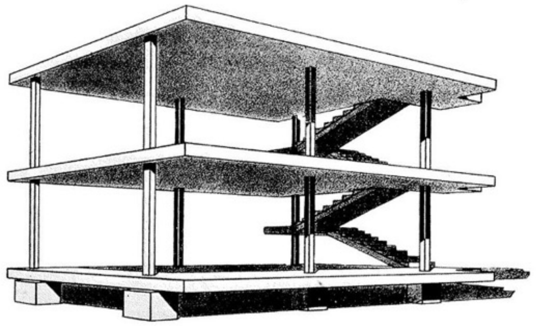
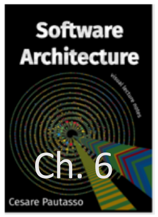
IN4315 Software Architecture Lecture 4: Architecting for Change

Arie van Deursen

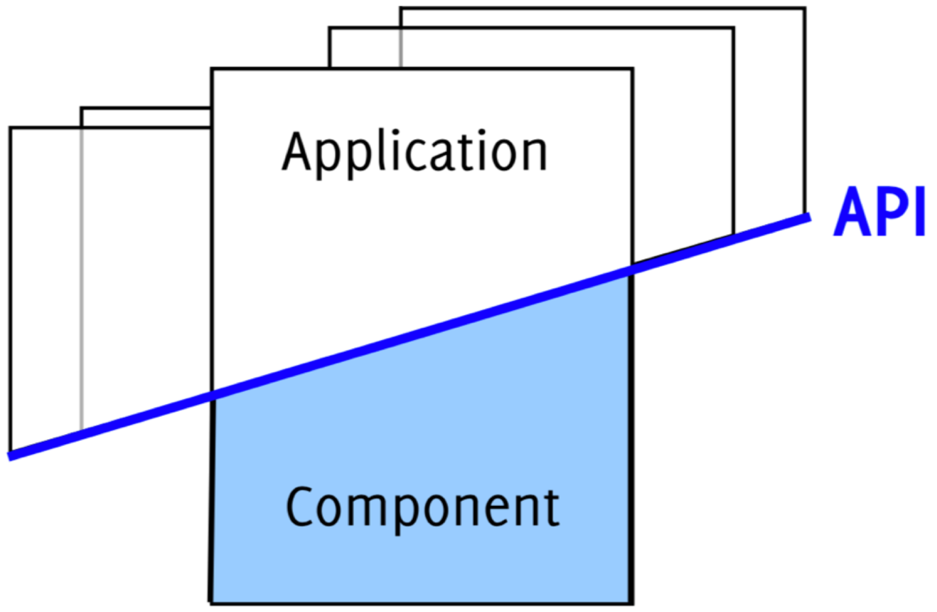


Wikipedia, Dom-Ino House, Corbusier

Application Programming Interfaces



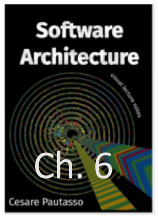
- APIs are not found in all architectures:
- APIs can be found in architectures that are designed to be
  - open and stable platforms
  - supporting externally developed components and applications.



API Design Principles: Your Answers!

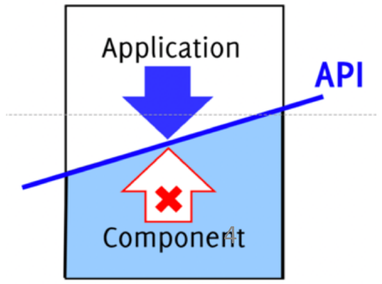
- Easy to understand
  - Usability
  - Simplicity
  - Small interfaces
- Quality of Service:
  - Scalability, Reliable, Available
- Compliance with standards
  - RESTful
- Licensing
- Naming consistency (end points, parameters, methods)
- Robust against untrusted clients
  - Security
  - Authentication
- Defensive API
- Meaningful error messages
- Compatibility

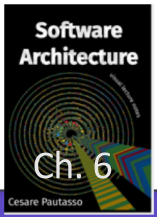
API Design Principles



- Explicit interfaces principle
- Principle of least surprise
- Small interfaces principle
- Uniform access principle
- Few interfaces principle
- Clear interfaces principle

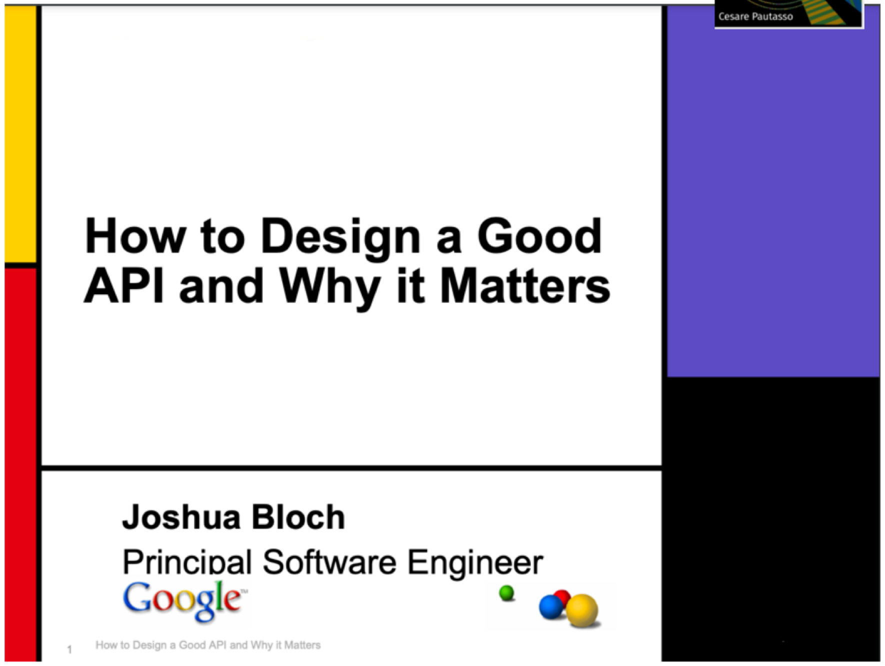
- Maximize information hiding
- 90% immediate use; 9% with effort; .9% misuse
- Balance usability and reusability
- Balance performance and reusability
- Design from client's perspective





