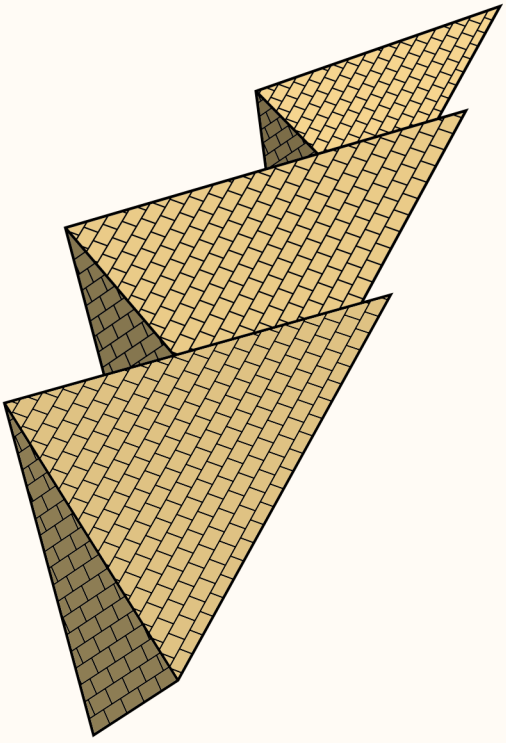
Design and Reality

Essays on Software Design



Rebecca Wirfs-Brock
and Mathias Verraes

Design and Reality

Essays on Software Design

Rebecca Wirfs-Brock and Mathias Verraes

This book is for sale at http://leanpub.com/design-and-reality

This version was published on 2022-10-04

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Rebecca Wirfs-Brock and Mathias Verraes by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading Design and Reality - Essays on Software Design by @mathiasverraes and @rebeccawb

Contents

About the Authors

Rebecca and Mathias met at the Domain-Driven Design Europe 2017 conference in Amsterdam, and started collaborating in 2021. This book collects soe of their essays.

Mathias Verraes

Mathias Verraes is the founder of Aardling, a software modelling & design consultancy, with a penchant for complex environments. His focus is on design strategy and messaging-centric domain modelling. Since leaving a lead developer job in 2011 and moving to consulting in 2011, Mathias has worked with clients in Finance, Government, Supply Chain, Mobility, Energy, E-Commerce, and more.

Mathias writes about software design at verraes.net since 2011. As a speaker, he's been at many major conferences such as NDC and Goto, and has been a keynote speaker DDD eXchange, ExploreDDD, KanDDDinsky, and others. Occasionally, he teaches courses on Domain-Driven Design & messaging architecture. Mathias is also the founder of the DDD Europe conference.

Mathias has a Masters in Music from the Royal Conservatory of Ghent, and is an autodidact on software. When he's at home in Kortrijk, Belgium, he helps his two sons build crazy Lego contraptions.

Rebecca Wirfs-Brock

Rebecca Wirfs-Brock is a software design pioneer who invented the set of object design practices known as Responsibility-Driven

Design (RDD) and popularized the x-Driven Design meme (RDD, TDD, DDD, BDD). She is an internationally recognized leader in the development of effective software design and architecture techniques. Among her widely used innovations are use case conversations and object role stereotypes. She was the design columnist for IEEE Software and the author of two influential texts, Designing Object-Oriented Software, and Object Design: Roles, Responsibilities and Collaborations.

In her work, she helps teams hone their design and architecture skills, manage and reduce technical debt, and address architecture risks. Although best known as a software design guru, she is also known as an innovator of techniques for simply expressing complex requirements and effectively developing and communicating software architecture. Her research interests focus on the cognitive and social aspects of software development including: Naturalistic decision-making (NDM) and software architecture decisions; decision-making models for software architects; design heuristics and their relationships to software patterns, guidelines; values and practices for sustainable software architecture and its evolution; software design and development ethnography; practical software design methods; agile architecture and design practices; patterns and pattern languages; object-oriented design; software modeling; domain modeling; documenting complex software systems; and communicating design intentions.

Design & Reality

> "The transition to a really deep model is a profound shift in your thinking and demands a major change to the design." — Domain-Driven Design[1], Eric Evans

There is a fallacy about how domain modelling works. The misconception is that we can design software by discovering all the relevant concepts in the domain, turn them into concepts in our design, add some behaviours, and voilà, we've solved our problem. It's a simplistic perception of how design works: a linear path from A to B:

1. understand the problem,
2. apply design,
3. end up with a solution.

That idea was so central to early Object-Oriented Design, that one of us (Rebecca) thought to refute it in her book:

> "Early object design books including [my own] Designing Object-Oriented Software [Wirfs-Brock 1990], speak of finding objects by identifying things (noun phrases) written in a design specification. In hindsight, this approach seems naive. Today, we don't advocate underlining nouns and simplistically modeling things in the real world. It's much more complicated than that. Finding good objects means identifying abstractions that are part of your application's domain and its execution machinery. Their correspondence to real-world

[1] https://amzn.to/3ljYdp5

> things may be tenuous, at best. Even when modeling domain concepts, you need to look carefully at how those objects fit into your overall application design."
> — Object Design[2], Rebecca Wirfs-Brock

Domain language vs Ubiquitous Language

The idea has persisted in many naive interpretations of Domain-Driven Design as well. Domain language and Ubiquitous Language are often conflated. They're not the same.

Domain language is what is used by people working in the domain. It's a natural language, and therefore messy. It's organic: concepts are introduced out of necessity, without deliberation, without agreement, without precision. Terminology spreads across the organisation or fades out. Meaning shifts. People adapt old terms into new meanings, or terms acquire multiple, ambiguous meanings. It exists because it works, at least well enough for human-to-human communication. A domain language (like all language) only works in the specific context it evolved in.

For us system designers, messy language is not good enough. We need precise language with well understood concepts, and explicit context. This is what a Ubiquitous Language is: a constructed, formalised language, agreed upon by stakeholders and designers, to serve the needs of our design. We need more control over this language than we have over the domain language. The Ubiquitous Language has to be deeply connected to the domain language, or there will be discord. The level of formality and precision in any Ubiquitous Language depends on its environment: a meme sharing app and an oil rig control system have different needs.

[2] https://www.informit.com/promotions/object-design-142314

Drilling Mud

Talking of oil rigs:

Rebecca was invited to consult for a company that makes hardware and software for oil rigs. She was asked to help with object design and modelling, working on redesigning the control system that monitors and manages sensors and equipment on the oil rig. Drilling causes a lot of friction, and "drilling mud" (a proprietary chemical substance) is used as a lubricant. It's also used as a carrier for the rocks and debris you get from drilling, lifting it all up and out of the hole. Equipment monitors the drilling mud pressure, and by changing the composition of the mud during drilling, you can control that pressure. Too much pressure is a really bad thing.

And then an oil rig in the gulf exploded.

As the news stories were coming out, the team found out that the rig was using a competitor's equipment. Whew! The team started speculating about what could have happened, and were thinking about how something like that could happen with their own systems. Was it faulty equipment, sensors, the telemetry, communications between various components, the software?

Scenarios

When in doubt, look for examples. The team ran through scenarios. What happens when a catastrophic condition occurs? How do people react? When something fails, it's a noisy environment for the oil rig engineers: sirens blaring, alarms going off, … We discovered that when a problem couldn't be fixed immediately, the engineers, in order to concentrate, would turn off the alarms after a while. When a failure is easy to fix, the control system logs reflect that the alarm went on and was turned off a few minutes later.

But for more consequential failures, even though these problems take much longer to resolve, it still shows up on the logs as being resolved within minutes. Then, when people study the logs, it looks like the failure was resolved quickly. But that's totally inaccurate. This may look like a software bug, but it's really a flaw in the model. And we should use it as an opportunity to improve that model.

The initial modelling assumption is that alarms are directly connected to the emergency conditions in the world. However, the system's perception of the world is distorted: when the engineers turn off the alarm, the system believes the emergency is over. But it's not, turning an alarm off doesn't change the emergency condition in the world. The alarms are only indirectly connected to the emergency. If it's indirectly connected, there's something else in between, that doesn't exist in our model. The model is an incomplete representation of a fact of the world, and that could be catastrophic.

A Breakthrough

The team explored scenarios, specifically the weird ones, the awkward edge cases where nobody really knows how the system behaves, or even how it should behave. One such scenario is when two separate sensor measurements raise alarms at the same time. The alarm sounds, an engineer turns it off, but what happens to the second alarm? Should the alarm still sound or not? Should turning off one turn off the other? If it didn't turn off, would the engineers think the off switch didn't work and just push it again?

By working through these scenarios, the team figured out there was a distinction between the alarm sounding, and the state of alertness. Now, in this new model, when measurements from the sensors exceed certain thresholds or exhibit certain patterns, the system doesn't sound the alarm directly anymore. Instead, it raises an alert condition, which is also logged. It's this alert condition

that is associated with the actual problem. The new alert concept is now responsible for sounding the alarm (or not). The alarm can still be turned off, but the alert condition remains. Two alert conditions with different causes can coexist without being confused by the single alarm. This model decouples the emergency from the sounding of the alarm.

The old model didn't make that distinction, and therefore it couldn't handle edge cases very well. When at last the team understood the need for separating alert conditions from the alarms, they couldn't unsee it. It's one of those aha-moments that seem obvious in retrospect. Such distinctions are not easily unearthed. It's what Eric Evans calls a Breakthrough.

