Service Thinking – Designing digital experiences with atomic functionality

Georgios Jason Efstathiou

03.05.2019

Last week, I travelled to Budapest for a weekend getaway.

I got up, took a shower, had a coffee, grabbed my things, took the train, went through airport security, found my gate, boarded the plane, watched an episode of Black Mirror, got out of the plane, left the airport, and went on to my day of sight seeing.

Without services, none of this would have been possible.

The water I used for showering reaches me through infrastructure provided by the government, which I pay for in taxes and monthly payments. The coffee beans I used for my coffee were delivered to me a few days prior, travelling through Amazon's vast logicists infrastructure after my request submitted through an app on my phone. Public transit, again, is a transportation service funded by the government and my monthly payments as a transit user. Airport security is necessary service that builds trust between me and airlines, and eventually the states allowing me to enter them without screening me specifically. Airlines are service providers that allow me to travel practically anywhere on this planet in less than a day. When I entered the Budapest airport, the Hungarian government simply allowed me in – not because they trust me, but because they trust the service that issued my passport.

As cizitens, users, and human beings in this day and age, we have become dependant on services. They allow us to have roofs over our heads, meet basic sanitary needs, be financially flexible, and provide us with the goods we need to biologically survive and procreate. Through society and monetary exchange, humanity has learned to exchange not only goods – but also efforts and experiences. We exchange value for value – money for affordances that allow us to depend on a third party to perform something we need.

In the late 20th century, digitalization happened. And with it, services have moved on to their digital habitat – which allowed them to scale to incredible levels. 300 hours of video are uploaded to YouTube every single minute. The MasterCard payment network processes 74 billion individual monetary exchanges per year. 63,000 search queries are submitted on Google every second. Social media users post 95 million individual images and videos on Instagram each day.

Each of these services are relatively simple on the surface. At the end of the day, they all provide you with a specific outcome for which you pay for in some way or the other. You, as a user, receive the outcome. And for it, the service receives something in return – something that, besides making a profit, it can use to uphold the vast infrastructure needed to provide its users desired outcomes.

Below the surface, every single one of these services starts to look a bit less simple. Service meshes of overwhelming complexity perform in itself incredible feats to provide parts of what is needed to provide that desired outcome. At this point, both the user and service are machines. Netflix, a video streaming service, hosts its video library on Google servers, using a service called the Google Cloud Platform. Google offers this service to essentially sub-lease its global infrastructure to third parties that want to offer digital experiences on their own. Netflix itself is entirely dependant on such a service, because building the kind of incredible network, with hundreds of data centers around the world and intricate mechanisms to intelligently balance load across the planet, is not its competence as a business entity. Netflix streams video – they don't host the videos themselves. Similarly, most digital experiences rely on tens to hundreds third party "subprocessors" – services that power things from marketing and data analysis to authentication and fraud detection.

This thesis aims to illustrate digital services in a unified model that can be applied to a service concept from the microscopic layers of so-called "atomic functionality" – small, dedicated mechanisms performing a single action – through the macroscopic layer that the defines the service and its performance itself, to "Service Landscapes"; a cluster of individual service models that represent a service provider's range of offering. We will be looking at existing models and papers about these layers that make up a service, and talk to experts in their respective fields in order to distill our model, attemping to deliver a coherent concept that can be used to design, define, analyze and optimize digital services.

The desired outcome is that you as the reader adopt a modular way of thinking that allows you to understand and oversee the development of digital services across different fields.

So far, it seems like the concept of services is overwhelming. We are now aware that we depend on services – services ranging from streaming cat videos online to the streets you walk on every day, provided to you by your government which you pay in taxes. We also know that a service, beyond its face, is made up of intricate mechanisms that often themselves depend on other services – and we know that in this network, often services themselves act as users to other services.

While there are many slightly different definitions of the "Service" concept to be found in literature, a common definition includes two actors – the service itself and its user(Iqbal 2018a).

The User in this model has a set of needs(Swearingen 2000a), which causes seeking the respective service in the first place.

Through economic defintions(Wolak 1998a), we can add an exchange of value between these two parties. After all, without any value being provided in any way, potential users would have no incentive to actually become users (*convert*)(Wolak 1998b). In parallel, the service provider would have no incentive to support the service itself.

The clear difference between traditional goods exchanges and services is that a service, at least in its core value defintion, *does not provide you goods*, but outcomes – with the service itself being *intagible*(Swearingen 2000b).

When purchasing an item, a value transfer takes place(Swearingen 2000c) – and after that, the client possesses the item that value was exchanged for. In a service context, the value is not manifested physically(Swearingen 2000d). For example, I can pay a service like Boldking on a monthly basis to deliver me razor blades every few weeks(BOLD Online BV 2019). While I do receive physical goods as part of this agreement, the *outcome* of this service is *receiving fresh razor blades every few weeks.* By subscribing to this model, the user can rely on a third party to supply them with a frequently needed physical good, shifting responsibility from the user to the service, in exchange for monetary value. This kind of recurring, stable model allows the service provider to optimize their service's intricacies to a point where subscribing to a recurring service performance is often more lucrative for the user than sourcing the desired item on demand(Chuang and Gellings 2009).

The service receives value from the user as well – may it be monetary, or something more abstract that represents value in another form. For example, searchgiant Google offers *free services* such as Google Mail or Google Maps. Those services do not require the user to pay in order to use them, but they generate valuable behavioral and personal data that Google is able to enrich by cross-referencing information from all their services, allowing them to lucratively monetize by using this information to provide another range of services – advertisment targeting(Google LLC 2019i).

By combining these offerings across *Service Landscapes*, Google is able to provide vast value(Alphabet Inc. 2019) to seperate markets and its shareholders.

We can define these aspects in our model by adding the *outcome*, which satisfies the user's needs, and an *agreement*, which represents the value exchange between the service provider (in this representation, the service provider is represented by the service itself).