Design Advice

- Keep it simple
  - Do One Thing and do it well
  - Do not surprise clients
- Keep it as small as possible but not smaller
  - When in doubt leave it out
  - You can always add more later
- Maximize information hiding
  - API First
  - Avoid leakage: implementation should not impact interface



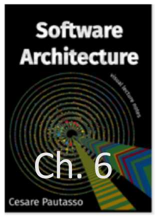
How to Design a Good API and Why it Matters

Joshua Bloch
Principal Software Engineer
Google

1 How to Design a Good API and Why it Matters

http://www.cs.bc.edu/~muller/teaching/cs102/s06/lib/pdf/api-design

Design Advice

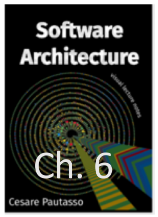


- Names Matter
  - Avoid cryptic acronyms
  - Use names consistently
- Internally Consistent
  - Naming Conventions
  - Argument Ordering
  - Return values
  - Error Handling
- Externally Consistent
  - Imitate similar APIs
  - Follow the conventions of the underlying platform

Joshua Bloch

https://www.youtube.com/watch?v=aAb7hSCtvGw

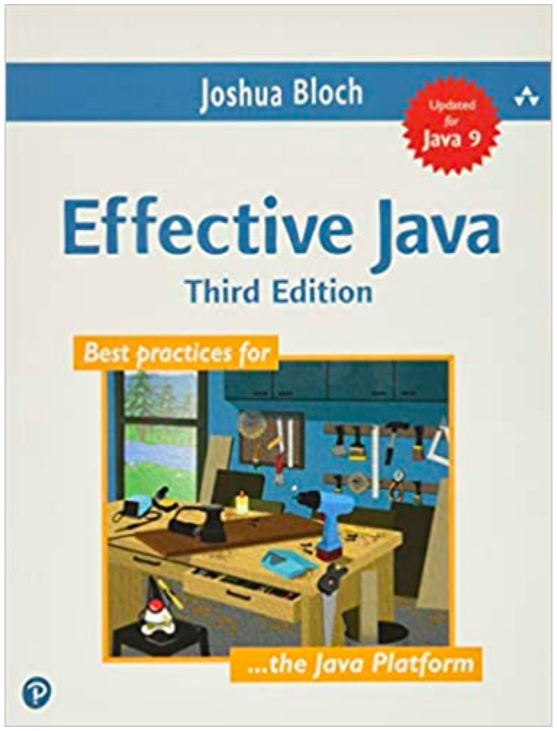
Design Advice



- Document Everything
  - Classes, Methods, Parameters
  - Include Correct Usage Examples
  - Quality of Documentation critical for success

- Make it easy to learn and easy to use
  - without having to read too much documentation
  - by copying examples

