer. Once it is validated and the customer value confirmed, it transitions to the differentiating functionality layer where the company focuses on maximizing the customer value and monetizing accordingly. Finally, once it starts to commoditize, the functionality flows to the bottom layer where the focus shifts to minimizing total cost of ownership. This notion of functionality moving through layers and the key priority of the R&D organization shifting for this chunk of functionality each time that it transitions is at the heart of 3LPM and for many the less obvious aspect of the model.

The second main aspect of the 3LPM are the interfaces between three layers. The 3LPM can be used in a conceptual fashion, where components are mapped to the three layers, or it can be used as a concrete architecture where functionality physically moves between layers when it changes status. Although the functional decomposition of architectures can follow many paths, using a top-level decomposition using the 3LPM allows the company to adjust the organizational structure of R&D to follow the same principles. This, then, allows each team to focus on the primary driver for the layer that it is responsible for, i.e. minimize total cost of ownership, optimize the value to customers and maximize the number of experiments. Stable (not static) interfaces can then facilitate the decoupling between teams and the transition of functionality between layers.

The 3LPM is has four primary uses. First, it can be used as a tool to assess the current state of the system architecture and the allocation of R&D resources as a means to control resource allocation. Second, it facilitates the definition of the desired state for a system and its architecture as well as and the development of a transition plan from current to desired state. Third, it provides a tool for describing the boundaries between platforms and systems built on top of those platforms. Finally, it helps an organization structure its engagement with the ecosystems around it by classifying functionality as internal and external, defining interfaces between the company and partners and the transition of functionality between these layers. The notion is that the company engages the innovation ecosystems around it to share the cost and

risk of innovation, connects with companies offering complementing functionality in its differentiation ecosystem and that it partners with the commodity ecosystems to minimize the total cost of ownership of commodity functionality. The difference of ecosystem engagement around the differentiating layer as compared to the other two layers is that the differentiation ecosystem engagement tends to focus on complementing the functionality delivered by the company with that of other companies. For the other two layers, the focus is on the functionality delivered by the company itself.

WKI: Unlocking the Internet of Things

WeKonnektIT (WKI) is a 1000+ person company offering products, solutions and services for connecting devices of all types to the internet. The company has been going for more than 25 years and was one of the earliest players that entered the machine to machine communication trend in the 1990s. As there initially were no technical solutions available for accomplishing connectivity, the company started of building its own hardware, protocols, software as well as mechanical solutions for deploying units out in the field.

The company has operated for many years as a fully vertically integrated product and solution provider. Its customers would request connectivity and WKI would provide a turnkey solution including a subsystem to be added to the customer's product or system, local routers to provide connectivity as well as data collection, analysis and reporting solutions. This allowed its customers to focus on what they were best at and WKI would solve everything else.

This strategy has served the company well over most of its 25 years, allowing it to grow from a two person shop to a large company with a sizeable R&D department. During the last years, however, the first dark clouds have been appearing on the horizon. The competitive landscape has changed with many new entrants and a significant push by customers to standardize solutions, protocols, connectivity and platforms with the intent of allowing for more interchangeable solutions. With the emergence of the Big Data trend, followed a few years later by the Internet of Things trend, the comfortable niche market in which WKI has lived for many years has now turned into a global and highly competitive market. On the one hand, this is great as the number of potential customers and the size of the market is just exploding. On the other hand, the competitive landscape suddenly contains hundreds of players, new and old, and the company is trying to make sense of how to respond to this.

The main challenge that the company meets is that its customers ask it for different deployments than the traditional turnkey solution. Sometimes it just wants the connectivity modules, sometimes customers ask for the data collection part. In other cases, WKI is asked to perform the integration of a complete solution but the solution is composed of non-WKI components. In short, although the company is customer-focused and wants to do right by the customer, it is clear that its current strategy is no longer working.

The leadership team, especially David, the recently appointed CEO, Nathan, head of sales and business development, and Adam, CTO and leading the R&D efforts, have worked hard on creating a new strategy that will allow the company to meet the new market reality. The first step was to analyze the current situation and to get clear on the primary challenges that need to be met. The team identified three main issues:

- From a turnkey solutions provider, the company needs to clearly separate where it provides standard products and where it offers services for the creation of customer solutions, integration of systems or even operational services. This will have organizational implications as it is difficult to focus on product development and service delivery with the same team.
- After receiving dozens of requests for APIs that would allow other companies to build solutions on top of the WKI systems, it is clear that the company needs to get serious about a platform that it can offer to third parties. Even though it would make it easier for others to integrate WKI based products into solutions, it also allows for third party app developers to enrich the solution space provided on top of WKI.
- The team spent significant time analysing where the current R&D efforts are being spent and got to the unsettling conclusion that the vast majority of R&D resources was spent on maintaining and evolving proprietary hardware, communication protocols, driver software and other commodity solutions for which industry standards and accompanying products already exist. It is clear that WKI needs to free up its own resources from these activities if they hope to stay competitive in the future.

Based on the analysis of these challenges, the company decides to kick off three teams, organized around each of the challenges and to ensure that there is governance by creating a governance team consisting of the leadership team and the team leads for the three teams. The teams receive the designations Team Liberation (freeing up resources from commodity functionality), Team 3rd Party (creating a platform API for third party developers) and Team New Growth (recommending a new business model and organization to deal with the changed competitive landscape).

In the coming parts of the book, we will use the WKI case to illustrate the concepts that are introduced. Although it of course is a fictive case, it combines experiences and learnings from numerous companies that we have worked with in the past. As such, it represents an accurate representation of the situation at many companies.

Part I: Controlling Resource Allocation

In organizations that traditionally have treated all software as equal, there typically is little awareness across the organization where the R&D resources are allocated. Consequently, the first challenge that we need to address is to gain understanding of where resources are going. We use the 3LPM to assess the relative and absolute allocation of resources to commodity, differentiating and innovative functionality. Once we understand where resources currently are

allocated, we can change allocation to constrain investment in commodity and focus resources on differentiation and innovation.

In the following, we start by describing the process for current state assessment of the architecture and follow with an assessment of the R&D resource allocation. Then we discuss the desired state. The process for assessing current state consists of the following steps:

- Categorize the architecture
    - Classify functionality
    - Assess intermingling
    - Assess dependencies
- Determine resource allocation
- Summarize results and present conclusions

Once the current state has been established, the next step is to define the desired state in terms of resource allocation and to define the actions required to realize the desired state.

Current State Assessment: Architecture

The current assessment of the software architecture starts with a first step where the functionality is categorized into the three different types. Then it assesses the intermingling of different categories of functionality and the complexity of the dependencies between these.

Classify Functionality

The first step that is required for us to make any progress is to categorize the functionality in the system. For this, we use the notion of a system element and assume that we can hierarchically break down system elements into smaller system elements. At this point, we are not concerned with the dependencies between these elements. We are decomposing the system until we reach a state where each of the leaf elements is cleanly categorized into one of three categories: commodity, differentiating and innovative.

The process for categorizing the functionality in the system is recursive decomposition. First, we start with the system as a whole. We call it the top-level system element. This system element is then categorized as commodity, differentiating, innovative or mixed.

If the system element is classified as mixed and can not be allocated to one category only, we break down the system element into smaller system elements and repeat the process for the new system elements.

An important observation here is that the assessment of the categorization of functionality needs to originate from the customer or from a role that is intimately familiar with the customer, such as product management, sales and marketing roles. The reason for avoiding the situation

where R&D is in charge of the assessment is that engineers tend to mark functionality as differentiating for a variety of reasons even though customers consider it to be commodity.

Example: One of the main internal platforms at WKI is the connectivity module that is added to a client's product or system to provide basic functionality. The connection module consists of an electronics board, a hardware abstraction interface, a connectivity module, a data storage module, a data acquisition module, a data analytics module and a monitoring module.

The main difference between WKI solutions and the new entrants is that WKI focuses on performing part of the data analytics on the connectivity module in order to communicate accumulated and analysed data as well as raise warnings and alarms more quickly.

Applying the process of categorizing functionality shows that the connectivity module is entirely commodity. The challenge, however, is that it contains several proprietary protocols that were developed before the emergence of industry standards. WKI can not stop support of these protocols due to legacy deployments. However, currently the R&D organization is still investing significant resources into the module and there are numerous opportunities to decrease this.

The monitoring module, on the other hand, is viewed as differentiating as it allows the company to offer "edge computing" solutions that facilitate monitoring, identification of anomalies as well as raising of warnings and alarms. The module is built such that algorithms for analysis can easily be embedded and the module offers a set of interfaces that are used by client projects.

The data storage module is a mixed component as it on the one hand offers basic database functionality, but on the other hand it contains differentiating and innovative algorithms for deciding on deleting data, smart storage of aggregated data and selective storage of anomalous events. As the module is marked as mixed, we further decompose it into a database module, an analytics module and a smart storage module. The former is classified as commodity and the latter two as differentiating.

As an observation, in our experience with different companies, it is clear that there are many approaches that architects use to present a high level view of the architecture. Therefore, we encourage companies to start from the model that is in place already. The two principles, though, that should really be enforced are (1) ensure that it is the customer's voice that decides the classification of system elements and not the internal beliefs of the organization and (2) ensure that each leaf element is allocated completely to one category. There will be significant tendencies in the organization to compromise on these principles, but the value of using the 3LPM for assessing resource allocation is by providing clear structure, numbers and insights.

Assess Intermingling

The result of the previous step is a hierarchical breakdown of the system into system elements where the leaves of the hierarchy are solely mapped to one of the three categories. Depending

on the level of design erosion in the code, the challenge is that the leaf system elements may no longer relate to actual components or modules in the system, but rather to different code segments in the same file.

Although we are not yet preparing any refactoring efforts, we are looking to assess the complexity of factoring out the functionality in different categories as we are looking to reach a point where each category is optimized for a different business metric. For innovative functionality, we want to make it as easy as possible to test different innovative ideas with customers. For differentiating functionality, the metric is concerned with maximizing business value, which often translates into supporting non-standard use cases, variation points and other mechanisms to allow for configuration and personalization. Finally, commodity functionality is concerned with minimizing total cost of ownership. We have to provide the functionality in this category, but want to do so at the lowest possible cost. If code is intermingled, it is impossible to optimize for the specific business metric. The result is that depending on the engineer or team working on that part of the system, it will be optimized for different metrics.

There are different ways for assessing intermingling, but the process for assessing intermingling is a bottom-up process, starting at the leaf system elements. The first step is to identify the system elements of different types that are present in the same component or module. The reason is that for these system elements, there are no interfaces and any separation between these would require code refactoring and the introduction of new interfaces.

The next step is to increase the abstraction level to system elements that are independent modules or components but that sit in the same higher-level system element while being of different types. As components within the same subsystem tend to have higher coupling than components in different subsystems, it is important to identify these components as a future refactoring to meet the requirements of the 3LPM would require these components to be decoupled. Decoupling often means the introduction of more abstract interfaces as well as the moving of code between components to reduce the need for interfaces.

The process repeats itself up the hierarchy of system elements until the top level, i.e. the system itself, is reached. Each system element that is mixed should be marked with an indication of the relative prevalence of commodity, differentiation and innovation. At the top of the hierarchy, after iteratively marking its components, the system also receives an indication of the relative allocation of the three types of functionality.

Note that at this stage, we are only recording the intermingling and not doing anything about it. However, as we move towards controlling resource allocation we can use our knowledge about the intermingling at different levels as a tool to ensure that, even though the functionality is not physically split into different components, we can still assess whether requests for new or changed functionality will predominantly affect commodity or other types of functionality inside system elements. If the requirement primarily affects commodity, the R&D organization can and

should try to decline the request and if not successful aim to minimize the investment and amount of actual work.

As an example from WKI: we already introduced the data storage module as a mixed component. On the one hand offers basic database functionality, but on the other hand it contains differentiating and innovative algorithms for deciding on deleting data, smart storage of aggregated data and selective storage of anomalous events. As we broke down the system element into conceptual modules, we identified a database module, an analytics module and a smart storage module. The former is classified as commodity and the latter two as differentiating. However, these conceptual modules are currently not separated into different system elements but part of the same source files. These files have been separated based on other principles than the 3LPM.

Assess Dependencies

Once we have categorized the components and modules of the system and assessed the intermingling of functionality, the next step is to assess the dependencies in the system. The reason for assessing dependencies is twofold. First, we need to establish the complexity of the dependencies between the different parts of the system as any future refactoring would need to focus on reducing these dependencies. Second, ideally commodity functionality only depends on other commodity functionality. Differentiating functionality depends only on other differentiating or commodity functionality. And, finally, innovative functionality can depend on the other types, but preferably less on other innovative functionality.

The rationale for constraining dependencies to the restrictions mentioned above is that the 3LPM seeks to organize functionality into three layers with defined interfaces between these. As is typical in layered architectures, higher level components can depend on lower level components, but not visa-versa. In addition, the reason for limiting dependencies between different types of innovative functionality is that this category of functionality is likely to be removed when customers do not appreciate the functionality and it does not deliver the expected value. In many cases, there is little consideration of removing functionality once it has been added to the system, but the fact is that functionality that, for whatever reason, does not deliver the expected benefits for the customer should be removed as soon as possible as the future total cost of ownership, due to increased complexity of the system itself as well as of the user interaction typically by far outweigh the cost of removing functionality soon after its introduction. In addition, waiting to remove obsolete functionality often causes other software to become dependent on this functionality, increasing the cost of removal over time.

One of the challenges in some industries is that customers are very powerful and can demand functionality to remain in the system for their use, even if the rest of the customer base is affected negatively. To avoid this, the communication with the customer should clearly indicate that functionality labeled as innovative is not guaranteed to remain in the system. If the company decides to remove it and some customers are very much attached to this new

functionality, it can be moved from the system to the customer-specific functionality that extends the system in similar ways as third party developers can build extensions to the system. That allows the company to keep the system clean while offering flexibility to customers. This interface can and perhaps even should be the same interface as the one between differentiating and innovative functionality. By doing so, it helps the company to maintain only one interface to external and internal partners seeking to extend the system. The architects in the R&D organization can then focus on ensuring the stability and expressiveness of the interface and to balance the two. Minimizing the number of interfaces to maintain and evolve tends to increase the quality and stability of the remaining ones.

As this step is concerned with assessing current state, the main activity is to identify the dependencies and to mark the ones that are violating the principles that we discussed above. This will give an indication of the amount of effort required when seeking to refactor parts of the architecture. However, at this stage the main focus is on documenting and creating awareness of the violating dependencies. During the selection of functionality to allocate resources to, one of the considerations should be whether illegal or undesirable dependencies will be involved in the new development. If this is the case, the team should investigate what can be done to avoid increased future refactoring cost and whether it is possible to include some refactoring into the development efforts to achieve the same outcome while removing some of the technical debt.

Example: At WKI, the team analysed the dependencies in the connection module and identified that there exist a number of intricate dependencies between the data acquisition module, the data analytics module and the monitoring module. As the data acquisition module is classified as commodity, the data analytics module is mixed and the monitoring module is differentiating, the team raises this as a concern. If the company would decide to re-architect the system following the 3LPM principles, these dependencies would need to be redesigned in order to ensure that the interfaces between these modules are sufficiently decoupled to allow for independent evolution.

Current State Assessment: Resource Allocation

Assessing resource allocation is a hard problem in most companies as the discussions around resources often take place in yearly budgeting processes with high-level follow-up during the year. This causes a situation where it is unclear where all the people in R&D are spending their time on. This frequently leads to confusing discussions between “the business side” and the R&D organization concerning the value added by the R&D organization of differentiating functionality and the number of new products created by the R&D organization. As in most organizations the vast majority of resources is allocated to commodity functionality, this is not a surprising state. However, often the R&D organization does not have a clear way of communicating the actual allocation of resources as it does not keep track of these at the level of detail required.

As we now have a hierarchy of system elements to allocate resource estimates to, we first have to get clear on what approach to resource estimation we will use as there are several alternatives. First, we could collect the total amount of resources allocated to each system element since the inception of the system or the specific system element. Second, we can collect the specific amount of resources allocated during the current time period, for instance for the system release currently under development. Third, we could predict the amount of resources that we expect to see invested in the system element for the foreseeable future. Finally, if the organization into teams follows the component structure, the size of the teams associated with the various components will give a direct measure of resources allocated to each component.

For the purpose intended here, we need to accept that resources allocated in the past are sunk cost and it is important to not fall in the trap of assuming that because something was expensive in the past, we should continue to invest in it. However, we can use the historical resource allocation as a predictor for the future and use it as a base estimate for the system element. Similarly, we can use the future predictions as a tool to increase or lower the base estimate. For instance, if it is clear that there will be significant new requirements affecting on part of the system, then the resource allocation estimate for the system elements in that part of the system should be increased.

Before we can make resource estimates, we also have to establish the unit in which we will express the resource estimate as well as the time window that we employ. To start with the latter, the time window used in the effort estimation should be in line with the ability of the company to take action in terms of refactoring and shifts in resource allocation. For most companies, this means that the window should at least be six to twelve months, even though three months might work for smaller, more agile companies with significant freedom in their resource allocation. Once the time window has been selected, the next decision is to select the unit. It is easy to use an ordinal scale such as “low, medium, high”, but selecting an absolute scale allows for many more uses of the results of the current state assessment. The unit can be units of work, e.g. person hours, days, weeks or months, or a monetary unit, such as € or $. This means that the resource allocation estimate for a system element may be expressed as, for instance, 120 person days/year.

