Refactoring with Connascence

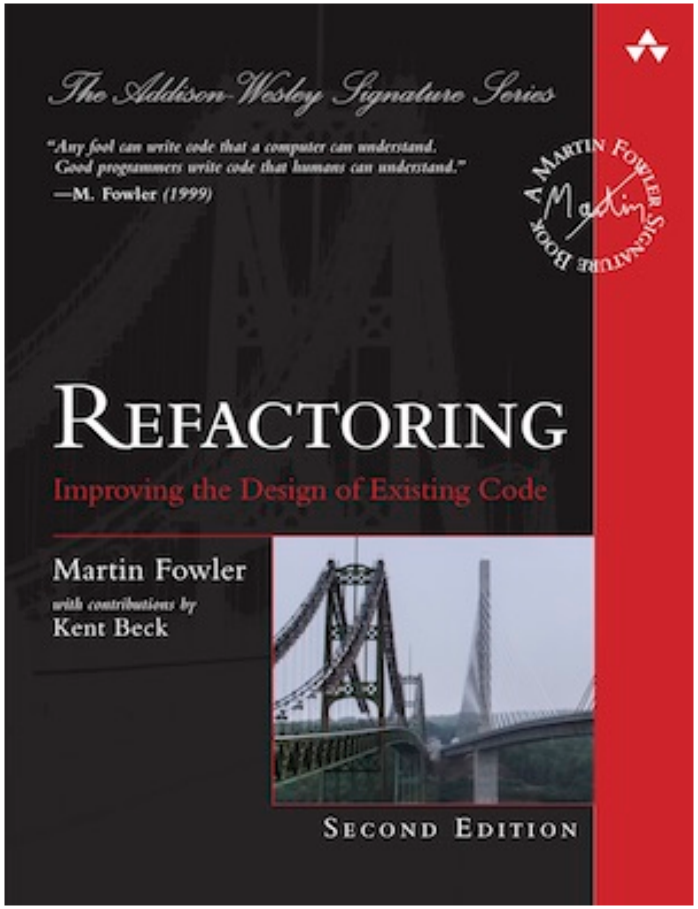
Softwerkskammer Nürnberg
06.12.2018

TDD Loop

- **Red**
  - write a failing test
- **Green**
  - make the test pass
- **Refactor**
  - improve the design
- **Repeat**

Refactoring

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."
(Martin Fowler)


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A Martin Fowler Signature Book
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Refactoring Workflows

- TDD Refactoring
- Litter-Pickup Refactoring (boy-scout rule)
- Comprehension Refactoring (better names)
- Preparatory Refactoring

  “for each desired change, make the change easy (warning: this may be hard), then make the easy change“ (Kent Beck)
- Planned Refactoring
- Long-Term Refactoring

Refactoring is a tool

- What tools are there for refactoring?

- Start in green (Test Runner)
- Small mechanical steps (Refactoring book)
- IDE with Refactoring support
- Most important tool?

The most important tool?

