

Daniel Terhorst-North presents...

“The most dangerous phrase”

...featuring SOLID, Scrum, and other antiquities

Disclaimer



This is not a **technical talk**,
this is a **thinking talk**
with some **technology** in it!

Why do we do what we do?

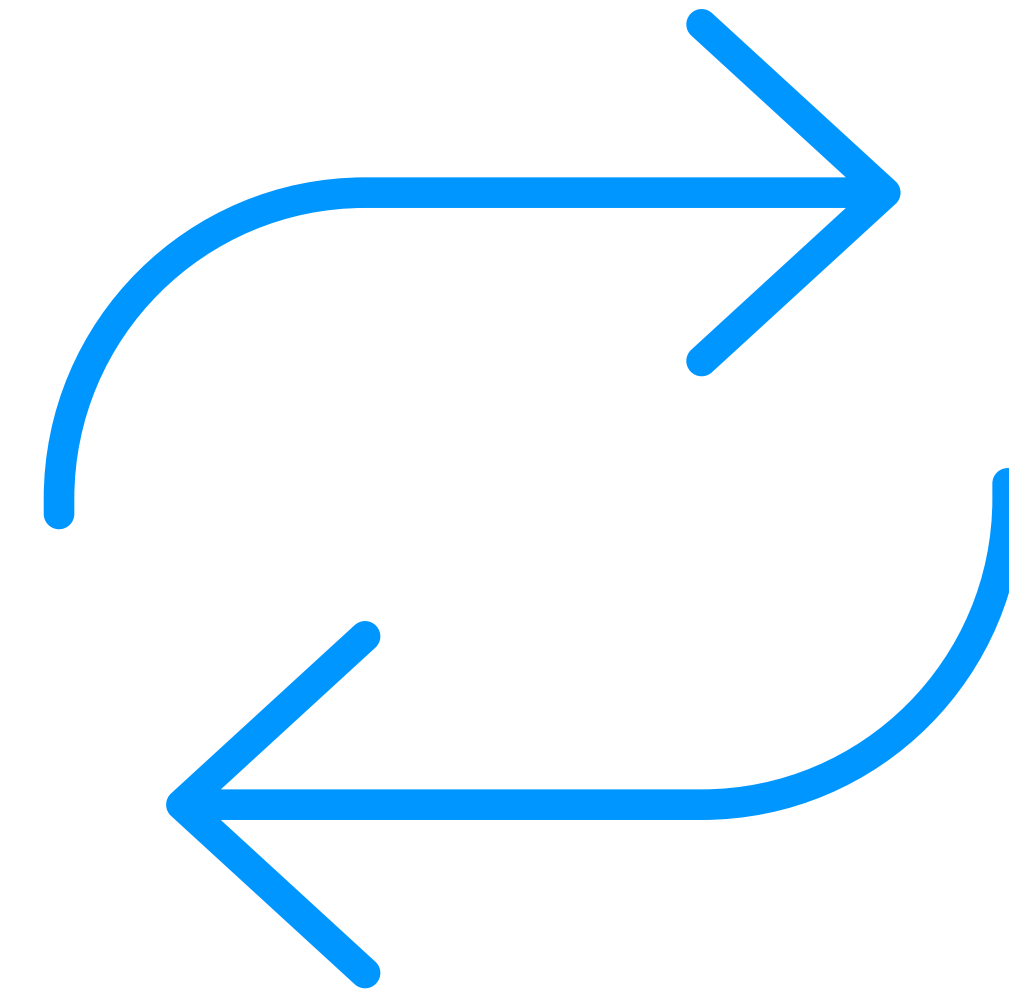
Out of **habit**

Because it is “**obviously right**”

Because this is **the way I was taught**

Because **everyone else** does it this way

Because **we’ve always done it** this way

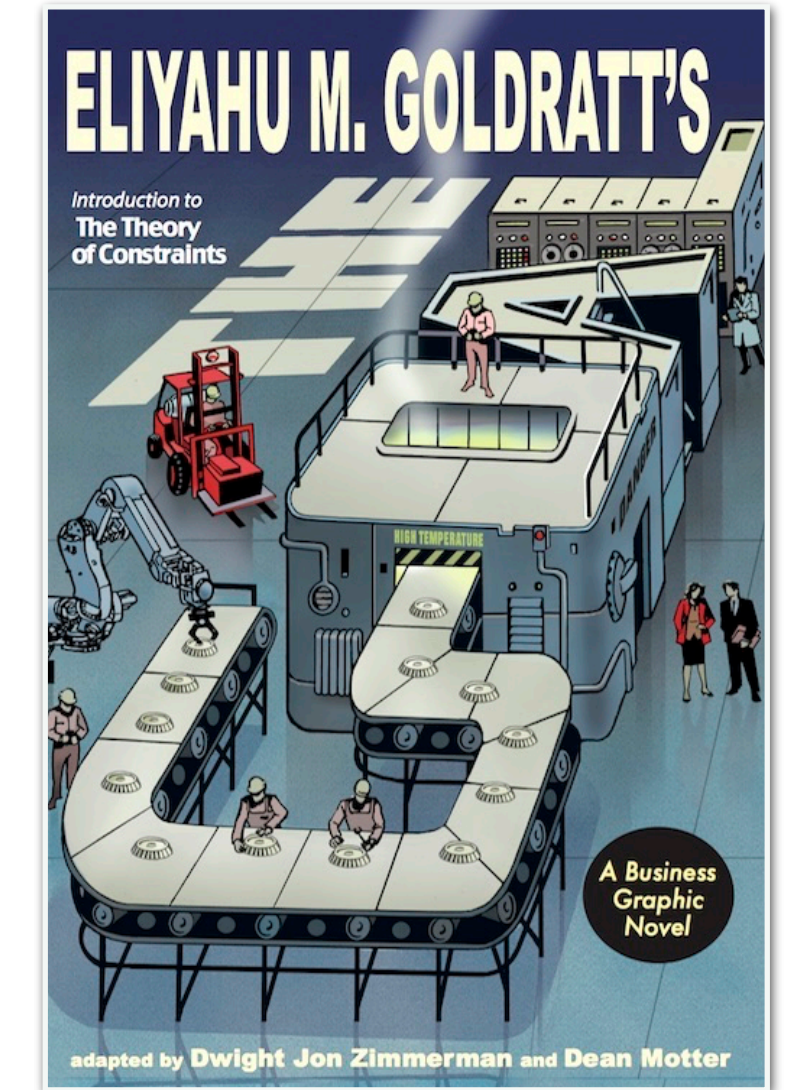
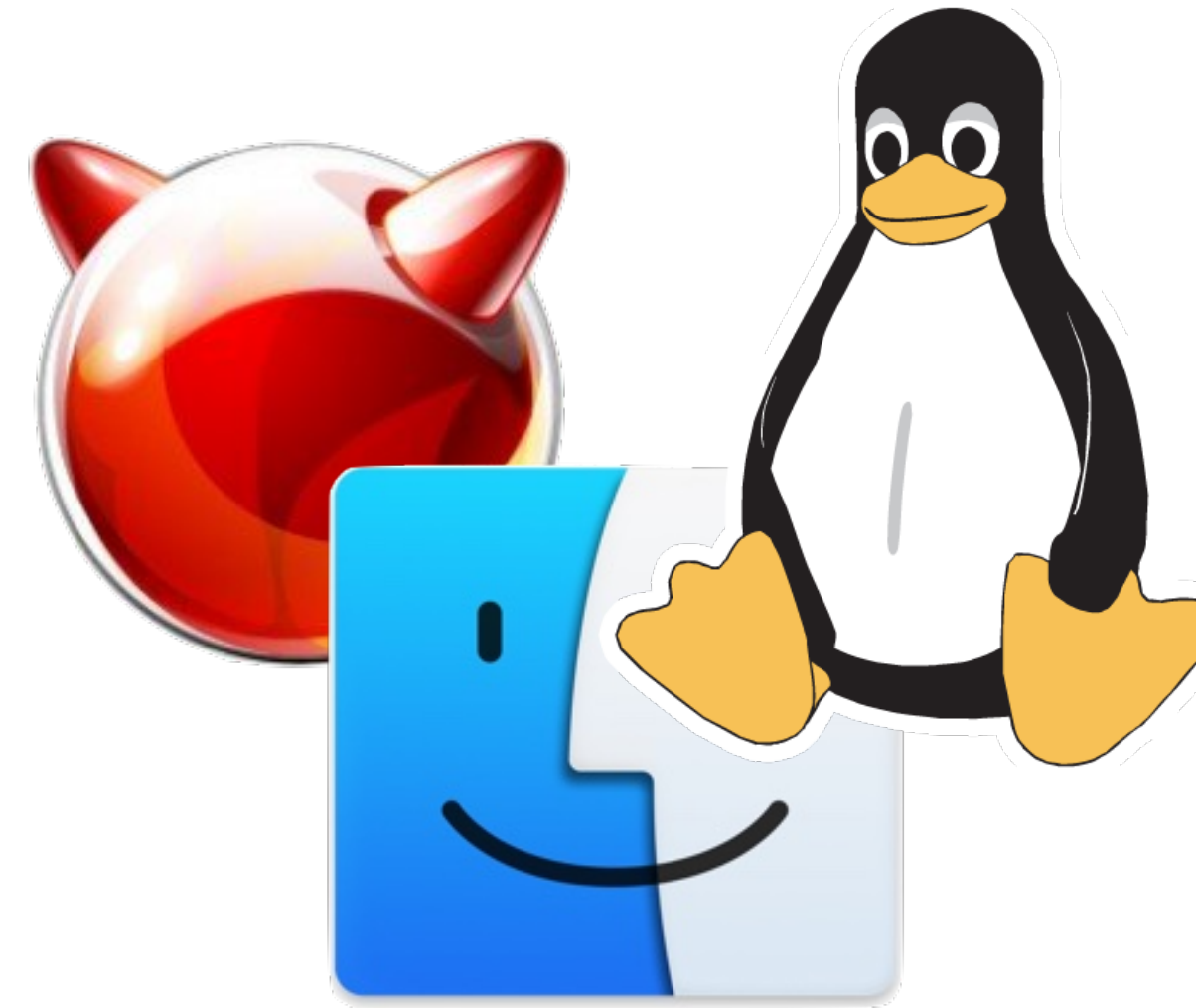