Now that we have established the principles of resource allocation, it is important to realize that this requires estimation (or even guestimation) at most companies. The process for allocating resource allocation estimates to system elements is an iterative process, starting from those parts of the system where known numbers exist or at least can be estimated with a high degree of accuracy. Typically, there is an overall resource number available for the system as a whole. For larger subsystems that are allocated to specific organizational units, resource estimates are often easy to establish. Once the “easy” numbers have been allocated, the process becomes iterative, with a “down” activity and an “up” activity. The “down” activity starts from a non-leaf system element for which a resource allocation is available. Using the overall resource allocation, the action is to estimate the allocation to its constituent parts. Once this has been

conducted for a part of the tree, the next step is the “up” activity. During the “up” activity, the bottom-up numbers are combined to verify that the sum of the bottom-up numbers aligns with the known numbers and, where missing, that the numbers at least look reasonable. The “down” and “up” activities are repeated until all system elements have received an estimate and the alignment between higher level and lower level estimates as well as horizontally between different subsystems has been verified.

In cases where the individuals conducting estimations feel particularly uncertain, the estimates can be extended with a confidence level. In later stages, the confidence level can be used to determine when decisions need to be made.

Example: WKI is a 1000+ person company, but it is very R&D heavy, meaning that two thirds of the staff work with R&D. Several of those work with customer projects, but around 400 people work on the products, components and modules that the company uses in its customer projects. When analysing the allocation of these 400 people, it showed that more than 340 people were working on commodity components. The reason for this is that the company built significant amounts of bespoke functionality during the time that there were no generic solutions available on the market. Over time, however, the situation has changed significantly and today many of the modules and components could be replaced by standard components available on the market against much lower prices than what WKI currently experiences in cost. The analysis clearly showed how, over time, the competitive position of the company started to be negatively affected.

Current State Assessment: Summarizing the Results

At this point, we have created a number of assets. The first is system element tree that shows a full breakdown of the system in its constituent parts. Depending on the size of the system and granularity that the assessment is conducted, this tree can match the system architecture, stop at a higher level of abstraction or, potentially, extend even deeper than the system architecture by breaking individual files into smaller parts. This system element tree not only captures the elements, but also classifies each element into it being commodity, differentiating, innovative or mixed. Mixed assets have a relative allocation to the three categories, calculated in a bottom up fashion. For instance, a component may be 70% commodity, 25% differentiating and 5% innovative. However, each leaf element will be allocated as purely being in one category.

The second asset is an overview of the dependencies in the system. As decoupling and independent evolution of system elements go hand in hand, it is critical to manage interfaces and dependencies. This relates strongly to the management of architectural technical debt, even if this is not the focus of this book.

The third asset is an overview of the estimated resources consumed by each part of the system. As we have classified the system elements, this allows us to calculate the relative and absolute allocation to commodity, differentiation and innovation.

These three assets, when combined, provide us with information that we need for the next step in the process: shifting the resource allocation around to align it with the business strategy and to ensure maximum return on investment of the R&D budget.

Controlling Resource Allocation

When we started the work of assessing our resource allocation, we did not have accurate data on the relative allocation of R&D resources to the three categories of functionality, i.e. commodity, differentiating and innovative. The purpose of this first part of the book is to gain clarity on the resource allocation and the summary that we created in the previous section contains that insight.

Understanding the current resource allocation is only the first step in the process. If your company is like most others, the vast majority - perhaps even 80-90% - of resources goes to commodity functionality. There are very good reasons to invest in commodity functionality: it helps to maintain existing revenue streams. Whenever there is an update to an operating system or some other software that the system depends on, maintenance will be required to bring the system up to date again. Because if we didn't, we wouldn't even be able to sell it!

So, the goal is not to bring the investment in commodity functionality down to zero. However, we do need to realize the price that we are paying in terms of opportunity cost. Most companies calculate their R&D budget as a percentage of revenue. This budget is then allocated to various activities. We know that investing in commodity functionality has a low return on investment (RoI) as customers will not pay you for functionality that already exists and used to work. At the same time, the risk associated with the investment is very low. For differentiating functionality, the RoI is much higher, but at the same time the risk is higher too. We are extending functionality with additional use cases and enriching it in different ways that we think adds value to customers. However, we will only know if it really adds value and can be considered differentiating after it has been released and we get the feedback from customers. Finally, innovative functionality has the highest RoI but also the highest risk. In innovation communities, the rule of thumb is that 9 out of 10 ideas fail which is why we seek to test and validate as many ideas as possible against the lowest cost per idea. So, if the company does allocate 80-90% of resources to commodity, there is a significant underinvestment in differentiation and innovation and the resulting RoI is very low.

As a general rule, our recommendation is to minimize investment in commodity to the lowest possible level. Then agree on a reasonable division between innovation and differentiation. Finally, based on the agreed division define the portfolio of R&D activities that optimizes the return on investment.

In practice, however, our experience is that, for most companies, the ideal situation is too hard to accomplish in one step. So, we recommend that once the investment level in commodity

functionality has been established, the first goal should be to cut that level in half. That means that if the result is that 80% of resources is allocated to commodity functionality, the first step should be to bring that down to 40% in the next resource allocation cycle.

One mechanism that works well is to use the system element hierarchy. The simple rule is that for system elements that have been labeled as commodity, only bug fixes and updates driven by external causes, e.g. operating system upgrades, are allowed to be put on the backlog. Every other request for new development in commodity components will be declined. As implementing this often requires a significant change to the behaviours, norms and values in the organization, our experience is that an escalation mechanism is necessary for the cases where some in the organization feel very strongly that certain commodity functionality should be added and constructed. Simply using this mechanism in a principled fashion will result in a 50% reduction of resource allocation to commodity in the cases that we have used this approach.

As part of planning the work to do in the next sprint, quarter or release, the new requirements and features requested by product management need to be evaluated and impact analysis needs to be conducted. If it shows that requested functionality requires changes and new development in components that have been classified as commodity, the requirement should be automatically rejected.

Concluding, in this first part of the book, we have focused on identifying in more detail where our R&D resources are allocated with the intent of shifting a significant part from commodity functionality to differentiation and innovation. Using the mechanisms and approaches described here, even without any refactoring and using 3LPM just as a conceptual model applied to the current system architecture, we can achieve a major improvement in the effectiveness of our R&D efforts. In the next parts we expand on the basis that we built here by using 3LPM for complementing purposes that will further streamline the R&D function and deliver increased levels of effectiveness.

Part II: Refactor Software

The 3LPM has multiple application areas in addition to controlling resource allocation. In this part of the book, we discuss the refactoring of an existing software system using the 3LPM as a basic architectural style. One might wonder, however, why we would care about refactoring the system if so many benefits can already be achieved just by using the model as a conceptual one. The primary reason is that employing 3LPM in a system that is fundamentally structured differently requires an enormous amount of discipline in the organization that often is difficult to sustain over the long term. This is because many components will contain two or all types of functionality and ensuring that commodity functionality only receives the minimally required investment, such as bug fixes, is difficult. In addition, mixing innovative functionality with the other functionality in the system complicates its removal in the case that the customer value is not established.

When we refactor the system to physically separate commodity, differentiating and innovative functionality and define interfaces between the layers, the principles underlying 3LPM become embedded in the architecture, processes and potentially even the organization. Rather than maintaining a high discipline level, abiding by the 3LPM principles would simply be part of the way we do things. In this part of the book, we describe the process of refactoring the software in your system according to the 3LPM.

In the first part of the book, we already created three assets, i.e. the system element tree, the dependency graph and the intermingling of functionality types. These three assets, when combined, provide us with part of the information that we need for the refactoring process. The refactoring process uses these assets and then conducts additional steps:

- **Define desired state**: This step is concerned with the formulation of the desired state for the system in terms of system structure, intermingling, dependencies, amount of commodity versus differentiating functionality as well as the allocation of resources.
- **Develop transition plan**: The transition plan defines how we're going to get from the current state to the desired state with clear steps, effort estimation, a time plan, prediction of observable benefits and building the mechanisms for tracking of the realization of these benefits.
- **Execute transition plan**: The final step deals with actually realizing the plan. This is of course the hardest part as this is where the organization as a whole needs to commit and perform the actions to accomplish the refactoring. As refactoring does not always deliver immediate and easily recognizable business benefits, most organizations struggle with successfully completing these types of plans.

In the remainder of this part of the book, we describe each of the steps in the refactoring process in more detail.

Defining Desired State

In this step, we create the same three assets as we developed in the previous phase, but now from an aspirational perspective: what should our system element tree, interfaces and intermingling as well as dependencies and the resource allocation look like in order to optimally deliver on the business strategy of the company. Thus, the goal here is to model the system from a perspective where the system is structured optimally from the perspective of the factors that we prioritize in the 3LPM. There are several factors associated with the system being optimally organized:

- First, it is structured in a way that is as closely aligned with the categorization of functionality into commodity, differentiation and innovation as possible.
- Second, system structure and interfaces between the 3LPM layers are aligned with the business strategy, meaning that the highest priority work can also be added the easiest to the system.

- Third, it is organized such that it is easy to transition functionality from the differentiation layer to the commodity layer.

As this point, it is important to realize that there are two “pure” approaches, the evolutionary and the revolutionary one. First, one can take the 3LPM architecture as a conceptual model, but keep the existing system architecture largely as it is. In this case, the functionality is labeled according to the 3LPM and the dependencies are labeled as “desired” or “obsolete” and the architects and development team can use this information during their development to avoid the system from becoming even more intermingled and to avoid dependencies that really should not be there. In addition, as every system requires a continuous investment in refactoring, the conceptual model gives the team guidance on the direction for refactoring. The evolutionary approach is the least effort consuming approach that already provides benefits.

The second, more radical approach is to use the 3LPM as the physical top-level architecture and to define APIs between commodity and differentiation and between differentiation and innovation. This offers many more benefits, such as organizational alignment (a la BAPO model [reference]) so that each team (commodity, differentiation and innovation) has clear, but distinct goals to pursue. Also, it clearly marks transitions between different phases that functionality evolves through. Finally, it allows for clear control of resource allocation to different types of work. However, reaping these benefits also requires a significantly higher level of investment and a quite fundamental refactoring of the system.

It should be noted that there is a third, hybrid approach where the evolutionary and radical approaches are combined. In practice, this means that the architecture is left intact in some parts of the system and organized according to the 3LPM in other areas. Although the purists among us will balk at this notion, there often are very good reasons why there is a need for a fundamental refactoring in one part of the system while leaving other parts alone. For instance, one part of the system may be affected by new and significant upcoming requirements and refactoring that part before incorporating these requirements may result in much lower overall cost and a significantly improved responsiveness in that part of the system. Meanwhile, a different part of the system may be poorly structured, but experience very little in terms of maintenance and evolution. In that case, the return on the investment required to refactor that part of the system will never be positive and hance refactoring should be avoided.

In the remainder of this section, we will assume the radical approach as this has the most implications and is the approach that provides the most benefits if your organization can muster the resources. The remainder of this section is organized as follows. First, we put each element (chunk of functionality) that is purely allocated to one type in the associated layer. This means, among others, that for intermingled elements, different parts of a file end up in different layers. Second, we review the dependencies between different elements to provide an understanding of all the dependencies. This allows for a design where some dependencies are removed, some are combined into higher level, more abstract interfaces and some are maintained as they are.

This then allows for the definition of the system level interfaces between innovation and differentiation layer and between differentiation and commodity layers.

Allocate System Elements

In the first phase, we broke the system into a hierarchy of system elements where the leaf elements are labelled as either commodity, differentiating or innovative. As such, the allocation of the leaf elements to the three layers of the 3LPM. However, it likely leads to a situation where functionality that in the current architecture is considered as belonging together is now divided over different layers.

The above brings us to one of the key points to consider when defining the desired state. Every system has an architecture and even if it has eroded over time, there will still be a basic decomposition and set of fundamental architecture design decisions that has lead to the structure of the system as it exists today. When defining the desired state, it is important to realize that this might require revisiting several of the original design decisions in order to structure the system according to the 3LPM. This will initially feel unnatural and as going against the architecture, especially as many of the main components will be divided over two and potentially three layers. However, it is important to realize that the 3LPM is imposing a new architectural style on the system. One that is driven by the need to separate commodity from differentiating and innovative functionality and less driven by the traditional design principle of keeping related functionality together. This does not mean that we throw the principles of coupling and cohesion completely out of the window. Instead, we reinterpret these principles as functionality that is commodity should not be tightly coupled to differentiating functionality, even if these two logically belong together. Similarly, functionality that is innovative should not be combined with differentiating or commodity functionality. We add a time dimension to coupling and cohesion as well as introducing a new criteria to decide when functionality can or should be logically put together.

Although the notion of defining the desired state is concerned with assigning functionality in the system elements to the three layers, this is not enough as the different chunks are just thrown together in the three buckets, but there is no architecture or design yet. In the next steps, we iteratively transition from assigning functionality to the three layers to a design that optimally captures the business strategy of the company.

Review Architecture Design Decisions

Once the different system elements have been allocated to the three layers, the next step is to understand the dependencies that exist between these elements. As part of this analysis, we have to understand that there are two types of dependencies. First, there are dependencies that are inherent in the problem domain. Two system elements simply need to be connected to exchange information, trigger actions or are in other ways dependent on each other. The only way to break this dependency is by re-architecting the design, meaning to either combine the system elements into one or to break up the functionality in the two system elements in two new

system elements that do not depend on each other. These dependencies refer to the complexity of the problem domain.

The second type of dependency is concerned with the complexity of the solution domain. These dependencies are not necessary for solving the “problem domain design” of the system, but occur due to the solution chosen by the original architects and designers of the system or, as is frequently the case, due to architecture erosion and the accumulation of technical debt. This means that these dependencies are a consequence of the solution domain and the erosion of the original solution that took place over time. This type of dependencies can be removed if the underlying architecture design decisions causing these dependencies are reconsidered.

The basic structure of the system is due to the architecture design decisions that were taken during the initial design of the system as well as the decisions taken during the evolution of the system. At this stage we need to review these design decisions as we have applied a fundamentally different starting point for the design of the architecture, i.e. the 3LPM. This decision may have invalidated several of the earlier design decisions.

There is one class of dependencies that are directly caused by the application of the 3LPM. As components are broken up and divided over the three layers, there is an immediate translation of the high cohesion that the well designed component has into a high degree of coupling. No matter where the decomposition happens, it will have this effect. To further complicate this, the 3LPM asks that functionality is moved between layers when the functionality becomes differentiating and finally commodity. So, rather than thinking about functionality that is added to the system as something that needs to be massaged deeply into the component where it logically belongs to, we need to find ways where the functionality that logically belongs together can be moved, as a unit, between layers with minimal effects on the larger component and subsystem in which it lives.

As we aim to define the desired architecture as a basis for transition planning, we need to keep a careful balance between starting from scratch and keeping part of the existing architecture. Starting from scratch would mean imposing the 3LPM on the categorized functionality and starting to take design decisions to structure the system again. Basically, this means doing a fundamental redesign of the system. Although this could be considered the most pure and perfect approach, the danger is that one ends up at an architecture that is significantly different from the current architecture but where the differences do not deliver significant business value. On the other hand, keeping part of the existing architecture will reduce effort during the transition phase, but risks leaving too much historical baggage in the system that no longer delivers value. As with many things in life, the right answer will be situation dependent and will require balancing the different pros and cons.

The best approach to take depends on the specific situation and requires expert judgement from the architects and R&D leaders. However, there are a few factors to keep in mind, including

dependencies, the ease of transitioning functionality over 3LPM layers and the conceptual integrity and simplicity of the architecture.

The result from this step is three sets of design decisions: those design decisions that need to be removed from the system, the design decisions that will stay in the refactored system and, finally, design decisions that need to be added to the system. The latter category is required to replace removed decisions or to realize the principles underlying the 3LPM. The refactoring process that we will execute later on is, among others, concerned with realizing this new set of design decisions.

Define Interfaces

The final activity in the definition of the desired state is the definition of the interfaces in the system. The challenge is that we experience a two dimensional decomposition that seeks to optimize interfaces for two concerns. The primary concern is the separation between commodity, differentiating and innovative functionality. The intended benefit has been discussed extensively in the previous sections. However, this does require that functionality that logically belongs together still needs to be decomposed between these boundaries. Although this may seem strange, it is a very old mechanism in the world of software engineering as platforms have always required the separation of domain components over boundaries. Similarly, in software product lines, there is a similar split between the functionality in the domain assets and the application/product assets.

The second dimension along which interfaces need to be defined is between different functional components of the architecture. Although this could be considered as regular architectural design, there are some challenges associated with combining a typically layered architecture with the 3LPM architectural style as there initially may be differentiating and innovative functionality in the lower layers of the architecture as well. However, as we define the desired state in this step, for every case where a component contains different types of functionality, it needs to be split and divided over the relevant layers. This requires that the new components will have a higher level of coupling as these originate from a single component. However, in practice the typical outcome is one where higher level components access higher level components and not the other way around. The reason for this is that the higher level components contain new, more innovative and differentiating functionality that will likely rely on older, commoditized functionality. This results in the normal situation where higher level layers call lower level layers. The other way around, commodity functionality needing to access differentiating or innovative functionality, is unlikely as the latter functionality did not exist when the former was written.

An example of re-architecting a component is shown in figure X below. The 3LPM principle applied to the architecture as a whole is recursively applied to the component. The majority of functionality tends to end up in the commodity layer. A little less in the differentiating layer and the least amount in the innovative functionality layer. The architecture rule that often works the

best when re-architecting a component using these principles is the relaxed layered architecture style. In this case, a higher level component can call all layers below it and not just the layer immediately under it.