An Act of Creation

There was a missing concept, and at the first the team didn't know something was missing. It wasn't obvious at first, because there wasn't a name for "alert condition" in the domain language. The oil rig engineers' job isn't designing software or creating a precise language, they just want to be able to respond to alarms and fix problems in peace. Alert conditions didn't turn up in a specification document, or in any communication between the oil rig engineers. The concept was not used implicitly by the engineers or the software; no, the whole concept did not exist.

Then where did the concept come from?

People in the domain experienced the problem, but without explicit terminology, they couldn't express the problem to the system designers. So it's us, the designers, who created it. It's an act of creative modelling. The concept is invented. In our oil rig monitoring domain, it was a novel way to perceive reality.

Of course, in English, alert and alarm exist. They are almost synonymous. But in our Ubiquitous Language, we agreed to make

them distinct. We designed our Ubiquitous Language to fit our purpose, and it's different from the domain language. After we introduced "alert conditions", the oil rig engineers incorporated it in their language. This change in the domain is driven by the design. This is a break with the linear, unidirectional understanding of moving from problem to solution through design. Instead, through design, we reframed the problem.

Is it a better model?

How do we know that this newly invented model is in fact better (specifically, more fit for purpose)? We find realistic scenarios and test them against the alert condition model, as well as other candidate models. In our case, with the new model, the logs will be more accurate, which was the original problem.

But in addition to helping with the original problem, a deeper model often opens new possibilities. This alert conditions model suggests several:

- Different measurements can be associated with the same alert.
- Alert conditions can be qualified.
- We can define alarm behaviours for simultaneous alert conditions, for example by spacing the alarms, or picking different sound patterns.
- Critical alerts could block less critical ones from hogging the alarm.
- Alert conditions can be lowered as the situation improves, without resolving them.
- …

These new options are relevant, and likely to bring value. Yet another sign we'd hit on a better model is that we had new

conversations with the domain experts. A lot of failure scenarios became easier to detect and respond to. We started asking, what other alert conditions could exist? What risks aren't we mitigating yet? How should we react?

Design Creates New Realities

In a world-centric view of design, only the sensors and the alarms existed in the real world, and the old software model reflected that accurately. Therefore it was an accurate model. The new model that includes alerts isn't more "accurate" than the old one, it doesn't come from the real world,it's not more realistic, and it isn't more "domain-ish". But it is more useful. Sensors and alarms are objective, compared to alert conditions. Something is an alert condition because in this environment, we believe it should be an alert condition, and that's subjective.

The model works for the domain and is connected to it, but it is not purely a model *of* the problem domain. It addresses the problems in the contexts we envision better. The solution clarified the problem. Having only a real world focus for modelling blinds us to better options and innovations.

These creative introductions of novel concepts into the model are rarely discussed in literature about modelling. Software design books talk about turning concepts into types and data structures, but what if the concept isn't there yet? Forming distinctions, not just abstractions, however, can help clarify a model. These distinctions create opportunities.

The model must be radically about its utility in solving the problem.

> "Our measure of success lies in how clearly we invent a software reality that satisfies our application's

> requirements—and not in how closely it resembles the real world." — Object Design[3], Rebecca Wirfs-Brock

[3] https://www.informit.com/promotions/object-design-142314

Models & Metaphors

One of us (Mathias) consulted for a client that acted as a broker for paying copyright holders for the use of their content. To do this, they figured out who the copyright holders of a work were. Then they tracked usage claims, calculated the amounts owed, collected the money, and made the payments. Understanding who owned what was one of the trickier parts of their business.

— "It's just a technical problem."

— "But nobody really understands how it works!"

— "Some of us understand most of it. It just happens to be a complicated problem."

— "Let's do a little bit of modelling anyway."

A few weeks later, we had a rich model. It had crisp concepts, and was easier to understand. Business stakeholders started to pay attention, and became actively involved in modelling. They saw the big idea, and could describe what they wanted. Here's how that happened.

Case Study

Determining ownership was a complicated data matching process which pulled data from a number of data sources:

- Research done by the company itself
- Offshore data cleaning
- Publicly available data from a wiki-style source
- Publicly available, curated data

- Private sources, for which the company paid a licence fee
- Direct submissions from individuals
- Agencies representing copyright holders

The company had a data quality problem. Because of the variety of data sources, there wasn't a single source of truth for any claim. The data was often incomplete and inconsistent. On top of that, there was a possibility for fraud: bad actors claimed ownership of authors' work. Most people acted in good faith. Even then, the data was always going to be messy, and it took considerable effort to sort things out. The data was in constant flux: even though the ownership of a work rarely changes, the data did.

Data Matching

The engineers were always improving the "data matching". That's what they called the process of reconciling the inconsistencies, and providing a clear view on who owned what and who had to pay whom. They used EventSourcing, and they could easily replay new matching algorithms on historic data. The data matching algorithms matched similar claims on the same works in the different data sources. When multiple data sources concurred, the match succeeded.

Initially, when most sources concurred on a claim, the algorithm ignored a lone exception. When there was more contention about a claim, it was less obvious what to do. The code reflected this lack of clarity. Later the team realised that a conflicting claim could tell them more: It was an indicator of the messiness of the data. If they used their records of noise in the data, they could learn about how often different data sources, parties, and individuals agreed on successful claims, and improve their algorithm.

For example, say a match was poor: 50% of sources point to one owner and 50% point to another owner. Based on that information

alone, it's impossible to decide who the owner is. But by using historical data, the algorithm could figure out which sources had been part of successful matches more often. They could give more weight to these sources, and tip the scales in one direction or the other. This way, even if 50% of sources claim A as the owner and 50% claim B, an answer can be found.

Domain Modelling

The code mixed responsibilities: pulling data, filtering, reformatting, interpreting, and applying matching rules. All the cases and rules made the data matching very complicated. Only a few engineers knew how it worked. Mathias noticed that the engineers couldn't explain how it worked very well. And the business people he talked to were unable to explain anything at all about how the system worked. They simply referred to it as the "data matching". The team wasn't concerned about this. In their eyes, the complexity was just something they had to deal with.

Mathias proposed a whiteboard modelling session. Initially, the engineers resisted. After all, they didn't feel this was a business domain, just a purely technical problem. However, Mathias argued, the quality of the results determined who got paid what, and mistakes meant customers would eventually move to a competitor. So even if the data matching was technical, it performed an essential function in the Core Domain. The knowledge about it was sketchy, engineering couldn't explain it, business didn't understand it. Because of that, they rarely discussed it, and when they did, it was in purely technical terms. If communication is hard, if conversations are cumbersome, you lack a good shared model.

Through modelling, the matching process became less opaque to the engineers. We made clearer distinctions between different steps to pull data, process it, identifying a match, and coming to a decision. The model included sources, claims, reconciliations,

exceptions. We drew the matching rules on the whiteboard as well, making those rules explicit first class concepts in the model. As the matching process became clearer, the underlying ideas that led to the system design started surfacing. From the "what", we moved to the "why". This put us in a good position to start discovering abstractions.

Trust

Gradually, the assumptions that they built the algorithm on, surfaced in the conversations. We stated those assumptions, wrote them on stickies and put them on the whiteboard. One accepted assumption was that when a data source is frequently in agreement with other sources, it is less likely to be wrong in the future. If a source is more reliable, it should be trusted more, therefore claims from that source pulled more weight in the decision of who has a claim to what. When doing domain discovery and modelling, it's good to be observant, and listen to subtleties in the language. Words like "reliable, "trust", "pull more weight", and "decision" were being used informally in these conversations. What works in these situations, is to have a healthy obsession with language. Add this language to the whiteboard. Ask questions: What does this word mean, in what context do you use it?

Through these discussions, the concept of "trust" grew in importance. It became explicit in the whiteboard models. It was tangible: you could see it, point to it, move it around. You could start telling stories about trust. Why would one source be more trusted? What would damage that trust? What edge cases could we find that would affect trust in different ways?

Trust as an Object

During the next modelling session, we talked about trust a lot. From a random word that people threw into the conversation, it had morphed into a meaningful term. Mathias suggested a little thought experiment: What if Trust was an actual object in the code? What would that look like? Quickly, a simple model of Trust emerged. Trust is a Value Object, and its value represents the "amount" of trust we have in a data source, or the trust we have in a claim on a work or usage, or the trust we have in the person making the claim. Trust is measured on a scale of -5 to 5. That number determines whether a claim is granted or not, whether it needs additional sources to confirm it, or whether the company needs to do further research.

It was a major mindshift.

The old code dynamically computed similar values to determine "matches". These computations were spread and duplicated across the code, hiding in many branches. The team didn't see that all these values and computations were really aspects of the same underlying concept. They didn't see that the computations could be shared, whether you're matching sources, people, or claims. There was no shared abstraction.

But now, in the new code, those values are encapsulated in a first class concept called Trust objects. This is where the magic happens: we move from a whiteboard concept, to making Trust an essential element in the design. The team cleaned up the ad hoc logic spread across the data matching code and replaced it with a single Trust concept.

Trust entered the Ubiquitous Language. The idea that degrees of Trust are ranked on a scale from -5 to 5, also became part of the language. And it gave us a new way to think about our Core Domain: We pay owners based on who earns our Trust.