**“The most dangerous
phrase you can use
is ‘We’ve always
done it that way.’”**

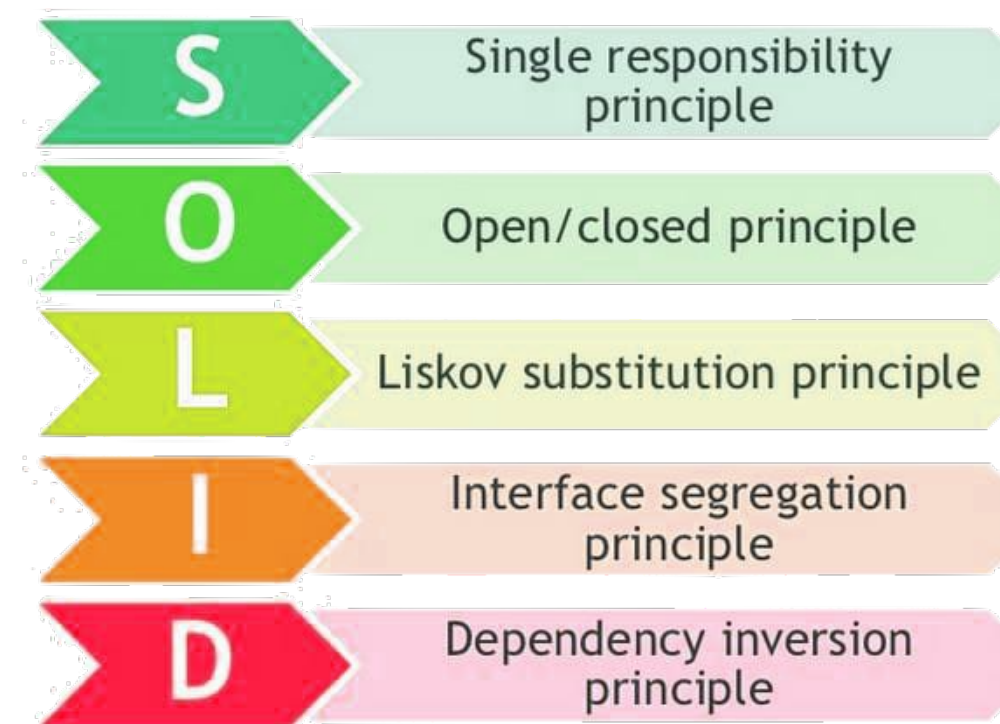
— Admiral Grace Hopper



Some ideas **age well**



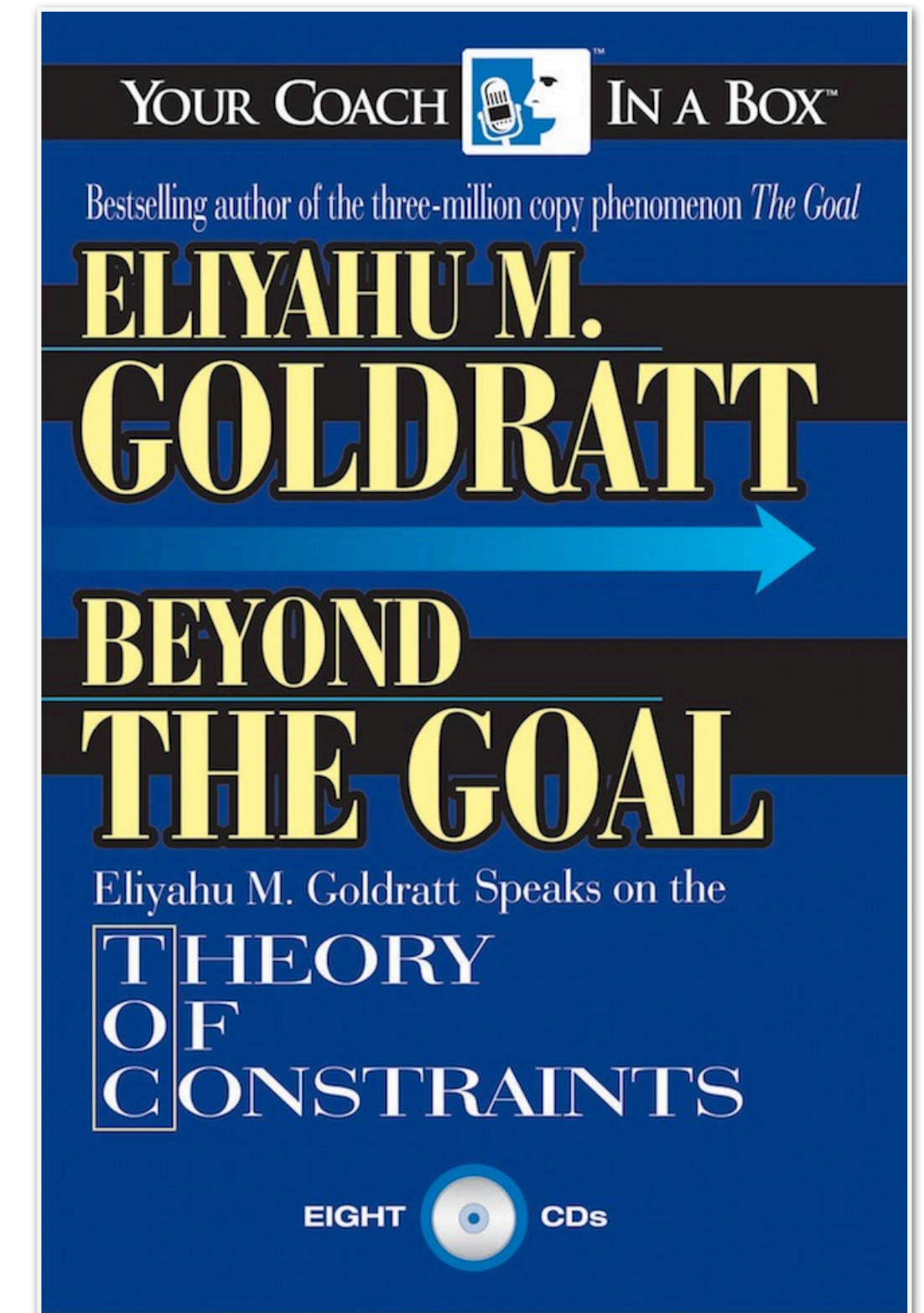
Others not so much



What does “ageing well” mean?

*“Technology can bring **benefits** if, and only if, it diminishes a **limitation**.”*

1. What is the **power** of the technology?
2. Which **limitation** does the technology **diminish**?
3. Which **rules** enabled us to **manage** this limitation?
4. Which **new rules** will we need?



What does “ageing well” mean?

*“Technology can bring benefits if, and only if,
it diminishes a limitation.”*

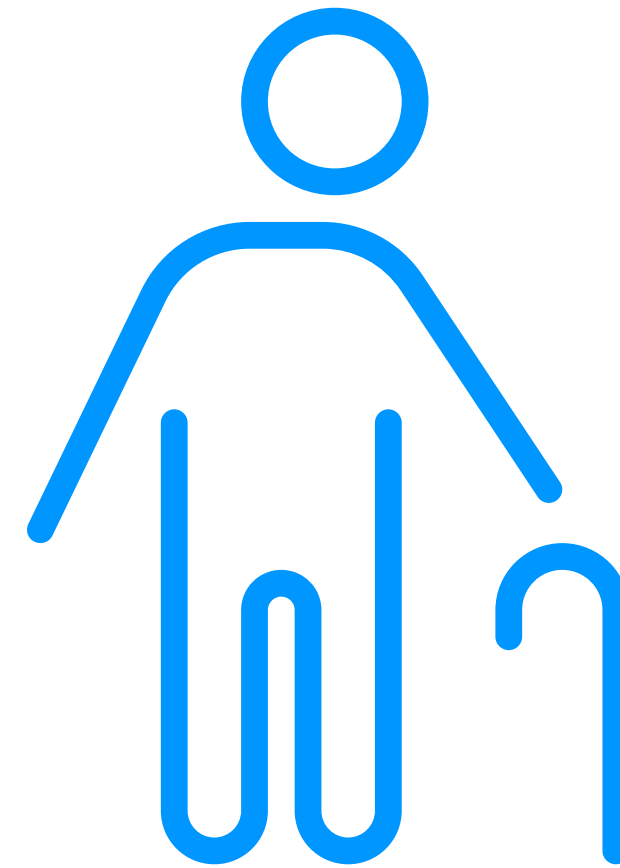
1. What is the power of the technology?
2. Which limitation does the technology diminish?
3. How has that limitation **changed** since?
4. What should we be doing **instead**?

Which **limitation** does the technology **diminish**?

It helps to understand the **context**

Talk to old people (who were there)

