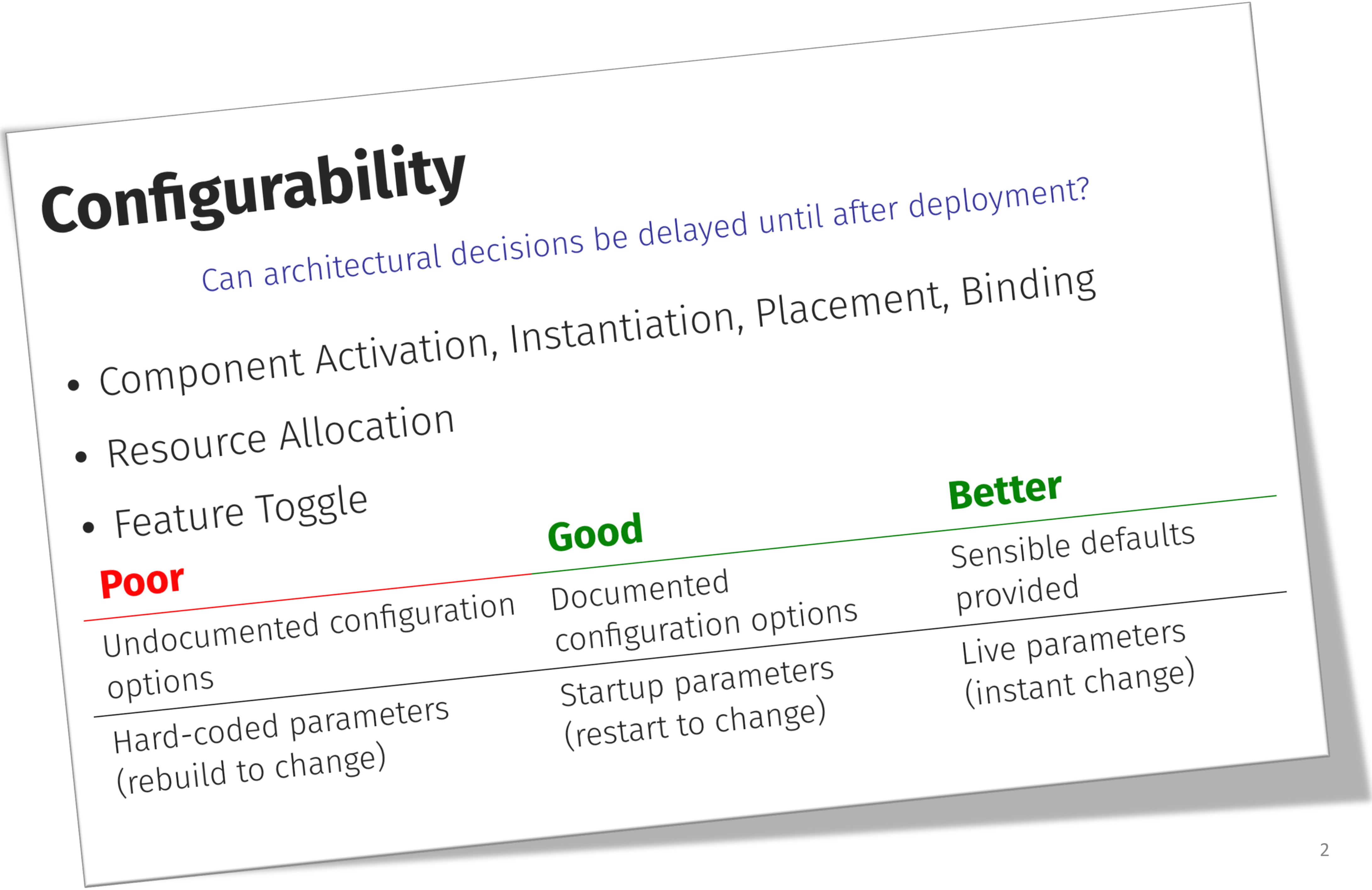
IN4315, Software Architecture Lecture 8

- Configurability
- Managing change
- Technical Debt
- Better Writing
- Wrap Up



Configurability

Can architectural decisions be delayed until after deployment?

- Component Activation, Instantiation, Placement, Binding
- Resource Allocation
- Feature Toggle

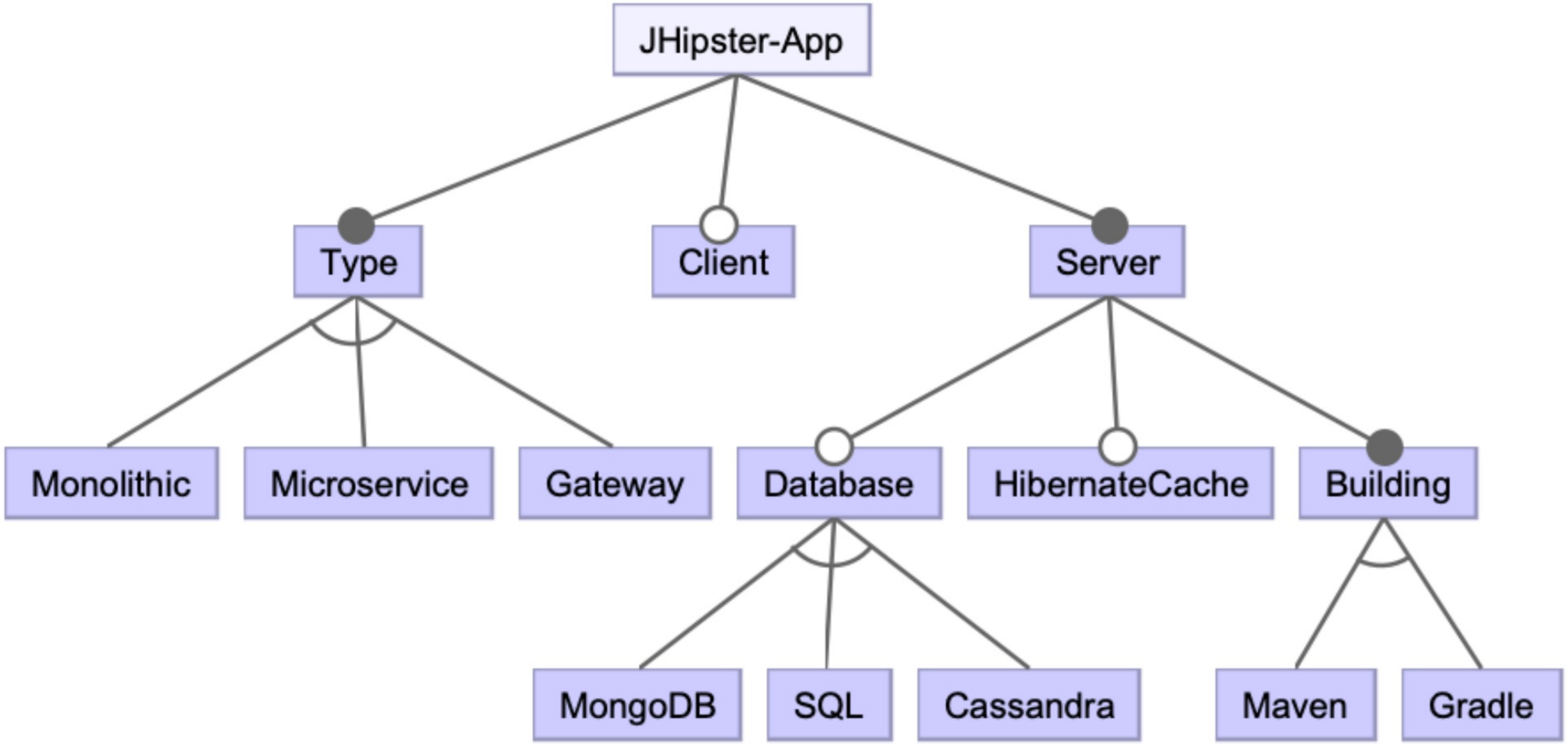
Poor	Good	Better
Undocumented configuration options	Documented configuration options	Sensible defaults provided
Hard-coded parameters (rebuild to change)	Startup parameters (restart to change)	Live parameters (instant change)

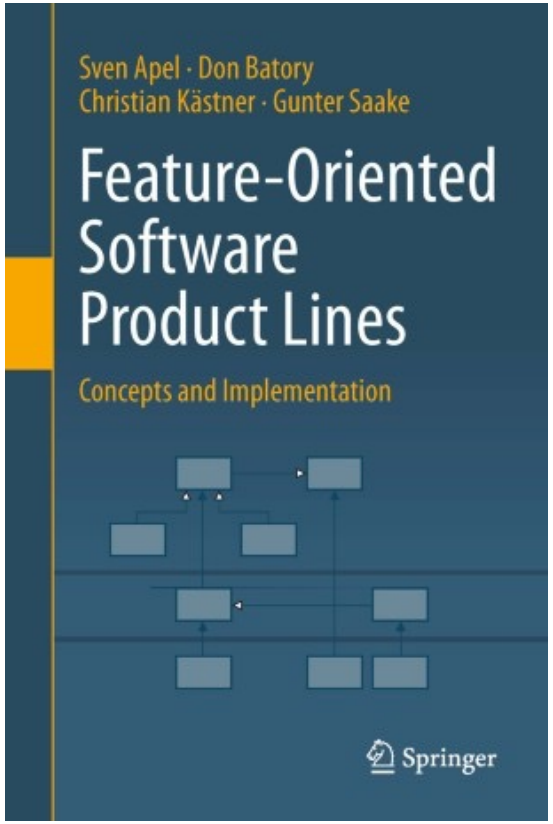
The Theory of *Software Product Lines*

Linux: 12,000 options
Thus: 2^12,000
different products

- Reusable software infrastructure
- With “variation points” that can be configured
- To derive many different software products
- Variation can be “bound” at compile time, start-up time, run time, ...
- Variation points: from Boolean flags to plugins loaded at run time
- Single variation point can crosscut many infrastructure “assets”
- Variability can be *modeled* using “feature diagrams”
- Widely studied research topic in design, implementation, testing, ...