- Make it hard to misuse

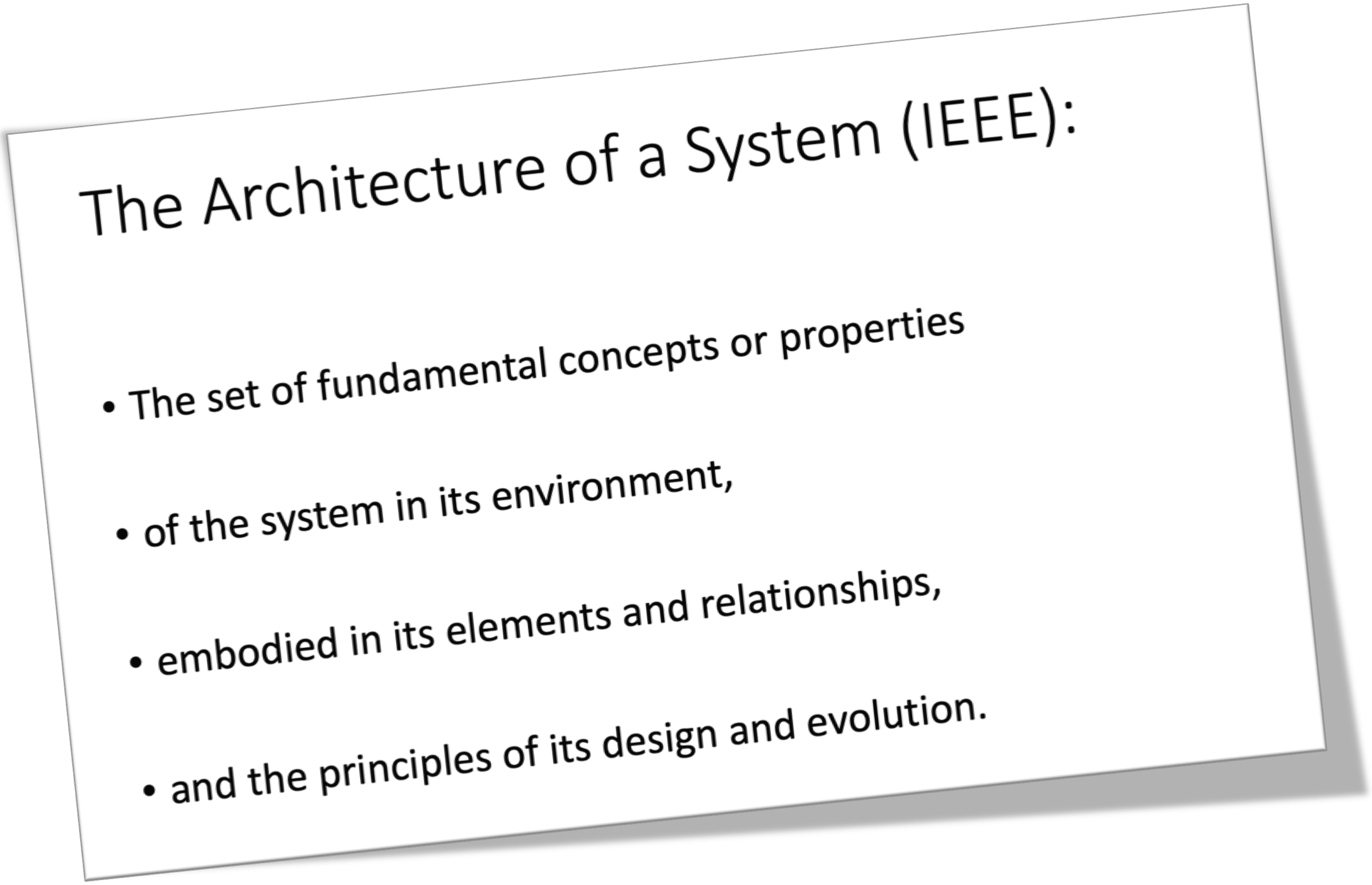


API Reflection

- Consider an application you know well
- Which public APIs does it expose?
- Does the API realize a clear, compelling function?
- Which of the principles discussed does it adhere to explicitly?
  - Which ones does it violate?
- Is the design rationale behind the API documented?

Essay 2: The System’s Architecture

1. The main architectural style or patterns applied (if relevant), such as layering or model-view-controller architectures.
2. Containers view: The main execution environments, if applicable, as used to deploy the system.
3. Components view: Structural decomposition into components with explicit interfaces, and their inter-dependencies
4. Connectors view: Main types of connectors used between components / containers.
5. Development view, covering the system decomposition and the main modules and their dependencies, as embodied in the source code.
6. Run time view, indicating how components interact at run time to realize key scenarios, including typical run time dependencies
7. How the architecture realizes key quality attributes, and how potential trade-offs between them have been resolved.
8. API design principles applied

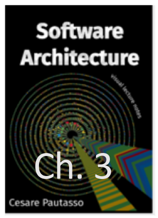


The Architecture of a System (IEEE):

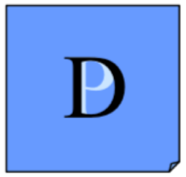
- The set of fundamental concepts or properties
- of the system in its environment,
- embodied in its elements and relationships,
- and the principles of its design and evolution.

Software Evolution

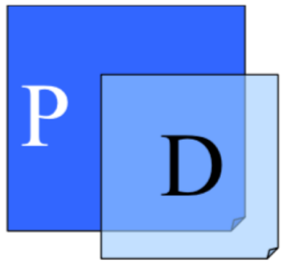
- Evolution is what makes software valuable
- Software success generates ideas for new system usage
  - Business opportunities
  - Integration with other systems
  - Legal constraints
- But ... lots of evolution may erode the system



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

Quality Assurance

- Collection of processes put in place to ensure that system continues to meet pre-set quality objectives
- Safety critical / business critical domains often highly regulated
  - Health, (self-) driving, aviation, finance, ...
- In other domains / open source this can be more ad hoc
  - Evolution speed / rate of change can be key driver
  - Need to ensure that evolution itself does not reduce (later) rate of change

Architecting for High Rates of Change

- Automate everything
- Clean code
- Coding standards
- Static analysis tools
- Code review
- Architectural integrity
- Continuous integration / delivery / deployment
- Software testing

Testing Activities (I)

- Testability as explicit quality attribute
- Testing as the *guide* that helps to reach a good design
- 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?
- Test coverage:
  - What types of coverage are monitored?
  - What is the coverage of key components?
  - How is test coverage information *used*?

Testing Activities (II)

- 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
- The setup of the test suites:
  - Test cases run on every commit (duration?)
  - Tailored test cases for performance, portability?
- Coding standards / style of the test code
  - Structuring into packages

Pull Request / Issue Ethnography

- Issues (or PRs) that involve many people
    - Likely to have substantial architectural consequences
    - Will uncover key designs
    - May reveal their *rationale*

- A good architecture should help guide such discussions

Hotspot Components / Centers of Evolution

- Hotspot components: involved in many code changes.
- Hotspot may be consequence of poor design
  - Monolithic class that no one dares to break up
- Hotspots of the past:
  - Likely to have suffered from too many changes
- Hotspots of the future:
  - Worth investing in their quality NOW!
  - Map future use cases from roadmap onto logical view

What Is Technical Debt?

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

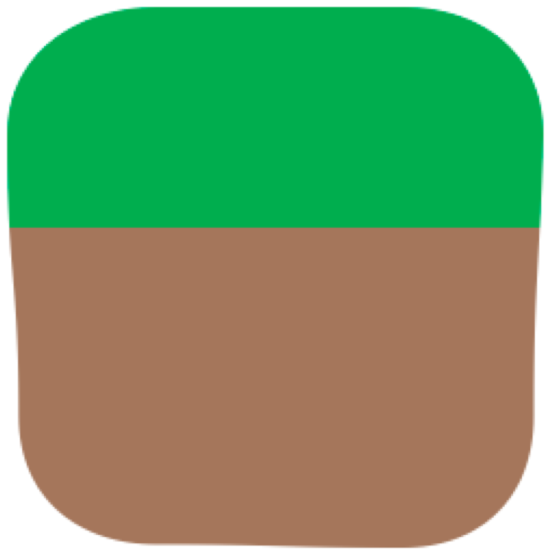
http://c2.com/cgi/wiki?WardExplainsDebtMetaphor