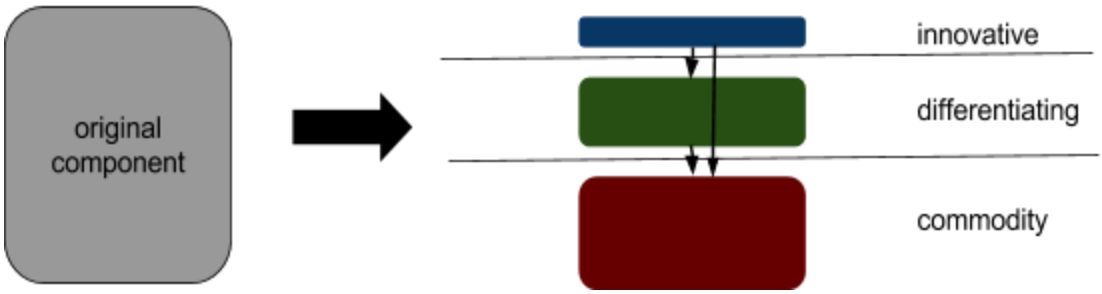

FIgure X: Re-architecting a component containing three types of functionality

As shown in figure X, we need interfaces defined between the new components that are stable over time while supporting the transitioning of functionality over boundaries downward. The intent of the interface definition activity is to ensure that the functionality ends up in the right layer while at the same time providing access to the necessary functionality across layers. The intent is to provide the intended separation and reduced coupling without introducing undue difficulties to access functionality that needs to be accessible.

An additional concern to consider while designing interfaces is that functionality will transition between layers as innovations become important differentiators and differentiating functionality commoditizes. This is an important driver of interface evolution and interface evolution is an effort consuming activity that does not add business value directly, even if it is necessary for the long term health of the system. Designing a good interface requires a careful balance between generalization and semantic richness and the periodic transitioning of functionality adds a further challenge to the activity. The details of interface design go beyond the scope of this short book, but it is important to design interfaces such that it is easy to evolve them.

Example: The team at WKI has divided its work into several activities. For the software architecture, the team has proposed an architecture where the commodity functionality is clearly separated from what the company assumes is differentiating. This includes its proprietary algorithms for early identification of issues and its edge computing functionality that allows for cost effective and resource efficient analytics.

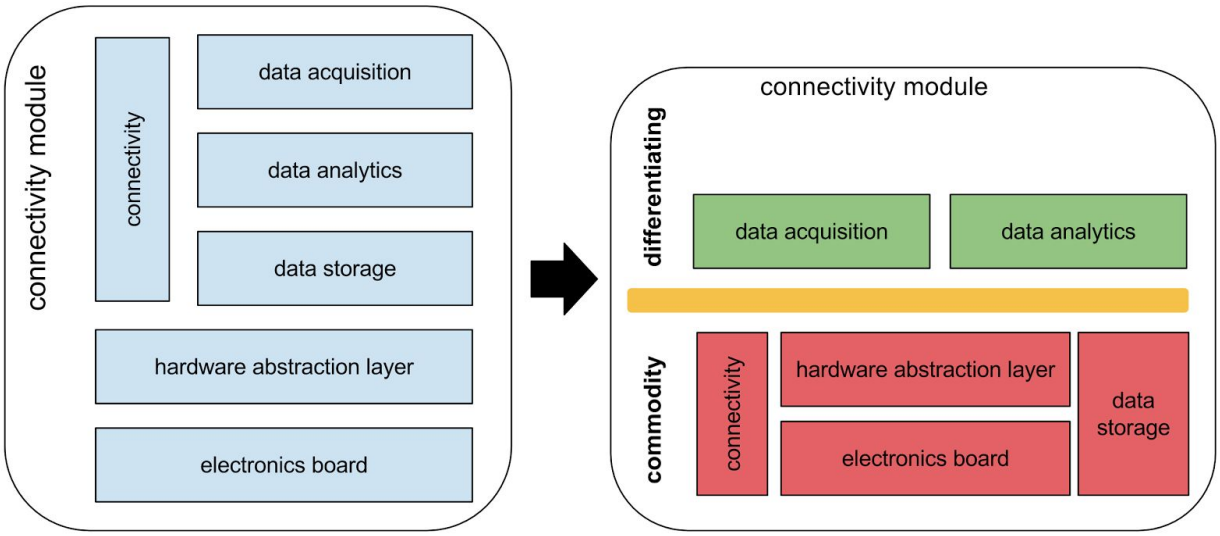

Figure X: The result of re-architecting the connectivity module

In the figure above, the results of re-architecting the connectivity module are shown. Most of the components are classified as commodity and only two as differentiating. The two that are differentiating do contain commodity parts, but this is left for later analysis and work. Analysis of the implications of the architectural change, however, has shown that there are many hidden dependencies between the modules and components that need to be resolved as part of the re-architecting effort. Although the architects and R&D managers are surprised at this, the front-line engineers feel so some extent vindicated because they have been struggling with these dependencies for many years. Despite raising this as a concern repeatedly, it is only now that the technical leadership in the company is starting to see the reality of the situation. A great example of the power of transparency!

Desired Resource Allocation

In the current state assessment, we also assessed the resources that are applied to each layer of the architecture. In the definition of the desired state, we need to provide a perspective on the desired state of resource allocation, if only to provide an insight into the benefits that can be reaped after we complete the refactoring.

The first response that we often receive when discussing this is: no resources to commodity! In an ideal world, it would be great if all the functionality that once has been created stays available and well integrated in the evolving system without any need for maintenance. However, in the real world, things don't work that way. Software, once created, will always require some level of R&D investment to stay current. The question is how much we can limit that investment without losing the stability and reliability that we have come to expect from mature functionality. For most organizations, at least the answer is that it's a lot lower than today's level of investment. However, for the organization to accept that, it needs to inspect its own beliefs and behaviours when it comes to how it decides on R&D investments in the

commodity layer. Many of these norms, values and behaviours have been hidden beneath the surface because the different types of functionality were so intertwined. As a general indication, the aim should be to lower the investment in commodity to at least half and preferably even a third of the original level.

The other question is the relative division of resources between innovation and differentiation. For most organizations, part of the freed up resources should be allocated to innovation as this is very often deprioritized between keeping the existing software working and the demands from customers concerning the differentiation layer. However, the right division between innovation and differentiation depends on the industry and the state of the system. As a general indication, investment in innovation should constitute at least 10% of the overall R&D budget. In addition, the majority of R&D resources overall should be allocated to differentiation.

Within these boundaries, the team conducting the work on establishing the desired state needs to set levels for each of the layers in the 3LPM that are defendable, but at the same time a significant deviation from the current state. When in doubt, it's better to be more ambitious and aim higher as the translation of the desired state into an actual one will require compromises. In general, it's better to aim for the stars with the hope of landing on the moon than it is to aim for the moon and realize that you may not even achieve escape velocity.

Example: The leadership team at WKI already had concluded that of the 400 people working in product R&D, at least 340 were working on commodity functionality. Understanding the need and urgency of their situation, they make the decision that this number needs to be reduced with 50%, meaning maximally 170 can work on adding functionality to commodity components and modules. The remaining 170 people are not all put on differentiation and innovation, but around half are put on refactoring the current software assets in accordance to the desired architecture that has been designed earlier in the process.

Although this results in significant disruptions for everyone in R&D, including those working with customer projects, the leadership team communicates that radical changes are needed if the company expects to be around a few years from now. They have a loyal customer base, but their cost structure and the limited amount of differentiation offered to the market is simply unsustainable.

In addition to the internal communication, the company tasks its account managers with informing their customers of the changes that are underway in the company. The intent is to convince customers that despite the short term pain, there will be significant benefits going forward that will benefit customers as much as WKI.

Conclusion

In this first step of the refactoring process, we have defined the desired state of the architecture of the system. This required us to allocate the system element tree that was the result from the

current state analysis to the three layers. The resulting, often unappealing, structure addresses the challenge of intermingling functionality to some extent, but it does not address the challenge of undesirable and unwanted dependencies. To address this, we need to review the architecture design decisions underlying the original architecture and decide which ones to remove and replace and which new ones to impose on the system with the intent of aligning with the 3LPM principles. Once we have concluded this activity, we defined the interfaces in the architecture.

Based on the desired architecture, we can define the desired resource allocation. Although it will be hard to define this in detail at this point, at least setting high-level boundaries will be really helpful. First, because it allows the organization to set a high and challenging ambition level, which often is required to get the company to rally behind the initiative. Second, because it will convince those not convinced that the potential benefits are significant and could materially change the competitive position of the company.

The result of this step is a definition of the desired architecture and resource allocation. Rather than a platonic ideal, this architecture captures the optimal solution under the constraints that exist in the organization, including those imposed by the original architecture, the business strategy and activities that add business value and those that do not as well as the evolution of the categorization of functionality over time. The desired architecture defines the basis for transition planning which is discussed in the next section. Similarly, the desired resource allocation is not created in some kind of vacuum but seeks to combine the realities of the organization with a highly ambitious goal for the organization to reach.

Transition Planning

After all the effort expended on understanding the current state and defining the desired state, the real work starts: actually realizing the transition. This is where the rubber meets the road as talking is easy, but doing is hard. The current architecture, ways of working and organization have emerged as these represent a balance between different drivers and forces in the company and between the company and it’s suppliers and customers. Consequently, there tends to be significant resistance against the changes that realizing the 3LPM approach will entail.

In order to overcome the organizational inertia, the transition plan should balance three drivers. The first, obvious, driver is the logical sequence of all the activities required. Some changes must be done before other changes because of technical or organizational constraints. Some changes are best done before other changes as this will lower the overall cost and expenditure, i.e. it’s more efficient to do changes in a certain order, even if it is possible to conduct the changes in another order. Finally, there are changes that are independent of each other and can largely be implemented without dependencies.

The second driver is concerned with maximizing benefits and minimizing “pain” or organizational upheaval. This driver is concerned with prioritizing the changes that require little from the

organization but that result in significant benefits early on in the change process in order to build momentum for the remainder of the change process. The challenge here is that sometimes changes need to be prioritized that offer a “wow” effect but have limited long term benefits for the company.

The third driver addresses the need for stepping stones. Realizing the change in one swell swoop is typically not realistic, so the transition plan should break the change into a series of milestones that offer points of success and celebration during the transition and that allow for “sprints” of transition. Not only offer these milestones rallying points for those driving the change, once accomplished these points often provide anchoring points that ensure that the organization doesn’t easily slide back to the original starting point when there is some hiccup in the market, the company or the transition process.

The remainder of this section consists of three main steps. First, we create an inventory of all the tasks that need to be conducted as well as the hard and soft dependencies between these. Second, we define a set of milestones that capture relevant accomplishments and create a transition plan following these milestones. Finally, we initiate a process where a detailed plan is generated for the first milestone. In addition, once the first milestone is approaching, detailed planning process should be initiated for achieving the subsequent milestones.

Inventory of Tasks

As a first step in transition planning, we need to create an overview of all the tasks that need to be conducted. For this, we have created the current and desired states of the architecture and system element trees as the gap between current and desired provides a clear starting point for identifying tasks.

As the architecture consists of components, the typical starting point is to identify the work that needs to be conducted for each component. If the component is purely in one category, there is no work required. However, if it contains functionality from different categories, it needs to be split, refactored to decrease coupling and an interface between the two types of functionality needs to be defined. This means that for each component not squarely in one functionality category, there is a task of breaking it up. As discussed earlier, this requires among others assigning interfaces between the new components that minimize coupling between the commodity and differentiating functionality.

The second main area is the set of architecture design decisions. During the definition of the desired state, we have identified the design decisions that need to be added and those that need to be removed. Design decisions have a structural effect on the architecture, but also add rules and constraints. This means that removing a design decision is not just concerned with removing the structural implications of the decision, but also undoing the effects of the rules and constraints that the design decision required.

As an example, especially before the widespread use of real-time operating systems in embedded systems, many systems would use an application level scheduler. This component would call a specific function or method in every component that required some type of active process or thread-like behaviour. For this to work, every component would, at instantiation time, need to register itself at the scheduler. Then, as the scheduler calls every component in its list, the function that contains the active, periodic behaviour for each component would need to limit the amount of computing time it used in order to make sure that the overall cycle time for the scheduler stays within certain bounds. This might for instance require a component to break up its periodic computational task into multiple parts and to execute only one part each time it is called by the scheduler. The design decision to use an application level scheduler has a structural implication, the introduction of a scheduler component, but also several rules and constraints. Two important rules are (1) every component that wants to be active needs to register at the scheduler when it is created and (2) the component needs to implement a function or method with a specific name that the scheduler will call. Finally, there is a constraint that states that every function or method that is called periodically by the scheduler has to limit its execution time to a certain number of milliseconds in order to ensure a sufficiently high frequency of calling each component.

When the team decides to replace the application level scheduler with an open source or commercial real-time operating system, which is a very typical strategy to reduce the amount of R&D resources allocated to commodity functionality, just removing the scheduler component would not be sufficient to remove the original design decision. It would also require the functions or methods that used to be called by the scheduler are converted into parallel threads or processes. As the operating system scheduler now functions preemptively, the constraint on limiting the execution time of each active function or method is no longer needed.

This example illustrates that many architecture design decisions are not localized in nature but affect several, many or all components in the system. Consequently, removing or imposing a design decision often is quite an involved and effort demanding activity. Because of this, there are many examples of teams that got half-way in removing or replacing a design decision and that ended up with systems that are an amalgamation of old and new design decisions as well as glue to make the parts work together. Thus, as we are collecting the tasks required to transition from the current to the desired state, keeping all actions related to removing, replacing or adding a specific design decision should preferably be kept together as much as possible.

The third area of focus is concerning the dependencies within the architecture. For most systems, over time dependencies develop between different parts of the system that really should not be connected to each other. Often this is a consequence of the accumulation of architecture technical debt and the organization underinvesting in managing this technical debt. When applying a major refactoring as when adopting the 3LPM, there will be a need to clean up the additional debt as well, resulting in a certain amount of piggybacking on the initiative. In many cases, this is concerned with dependency management. So, the action is to review all dependencies between components, both current and after the relevant components have been

split up, and identify all the dependencies that violate the guidelines. For every violation, a task needs to be defined to resolve the unwanted dependency.

Although the steps are presented here as sequential, one can actually start with each one. If the team is more focused on architecture, the best starting point would likely be the architecture design decisions. However, for many engineers, design decisions are a bit of a fuzzy and vague concept and their focus is on code. In that case, starting from the components is probably the better starting point.

Milestones

The typical refactoring initiative, especially one of the type we're discussing here, where quite fundamental changes are introduced to an existing system, requires a significant amount of R&D resources of the right seniority to realize. As the refactoring effort occurs in parallel to the normal development of new functionality, delivery of products to customers, dealing with trouble reports from the field, etc. it is likely that the refactoring effort will require a significant amount of calendar time. Depending on the organization, the refactoring initiative may take several quarters or even several years. Most organization have difficulty to sustain a refactoring effort of such a long period of time. The danger is that in the middle of the refactoring, urgent events happen in the field or some critical customer demands immediate action that was not planned for, resulting in resources being taken off the refactoring effort. As refactoring is, in many ways, hard work that does not clearly deliver customer value in the short term (a major motivator for many engineers), the risk exists that refactoring efforts are abandoned in the middle of realizing an important change. In fact, it may result in the architecture ending up in a place worse than before the start of the refactoring as part of the system has been transformed and the rest of the system is still in the old structure.

In our experience driving change at companies, one of the techniques that help to avoid the problems described above is to define milestones that can be reached with a focused and sustained effort, but that take a limited amount of time. Also, the milestone, if chosen well, will provide an anchoring point where the new architectural structures as well as the associated behaviours will hold and avoid the organization to slip back.

The behavioural side of software development is important to realize. In all R&D organizations, there exists a culture where certain things are allowed and endorsed and others are frowned upon. Frequently this leads to a situation where there is a gap between the formal rules as defined by the architecture and the way developers follow or violate the rules. For instance, in some organizations, hero developers receive significant recognition for delivering complex functionality right before the deadline or fixing complicated problems in high pressure situations. At those points, all rules go out of the window as all the team wants to do is deliver the system. At this time, all kinds of dependencies between components get introduced that really should not exist according to the formal architecture. This is where architecture technical debt gets introduced that, over time, has enormously negative effects on the overall productivity of the

R&D organization and it causes even more of the resources to be spent on commodity functionality. So, in a refactoring, the concern is not just with changing the structure of the architecture and the software, but also with changing the behaviours, norms and values of the R&D organization. When adopting the 3LPM, managing dependencies over the three layers of commodity, differentiation and innovation is extremely important if we want to reap the benefits. If we allow for the wrong types of dependencies to be introduced to the system, we are unable to control the R&D resources allocated to commodity and all our efforts are for naught. Introducing milestones allows us to define anchoring points that both deliver on the restructuring and on the behavioural change that is required in the R&D organization.

It is difficult to provide generic advice to selecting the right milestones as systems are so unique, but there are a number of more general pieces of advice that we can provide. First, the number of milestones should be limited and the typical range is three to five, depending on the size of the system and the R&D organization. Second, the time period for until reaching a milestone should be limited to one or maximally two quarters. Maintaining focus and commitment in the organization for a change project without reaching a point of celebration is often difficult. Third, each milestone should spell out explicitly which structural changes as well as which behavioural changes we are aspiring to realize as well as the business benefit that is realized by it. Fourth, when reviewing the inventory of tasks, aim to start by selecting clusters of tasks that maximize the separation of commodity functionality against the lowest amount of work. The more commodity functionality can be put at the other side of an interface, the easier it is to gain insight into why R&D resources are still invested in adding and changing functionality at the “wrong” side of the interface. Finally, especially the first milestones should aim to also generate a few high-visibility outcomes that are intended to build the confidence in the organization that this initiative, that is stealing resources from customer projects left and right that could have resulted in real, short term value, is indeed delivering on the expected benefits. For all that we’re rational engineers, the fact is that humans are storytelling machines and the milestone planning needs to help everyone involved to tell an easy to understand and compelling story.