Feature (= Option) Modeling



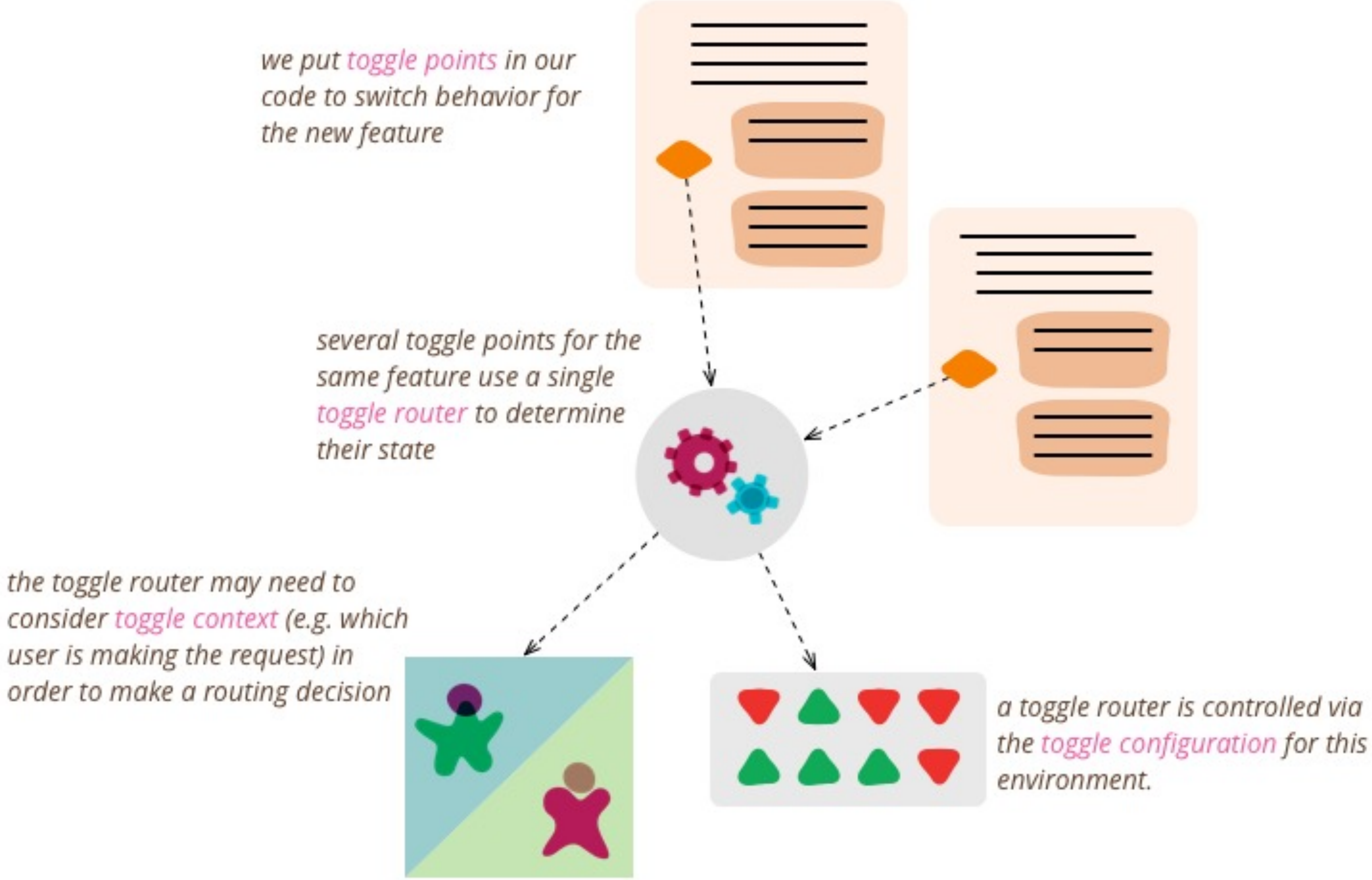

Sven Apel · Don Batory
Christian Kästner · Gunter Saake

Feature-Oriented Software Product Lines

Concepts and Implementation

Springer

https://martinfowler.com/articles/feature-toggles.html

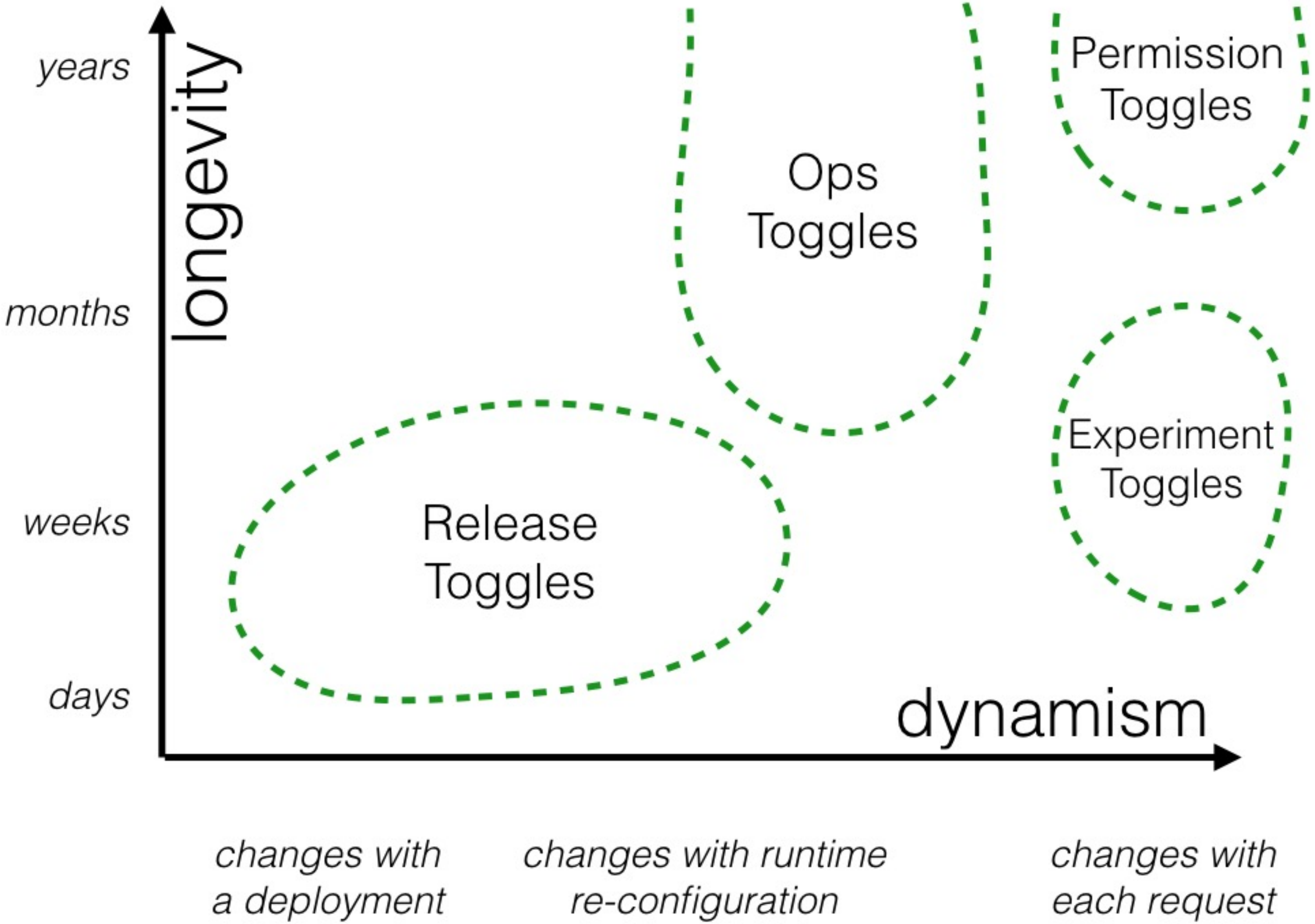


we put toggle points in our code to switch behavior for the new feature

several toggle points for the same feature use a single toggle router to determine their state

the toggle router may need to consider toggle context (e.g. which user is making the request) in order to make a routing decision

a toggle router is controlled via the toggle configuration for this environment.



https://martinfowler.com/articles/feature-toggles.html

The Architecture of a System (IEEE):

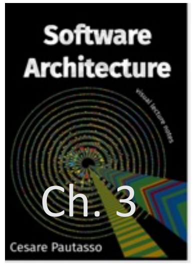
- The set of fundamental concepts or properties
- of the system in its environment,
- embodied in its elements and relationships,
- and the principles of its design and evolution.

Software Evolution

- Evolution is what makes software valuable

- Software success generates ideas for new system usage
  - Business opportunities
  - Integration with other systems
  - Legal constraints

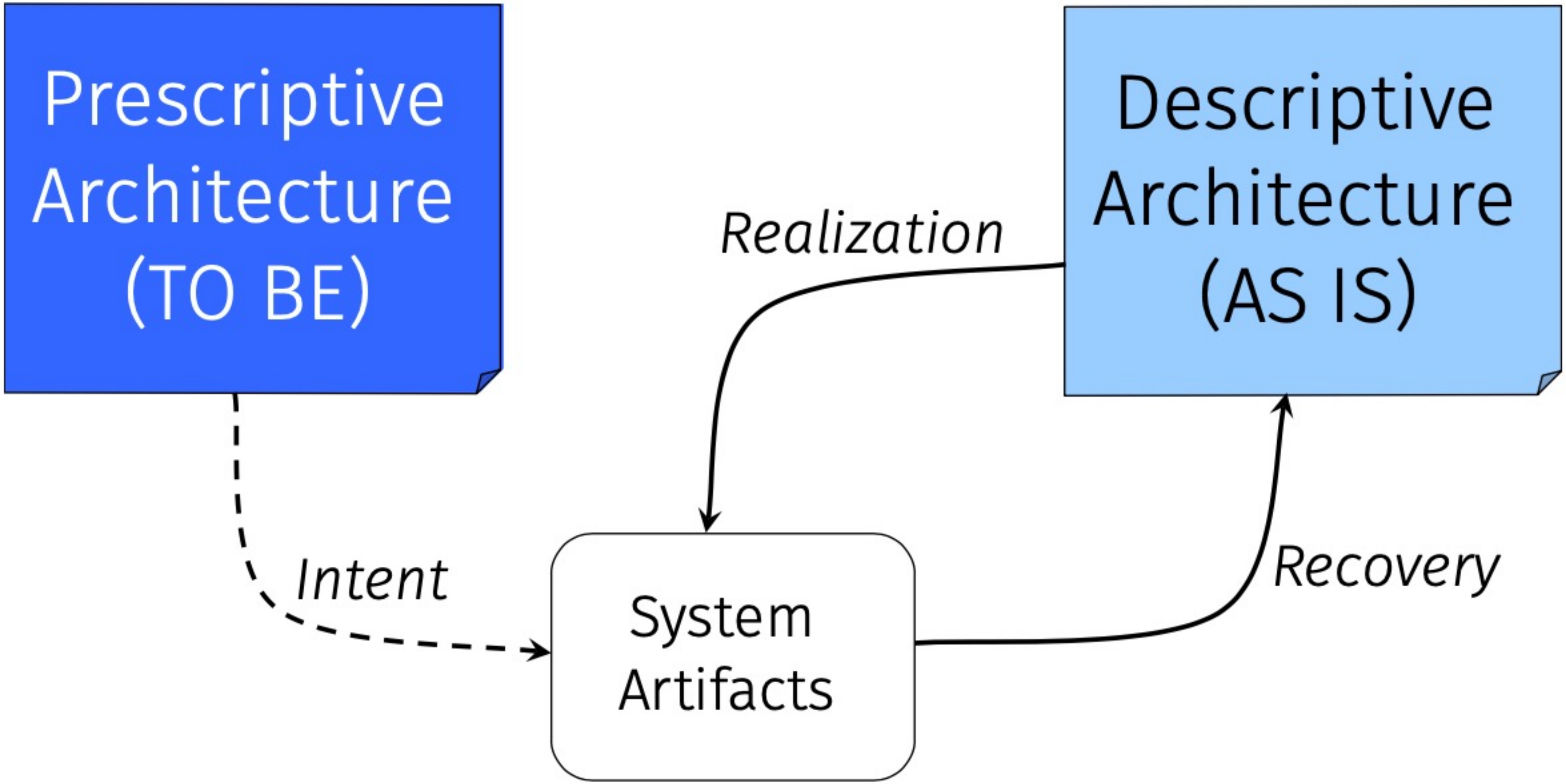
- But ... lots of evolution may erode the system
  - The “software evolution paradox”



Software Architecture

Ch. 3

Cesare Pautasso



Prescriptive Architecture (TO BE)

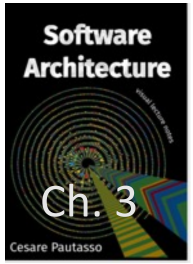
Descriptive Architecture (AS IS)

Realization

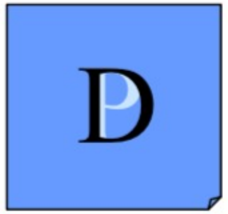
Intent

System Artifacts

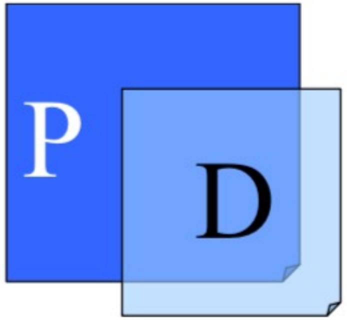
Recovery



Architectural Degradation



Ideal Case



Realistic Case

- Ideal Case (P = D)
- D always a perfect realization of P

- Not all P decisions can be implemented
- Over time, P and D change independently and drift apart

Keeping Software *Soft*

- Ensuring software can be easily changed
- Ensuring changes don’t deteriorate future evolvability

- Relevant to virtually any software system that is actually used:
    - Especially the active open source systems as studied in this course

- How can you architect for “high rates of change”?

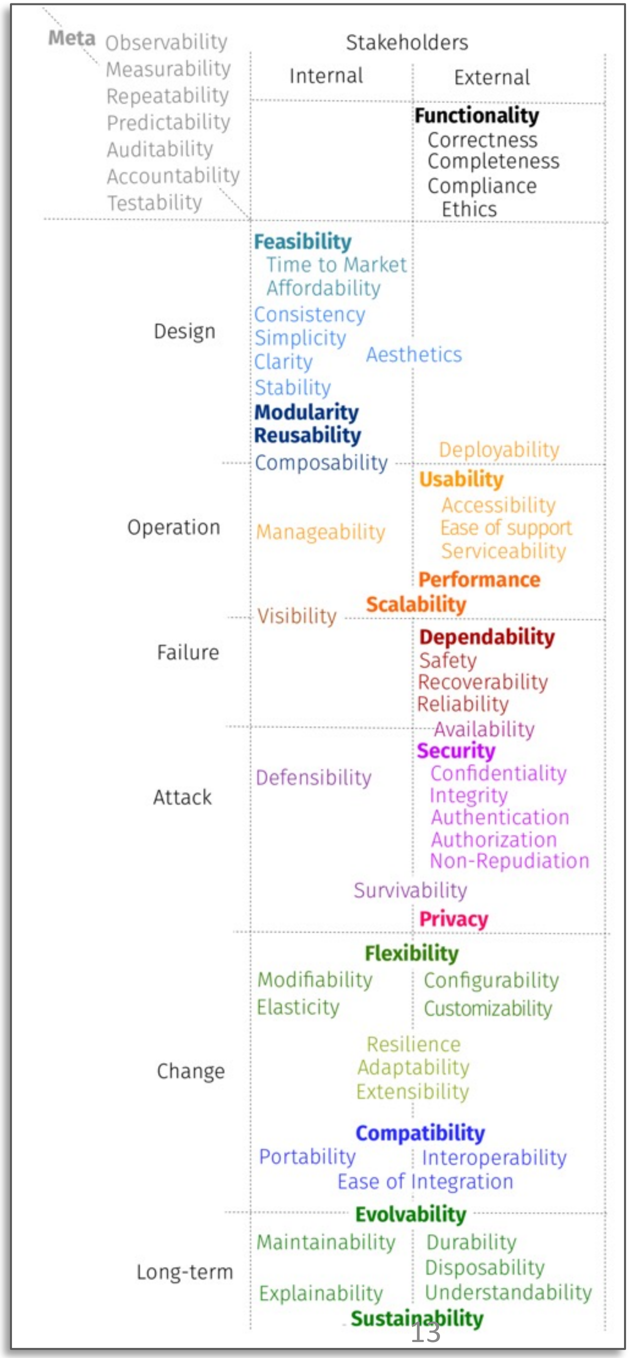
Architecting for High Rates of Change?

- Clear separation of concerns
- Clear rules for architectural integrity
  - Patterns, constraints, ...
- “Automate Everything”
  - Infrastructure as code, test execution, linters, ...
- Carefully designed test infrastructure
  - Making it easy to test new features
- Well-defined code reviewing process
- Continuous integration / delivery / deployment

(Automating) Quality Assurance

- Collection of procedures put in place to ensure that system continues to meet (pre-set) quality objectives
- Procedures will differ per quality attribute
- High levels of automation facilitate continuous evolution

- [Can be **highly regulated** for safety-critical domains:
  - Health, (self-)driving cars, aviation, finance, ...]



Meta Observability Measurability Repeatability Predictability Auditability Accountability Testability

Stakeholders: Internal | External

Functionality: Correctness, Completeness, Compliance, Ethics

Design: **Feasibility** (Time to Market, Affordability), Consistency, Simplicity, Clarity, Stability, **Modularity**, **Reusability**, Composability; Aesthetics; Deployability

Operation: Manageability; **Usability** (Accessibility, Ease of support, Serviceability); **Performance**; **Scalability**; Visibility

Failure: **Dependability** (Safety, Recoverability, Reliability, Availability)

Attack: Defensibility; **Security** (Confidentiality, Integrity, Authentication, Authorization, Non-Repudiation); Survivability; **Privacy**

Change: **Flexibility** (Modifiability, Elasticity, Configurability, Customizability); Resilience, Adaptability, Extensibility; **Compatibility** (Portability, Interoperability, Ease of Integration)

Long-term: **Evolvability** (Maintainability, Explainability, Durability, Disposability, Understandability); **Sustainability**

Testing and Architecture

- Testability as explicit quality attribute
- Testing as the *guide* that helps to reach a good design
- The test harness:
  - Common (mock, stub) objects that ease testing
  - Example test cases that can be easily adjusted
  - Reuse of test code among test cases
- Test coverage:
  - What types of coverage are monitored?
  - What is the coverage of key components?
  - How is test coverage information *used*?

Testing and Architecture (cont.)

- Test execution:
  - Test cases run on every commit (duration?)
  - Long-running / expensive test cases for performance, portability?
- Coding standards / style of the test code
- Test suite modularization
- Test culture (what people actually do):
  - Are tests part of the discussion in pull requests?
  - Do pull requests typically come with test cases?
  - What is the test code / production code ratio?

Assess for your system!