Trust as a Process

The team was designing an EventSourced system, so naturally, the conversation moved to what events could affect Trust. How does Trust evolve over time? What used to be matching claims in the old model, now became events that positively or negatively affected our Trust in a claim. Earning Trust (or losing it) was now thought of as a process. A new claim was an event in that process. Trust was now seen as a snapshot of the Trust earning process. If a claim was denied, but new evidence emerged, Trust increased and the claim was granted. Certain sources, like the private databases that the company bought a licence for, were highly trusted and stable. For others, like the wiki-style sources where people could submit claims, Trust was more volatile.

Business Involvement

During the discussions about the new Trust and Trust-building concepts, the team went back to the business regularly to make sure the concepts worked. They asked for their insights into how they should assign Trust, and what criteria they should use. We saw an interesting effect: people in the business became invested in these conversations and joined in modelling sessions. Data matching faded from the conversations, and Trust took over. There was a general excitement about being able to assign and evolve Trust. The engineers' new model became a shared model within the business.

Trust as an Arithmetic

The copyright brokerage domain experts started throwing scenarios at the team: What if a Source A with a Trust of 0 made a claim that

was corroborated by a Source B with a Trust of 5? The claim itself was now highly trusted, but what was the impact on Source A? One swallow doesn't make Spring, so surely Source A shouldn't be granted the same level of Trust as Source B. A repeated pattern of corroborated Trust on the other hand, should reflect in higher Trust for Source A.

During these continued explorations, people from the business and engineering listed the rules for how different events impacted Trust, and coding them. By seeing the rules in code, a new idea emerged. Trust could have its own arithmetic: a set of rules that defined how Trust was accumulated. For example, a claim with a Trust of 3, that was corroborated by a claim with a Trust of 5, would now be assigned a new Trust of 4. The larger set of arithmetics addressed various permutations of claims corroborating claims, sources corroborating sources, and patterns of corroboration over time. The Trust object encapsulated this arithmetic, and managed the properties and behaviours for it.

From an anaemic Trust object, we had now arrived at a richer model of Trust that was responsible for all these operations. The team came up with polymorphic Strategy objects. These allowed them to swap out different mechanisms for assigning and evolving Trust. The old data matching code had mixed fetching and storing information with the sprawling logic. Now, the team found it easy to separate it into a layer that dealt with the plumbing separate from the clean Trust model.

The Evolution of the Model

In summary, this was the evolution:

1. Ad hoc code that computes values for matches.
2. Using Trust in conversations that explained how the current system worked.

3. Trust as a Value Object in the code.
4. Evolving Trust as a process, with events (such as finding a matching claim) that assigned new values of Trust.
5. Trust as a shared term between business and engineering, that replaced the old language of technical data matching.
6. Exploring how to assign Trust using more real-world scenarios.
7. Building an arithmetic that controls the computation of Trust.
8. Polymorphic Strategies for assigning Trust.

When you find a better, more meaningful abstraction, it becomes a catalyst: it enables other modelling constructs, allowing other ideas to form around that concept. It takes exploration, coding, conversations, trying scenarios, … There's no golden recipe for making this happen. You need to be open to possibility, and take the time for it.

Discussion

The engineers originally introduced the concept of "matching", but that was an anaemic description of the algorithm itself, not the purpose. "If this value equals that value, do this". Data matching was devoid of meaning. That's what Trust introduces: conceptual scaffolding for the meaning of the system. Trust is a magnet, an attractor for a way of thinking about and organising the design.

Initially, the technical details of the problem were so complicated, and provided such interesting challenges to the engineers, that that was all they talked about with the business stakeholders. Those details got in the way of designing a useful Ubiquitous Language. The engineers had assumed that their code looked the way it needed to look. In their eyes, the code was complex because the problem of matching was complex. The code simply manifested that complexity. They didn't see the complexity of that code as

a problem in its own right. The belief that there wasn't a better model to be found, obscured the Core Domain for both business and engineering.

The domain experts were indeed experts in the copyrights domain, and had crisp concepts for ownership, claims, intellectual property, the laws, and the industry practices. But that was not their Core Domain. The real Core was the efficient, automated business they're trying to build out of it. That was their new domain. That explains why knowledge of copyright concepts alone wasn't sufficient to make a great model.

Before they developed an understanding of Trust, business stakeholders could tell you detailed stories about how the system should behave in specific situations. But they had lacked the language to talk about these stories in terms of the bigger idea that governs them. They were missing crisp concepts for them.

Good Metaphors

We moved from raw code, to a model based on the new concept of Trust. But what kind of thing is this Trust concept? Trust is a metaphor.[4] Actual trust is a human emotion, and partly irrational. You trust someone instinctively, and for entirely subjective reasons that might change. Machines don't have these emotions. We have an artificial metric in our system, with algorithms to manipulate it, and we named it Trust. It's a proxy term.

This metaphor enables a more compact conversation, as evidenced by the fact that engineers and domain experts alike can discuss Trust without losing each other in technical details. A sentence like "The claims from this source were repeatedly confirmed by other sources" was replaced by "This source has built up trust", and all knew what that entailed.

[4] One of the authors published a book about a sixth stage in 2020.

The metaphor allows us to handle the same degree of complexity, but we can reason about determining Trust without having to understand every detail at the point where it's used. For those of us without Einstein brains, it's now a lot easier to work on the code, it lowers the cognitive load.

A good metaphor in the right context, such as Trust, enables us to achieve things we couldn't easily do before. The team reconsidered a feature that would allow them to swap out different strategies for matching claims. Originally they had dismissed the idea, because, in the old code, it would have been prohibitively expensive to build. It would have resulted in huge condition trees and sprawling dependencies on shared state. They'd have to be very careful, and it would be difficult to test that logic. With the new model, swapping out polymorphic Strategy objects is trivial. The new model allows testing low level units like the Trust object, higher level logic like the Trust-building process, and individual Claims Strategies, with each test remaining at a single level of abstraction.

Our Trust model not only organises the details better, but it is also concise. We can go to a single point in the code and know how something is determined. A Trust object computes its own value, in a single place in the code. We don't have to look at twenty different conditionals across the code to understand the behaviour; instead we can look at a single strategy. It's much easier to spot bugs, which in turn helps us make the code more correct.

A good model helps you reason about the behaviour of a system. A good metaphor helps you reason about the desired behaviour of a system.

The Trust metaphor unlocked a path to tackle complexity. We discovered it by listening closely to the language used to describe the solution, using that language in examples, and trying thought experiments. We're not matching data anymore, we're determining Trust and using it to resolve claims. Instead of coding the rules, we're now encoding them. We're better copyright brokers because

of this.

Bad Metaphors

Be wary of bad, ill-fitting metaphors. Imagine the team had come with Star Ratings as the metaphor. Sure, it also works as a quantification, but it's based on popularity, and calculates the average. We could still have built all the same behaviour of the Trust model, but with a lot of bizarre rules, like "Our own sources get 20 five-star ratings". When you notice that you have to force-fit elements of your problem space into a metaphor, and there's friction between what you want to say and what that metaphor allows you to say, you need to get rid of it. No metaphor will make a perfect fit, but a bad metaphor leads you into awkward conversations without buying you clarity.

To make things trickier, whenever you introduce a new metaphor, it can be awkward at first. In our case study, Trust didn't instantly become a fully explored and accepted metaphor. There's a delicate line between the early struggles of adopting a new good metaphor, and one that is simply bad. Keep trying, work on using your new metaphor, see if it buys you explanatory power, and don't be afraid to drop it if it does not.

And sometimes, there simply isn't any good metaphor, or even a simpler model to be found. In those cases, you just have to crunch it. There's no simplification to be found. You just have to work out all the rules, list all the cases, and deal with the complexity as is.

Conclusion

To find good metaphors, put yourself in a position where you'll notice them in conversation. Invite diverse roles into your design

discussions. Have a healthy obsession with language: what does this mean? Is this the best way to say it? Be observant about this language, listen for terms that people say off the cuff. Capture any metaphors that people use. Reinforce them in conversations, but be ready to drop them if you feel you have to force-fit them. Is a metaphor bringing clarity? Does it help you express the problem better? Try scenarios and edge cases, even if they're highly unlikely. They'll teach you about the limits of your metaphor. Then distil the metaphor, agree on a precise meaning. Use it in your model, and then translate it to your code and tests. Metaphors are how language works, how our brains attach meaning, and we're using that to our advantage.

Critically Engaging with Models

You might be familiar with The Five Stages of Grief aka the Kübler-Ross model for processing grief (denial, anger, bargaining, depression, and acceptance). Independent of what the authors intended, this model gives us a framing right off the bat:

- The word "Five" suggests a fixed set.[5]
- The word "Stages" suggests that they are discrete phases.
- It also suggests that these stages come in a fixed order.
- The word "The" suggests that you always go through these five stages when grieving.
- And it is our impression that people interpret this model as prescriptive: in order to process grief, you *must* go through these stages in order.

We're not interested here in whether these statements are accurate or not. Perhaps the authors' wording was accidental, and it could just as well have been named "Various Feelings of Grief". We're interested in how the language suggests ordered stages, and that's how most people will perceive it.

When a model puts us in a certain framing, how can we tell? How do we understand what framing a model imposes on us? How do we engage with any model? How can we evaluate it? Should we break out of that framing? Do we accept the model as is? Or do we use it as scaffolding for finding better models, that are better suited to our situation?

[5] One of the authors published a book about a sixth stage in 2020.

Models Are Worldviews

Models, whether for a software system, a development process, diseases, political systems, or otherwise, are a way to look at (a part of) the world. They explain how something behaves and how to modify that behaviour. Every model makes a choice about what is important, what categories we classify things in, what we see, what's invisible, what's valued, or even what's valid. Models are reductionist; that is, they only show a selection of the subject they're describing, and lose something in the process. And models are biased: they implicitly reflect the assumptions, constraints, and values of the model's author.

