Beyond Functional

by Paul Slaughter

Photo by Juan Sisinni on Unsplash

Paul Slaughter

Title: Senior Frontend Engineer @ GitLab

Education: Masters in Software Engineering

Interests: Programming, music, art, writing, BJJ

conventional: comments

Comments that are easy to grok and grep

https://conventionalcomments.org

osfn()

// The Open Source Function //

https://www.youtube.com/channel/UCTleK5BarSReiZMxZSvnNQA

https://souldzin.com/

@souldzin or @pslaughter

@souldzin

@souldzin

Requirement: When user hits the “Play” button, selected song plays.

https://en.wikipedia.org/wiki/File:Zune80and4.jpg

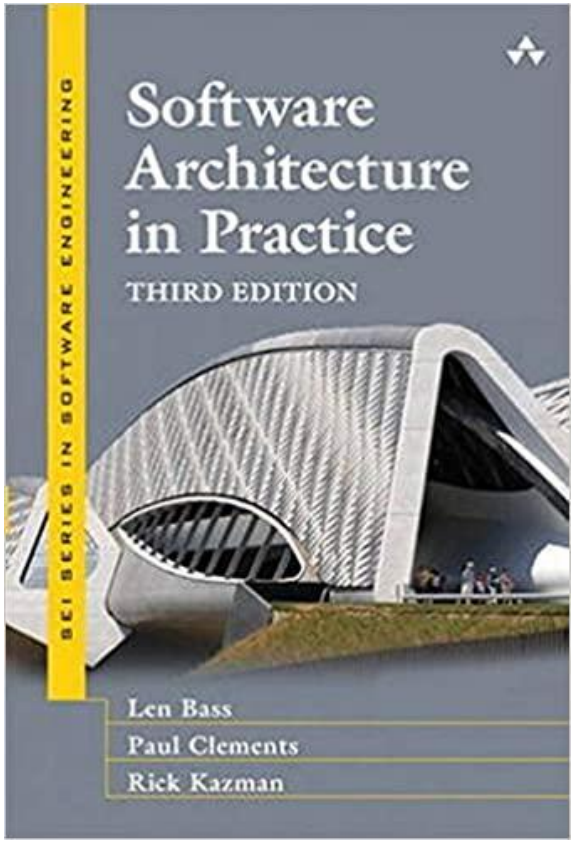
https://commons.wikimedia.org/wiki/File:Ipodclassic80gb.jpg

Same requirements...

What separates a ***poor*** implementation from a ***loveable*** one?

Sometimes the user doesn't even know!

Photo by Rayson Tan on Unsplash



From Chapter 4:

"...many factors determine the qualities that must be provided for in a system's architecture. The qualities ***go beyond functionality***..."

"Systems are frequently redesigned not because they are functionally deficient... but because they are difficult to maintain, port, or scale; or the are too slow; or they have been compromised by hackers."

Software Development Activities

Requirements issues are the most expensive!

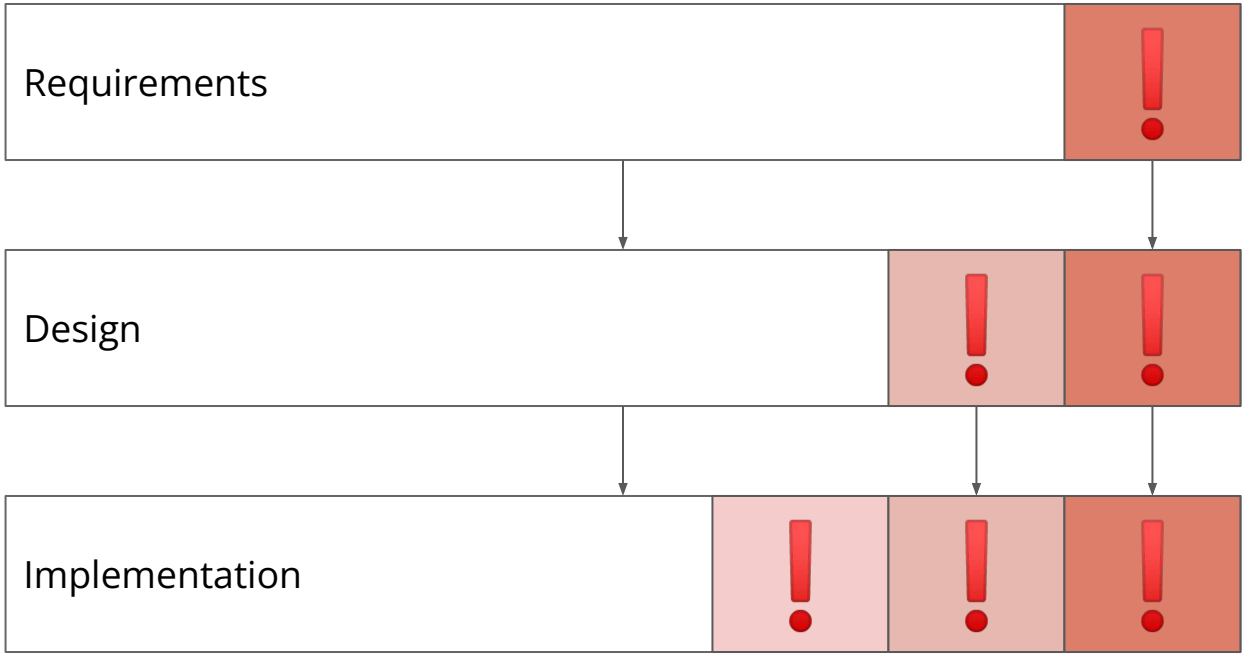
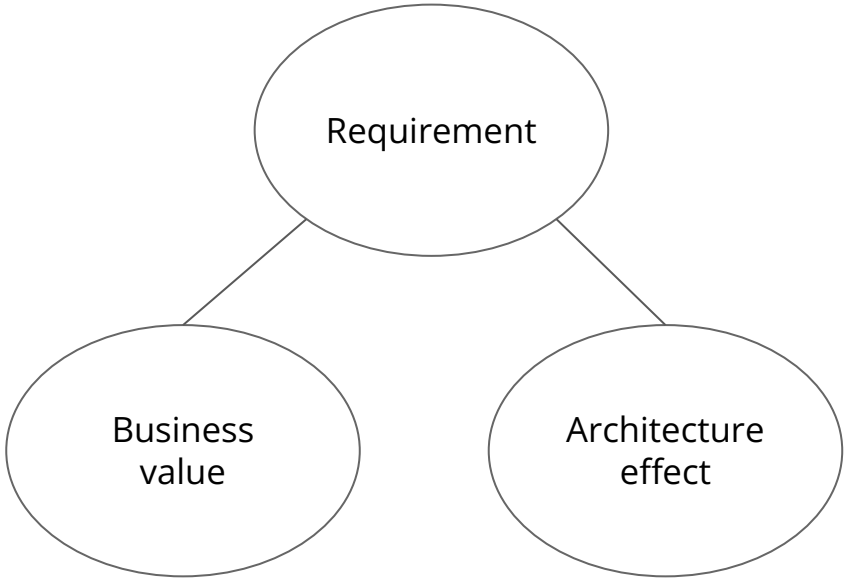