The Quality *Culture*

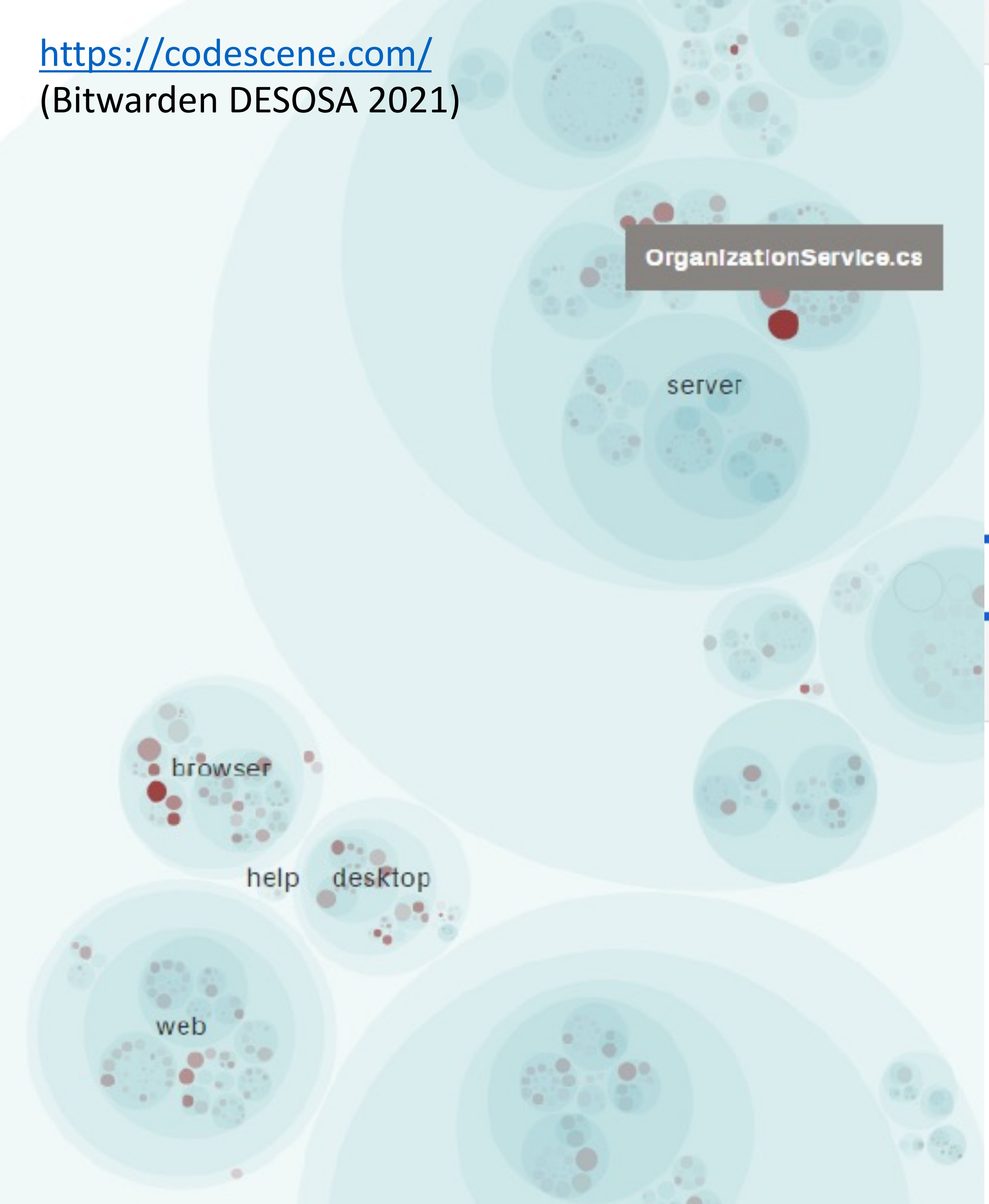
- Prescriptive: Quality practices as they *should* take place
- Descriptive: Quality practices as actually adopted.

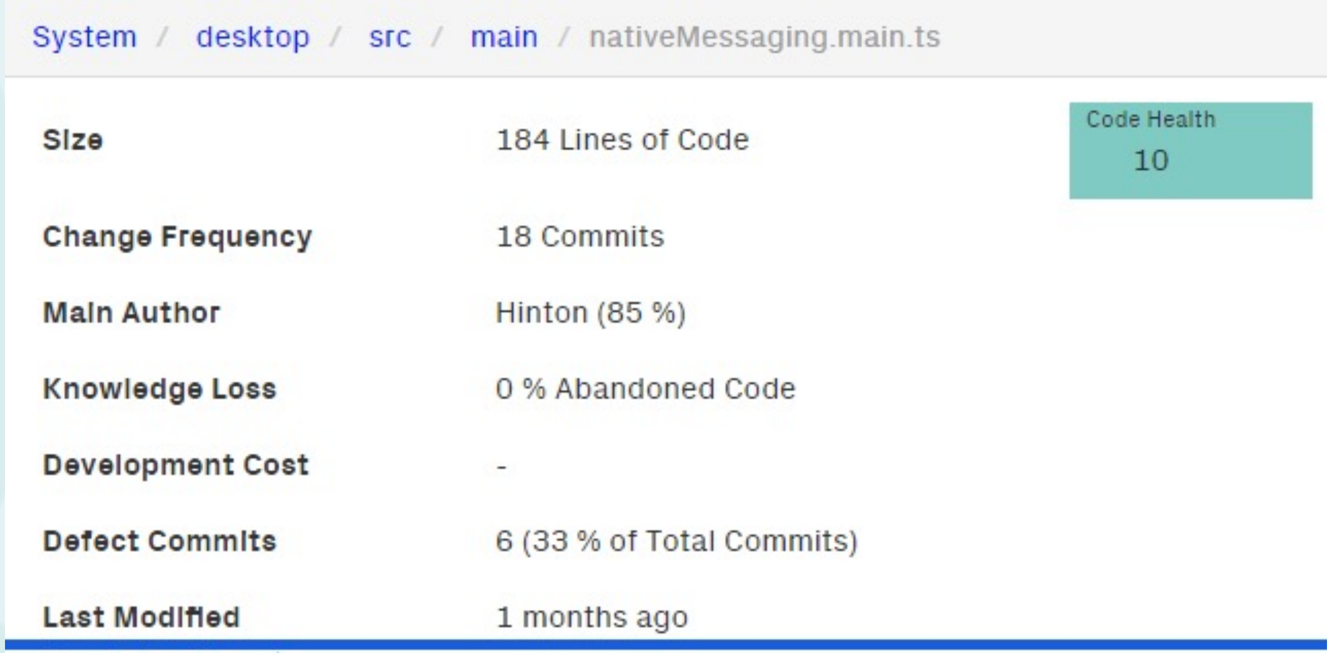
- “Ethnography” of open source activities can reveal actual practices
- Discussions in issues and pull requests:
  - Will help reveal properties that really matter in the system
  - And how those properties are safeguarded

Differences in Rates of Change: Hotspots

- Not all components may change at equal rates
- *Hotspot* components change faster than others
- Hotspot may be consequence of poor design
  - Monolithic class that no one dares to break up – need to refactor?
- Hotspots of the past:
  - Likely to have suffered from too many changes – should it be cleaned up?
- Hotspots of the future:
  - Map future use cases from roadmap onto component view
  - Consider refactoring now to isolate change

https://codescene.com/
(Bitwarden DESOSA 2021)



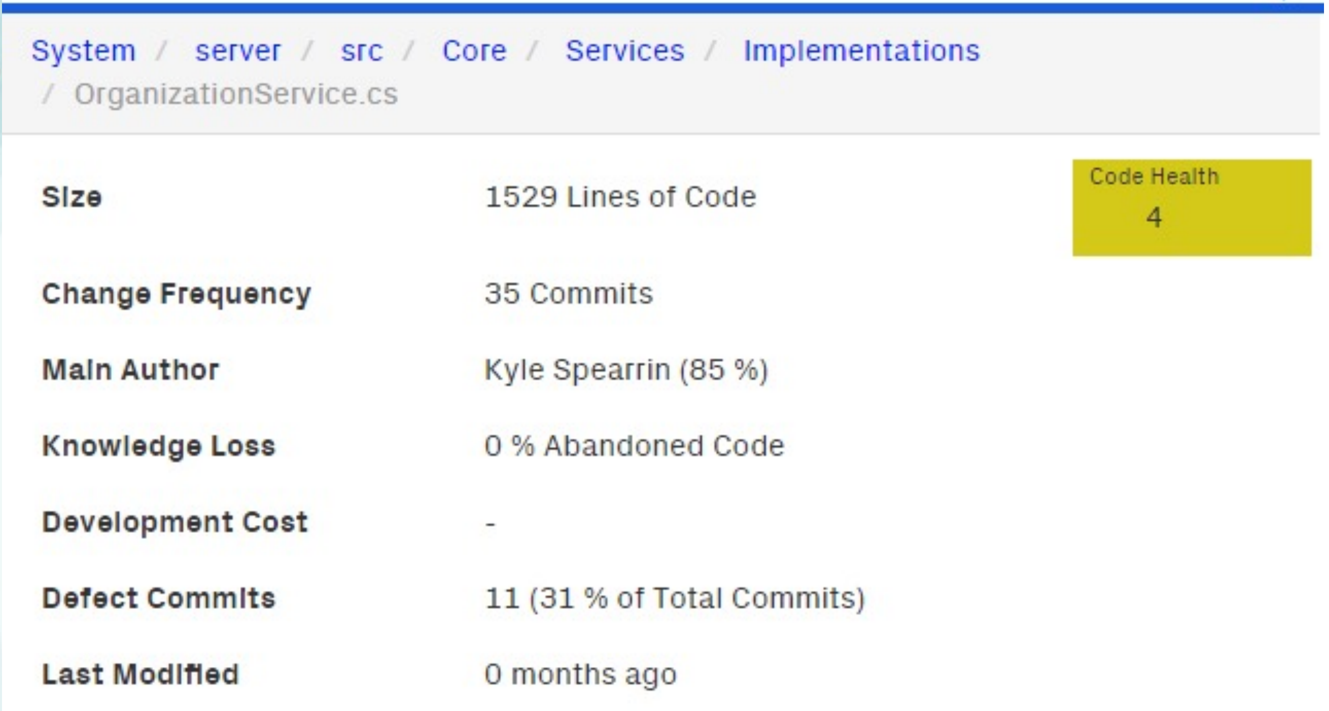

System / desktop / src / main / nativeMessaging.main.ts

Size	184 Lines of Code
Change Frequency	18 Commits
Main Author	Hinton (85 %)
Knowledge Loss	0 % Abandoned Code
Development Cost	-
Defect Commits	6 (33 % of Total Commits)
Last Modified	1 months ago

Code Health 10

Healthy file ↑

Unhealthy file ↓


System / server / src / Core / Services / Implementations / OrganizationService.cs

Size	1529 Lines of Code
Change Frequency	35 Commits
Main Author	Kyle Spearrin (85 %)
Knowledge Loss	0 % Abandoned Code
Development Cost	-
Defect Commits	11 (31 % of Total Commits)
Last Modified	0 months ago

Code Health 4

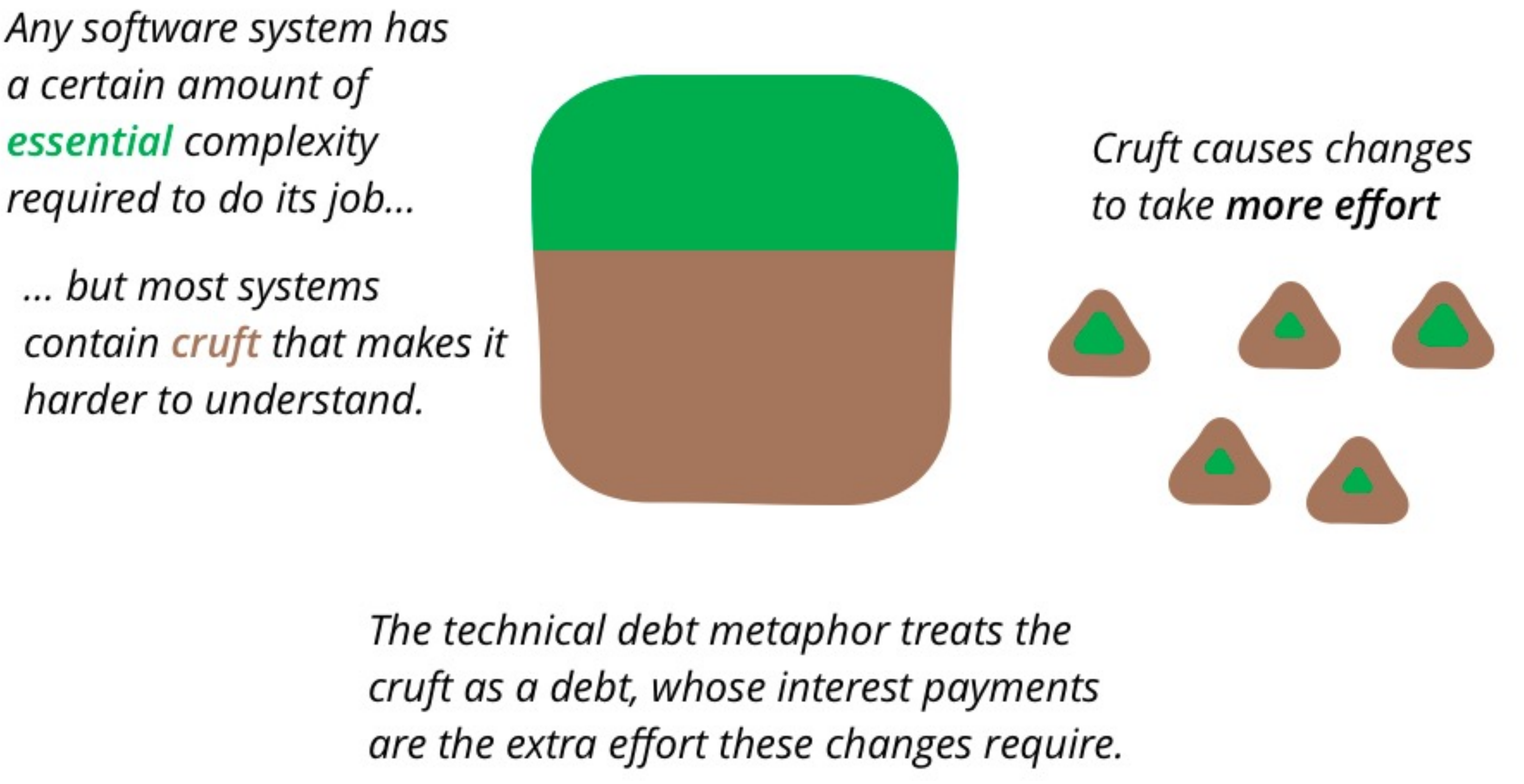
What Is Technical Debt?

- Ward Cunningham:
  - “I coined the debt metaphor to explain the refactoring that we were doing.”

http://c2.com/cgi/wiki?WardExplainsDebtMetaphor

- Michael Feathers:
  - “The refactoring effort needed to add a feature non invasively”

https://www.youtube.com/watch?v=7hL6g1aTGvo



Any software system has a certain amount of essential complexity required to do its job...