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

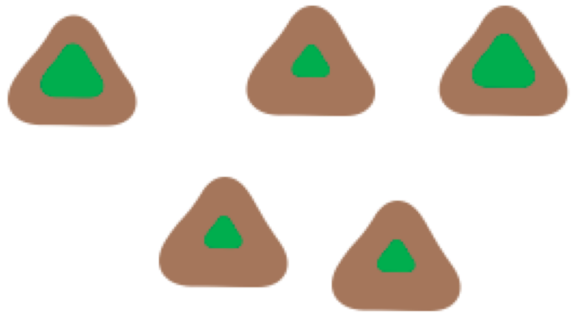
https://www.youtube.com/watch?v=7hL6g1aTGvo

*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

- 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"

sonarqube

JArchitect

http://www.sqale.org/

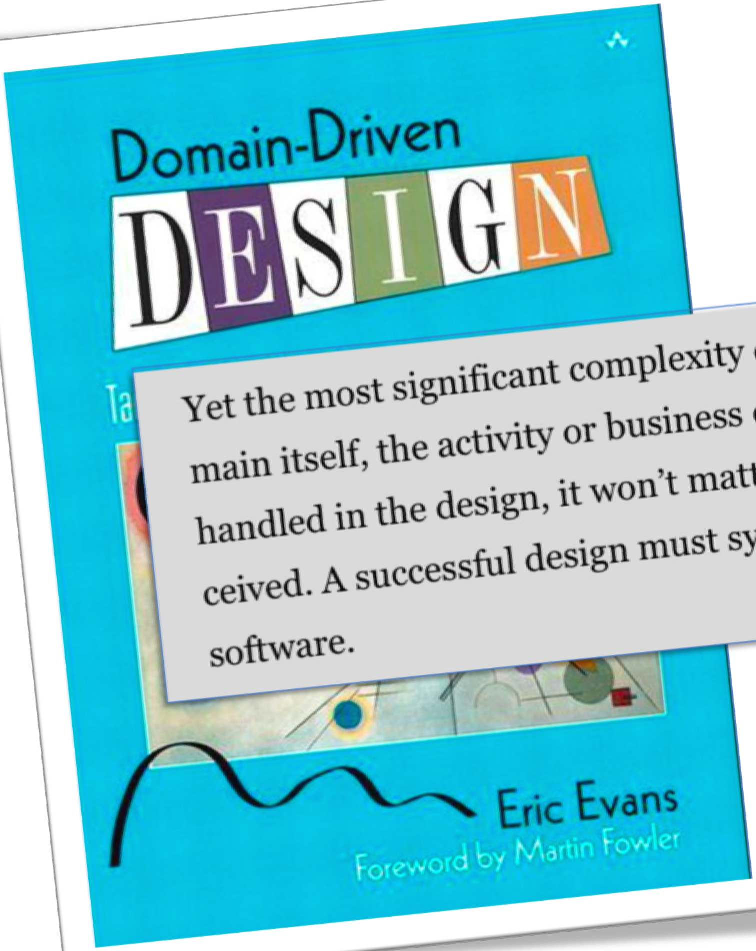
Software Improvement Group

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.

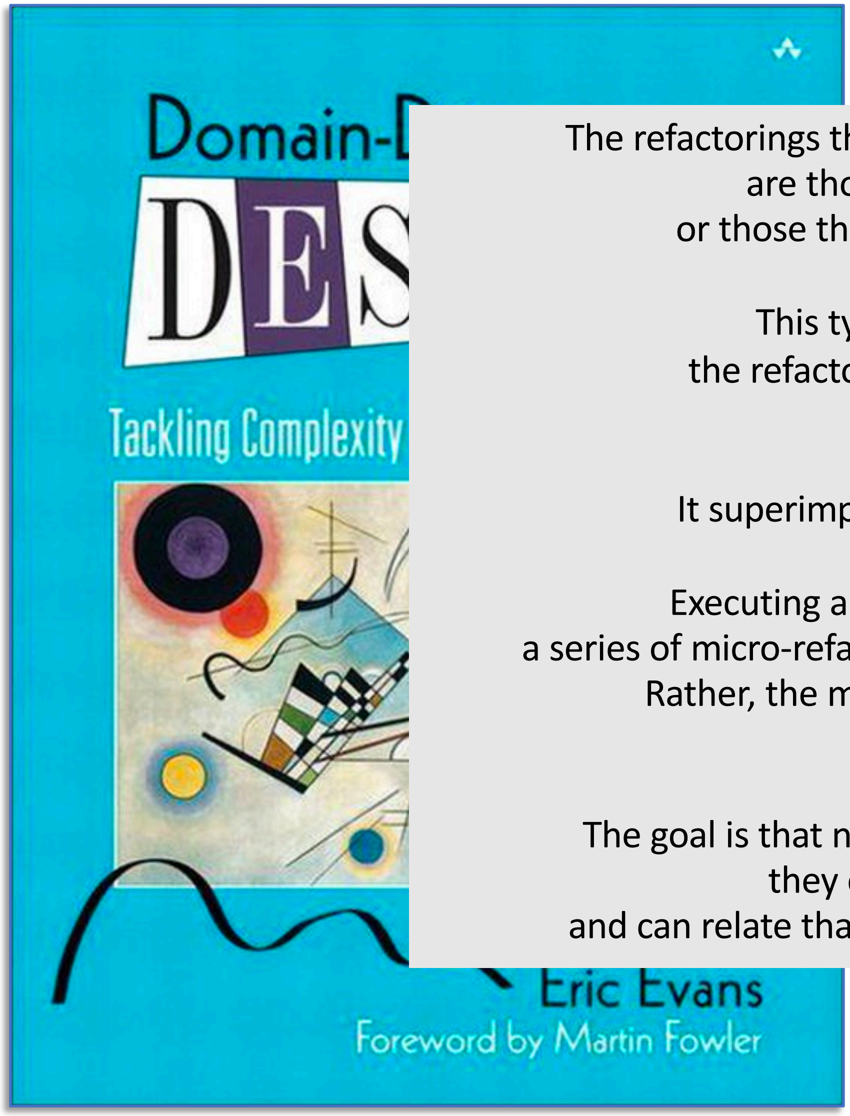
<u>I'm never in favor of writing code poorly</u>.