Most of the time, when you adopt a model created by someone else, you assimilate it into your worldview without much thought. You acquire a new way of seeing something and accept it. It's how learning works. However, when you do that, you may not understand the model's limitations.

Models mess with you. They impose a distinct perspective, and a set of rules for operating within them. They frame how you look at your problem. And you're usually not aware of this perspective and framing.

But you can choose to look at a model more intentionally. If you're looking at a model for the first time, you can use your fresh perspective to see what it includes and what it leaves out. You can critically assess whether that model fits your view and your needs. Models are a powerful lens for perceiving a subject, and you should be deliberate when wielding them.

We'd like to share some tools for critically evaluating any models that come your way. We'll do this by example. We have picked a number of organisational models to illustrate how to examine them and see whether they offer what you need to solve your problems. First, we'll discuss the hierarchical, social network, and the value creation models. Then, we'll examine three organisational models

that are specific to software development: the Spotify Model, the Agile Fluency Model, and Team Topologies.

Our goal is not to voice an opinion one way or the other about these different models, or to suggest which ones to use for your organisation. Our interpretation of these models is by necessity brief. If you are familiar with any of them, you might feel we do them a great injustice by our summarizations. We ask that you look past the specifics of the organisational models we examine, and instead focus on the methods we demonstrate for evaluating any model, in any context.

Here we use well-known published models to illustrate approaches for understanding and comparing them. But these techniques apply equally well to various models that you may encounter or create, such as domain models, working agreements, business processes, or politics.

Hierarchies and Networks

A traditional way of looking at organisations is the hierarchical model. In this model, power is concentrated at the "top" and authority is structured into levels. People talk about "going up the chain of command" in order to reach the "right person" in the hierarchy who has the authority to make a decision. People have often mistaken this model for the whole: "an organisation *is* a hierarchy". In reality, the model only describes one aspect of an organisation.

It's often by looking at alternative models, that we see the limitations of this hierarchical model. For example, a competing model says: "an organisation consists of a hierarchical control structure *and* a social network". The social network is who you talk to, and how you informally acquire information you learn at the coffee machine or over lunch. This model is an improvement over the

hierarchical model: it shows us that human interactions can affect the functions of the organisation. A manager might want to encourage their organisation's social network, or change the dynamics to improve opportunities for informal interactions. They'll try to improve the functions of the organisation, and make sure the social network doesn't hinder it.

In our role of model evaluators, this teaches us that an organisational model can have two networks operating simultaneously, each representing a different function of the organisation. That's significant, because if our organisation has two networks, it is possible to have more.

> **Heuristic**
> If there are two ways of looking at something, look for a third.

Another model augments these two organisational networks with a third network: the value creation network. This is the network you turn to to produce meaningful outcomes. It's knowing which people to ask when you need something done, who the experts are, who can execute it. By modelling our organisation now as a hierarchy, a social network, *and* a value creation network, we're challenging the assumptions made by the previous two models. The new model says: "value creation doesn't happen through the same channels as control or gossip".

Before we adopted this third organisational model, there were two possibilities: either value creation was not on our radar at all, or value creation was a responsibility of one of the other two networks (most likely the hierarchy).

When we accept this third model, we believe that value creation:

- is important enough to be recognized as an organisational responsibility;

- doesn't happen in a hierarchy or social network;
- it happens in its own network.

This new model doesn't simply add a third network. The new model supplants the prior belief that value is created in the hierarchy, with the belief that it is created through a value creation network. Through the addition of the third network, the other two networks in the model have acquired a new purpose. The third network adjusts your perception of the other networks.

Say a manager has been mandating that all communications and decision-making happens along the lines of the hierarchy. Now, as they accept the new model, they change how they operate, and instead allow and encourage social and value creation networks to thrive. The manager will no longer demand that all communications take a certain shape, and will choose not the exert control over all aspects of communication.

None of these three models capture reality. Instead, they capture beliefs about reality that, in this case, drives business decision-making. Each model alters the way we perceive reality, and how we take action based on that perception. By accepting any model, we accept the belief system that comes along with it, and we change our behaviour accordingly.

Heuristics

From our short analysis of three organisational models, we can derive several heuristics for helping to evaluate any organisational model:

> **Heuristic**
> Compare different models to figure out what one model adds or omits, emphasizes, or downplays.

That is, use one model to discover if there are any gaps in the other. Look for ways one model might give a lot of attention to one aspect and not to another.

> **Heuristic**
> Understand the underlying belief system that comes with the model.

If I accept this model, how does it change my belief about part of the world that this model addresses?

> **Heuristic**
> Determine whether the model confirms my existing belief system and values, or conflicts with them.

Am I choosing models based on how well they conform to something I already believe?

> **Heuristic**
> Understand whether the model addresses or solves a problem that I'm interested in solving.

It may be an appealing model, but do I really need it?

> **Heuristic**
> Ascertain whether the model points me to problems I might have but haven't yet considered.

We find these heuristics are useful for critically evaluating any model, not just organisational models. In this text, however, we'll stick to organisational models.

Models Distract

As we have shown, a model creates a lens, a way of looking at the problem. Additionally, models often come with a richer set of guidelines or instruction sets, that tell you how to use them.

The organisational hierarchy model, for example, comes with advice on how to achieve a more desirable organisational structure, such as "Don't have more than 6 levels", or "Have between 5-30 direct reports per manager", or "When your organisation gets too big, split it along these functions". These guidelines are congruent with the building blocks of the hierarchical model and with the belief system that underlies it.

Guidelines are useful for implementing this model. Whenever you embrace a model, its instructions lead you to considerations that it views as important. It focuses you on the changes that it wants you to make. It's saying "this is an important thing to worry about". The things it finds important have names: the levels, the functions, the reports, … Later, we'll look at models that don't include names for levels and reports, but do include names for teams, value streams, fluency, or evolution. The choices of things a model names are what puts us in a frame.

But by discussing "How many levels do we need?" we're distracted from asking the deeper question of "Do we need levels at all?". Because the hierarchy model comes with a name for levels, it imposes the need for levels.

The methods and tools that come with any model are insidious in that sense. The presuppositions that a model makes are:

- The model is adequate (an organisation is indeed a hierarchy);
- The tools and techniques are adequate (choosing the right levels is the right solution for organisational problems);
- If you fail, you haven't applied them well (choosing not enough or too many levels).

Models rarely include a technique for determining whether the model fits your environment.

If we look at the flat organisational model, we see that it also makes some presuppositions (which conflict with the hierarchy model):

- The model is adequate (an organisation should indeed be flatter);
- The tools and techniques are adequate (removing middle management is the right solution to improve your organisation);
- If you fail, you haven't applied them well (you've allowed too much of the old hierarchy to survive).

But, again, if we only discuss techniques for removing middle management, we can forget to ask if any functions of middle management are not being handled with a flat organization model. The flat model doesn't tell us to look for valid functions of middle management, and it doesn't provide a pattern language for understanding these functions.

Models help you ask questions, but not necessarily the right ones. If we compare different models however, we can take the questions from one model, and ask them in the context of another. By looking at their difference answers, we can see where they diverge. And in the case of the hierarchy and flat organisational models, we can see they have diametrically opposing world views. Let's look at some software development specific organisational models next.

The Spotify Model

A popular model for structuring organisations is the Spotify Model.[6] Again, this model introduces elements not found in

[6]The company Spotify doesn't use the Spotify Model. We'll use Spotify's model and the Spotify Model interchangeably here.

previous models we examined. Roughly, the model consists of Squads (cross-functional autonomous teams that are responsible for a discrete part of the product, led by a triumvirate of a Product Owner, Tribe Lead, and Agile Coach), Tribes (for coordinating multiple Squads), a Tribe lead that coordinates with other Tribes, and Alliances (for collaboration across Tribes). Then you have Chapters (to deal with shared interests, such as standardisation and preferred practices in a field). Finally, a Guild is an ad hoc grouping of people interested in learning a specific topic.

Although the language is different from our previous models, there's still a clear sense of hierarchy in this model, but it's fairly shallow. This probably explains why it's a popular model: it appeals to managers who believe some form of hierarchy is critical to delivering anything, but it doesn't emphasize hierarchy at all.

The interesting thing about this model is that it introduces two styles of learning into organisational models: one for broad institutional learning (Chapters grow the skills of the organisation as a whole and capture that in standards and common practices), and another for individual upskilling (Guilds). Both are knowledge creation networks. All organisations have a need for learning, of course. But by explicitly naming Chapters and Guilds, the Spotify Model is saying that learning is important enough that you should deliberately structure your organisation to support it. It says that creating the knowledge network should be on people's radar as a distinct responsibility, not just an accidental by-product. Underlying this model is the belief that knowledge work requires structural support.

By adopting the Spotify Model, you're accepting that you need some hierarchy, but that it should be limited; and that besides value creation, individual and institutional learning are critical problems to organise for.