... but most systems contain cruft that makes it harder to understand.

Cruft causes changes to take ***more effort***

The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

https://martinfowler.com/bliki/TechnicalDebt.html

Technical Debt vs Code Quality

sonarqube

JArchitect

http://www.sqale.org/

Software
Improvement
Group

- Several static analysis tools to detect
  - Code smells
  - Maintainability issues
  - Vulnerabilities
- Very useful code analysis tools
- Substantial insight in code (structure) / quality
- Somewhat narrow interpretation of technical debt. E.g. SonarQube:

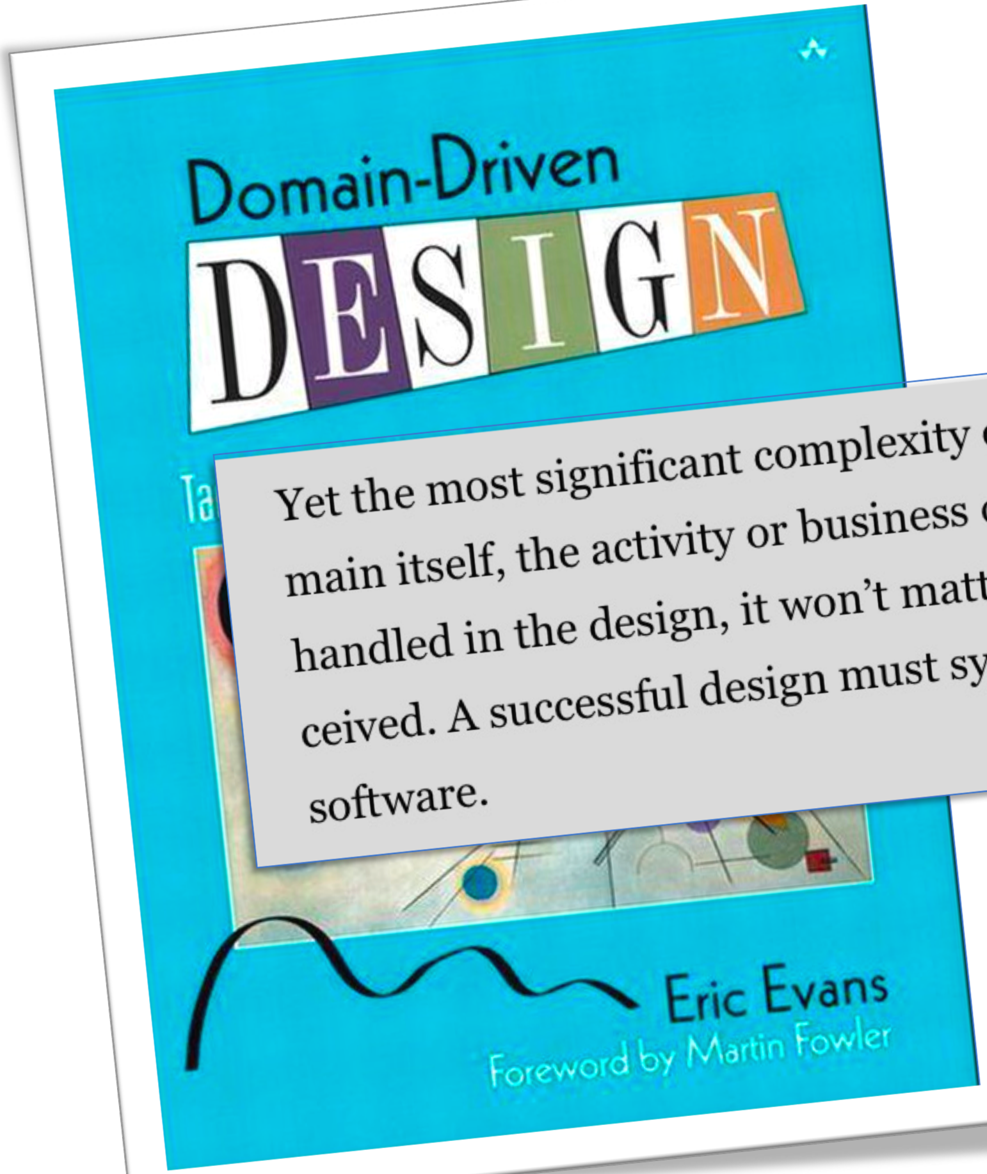
  “Estimated time required to fix all maintainability issues / code smells”

A lot of bloggers at least
have explained the debt metaphor and confused it, I think,
with the idea that you could write code poorly
with the intention of doing a good job later
and thinking that that was the primary source of debt.

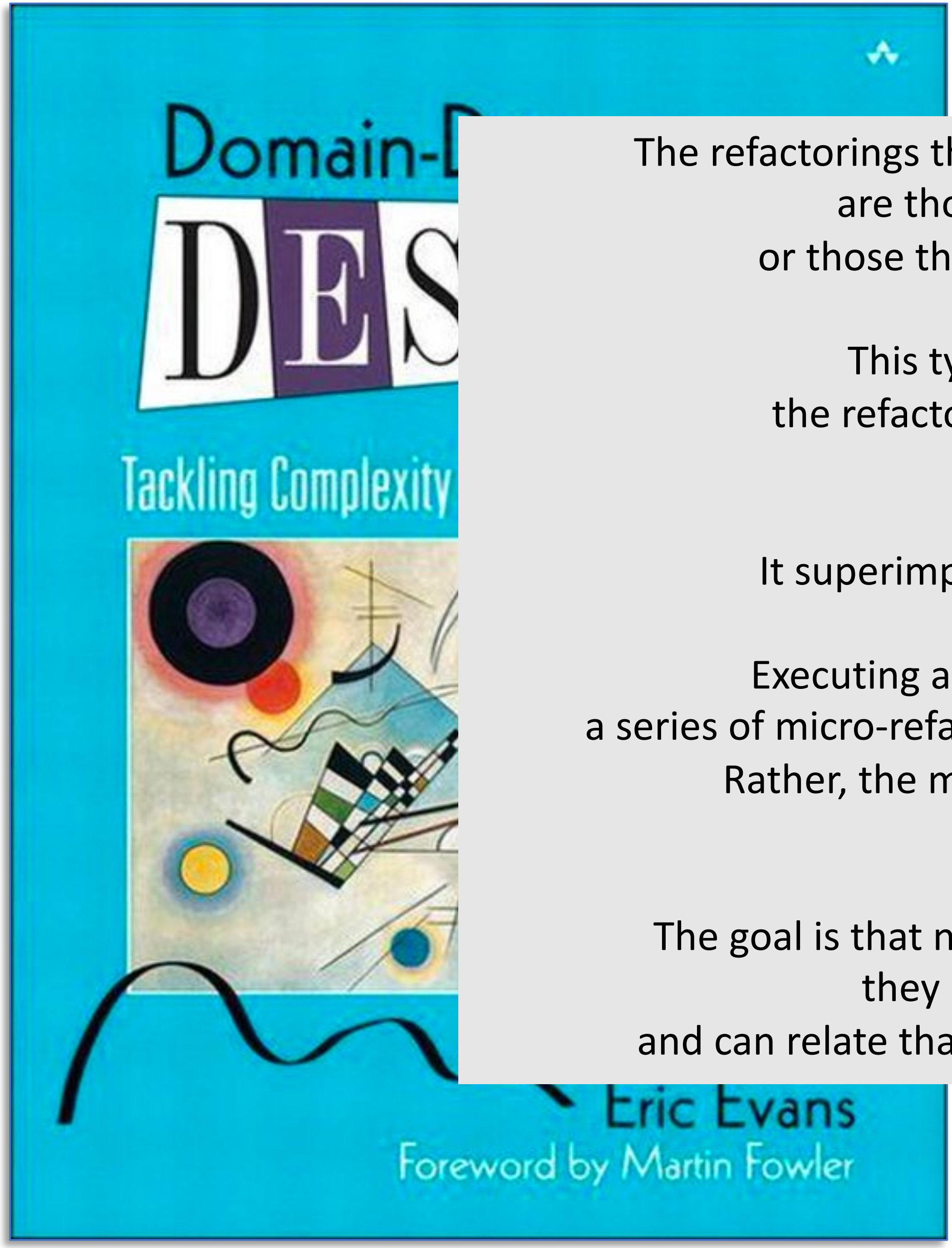
I'm never in favor of writing code poorly.

*I am in favor of writing code to reflect
your current understanding of a problem
even if that understanding is partial*

http://c2.com/cgi/wiki?WardExplainsDebtMetaphor



> Yet the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not handled in the design, it won't matter that the infrastructural technology is well conceived. A successful design must systematically deal with this central aspect of the software.



The refactorings that have the greatest impact on the viability of the system
are those motivated by new insights into the domain
or those that clarify the model’s expression through the code.

This type of refactoring does not in any way replace
the refactorings to design patterns or the micro-refactorings,
which should proceed continuously.

It superimposes another level: refactoring to a deeper model.

Executing a refactoring based on domain insight often involves
a series of micro-refactorings, but the motivation is not just the state of the code.
Rather, the micro-refactorings provide convenient units of change
toward a more insightful model.

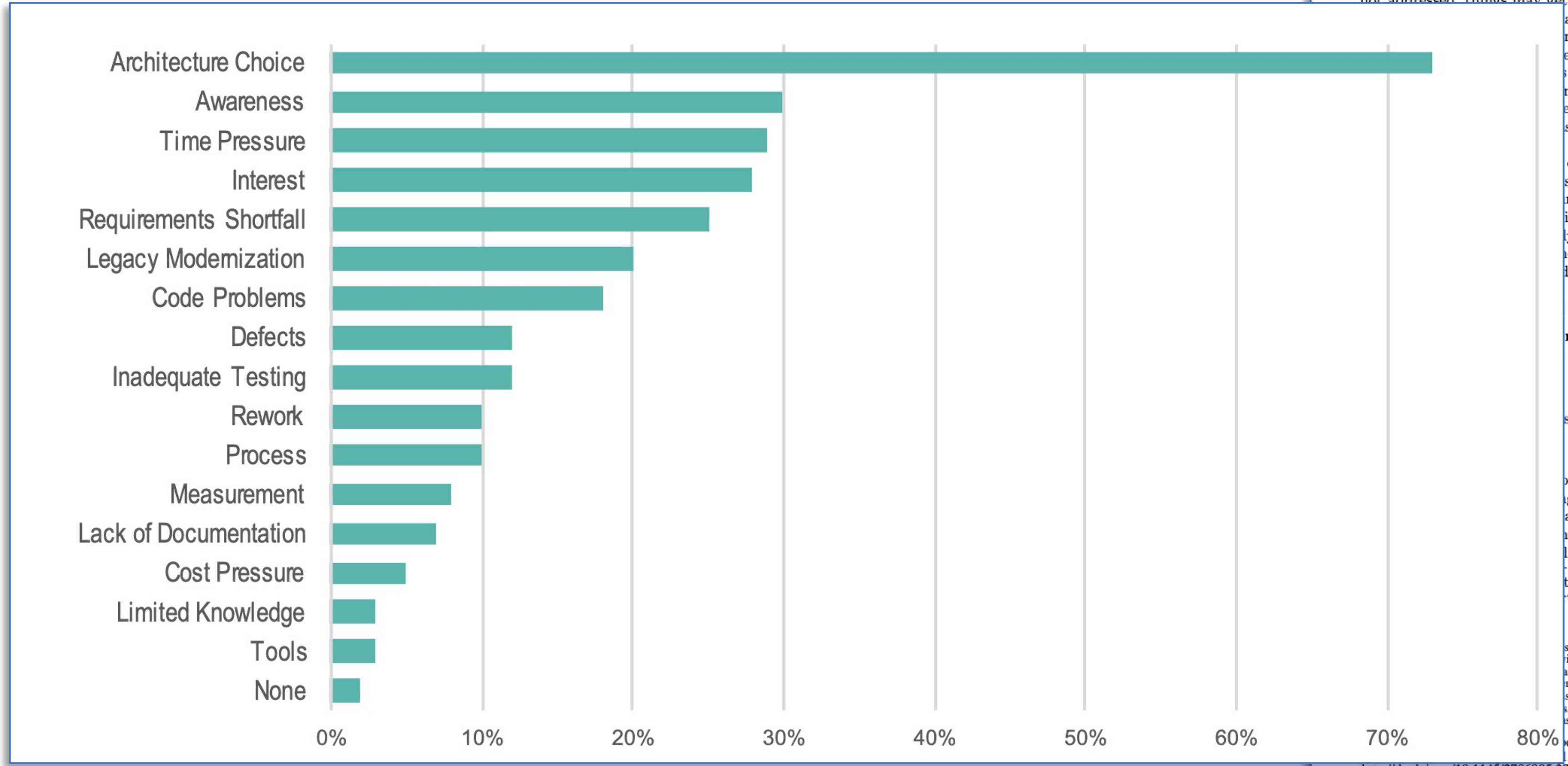
The goal is that not only can a developer understand what the code does;
they can also understand *why* it does what it does
and can relate that to the ongoing communication with the domain experts