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.

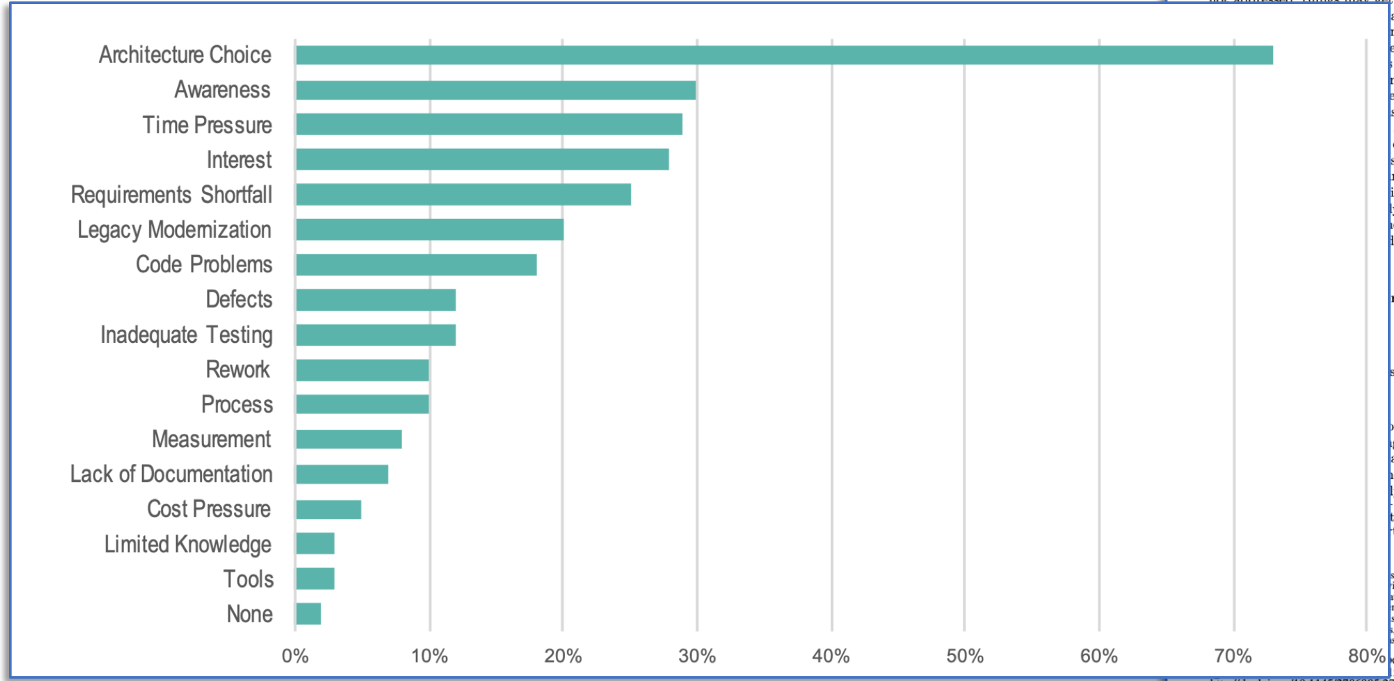
Executing a refactoring based on domain insight often involves
a series of micro-refactorings, but the motivation is not just the state of the code.
Rather, the micro-refactorings provide convenient units of change
toward a more insightful model.