In order to provide an outcome, a service *performs*(Iqbal 2018b). The *performance* is commonly defined as the sum of all mechanisms that act together as part of the service to create the desired outcome. When a user orders an item on Amazon, for example, the Amazon E-Commerce platform performs feats of logistics operations with the outcome being the desired item in your hands in often less than 48 hours(Amazon Inc., BusinessWire 2018).

In this model, we can define the *performance* as the connection between the service, as the performing actor, and the *outcome*, which said performance leads to.

And with that – we have our base model and definition.

> Services provide outcomes that satisfy user needs by performing the feats necessary to achieve said outcome.

The layer model

In order to explore our service model in greater detail, let us break down the service concept into multiple layers.

What we looked at so far can be defined as the *Macro layer*, where we look at a single service – outcome – user relationship. The *performance* in this model a cluster of more services, each performing a smaller action that together allow our macroscopic service model to function and perform. Let us define this as the *Micro layer*, which represents all of these smaller sub-services as a whole.

Multiple services often work alongside each other to make sense together. The actors in the above defined model do not necessarily need to be aware of this fact. Let us go back to Google's core business model – offering consumer services that generate valuable data to power an advertising platform capable of detailled profiling and campaign targeting. For this reason, let us add another layer on top of our macro layer: the *Service Landscape*.

In the following chapters, we will explore these layers in more detail, bottom to top.

Let us begin by taking a look at the first level in our model: The micro-layer.

As we discussed in Chapter 1, a service *performs* in order to provide an *outcome*. In this chapter, we will study a selection of patterns from the world of engineering in order to distill a model that allows us to represent and think about *performances* in an approachable fashion.

Frequently, the core value definition of a service directly references its technical capabilities. For example, one of the main selling points of Google Docs, an online document editing platform, is its real-time document sharing, where multiple parties can work on the same document over the Internet(Google LLC 2019d). The core offering of Google Docs has fundamental implications for its system architecture: Communication and databases need to work in real-time and instantly synchronize across multiple clients(Barashkov 2018). This illustrates why the technology behind a service is not just a necessity, but a core part of its design as a whole: User Experience specifications impact key system architecture decisions directly(Thomson 2016).

Engineering at scale

The concept of *technical debt* defines how hard a codebase is to expand, adjust and scale(Fowler 2003) – in short, to maintain. A frequent contributor to *tech debt* is code duplication — the same mechanism implemented repeatedly in different parts of the codebase(Spinellis 2006). This, together with a lack of documentation, elevates maintainability effort(Endenburg 2016), which directly equates to technical debt.

Thus, one of the most popular principles in software engineering is DRY coding(Diggins 2011) – Don't repeat yourself.

Keeping it dry

The DRY principle is stated as:

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system(A. Hunt 1999).

Any repeatedly occuring constant should be defined so that any dependant function simply references this single, global definition.

It is easy to imagine pragmatic examples for this concept: When a service with an unfortunate code quality goes through a rebranding procedure, changing one of its primary brand colors represents an excruciating task of hunting down every occurence of the hard-coded color variable and changing it individually. If a *single source of truth* were referenced instead however, this global could simply be updated and the change would instantaneously propagate through all of your interfaces automatically.

The concept of storing fundamental constants and repetitive patterns globally in the context of visual design for digital products is frequently applied as part of *Design Systems*. A Design System is famously defined as a library of reusable User Interface components, often co-existing as design specification and production-ready code components(Clark 2018).

Design consultant B. Frost in 2013 coined the term "Atomic Design", which is a methodology for creating Design Systems divided into 5 levels:

1. *Atoms*, tiny, abstract elements, "like color palettes, fonts and even more invisible aspects of an interface like animations", or basic HTML tags in the context of a website, such as a form label, an input or a button(Frost 2013a).
2. *Molecules*, "relatively simple combinations of atoms built for reuse", such as form fields jointed into a full form(Frost 2013b).
3. *Organisms*, "groups of molecules joined together to form a relatively complex, distinct section of an interface", such as the header of a website. This is where the analogy starts becoming specific to distinct projects(Frost 2013c).
4. *Templates*, "groups of organisms stitched together to form pages", such as a generic, wireframe version of an article page for a news website(Frost 2013d).
5. *Pages*, "specific instances of templates"(Frost 2013e).

A specific implementation of a design system, "Lightning Design" by CRM behemoth Salesforce, takes *Atoms* a step further and proposes a single source of truth for *atomic constants*, the very fundamental values that encode and express a brand visually.

> Design tokens are the visual design atoms of the design system — specifically, they are named entities that store visual design attributes. We use them in place of hard-coded values (such as hex values for color or pixel values for spacing) in order to maintain a scalable and consistent visual system for UI development.(Salesforce.com, inc. 2019)

In essence, this allows sharing constants like colors not just across one application, but a full suite of implementations. If the source of truth is adjusted, everything updates globally.

The Design System concept exhibits strong parallels to *Composability*, an often-cited principle in system design, generally referring to the design of systems using self-contained, modular pieces that can be individually adjusted, changed and upgraded(Peter G. Neumann 2004).

The Unix Philosophy

A well-known manifestation of the DRY paradigm as a composable system is the Unix Philosophy, originated by Computer Science pioneer Ken Thompson in the late 1970s. Unix is a command-line based operating system developed at Bell Labratories that has since grown to be the basis of modern operating systems such as Ubuntu and Mac OS to this date. If you open the Terminal application on a MacBook today, you are interacting with an Unix system directly. User interfaces on these systems are simply executing commands like these behind the scenes.

The Unix philosophy, summarized by computer scientist Peter H. Saulus in 1994, entails:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

The first paradigm represents a striking similarity to DRY. In Unix, it is possible to string multiple commands together and *pipe* (represented by the pipe character |) the first one's input into the next. For example, the command `ls` (for *list*) simply prints the names of files and folders in the current working directory to the screen:

```
$ ls
uglyCat.jpg
beautifulCat.jpg
dog.jpg
```

Another Unix application, `grep`, takes an input and outputs a filtered version based on a search term. By *piping* the output of `ls` into `grep`, we can search the working directory's contents. Let us say we want to find the image of the dog in this folder.

```
$ ls | grep "dog"
dog.jpg
```

As you can see, `grep` filtered the output of `ls` so that the resulting output is only the line including the term "dog".