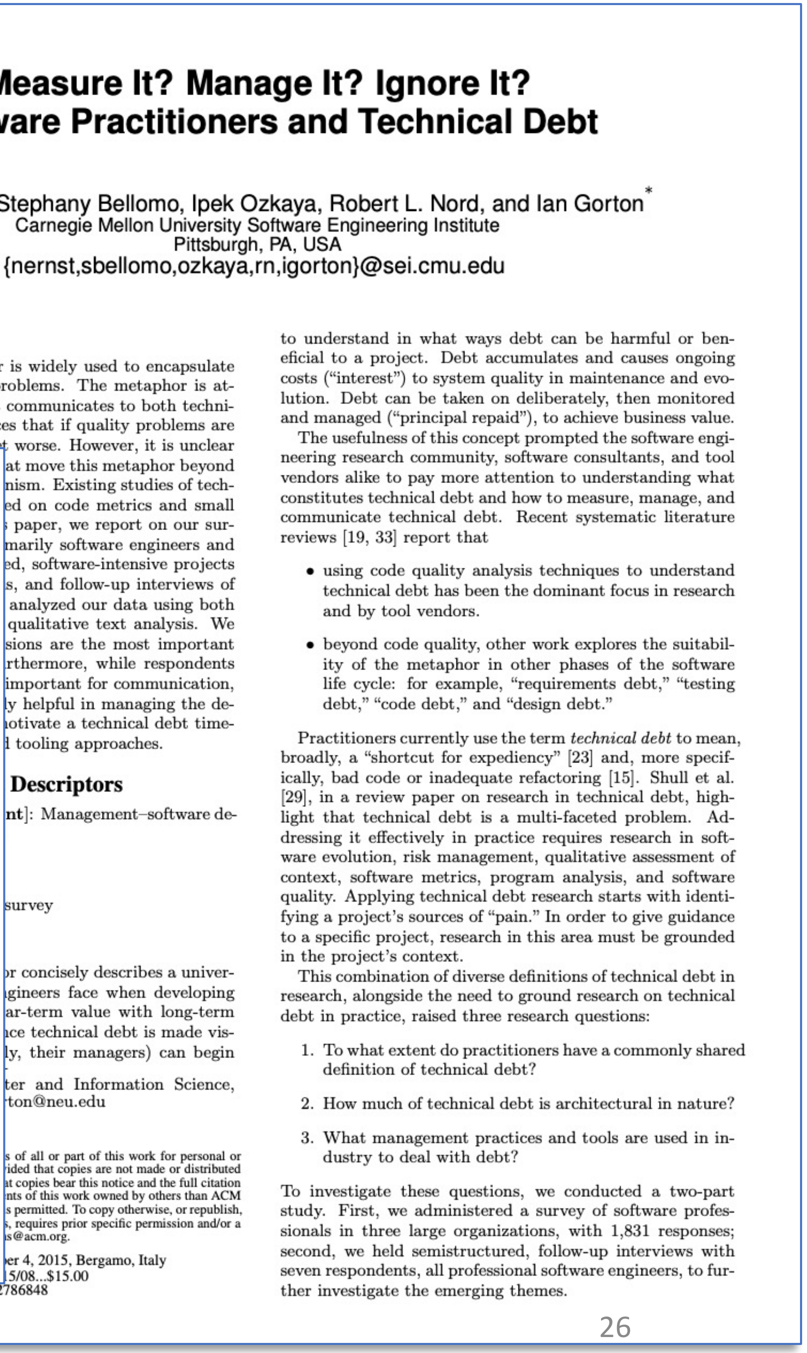
Debt Versus Value

	Visible	**Invisible**
Positive value	New functionality	Architecture Structure facilitating change
Negative value	Bugs	Technical debt

Kruchten, 2013:
The (missing) value of software architecture

Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt





Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt

Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton*
Carnegie Mellon University Software Engineering Institute
Pittsburgh, PA, USA
{nernst,sbellomo,ozkaya,rn,igorton}@sei.cmu.edu

ABSTRACT

The technical debt metaphor is widely used to encapsulate numerous software quality problems. The metaphor is attractive to practitioners as it communicates to both technical and nontechnical audiences that if quality problems are not addressed, things may get worse. However, it is unclear
at move this metaphor beyond
nism. Existing studies of tech-
ed on code metrics and small
paper, we report on our sur-
marily software engineers and
ed, software-intensive projects
s, and follow-up interviews of
analyzed our data using both
qualitative text analysis. We
sions are the most important
rthermore, while respondents
important for communication,
ly helpful in managing the de-
otivate a technical debt time-
l tooling approaches.

Descriptors

nt]: Management–software de-

survey

or concisely describes a univer-
gineers face when developing
ar-term value with long-term
ce technical debt is made vis-
ly, their managers) can begin

ter and Information Science,
ton@neu.edu

s of all or part of this work for personal or
ided that copies are not made or distributed
t copies bear this notice and the full citation
nts of this work owned by others than ACM
s permitted. To copy otherwise, or republish,
, requires prior specific permission and/or a
s@acm.org.
er 4, 2015, Bergamo, Italy
5/08...$15.00
http://dx.doi.org/10.1145/2786805.2786848

to understand in what ways debt can be harmful or beneficial to a project. Debt accumulates and causes ongoing costs ("interest") to system quality in maintenance and evolution. Debt can be taken on deliberately, then monitored and managed ("principal repaid"), to achieve business value.

The usefulness of this concept prompted the software engineering research community, software consultants, and tool vendors alike to pay more attention to understanding what constitutes technical debt and how to measure, manage, and communicate technical debt. Recent systematic literature reviews [19, 33] report that

- using code quality analysis techniques to understand technical debt has been the dominant focus in research and by tool vendors.
- beyond code quality, other work explores the suitability of the metaphor in other phases of the software life cycle: for example, "requirements debt," "testing debt," "code debt," and "design debt."

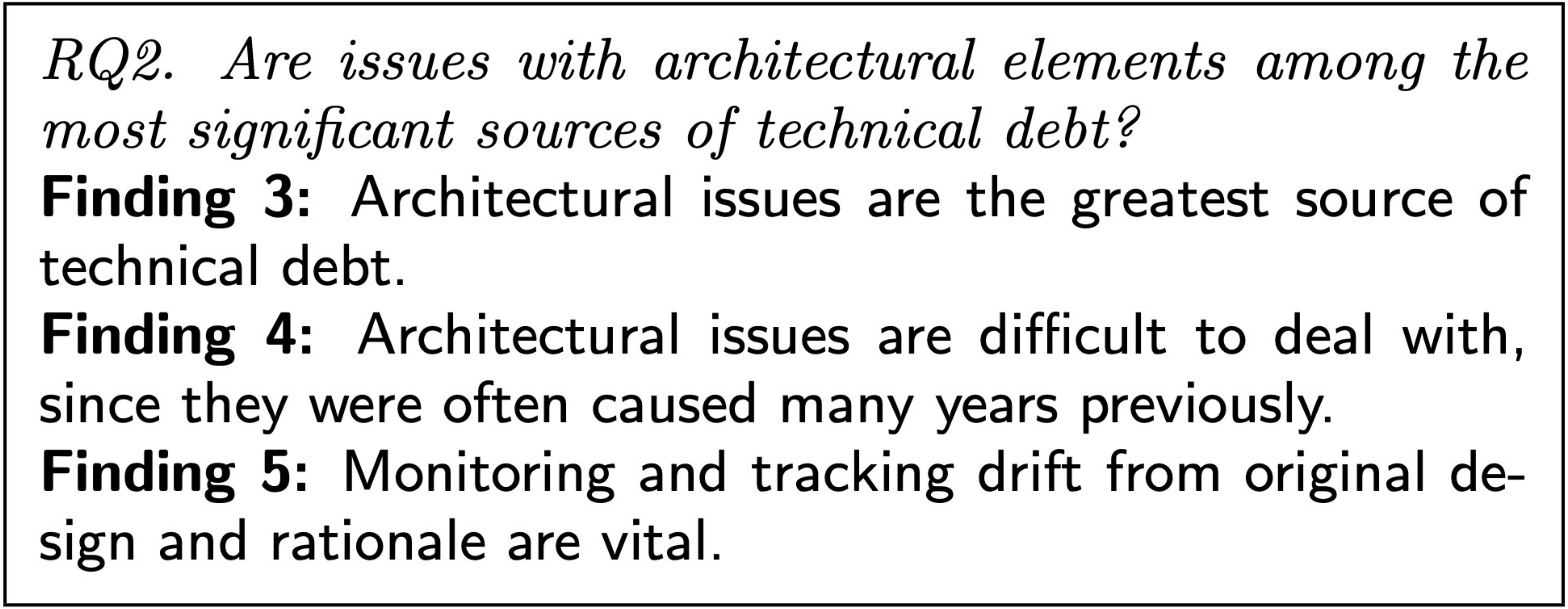
Practitioners currently use the term *technical debt* to mean, broadly, a "shortcut for expediency" [23] and, more specifically, bad code or inadequate refactoring [15]. Shull et al. [29], in a review paper on research in technical debt, highlight that technical debt is a multi-faceted problem. Addressing it effectively in practice requires research in software evolution, risk management, qualitative assessment of context, software metrics, program analysis, and software quality. Applying technical debt research starts with identifying a project's sources of "pain." In order to give guidance to a specific project, research in this area must be grounded in the project's context.

This combination of diverse definitions of technical debt in research, alongside the need to ground research on technical debt in practice, raised three research questions:

1. To what extent do practitioners have a commonly shared definition of technical debt?
2. How much of technical debt is architectural in nature?
3. What management practices and tools are used in industry to deal with debt?

To investigate these questions, we conducted a two-part study. First, we administered a survey of software professionals in three large organizations, with 1,831 responses; second, we held semistructured, follow-up interviews with seven respondents, all professional software engineers, to further investigate the emerging themes.

Ernst et al, ESEC/FSE 2015



RQ2. Are issues with architectural elements among the most significant sources of technical debt?

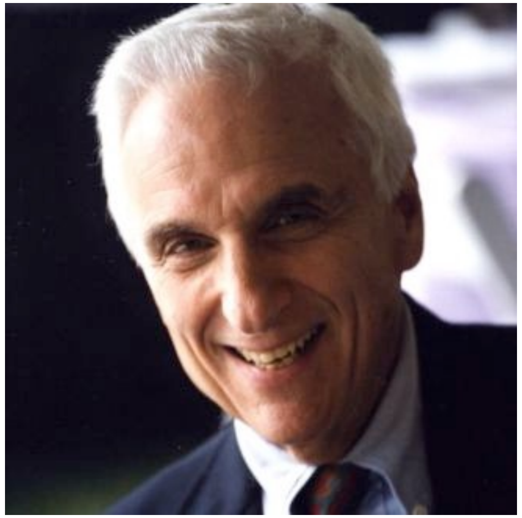
Finding 3: Architectural issues are the greatest source of technical debt.

Finding 4: Architectural issues are difficult to deal with, since they were often caused many years previously.

Finding 5: Monitoring and tracking drift from original design and rationale are vital.

Beware: Debt is Relative

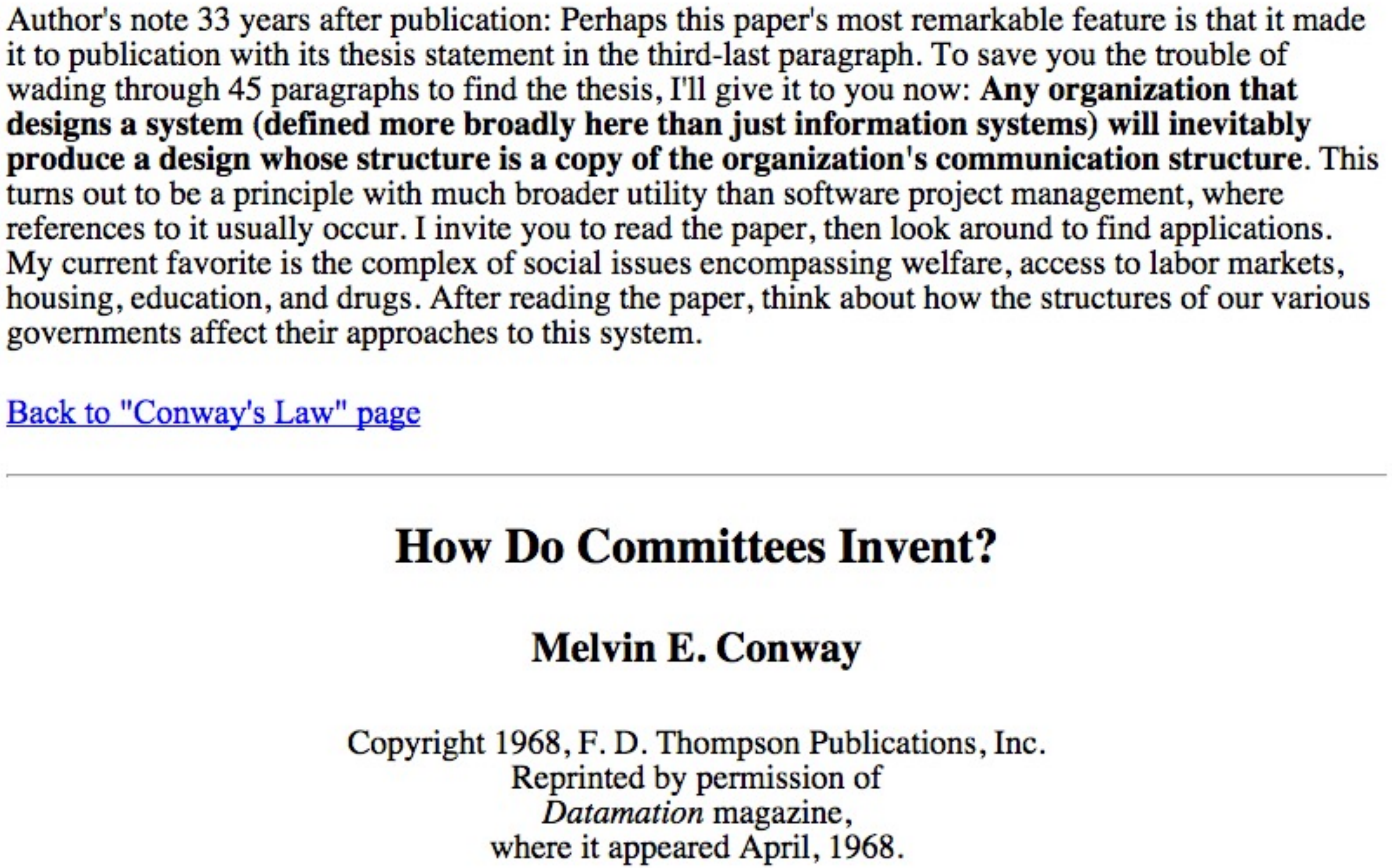
- *The refactoring effort needed to resolve issue non invasively*
  - Debt depends on features and issues to solve
  - Debt relevance depends on system’s roadmap
- Systems are used and society progresses
  - New libraries and versions come available, may make code obsolete
  - Actual usage affects our understanding of what matters
- Debt quantifications are only useful when they lead to *action.*
  - Rants / complaints that all code is bad are not helpful;
  - Propose rational action instead.



Conway’s Law

“Organizations which design systems ...
are constrained to produce designs
which are
copies of the communication structures
of these organizations”

Melvin Conway, 1968



Author's note 33 years after publication: Perhaps this paper's most remarkable feature is that it made it to publication with its thesis statement in the third-last paragraph. To save you the trouble of wading through 45 paragraphs to find the thesis, I'll give it to you now: **Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure**. This turns out to be a principle with much broader utility than software project management, where references to it usually occur. I invite you to read the paper, then look around to find applications. My current favorite is the complex of social issues encompassing welfare, access to labor markets, housing, education, and drugs. After reading the paper, think about how the structures of our various governments affect their approaches to this system.