Once we have defined the milestones, the next step is to allocate clusters of tasks to each milestones. As we discussed earlier, tasks are not independent of each other. Some require certain other tasks to have already been completed or will at least require much lower effort if done later in the sequence. This is why we use the notion of clusters of tasks, even though there are a number of types. One type of cluster contains tasks that either gravitate towards the same logical part of the system. As all these tasks affect the same component or set of components, it is often easier to find synergy between different tasks and in that way lower the overall R&D effort. The second type of cluster is concerned with adding or removing the implications of a cross-cutting architecture design decision. Although some design decisions mostly have a local effect, many introduce rules and constraints that require many or all components in the system to either support certain interfaces or protocols, behave in certain ways, etc. The second type of cluster combines all the tasks associated with removing or adding such an architecture decision with the intent of increasing efficiency by having to repeat the same set of actions repeatedly for several components. A third type of cluster often is

concerned with creating the interface between differentiating and commodity functionality or between innovative and differentiating functionality. The design of these interfaces is hard as we on the one hand would like to have one clear and well defined interface and on the other hand, we have several vertical slices of functionality that falls in two or three categories and each of the vertical slices needs its own interface. Finally, depending on the type of system there might be other types of clusters.

The intent of this step is to identify the three to five milestones that make up the set of actions required for adopting the 3LPM and properly separating commodity, differentiating and innovative functionality. Based on the milestones, we assign the clusters of tasks that we have identified to the relevant milestone. Once we have completed this activity, we can move to the next step which is concerned with the detailed planning of the tasks for the first milestone.

Detailed Plan

Once the set of tasks allocated to the first milestone is clear and agreed upon, we can start the detailed planning. As indicated earlier, the time period for executing towards a milestone should be one to two quarters as it is hard for an organization to maintain its focus for a longer period of time without achieving and celebrating some success. This means that a milestone will take 12 to 24 weeks which means, assuming 3 week sprints, 4 to 8 sprints.

Once the number of sprints is clear, we need to identify the teams that will be involved in working towards the milestone. Here, again, we have some choices to make. Either we select one or more teams to work with refactoring full time or we distribute the refactoring tasks over the backlogs of the different teams. If the organization uses component teams, obviously the refactoring tasks need to be allocated to the relevant teams. If the organization uses feature teams, it indeed is a choice.

One factor to keep in mind is that during the time of refactoring and executing the set of tasks associated with the milestone, there will also be new development ongoing. The refactoring tasks obviously do not take place in a vacuum. So part of the choice of full time refactoring teams or distributing the load depends on the potential synergies that can be achieved by combining refactoring and new development activities. If these are significant, distributing the refactoring work over the teams and pairing it up with related new features can be very effective. The risk of course is that teams fail to prioritize refactoring tasks in their backlog and focus on new development. The consequence of that would be that the schedule starts slipping as tasks get pushed to later springs all the time.

Based on the above, each task in the set needs to be allocated to a sprint. Once the tasks have been sequenced over the sprints, the tasks need to be allocated to teams. Although this is obvious, no plan survives contact with reality so there will be a need for constant adjustment and change. This includes leaving space at the end of the milestone period for tasks that slipped or work that turned out to be required but that was earlier not identified.

Summarizing

In this part of the book, we describe how to refactor the software based on the Three Layer Product Model. There are several reasons why companies are willing to undergo this effort. For instance, without clear architectural separation, it will be difficult for R&D organizations to systematically avoid investing in commodity. Also, once the interfaces between commodity, differentiating and innovative functionality have been defined, sets of teams can be allocated to each layer, allowing for a much easier controlling of resources and each set of teams knows what metric it is optimizing for.

The first step in the process of refactoring the software to comply with the 3LPM architectural style is the definition of the desired state. This requires the allocation of system elements to the three layers, the careful review of design decisions to decide which need to be added and which need to be removed or replaced, the definition of the interfaces between the different layers and a high level decision on the desired resource allocation to each of the levels. Especially for commodity functionality it is hard to decide what a reasonable level of resource allocation is. So, as a rule of thumb we recommend to cut resources to the commodity layer at least to 50% of the original level.

Once the desired state is clear, the next step is to identify all the tasks that are to be conducted, identify technical dependencies between the tasks (some things have to be done before other things) and prioritize the tasks. These tasks typically have the nature of refactoring efforts, such as changing the component structure, removing dependencies and introducing interfaces.

However, there will be several other change activities that may include changes to the way the company sells its projects (e.g. restricting customers from requesting changes in the commodity layer), shifts in what the company does in-house and where it relies on its ecosystem partners (to be discussed later in the book), new processes surrounding R&D but involving other business functions to improve the alignment with business strategy and tactical opportunities as well as organizational changes, such as allocating a team to each layer of the 3LPM. The plan of course not only lists the set of required changes and tasks, but also provides insight into the required time and resources and provides a justification in terms of the expected observable benefits. Finally, each issue that needs to be refactored but that is the source of technical debt rather than a direct consequence of transitioning to 3LPM is the consequence of inappropriate norms and values in the organization. So, in addition to changing the structure through refactoring, part of the tasks need to be concerned with changing the norms, values, attitudes and behaviours of the engineers in the R&D organization.

We recommend that the change effort is organized according to a number of milestones (3-5) and that achieving each milestone takes 1-2 quarters or, for 3 week sprints, 4-8 sprints. The goal is for each milestone to reach a point where the organization can celebrate and anchor

what has been achieved so that we don't slide back into bad behaviours and consequently bad architecture and bad code.

Once the milestones are clear, we conduct detailed planning for the first milestone and, later, on, detailed planning for each subsequent milestone once the previous one is being finalized. The execution of the plan requires tracking of progress according to the plan, but also tracking of the realization of the expected benefits. For instance, an architecture refactoring effort may be prioritized as the expectation is that the lead time for new features will be significantly reduced. If the lead time does not decrease as the refactoring progresses, then this needs to be investigated, potentially leading to changes to the plan.

Finally, once the refactoring according to 3LPM is reaching the later stages, one can consider to reorganize the R&D organization and to allocate sets of teams to each layer. Although this is not always suitable, it does provide effective ways to control resource allocation (the backlog for the teams just grows) and it ensures that each team knows what business metric it is optimizing for.

In the next part, we discuss the implications of recursively applying the 3LPM and the use of platforms as a mechanism to reap even more benefits of using the 3LPM.

Part III: Towards Platforms

The notion of layering functionality according to the 3LPM principles is of course analogous to the notion of software platforms and software product lines. As we have discussed the 3LPM approach earlier in the book, the basic principle is separating the commodity layer and differentiating functionality layers from each other even though these are maintained and evolved within the same R&D organization. However, there are many situations where the functionality in commodity layer is useful for more than one system and R&D team. That requires an even further separation of the commodity layer from the differentiating functionality layer. In addition, it requires improvement of the stability of the interface between the two. In fact, in this case the commodity functionality layer becomes a platform for use inside the organization.

Of course, the concept of platforms is far from a new idea. In fact is has been around half a century in software and even longer in other technology fields. Under the overall heading of software reuse, for decades the software engineering community has searched for the holy grail of reusing software with a low associated cost. Starting with reusable functions and then modules, followed by objects and frameworks, service oriented architectures and more recently microservices, we have seen a wide variety of more or less successful attempts at effective reuse of software assets.

A software product line or platform approach is concerned with separating domain engineering from application engineering, which in our case translates to separating the commodity layer

and differentiating functionality layer and assigning these to different R&D organizations residing in the same company. One can go one step further, towards the adoption of software ecosystems where the commodity, differentiation and innovation layers are separated and the R&D efforts occur in different organizations altogether. However, this is the focus of part IV. In this part, we focus on applying the 3LPM to an intra-organizational scope.

One of the reasons that software product lines proved to be so successful in many companies is because they included more important dimensions of running a software-intensive company. Whereas other approaches predominantly focus on technology, software product lines address all four dimensions of the BAPO model. As shown in the figure below, the BAPO model starts from the business and business strategy. From this, the system architecture and technology choices need to be derived. The architecture of the system is then, in turn, used as a basis for define the processes, ways of working and tooling employed by the company. Finally, from this, the organizational structure and setup in terms of roles and responsibilities is derived.

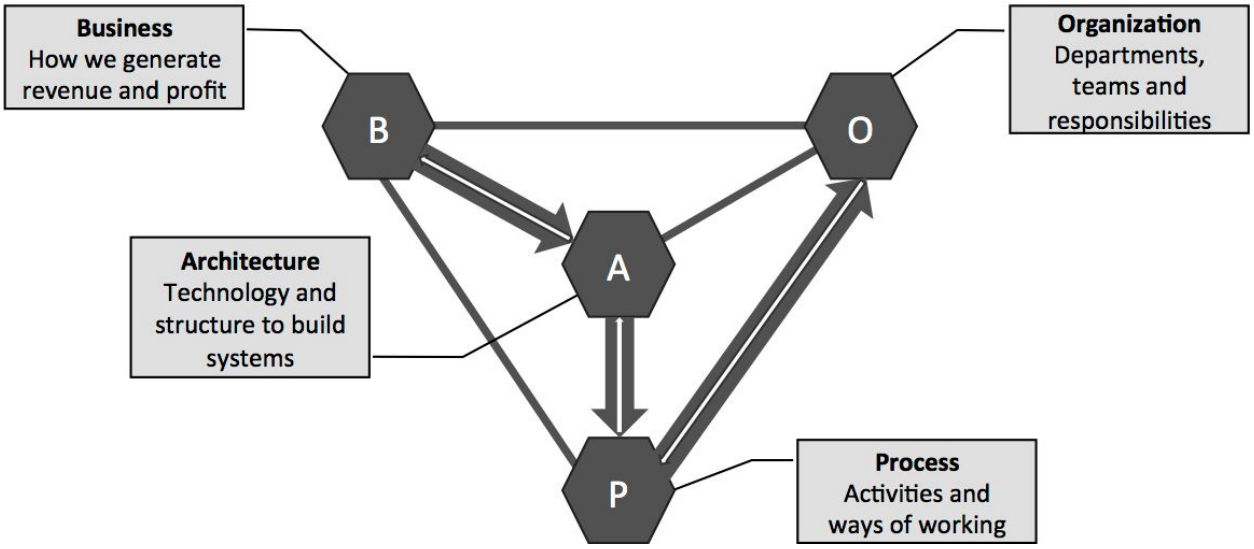

Figure X. The BAPO model

Although it of course seldomly, if ever, happens that an organization starts from a pure greenfield approach, applying the model to establish desired state for the organization can be incredibly powerful as it allows one to establish what an organization perfectly aligned with the current or potentially new business strategy would need to look like. This can be applied to existing organizations as well. One of the benefits of applying the BAPO model in these cases is that it forces one to start from the business strategy and work your way through the implications. Without the use of the model, many companies start from the existing organization and focus on the minimal changes required in response to the forces that required them to revisit the organization.

In the figure, it is clear that the main arrows flow from “B” to “O”. However, each arrow contains a smaller arrow pointing backwards. This smaller arrow indicates that the existing organization,

processes, architectures and business strategy feed back into the model. When applying the BAPO model to an existing setup, the BAPO model should focus on the delta between desired and current state that is providing tangible and convincing business value and avoid proposing changes that would perhaps make sense in a greenfield approach but where the cost of changing an existing organization outweigh the accomplished benefits.

The overall benefit of the BAPO model is that it forces organizations and their leaders to take a holistic approach and to avoid focusing on just one dimension. The second advantage is that it clearly shows that the organizational setup is the last dimension to address, after the other dimensions have been sorted out. In many reorganizations, the exact opposite happens: some senior manager decides on a reorg, proposes the new structure and appoints responsibles who then run around trying to figure out the process, ways of working and architectural implications. Very little attention is paid to the business and business strategy as the reorg is viewed as an internal event that has no implications outside the company. Of course, this is a fallacy as the boundary between the company and its ecosystem has much higher permeability than what many naively believe.

In this third part of the book, we apply the 3LPM to an entire product line and perhaps a complete business unit or company, rather than to an individual system or product. As the size of the R&D organization and the need for decoupling is much larger, we need to introduce the notion of cascading 3LPMs. In that case, we have multiple 3LPMs and the commodity layer for a product matches with the differentiation layer of the 3LPM of the platform that it is built on.

The notion of cascading 3LPMs has parallels to the concept of software product lines. However, there are material differences between the two approaches. In order to highlight those, we first present a brief introduction into software product lines and then introduce the concept of cascading 3LPMs and discuss the differences between the concepts. Subsequently we dive into applying the 3LPM model at the company level and for establishing platforms.

Software Product Lines

Software product lines (SPLs) emerged in the 1990s as an approach to software reuse that mimicked mechanical product lines. The approach separates between domain engineering and application engineering. Domain engineering is concerned with the creation and evolution of software assets that can be reused by multiple applications (or products) whereas application engineering is concerned with the derivation of applications (products) from these reusable assets.

The intent is that the reusable assets are reused by multiple products, in order to amortize the cost of the asset, and to minimize the cost of reuse. As the products reusing the shared asset are different, their demands on the shared assets are different too and consequently, domain engineering has to focus on providing variation points where the behaviour of reusable assets can be adjusted to the specific needs of each product. The cost of reuse can be reduced by

offering a set of variation points that is as small as possible for the required diversity in behaviour and yet sufficiently expressive to meet all the needs.

Variation points are realized through the use of variability mechanisms. There is a variety of mechanisms available and a deep dive into these goes beyond the scope of this short book. However, mechanisms vary from selecting a variant out of a list of alternatives to extension points where the product (application) engineering team can write code to extend the behaviour of the domain asset with product specific functionality.

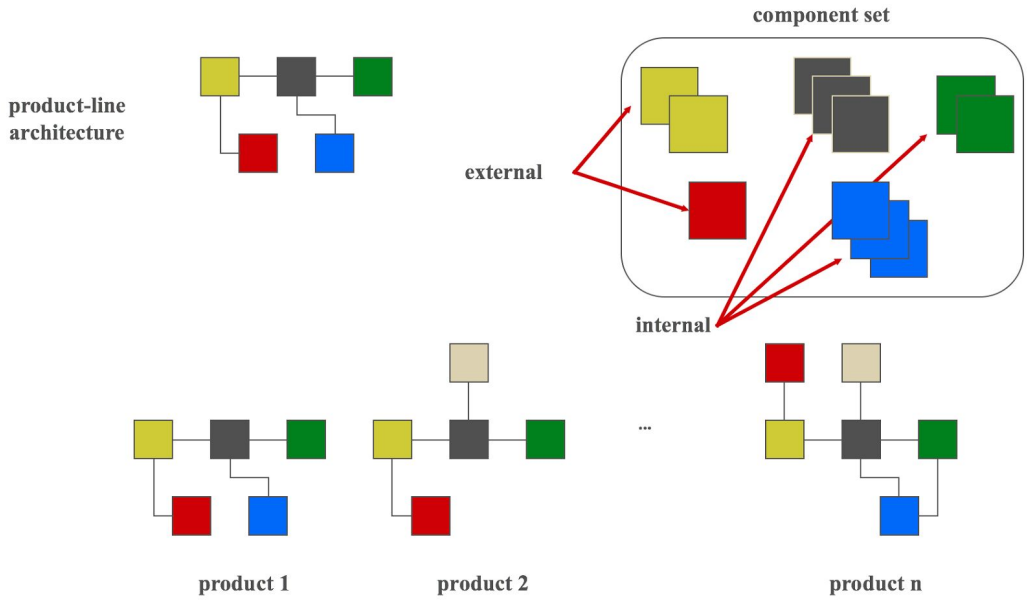

Figure <X>: Simplified overview of a software product line

As shown in the figure above, a software product line consists of a product line architecture, a set of reusable components and a set of products that can be derived from the architecture and reusable components. The set of reusable components can have multiple realizations for the same component in the product-line architecture. Also, these components can have been developed internally or be sourced from outside the company. Finally, each product can either follow the product line architecture, deviate from it slightly or deviate from it significantly.

SPLs have proven to be very successful in many industrial deployments and have allowed companies to achieve business goals such as increased product portfolio diversity, common user experience across their product range or significantly reduced R&D spending. In fact, SPLs can be viewed as the first truly successful approach to intra-organizational reuse. One of the reasons is that SPLs address the most important dimensions for a software intensive company, as defined by the BAPO model.

As there is no silver bullet, also SPLs have their set of challenges. When not being vigilant, the complexity of operating in an SPL context can easily get very high since teams that earlier were

not dependent on each other now do become dependent. That can easily result in a web of dependencies that slows everyone down due to very high coordination cost. For instance, although it conceptually may seem logical to reuse a platform as a product team, the product team does add an important external dependency to their system. If the platform team frequently breaks the interface, either syntactically or semantically, the product team will experience significant overhead and see their system fail or break regularly, but entirely unpredictably, and through no fault of their own. Consequently, it is clear that the organization needs a significant level of maturity in its software engineering capability in order to reap the benefits of software product lines.