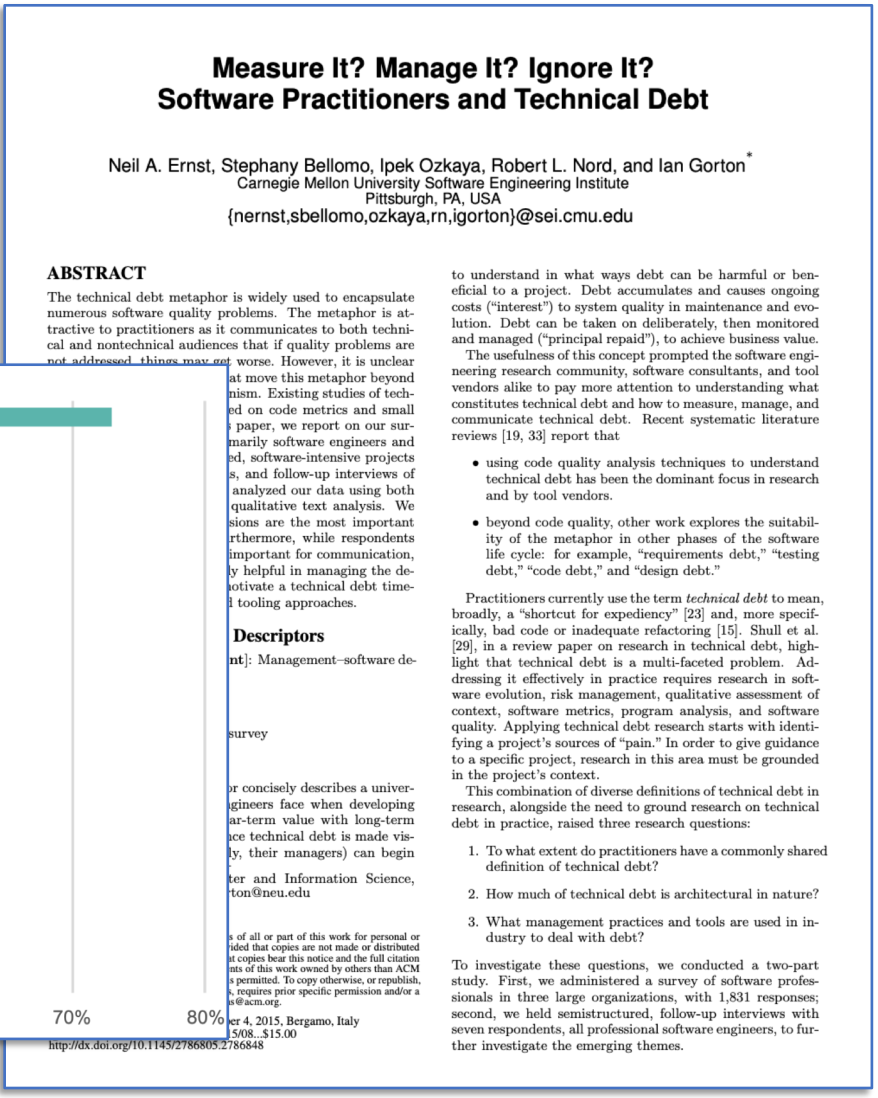
The goal is that not only can a developer understand what the code does;
they can also understand *why* it does what it does
and can relate that to the ongoing communication with the domain experts

	Visible	Invisible
Positive Value	New features Added functionality	Architectural, Structural features
Negative Value	Defects	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,
y helpful in managing the de-
otivate a technical debt time-
l tooling approaches.

Descriptors

nt]: Management–software de-

survey

r 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

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.

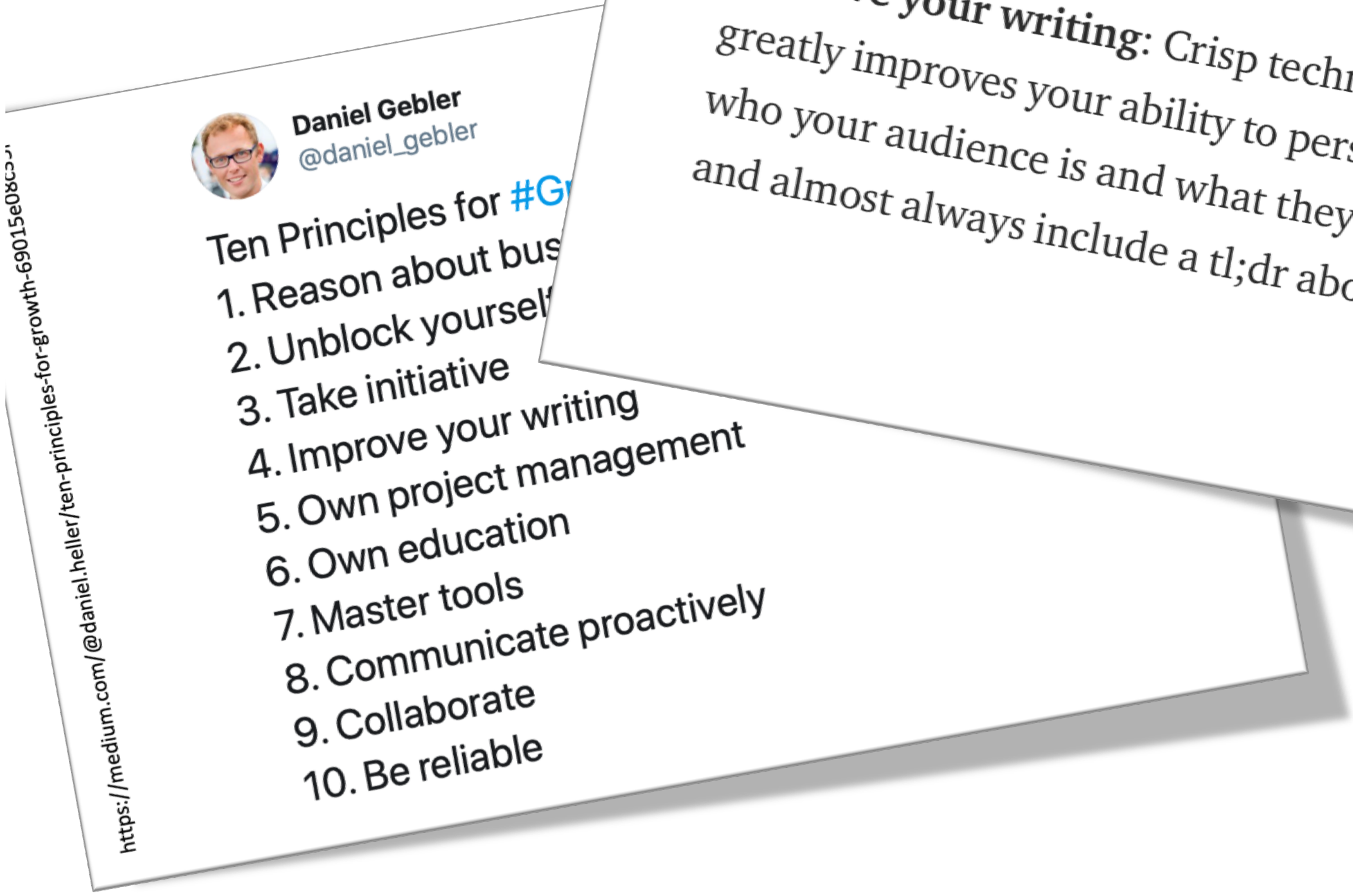
Essay 3: Quality and Evolution

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 overall software quality processes that apply to your system
2. The key elements of the system's continuous integration processes
3. The rigor of the test processes and the role of test coverage
4. Hotspot components from the past (previously changed a lot) and the future (needed for roadmap)
5. The code quality, with a focus on hotspot components
6. 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)
7. An assessment of technical debt present in the system.