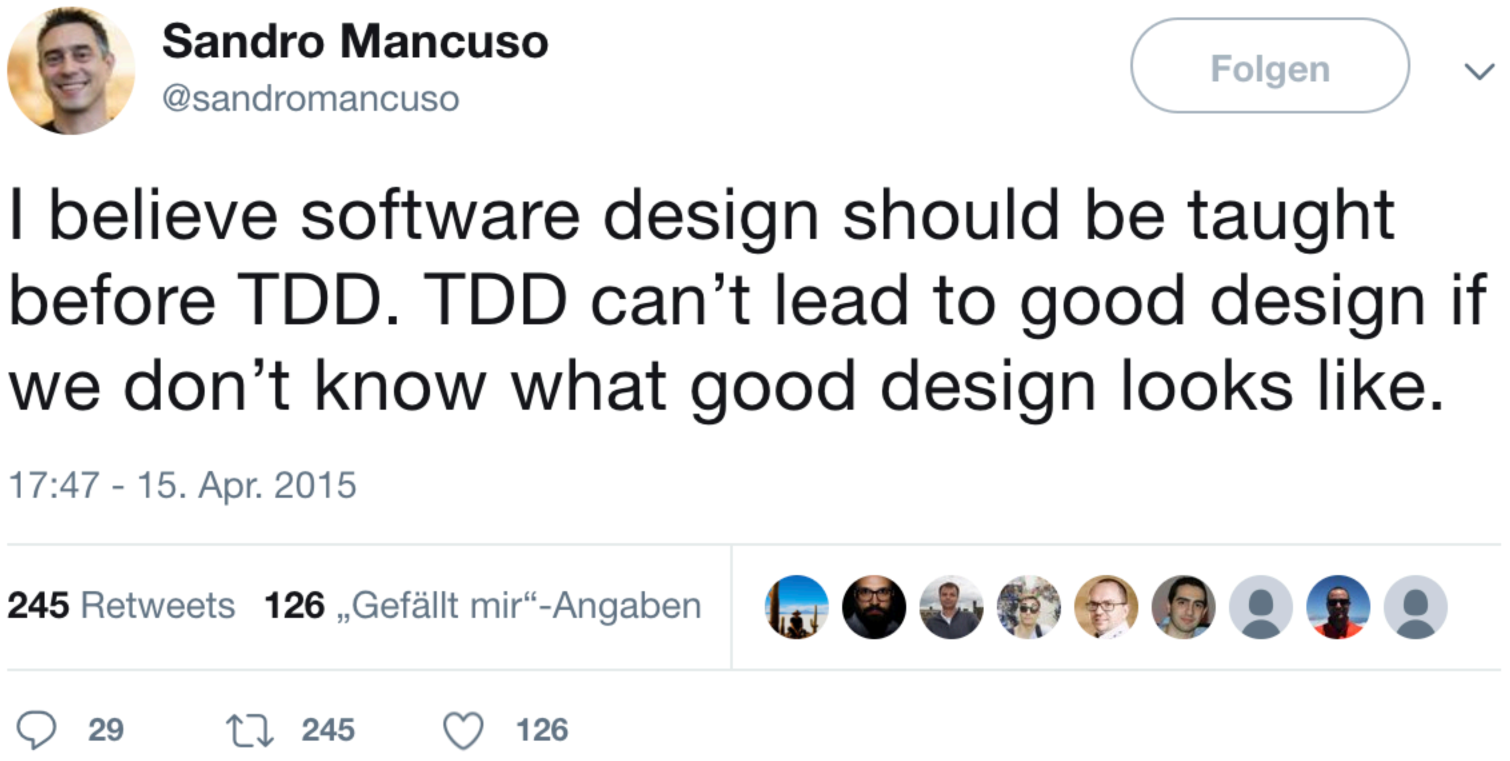
- Our brain

  !?!

- TDD cycle really is

  Think-Red-Think-Green-Think-Refactor-Think …

- What can help us to reason about our code?



Sandro Mancuso
@sandromancuso

Folgen

I believe software design should be taught before TDD. TDD can't lead to good design if we don't know what good design looks like.

17:47 - 15. Apr. 2015

245 Retweets 126 „Gefällt mir"-Angaben

29 245 126

https://codurance.com/2015/05/12/does-tdd-lead-to-good-design/

Better Design?

- There are **principles**
- E.g. SOLID
  - Single Responsibility
  - Open/Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion

Better Design?