Photo by Mia Baker on Unsplash



Not all requirements are the same!

What if we missed a high "Value" + high "Architecture" requirement? What about a high "Value" + low "Architecture" requirement?

Functional Requirements

What the system does. Specific ***behavior*** of the system.

Non-Functional Requirements (AKA Quality Attributes)

How the system does the thing. A ***trait*** of the system's behavior.

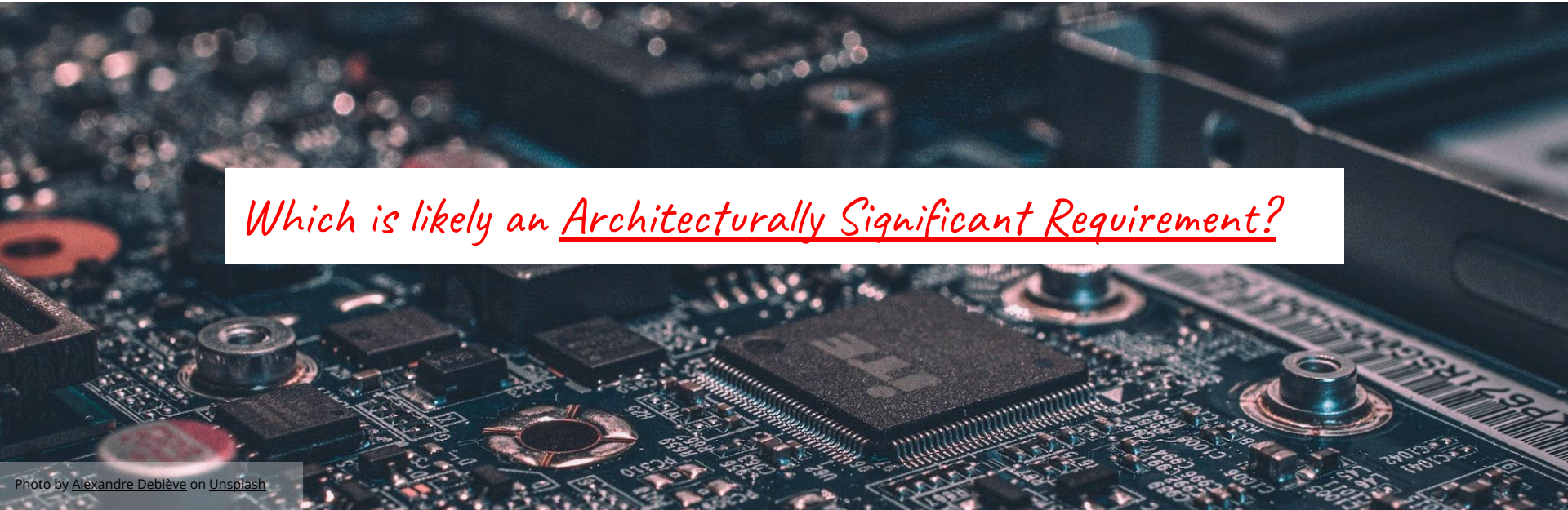
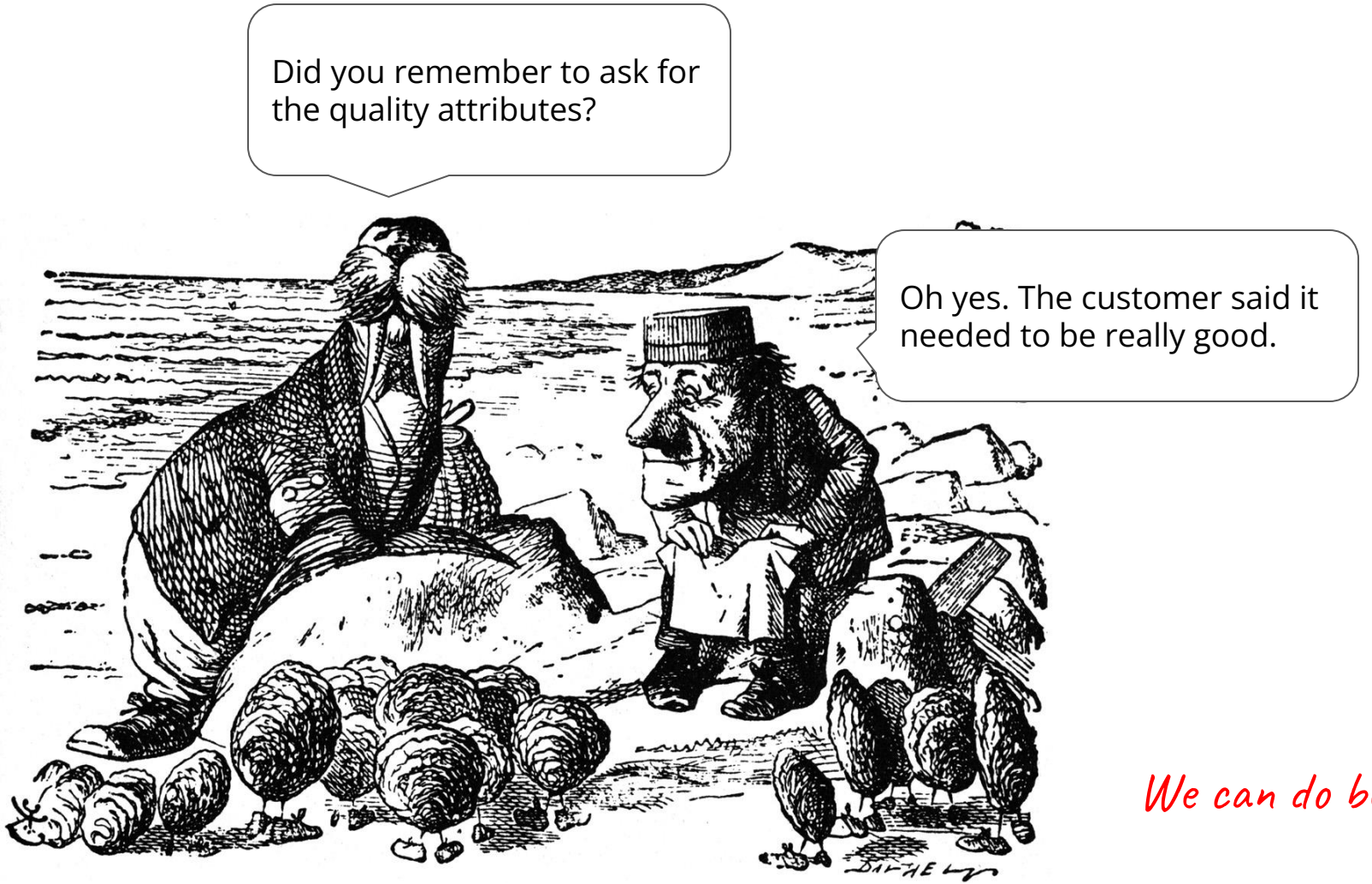
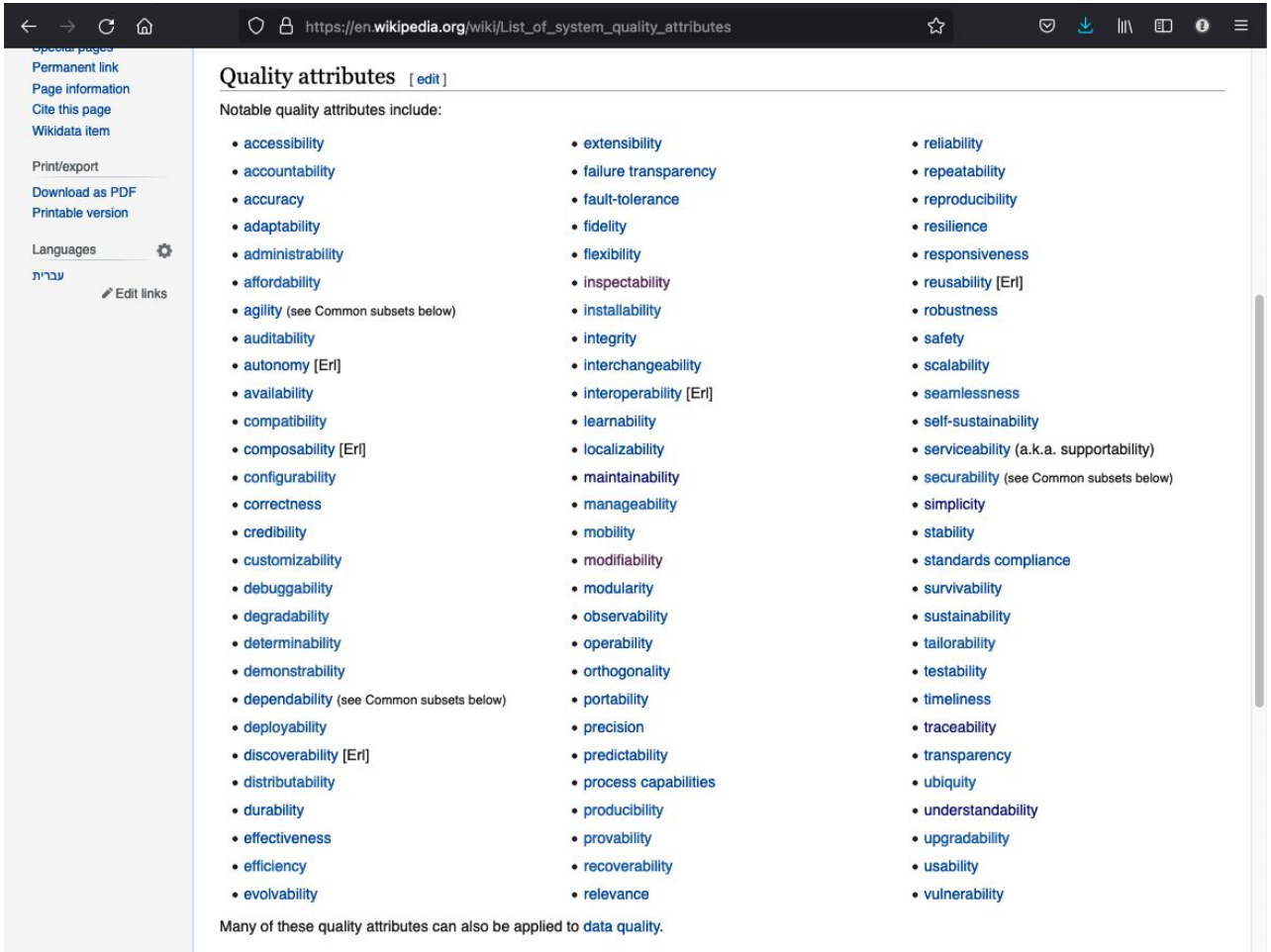


Photo by Alexandre Debiève on Unsplash



We can do better!



https://en.wikipedia.org/wiki/List_of_system_quality_attributes

Functionality ← For some reason, incl. "Security"

Usability

Reliability (incl. Availability, Resiliency, Stability)

Performance (incl. Speed, Efficiency)

~~**S**upportability (aka. Maintainability)~~ Security

+ (and more!) Maintainability, (and more!)

https://web.archive.org/web/20201112020231/http://www.ibm.com/developerworks/rational/library/4706.html#N100A7

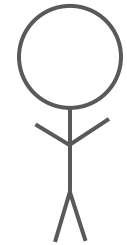
Keep in mind!