I am old people, **talk to me!**



Not **gatekeeping**, just getting **closer to the source**

Remember, different old people have **different stories**

Don't believe any **single source**, not even **me!**

When **good** advice turns **bad**

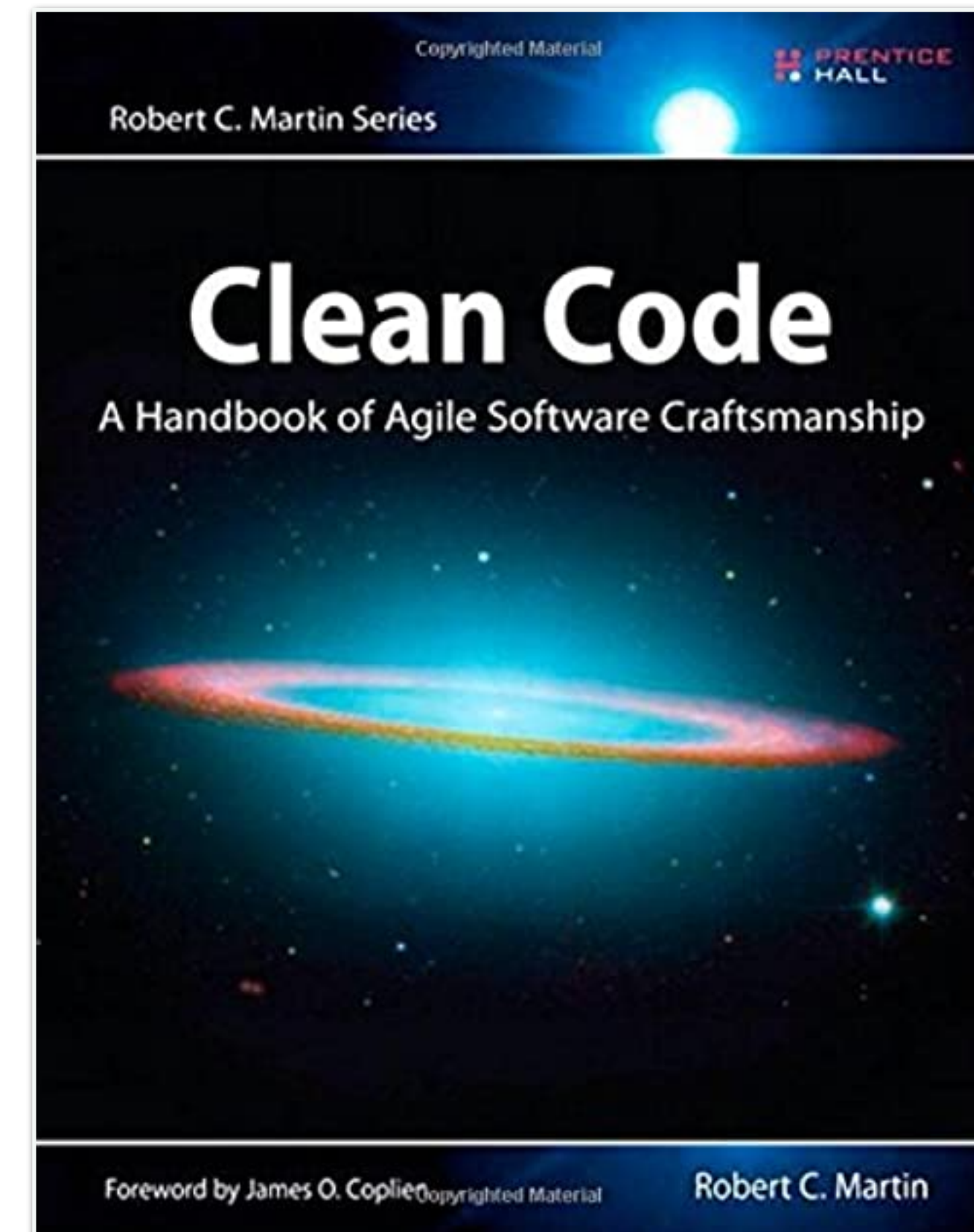
Single Responsibility Principle

Open-Closed Principle

Liskov Substitutability Principle

Interface Segregation Principle

Dependency Inversion Principle



Single Responsibility Principle

“A class should have only **one reason to change** [or **serve one actor**]”

“Gather together the things that **change for the same reasons**. Separate those things that **change for different reasons**.”

“[T]he role of an **accountant** is different from a **database administrator**.”

“This is the reason we do not put **SQL** in **JSPs**. This is the reason we do not **generate HTML** in the modules that **compute results**. This is the reason that **business rules** should not know the **database schema**. This is the reason we **separate concerns**.”

Single Responsibility Principle

```
 public void printReport() {  
    data = getData();  
    calc = doCalc(data);  
    render(calc);  
}
```

Single Responsibility Principle

```
 public void printReport() {  
    data = getData();  
    calc = doCalc(data);  
    render(calc);  
}
```

```
 public Data getData() {  
    ...  
}
```



```
 public Calc doCalc(Data data) {  
    ...  
}
```



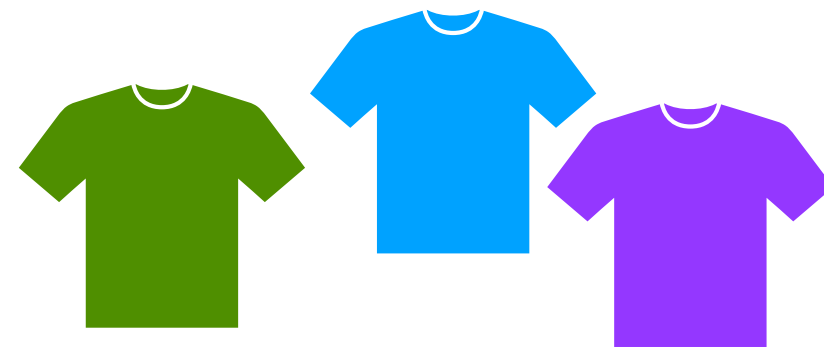
```
 public void render(Calc calc) {  
    ...  
}
```



Single Responsibility Principle

```
 public void printReport() {  
    data = getData();  
    calc = doCalc(data);  
    render(calc);  
}
```

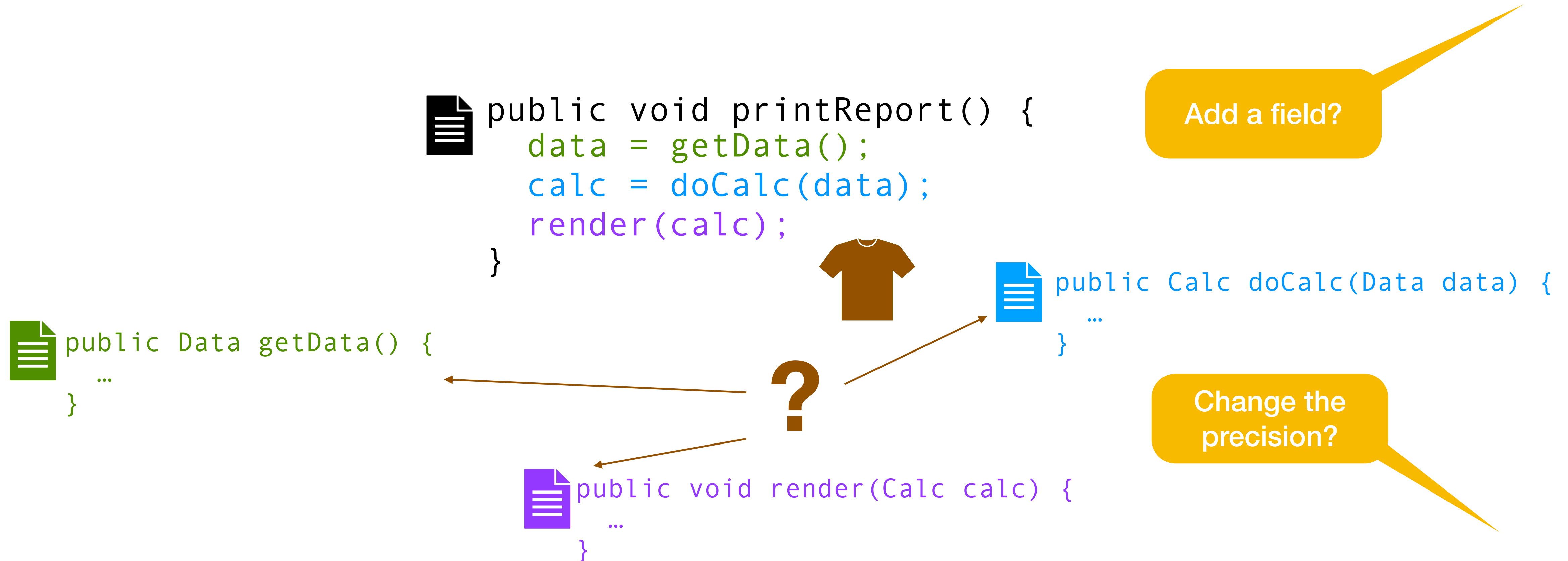
```
 public Data getData() {  
    ...  
}
```



```
 public Calc doCalc(Data data) {  
    ...  
}
```

```
 public void render(Calc calc) {  
    ...  
}
```

Single Responsibility Principle



Single Responsibility Principle

Logical conclusion:

Ruby on Rails

“Separation of Concerns”

```
app
├── assets
│   ├── config
│   ├── images
│   └── stylesheets
├── channels
│   └── application_cable
├── controllers
│   └── concerns
├── helpers
├── javascript
│   └── controllers
├── jobs
├── mailers
├── models
│   └── concerns
└── views
    └── layouts
```

[abridged]

Separation of Concerns

Holistic, e.g. correct vs efficient

*“...being **one-** and **multiple-track** minded **simultaneously**.”*

*“The task of ‘making a thing satisfying our needs’ is split into two parts ‘**stating the properties** of a thing’ and ‘**making** a thing guaranteed to have the stated properties’.*

*“Business data processing systems are **sufficiently complicated** to require such a separation of concerns.”*

— E. W. Dijkstra, *On the Role of Scientific Thought*, 1974

Let's actually separate concerns!