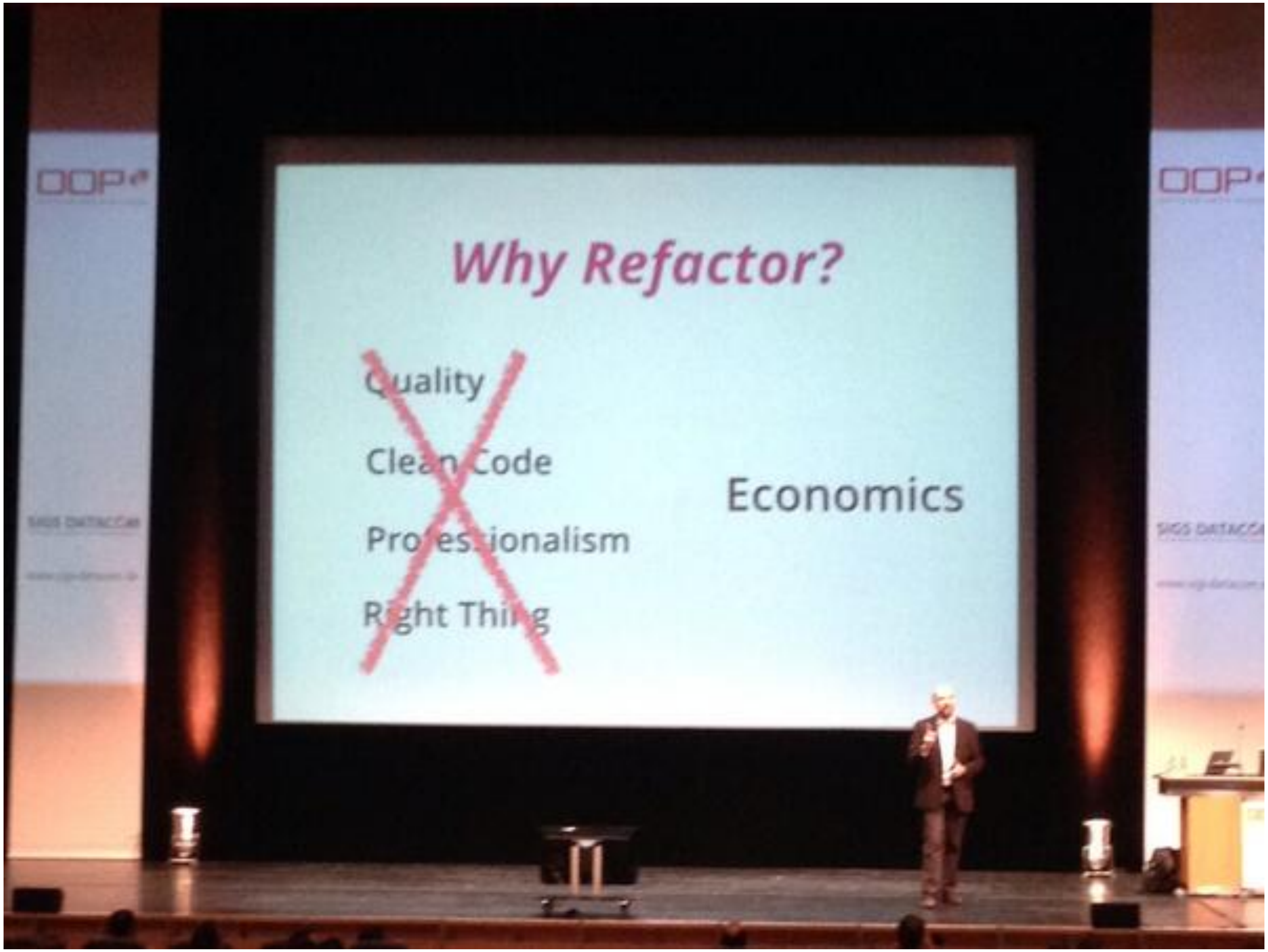
- And there are **rules**
- E.g. the 4 elements of simple design (Kent Beck, here J.B. Rainsberger's version)
  - Passes its tests
  - Minimizes duplication
  - Maximizes clarity
  - Has fewer elements

Better Design?

- And there are even **laws**
- E.g. the Law of Demeter
  - “Don’t talk to strangers“
- More specifically
  - “Use only one dot“

Better Design?

- But also **subjective opinions**
  - Listen to your tests
    - Bad tests signal bad design
  - Code smells
    - Kevin Rutherford: The problem with code smells
  - Gut Driven Development?
    - Kevin Mahoney: Understand your biases, e.g.
    - Something is better because it is familiar to you
    - Something is better because it is new
    - **So examine your opinions and find some objective advantage**



Why Refactor?

~~Quality~~

~~Clean Code~~

~~Professionalism~~

~~Right Thing~~

Economics

Economics of software change

- Cost of reading and understanding
- Cost of finding the things that have to change
- Cost of making the change
- Cost of testing the change
- Cost of building and deploying the change
- Cost of maintaining the change