- Quality attributes are at odds with each other.
- Lists are not complete!
- Try to ***disprove*** your understanding of the problem (hypothesis testing).

Let’s explore some quality attributes!

Usability

👩 Actors



End User

💭 Goals

Learn how to use the system.

Use the system efficiently.

Minimize user errors.

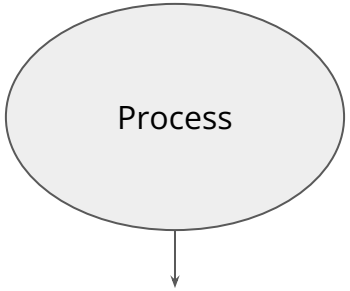
Adapt to user environment.

📈 Measures

- Time to complete a task (for newcomers/experts)
- Number of user errors
- Variety of user environments
- User satisfaction

Photo by Marissa Lewis on Unsplash

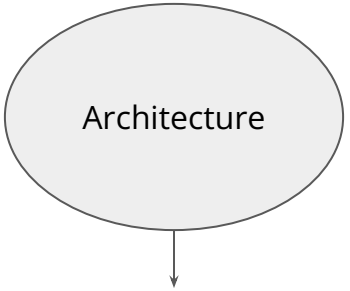
Usability Tactics



Involve UX-pert in requirements and code review

Involve actual users

Use UI component framework [1] and design system [2].

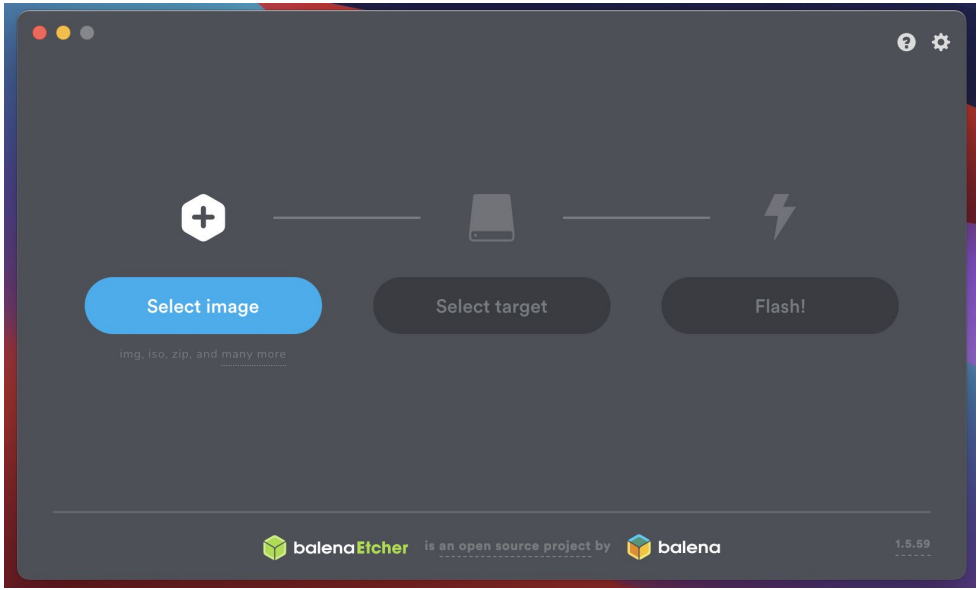


Architect for most-constrained environment first

Prioritize user-error prevention, recovery, and reporting (e.g. cancel/undo actions, helpful error messages).

[1]: https://tailwindui.com/
[2]: https://design.gitlab.com/

Usability Case - balenaEtcher



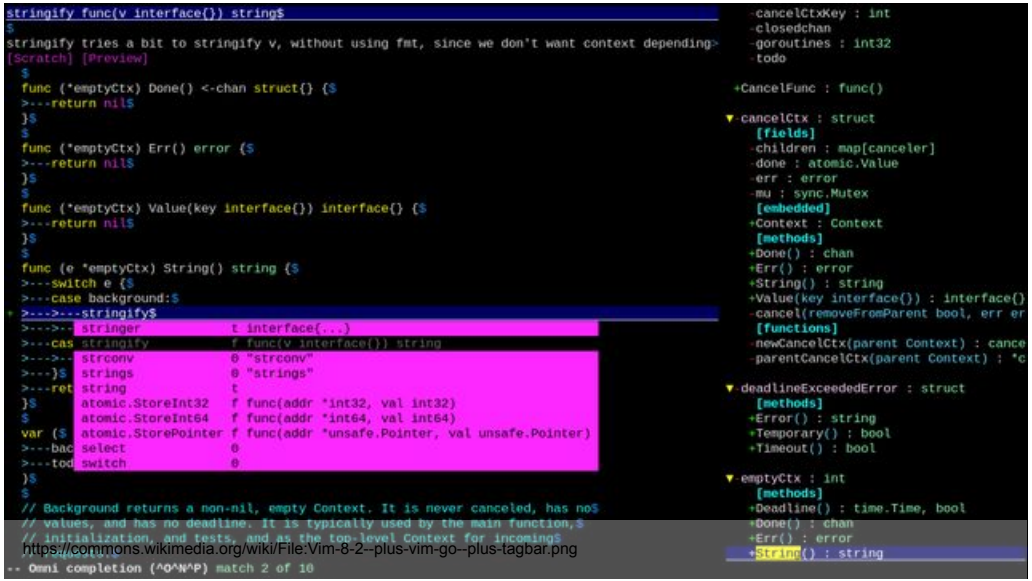
https://www.balena.io/etcher/

Usability Case - Boeing 737

Photo by Dan Lohmar on Unsplash

https://theaircurrent.com/aviation-safety/checklists-come-into-focus-as-pace-setter-for-737-max-return/

Usability Cases - vim

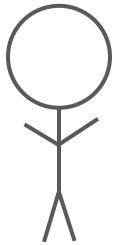


https://commons.wikimedia.org/wiki/File:Vim-8-2--plus-vim-go--plus-tagbar.png

https://www.vim.org/

Reliability

👩 Actors



Internal/External
“Client”

💭 Goals

Respond normally.