This demonstrates the paradigm perfectly: Both `ls` and `grep` do one thing, and they do it well. By stringing together multiple applications and making them interact with each other, we can quickly perform much more complicated patterns.

What does this have to do with services?, you might ask – let me tell you.

Service Thinking in System Architecture

Let us think about what `grep` and `ls` really are. We, as the user, have a need – we need to find a file in our directory. `ls` and `grep` individually perform parts of what is needed in order to generate an outcome that satisfies our need – a filtered list of files in a directory.

You might already be able to tell where this is going – we can treat `ls` and `grep` as *services* as defined in Chapter 1. For the sake of simplicity, let us assume we write a simple script that encapsulates `ls`'s output piped into `grep` as a single command – `sd`, for *search directory.*

```
$ sd "dog"
dog.jpg
```

Now, we can define `sd` as our macroscopic service – with `ls` and `grep` acting as the atomic functionality inside, allowing `sD` to perform.

Since `sd` is very simple and we know exactly just how it achieves its outcome, we can go a step further and visualize the performance. Now, `sd` acts as the user to our two other services – `ls` and `grep`. By chaining our two bits of atomic functionality together, `sd` is capable of performing a vastly more complicated action.

Both `ls` and `cd` are entirely unaware of their role in this system. They simply perform the action they are designed to perform, nothing more, nothing else. They receive an input and provide an output – how said output is processed afterwards is irrelevant.

In this example, `sd` is our macroscopic *service*, and `ls` and `cd` act as *microservices.*

Microservices in real-world applications

More and more frequently, sophisticated systems are designed with the *Microservice* paradigm, directly derived from DRY and the Unix philosophy. In a Mircoservice pattern, business logic is split into *single responsibility* atomic bits that each *do one thing, and do it well.* These services are *decoupled*, meaning they each have their own seperate place to live in and act, and do not know or care about others performing tasks around them.

> A microservice should be so tiny that when someone wants to adjust its behaviour, it's a no-brainer to just rewrite it from scratch.

This kind of architecture comes with lots of benefits. Each little mechanism is ideally so small that its functionality can be described and documented in an extremely simple to understand way. More complex logic is simply a result of orchestrating individual small mechanisms in order to achieve the desired end result.

In this chapter, we will take a closer look at the macroscopic level – simply referred to as *the service* itself. This layer is what our end-user is actually interacting with[1]; the face of the complicated inner workings that all together perform to generate the service's outcome.

Humans as users

While in Chapter 2's micro-layer, the users of our services are machines, at the macro-level, our users are humans. Since our end-goal is solving needs for these very human users, this layer defines our design approach and ultimately dictates our service's design as a whole. This is where *User experience* (UX) and *User interface* (UI) design come into play.

[1]In theory, our macro-level approach can also be applied to machine—machine service relationships. For the sake of simplicity, this chapter will assume that we define our macro-level as the level at which end-users interact with our system. Read "Drawing the line" in Chapter 4 for more information as to why this might not always be the case.

The international standard ISO 9241-210 defines *User Experience*(International Organization for Standardization 2009a):

> User Experience is a person's perceptions and responses that result from the use or anticipated use of a product, system or service.

Most services, unless their existance is justified by necessity or scarcity alone, need to optimize User Experience in order to convert and retain their customers. And because our end-users are human, human-centered design is one of the most important paradigms in shaping and designing a service.

User Experience and User Interface are often used interchangably – but in reality, their meaning is different in very important ways.

The Oxford Dictionary defines *User Interface*(International Organization for Standardization 2009b):

> The means by which the user and a computer system interact, in particular the use of input devices and software.

The User Interface can thus be summarized as the *medium* our user interacts with our service through.

Many aspects of this are standardized; in the example of a web application or website for instance, a user is as of writing 63.4% likely to access a website through a mobile device, such as a smartphone(Statista 2019). As such, naturally the scope of user interface design for such a product is restricted to the website itself – typically living inside a web browser. *User Experience Design*, however, goes beyond this – in fact, the conclusion of choosing the web as a medium is, due to the vast implications on perception, a key User Experience decision.

To sum it up; *User Experience Design* is optimizing the qualitative sum of experiences and perceptions in the context of interacting with a product. The *User Interface* is the medium that a user interacts with while using a service.

Defining the service baseline

In order to bring end-users into the equation, our model needs an indicator of end-user *interaction.* In order to represent the point at which users actually come into contact with our digital product, we can define the *service baseline* as this very point — the face of our service.

In essence, we can define the *service baseline* as the point-of-contact in our system that sits directly adjacent to the *user interface* in order to represent the point in our system that shapes our service as part of a *user-driven design* process.

Lowering cognitive complexity

In Chapter 2, we have briefly looked at the world of engineering and one of its most important paradigms — "don't repeat yourself": Defining constants and functionality as re-usable atomic bits that are shared across all dependent mechanisms.

Product & User Experience Designer B. Oxendine highlights a similar line of thinking in designing the actual interfaces that users interact with.

> (...) if you're starting with a system — a system of components, a structure of where things in the app live — then it should be relatively clear where to place things. It's important to understand the basic functions of your app, like let's say adding an item in a to-do app, and putting those basic functions into logical groups where they can be reused elsewhere in the experience. So you don't end up designing two ways of adding a to-do in two different locations.("Interview with B. Oxendine"a)

When asked about why having two different ways to achieve the same thing is not optimal, he continued:

> If you reduce the total amount of concepts a user has to learn in order to use the app, then the app becomes easier to understand. (...) More screens, really, more anything in your app automatically makes it harder to understand, naturally. Your users won't understand why there's two ways to do the same thing, they'll think that there has to be a difference somehow.("Interview with B. Oxendine"b)

In Chapter 1, we have learned from Engineers that achieving the same end-result in different ways in the context of a single service introduces problems — in User Experience Design, it is also considered bad practice, but resulting from a very different perspective: The unnecessary complexity directly affects your users, who after all are the ones that actually need to understand a user interface in order to use a service.