Essay 4

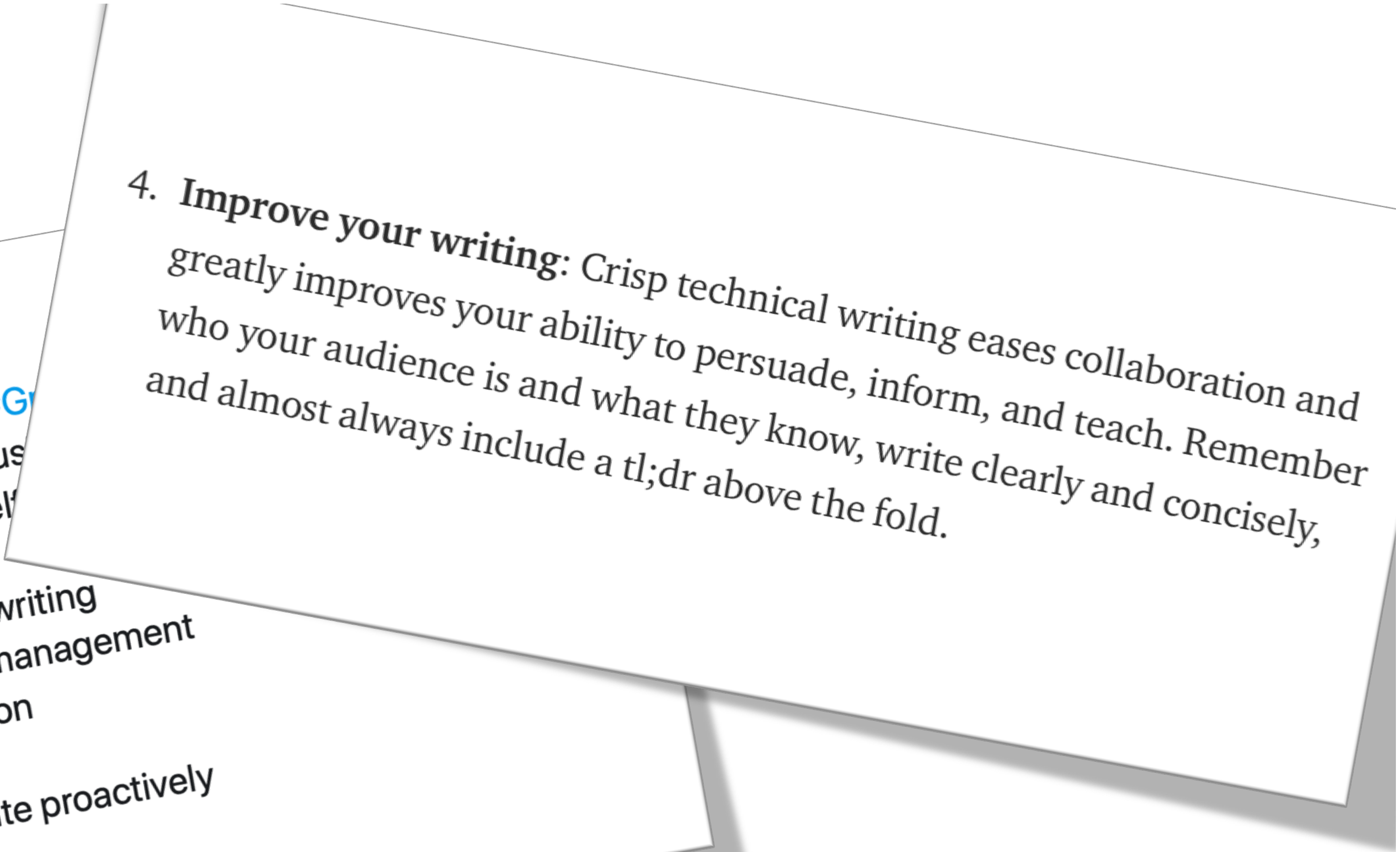
- 4.SUS: Sustainability Analysis (lecture Luís Cruz)
- 4.DIS: Distribution Analysis (Lecture Burcu Kulahcioglu Ozkan)
- 4.VAR: Variability Analysis (Lecture Xavier Devroey)
- 4.AMD: Get an ARCHITECTURE.md file merged (HT Aleksey Kladov)
- 4.OTH: Topic of choice