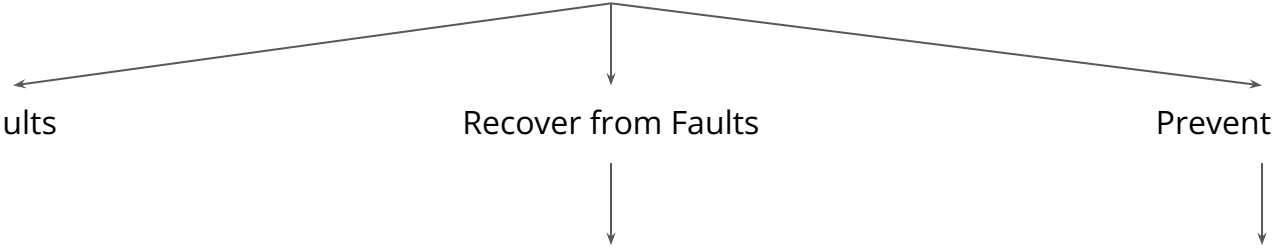
If fault, respond with information.

If fault, recover quickly.

📈 Measures

- Uptime
- Time to detect fault
- Time to recover from fault

Reliability Tactics



Monitoring

Ping/Echo

Self-test

Active redundancy

Passive redundancy

Retry

Rollback

Tracing

Transactions

End-to-end shadow testing [1]

Exceptional unit testing

[1]: https://docs.gitlab.com/14.4/ee/development/testing_guide/end_to_end/index.html

Reliability Case - Netflix

[1]: https://netflixtechblog.com/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a
[2]: https://netflix.github.io/chaosmonkey/

Reliability Case - HealthCare.gov

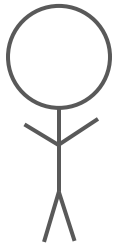
https://www.computerworld.com/article/2485995/6-software-development-lessons-from-healthcare-gov-s-failed-launch.html

Thoughts on Performance

- Performance is the “enemy” of all other quality attributes.
- Time-complexity and space-complexity are at odds with each-other.
- There is no perfect system, only “better fit” ones.

Security

👩 Actors



Attacker

💭 Goals

- Unauthorized access to secrets.
- Unauthorized data modification.
- Unauthorized change to system's behavior.
- Reduce availability.

📈 Measures

- Time passed before attack detected.
- Rate of attacks prevented.
- Security health of dependencies.

Photo by AltumCode on Unsplash

Security Tactics

Resist Attacks

- Identify + authenticate actors
- Limit process access
- Keep dependencies updated
- Use battle-tested frameworks

React/Recover from Attacks

- Revoke access
- Maintain audit trail
- Restore data
- Reset compromised secrets

Security Case - Tay

https://en.wikipedia.org/wiki/Tay_(bot)

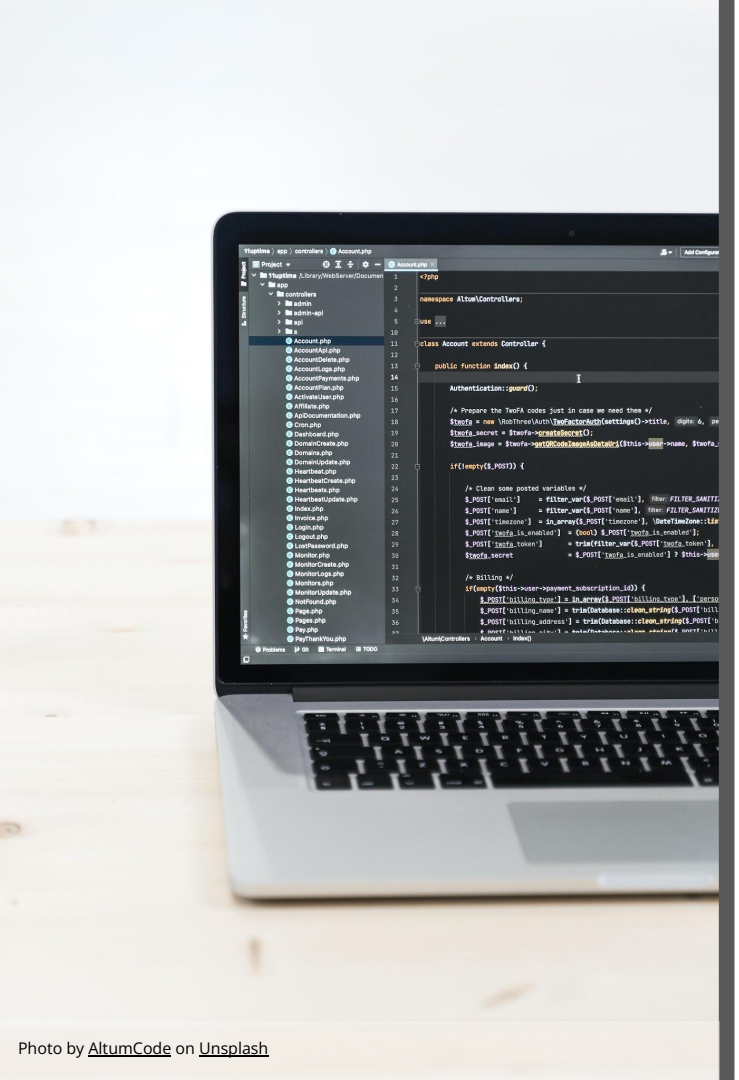
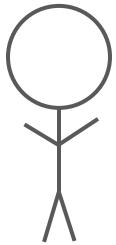


Photo by AltumCode on Unsplash

How could ***usability*** tradeoff with ***security***?