Matching concepts across use-cases

B. Oxendine brings up the example of a to-do service, in which you can create to-dos and then set reminders for those to-dos.

> (...) think of the things you can do with a to-do in this theoretical to-do app: you can create one, edit it, delete it, maybe put a reminder on it. (...) It wouldn't be good to create to-dos and reminders as separate concepts. They should exist in the same system.("Interview with B. Oxendine"c)

What becomes apparent now: A service that solves multiple interconnected use-cases should tie their resolutions together so that the service as a whole is aligned and easy to understand. But what is it about reminders and to-dos specifically that makes it natural to merge them into one "concept"?

> Well, a to-do has everything a reminder needs to be useful, like, some text, a deadline and a checkbox, and a reminder is like an extension of that. In this case maybe you get a notification about your to-do. You can create reminder functionality by notifying a user about a to-do and the other way around it doesn't really work.("Interview with B. Oxendine"d)

When asked about why exactly "it doesn't really work" the other way around, Oxendine clarifies that he is talking from experience and intuition ("I mean this is all hypothetical — actually, you'd want to test this kind of stuff with real users"("Interview with B. Oxendine"e)), but raises an interesting point:

> the to-do should be the basis of this kind of app, because you can extend a to-do with a reminder. You could also create reminders and then remove the notification to get a simple to do. But see, now, the concept is getting harder to understand. Generally, the whole point I'm trying to make is — one way or the other, these two ideas are so similar, they should be the same concept (. . .).("Interview with B. Oxendine"f)

Oxendine's line of thinking in general can be summed up to making a service as simple as possible — and that entails breaking down use-cases (like creating a to-do and creating a reminder) into small parts, which can then be assembled into patterns that reduce the amount of *concepts* required to be understood in order to use a digital product.

Need diversity

Now, let us apply this knowledge to our service model by breaking down needs and outcomes into atomic bits.

In Chapter 1, we have defined *Needs* and a service's *outcome* as main aspects of defining a service; the user approaches with their needs, and our service, through its performance, provides an outcome that aims to resolve those needs.

Of course, as diverse as humans are, so are the exact set of needs of users. Depending on context and the specialization of a service, there might be a narrow or extremely wide range of specific use-cases – manifesting in patterns in respective sets of needs.

By comparing the set of needs with the ultimately provided outcome, we can measure *how well* our service *fits* the need pattern associated with a specific usecase. This approach exhibits parrallels to the concept of *Product/Market Fit* (PMF), an important dimension in Product Development and User Research. American Entrepeneur Marc Andreesen defined PMF as "being in a good market with a product that can satisfy that market."(Andreesen 2007).

Case resolution

In our model, we can thus define a *resolution* for each *use-case* (use-case = recurring pattern in set of needs). Let us refer to this as the *case resolution*. When looking at the relevant parts from our diagram in Chapter 1, we can represent the case resolution as an overlap between the user's set of needs and our service's end result.

In this representation, everything a service provides that is not part of the set of needs is represented as *overhead*. In parallel, everything in the set of needs that is *not* part of the outcome is *loss*. When optimizing for a certain usecase, *case resolution* should grow, in turn minimizing *loss*. By reducing *overhead*, ultimately unnecessary functionality can be cut, improving agility and reducing cost of the product development and maintenance process. Of course, the overhead in a specific use-case might be a vital contribution towards the case resolution of other use-cases("For the Sake of Simplicity, the Set of Needs and Outcome Are Set to a Constant Size. at a Later Stage, We Will Look into Quantifying Their Size Further." n.d.).

Service resolution

By summing up all sets of needs we can define a macro-level dimension of our service model: the *service resolution*. The service resolution describes to what amount the service's outcomes matches user needs by representing how well our service solves real user needs across all use-cases.

In order to calculate service resolution across use-cases, let us calculate the average of case resolutions.

Quantifying Needs and Outcomes

So far, we have looked at the Service Resolution only as the intersection of the Set of Needs and Outcome, with both being of arbitrary *breadth*. In reality of course, some Sets of Needs are *more diverse* in themselves — meaning, their resolution requires more or less losely defined effort. For example, two use-cases of a banking app might include "Viewing account balance" and "Making a transfer". In order to inform the user of their account balance, the balance simply needs to be displayed on screen. In order to initiate a standard wire transfer, at least the beneficiary's name, IBAN and BIC is required(The European Parliament and the Council of the European Union 2009). The latter need is thus more complicated to solve as a direct result of the sum of *resolution complexities* for all resolutions involved. Of course, a singular use-case may include many seperate features, and multiple use-cases may share the same performance. For example, viewing the updated account balance after submitting a transfer may be considered part of the "Making a transfer" use-case.

By defining *resolutions* as small building-blocks that meet their directly equivalent *need*, we are able to quantify *breadth*:

This way, we can assess *case resolution* on a need-by-need basis, allowing us to exactly pinpoint which resolutions are actually required to meet our main use-cases.

Service breadth

Analog to *case resolution* and *service resolution*, we can upscale *breadth* of case-specific needs and outcomes — *service breadth*. Instead of taking the average *breadth* across all use-cases, as we are now counting an absolute number of *needs* and *resolutions*, we can simply add together the number of unique needs. This means that, going back to the banking app example, the need "View account balance", which may be part of use-cases "Making a transfer" and, naturally, "Viewing account balance", is only counted once despite being part of two seperate sets of needs.

Macro-dimensions

Let us quickly sum up the dimensions we have looked at in this chapter.