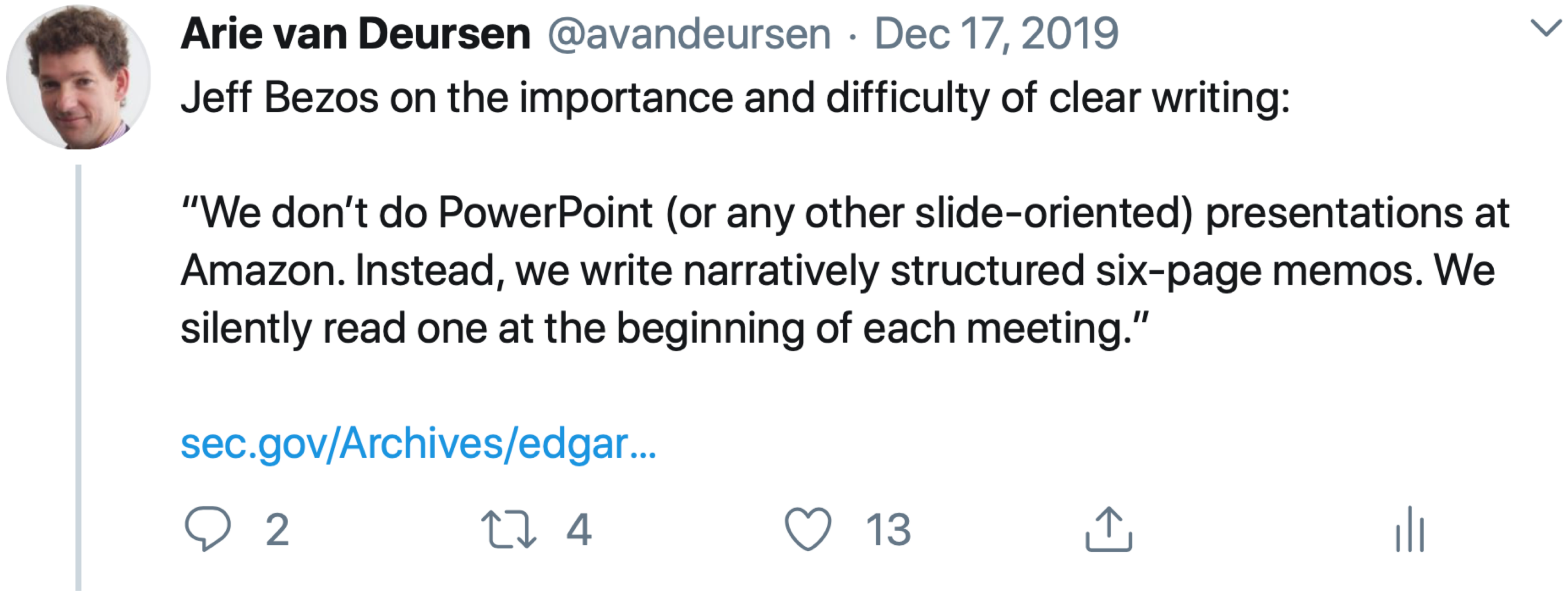
https://medium.com/@daniel.heller/ten-principles-for-growth-69015e08c

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.


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…

amazon

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.

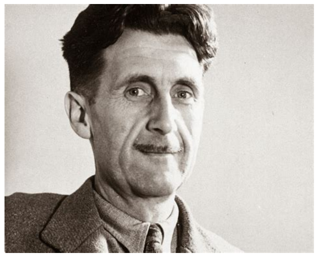
Four Roles in the Writing Process

- **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

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

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

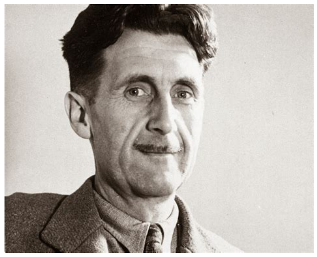


Vague expressions, euphemisms, and jargon are often manifestations of not being entirely sure of one’s point or purpose, and they hold us back.

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.

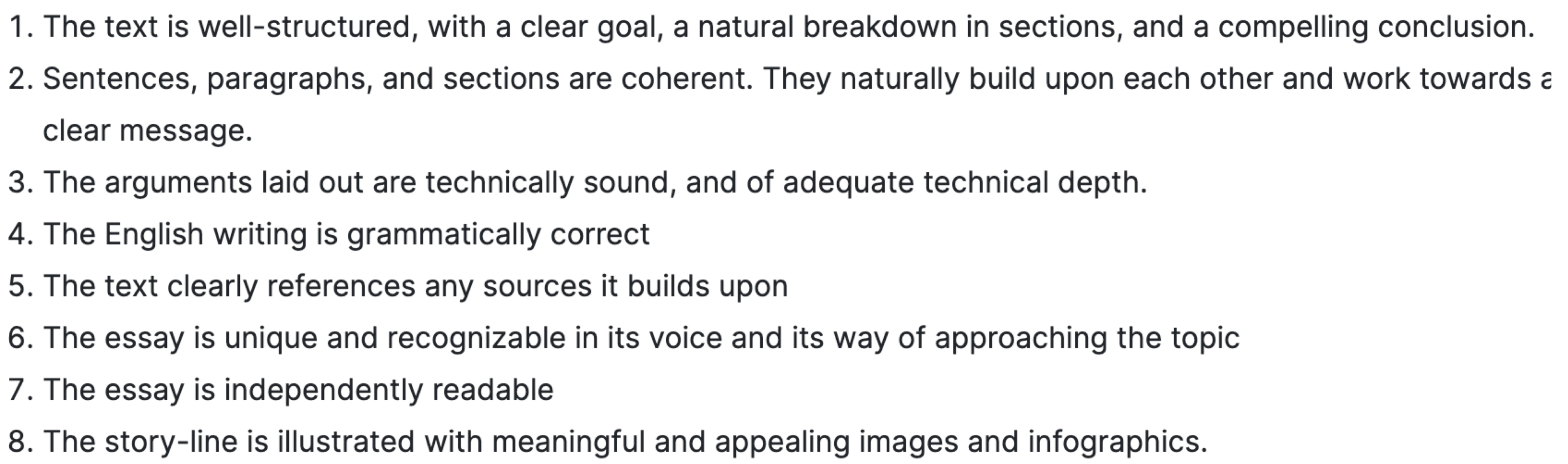
I reread it every year as a reminder to
'never use a long word when a short one will do’
and to
'let the meaning choose the word, and not the other way around.'"

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. The text clearly references any sources it builds upon
6. The essay is unique and recognizable in its voice and its way of approaching the topic
7. The essay is independently readable
8. 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