Maintainability

👩 Actors



Developer/
Contributor

💭 Goals

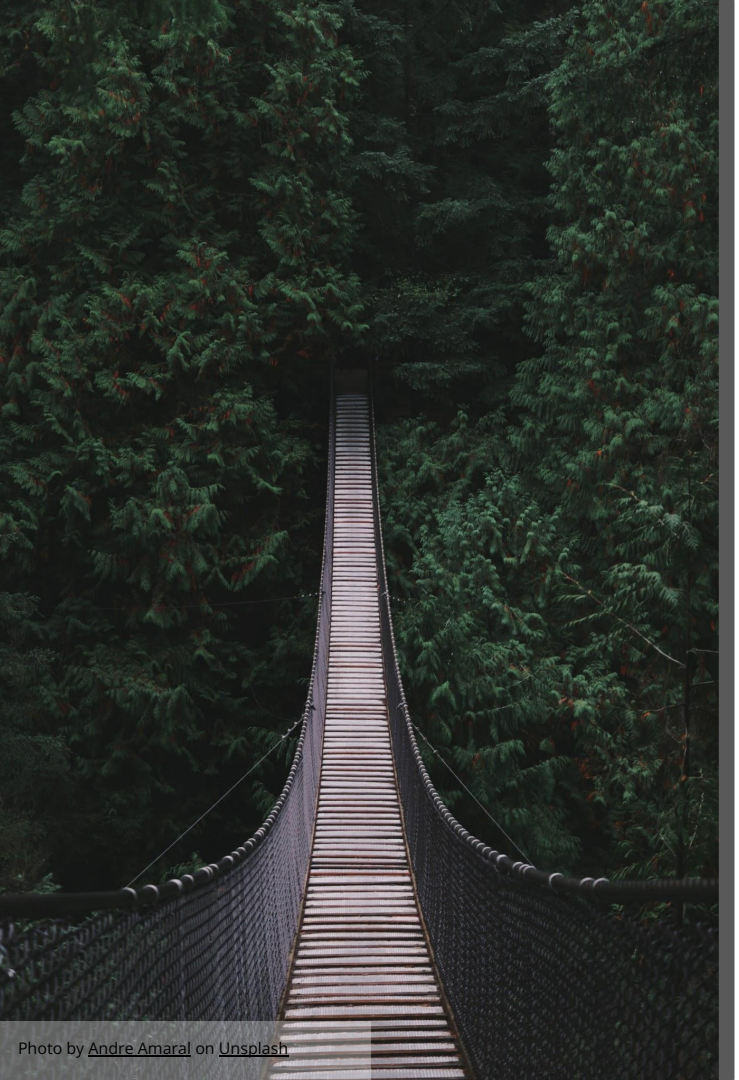
- Fix defect
- Add new behavior
- Change behavior
- Refactor

📈 Measures

- Time spent
- Velocity (number of changes in a time period)
- Regressions introduced per change
- Time for new contributor to be productive

Increase Cohesion/Reduce Coupling

1. Anticipate changes in requirements. Single-responsibility principle (SRP) implies that a module should have only 1 reason to change.
2. Hide as much information as possible between each module.
3. Prefer constraints over convenience.
4. Lint for formatting inconsistencies. This should free up code review discussions.



Prioritize testability

1. Testability implies reduced coupling.
2. Practice Test-Driven-Development when appropriate.
3. If something is hard to test, it is hard to maintain.

Photo by Barney Yau on Unsplash

Let’s talk about ***beyond functional teams***.

“Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure”

- Melvin Conway

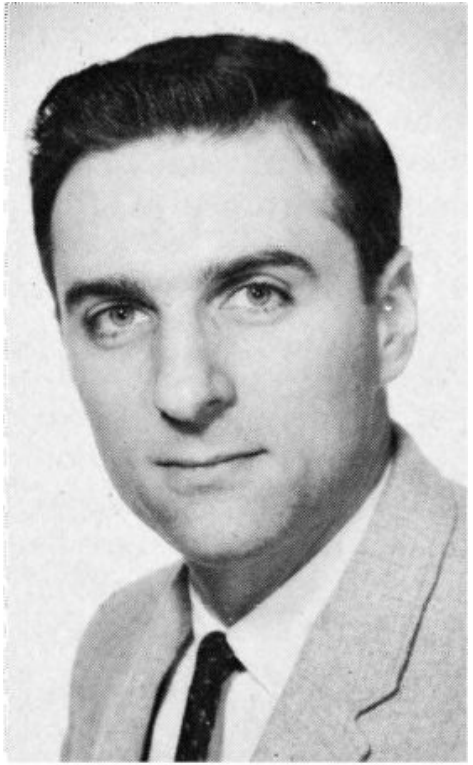
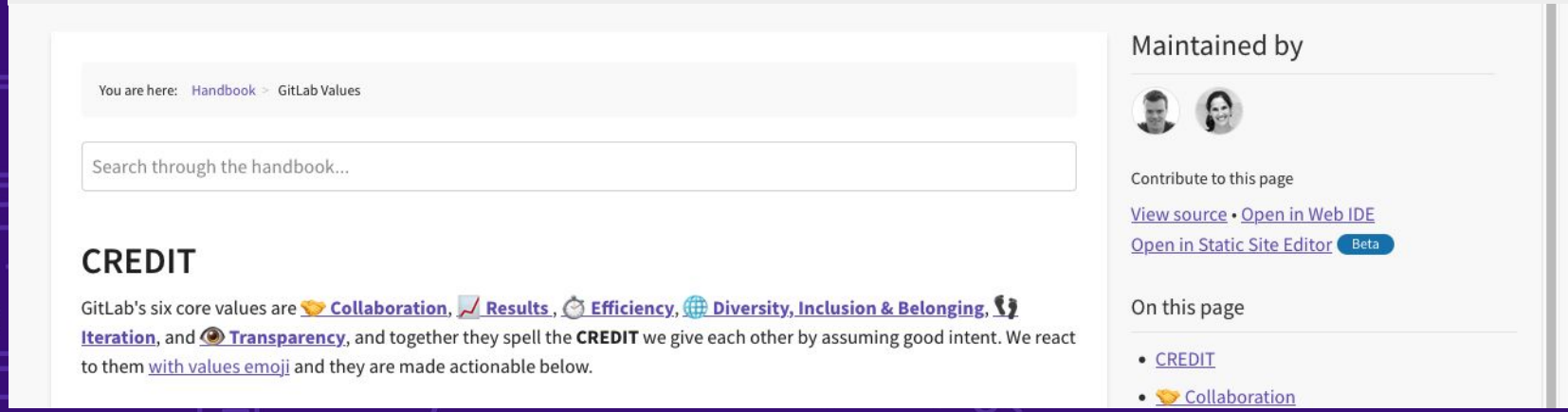