On the use-case level, where we look at specific use-cases:

- *Case resolution*, the percentage of needs in a given use-case that the outcome our service provides resolves. This is an indicator for how well our service is able to support individual use-cases.
- *Service resolution*, the average of all case resolutions across use-cases. This is an indicator for how well our service fits real needs.
- *Breadth*, the complexity of a set of needs for a given use-case or an analog *outcome*, measured in the absolute number of *needs* in a set of needs or *resolutions* as part of a provided *outcome*. This is an indicator for the complexity of use-cases and the complexity of resolving them.
- *Service Breadth*, the absolute number of *resolutions* provided by the service across all use-cases. This is an indicator for the overall complexity of our service.
- *Loss*, needs for a specific use-case that is not resolved by the outcome.
- *Overhead*, resolutions for a specific use-case that are not needed to resolve the set of needs.
- *Service Loss*, the result of breaking down a set of needs into individual needs, breaking down outcome into individual *resolutions* mapped onto needs, and then counting unresolved needs globally across use-cases.
- *Service Overhead*, the result of breaking down a set of needs into individual needs, breaking down outcome into individual resolutions mapped onto needs, and then counting those resolutions that do not map onto any need, globally across use-cases.

We can display *service resolution* and *service breadth* in a representation, where we look at either *service* or *use-case* levels:

By comparing these two dimensions, we can easily analyze the *specialization* of a respective service. In the above example for instance, we can see a highly specialized Service B, which exhibits extremely high resolution with a narrow breadth. In practice this means that this service is confronted with a narrow range of user needs, and is thus able to maintain a high resolution with a small amount of complexity — in short, it solves a high percentage of needs, and it needs only a small amount of specific *resolutions* to do so. Service C on the other hand has a very high breadth, with overall little resolution. This might be resolut of focussing on a high amount of usecases; as a result, the service may be able to attract a wider range of users, but the quality of specific outcomes and thus experience of users may suffer as a result.

In order to be able to represent *service loss* and *service overhead*, we can compile a bar chart in which we add up individual resolutions and unresolved needs.

In this representation, we can view at a glance how well our service maps onto real user needs, as well as assess its overall *breadth*. Service D in this example has high loss, meaning some use-cases are not resolved entirely. Service E on the other hand has substantial overhead, indicating that it is attempting to solve problems that users do not actually need solved. Service F, on the other hand, has rather minimal loss and overhead but a high amount of resolved needs, indicating a balanced service which, without being overlay complicated, is able to solve a healthy amount of real user needs.

By adding up *overhead*, *loss* and *resolved needs*, we can easily express these dimensions as percentages – all ultimately unnecessary resolutions (*service overhead*), plus all unresolved needs (*service loss*), plus all resolved needs gives us a total number of needs. By simply dividing the amount of either of these dimensions by the absolute sum of them all together, we can express *service overhead*, *service loss* and *service resolution* as percentages of the total amount of needs.

Extending with custom dimensions

A common term in many businesses is “Key Performance Indicator”, KPI for short, defined as “the critical (key) indicators of progress toward an intended result”(KPI.org, a Strategy Management Group company 2019). Our macro-level dimensions defined so far represent general properties of any service, but for the sake of specializing the model for specific situations and business needs, it shall be noted that any KPI may be used alongside here-defined dimensions in order to analyze a given service with greater detail.

So far, we have defined our *service* model, we have looked at how we can break down inner workings into atomic functionality, and we have defined dimensions and indicators for how efficiently our service solves real-world use-cases.

In the real world, services rarely come alone. Different services by a singular service provider are frequently offered alongside each other in order to deliver a range of offering solving problems in a particular market.

For example, the Berlin-based fintech organization N26 offers solutions for mobile banking. Their range of offering includes, amongst others, a bank account(N26 GmbH 2019a), savings deposits(N26 GmbH 2019d), credit(N26 GmbH 2019c) and “Spaces”(N26 GmbH 2019e) (the ability to create sub-accounts of sorts to save up towards specific goals). This pattern can be spotted all across the industry — other examples include search-giant Google offering almost 100 user- and business-facing services(Google LLC 2019j), Amazon Web Services including around 150 distinct services(Amazon Inc. 2019g), and Uber running a meal delivery service(Uber Technologies Inc. 2019) alongside their main transportation offering.

Drawing the line

As per Chapter 2, micro-services that enable performances as part of a macroscopic service are *services* as well, and all models defined so far apply to any *service*, regardless of context. It is entirely possible to shift focus of our analysis onto any *service*, including microservices. In Chapter 3, we have defined *the service baseline* as the level at which our end-users interact with our product directly.

Generally, as a result of the above, it is possible to shift the macro-layer of our analysis model to any *service* instance, freely, regardless of context. This means that we can define a microservice that is just one of the many parts enabling the performance at the baseline level as the macro-layer, at which point all sibling-microservices in a given system form a *Service landscape*, and all even more fundamental bits of atomic functionality that may be at work within a given microservice in turn shape the micro-level. In this situation, just as is given on the micro-level regardless, both user and service on the macro-level are machines.

Enabling a broad overview

N26 as a whole can be described as a service, with one baseline — and concepts like "making a savings deposit" can be defined as seperate use-cases for this service. However, as we are now inspecting a whole range of offering from a service provider, our approach will soon become overly complicated: We would be counting hundreds of individual needs, spread across hundreds of individual use-cases. For this reason, a more sensible approach would be to treat each individual offering (defined by a clearly seperated value proposition) as their own *service*, and introducing a final, global layer on top: The *Service Landscape*.

Another argument is sufficient *outcome discrepancy*. Let us take another look at the example of Amazon Web Services (AWS). As already mentioned above, AWS, as of writing, constists of around 150 distinct services that each provide a specific value(Amazon Inc. 2019h). Each individual offering furthermore provides fundamentally different outcomes; ranging from database storage(Amazon Inc. 2019d) to speech synthesis(Amazon Inc. 2019c) to Blockchain solutions(Amazon Inc. 2019b). Because their individual outcomes are so diverse — let us refer to this as a high *outcome discrepancy* — they naturally become seperate service models. However, these seperate service entities still coexist in a shared space alongside each other.