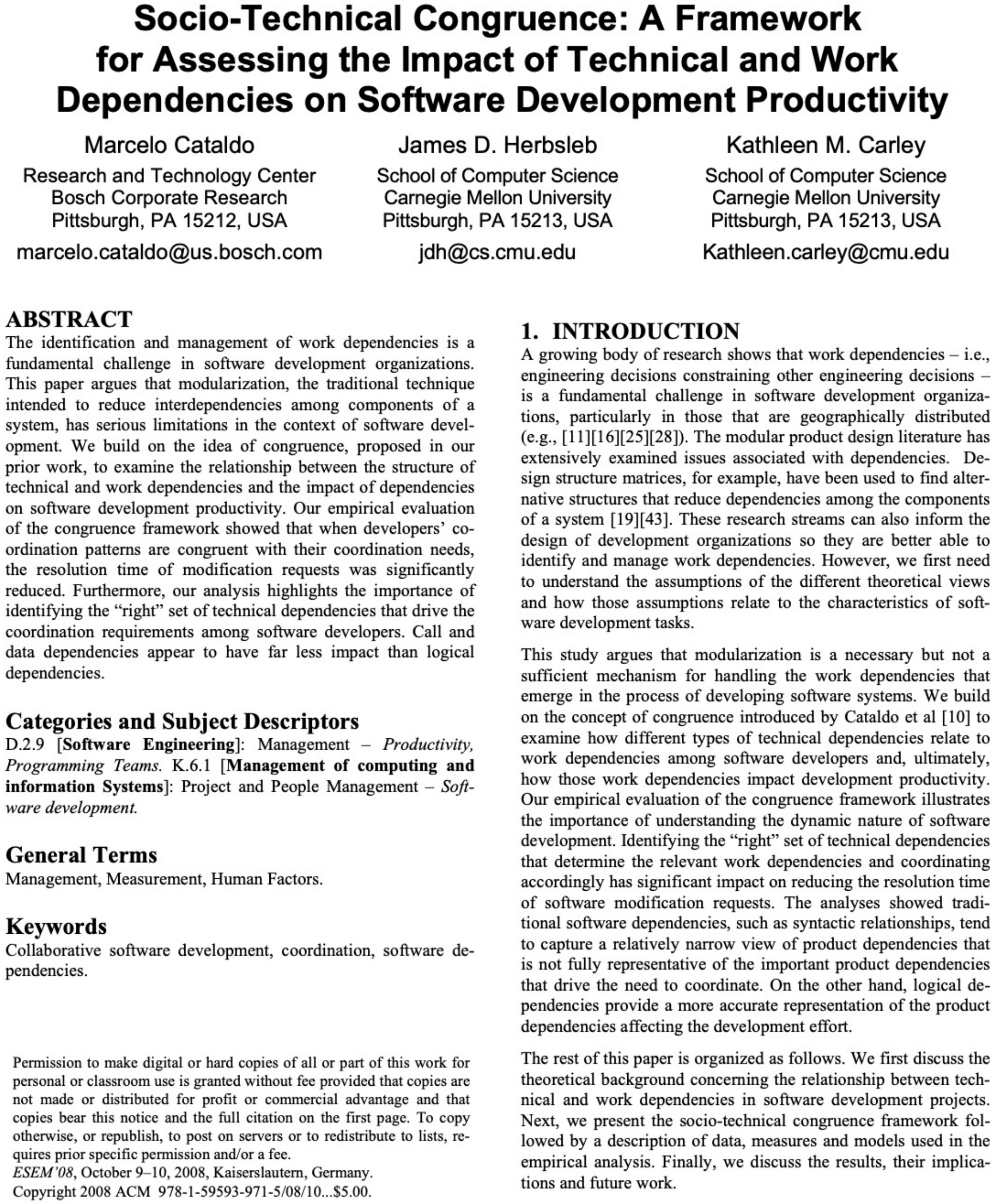
Back to "Conway's Law" page

How Do Committees Invent?

Melvin E. Conway

Socio-Technical Congruence

- Are technical dependencies aligned with organizational dependencies?
- Who is talking to whom?
- Does component-to-team allocation affect productivity?



Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity

Marcelo Cataldo
Research and Technology Center
Bosch Corporate Research
Pittsburgh, PA 15212, USA
marcelo.cataldo@us.bosch.com

James D. Herbsleb
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jdh@cs.cmu.edu

Kathleen M. Carley
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Kathleen.carley@cmu.edu

ABSTRACT

The identification and management of work dependencies is a fundamental challenge in software development organizations. This paper argues that modularization, the traditional technique intended to reduce interdependencies among components of a system, has serious limitations in the context of software development. We build on the idea of congruence, proposed in our prior work, to examine the relationship between the structure of technical and work dependencies and the impact of dependencies on software development productivity. Our empirical evaluation of the congruence framework showed that when developers' coordination patterns are congruent with their coordination needs, the resolution time of modification requests was significantly reduced. Furthermore, our analysis highlights the importance of identifying the "right" set of technical dependencies that drive the coordination requirements among software developers. Call and data dependencies appear to have far less impact than logical dependencies.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *Productivity, Programming Teams.* K.6.1 [**Management of computing and information Systems**]: Project and People Management – *Software development.*

General Terms

Management, Measurement, Human Factors.

Keywords

Collaborative software development, coordination, software dependencies.

1. INTRODUCTION

A growing body of research shows that work dependencies – i.e., engineering decisions constraining other engineering decisions – is a fundamental challenge in software development organizations, particularly in those that are geographically distributed (e.g., [11][16][25][28]). The modular product design literature has extensively examined issues associated with dependencies. Design structure matrices, for example, have been used to find alternative structures that reduce dependencies among the components of a system [19][43]. These research streams can also inform the design of development organizations so they are better able to identify and manage work dependencies. However, we first need to understand the assumptions of the different theoretical views and how those assumptions relate to the characteristics of software development tasks.

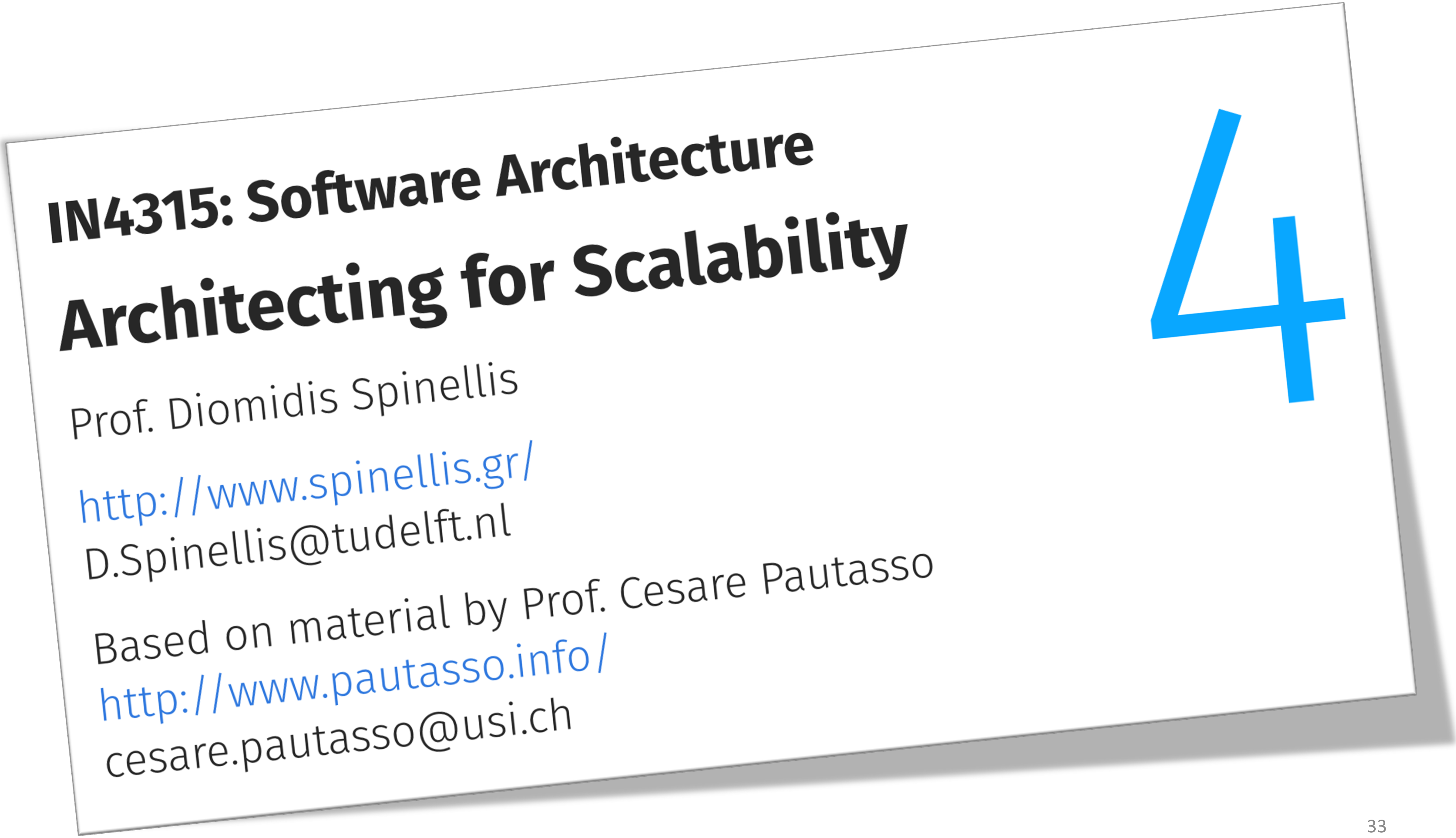
This study argues that modularization is a necessary but not a sufficient mechanism for handling the work dependencies that emerge in the process of developing software systems. We build on the concept of congruence introduced by Cataldo et al [10] to examine how different types of technical dependencies relate to work dependencies among software developers and, ultimately, how those work dependencies impact development productivity. Our empirical evaluation of the congruence framework illustrates the importance of understanding the dynamic nature of software development. Identifying the "right" set of technical dependencies that determine the relevant work dependencies and coordinating accordingly has significant impact on reducing the resolution time of software modification requests. The analyses showed traditional software dependencies, such as syntactic relationships, tend to capture a relatively narrow view of product dependencies that is not fully representative of the important product dependencies that drive the need to coordinate. On the other hand, logical dependencies provide a more accurate representation of the product dependencies affecting the development effort.

The rest of this paper is organized as follows. We first discuss the theoretical background concerning the relationship between technical and work dependencies in software development projects. Next, we present the socio-technical congruence framework followed by a description of data, measures and models used in the empirical analysis. Finally, we discuss the results, their implications and future work.

Essay 3: Quality and Evolution

With key aspects of the architecture described, the third essay focuses on means to safeguard the quality and architectural integrity of the underlying system, with special empahsis on the rate of change. Aspects to take into account include:

1. The system's key quality attributes and the degree to which they are currently satisfied
2. The overall software quality processes that apply to your system
3. The key elements of the system's continuous integration processes
4. The rigor of the test processes and the role of test coverage
5. Hotspot components from the past (previously changed a lot) and the future (needed for roadmap)
6. The code quality, with a focus on hotspot components
7. The quality culture, as evidenced in actual discussions and tests taking place in architecturally significant feature and pull requests (identify and analyze at least 10 such issues and 10 such pull requests)
8. An assessment of technical debt present in the system.



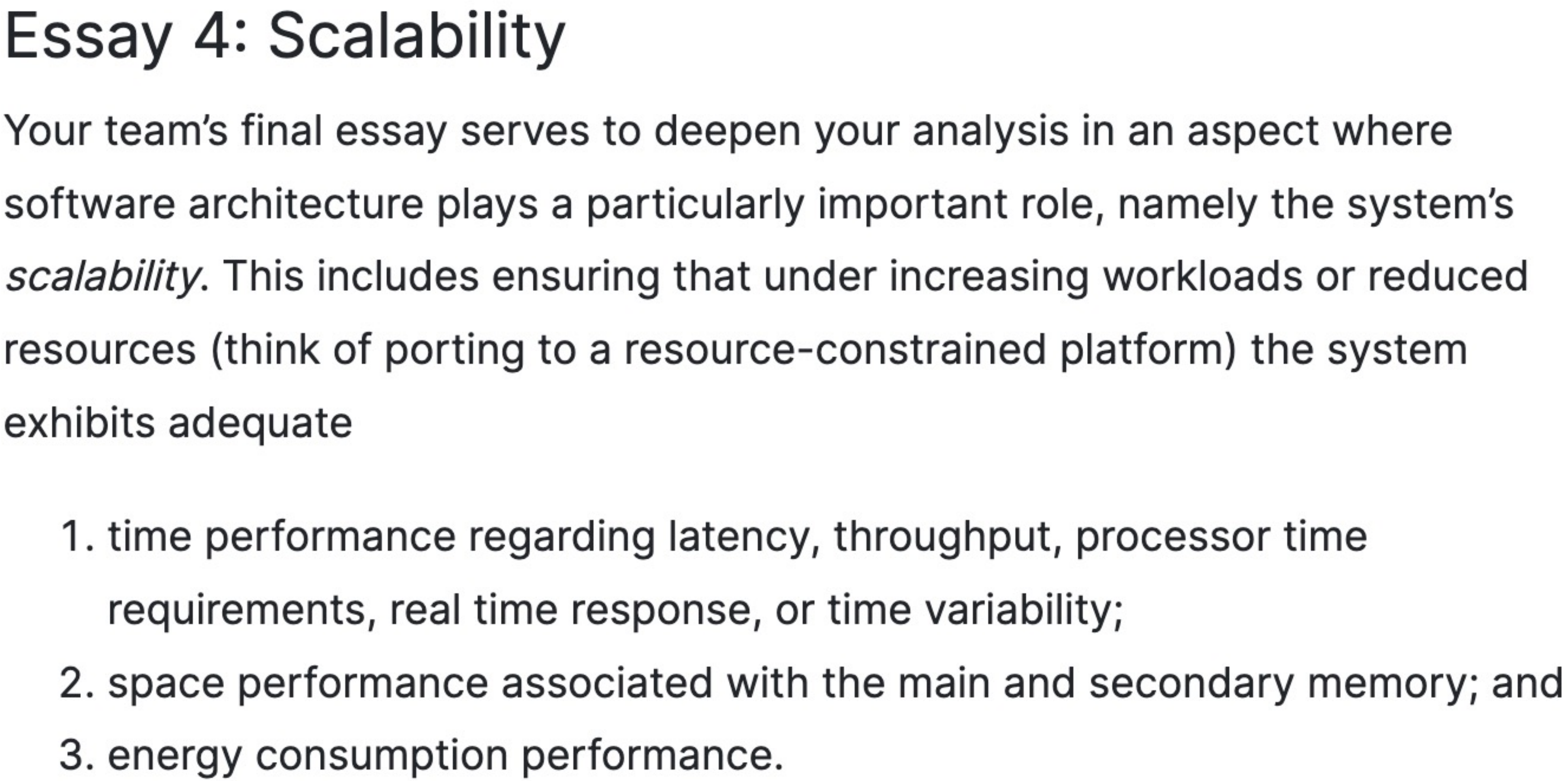
IN4315: Software Architecture

Architecting for Scalability

Prof. Diomidis Spinellis

http://www.spinellis.gr/
D.Spinellis@tudelft.nl

Based on material by Prof. Cesare Pautasso
http://www.pautasso.info/
cesare.pautasso@usi.ch