Behaviour is a concern; code structure is a separate concern

Keep all the code together while it Fits In Your* Head

- *where Your Head understand the language, its toolchain, its idioms, the problem domain

Only then, move the code to “gather together things that change together”!

- assume that programmers are polyglot and that changes affect the whole stack
- self-contained sub-reports rather than ReportViews, ReportModels, etc.

HTMX: a counterpoint

Motivation:

Why should only `<a>` and `<form>` be able to make HTTP requests?

Why should only `click` and `submit` events trigger them?

Why should only `GET` and `POST` methods be available?

Why should you only be able to replace the entire [page]?

By removing these arbitrary constraints, htmx completes HTML as a hypertext.

```
<button hx-post="/clicked"
        hx-trigger="click"
        hx-target="#parent-div"
        hx-swap="outerHTML"
>
    Click Me!
</button>
```

<https://htmx.org/essays>

**“The most dangerous
phrase you can use
is ‘We’ve always
done it that way.’”**

— Admiral Grace Hopper



Open-Closed Principle

“A class should be **open for extension**, **closed for modification**.”

Answers the question: “*But what happens if..?*” [we add a new case]

Two flavours

- Bertrand Meyer uses abstract classes
- Robert Martin uses interfaces

Open-Closed Principle

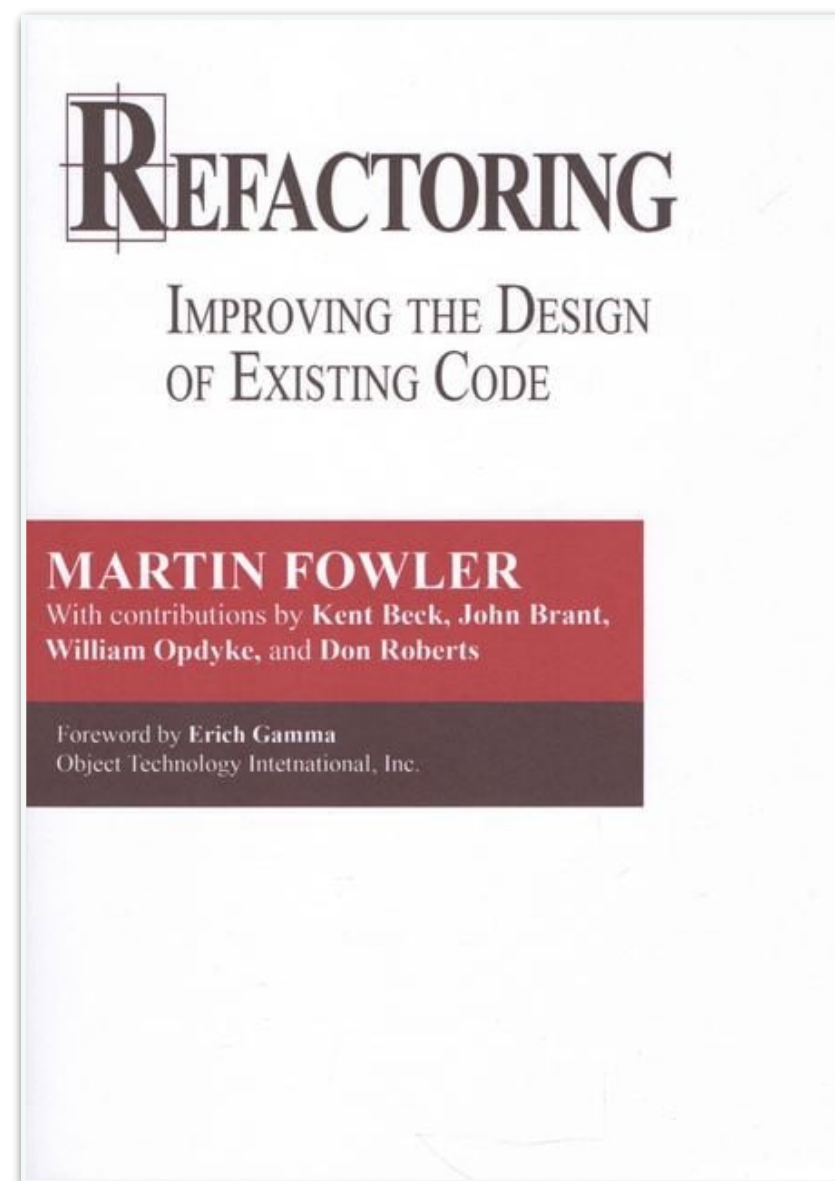
Source code was an **asset**

- You would sell a **specialised source library** for e.g. healthcare or finance
- **Clear extension points** so the customer wouldn't mess up the internals