One main challenge only partially addressed in SPLs is how one decides what to put in the platform and what should go in the products. The basic rule in SPLs is that functionality that is applicable to multiple products should be in the platform and functionality useful for one product should be in the product specific code. However, in practice the line is not as clear cut. For instance, the functionality used by some but not all products is sometimes better off in the product code and sometimes in the platform. Similarly, even code used by multiple products but under a high degree of evolution may still be better off in the product code as it will slow down the product teams to wait for the platform team to do its job.

The 3LPM model provides a mechanism to decide on this in an effective fashion in alignment with the business strategy: if functionality is commodity, it should be in the platform. Otherwise, I should be in the product specific code. The interesting dilemma or concern that is raised in this context refers to efficiency: even if differentiating functionality has significant commonality between products, should it really still only sit in the product code? Does it make sense to not exploit the efficiency that could be reached by sharing the differentiating functionality between products? Our experience is that although there are exceptions, in almost all cases differentiating functionality will differ between products. The needs of customers in different product categories tend to be slightly or significantly different even if it looks similar or even identical at a high level. And, finally, that the goal of the teams should be to maximize the customer value for differentiating functionality and this typically requires deep customer understanding and the creation of several variants and alternative realizations of parts of the functionality as a mechanism for experimenting with maximizing customer value.

The second reason for being careful with bringing differentiating functionality into the platform code is speed: software that is in full control of a product team will simply be easier to change, evolve and deploy than software that lives in the platform and where interaction and alignment between the platform and product teams. Most organizations recognize the importance of efficiency in software development and prioritize for it, but fail to identify the costs associated with the focus on efficiency, such as reduced speed of development. Especially in fast moving markets, organizations need to prioritize speed over efficiency.

Concluding, although software product lines present the first software reuse concept that achieved significant success in industry, the 3LPM provides an additional perspective that helps

organizations balance efficiency and speed for the areas where it matters. Identifying where it matters requires the inclusion of business strategy, product management and customer understanding. These areas have traditionally been outside the domain of R&D, but achieving the benefits promised by 3LPM requires the cross-functional alignment around these topics. In the next section, we introduce the notion of cascading 3LPMs as a mechanism to realize effective reuse.

Cascading 3LPMs

In the first parts of this short book, we have use the 3LPM for a single product and R&D organization. In the case of platforms, we are entering a situation where there are multiple software assets and R&D organizations. At this point, we need to recognize that each asset has its own 3LPM and the challenge in this situation is to align the 3LPMs of multiple software assets. In addition, the R&D teams responsible for these assets need to be aligned in order to ensure that the alignment between assets takes place in the way that is intended.

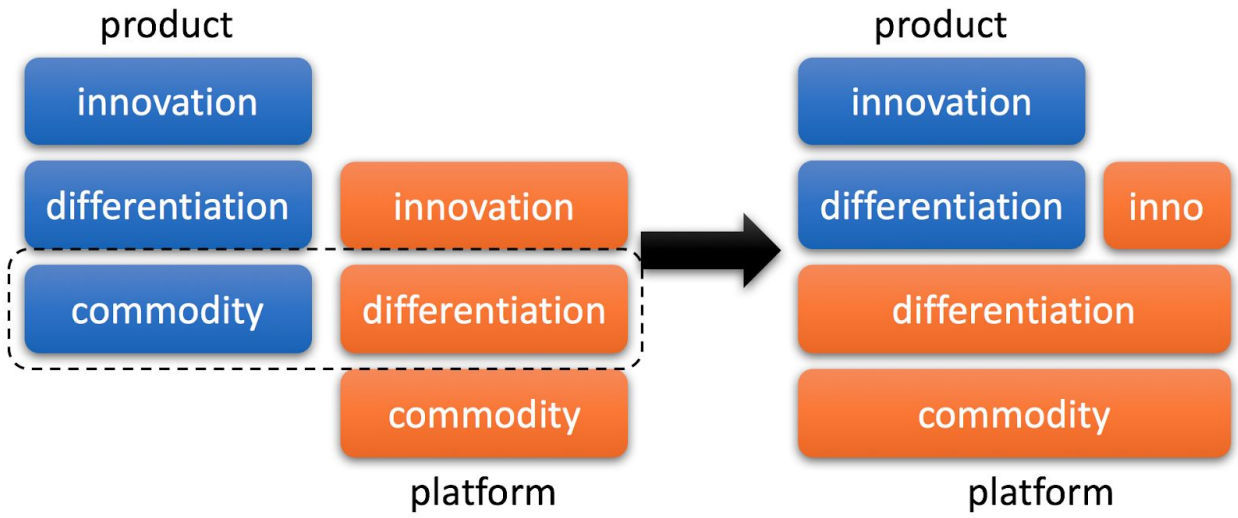

Figure X: Illustrating relationship between product and platform

In the figure above, we illustrate the relationship between a product and a platform. As the figure illustrates, the commodity layer of the product is aligned with the differentiation layer of the platform. The intent is, of course, for the platform to provide the commodity layer of the product so that the product R&D team can focus its energy on innovation and differentiation of the product. The platform team maintains a smaller innovation activity, which is concerned with preparing the platform for incorporating new functionality that currently is differentiating in the product, but that in the foreseeable future will commoditize and flow into the platform.

As there typically are multiple products that are built on the same platform, the challenge for the platform team is to align the inflow of new functionality from the different products in order to harmonize the different realizations of similar functionality in such a way that the platform only

holds one instance. One of the hardest challenges is to convince product teams to change their interface towards commoditizing functionality just so the platform organization can ensure that only one implementation of certain functionality exists. As this change in which the functionality is used typically adds zero business value, it is very hard to prioritize for product teams. And yet, if we don't the overall R&D investment in commodity functionality increases again.

In fact, if all products on top of a platform do not adhere to one interface, it will be impossible to over time replace the internal, proprietary implementation of functionality with one that is developed outside the company, either commercially or in open-source. And, in the end, that is the only way to remove all investment in maintenance of commodity functionality, short of stopping support for the functionality altogether.

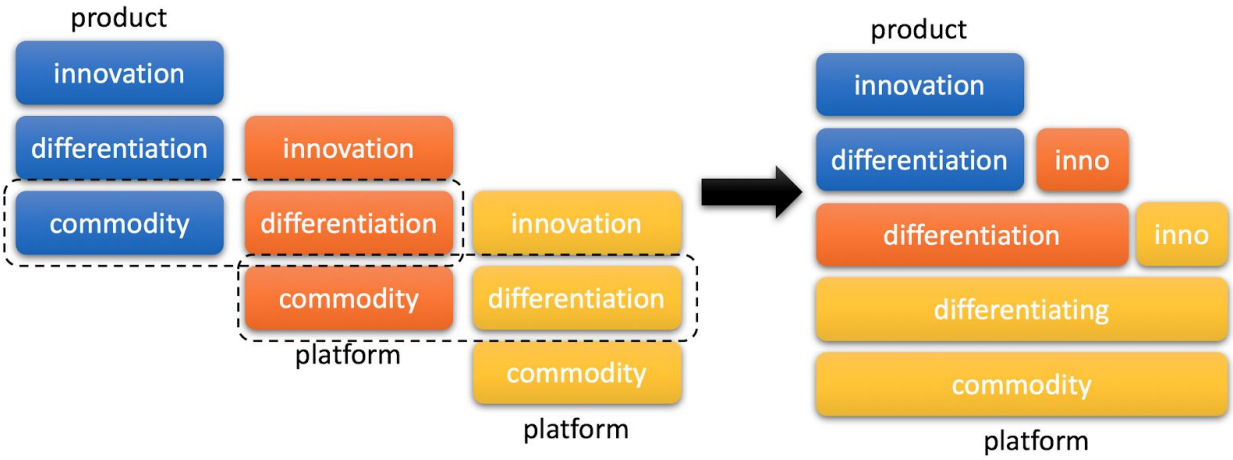

Figure <X>: Cascading 3LPMs

We refer to this model as cascading 3LPMs as the company doesn't need to stop with a single platform. As shown in the figure above, a software asset used at one level can again be built on top of another, more generic platform. Although the figure only shows two platforms, the pattern in itself is of course recursive and can be repeated many times. Also, it of course extends into software assets that have been developed outside the organization, but then we enter the ecosystem domain which is the subject of the next part of the book.

The final perspective that we need to discuss in this section is illustrated in the figure below. In most organizations, the notion of a platform is presented as a single entity where all commoditizing functionality flows into. However, there are alternative approaches that one can employ. In the figure below, the product software is organized into four subsystems that each have their own 3LPM. Although not for all companies, a subsystem centric structure may simplify the reuse of individual, domain-specific platforms in other products. Also, it often simplifies replacing internally developed functionality with open-source alternatives or putting internally developed software in open-source to share the cost of maintenance with the community.

Figure <X>: Subsystem specific 3LPMs

Concluding, the main purpose of this section was to indicate that the 3LPM can be applied recursively, resulting in cascading 3LPMs. The commodity layer in the first 3LPM becomes the differentiating layer in the next 3LPM and so forth. As a second purpose, it is important to realize that although it's easy to think of 3LPM as the model for platforms, it can also be applied in other ways, such as a subsystem specific or even component centric 3LPMs. Although we do not provide much detail about the detailed and precise implementation of this concept, based on experience, I can share that conceptually it is quite easy to understand and to translate to a concrete realization in most organizations.

Using 3LPM For Platforms

The 3LPM approach is of course quite similar software product lines and it has several parallels. However, there are a number of differences too in that the 3LPM is both more narrow and more broad than software product lines. The main differences are threefold. First, the 3LPM focuses more on the "B" and the "A" in the BAPO model whereas software product lines focus, by and large, more on the "P" and the "O". Second, the 3LPM insists on real interfaces between the three layers whereas software product lines often have few constraints on the architecture. Third, and perhaps most important, the 3LPM enforces a discussion in the organization concerning differentiation and commodity. This discussion in itself is immensely powerful as it forces the organization to align itself around its key differentiators and treat everything else needing to be optimized for minimal cost of ownership. Below, we discuss each of these in more detail.

Software product lines stress the difference between platform engineering and product (or application) engineering. As a consequence, the focus in software product lines tends to be on

the process of development and alignment between product and platform engineering and on the organizational setup for this way of working. There will be a platform engineering team that aligns its development activities with the product or application teams with the intent of building the functionality required by product teams only once while allowing the product teams to focus on their specific applications and products.

The 3LPM focuses on the business strategy of the organization and uses that to determine the boundary between differentiating functionality and commodity functionality. As we will discuss below, this is a really important point as it forces the company to make a choice about its differentiation and strategy and, in turn, to force the R&D organization to align itself around these choices. Once we get to the point of clarity and commitment around differentiation and commodity, the basic architecture can easily be derived from the business strategy as the functionality immediately can be allocated to the three layers. Once the functionality has been assigned to layers, interfaces need to be designed to allow for stability at the top level while allowing for autonomy of teams within the layers.

From a process and organizational perspective, the basic principle behind 3LPM is to empower teams by decoupling their work from other teams. This is important for many reasons. The first is of course that teams that can conduct their work within a sprint without having to coordinate with other teams will have higher productivity and output when compared to teams that have to align, coordinate and meet with others on a continuous basis. The second reason is that teams in different layers have very different goals. Teams working on commodity are focused on reducing the total cost of ownership of functionality in that layer. There are many strategies for accomplishing this, but the typical steps include centralizing and standardizing. Centralizing this case means bringing together all the versions, variants and alternatives and to assign this functionality to a single team. The next step then can be to standardize, which means reducing the alternatives to, preferably, a single version or at least as few as possible in order to reduce maintenance cost. Assuming that commercial or open-source versions of part of the software are available, the functionality should be modularized with the intent of replacing bespoke functionality with external software. The goal of teams working in commodity should be to get rid of internally developed software where possible and to minimize the cost of maintenance on the remaining internally developed and maintained software. On the other hand, teams working on differentiation do the exact opposite: they work to maximize the value of differentiating functionality to customers. This often required the introduction of variants, alternative workflows and customization with the intent of increasing customer satisfaction. As the goals and priorities of these teams are so different, we should aim to minimize the dependencies and connections between them as much as possible.

Experience shows that for the aforementioned to work well, it requires that the organization as a whole has a very good understanding of the classification of functionality and mechanisms to identify and decide when functionality flows from differentiation to commodity. Where feasible, the organization should have data driven mechanisms to determine this, rather than relying on

human interpretation. The reason for this is that experience shows that companies view functionality to be differentiating far longer than their customers.

Organizing for Cascading 3LPMs

This part of the book is focused on cascading 3LPMs and the underlying assumption in the text has been that these 3LPMs are maintained and evolved by different organizational units. The reason to break R&D efforts into different organizational units is because of size, too many people working on a single piece of software, but more typically because some of these 3LPMs serve as platforms for multiple products or other software assets and consequently require an organization delivering the asset to multiple customers. So, in the figures that we showed earlier in this chapter, it is typical that each of the 3LPMs belongs to a separate organizational unit.

Although the concept of cascading 3LPMs is conceptually easy to understand, actually realizing this in an organization is not trivial and requires some elaboration. Successfully employing the concept of 3LPM at the organizational level, requires efforts at multiple levels. The three levels that need to be considered as a minimum are the organizational level, the 3LPM level and the layer level. Let us consider a set of three cascading 3LPMs with two 3LPMs at the top level (for instance two product teams building building on the 2nd level 3LPM and two platforms building on the base 3LPM. In the figure below, we visualize this setup.

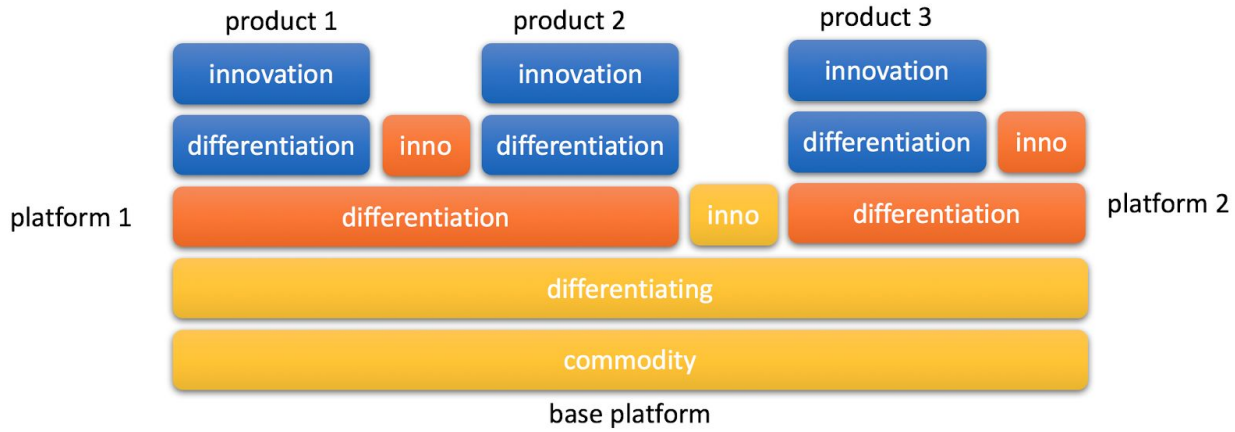

Figure X. An example cascading 3LPM setup

From the figure, it is obvious that there are three scopes of operating in this structure. First, there is a need to explore the scope where all 3LPMs are considered. One could refer to this as the portfolio level. The next level is to treat each 3LPM as its own entity. At this level, the main concerns are to coordinate between the different layers and to align with the other layers that are in parallel to the current layer. Finally, there is the level of the individual layer where there also is need to coordinate with the other layers as well as internal work.

Although the specifics of the work on each scope has a different target, the activities are quite similar for each scope. The main activities are concerned with roadmap and backlog management, interface management, variant management, coordinating upstream work and resource allocation. Below, we discuss each of these activities in more detail.

Example: At WKI, it has become clear that its electronics board and hardware abstraction layer are commodity. At the same time, however, for a variety of reasons, including intellectual property, the company is unable to outsource this development. After extensive deliberation, the company decides to start a small development site in Vietnam that will be responsible for these two components.

The decision has three major implications. First, even considering the communication overhead, time zone difference and other inefficiencies, the estimated cost savings will exceed 50% if not more. Second, although the architects at WKI talk about decoupling and interfaces continuously, the culture in the company is that dependencies, hidden or public, appear all the time as developers, often with the blessing of their peers, introduce shortcuts to gain short-term speed, but at the expense of introducing additional technical debt. With the hardware team no longer co-located with the software R&D, the company finally needs to take interface management serious, at least between the hardware abstraction layer and the rest of the software.

The final implication is the signalling function that this decision sends throughout the company. All this talk about commodity, reducing R&D investment into commodity, etc. suddenly becomes real when people jobs, positions, team structure, etc. change. Everyone understands that the company is serious about these changes causing a transformation in how people talk about value, return on investment of R&D activities, etc.

Roadmap and Backlog Management

The first activity is roadmap and backlog management. At each level, the responsible team or organization needs to have an roadmap and a backlog to plan and prioritize work. The roadmap should be driven by strategic needs of the company as a whole, but the focus of each area is driven by the scope addressed.

For the portfolio level, the roadmap and backlog are concerned with the work that realizes the business strategy for the company. Overall, the business strategy may be concerned with introducing new products, extending existing products with features as well as optimizing features already deployed in the field. As we are looking to, on the one hand, offer long(er) term commitments and, on the other hand, want to ensure business agility, the roadmap work needs to be organized such that items with specific deadlines associated with them should not constitute more than 30% (or max 50%) of the available R&D resources.

As we are looking to work in an agile fashion, we need to derive a backlog for the portfolio level that can be used by the products and 3LPMs to create their roadmaps and backlogs. The

difference between the roadmap and the backlog, however, is that the roadmap is proactively looking into the future and puts stakes in the ground with respect to major milestones for the company. The backlog is filled using a pull based approach where the organization limits the amount of work on the backlog to a period limited to one or two sprints.