Thus, in this chapter, we will define *service landscapes* as the top-most layer in our model — the shared space serving as an umbrella for distinct services. Furthermore, we will research common industry patterns regarding the relationships between services that are part of a service landscape and expand our model accordingly.

The common denominator

In order to understand and define service landscapes, let us look at real-world examples of multiple services coexisting.

AWS' many services all share the same branding — and they are aligned under a single, clear *value proposition*:

> Cloud Computing with Amazon Web Services. Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow. Explore how millions of customers are currently leveraging AWS cloud products and solutions to build sophisticated applications with increased flexibility, scalability and reliability.(Amazon Inc. 2019i)

Similarly, banking-app N26's services are all branded homogeneously, with a shared *value proposition*:

> The first bank you'll love. Take control of your finances. With just one app.

In both examples, users need to sign up for a global *account* before being able to use individual services (e.g. applying for a loan on N26(N26 GmbH 2019b), managing identity on AWS(Amazon Inc. 2019f)). Individual services are sub-brands of the service provider (e.g. Amazon -> Amazon Redshift(Amazon Inc. 2019e), N26 -> N26 Spaces(N26 GmbH 2019f)). This pattern is commonly found across digital service providers that offer more than one core service; examples include G Suite, Google's business-to-business offering consisting of 19 distinct services(Google LLC 2019b), SaaS communication platform Intercom offering 3 core services(Intercom Inc. 2019), and video-sharing platform YouTube providing 9 distinct services besides their main offering(YouTube Llc 2019a). In addition, every one of these examples has all individual services operate in roughly the same *domain*; for example, AWS services can be summed up as "Cloud Services", G Suite services are business productivity tools and Intercom offers tools for business-to-customer communication.

By the multitude of examples, we can add several attributes to our *service landscape* concept:

- *Branding*: Services in a service landscape usually inherit its brand.
- *Authentication*: User accounts are usually created on the service landscape level and shared across child services.
- *Value proposition*: A service landscape usually cites a clear value proposition that holds true for its child services, whereas child services themselves are usually, in addition, individually marketed, with their own value propositions (e.g. Amazon EC2(Amazon Inc. 2019a), Intercom Inbox(Intercom, Inc. 2019)).
- *Homogeneous Domain*: A service landscape usually includes individual services that operate in the same category, solving problems from a specific field (e.g. Cloud Computing) or a specific customer segment (e.g. Businesses, Private, Marketing Agencies etc.)

Nested Landscapes

However, with the example of Google's range of consumer services, things are a lot less clear. While each service is branded consistently, their services appear to be positioned across a much broader spectrum of categories, listing only a *mission statement*:

> Our mission is to organize the world's information and make it universally accessible and useful.

With the Google Account, Google's services do share an authentication system. However, not all of their services inherit Google's brand (e.g. YouTube(YouTube Llc 2019b)), a general value proposition beyond the before-mentioned mission statement is nowhere to be found, and with services ranging from education software(Google LLC 2019k) to a mobile payment solution(Google LLC 2019l), a homogeneous domain is not defined either. Thus, Google's range of services does not constitute a *service landscape* as we defined it.

When taking a closer look, Google does however maintain several service clusters with clear attributes of service landscapes – a selection of examples being Google Drive; a selection of online document editing services built around an online file storage system(Google LLC 2019e), Google Cloud; a cloud-computing competitor to AWS(Google LLC 2019a), Google Play; a suite of mobile apps enabling purchase and consumption of digital media such as E-Books(Google LLC 2019f), Movies(Google LLC 2019g) and Music(Google LLC 2019h), and G Suite; a range of business productivity tools(Google LLC 2019c). Each one of these service clusters meet the exact criteria for being considered a *service landscape.*

Interestingly, G Suite is part of Google Cloud, in itself however is a *service landscape*, while placed next the individual services constituting Google Cloud.

Due to the complexity in real-life business offerings, it is sensible to allow shaping *service landscapes* more freely. We can add the ability to represent free-standing services, as well as nesting multiple *service landscapes* into one-another.

In this situation, nested *service landscapes* inherit attributes from their parent.

Encoding bussiness capabilities in service architecture

While we are currently looking at a high-level representation of services — even above a service itself —, let me remind you that we defined services as entities that are built from microservices, that in themselves equally constitute a service model, and the service in itself resolves *needs*, which are defined as atomic bits directly mapped onto *resolutions* in the service's outcome. In Chapter 2, we have taken a brief look at concepts from Software Engineering, such as DRY

(*Don't Repeat Yourself*), and we have learned about the preffered practice of defining global constants in a single place. In Chapter 3, we have defined *needs* and *resolutions* as *shared between use-cases*, because in order to quantify them across use-cases, equivalent resolutions may only be counted once, and reducing the amount of concepts in a digital service positively impacts User Experience.

Now that we are looking at multiple services in conjuction with *service landscapes*, we can go a step further and study patterns of sharing *atomic functionality*, *resolutions* and *constants* not just between use-cases, but between services.

Sharing atomic functionality across services

One of our main properties of *service landscapes* is the *shared authentication* pattern: The user uses a single account to sign up at the landscape level and through this process gains access to individual services. This highlights the need of implementing the raw atomic functionality enabling authentication as a discrete service in order to prevent duplicated and potentially divergent implementations on the level of individual services. In Chapter 2, we have also briefly learned about *Design Systems*, which are a analog to DRY engineering for user interfaces and frequently reused constants such as colors. Since a *service landscape* by definition shares a brand between itself and children, Design System constants will naturally rise above individual services in order not to violate DRY. Similarly, *service landscapes* also impose a *homogeneous domain* that is passed down to its children. As a result of this, the likeliness of specific *resolutions* appearing in two seperate services within the same landscape is inherently high.

In order to follow the DRY paradigm, every time a specific *resolution* is found to be present in at least two seperate services, the *atomic functionality* enabling the *resolution* should be decoupled into a seperate service entity that can be re-used across the *service landscape*.