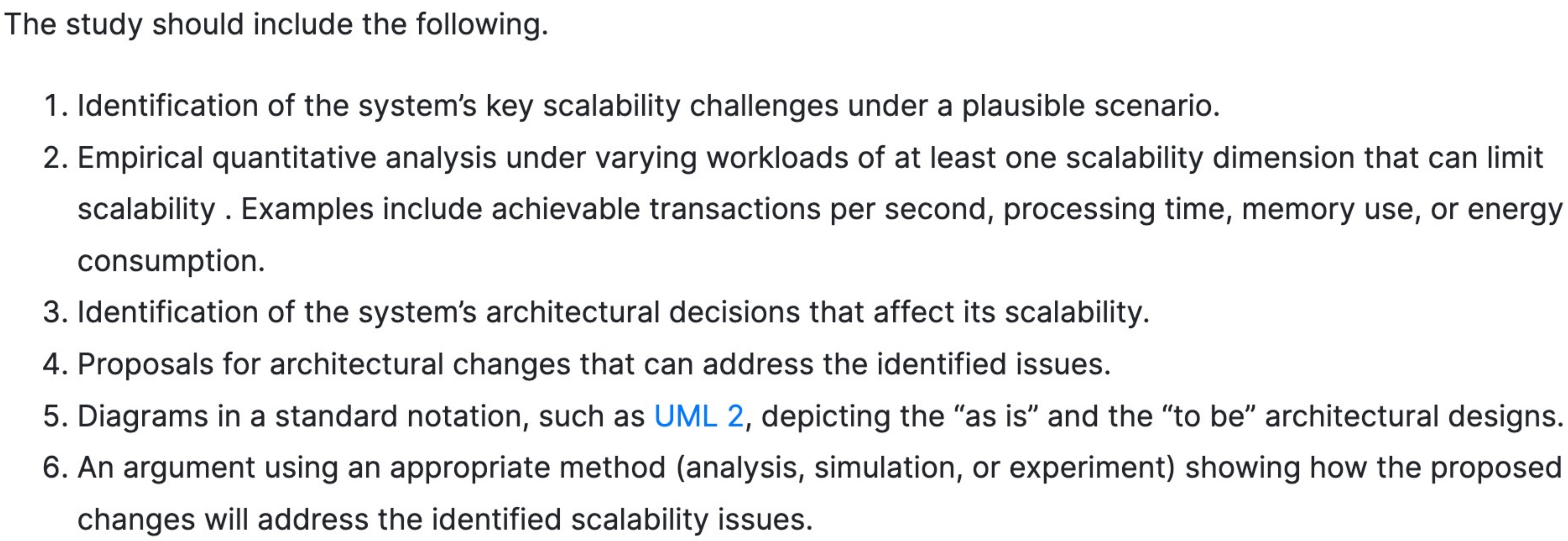


Essay 4: Scalability

Your team’s final essay serves to deepen your analysis in an aspect where software architecture plays a particularly important role, namely the system’s *scalability*. This includes ensuring that under increasing workloads or reduced resources (think of porting to a resource-constrained platform) the system exhibits adequate

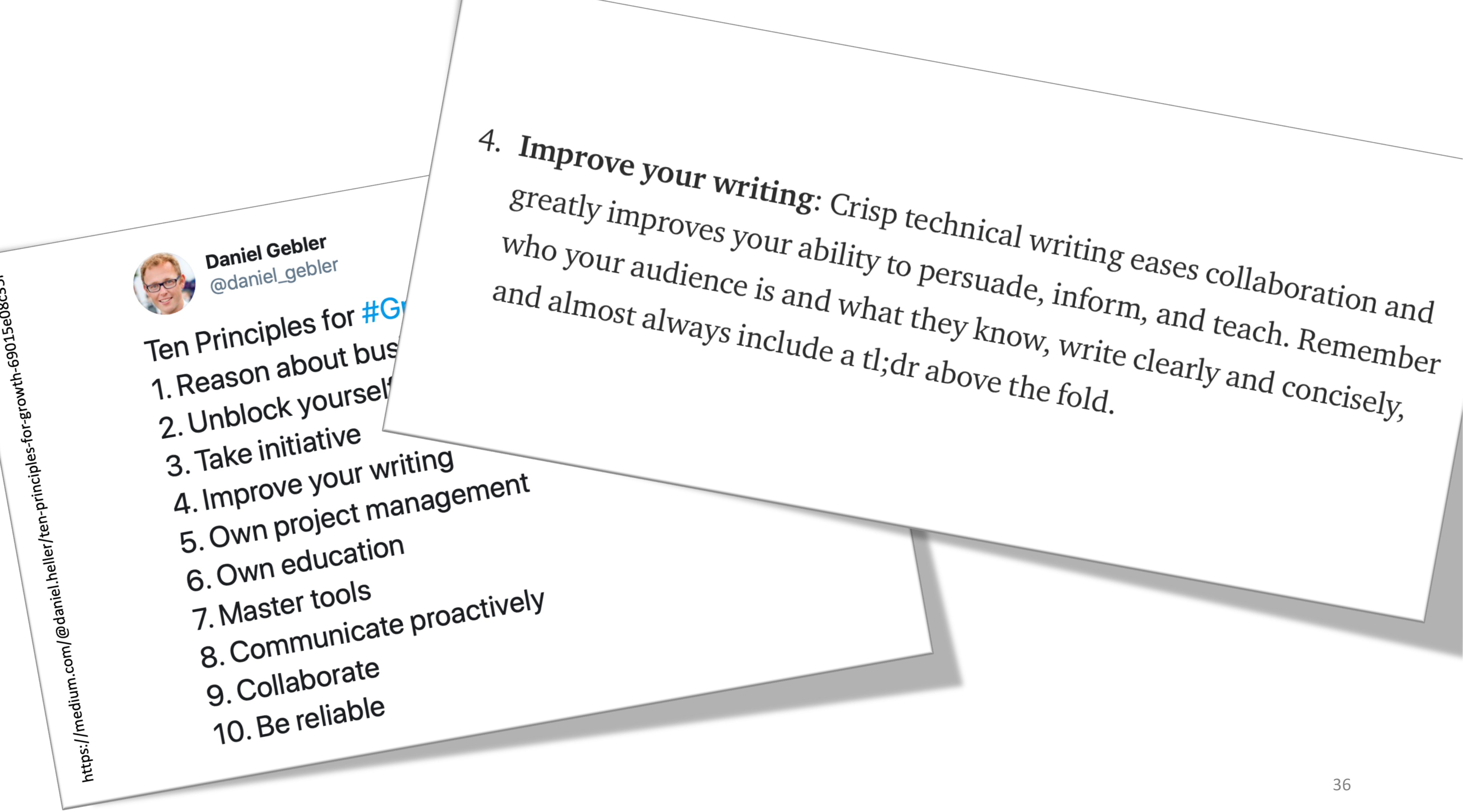
1. time performance regarding latency, throughput, processor time requirements, real time response, or time variability;
2. space performance associated with the main and secondary memory; and
3. energy consumption performance.

Essay 4: Scalability



The study should include the following.

1. Identification of the system’s key scalability challenges under a plausible scenario.
2. Empirical quantitative analysis under varying workloads of at least one scalability dimension that can limit scalability . Examples include achievable transactions per second, processing time, memory use, or energy consumption.
3. Identification of the system’s architectural decisions that affect its scalability.
4. Proposals for architectural changes that can address the identified issues.
5. Diagrams in a standard notation, such as UML 2, depicting the “as is” and the “to be” architectural designs.
6. An argument using an appropriate method (analysis, simulation, or experiment) showing how the proposed changes will address the identified scalability issues.

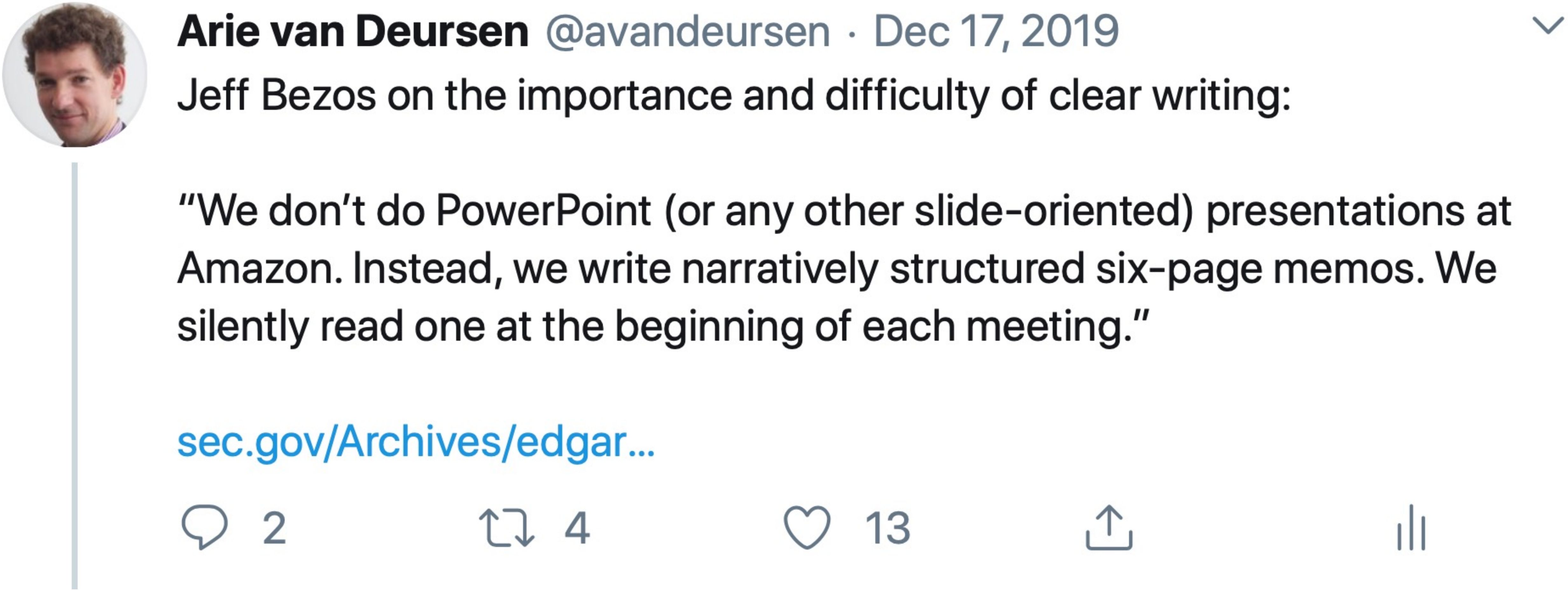


Daniel Gebler
@daniel_gebler

Ten Principles for #G
1. Reason about bus
2. Unblock yoursel
3. Take initiative
4. Improve your writing
5. Own project management
6. Own education
7. Master tools
8. Communicate proactively
9. Collaborate
10. Be reliable

4. **Improve your writing**: Crisp technical writing eases collaboration and greatly improves your ability to persuade, inform, and teach. Remember who your audience is and what they know, write clearly and concisely, and almost always include a tl;dr above the fold.

https://medium.com/@daniel.heller/ten-principles-for-growth-69015e08c53



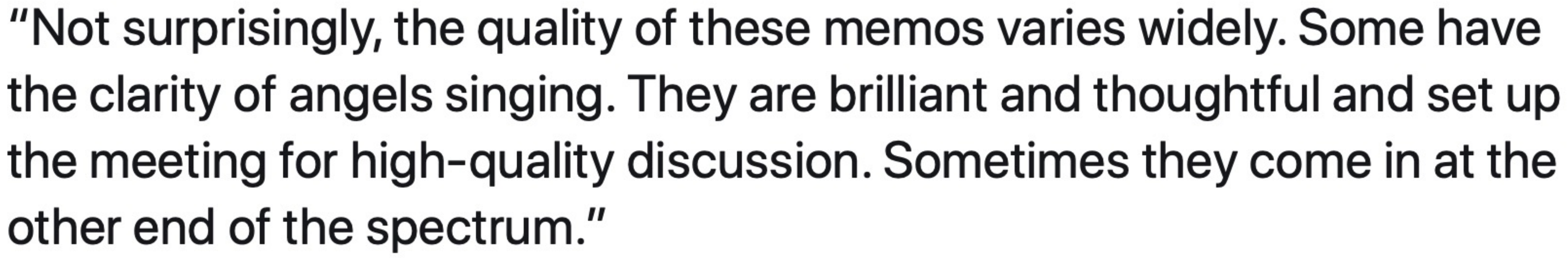
Arie van Deursen @avandeursen · Dec 17, 2019

Jeff Bezos on the importance and difficulty of clear writing:

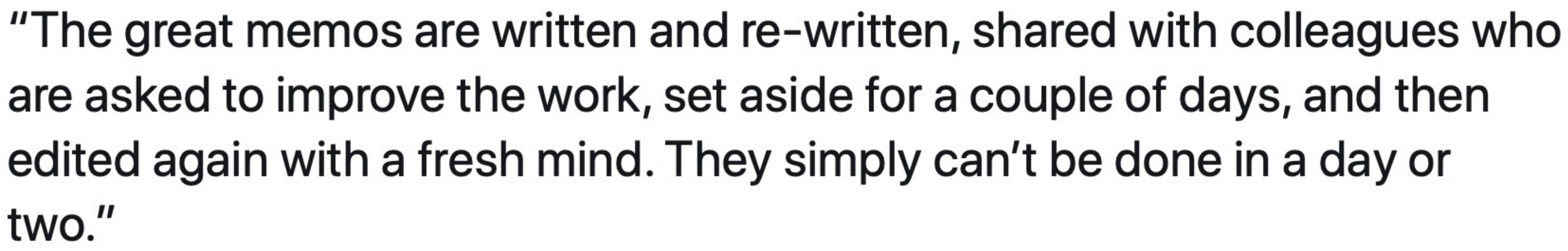
“We don’t do PowerPoint (or any other slide-oriented) presentations at Amazon. Instead, we write narratively structured six-page memos. We silently read one at the beginning of each meeting.”

sec.gov/Archives/edgar…

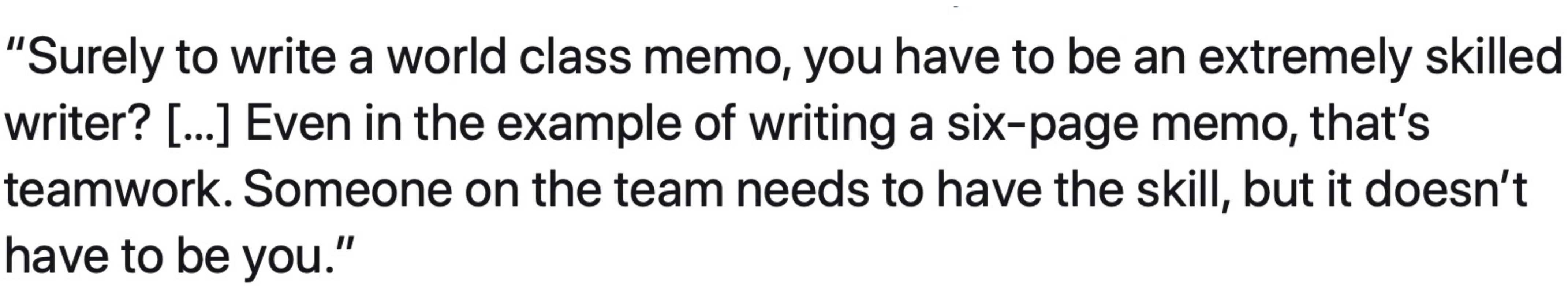
https://www.sec.gov/Archives/edgar/data/1018724/000119312518121161/d456916dex991.htm



“Not surprisingly, the quality of these memos varies widely. Some have the clarity of angels singing. They are brilliant and thoughtful and set up the meeting for high-quality discussion. Sometimes they come in at the other end of the spectrum.”