Interface Management

Independent of the level at which one sets the focus, over time functionality will flow over layer boundaries. This may be entirely new functionality for a platform that is added after commoditizing at the product level or it can be differentiating functionality that now transitions in the commodity layer. As functionality moves between layers, there is a need to adjust interfaces as functionality that could be called directly as it sat in the same layer now needs to be accessed through an inter-layer interface.

Although the introduction of interfaces may seem like a hassle, the whole point of software engineering is to find appropriate points of decoupling in order to manage complexity. Functionality that is commoditizing should be decoupled from differentiating functionality as it provides the most effective mechanism to ensure proper resource allocation. If the layers also are assigned to different organizational units, the presence of an interface is even more important. Teams and organizational units can coordinate through architecture or through process. Process means meetings and other time wasting human-driven coordination processes. Coordinating through architecture means that each team or organizational unit can operate independently at its side of the interface.

As functionality moves, the interface between layers needs to evolve to provide access to the new functionality. As this evolution takes place, it may also offer the opportunity to simplify and reduce the size of the interface by removing or reducing access to even older functionality. This means that the new interface may introduce binary breaks as compared to the old interface. In order to avoid deep coordination between the teams at each side of the interface, the recommendation is to use the typical interface management mechanisms as, for instance, found in the Java community. This means that when introducing a new interface, the old interface is maintained for a while but marked as deprecated. During the time that the interface is deprecated, but available the teams relying on the interface can change their code to evolve to the new interface. Once all teams have moved on, typically after one or a few sprints, the old interface can be removed together the associated, now obsolete, code.

The responsibility for the interface is with the team or organization that is providing it. However, the architects have a responsibility to ask for advice from the teams using the interface and to inform these teams of their decisions and the rationale behind these decisions. As is typical in empowered organizations, asking for advice is obligatory, but following the advice is not. This means that the architects can decide to ignore some of the advice that they have received.

Finally, there often is a tendency to only add to the interface and to never remove anything. There always is some team that still uses some old functionality and it's easier for the team maintaining the interface and the software in the layer below it to not change anything that used to work. Over time, however, this creates an increasingly complicated and large software asset and the very reason for introducing the 3LPM gets violated inside the layer itself. It is the responsibility of the team to constantly seek to narrow the interface that it offers to others. Interfaces offer one of the most powerful decoupling mechanisms and should be used accordingly.

Variant Management

The third area that requires focus is managing the variants present for functionality in the system. During innovation, the focus is on minimal viable (or loveable) features and functionality to confirm customer relevance against the lowest amount of effort from the company. However, once it is confirmed that some new functionality resonates with customers, it enters the differentiation layer and the focus shifts to maximizing the amount of value that can be delivered to customers. This frequently includes the introduction of variants and alternatives for specific customer segments and occasionally even specific customers. Once functionality starts to commoditize, though, the aim should be to reduce the amount of variants and bring this down to the smallest possible number (preferably only one).

Although this is simple to describe conceptually, experience shows that companies often experience significant difficulty in actually realizing the reduction of variants. The consequence is vast numbers of non-value adding variants present in the commodity layers of the software stack. The problem of this is not just complexity, but also the amount of alternative configurations, the increased likelihood of faults and errors found post-deployment because of the company's inability to test all configurations out in the field and, consequently, the constantly increasing need for R&D resources for commodity functionality.

The only solution to this challenge is to reduce the number of available variants once functionality starts to commoditize. This often involves difficult discussions with other teams and organizational units as well as the customers of the company. However, shying away from this hard work will result in a much worse situation. Ideally, as functionality flows down through the product and platform layers, the number of variants should be reduced until there is only one realization of the functionality. Subsequently, the functionality should be structured and refactored with the intent of replacing internally developed functionality with open-source or commercial off-the-shelf software or, if possible, removing the functionality from the system completely.

Although the removal of variants can be conducted at all times, especially the transitioning of functionality over layer boundaries provides a particularly suitable point in time to drive this discussion. This means that it can be coordinated with interface management and the

coordination of upstream work, discussed below, in order to achieve step function improvements of R&D effectiveness.

Coordinating Upstream Work

All of the activities that we have discussed so far do not just affect the team or organizational unit managing a certain layer, but also have implications on the teams and units “upstream”, meaning the teams that are dependent on the layer in question. This upstream work can relate to changing interfaces, removed variants or changed functionality for the same area.

As we’ll discuss in the next section, each team or organizational unit has a certain amount of resources allocated to it and consequently, changes affecting this team or unit need to be coordinated as there is a limit to the absorption capacity for every organization. Although we earlier discussed the limitations of doing this coordination through processes, it is typically unavoidable to use human based decision making and prioritization and hence this work needs to be included in the roadmapping and backlog activities.

In some cases, there is a situation where all teams but one are able to transition to the new interfaces and reduced variants. In this situation, the one team can negotiate with the team providing the layer to maintain a deprecated interface for a longer period than usual so that the team has more time to complete the transformation. Although this may sometimes be necessary, it is important to note that this is akin to borrowing from the future as the team may earn a bit of efficiency in the short term, but at the expense of the platform team as well as itself as the future transition often requires more effort.

Resource Allocation

At the portfolio level, the goal of resource allocation is to maximize the overall return on investment on the R&D resources allocated by the company. This means that at this level, the team needs to decide on the best allocation of resources between products and the different 3LPMs. This, obviously, needs careful balancing but also allows for explicit management of the resource allocation between innovation, differentiation and commodity at the portfolio level. As a rule of thumb, it is desirable to allocate at least 10% on innovation and to maximize the investment in differentiation. Especially early in the adoption of 3LPM, a good rule of thumb is to seek a 50/50 division of resources between differentiation and commodity. Even if an even more aggressive shift towards differentiation is often required and even strategically necessary, the shift from a 80% investment in commodity to a 40% investment often involves a significant change process in and of itself. Achieving this milestone before proceeding to more aggressive goals is often advisable.

In a cascading 3LPM context with multiple products and 3LPMs, resource allocation depends on the specific context. However, the basic premise should be that the 3LPMs exist to optimally support the revenue generating products, services and solutions and that the majority of resources should be pushed to higher levels in the organization. There are counter-forces that

need to be considered, though. In the situation where certain types of new, differentiating functionality can be introduced for all products at once, it may be justified to introduce this in the 3LPM layer that serves all these products. Similarly, when the organization is moving towards automatically deriving products from a common platform, more effort may be required on the platform level. These situations justify a higher investment into lower layers of the software.

Part IV: Engaging the Ecosystem

The late 18th and early 19th century saw a development that was novel for society: the emergence of large organizations with thousands, tens of thousands or even more than a hundred thousand employees. Adam Smith and his book “The Wealth of Nations” is often quoted in this context as it describes how specialization by workers allows for significant increases in productivity, which in turn allows large companies to be much more efficient than the prevailing cottage industries of the time.

During the 1930s the first research started to appear that sought to explain in more detail why large companies are so much more effective and the prominent theory that is still in sway today is *transaction cost theory*. This theory attributes the emergence of large organizations to the observation that the cost of transactions within an organization is lower than the cost between organizations. So, even though there are cost within the company, the larger firm still has a significant cost advantage as compared to smaller companies. As long as the internal cost are lower than the external cost, the company will continue to grow as there is an advantage of doing so.

Over the last decades, however, there have been many developments that allow for a significant reduction in the cross-company coordination cost. Starting with phones, then fax machines followed by email and then standardization and automation of many business processes, it become increasingly cost effective to coordinate work across boundaries both inside and outside organizations. The largest effect, however, was on the cross-company transaction cost. This lead to the enormous wave of outsourcing that we’ve seen since the 1980s.

Initially, partners were selected through an elaborate, laborious process and once a partner had been selected to play a certain role, a standardized interface and service level agreements (SLAs) were defined and the partner would maintain its position for a long time. This worked because the partner selection focus was on operational processes that were highly standardized.

Now we see that companies increasingly focus on partnering around R&D and innovation. As these processes are much less standardized and predictable, the interface between companies and their ecosystem partners needs to shift as well. The transactional model where partners get paid for each delivered service or item within agreed SLA levels needs to be replaced with

alternative business models. These models can range from cost-based approaches where the company pays for the actual cost of staff at a partner with an overhead margin on top to revenue share models where the partner takes as much business risk as the company.

Although traditional partner selection was slow, laborious and long-term, we see an alternative model developing where organizations operate in a network of partners. For each project or task, a constellation of these partners is put together that is uniquely suited to perform the activity optimally, be it from a cost, quality or user satisfaction perspective. The role of the network is to define a set of rules that allows for very efficient agreement on terms and conditions. This drives down the cost of “transacting” in the network, allowing groups of smaller companies to operate on par and over time significantly more efficient than large companies.

This brings us to the notion of business ecosystems. These partner networks can, of course, be described much more accurately described as business ecosystems. In 1993, Moore [Moore 93] defined a business ecosystem as an “economic community supported by a foundation of interacting organizations and individuals, which can also be perceived as organisms of the business world”. This ecosystem has three main characteristics. First, there is a symbiotic relationship between the partners. This means that the partners should all benefit from being part of the ecosystem and be better off than if they would be outside it. Second, the members of the ecosystem co-evolve, meaning that when one partner adds to its capabilities, others evolve to make use of these capabilities. The constant evolution by all partners and the dynamic adjustment by all partners within the ecosystem allows the ecosystem as a whole to innovate faster than individual organizations. Finally, a business ecosystem is organized around a platform. As a software engineer, it’s easy to think about a software platform, such as a Linux, iOS or Android, but the type of platform depends on the industry. It can also be a set of standard services, agreed upon legal agreements or an open innovation platform orchestrated by a large company.

The Challenge of R&D Ecosystems

If we switch perspective from the macro - how does an ecosystem operate - to the micro - how should I, as a company, operate in an ecosystem, it turns out that there is surprisingly little guidance on how to organize a company’s R&D efforts in the context of an ecosystem. The business side of the company is eager to repeat the efficiency improvements of outsourcing in the context of R&D as well. Within R&D, this frequently results in decisions being taken to engage partners in R&D in a rather ad-hoc or tactical fashion. This results in less than optimal and even counter productive partnering decisions.

The purpose of this part of the book is to provide a hands-on guide and framework on strategically engaging with the ecosystems around your organization. As we discussed in the previous section, the world is moving towards networks and ecosystems as these provide significant benefits over doing everything inside the company. Although each company will work hard on hiring the best, smartest and most talented workforce, the fact remains that even for the

largest companies, to paraphrase Bill Joy, one of the founders of Sun Microsystems, more than 99.9% of all the smart people in the world do not work for your company. Engaging with the ecosystems allows the company to dynamically partner with those that are the best for each particular activity where it's own people would be second rate at best. Hence, R&D in the 21st century will to a large extent be driving by its ability to connect to the best ecosystem partners where it matters and to provide world-class R&D where work should be done internally.

Before moving towards our framework and approach, we first need to discuss how companies conduct internal R&D and what the key challenges are. As we have discussed throughout this book, the primary challenge experienced by software-intensive systems companies is that over time, the amount of R&D resources allocated to commodity functionality tends to grow continuously. This results in low investment in innovation and differentiation. A simple categorization of functionality in a software-intensive system between commodity functionality and differentiating functionality. Commodity is those functions that you need for your product to work and satisfy the needs of your customers. However, no customer will select the product because of this functionality. As an example, customers will not select a new smartphone because of its ability to make a phone call. Everyone expects that the phone will have the ability to make calls, but this is not a differentiator as all smart phones have this capability. However, customers will be extremely upset if it turns out that a smartphone is incapable of making calls.

Next to commodity functionality, the second category is differentiating functionality, which refers to capabilities that are unique and drive customer interest and buying behaviour. Differentiation can take many forms ranging from being first to market with new functionality, superior user experience, better integration between functions, etc. The important part is that differentiation is defined by customers and not by the company itself. It is customers whose buying behaviour is influenced and hence they are the yardstick to measure differentiation against. The challenge of differentiating functionality is that the moment customers show interest in unique and differentiating capabilities of a product, competitors will rapidly identify this and seek to emulate and deliver similar functionality in their products. The moment that the competition catches up, the functionality will transition to commodity. Of course, this is more of a gradual process that may occur over weeks, months or, in some industries, even years, but in the end, functionality that was once differentiating is now commodity.

The primary challenge that many software intensive systems companies face is that over time, more and more of the total size of the software is dedicated to commodity functionality and less and less of the total size is differentiating. Although it is easy to think that commodity software is static, the fact is that all software requires maintenance for a wide variety of reasons, ranging from new regulation to new hardware architectures and from new versions of operating systems to new network protocols. With the amount of commodity functionality increasing, the relative percentage of resources allocated to commodity will automatically go up as well. The obvious implication is that fewer and fewer of the R&D resources are allocated to differentiating and innovative functionality. As long as all competitors are in the same situation, this does not hurt the competitive position of the company. However, with the increasing prevalence of startup

companies in virtually any industry, the risk of disruption due to lack of innovation becomes too large to be ignored.

A second challenge, a direct consequence of the above, is that the agility and ability of the company to rapidly respond to market changes and customer requests and experiment with innovative functionality becomes increasingly constrained over time. The constantly increasing size of the software in the systems that the company offers and the decreasing percentage of R&D available for differentiation results in a double edged sword where any request for agility is hampered both by the size of the system and the lack of available resources. Thus when competitors or new entrants make surprising moves that require a rapid response from the company, it is surprisingly difficult to even realize small extensions of functionality. This is exacerbated by the tendency of especially larger organizations to organize their R&D as a factory where the lifecycle of functionality from planning, development, validation to release is measured in quarters and years rather than weeks and months.

A final challenge is that companies, unable to adjust their R&D, feel forced to take the radical step of wholesale replacing existing products and platforms with assets built from scratch. This is problematic as it, first, requires new products and platforms to reach feature parity before they can replace existing assets. Second, as the organization needs to maintain and evolve the existing assets while building their replacements, the R&D investment has a significant “peak”. Finally, building product and platform replacements from scratch has proven to be a high-risk activity with many companies failing to successfully accomplish the intended goal while wasting dozens if not hundreds of person years.

We need a significant reinvention of the way we manage software assets and allocate R&D resources to these assets. Although we focused on internal initiatives that the company can undertake in the first three parts of the book, in this part we focus on engaging the software ecosystems surrounding the organization.

Three Ecosystem Types

In this part of the book, we focus on the ways in which a company can use its ecosystem to reduce investment in commodity and to focus its own resources on what is most differentiating to customers. The first step towards accomplishing this, however, is that we need to recognize that it’s not about a single ecosystem, but rather three types of ecosystems. Each layer of 3LPM has a separate type of ecosystem associated with it. Below we discuss each type of ecosystem in more detail.

Innovation Ecosystem

At the top of the 3LPM, we have the innovation layer. There is an ecosystem associated with innovation where the company partners with others in its ecosystem to collaboratively

experiment with new functionality and ideas with the intent of delivering new, future differentiating functionality to market.

As it pertains to innovation, the collaboration around innovation often concerns customers, suppliers and 3rd party developers complementing the core offering of the company. Typically, however, competitors are left out as the main focus of innovation is to find future differentiation.

The main advantages of engaging the ecosystem for innovation are twofold. First, to reach a broader set of innovative ideas as employees of a company often experience a common view on the world whereas creativity and new insights often require alternative and even conflicting viewpoints. Second, it allows the company to share the cost of innovation with others so it won't have to carry the burden by itself. This allows the company to test and validate more ideas and concepts as compared to the situation where it would seek to do everything by itself.

There is one concern when opening up the innovation process and that is to ensure ownership or at least the ability for the company to capitalize on successful innovations. The best way to ensure this depends on the industry and context, but may include legal agreements, moving faster than others or other techniques.

Differentiation Ecosystem

The ecosystem around differentiating functionality is often more difficult to engage as the differentiating functionality gives the company its key revenue and pricing power. Consequently, organizations typically are highly restrictive in sharing differentiation with others. However, one typical situation where this is very effective, however, is where partnerships are formed with companies that provide complementing and non-competitive functionality. When such a partnership can provide a valuable offering to the customer by combining contributions from different companies, engaging the ecosystem may be particularly helpful.

Extensions to serve niches and individual customers is a second area where companies may be open to sharing differentiation with partners. Especially for companies that serve mass markets, providing dedicated solutions for niches or customization for individual customers is not feasible as the company is geared towards serving bigger markets. Using third party development partners is then an effective strategy to ensure that all customers receive a suitable solution.

The main use case, however, is when an innovation is developed collaboratively and then needs to become an important part of the differentiation layer. In this case, the company needs to find mechanisms to ensure access to and monetization of the differentiating functionality, even if a partner provides a critical part of the innovation. Different strategies are applied by companies, such as acquisition of partners or the technology, converting the partner into a supplier with strict contractual constraints, IP strategies that ensure exclusive access, etc.

Commodity Ecosystem

The final ecosystem is focused on the commodity layer. This is where we look to minimize the cost of ownership for providing the functionality that is required but does not contribute to differentiation. In this ecosystem, the focus is exclusively on reducing cost and resources that the company has to spend on the functionality and consequently the set of partners includes everyone who can contribute to this goal. This may include competitors and partners from different industries.

Our research shows that companies use a variety of different strategies to involve the ecosystem in the provisioning of commodity functionality. However, it is also clear that companies typically wait too long with classifying functionality as commodity, resulting in a potentially significant opportunity cost and reduced competitiveness.