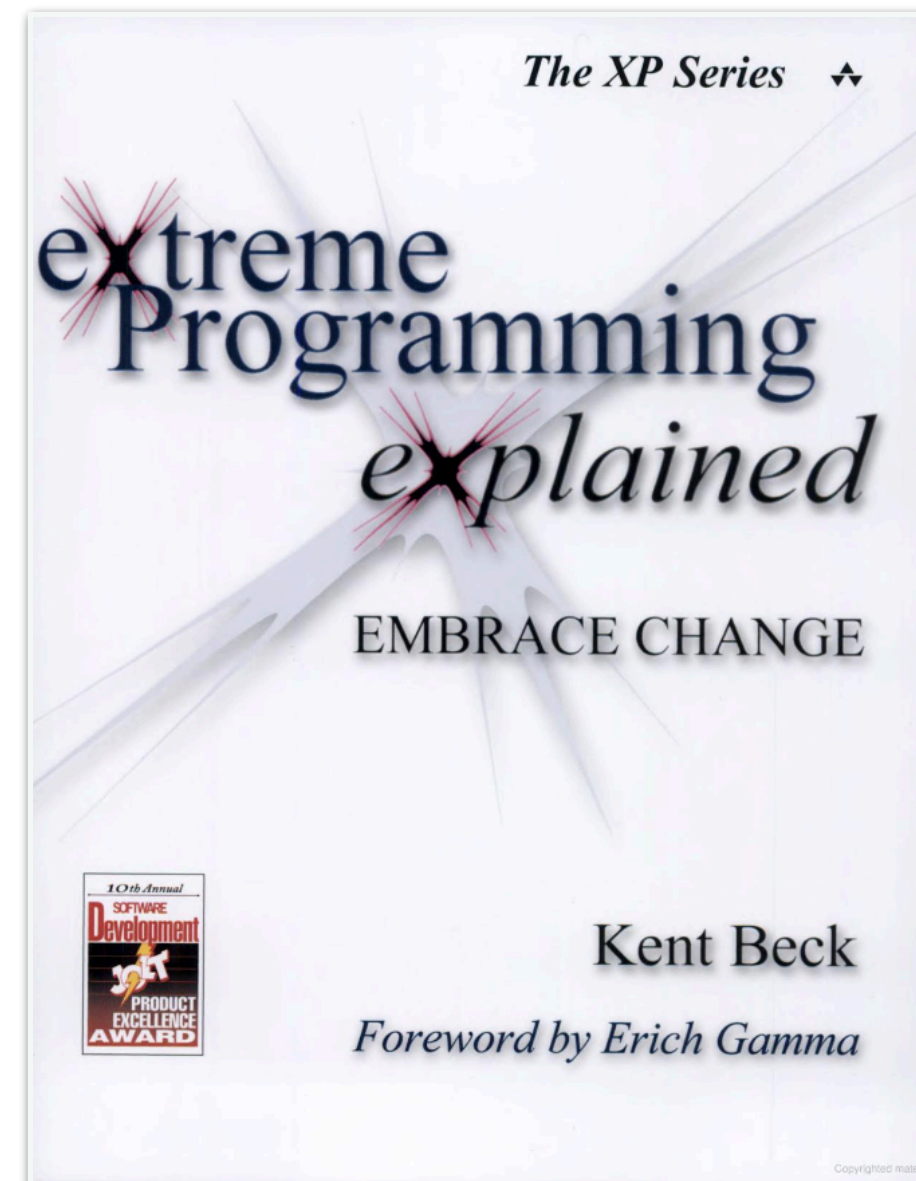
Changing code was **risky**

- **poor tooling** for deterministic code changes
- **little automated testing** in the C++ or Enterprise Java cultures