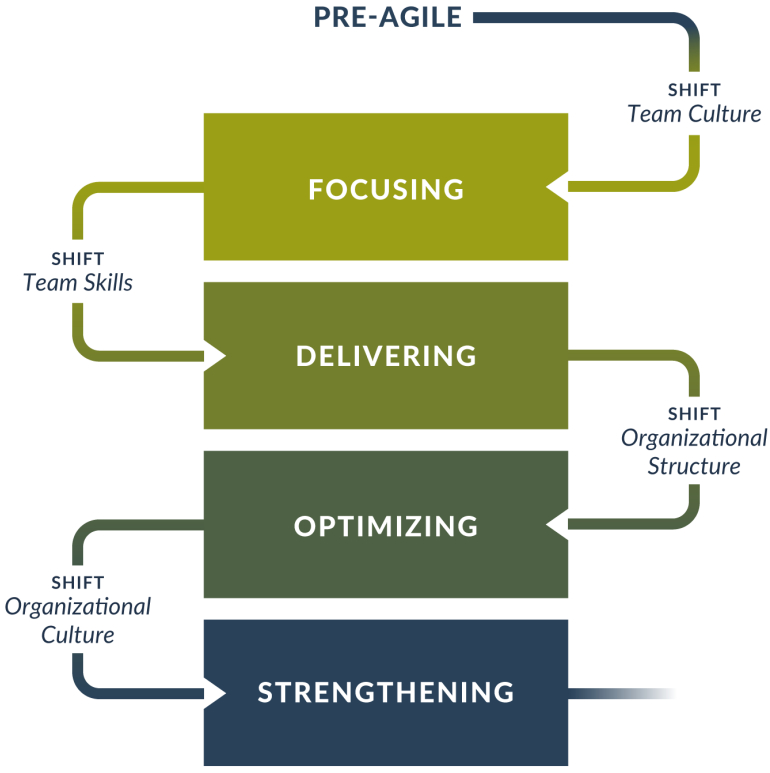
When you first learn about the Spotify Model, you accept implicitly that learning is an important concern. But if instead of merely

accepting this model, you actively compare it to other models, you can ask: "Why is it that value creation and knowledge are considered important? Which other aspects does this model exclude, and why?" Comparing models makes it easier to see what each model brings to the table.

How do other models solve for growing organisational knowledge? For example, some organisations introduce Centers of Excellence. These are standalone teams with dedicated skilled members. This is different from Chapters in the Spotify Model, where the members of a Chapter are also members of different Tribes. Because we are aware now that we can organise for knowledge, we can actively seek out other models, and compare how they address it.

Agile Fluency

Let's take a quick look at another model which is focused on learning. Agile Fluency is an organisational model that represents teams and organizations as being in different stages of their progress in becoming agile. These stages are defined by the activities they do to improve their capabilities and value to the business.



Agile Fluency

Zone

Benefit: Greater visibility into teams' work; ability to redirect.

Investment: Team development and work process design.

Learn From: Scrum, Kanban, non-technical XP

Time to Fluency: 2-6 months

.

Delivering

Benefit: Low defects and high productivity.

Investment: Lowered productivity during technical skill development.

Learn From: Extreme Programming, DevOps movement

Time to Fluency: +3-24 months

.

Optimizing

Benefit: Higher-value deliveries and better product decisions.

Investment: Social capital expended on moving business decisions and expertise into team.

Learn From: Lean Software Development, Lean Startup, Beyond Budgeting

Time to Fluency: +1-5 years

.

Strengthening

Benefit: Cross-team learning and better organizational decisions.

Investment: Time and risk in developing new approaches to managing the organization.

Learn From: Organization design and complexity theories

Time to Fluency: unknown

(Image and table: Diana Larsen and James Shore, agilefluency.org)

The picture alone doesn't do Agile Fluency justice of course. The underlying assumption of Agile Fluency is that when your organisation isn't producing enough value, you can solve this problem by first improving individual teams' capabilities and ways of working, and then restructuring the organisation itself.

Like the Spotify Model, this model addresses learning as an explicit organisational need (as opposed to something you tack on). Unlike the Spotify Model, however, Agile Fluency is not a model for structuring your organisation. It does recommend that you address

your organisational structure, but leaves that to other models. The scope of the Agile Fluency Model is about agile culture and skills.

What this model does introduce is the idea that teams, and therefore the organisation, progress over time. This notion of progression is not present (or in any case it is not very obvious) in the previous models we have discussed. It's not that those models force you into stasis; you can always change things around. But the previous models don't have the building blocks for representing progressive change. You're just supposed to change when needed, with no support from the model, and no language for explicitly reasoning about change.

In Agile Fluency, you can impact progress by choosing the appropriate interventions (focusing, delivering, optimizing, strengthening) at the right time, in the right order. The model's worldview is that doing these activities out of order is not going to yield the results you're looking for, because capabilities depend on previously acquired skills.

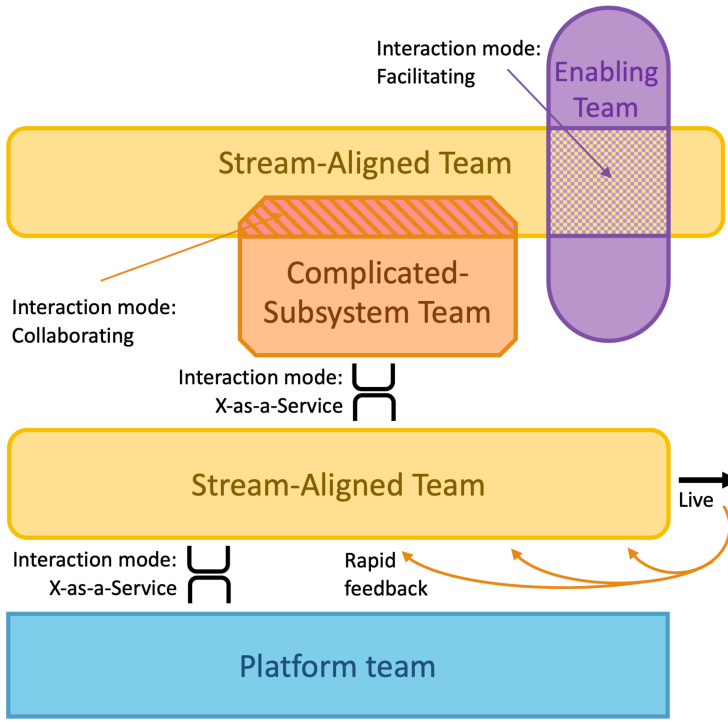
Now that we understand how the building blocks of Agile Fluency represent progress (instead of structure), we can use this view to compare it to other models.

The Spotify Model allows for evolution (you can split up teams and reorganize them), it allows for learning and encourages it by introducing structure for learning. But it stops short at explaining how to use that learning to progress. What should progress or evolution in the hierarchy model, or in the Spotify Model, or any other look like? The answer you come up with, leads to the next question: do I care about progress in my context? Maybe your situation doesn't call for progression. If it does, you've learned something about the gaps and assumptions of the model you use.

Heuristic
Compare different models to understand what you actually care about.

Heuristic
If a model doesn't explicitly address something you value highly, try fitting in something from another model and see if the model still hangs together.

Another interesting aspect of the Agile Fluency model is that it has an aspirational aspect, and admits this openly. True fluency playing a musical instrument is achieved over a very long time, and after a lifetime you can still learn more. Similarly, with the Agile Fluency model there is no achievable end state, but you should still strive to get in the strengthening zone. It is worth increasing your skills and capabilities, even if you never obtain all its benefits.



Team Topologies

(Image: Henny Portman)

Team Topologies is a software organisational model that focuses on fast flow and value creation. It advocates composing your engineering organisation from four types of teams and three types of team interactions. You should aligned teams on a common purpose, and reduce their cognitive load, so they can be efficient and focused. This model explicitly values reducing unnecessary cognitive load. The patterns (or topologies as they call them) are

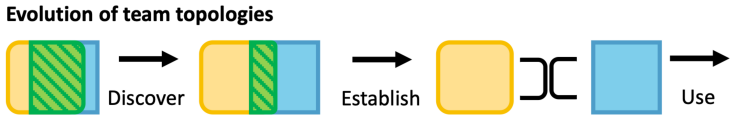
Stream-Aligned Teams, Enabling Teams, Complicated Subsystem Teams, and Platform Teams. These teams interact using three communication "modes": Collaboration, Facilitating, and X-as-a-Service.

When you have teams that don't map directly into these patterns, they "should either be dissolved, with the work going into stream-aligned teams or converted into another team type … [and] the team should adopt the corresponding team behaviours and interaction modes."[7] Of course there's more to Team Topologies than our summary, but team types and communication styles are the building blocks of the model.

With Team Topologies, there isn't a static structure with fixed hierarchies as in the Spotify Model. Instead, you combine the team types and interactions to suit your business goals in order to create value. Another difference is that the Spotify Model values team autonomy so much that it doesn't directly address cross-team collaboration. Team Topologies advocates autonomy as well, but includes collaboration building blocks. However, it advises to keep interactions between teams tightly constrained.

What's particularly interesting about Team Topologies in our opinion, is that it defines a set of composable patterns. As an implementer of Team Topologies, you are expected to combine these patterns in ways that optimise for flow in your context. Not only that, but you should expect this structure to evolve. When you sense that your context has evolved, you are expected reshape your organisation's team structure by reapplying these patterns. If this sounds like the design pattern catalogs that we have in software design, it's because it is very much inspired by that idea.

[7] "Team Topologies", Matthew Skelton & Manuel Pais, IT Revolution Press, 2019



Evolution of Team Topologies

(Image: Henny Portman)

Like software design patterns, Team Topologies even comes with some anti-patterns, such as team structures and communication styles to avoid. Unlike pattern catalogs, which typically are open-ended (or at least admit that they are incomplete), Team Topologies doesn't encourage us to extend the model with new team types and interaction modes.