“The great memos are written and re-written, shared with colleagues who are asked to improve the work, set aside for a couple of days, and then edited again with a fresh mind. They simply can’t be done in a day or two.”



“Surely to write a world class memo, you have to be an extremely skilled writer? […] Even in the example of writing a six-page memo, that’s teamwork. Someone on the team needs to have the skill, but it doesn’t have to be you.”

https://100x.engineering/the-power-of-the-narrative/

The Science of Scientific Writing

- Stress position:
  - The **end** of a sentence
  - Save the best for the last
- The **topic** position:
  - The **start** of the sentence
  - Gives meaning to what will come
  - Builds on / is connected to preceding arguments
- Connect sentences, paragraphs, chapters like this

1. The backward-linking old information appears in the topic position.
2. The person, thing or concept whose story it is, appears in the topic position.
3. The new, emphasis-worthy information appears in the stress position.

https://www.americanscientist.org/blog/the-long-view/the-science-of-scientific-writing

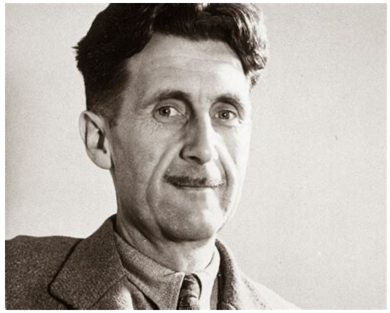
Four Roles in the Writing Process

- **Madman:** Full of ideas, writes crazily and perhaps rather sloppily, gets carried away by enthusiasm or anger, and if really let loose, could turn out ten pages an hour.
- **Architect:** Select large chunks of material and arrange them in a pattern that might form an argument. The thinking is large, organizational, paragraph-level --- the architect doesn't worry about sentence structure.
- **Carpenter:** Nails these ideas together in a logical sequence, making sure each sentence is clearly written, contributes to the argument of the paragraph, and leads logically and gracefully to the next sentence.
- **Judge:** Punctuation, spelling, grammar, tone --- all the details which result in a polished essay

http://www.ut-ie.com/b/b_flowers.html

Samantha Power

George Orwell



"Crisp, clear writing is essential to communicating on behalf of oneself and one's causes.

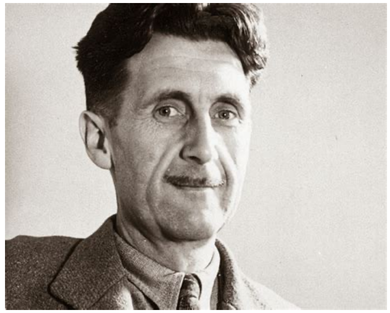
Vague expressions, euphemisms, and jargon are often manifestations of not being entirely sure of one's point or purpose, and they hold us back.

In 1946 Orwell was so exasperated by the debasement of language he saw around him that he wrote a short pamphlet with guidelines for precision.

I reread it every year as a reminder to
'never use a long word when a short one will do', and to
'let the meaning choose the word, and not the other way around.'"

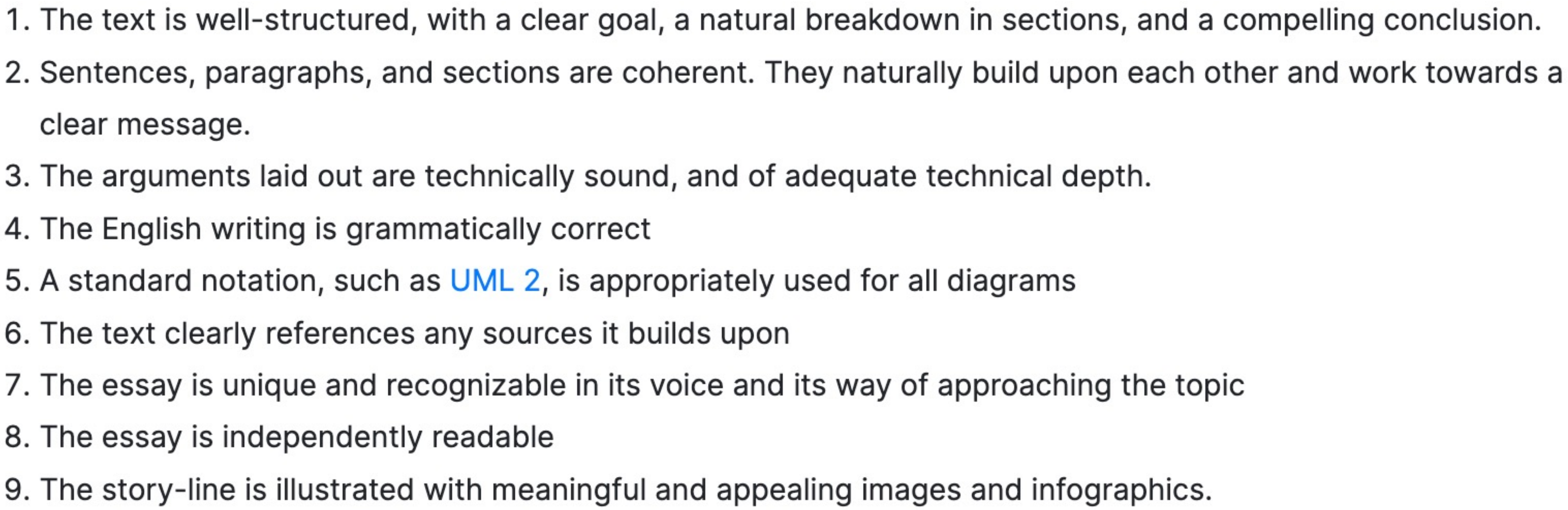
https://www.glamour.com/gallery/books-international-day-of-the-girl

Orwell on Writing



- Never use a long word where a short one will do.
- If it is possible to cut a word out, always cut it out.
- Never use a foreign phrase, a scientific word or a jargon word if you can think of an everyday English equivalent.
- Never use the passive where you can use the active.
- Never use a metaphor, simile or other figure of speech which you are used to seeing in print.
- Break any of these rules sooner than say anything barbarous.

Essay Evaluation Criteria



1. The text is well-structured, with a clear goal, a natural breakdown in sections, and a compelling conclusion.
2. Sentences, paragraphs, and sections are coherent. They naturally build upon each other and work towards a clear message.
3. The arguments laid out are technically sound, and of adequate technical depth.
4. The English writing is grammatically correct
5. A standard notation, such as UML 2, is appropriately used for all diagrams
6. The text clearly references any sources it builds upon
7. The essay is unique and recognizable in its voice and its way of approaching the topic
8. The essay is independently readable
9. The story-line is illustrated with meaningful and appealing images and infographics.

Essay Peer Review

Peer Review

- Objective 1: *Learn* from other essay
- Objective 2: Give other team *feedback*

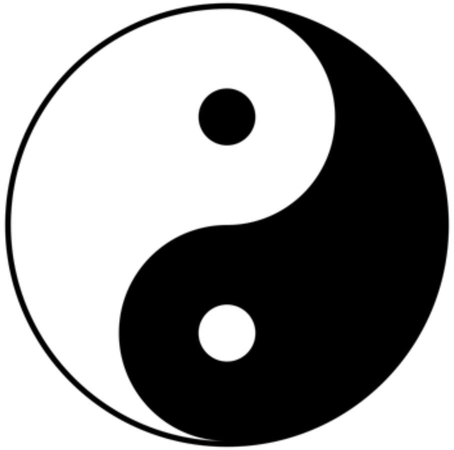
- Open questions (free text):
  - How you understood the essay (the key take-aways)
  - Strengths and points for improvement
- Closed questions (Likert scale):
  - Specific questions on scale from 1-5

Recommendation: Allocate 4 hours per review

Dialectic Learning in Architecture

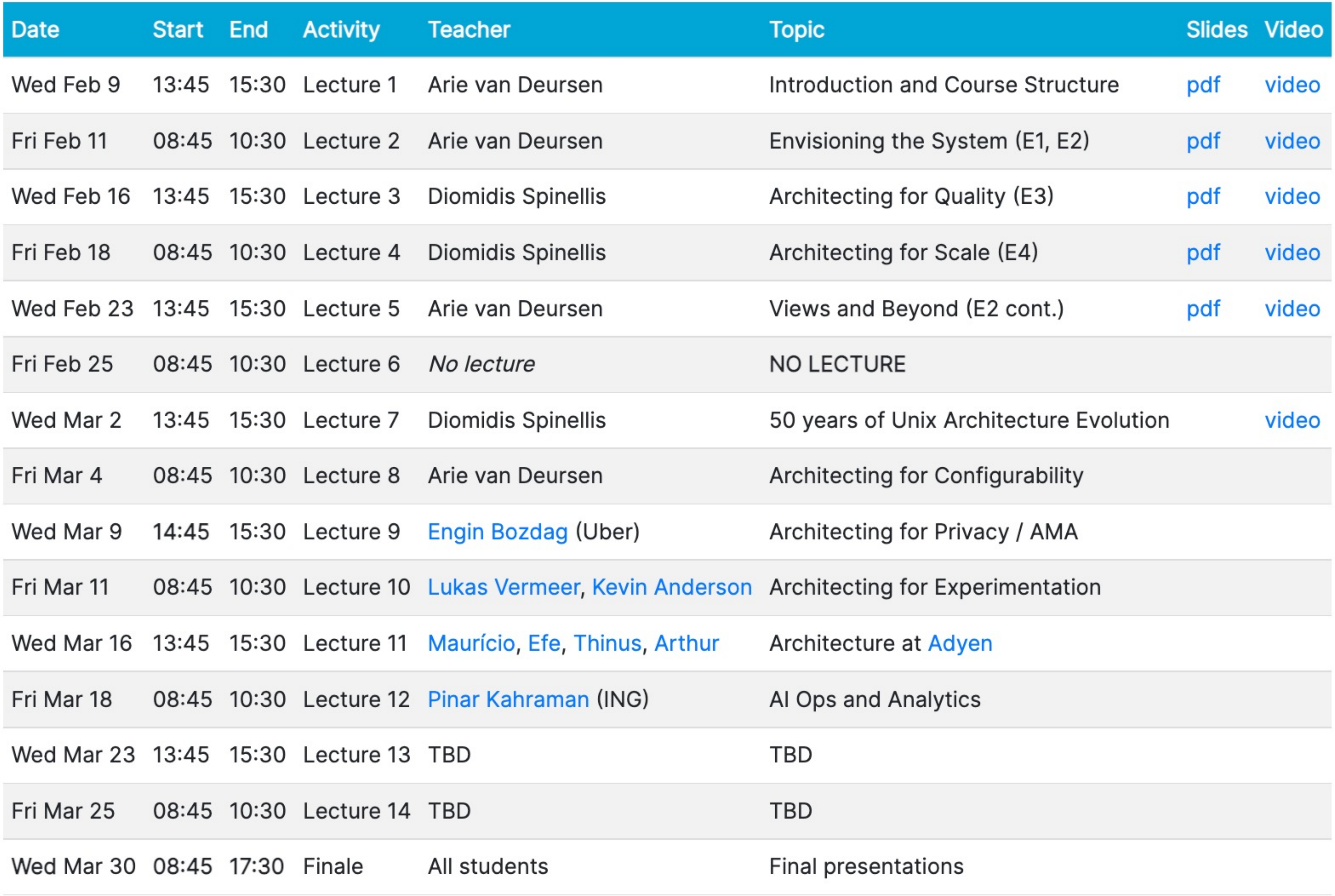
1. Just do it: Engage in architectural activities in realistic setting
2. Study / *internalize* existing theories and approaches
3. Confront the two with each other
   - How does this theory really work?
   - Does this theory apply to my system? Why? Why not?

Thesis:	Theory
Anti-Thesis:	Practice
Synthesis:	Understanding



Q&A

- What did you learn from the coaches?
- What can others learn from you?
- What is unclear?
- How are the essays progressing?
- How are the contributions progressing?
- Do your Mattermost and journals reflect your true activity?
- …



Date	Start	End	Activity	Teacher	Topic	Slides	Video
Wed Feb 9	13:45	15:30	Lecture 1	Arie van Deursen	Introduction and Course Structure	pdf	video
Fri Feb 11	08:45	10:30	Lecture 2	Arie van Deursen	Envisioning the System (E1, E2)	pdf	video
Wed Feb 16	13:45	15:30	Lecture 3	Diomidis Spinellis	Architecting for Quality (E3)	pdf	video
Fri Feb 18	08:45	10:30	Lecture 4	Diomidis Spinellis	Architecting for Scale (E4)	pdf	video
Wed Feb 23	13:45	15:30	Lecture 5	Arie van Deursen	Views and Beyond (E2 cont.)	pdf	video
Fri Feb 25	08:45	10:30	Lecture 6	*No lecture*	NO LECTURE		
Wed Mar 2	13:45	15:30	Lecture 7	Diomidis Spinellis	50 years of Unix Architecture Evolution		video
Fri Mar 4	08:45	10:30	Lecture 8	Arie van Deursen	Architecting for Configurability		
Wed Mar 9	14:45	15:30	Lecture 9	Engin Bozdag (Uber)	Architecting for Privacy / AMA		
Fri Mar 11	08:45	10:30	Lecture 10	Lukas Vermeer, Kevin Anderson	Architecting for Experimentation		
Wed Mar 16	13:45	15:30	Lecture 11	Maurício, Efe, Thinus, Arthur	Architecture at Adyen		
Fri Mar 18	08:45	10:30	Lecture 12	Pinar Kahraman (ING)	AI Ops and Analytics		
Wed Mar 23	13:45	15:30	Lecture 13	TBD	TBD		
Fri Mar 25	08:45	10:30	Lecture 14	TBD	TBD		
Wed Mar 30	08:45	17:30	Finale	All students	Final presentations		