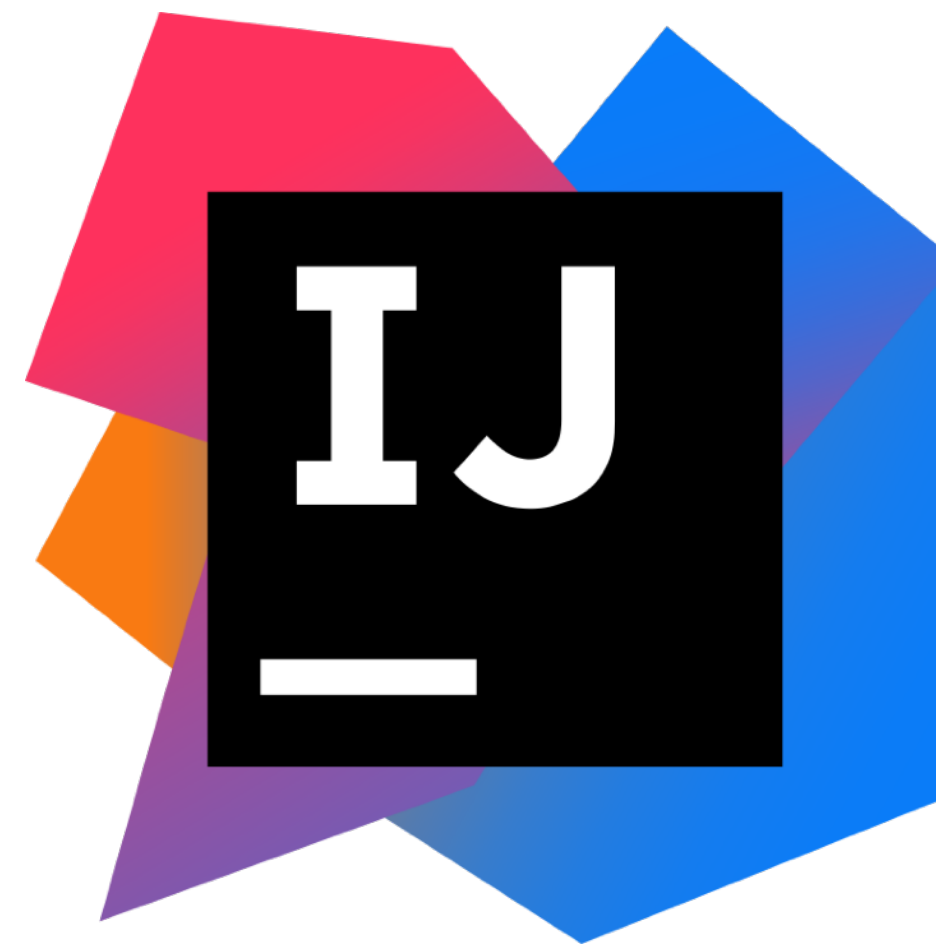
Open-Closed Principle



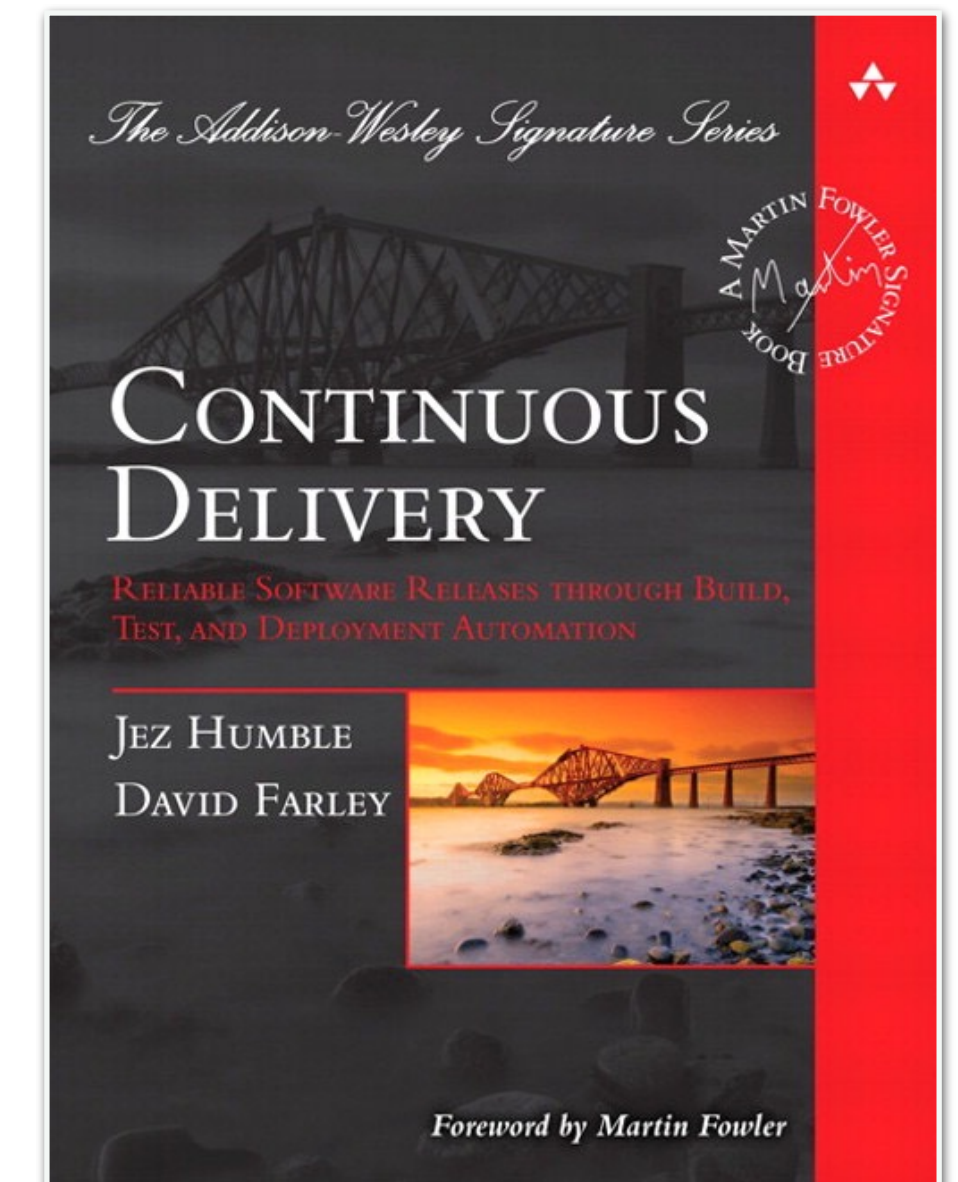
1999



2000



2001



2010

Open-Closed Principle

Changing code became **easy** and **deterministic**!