In our context of evaluating organisational models, using patterns and a catalog of patterns that can be assembled to structure an organisation is a powerful idea worth exploring. Perhaps the other models would be better served with such a pattern language. Future models could take this approach even further and create pattern languages for other kinds of organisations that aren't doing software engineering.

Team Topologies is rooted in the DevOps-philosophy. Although it seems to claim to be suitable for all software organisations, it only addresses product-centric organisations. We don't feel it is suitable for systems like life-critical medical devices or railway infrastructure, because it doesn't have building blocks that address reliability or safety. It is biased towards this premise: create value and enable flow in product-oriented software organisations. It proposes a small, specific set of building blocks as the means to achieve that. If we wanted to, say, organise for human safety over fast flow, we'd need to come up with a different set of patterns for doing so. If we organised for software quality, correctness, worker wellbeing, learning, or domain understanding, our choices will be different again. Optimising for flow may ignore other important aspects of systems that you may need to consider.

Heuristic
All models are created within a specific context. Does the original context match yours? Can it be adapted?

You could fit a need for considering human safety into Team Topologies, by moving the responsibility for overall safety into a stream-aligned team, having an enabling team to support them, and a platform team to build the necessary infrastructure. But that is not the same as the model being explicitly designed to enable safety. None of Team Topologies' defined building blocks suggest that a specific team type or interaction mode serves to enable safety. Likewise, nothing in Team Topologies explicitly addresses learning: you can fit learning in there (the book suggests that some enabling teams are really like chapters and guilds from the Spotify Model), but nothing in the model explicitly supports learning. Team Topologies doesn't have a rich model and language for how teams learn, as exists in the Agile Fluency model.

These omissions are not a problem with Team Topologies per se, but rather a consequence of its focus. Team Topologies has value creation and flow as its scope. By presenting only building blocks for managing flow, it frames flow as the central concern worth organising for. Subsequently, it does not include explicit building blocks for learning or safety or programmer happiness. These omissions should be no surprise.

A problem arises, however, if we adopt the Team Topologies model, with its building blocks and patterns and language —thinking it will solve our organisational problem— without first asking whether improving flow is indeed our most pressing need, and whether we have other needs. If we ask such questions, we can avoid the Golden Hammer Syndrome. Furthermore, by comparing models we can figure out what we need from a model and what its benefits and limitations are.

Explicit Building Blocks

Building blocks are explicit, named concepts in a model. They're not the focus point of the model (like "control" in the hierarchy), but they're the elements the model uses to create that focus (like the "levels" in the hierarchy).

A building block is a lightning rod. It predictably focuses attention and energy. The things you find most important should be first class building blocks in the model you're introducing.

> **Heuristic**
> If you care about something, have a building block for it.

When you have a building block for something important, it draws people's attention. Simply by existing, that building block tells people that this is something that requires their consideration. If, on the other hand, there is no building block for it, people might still give it attention, but there is no home for it. Any attention given could be spotty, brittle, temporary or even deemed unwelcome.

Most of the time, when you introduce a new model, you've spent the time understand it and buy into all it has to offer. But others will not understand it as deeply as you do, nor will they apply it as rigourously as you do. You may understand where the model can be extended or adapted, but others will apply the model as is. So even if you can make the model work for your context by combining its building blocks with your adaptations, it doesn't mean others will. They are more likely to do it by the book (or by following your instructions). Adapting the models and their building blocks to your needs, gives you influence over people's attention, and therefore over the outcomes they produce. If you don't adapt these models, bringing focus to what you value, will be much harder.

If, say, you're reorganising and you want to improve how institutional learning happens in your organisation, you can either pick a model that explicitly addresses it with its building blocks (for example, Chapters and Guilds in the Spotify Model), or you can adapt another model to suit your needs. It's not that this learning won't happen in, say, a hierarchy or Team Topologies. But without the attention-grabbing power of a good, explicit, and well-explained set of building blocks, there's not much that tells others what to do and how to do it.

In all knowledge organisations, individual and institutional learning is important. If you're looking to apply an organisational model, you can ask how they deal with learning, and whether that's important to you. If learning is happening smoothly, you don't need to touch it. But if you feel it requires change, you need to have learning explicit in the model.

If instead of improving learning or flow, you want to improve the quality of delivery, or the safety of life-critical systems, your organisational building blocks would need to express that, whether they're blocks for types of teams, interactions, activities, meetings, or reports.

When the Team Gets Too Big: a Case Study

Models guide our choices for appropriate actions to take. As an example, let's ask what we should do when a team gets too big. We'll compare what different models suggest we do, to see what they teach us about their approach, and their underlying value systems.

Heuristic
When you're evaluating models, find questions that are relevant in your context, and use them to compare the models.

In a traditional hierarchy, when a branch has too many people, you add more subbranches and a corresponding layer of management to manage the people who report to them. At some point, when a hierarchy gets too deep, you make it wider: you split it apart by adding departments or business units with their own management, and allow these to each have their own hierarchies. The value system behind hierarchy is control. It assumes that to scale control, you need to delegate control and control the controllers; so the idea of growing deeper or wider is embedded in the model.

From Jason Yip[8], a Senior Agile Coach at Spotify, we get some clues on how Spotify deals with splitting squads or tribes.

"You wait for seams to appear: clunkiness in communication flow and interaction patterns. (…) Then you nudge things apart. Maybe a subgroup of people has a slightly different rhythm and events. Nothing formal, just nudging things apart. Over time you notice, hey, I'm not interacting with the other people in the team anyway. So you just formalise that [split]. If done correctly, this is mostly an acknowledgment and a non-event." Jason calls this "organic organisational design". In that model, structure should always follow strategy.

In Team Topologies, when the flow of a Stream-Aligned Team slows, or the team is experiencing increased cognitive load, then it's grown too big. To fix this you would look for multiple value streams hiding in there, and split the team according to those lines. You could also look for people working on a highly specialised aspect of the value stream, and move them to a separate Complicated Subsystem Team.

[8] "3 insights from 4 years at Spotify", Jason Yip, presentation at YOW! 2019. Slides and video.

When a Platform Team gets too big, Team Topologies suggests that the topologies are fractal. The Platform Team acquires internal topologies, consisting of internal Stream-Aligned Teams, Complicated Subsystem Teams, and Enabling Teams.

The Spotify Model makes no mention of Platform teams, but how would it solve the problem of a Platform Team getting too big? We think that you would create a Chapter that oversees and supports multiple Platform Teams, helping them with standardisation and tooling, but you wouldn't have them impose too much. This solution is different from a Team Topologies' Enabling Team, because the Chapter is composed of people from across the different Platform Teams.

Contrasting Spotify and Team Topologies

Interestingly, Team Topologies has no building blocks for any kind of group that spans people from different teams. Of course, a Team Topologies organisation could have such groups, but as a model it doesn't tell us to have them, doesn't tell us how to organise them, or even to recognize that such groups exist.

In the same sense, Tribes are another key difference between the model at Spotify and Team Topologies. Tribes are larger structures than squads. A Tribe manages strategy and goal-setting at a higher level. According to Jason Yip, Squads are less important to preserve than Tribes. Tribes can evolve fast and react organically to change. Team Topologies puts teams first, suggesting to make them long-living and seeing teams as the central unit of organisation. Teams are the only tool for looking at organisations, so there's no natural place for organising in terms of business strategy. You organise for flow.

Another fundamental difference between these two models is that

the Spotify Model doesn't mention cognitive load. Instead, it talks about clunkiness, friction, long meetings, … The term "cognitive load" doesn't show up in its vocabulary. By naming cognitive load, Team Topologies calls attention to it, telling us explicitly to look for and reduce it.

Team Topologies and the Spotify Model also perceive interaction between teams differently. The Spotify Model assumes interactions happen organically inside Squads, between Squads in Tribes, and cross-cutting though Chapters and Guilds.

Team Topologies sees too much interaction as an inhibitor to fast flow. Consequently, interactions need to be designed explicitly. Teams are supposed to be limited to as a few as possible interactions and interaction styles. Ideally, these interactions are expressed in a "team API": a set of well-defined places where communication happens, such as pull requests, issue trackers, or chat channels.

Finally, when an interaction doesn't work well in the Spotify Model, an Agile Coach will intervene. That Agile Coach is an explicit building block in the Spotify Model. In Team Topologies, there's no building blocks for coaches, managers, or other roles that can intervene. (Possibly, Team Topologies wants teams to self-organise, but there's also no mention of that option.) Again, lack of an explicit building block doesn't preclude you from having manager roles in Team Topologies, but you won't get any support for them from the model. Team Topologies doesn't tell you to focus on such a role or the value it might bring to the organisation.

Comparing Models Yourself

We encourage you to compare models to evaluate them. As we have shown, one way to compare models is to ask questions about how those different models might handle a particular problem you are interested in solving. To get started, pick situations (real or

imagined) relevant to your environment and test them against the models. Then, try to imagine how each model would handle them.

This is speculative: during this evaluation you're taking a theoretical view of how the models work and how they stack up against each other. That will never beat the insights you can get from practical experience applying the models. But it's much cheaper to test-drive a model through realistic scenarios, than it is to put it into practice only to uncover big surprises a year down the road. Besides the measurable costs, reorganisations lead to "reorg fatigue". Likewise, models that address aspects other than organisations might have varying costs. Running thought experiments with scenarios this way teaches you something about the value system of each model early on. It also exposes whether the model cares about the same things you find important.