It is important to point out, though, that even though functionality may be considered commodity for the company, other partners in the ecosystem may rightfully view this functionality as highly differentiating. Similar to the cascading 3LPMs described earlier, the same pattern can be seen over organizational boundaries. In the figure below, we illustrate how the cascading 3LPMs that we introduced in the previous part evolve into the three ecosystems that we discussed in this section.

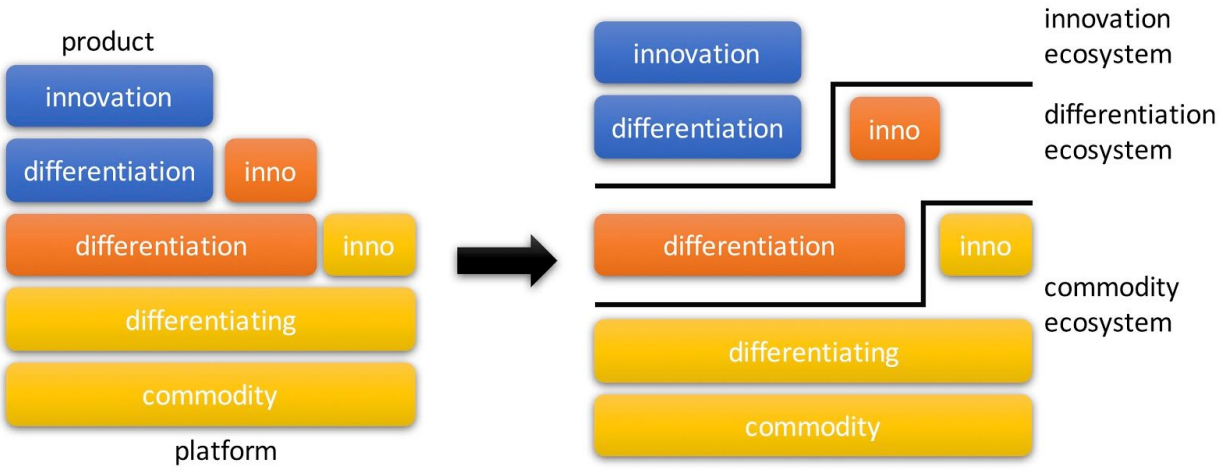

Figure X: From internal cascading 3LPMs to an ecosystem approach

Engaging the Ecosystem

In the previous section, we identified that the company is involved in not one, but at least three ecosystems. However, the way that the company engages with each ecosystem is similar in structure. The primary difference is the target of the ecosystem engagement. For the innovation ecosystem, the goal is to widen the innovation funnel and to share the cost of innovation. For

the differentiation ecosystem, the goal is to maximize the value of the differentiating functionality by providing complementing functionality provided by others. Finally, the goal of the commodity ecosystem is to minimize total cost of ownership. Independent of the specific goal, the ecosystem engagement follows a number of similar steps:

- **Define strategy**: Although the goal of the ecosystem engagement is clear, it often does not automatically translate into a clear strategy and set of actions to pursue. Hence, the first step has to be to develop a strategy that describes how the company seeks to accomplish the desired goal.
- **Assign resources**: Even though it may seem obvious, experience shows that too many companies express a desire to engage the ecosystem, but then fail to allocate resources to make it happen. This may seem an obvious mistake, but if the decision maker is unable to understand him or herself what these resources should be doing concretely while at the same time several clear priorities are under-resourced, it becomes hard to allocate the people to this task. Instead some poor individual gets a small slice of his or her time reserved for this and is expected to deliver on the strategy.
- **Engage first partners**: The first step in a new ecosystem engagement needs to be to engage the first partners in a directed, proactive fashion. In some cases, we've seen companies open up some part of their product or system by offering an API and expect the entire world to show up and use it. Often, unfortunately, what is happens is exactly nothing. The lesson is that for any company to engage its ecosystem, it will have to start by proactively reaching out to the first partners and working with them to learn what matters to partners and how to grow the engagement over time.
- **Quantitative tracking**: As it easy to get stuck in some qualitative state of "this is going great", experience shows that it is important to find mechanisms for ensuring ecosystem health without opinions or other human factors involved to color the data. That will allow us to identify concerns early, experiment with alternatives and iteratively improve engagement. The important factor here is that there has to be value exchange of some kind between the ecosystem partners. If there is no identifiable value exchanged or if the exchange is unbalanced, this rapidly results in deteriorating ecosystem health.
- **Scale engagement**: Once the first partners, who are handpicked and directly engaged with, are up and running and the metrics that we selected to track ecosystem health are successful, the next step is to increase the engagement by adding additional partners to the ecosystem. It is important to note that during the initial phases, the hands-on and direct interaction with partners may seem very similar to traditional business development engagements. There is a fundamental difference, however: for business development, the engagement with a selected partner is the end goal whereas for ecosystem engagement, it is the starting point. Our goal is to reach a state where partners join the ecosystem of their own volition and to start adding value without a direct human-based engagement with the keystone partner. This requires various techniques and ways to automate processes. The good news is that the level of automation and associated investment required develops with the size of the ecosystem and, consequently, the total value created.

In the sections below, we will discuss what the generic steps and activities outlined above translate into for each type of ecosystem.

Innovation Ecosystem

The innovation ecosystem is concerned with increasing the amount of innovation activities and broadening the innovation funnel through the use of innovation partners. Engaging ecosystem to drive innovation is not a new topic and has been around in different incarnations, such open innovation, collaborative innovation, customer-driven innovation, etc. Experience shows, however, that companies still have a strong tendency to be internally focused in their innovation activities. One of the key reasons is that most innovation resources are focused on improving product performance whereas many other types of innovation exist that often are ignored, but that are perhaps easier to conduct together with ecosystem partners.

For the innovation ecosystem, the steps to engage the ecosystem play out as follows:

- **Define strategy**: Although we already indicated that the goal of the innovation ecosystem is to broaden the funnel and share the cost of innovation, how to achieve this goal and where to start is not necessarily an easy challenge. Our experience shows that it typically is easier to start collaborative innovation activities in areas where the company does not necessarily considers itself an expert, but rather where opportunities exist to combine internal and external skills and expertise into a new area where the organization seeks to develop new business opportunities. In addition, the strategy should state explicit and quantitative goals that allow for tracking progress. The goals should initially be predominantly focused on the number of partners and only secondarily on the revenue generated. Over time, as the number of partners grows, these goals should shift towards revenue and value generated in the ecosystem.
- **Assign resources**: No innovation, internal or external, will materialize without assignment of resources to the activity. Typically, there is a choice between allocation of centralized and full-time resources or decentralized, part-time resources. Google famously allocates 20% of everyone's time to innovation activities and these activities can easily involve collaborative innovation activities. As both approaches have advantages and disadvantages, the preferred approach depends on the company. However, there typically needs to be some centralized support for partner management, agreements on intellectual property and similar topics.
- **Engage first partners**: As we are looking to get to critical mass in some area of innovation with respect to ecosystem engagement, it is important to focus the selection of first partners in a specific area. Picking one partner per innovation area will complicate the scaling of the innovation initiative over time as the company does not build the capability to manage multiple partners that may be competing with each other in this stage. It will be the job of the company to reach out to potential innovation partners, to explain the rules of engagement and to agree on practicalities. As the outcome of innovation activities, that often are highly speculative and experimental, is typically

difficult to predict, making precise agreements is hard. However, setting the ground rules is quite feasible.

- **Quantitative tracking**: During the strategy development, quantitative goals have been formulated. Although these may evolve over time, it is important to track the progress towards these goals and to do so quantitatively. Especially in the area of innovation, there easily are a lot of smoke and mirrors and people passionately pleading for specific initiatives to be continued or stopped. The focus should be on running as many innovation experiments as possible and to shut down these when the data shows that the experiment does not offer the outcomes expected. This may be hard to realize when involving innovation partners, which is why agreement on ground rules is so important.
- **Scale engagement**: Once we've found the best ways to engage innovation partners, scaling the engagement with the innovation ecosystem allows us to also scale the number of innovation experiments ongoing. The challenge is that many initiatives require someone from the company to be involved as it will be hard to capitalize on successful innovations otherwise. Of course, the most effective way to scale is by offering an API that the ecosystem can use, but again the more powerful the API, allowing for more interesting innovations, the more certification and validation of partners needs to take place in order to avoid abuse. Finally, it is important to realize that if the innovation ecosystem delivers on the desired metrics, it is OK for the scaling to demand resources. It means that the company will generate sufficient revenue from the innovations to warrant the investment.

Example: At WKI, the monitoring module offers a mechanism for adding new algorithms easily to the module. This has been important for the company as different customers have different needs, data profiles and patterns that they are looking to identify.

Although the company has mostly offered turn-key solutions to customers, over the years several customers have asked for ways to add their own algorithms to the monitoring module. Up to now, the company has declined this but instead offered consulting services to implement and add the customer-specific algorithms on behalf of the customer.

With the emergence of ecosystems, the company decides that the time has come to change strategy. The interface between the monitoring module and the analytics algorithms has been firmed up to the point that an algorithm will run in a sandbox with little opportunity to affect the rest of the system.

The company has now made the new interface available for some of their most advanced customers that are now building the first algorithms for operating in their own deployments. In addition, the company has started to discuss with some analytics companies specializing on the IoT domain to explore if there is an opportunity to create a marketplace for edge-computing analytics algorithms.

Differentiation Ecosystem

The differentiation ecosystem is concerned with maximizing the delivery of customer value. This can be achieved by internal means, as we have discussed earlier in the book, or by engaging the ecosystem. Engaging the ecosystem can be done in two ways. First, partners can engage with individual customers to maximize the value of the platform by customization and personalization. Second, we can engage with partners that offer solutions from other business domains that can be combined with our solution to deliver more value to customers through superior integration. For instance, in the financial services industry, partnerships with retailers are a proven strategy as both types of companies work with the same customers but offer very different services and are consequently not competing.

Also for the differentiation ecosystem, the same steps to engage the ecosystem are of relevance and below we discuss in more detail how these best operate in the ecosystem:

- **Define strategy**: The concept of using ecosystem partners to increase differentiation sounds really simple in theory, but realizing this in practice is a bit more involved. The first strategic question that needs to be answered is to decide what the key areas are where we seek to engage ecosystem partners. As discussed above, this can be focused on complementing solutions and services, on providing customization and personalization or on simplifying the integration of our offering in the customer's infrastructure to ensure that the full value present is realized. Although one can decide to do everything at once, it is often better to sequence the steps to ensure that each part is delivering the expected outcomes. Part of the strategy needs to be the formulation of concrete, quantitative measures to track. The primary driver is, obviously, to drive revenue from the involvement of ecosystem partners. However, we likely need early indicator metrics such as the number of partners, the activity level of partners, customer success metrics, etc.
- **Assign resources**: Defining the strategy is important as it helps us understand what internal resources to assign to engaging ecosystem partners. The challenge that some companies experience is that the team that is tasked with engaging ecosystem partners actually is in competition with these partners. For instance, the professional services team may be asked to engage with ecosystem partners to allow for easier onboarding of new customers. However, up to this point, this has been the responsibility of the professional services organization. These kinds of conflicts have to be managed carefully in order to ensure that the initiative is not torpedoed by internal strife.
- **Engage first partners**: As the differentiation ecosystem is directly concerned with generating and maximizing revenue through value delivery to customers, engaging the first partners needs to be carefully planned as we are concerned with delivering real and tangible value to customers. Failure to deliver by ecosystem partners may easily reflect badly on the company itself. Consequently, agreeing on expectations, behaviors and responsibilities as well as finding ways to quantify can help, but cultural alignment between the first partners and the company goes a long way to avoid disappointments.

- **Quantitative tracking**: As differentiating functionality is not just important for the company, but even more so for its customers, we need to track and find effective ways to quantify the ecosystem engagement, the ability of partners to deliver for customers as well as other metrics defined in the strategy. As long as the ecosystem engagement is small scale, qualitative evaluation by company staff will work reasonably well, but it fails to scale and hence we need to build these mechanisms before we start scaling.
- **Scale engagement**: Finally, once we have found a repeatable mechanism to engage ecosystem partners, the next step is to scale this engagement and to find ways to streamline and where possible automate the onboarding of ecosystem partners.

Commodity Ecosystem

The commodity ecosystem again has a different focus from the other ecosystems: providing the commodity functionality at the lowest possible cost by engaging ecosystem partners while ensuring that the functionality is still delivered at quality. We can achieve this by collaborative approaches or by full outsourcing of certain functionality.

Although the commodity ecosystem can easily be perceived as fundamentally different from the other ecosystems that we discussed so far, the generic steps to engage the ecosystem that we introduced earlier still apply:

- **Define strategy**: Especially for a company that has used a reactive and late adoption approach to reducing cost around commodity functionality, the strategy has to focus on low hanging fruit and quick wins. As shifting from internal maintenance and evolution of commodity functionality to a collaborative or even outsourced approach will require significant cultural and operational changes for the company, it is important to set quantitative targets and to ensure that these are accomplished. That will provide significant support for the change across the organization.
- **Assign resources**: Similar to the differentiating ecosystem, it is quite typical in this case that the best resources for assigning to engaging the ecosystem are the ones that used to do the work in the past. This may easily lead to forms of conflict or at least perceived conflict as the individuals may easily find themselves be obsolete. Hence, the resources assigned should be given a clear future career path, either as ecosystem coordinator or by being reassigned to differentiating functionality.
- **Engage first partners**: The first partners to engage depends on the selected strategy, but these can be existing partners, such as suppliers, competitors from the same industry, players from other industries that have similar commodity functionality or communities such as open-source software communities. The key is to start small, learn and iterate to ensure that the desired outcomes are accomplished.
- **Quantitative tracking**: Although the strategy determines what the specific targets are, one of the key metrics has to be to decrease the amount of internal R&D resources dedicated to commodity functionality. The engagement with the first partners as well as the subsequent scaling of different partner categories should be quantitatively tracked with the intent of reducing internal R&D resources concerned with commodity.

- **Scale engagement**: Scaling the commodity ecosystem is concerned with ensuring that all partners generate sufficient value from participating in the ecosystem. This means that just paying suppliers more money is not a scalable strategy, but rather that the partners have to find other monetary or non-monetary forms of value that are exchanged in order to justify active participation. Although the company may need to seed the ecosystem to reach "ignition", in the end, the ecosystem needs to reach a point of self-sustainment. In this context, the number of partners is a relevant metric, but it also has to be concerned with the right partners.

Example: The data storage component at WKI contains a proprietary database, originally built by WKI. This database is optimized for deployment in small, resource constrained environments with intermittent connectivity. For instance, whenever the database is running out of space to store data, it automatically triggers algorithms to replace the oldest data with aggregated data that captures the key characteristics of the specific time period.

Although the database is commodity, WKI has not been able to find a commercial or open-source replacement. Instead, it has reached out to some competitors and other companies in the IoT and automotive space and agreed to open-source the database in return for active development and contributions from the other parties.

Although some in the company can't understand why WKI would share code with competitors, many in the company realize that if their database becomes a successful OSS component with contributions from many developers, WKI will greatly benefit from those efforts. And not have to re-architect that part of their system to incorporate a new database while having minimal maintenance cost for the database obviously offers significant benefits.

Strategies for Ecosystem Engagement

Over the years, we have worked with a variety of companies concerning business and software ecosystems. As part of this work, we identified and collected the strategies that the companies use for engagement in the three ecosystems that we have discussed in this part of the book. We brought this together in the Three Layer Ecosystem Strategy Model [TeLESM]. In this section, we summarize these strategies

Innovation Ecosystem Strategies

Companies can employ a number of strategies to engage with the innovation ecosystem. In the list below, we present a set of innovation strategies that companies can employ. These strategies fall into three categories, internal, collaborative and external.

Me-myself-I strategy (internal): Using this strategy, the company conducts all innovation activities internally, ranging from ideation to market validation. Typically, especially traditional

companies with research labs brought often technology oriented innovations to market through the use of this strategy.

Be-my-friend strategy (internal): Here a similar approach as the previous strategy is used, but with one exception: upon identifying the most promising internal innovations, the company aims to find partners to realize and validate the concept and bring it to market. Partners in this model are treated as suppliers that perform the activities in a contracted fashion.

Customer co-creation strategy (collaborative): Typically customer-focused companies employ this strategy where they collaborate with customers to improve existing products or to create entirely new products. Together with one or a few customers a solution is created and then it is generalized for the entire customer base.

Supplier co-creation strategy (collaborative): Here, rather than customers, suppliers are engaged as innovation partners. This strategy is selected when suppliers introduce new or improved technologies that allow for new use cases to be addressed or existing use cases to be improved upon.

Peer co-creation strategy (collaborative): Especially in large companies, this strategy allows for different internal groups to collaborate on innovation. These groups can also be from different companies, but the key is that they should not have a competitive relationship.

Expert co-creation strategy (collaborative): With the emergence of Open Innovation, some organizations have developed expert networks that can be approached with innovation challenges. Experts may compete or collaborate with each other to develop a solution that meets the criteria.

Copycat strategy (external): This strategy is concerned with copying innovations that competitors have validated with customers. Although the company often will attempt to make it somewhat different from the competitor offerings, but the differences often are small.

Cherry-picking strategy (external): When many innovations are available to the company, it can employ a cherry-picking strategy where it evaluates the available innovations and then selects the most promising and successful ones for incorporation.

Orchestration strategy (external): When multiple partners from different industries need to come together to deliver an innovation, one keystone company has to orchestrate the partners and the innovation.