Let us apply this concept to the authentication system, which, as we know, is shared across our *service landscape*. As authentication is likely to be required in order to access any of the services in a specific landscape, authentication automatically becomes a part of every single *set of needs*.

With *Branding*, *Authentication*, *Value Proposition* and *Homogeneous Domain*, we have already represented top-down **inheritance** at a high level:

We can extend this concept to allow representation of sharing standardized bits of functionality between services. In order to do this, we need to define an indicator for distinguishing between user-facing and machine-to-machine services. In essence, we are required to tell if a service includes a *service baseline* (defined as the point-of-contact between an end-user and the service). Ultimately, this allows telling apart user-facing from shared, internal microservices at the landscape level("Please Note That This Differentiation Is Only Required If the Scope of Representation Includes Both Machine-to-Machine and User-Facing Services. When Representing a Sole Microservice Cluster, for Instance, This Differentation Becomes Unnecessary." n.d.).

Here, Service A represents a service that end-users interact with, while Service B represents an internal service shared across a *service landscape*.

Let us now represent the sharing of an authentication service within a *service landscape*:

We are representing the ability of each service nested inside a *service landscape* to access the authentication service by displaying top-down inheritance. Please note that just as *service landscapes* inherit their parent's attributes and pass them down to their respective children, the same is true for services: Here, Service A, B and C all can use the authentication service, while Service D, not being a child of the *service landscape*, can not.

Microservice architecture as a mirror of business capabilities

An established and common pattern in the design of microservice architecture is directly translating business capabilities such as displaying and shipping a product directly into system architecture(Richardson 2018). This entails manifesting core mechanisms as entirely seperated systems that can be developed, maintained and deployed seperately from one-another, allowing for a *single-responsibility* model as described in the Unix Philosophy (Chapter 2).

As part of our service model, we can define a simple process for distilling these core mechanisms using our existing concept of *resolutions* (atomic bits of functionality directly mapped against a user's needs).("Please Note That 'Service' in This Concept Does Not Refer to a 'Distributed Microservice' in a Pure Engineering Sense. Under This Model, an Imported Function in a Monolithic Architecture Can Be Described as a Service – When Code Calls a Function, It Does so Out of the Need for Functionality or Data, and the Function Performs to Provide an Outcome." n.d.)

1. Identify similar *needs* across services in a service landscape.
2. Design a single *resolution* that is capable of resolving all these similar *needs*.
3. Inject an internal service capable of performing the designed *resolution* so that all services which share the initial *need* are able to access it.

The introduction to this thesis was an attempt to highlight how extremely omnipotent *services* are in our daily lives — not *just* digital services like Spotify and Google, but basic needs such as roofs over our heads. However, for the scope of this paper, I chose to focus on digital services alone — the simple reason being the desired length of 60,000 characters.

What struck me over and over while writing this paper: Many of these concepts I thought about and wrote down after deep-diving into these four levels of my basic "service" model apply to a surprising amount of situations outside the digital realm.

Take this illustration from Chapter 1, for instance:

Let us

> This thesis aims to illustrate digital services in a unified model that can be applied to a service concept from the microscopic layers of so-called "atomic functionality" — small, dedicated mechanisms with a single responsibility — to the macroscopic layer that the defines the service and its performance itself, to "Service Landscapes"; a cluster of individual service models that can be used to express relationships between individual services. We will study each layer in detail and talk to experts in their respective fields in order to distill our model, attempting to deliver a coherent concept that can be used to design, define, analyze and optimize digital services.

A. Hunt, D. Thomas. 1999. "The Pragmatic Programmer — from Journeyman to Master." Addison Wesley.

Alphabet Inc. 2019. "Alphabet Inc. Investor Report." abc.xyz/investor/.

Amazon Inc. 2019a. "Amazon Ec2." aws.amazon.com/ec2/.

———. 2019b. "Amazon Managed Blockchain." aws.amazon.com/managed-blockchain/.

———. 2019c. "Amazon Polly." aws.amazon.com/polly/.

———. 2019d. "Amazon Redshift." aws.amazon.com/redshift/.

———. 2019e. "Amazon Redshift." aws.amazon.com/redshift/.

———. 2019f. "AWS Identity and Access Management – Getting Set up." docs.aws.amazon.com/IAM/latest/UserGuide/getting-set-up.html.

———. 2019g. "AWS – Explore Our Services." aws.amazon.com/products/.

———. 2019h. "AWS – Explore Our Services." aws.amazon.com/products/.

———. 2019i. "Cloud Computing with Amazon Web

Services." aws.amazon.com/what-is-aws/.

Amazon Inc., BusinessWire. 2018. "Amazon's Best of Prime 2017 Reveals the Year's Biggest Trends —More Than 5 Billion Items Shipped with Prime in 2017." businesswire.com/news/home/20180102005390/en/.

Andreesen, Marc. 2007. "EE204: Product/Market Fit." web.stanford.edu/class/ee204/ProductMarketFit.html.

Barashkov, Alex. 2018. "Real Time + Postgres = ?" dev.to/alex_barashkov/real-time--postgres---2j6d.

BOLD Online BV. 2019. "Boldking Homepage." boldking.com.

Chuang, A., and C. Gellings. 2009. "Demand-Side Integration for Customer Choice Through Variable Service Subscription." In *2009 Ieee Power Energy Society General Meeting*, 67. IEEE.

Clark, Courtney. 2018. "What Is a Design System?" forumone.com/ideas/what-is-design-system/.

Diggins, Christopher. 2011. "Principles of Good Programming." artima.com/weblogs/viewpost.jsp?thread=331531.

Endenburg, Martijn. 2016. "How Code Duplication Impacts Software Maintainability." linkedin.com/pulse/how-code-duplication-impacts-software-maintainability-endenburg/.

"For the Sake of Simplicity, the Set of Needs and Outcome Are Set to a Constant Size. at a Later Stage, We Will Look into Quantifying Their Size Further." n.d.