Photo by Lagos Techie on Unsplash

GitLab

Everyone can contribute.

https://about.gitlab.com/handbook/values



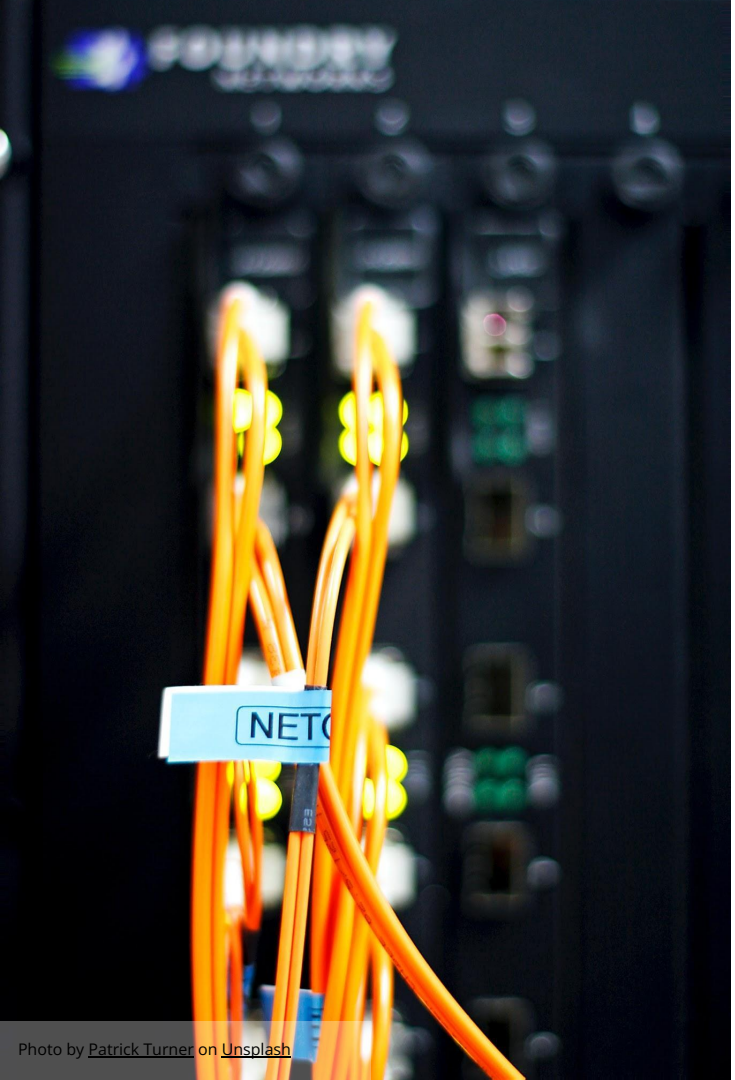


Photo by Patrick Turner on Unsplash

Collaboration Tactics

Culture

"Everyone can contribute"

Short toes

No ego + eliminating internal competition

Practice

Pair programming (sync)

Code Review (async)

Iteration Tactics

“Everything is in draft”

Minimal-Viable-Changes

Make Two-Way Door Decisions

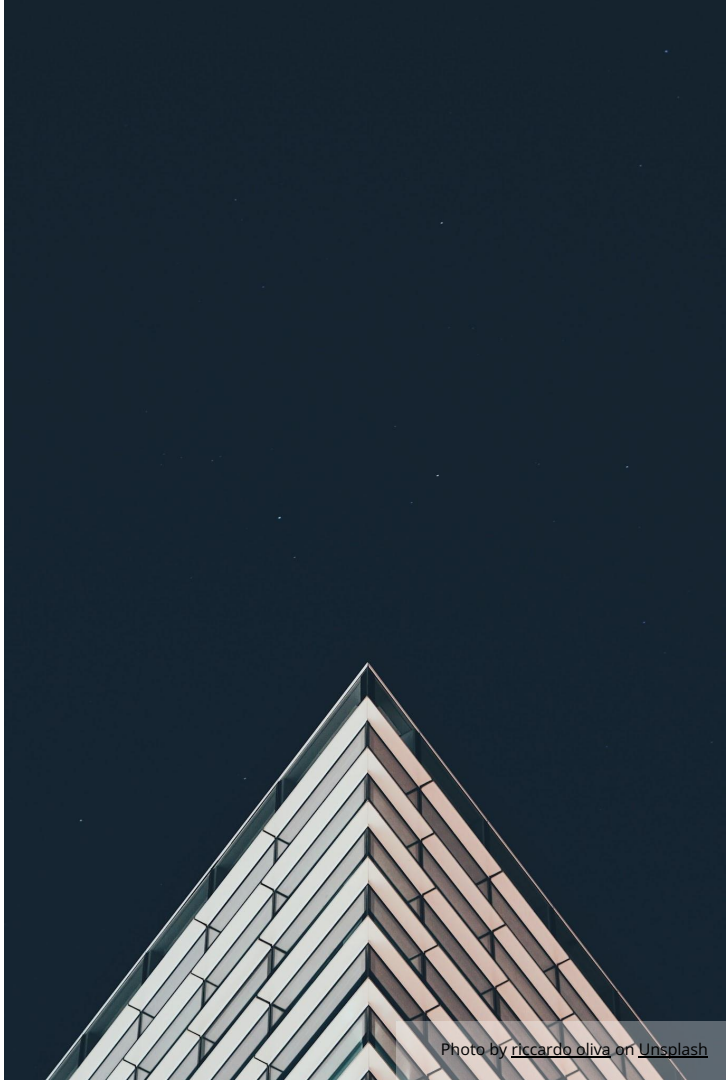
Changing proposals isn’t iteration

Transparency Tactics

Everything is public by default

Information is broadcast across relevant channels

Always say “Why” not just “What”



Final Thoughts

There is no perfect system.