As you run various scenarios, you'll notice that:

- some models may give very detailed advice (eg "split a team when it exceeds 10 members");
- others offer only vague guidance (eg. "teams shouldn't be too big", without quantifying what is too big or what should be done);
- some models have nothing to say about that specific situation.

None of these are bad things. All models include and omit different things, and give more practical or more broad advice. But when applying model you need to understand who you're getting married to.

Now that you have some ideas on how to compare models, we'd like to give you some more clues. First, we'll discuss how some models (or their authors) are authoritative. They provide one way, and aren't open to being extended. Then, we'll discuss how our own cognitive biases can influence our perception of models. Finally, we'll look into how models are intended to be used: as observational

tools, or as prescriptions of how to act, or even as aspirational guidelines for a future state.

Bossy Models

You might be familiar with SOLID. It's a set of five object-oriented software design principles. The first letter of each principle forms the word SOLID. For example the Single Responsibility Principle is the "S" in SOLID. This model gives us a framing right off the bat:

- There are only five design principles worth caring about.
- It's these five that matter, and not any others.
- They apply to all software design, independent of context.
- You're not just supposed to apply these principles in an ad hoc fashion, but you're supposed to apply them at all times.
- The name implies that by applying them, your design will in fact be solid, which is a good thing.

As before, we want to break out of the framing of this model, so that we can understand it better. Here are some questions we ask: Why five? Why these five and not others? What if a sixth principle is discovered? In what context were these principles formulated? Do they apply to the design of all kinds of software in all contexts? Are they as true now as they were 25 years ago? Has software design changed since then?

As evidenced by their framing, it's clear that the authors intended for this model to be authoritative. As an adopter of this model, you're not supposed to extend it, or challenge it. (And many of its adopters seem to buy into this framing.)

In reality, there are dozens (if not hundreds) of design principles that have been discovered, long before and long after SOLID. And yet, the SOLID model didn't adapt. Even if you wanted to evolve it, SOLID doesn't offer any hooks for extensibility.

The Hierarchical organisation model is another example of a definitive model. The building blocks are the hierarchy, where decisions made at the correct level are pushed down, and the reporting goes up the hierarchy chain. There's no room in the scope of the model to extend it. Extend the hierarchy differently, altering decision-making, or changing the reporting structure, doesn't fit in the belief system that underlies this model.

Some models are authoritative and definitive, while others are open to being extended. But models rarely make this distinction explicit. They don't tell you whether to extend them and in what ways. It's hidden in the language they use, the names and the structures they propose, and in the communications of the authors about the model.

Often, all you need is a straightforward model to tell you how to do something. In this case, the model's simple building blocks are an *enabling constraint*: they help you to see something clearly, and make progress faster. In other cases, an authoritative model that seems suitable at first sight, can become a *limiting constraint* over time. In this case, on top of dealing with your original problem, you're now also faced with problems caused by the limitations imposed by the model. You start force-fitting things into the building blocks, instead of using the building blocks to achieve your goals. Force-fitting causes you to lose essential components of your context because they don't fit the model. Inversely, it also causes you to overemphasize unimportant components of your context because the model gives them focus. The metaphors that the model uses, can distort your perception of your context.

Finding alternative models, again, can help us out of this problem. In the case of the Hierarchical model, it's the Social Network and the Value Creation Network models that break us out of constraints. They tell us to stop accepting the framing of the Hierarchy.

> **Heuristic**
> Are you fitting the model to your context, or are you force-fitting your context to the model?

Getting Too Cosy with the Model

When we are faced with something new, like a model, we are initially inclined to question it. But we want to believe. We like certainty. So as soon as we're comfortable with the model, we stop questioning. Being in a state of cognitive ease makes us more accepting of new information or ideas.

Successful models tend to have some characteristics that contribute to our cognitive ease. They have a small number of building blocks clearly presented (perhaps even simplistically so), usually 7 or less. There's symmetry, balance, and the structures are often repeating. Strangely, these characteristics make us believe the model is more sound, more rigorous, better thought out. It may well be, however, that the model's structure has been simplified to be symmetrical, balanced, repetitive, and slim. We associate feeling at ease with goodness. Visualisations that fit our sense of "good" structure also help sell a model: quadrants, pyramids, funnels, radars, Venn diagrams with three neatly intersecting sets.[9]

Anecdotes are another tool for putting us at ease. If something is explained to us through stories, we're more likely to accept it. We trust the story that a familiar face tells us more than when someone gives us critical facts. When a model comes to us with anecdotes of how it contributed to a successful outcome, we don't question the story.

When we're at cognitive ease, we trust our intuitions. We narrowly frame our problems to fit the models we're comfortable with. We become overconfident.

The remedy to cognitive ease is a healthy dose of skepticism, aimed not just at new models, but at our existing models as well. Skeptics examine things, even if this makes them uncomfortable. They let go of certainty in order to see more of what the problem really is and

[9]Outside of mathematics, when is the last time you saw three sets in a Venn diagram intersecting, but one of the intersections is empty?

what the model has to offer. We use "getting out of your comfort zone" for social situations or challenging tasks, but it applies to our inner world as well.

> **Heuristic**
> Be more skeptical of nice models, be more open minded to jarring models.

Be on the lookout for things that don't fit, that are slightly awkward, that break the nice structure of the model you're using. Allow yourself to entertain multiple competing models at the same time, and to context-switch between them liberally. This gives you a certain freedom: no longer boxed in by a single model, you can come to see the models for what they are: potentially useful worldviews, not "truths".

> **Heuristic**
> Models are potentially useful worldviews, not truths.

Prescriptive, Descriptive, Aspirational

In the 16th century, the word network referred to wires or threads arranged in a fishing net like construct. In the 18th century, the term network was adopted to first represent railroads and canals, by the 1940's it meant radio broadcasting, then linked computers in the 1970's, and finally, in the 1980's, it acquired the meaning of building human relationships. Our modern use of the word is a metaphor, with extra steps.

The model that introduced the social network inside organisations is descriptive. Someone observed their environment, and noticed a

pattern of informal communications. They then mapped this pattern to the metaphor of the network. Using the network metaphor, they found a shortcut to describe these real-world patterns. Perhaps you have seen these informal communications as well, but you had never truly noticed them. But then, when someone points out to you that it is a social network, a lightbulb goes on. You now have a language for it.

A descriptive model doesn't have an intention, it only means to clarify, and help you see something in a new light. This change of perspective does have consequences: you might change your behaviour, for example, by creating opportunities for people in your organisation to mingle socially.

A prescriptive model, on the other hand, deliberately aims to change your behaviour. The Spotify Model clearly has this intention. It's telling us that our situation will improve if we restructure our organisation according to the prescribed building blocks. The Spotify Model originated as a descriptive model of how people organised, before it was turned into a prescriptive model for organisations by others later.[10]

> **Heuristic**
> Does this model help me see something clearly, or does it intend to create a change in behaviour?

The distinction between descriptive and prescriptive is not as a clear cut as we might like. Only in science do we find purely descriptive models. While the social network model intends to be descriptive, by introducing the term into your organisation, you make a value choice. You're saying that organisations should care about the social network. Usually this is followed by a prescription of how to do that. You may want to enable this network, leave it to evolve organically, or prohibit it. When you adopt a model

[10]"Why Do Hotel Bathrooms Lack Toothpaste? Prescriptivists vs Descriptivists and You", Romeu Moura, presentation at Domain-Driven Design Europe 2019.

that you believe to be descriptive, but that actually comes with an underlying value system that you unconsciously adopt, then the model controls you. You are inadvertently allowing it to affect your behaviour. By critically evaluating whether a model is mostly descriptive or prescriptive, you can choose to adopt its values and use its prescriptions more intentionally.

> **Heuristic**
> Find the prescriptions hidden in innocent looking models, and use them to your advantage.

Some models go even further: they are aspirational. They describe an ideal situation that has not been observed before. The author has created a model of how they believe the world could work, but has not actually applied it, or only partially. Aspirational models serve as a call to action. They speculate that, if we were to adopt all its prescriptions, we could get close to that ideal state, but perhaps never reach it. If we understand that a model is aspirational, then our goal is not to apply it precisely to the letter, but to get as close to it as is useful and practical. Interestingly, some models start as aspirational, but as people start getting better at explaining and implementing them, they observe that the model does indeed give them its stated benefits. If used wisely, aspirational models can drive positive change by getting people to see a new possibility.

> **Heuristic**
> Consider that some models may not actually have been tried by the authors.

Putting the Model to Work

With many of the models we acquire throughout our lives, we only engage superficially, and that's fine. We absorb and assimilate a lot

of things without ever deeply engaging with them. This happens when, for example, you're being onboarded at a new employer, and you simply learn their operating models for doing things in certain ways. The same goes when you're given some design specifications and implement them as stated. And as a student, you're usually expected to learn and adopt the models you are taught.

But when we're trying to achieve something important, we need to take a more critical view. After learning of a new model, we need to decide whether we want to put it to work to help solve our problem. We need to engage more deeply with that model before we decide to commit to it. What if we could make this engagement an intentional process?

What we've been trying here, is to identify competing models and to compare them to understand their benefits and limitations, their framing, their values, and the contexts in which they apply. This comparison breaks us out of thinking there is only one option to solve our problem.