- The IDE **refactored code** under your fingers
- TDD enabled **iterative design**; test automation provided **feedback**
- deterministic build pipelines meant we could **fix forward**

Services became the asset!

- You would integrate with a **SaaS** through its **API**
- no **abstract classes** were harmed

All code was both **open** and **closed**, all the time!

- code became **malleable**, not assembled **chunks**

**“The most dangerous
phrase you can use
is ‘We’ve always
done it that way.’”**

— Admiral Grace Hopper



The others are left as an exercise*

*or read “*CUPID—the back story*”

Single Responsibility Principle

Open-Closed Principle

Liskov Substitutability Principle

Interface Segregation Principle

Dependency Inversion Principle



Scrum

Started* in the early 1990s with a **single constraint**

- **Martine Devos**: deliver *something in 12 weeks* with a checkpoint at 6 weeks (“sprint to halfway”)



fic.kr/p/9LLNfg

Novel practices emerged

- multi-year **release** became 6-week **Sprints**
- detailed **Gantt chart** became lightweight **Sprint Plan**
- multi-month **features** became vertical **Backlog items**
- **monthly steering** became daily **Scrum**
- comprehensive **Business Requirements Document** became **Backlog** of desired items
- time-consuming **hand-offs** became **Dev Team**

Scrum

No one could believe what they are seeing!

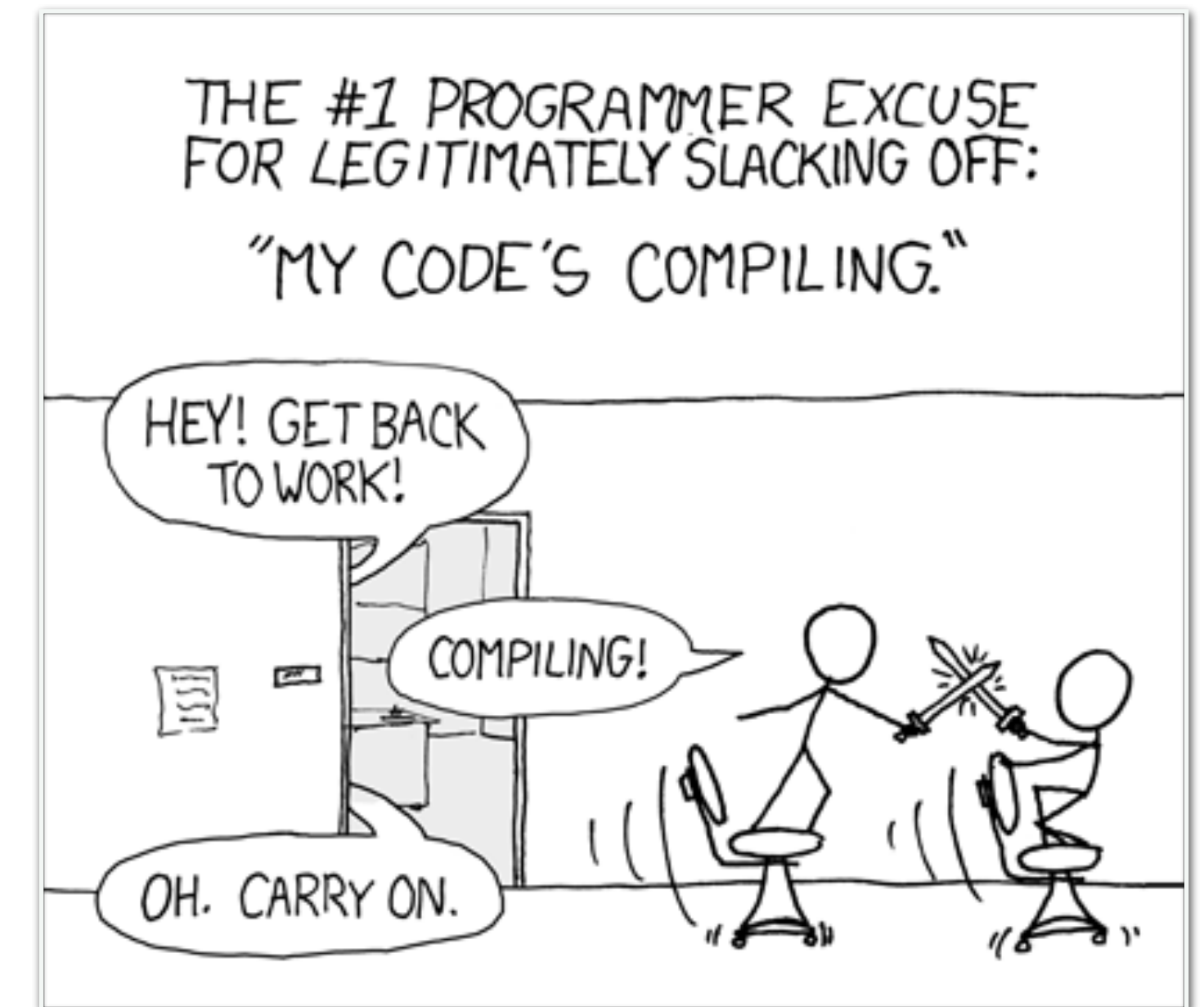
- Delivering within a quarter was “impossible”
- Hardware procurement took months, provisioning took weeks
- Compiling code took hours, deployment even longer

Two of the originators of Scrum wrote the Guide

- Certification provided a direct lineage to real practitioners

Scrum exploded into a \$billion certification industry

- eclipsed all other Agile methods, became synonymous with “Agile”



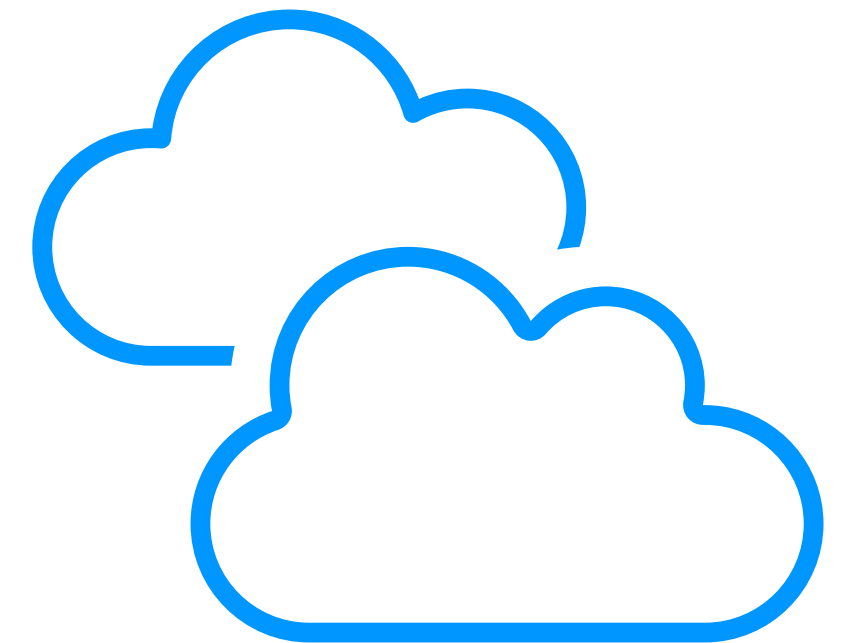
xkcd.com/303

Scrum

In the 30 years since...