You must invest in problem analysis.

Realize quality attributes by formalizing and prioritizing them.

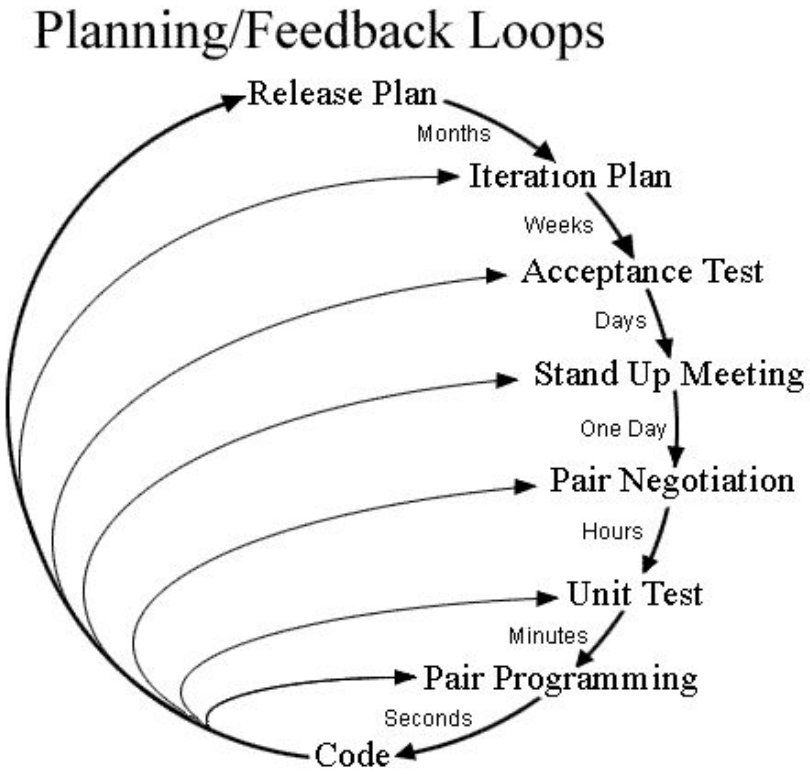
SEE YOU SPACE COWBOY...

What about agile?

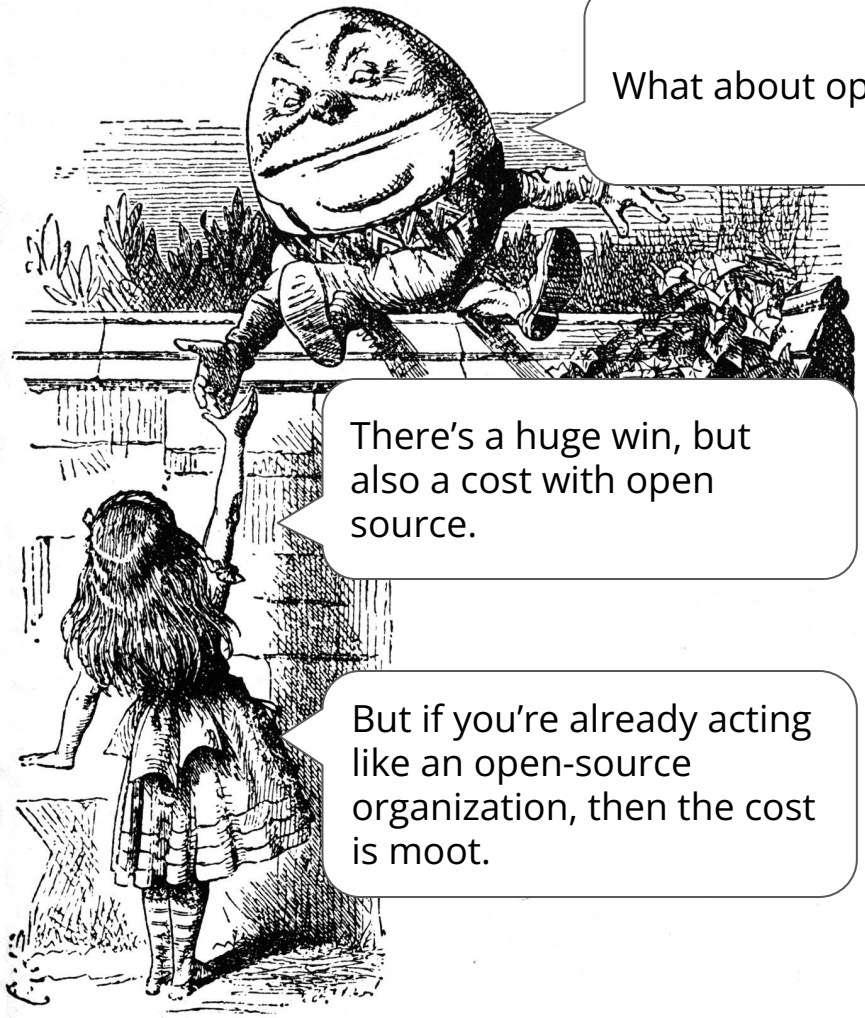
Do we know enough about the requirements to make an architectural decision?

If not, how can we reduce risk of unknown Architecturally Significant Requirements?

Small feedback loops, of course!



http://www.extremeprogramming.org/index.html



Open Source Wins	Open Source Costs
Direct involvement with consumers.	Investment in newcomer friendliness.
Large workforce of contributors.	Investment in code review maintainers and quality automation.
Good report with wider developer community.	All bugs are shallow.
	Risk of "Fork & Profit" (depends on license)