Supplier strategy (external): Here, external partners are selected for collaborative innovation, but the company aims to turn these partners into “supplier-like” relationships. This allows the company to exercise more control and better ways to monetize successful innovations while sharing the cost of innovation through joint development.

Preferred partner strategy (external): Using designated preferred partners allows the company to create alliances with selected external stakeholders that, over time, can become part of the differentiation ecosystem. A preferred partner may more readily sacrifice some short-term revenue or benefit for the relationship.

Acquisition strategy (external): This strategy focuses most or even all innovation resources on identifying and acquiring companies with proven innovations and integrate these. Although expensive, it frees the company the funding and management attention concerned with internal innovation initiatives, allowing it to maximize the revenue growth and profit from the existing businesses.

Although the strategies may not capture all opportunities, these provide a comprehensive overview of strategies used in industry based on research including more than a dozen companies.

Differentiation Ecosystem Strategies

Although companies use a wide variety of ecosystem strategies for innovation, we found that once functionality becomes part of differentiation, the strategies used in practice are focused internally.

Increase control strategy (internal): When new differentiating functionality is introduced, early on quite diverse and customer-specific realizations are created. At some point, however, consolidation occurs and at this point, the company incorporates the functionality in its portfolio to increase control over new functionality, and reduce complexity for customers.
Incremental change strategy (internal): As companies seek to extend the life of differentiating functionality, an effective strategy is to frequently release updates and improvements to the differentiating functionality to have the product advance in an evolutionary fashion.
Radical change strategy (internal): In cases where it proves to be difficult to continuously deploy new functionality, this strategy is concerned with distributing infrequent but significant improvements to differentiating functionality to slow down commoditization of functionality and, where possible, attract new customer segments and/or new markets.
Complementing strategy (collaborative): In this strategy, the company collaborates with partners that provide non-competing functionality to the same customer base while offering additional value resulting from the integration of these types of functionality.
Platform control strategy (external): This strategy is employed by companies that provide a platform that others generate value on, such as two-sided markets. In this case, it is the ecosystem partners that innovate and seek to differentiate themselves. The platform company only seeks to ensure that these partners are critically dependent on the platform and are unable to leave the ecosystem.

The strategies discussed here are fewer than for innovation and focused on obtaining and maintaining control of differentiation and to delay the commoditization of functionality. This is important as the main revenue drivers for companies originate in this category of functionality.

Commodity Ecosystem Strategies

Perhaps the most important set of strategies is concerned with freeing up internal resources allocated to commoditized functionality through the use of the ecosystem. Below we discuss some of the predominant strategies used in industry.

COTS adoption strategy (external): A straightforward approach is to replace internally developed commoditized functionality with commercial off-the-shelf products. To do so requires some internal activities to reduce the number of variants of the functionality, standardize the

interfaces and align these with the interfaces provided by the COTS component. Once done, the proprietary functionality can be replaced with the COTS component.
OSS integration strategy (collaborative): Bespoke functionality can also be replaced with an open-source component instead of a COTS component. This strategy is collaborative as the community expects active collaboration and contribution from the companies adopting the component. Engaging with the community has as an advantage that additional commoditizing functionality that fits the component could be integrated into the OSS component as well.
OSS creation strategy (collaborative): One, admittedly ambitious, approach is to initiate a new OSS community around commoditizing functionality for which no component exists. One strategy is to partner with competitors in the same domain in order to jointly develop a stack that all partners share in maintaining and evolving.
Partnership strategy (external): Instead of creating an open OSS community, an alternative is to take a more closed and selective approach. In that case, one selects and approaches specific partners with the intent sharing of source code between selected partners. In this case, the partners can be internal to the company, such as in the case of inner source, or between companies, by, for instance, shared supplier arrangements.
Rationalized in-sourcing strategy (internal): A very typical strategy in large, multi-national organizations is to shift maintenance and evolution of commoditizing functionality to other internal units that are based in low-cost geographical locations. Due to the lower salary cost, this reduces cost and does not require the company to open up to other partners.
Outsourcing strategy (external): Alternatively, the company can outsource the maintenance and evolution to an external supplier who moves the responsibility for the functionality to one of its own low-cost sites and take responsibility for maintaining the component or subsystem.
Push-out strategy (internal): The most obvious strategy, at least in theory, is to drop commoditized functionality by terminating maintenance and support and finally even removing the functionality from the code. This is of course the most cost effective solution. In practice, however, companies often don't know what customers are still using the functionality and the perceived risk is often so high that companies do not even dare to consider it.

There are several ways to reduce internal R&D investment in commodity functionality. In the earlier parts of the book we focused on internal strategies whereas here we focused on ecosystem strategies. The goal is the same: maximize the investment in differentiation and deliver commodity at the lowest total cost of ownership.

Concluding Remarks

This part of the book is concerned with engaging the ecosystem surrounding the company. We identified that there are three ecosystems that companies need to focus on, the innovation ecosystem, the differentiating functionality ecosystem and the commodity functionality ecosystem. Similar to the 3LPM model, each of these ecosystems have different goals and success metrics. We discussed the process for engaging each of the ecosystem types using a standard set of steps, starting from agreeing on strategy, assigning resources, engaging the first partners, quantitative measurement and finally scaling the engagement.

Companies can use a variety of strategies to operate in each of these ecosystems and in the latter part, we introduced the three layer ecosystem strategy model (TeLESM) and the strategies that we have identified in our research. In the figure below, we provide an overview of the various strategies.

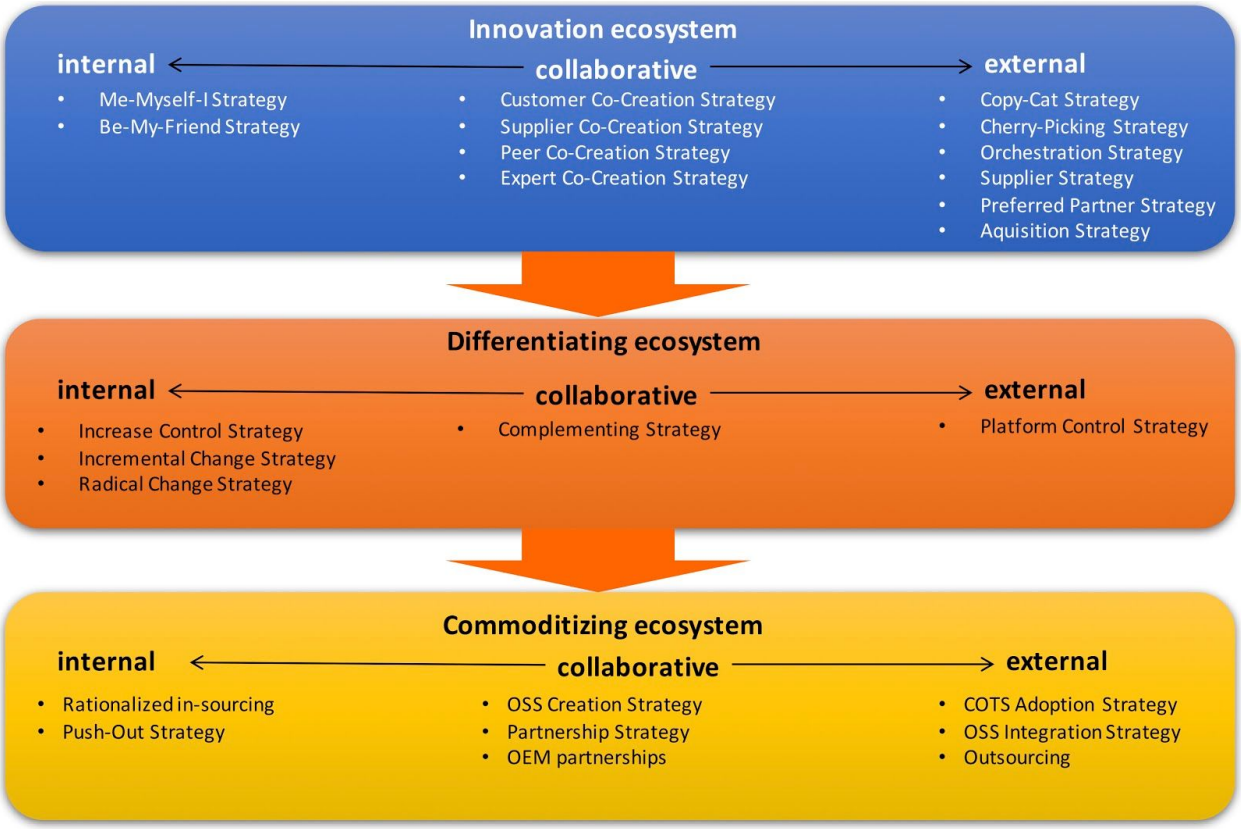

Figure X: Overview of the TeLESM strategies

Conclusion

The role of software in society is increasing exponentially. There are several drivers for this. First, Moore's law allows for doubling the amount of software in a system every 18 months without increasing the cost for the supporting electronics. Second, the flexibility and malleability that software offers allows companies to standardize mechanics and electronics and to differentiate products and systems through software. Third, continuous deployment allows companies to deliver updates and improvements to products and systems even after these have been deployed in the field.

As a consequence of the aforementioned developments, however, the size of software in systems has been increasing rapidly as well and research shows that, depending on the industry, the size of software increases with an order of magnitude every five to ten years. This

has many implications on software R&D but one of the primary ones is that a rapidly increasing amount of R&D resources are spent on software functionality that is commodity, meaning that it does not offer any differentiation as compared to competitors. In fact, in most companies as much as 80-90% of all R&D resources are expended on commodity functionality.

This book provides one set of tools to address the aforementioned challenge of effectively and systematically reasoning about software assets, resource allocation, refactoring, platforms and engaging the ecosystem surrounding your organization. As the foundation for this set of tools we have developed the three layer product model (3LPM) as a tool and framework to reason about strategic use of software in large scale software engineering. The 3LPM, either conceptually or physically, organizes the software functionality into one of three types, i.e. commodity functionality, differentiating functionality or innovative and experimental functionality. By categorizing the functionality in a system, the 3LPM supports several types of decision making that allow for a much more strategic and focused approach to software R&D. The figure below shows the basic elements and structure.

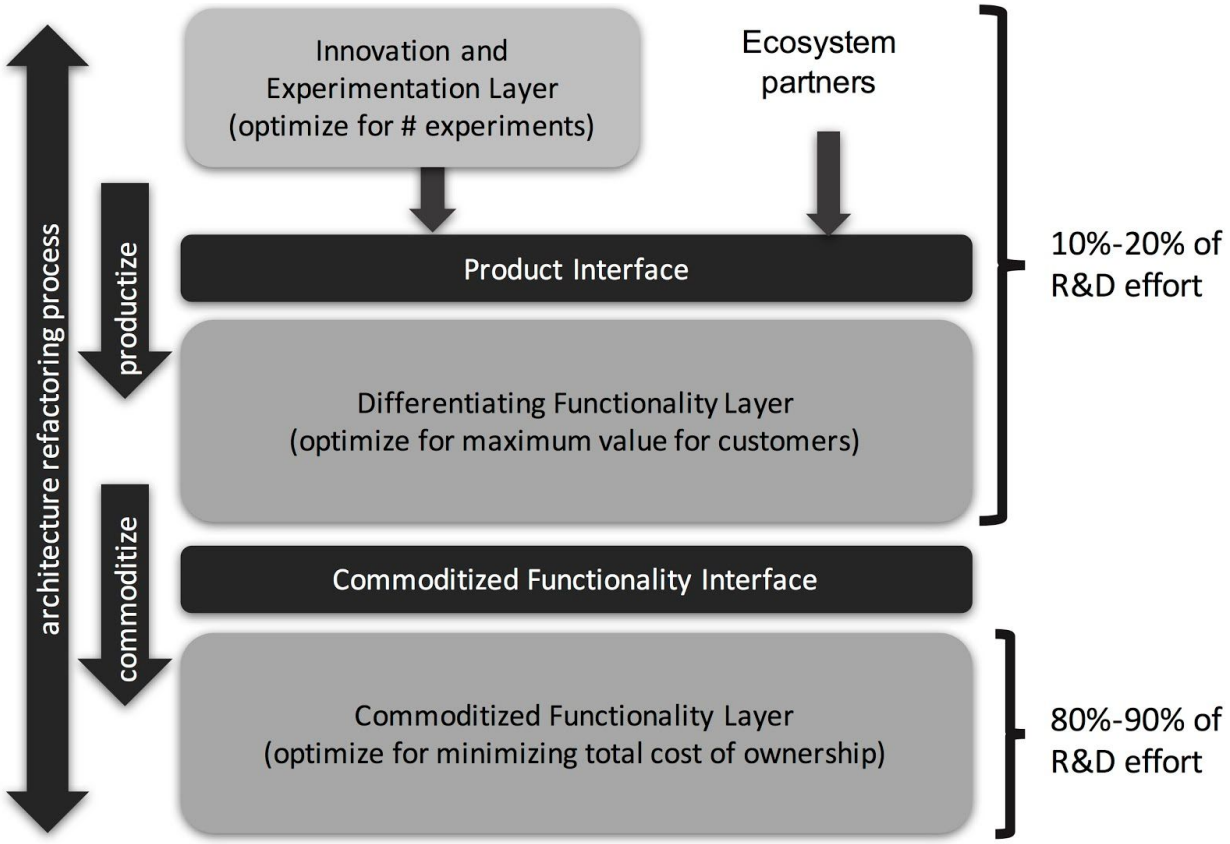

Figure X: The Three Layer Product Model

Although the 3LPM may easily look like a high-level architecture picture, it can be used for several use cases concerning resource allocation, architecture refactoring, software platforms and ecosystems. In this short book, we have discussed the four primary use cases, organized as the four main parts of the book:

1. **Strategic resource allocation**: In part I, we focused on the key problem that we raised earlier: resource allocation into software R&D is often conducted without any understanding or awareness of the differences between different areas of functionality in the system. Allocating resources in an undifferentiated fashion easily results in the majority of resources being allocated to commodity functionality rather than innovation and differentiation. In this part of the book, we started by categorizing the components in existing software as innovative, differentiating or commodity. Based on this, resource allocation can be controlled by allocating resources predominantly to innovation and differentiating functionality whereas only resources for bug fixing are allocated to commodity components. This facilitates limiting resources to commodity functionality and results in a much more strategic allocation of resources.
2. **Refactor software**: Although part I is concerned with strategic resource allocation, it does not affect the structure of existing software assets. The second part of the book is concerned with refactoring the software to align with the architectural structure proposed by the 3LPM. This provides much simpler management of software assets and allows for organizational alignment with the main software layers. In part II, we focus on assessing the current architecture, designing the desired architecture and planning and implementing the transformation.
3. **Towards platforms**: As the structure of the 3LPM suggests, there is a very natural transition from a single 3LPM for a system or product to a platform where the 3LPM is used to model where functionality is allocated and how it transitions. When a software asset (or multiple) are split into a platform and products on top of the platform, the interface and the process for moving functionality between products and platform need to be discussed. Of course, at this point it will also have become obvious that one can experience a cascading series of 3LPM where the commodity layer of one model, for instance a product, is aligned with the differentiation layer of the platform model that it is built on top of. This is the topic of part III.
4. **Engaging the ecosystem**: In the final part of the book, we discussed the use of the 3LPM to engage with the ecosystems around the company. Our research shows that it is helpful to consider that an organization is involved in at least three ecosystems, organized around the 3LPM layers, i.e. an innovation, differentiation and commodity ecosystem. As companies are no islands, the ability to engage ecosystem partners, either in a directed or undirected fashion, for those parts where the company itself does not have a unique differentiation is critical. However, once again, companies often lack a systematic and effective model for deciding when and where to engage their ecosystems. In this context, we'll discuss the Three Layer Ecosystem Strategy Model (TeLESM) to provide strategic guidance on ecosystem engagement.

Conceptually simple as it may be, the 3LPM model, using the four main strategies described in this book, can help you and your organization to double the effectiveness of R&D. By freeing up 50% or more the resources allocated to commodity functionality and reassigning these resources to innovation and differentiating functionality, we can significantly increase the business value generated per unit of R&D effort invested.

Together with another publication [Bosch 17b] where we discuss the use of data to build better products and at least double the ratio of valuable functionality versus unused functionality, this book offers you the opportunity to materially shift the strategic and competitive position of the company by focusing your R&D resources on the activities that matter and to free up resources in the areas where it doesn't. Even though R&D managers often feel that many investments are unavoidable and as such feel victims of decisions made in years past, this book gives you the opportunity to retake control over your R&D investments and to ensure that decisions concerning R&D are closely aligned with your business strategy. Time to take back control and to focus on what really matters!

References and Further Reading

[Moore 93] Moore, James F. "Predators and prey: a new ecology of competition." Harvard business review 71.3 (1993): 75-83.

[Bosch et al 13] Bosch, Jan; Olsson, Helena Holmström; Björk, Jens; Ljungblad, Jens; The early stage software startup development model: A framework for operationalizing lean principles in software startups, Lean Enterprise Software and Systems, 15-Jan 2013, Springer Berlin Heidelberg.

[Bosch 17a] Bosch, Jan; Speed, Data, and Ecosystems: Excelling in a Software-Driven World, 2017, CRC Press, ISBN 9781138198180

[Bosch 17b] Bosch, Jan; Using Data to Build Better Products, CreateSpace Independent Publishing Platform, ISBN-10: 1541210808, 2017.

[TeLESM] Holmström Olsson, Helena and Bosch, Jan, From ad hoc to strategic ecosystem management: the "Three-Layer Ecosystem Strategy Model" (TeLESM), Journal of Software: Evolution and Process, Volume 29, No 7, 2017.