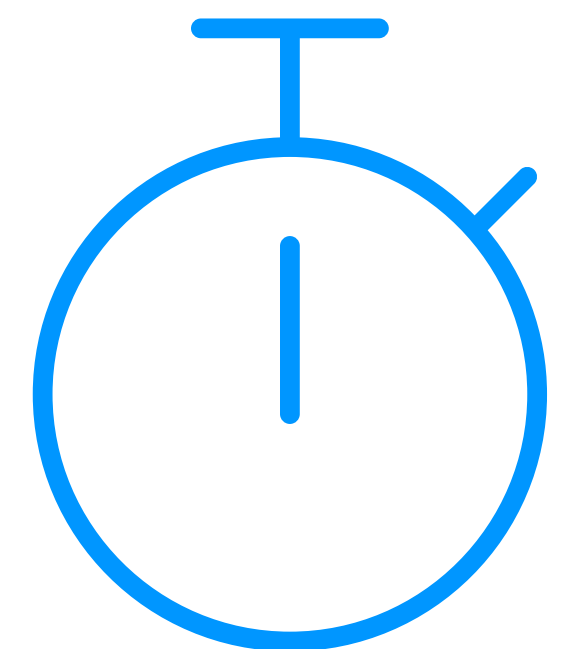
Technical landscape has evolved beyond recognition

- Languages that compile in milliseconds
- Continuous Delivery from source code check-in to tested, running system
- Procurement and provisioning of compute infrastructure on demand
- Web deployment and beta app channels enable rapid and rich user feedback



Scrum moved the game from years to months or weeks

- now it moves the game from minutes to months or weeks
- solves for problems that no longer exist (except where they do!)



A modern alternative to Scrum

Two planning cadences: quarterly OKRs and in the moment

Organise teams around anticipated demand

- As fluid as the demand is dynamic, quarterly re-teaming
- Some people not even in teams!

Deployment is a technical decision, release is a commercial one

Daily steering is about what is left, not what is done

Why do we cling to **obsolete technology**?

Commitment & Consistency Bias

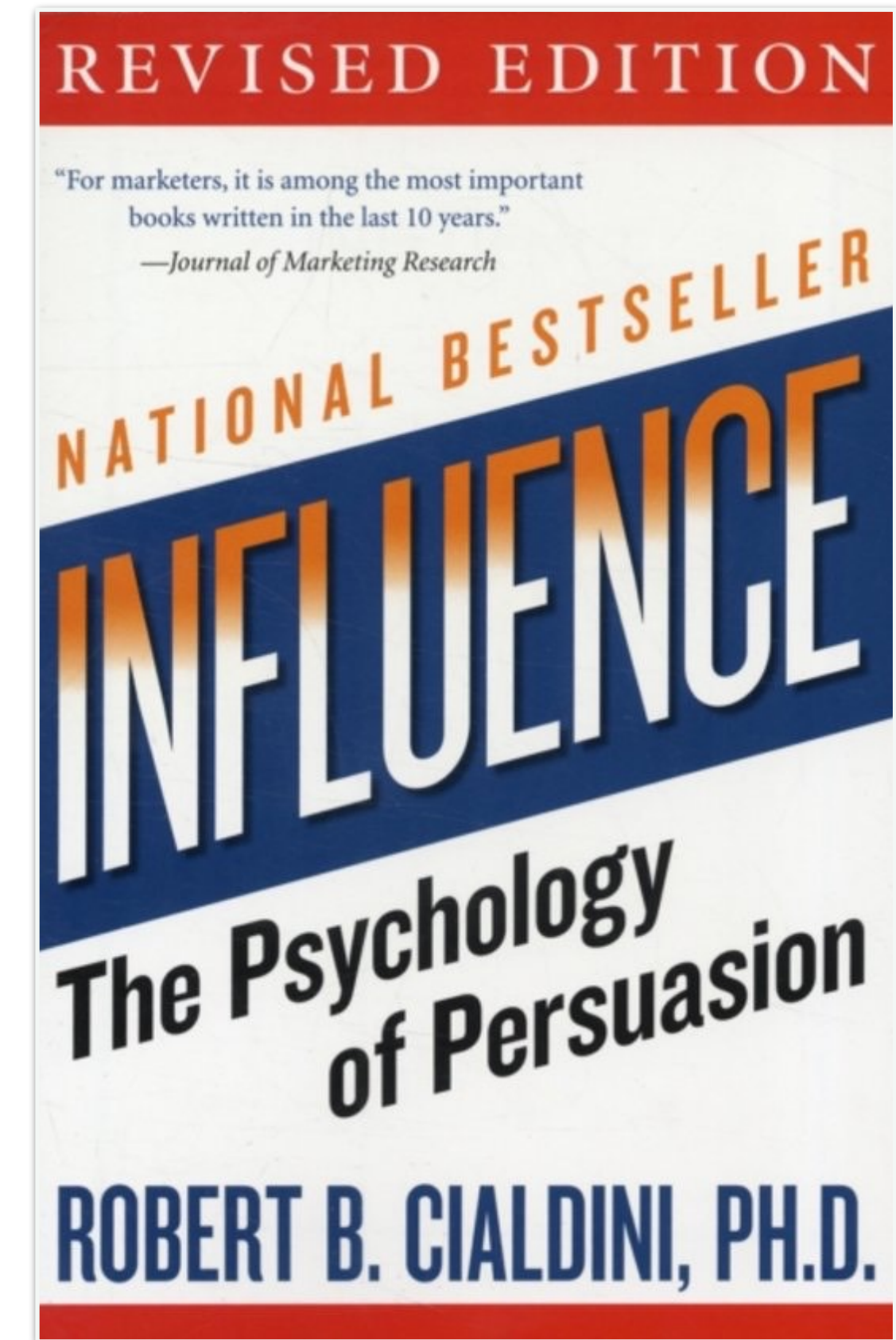
- “No one ever got fired...”

Momentum, mindshare, money accrete over time

- Do not assume malice or greed, just **structural inertia**

Bounded rather than **centred** communities

- Boundary **prevents** evolution
- **Tweaking** for “clarification”
- **Gatekeeping** by the founders



“You must think for yourselves!”

Look past the Scrum and SOLID dogma

Assess your own tools & practices

- Which should you change, and towards what?

Talk to some old people

- and remember, I am old people!

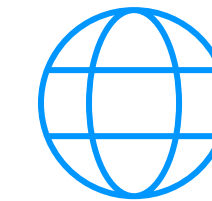
**“The most dangerous
phrase you can use
is ‘We’ve always
done it that way.’”**

— Admiral Grace Hopper



Thank you

<https://dannorth.net>



daniel@dannorth.net



[@tastapod](#)



[@tastapod@mastodon.social](#)



Come and find me!