We're making an upfront analysis of a model, before we invest in the model too heavily. This comes with risks: Because we haven't lived with this model yet, we might be overanalysing things. We may discount some of benefits because they only surface after spending some effort implementing the model. Still, if we do this analysis upfront, it saves us effort that we would otherwise waste on implementing an unsuitable model. As we're not invested deeply yet, it's much cheaper to identify faults in a model and how address them. To avoid the risk of "analysis paralysis", we remain conscious that no model is perfect, but may still be useful.

> **Heuristic**
> There's always another possible model.

Then, when we find a new model that has a reasonable representation of the problem we're trying to solve, we can adopt it. We apply

its building blocks, and execute the activities it prescribes. As we do, we need to be observant: there is a thin line between learning to adopt a model, and overfitting a model to our situation. This friction is valuable data. Maybe the model has problems, lacks detail, has too much detail, focuses on the wrong aspects, has unintended consequences, or doesn't directly tackle the problem we're actually trying to solve. Maybe the model doesn't fit that well. Or perhaps we don't want to use the model in the way it's prescribed. Perhaps we have different values than those supported by the model. If we collect this feedback early, we can use it to reexamine the model.

> **Heuristic**
> Following a model as prescribed is not the same as achieving success.

In other words, no matter how beautiful your diagram of hierarchies, Squads and Tribes, or Stream-Aligned and Enabling Teams is, the proof is in the pudding.

As we gain more experience with a model, and learn from feedback, we'll feel more comfortable reshaping the model, bringing in elements from other models we previously dismissed, adjusting it to our context, and making it truly our own.

In summary, these are the five steps for engaging with a model:

- Intentionally study models
- Analyse and compare them critically
- Adopt a model in your context
- Gather feedback about the impact
- Reshape it to your needs

Did we just present to you a 5-step model for evaluating models? Are there other models for evaluating models you can compare this one to? Should you adopt our model or create your own? Does our model put you in a certain framing that isn't relevant to your context? We leave all these question as an exercise for the reader.

Conclusions

Our worldviews and behaviours are mostly the result of other people's models that we've assimilated, along with their biases. Models give us a framing, they highlight some aspects of the problem and obscure others. This is useful, assuming they focus on aspects that are important to us.

When we introduce an unsuitable model into our environment, it distracts us. It makes us ask questions in the framing of the model, and we don't see that we should be asking different questions. It does this through its building blocks. These building blocks act like lightning rods. They attract the attention and energy of the people who are engaging with them.

Understanding a model's framing is hard, precisely because we've been put into that framing. In that sense, we're like the fish who can't understand the world outside of the fishbowl, because nothing in their environment offers any building blocks to explain and structure it.

Acknowledging that you have limited agency is liberating. Comparing a model to other models gives us an edge: we can see more clearly which aspects we could highlight or obscure. By comparing models, we can break out of their constraints, and engage critically with the models.

You can choose which models to let in.

Splitting a Domain Across Multiple Bounded Contexts

Imagine a wholesaler of parts for agricultural machines. They've built a B2B webshop that resellers and machine servicing companies use to order. In their Ubiquitous Language, an Order represents this automated process. It enables customers to pick products, apply the right discounts, and push it to Shipment.

Our wholesaler merges with a competitor: They're an older player with a solid customer base and a huge catalog. They also have an ordering system, but it's much more traditional: customers call, and an account manager enters the order, applies an arbitrary discount, and pushes it to Shipment.

The merged company still has a single Sales Subdomain, but it now has two Sales Bounded Contexts. They both have concepts like Order and Discount in their models, and these concepts have fundamentally the same meaning. The employees from both wholesalers agree on what an order or a discount is. But they have different processes for using them, they enter different information in the forms, and there are different business rules.

In the technical sense that difference is expressed in the object design, the methods, the workflows, the logic for applying discounts, the database representation, and some of the language. It runs deeper though: for a software designer to be productive in either Bounded Context, they'd have to understand the many distinctions between the two models. Bounded Contexts represent Understandability Boundaries.

In a perfectly designed system, our ideal Bounded Contexts would usually align nicely with the boundaries of the subdomains. In reality though, Bounded Contexts follow the contours of the evolution of the system. Systems evolve along with the needs and opportunities of organisations. And unfortunately, needs and opportunities don't often arise in ways that match our design sensibilities. We're uncomfortable with code that could be made more consistent, if only we had the time to do so. We want to unify concepts and craft clean abstractions, because we think that is what we should do to create a well-designed system. But that might not be the better option.

Deliberate Design Choices

The example above trivially shows that a single subdomain may be represented by multiple Bounded Contexts.

Bounded Contexts may or may not align with app or service boundaries. Similarly, they may or may not align with domain boundaries. Domains live in the problem space. They are how an organisation perceives its areas of activity and expertise. Bounded Contexts are part of the solution space; they are deliberate design choices. As a systems designer, you choose these boundaries to manage the understandability of the system, by using different models to solve different aspects of the domain.

You might argue that in the wholesaler merger, the designers didn't have a choice. It's true that the engineers didn't choose to merge the companies. And there will always be external triggers that put constraints on our designs. At this point however, the systems designers can make a case for:

- merging the two Sales Contexts,
- migrating one to the other,
- building a new Sales Context to replace both,

- postponing this effort,
- or not doing anything and keeping the existing two Contexts.

These are design choices, even if ultimately the CEO picks from those options (because of the expected ROI for example). Perhaps, after considering the trade-offs, keeping the two Sales Contexts side by side is the best strategic design choice for now, as it allows the merged company to focus on new opportunities. After all, the existing systems do serve their purpose. The takeaway here is that having two Bounded Contexts for a single Subdomain can be a perfectly valid choice.

Twenty Commodities Traders

When there is no external trigger (as in the wholesaler merger), would you ever choose to split a single domain over multiple Contexts, deliberately?

One of us was brought in to consult for a small commodities trader. They were focused on a few specialty commodities, and consisted of 20 traders, some operational support roles, and around 10 developers. The lead engineer gave a tour of the system, which consisted of 20 Bounded Contexts for the single Trading Domain. There was one Context for each trader.

This seemed odd, and our instinct was to identify similarities, abstract them, and make them reusable. The developers were doing none of that. At best they were copy-pasting chunks of each other's code. The lead engineer's main concern was "Are we doing the optimal design for our situation?" They were worried they were in a big mess.

Were they?

Every trader had their own representation of a trade. There were

even multiple representations of money[11] throughout the company. The traders' algorithms were different, although many were doing somewhat similar things. Each trader had a different dashboard. The developers used the same third party libraries, but when they shared their own code between each other, they made no attempt at unifying it. Instead, they copied code, and modified it over time as they saw fit. A lot of the work involved mathematical algorithms, more than typical business oriented IT.

It turned out that every trader had unique needs. They needed to move fast: they experimented with different algorithms, projections, and ways of looking at the market. The developers were serving the traders, and worked in close collaboration with them, constantly turning their ideas into new code. The traders were highly valued, they were the prima donnas in a high stress, highly competitive environment. You couldn't be a slow coder or a math slacker if you wanted to be part of this. There were no (Jira) tickets, no feature backlogs. It was the ultimate fast feedback loop between domain experts and programmers.

Things were changing very rapidly, every day. Finding the right abstractions would have taken a lot of coordination, it would slow development down drastically. An attempt at unifying the code would have destroyed the company.

This design was not full of technical debt. It also wasn't a legacy problem, where the design had accidentally grown this way over the years. The code was working. This lack of unifying abstractions was a deliberate design choice, fit for purpose, even if it seems like a radical choice at first. And all the developers and traders were happy with it.

These weren't merely separate programs with a lot of repeated code either. This was a single domain, split over 20 Bounded Contexts, each with their own domain model, their own Ubiquitous Language, and their own rate of change. Coordinating the language

[11] https://verraes.net/2019/06/emergent-contexts-through-refinement/

and concepts in the models, would have increased the friction in this high speed environment. By deliberately choosing to design individual Contexts, they eliminated that friction.

Trade-offs

There are consequences of this design choice: When a developer wanted help with a problem, they had to bring that other developer up to speed. Each developer, when working in another bounded context, expected that they'd have to make a context switch. After all, their terms and concepts were different from each other, even though they shared similar terminology. Context-switching has a cost, which you've probably experienced if you've work on different projects throughout a day. But here, because the Contexts were clearly well-bounded, this didn't cause many problems. And sometimes, by explaining a problem to another developer with a similar background (but a different Bounded Context), solutions became obvious.

Multiple Bounded Contexts in Ordinary IT Systems

The trading system is an extreme example, and you won't come across many environments where a single Subdomain with 20 Bounded Contexts would make sense. But there are many common situations where you should consider splitting a domain. If in your company, the rules about pricing for individual and corporate customers are different, perhaps efforts to unify these rules in a single domain model will cost more than it is worth. Or in a payroll system, where the rules and processes for salaried and hourly employees are different, you might be better off splitting this domain.

Conclusion

The question is not: Can I unify this? Of course you can. But should you? Perhaps not. The right Context boundaries follow the contours of the business. Different areas change at different times and at different speeds. And over time what appears similar may diverge in surprising and unexpectedly productive ways (if given the opportunity). Squeezing concepts into a single model is constraining. You're complicating your model by making it serve two distinct business needs, and taking a continued coordination cost. It's a hidden dependency debt.

There's two heuristics we can derive here:

1. Bounded Contexts shouldn't serve the designer's sensibilities and need for perfection, but enable business opportunities.
2. The Rate of Change Heuristic: Consider organising Bounded Contexts so they manage related concepts that change at the same pace.