Fowler, Martin. 2003. "TechnicalDebt." martinfowler.com/bliki/TechnicalDebt.html.

Frost, Brad. 2013a. "Atomic Design." http://bradfrost.com/blog/post/atomic-web-design/.

———. 2013b. "Atomic Design." http://bradfrost.com/blog/post/atomic-web-design/.

———. 2013c. "Atomic Design." http://bradfrost.com/blog/post/atomic-web-design/.

———. 2013d. "Atomic Design." http://bradfrost.com/blog/post/atomic-web-design/.

———. 2013e. "Atomic Design." http://bradfrost.com/blog/post/atomic-web-design/.

Google LLC. 2019a. "Cloud Computing Services | Google Cloud." cloud.google.com.

———. 2019b. "Google Cloud – G Suite." gsuite.google.com.

———. 2019c. "Google Cloud – G Suite." gsuite.google.com.

———. 2019d. "Google Cloud — Docs." gsuite.google.com/products/docs/.

———. 2019e. "Google Drive: Free Cloud Storage for Personal Use." google.com/drive.

———. 2019f. "Google Play Books." play.google.com/store/apps/details?id=com.google.android.apps.books.

———. 2019g. "Google Play Movies." play.google.com/store/apps/details?id=com.google.android.videos.

———. 2019h. "Google Play Music." play.google.com/store/apps/details?id=com.google.android.music.

———. 2019i. "Google Privacy Policy." https://policies.google.com/privacy.

———. 2019j. "Google — Our Products." about.google/intl/en/products/.

———. 2019k. "Manage Teaching and Learning with Classroom." edu.google.com/products/classroom/.

———. 2019l. "Pay for Whatever, Whenever with Google Pay." pay.google.com/about/.

Intercom Inc. 2019. "A New and Better Way to Acquire, Engage and Retain Customers." intercom.com.

Intercom, Inc. 2019. "The Best Team Inbox for Sales and Support." intercom.com/inbox.

International Organization for Standardization. 2009a. "Ergonomics of Human System Interaction:

Human-Centered Design for Interactive Systems." International Organization for Standardization.

———. 2009b. "Human-Centered Design for Interactive Systems." International Organization for Standardization.

"Interview with B. Oxendine"a.

"Interview with B. Oxendine"b.

"Interview with B. Oxendine"c.

"Interview with B. Oxendine"d.

"Interview with B. Oxendine"e.

"Interview with B. Oxendine"f.

Iqbal, M. 2018a. "Thinking in Services." In. BIS Publishers.

———. 2018b. "Thinking in Services." BIS Publishers.

KPI.org, a Strategy Management Group company. 2019. "What Is a Key Performance Indicator (Kpi)?" kpi.org/KPI-Basics.

N26 GmbH. 2019a. "A Bank Account with Features Built for Real People." n26.com/en-de/bank-account.

———. 2019b. "How to Get a Loan?" support.n26.com/en-de/get-more-out-of-n26/credit/how-to-get-a-loan.

———. 2019c. "N26 Credit – Turn Next Year's Dreams into Today's Reality." n26.com/en-de/credit.

———. 2019d. "N26 Savings – Open a Savings Account on Your Phone." n26.com/en-de/savings.

———. 2019e. "Your Dreams Are Big. They Need Some Space." n26.com/en-de/spaces.

———. 2019f. "Your Dreams Are Big. They Need Some Space." n26.com/en-de/spaces.

Peter G. Neumann. 2004. "Principled Assuredly Trustworthy Composable Architectures." ACDRL.

"Please Note That This Differentiation Is Only Required If the Scope of Representation Includes Both Machine-to-Machine and User-Facing Services. When Representing a Sole Microservice Cluster, for Instance, This Differentation Becomes Unnecessary." n.d.

"Please Note That 'Service' in This Concept Does Not Refer to a 'Distributed Microservice' in a Pure Engineering Sense. Under This Model, an Imported Function in a Monolithic Architecture Can Be Described as a Service – When Code Calls a Function, It Does so Out of the Need for Functionality or Data, and the Function Performs to Provide an Outcome." n.d.

Richardson, C. 2018. "Pattern: Decompose by Business Capability." microservices.io/patterns/decomposition/decompose-by-business-capability.html.

Salesforce.com, inc. 2019. "Lightning Design System — Design Tokens." lightningdesignsystem.com/design-tokens/.

Spinellis, Diomidis. 2006. "The Bad Code Spotter's Guide." informit.com/articles/article.aspx?p=457502&seqNum=5.

Statista. 2019. "Mobile Phone Internet User Penetration Worldwide from 2014 to 2019." statista.com/statistics/284202/mobile-phone-internet-user-penetration-worldwide/.

Swearingen, J. 2000a. "Operations Management – Characteristics of Services." University of Illionois.

———. 2000b. "Operations Management – Characteristics of Services." University of Illionois – Lectures 2000.

———. 2000c. "Operations Management – Characteristics of Services." University of Illionois – Lectures 2000.

———. 2000d. "Operations Management – Characteristics of Services." University of Illionois – Lectures 2000.

The European Parliament and the Council of the European Union. 2009. "REGULATION (Ec) No 924/2009 of the European Parliament and of the Council." Official Journal of the European Union.

Thomson, Thane. 2016. "UX and System Architecture.” thanethomson.com/2016/07/12/ux-and-system-architecture/.

Uber Technologies Inc. 2019. “How Uber Eats Works.” about.ubereats.com.

Wolak, Harris, Kalafatis. 1998a. “An Investigation into Four Characteristics of Services.” *Journal of Empirical Generalisations in Marketing Science* 3: 25–27.

———. 1998b. “An Investigation into Four Characteristics of Services.” *Journal of Empirical Generalisations in Marketing Science.*

YouTube Llc. 2019a. “YouTube Experiences.” youtube.com/yt/about/experiences/.

———. 2019b. “YouTube – Brand Resources.” www.youtube.com/yt/about/brand-resources.