Coupling and Cohesion

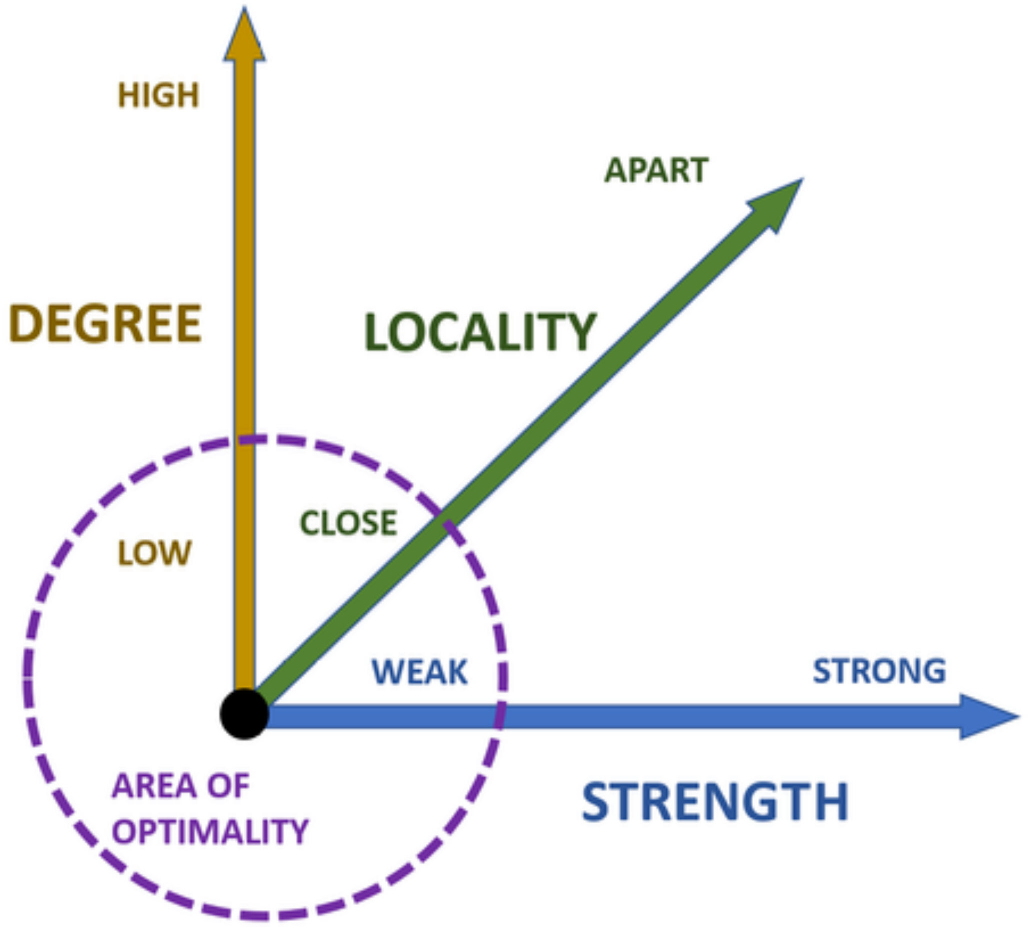
- Coupling
  - Degree of interdependence between software components
- Cohesion
  - Degree to which the elements of a component belong together
- Good design: low coupling, strong cohesion
  - So cohesion is "good coupling" because it is local!?
- Let’s go multi-dimensional!

Connascence

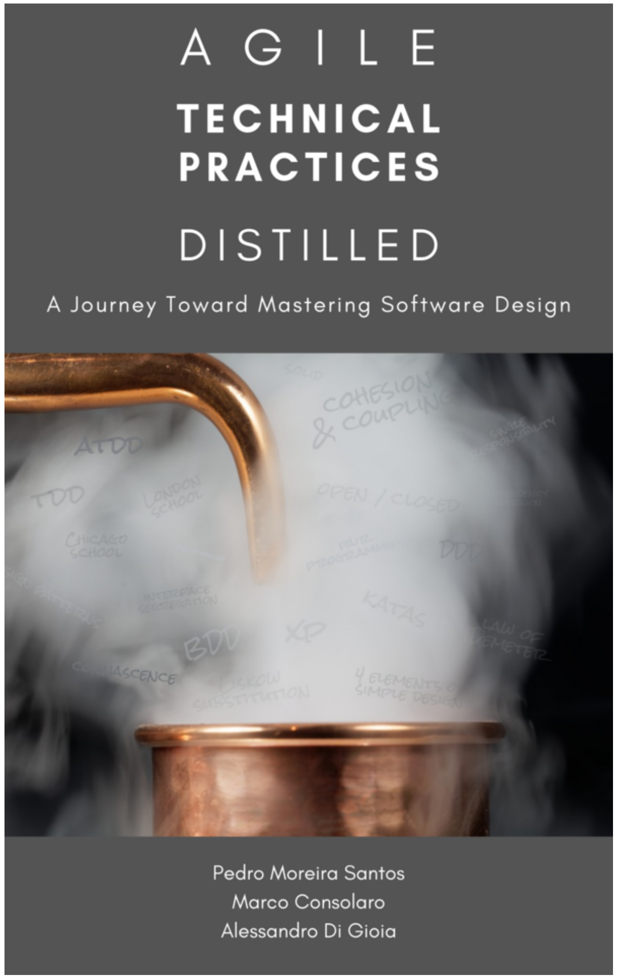
- **Meaning**
- “being born and growing together“

- **Definition (software)**
- Two or more elements are connascent when a change in one element requires a change in the others in order to keep the system working correctly

3 Dimensions of Connascence



- Pedro Moreira Santos, Marco Consolaro and Alessandro Di Gioia: Agile Technical Practices Distilled
- https://leanpub.com/agiletechnicalpracticesdistilled



Degree

- lower is better
- how many elements are involved in this connascence (**multiplicity**)
- higher degree means: harder to discover, especially all of them

Locality

- closer is better
- how close together in our codebase (functions, objects, packages, modules, applications, ...) are the elements of this connascence
- closer means: easier to discover

Strength

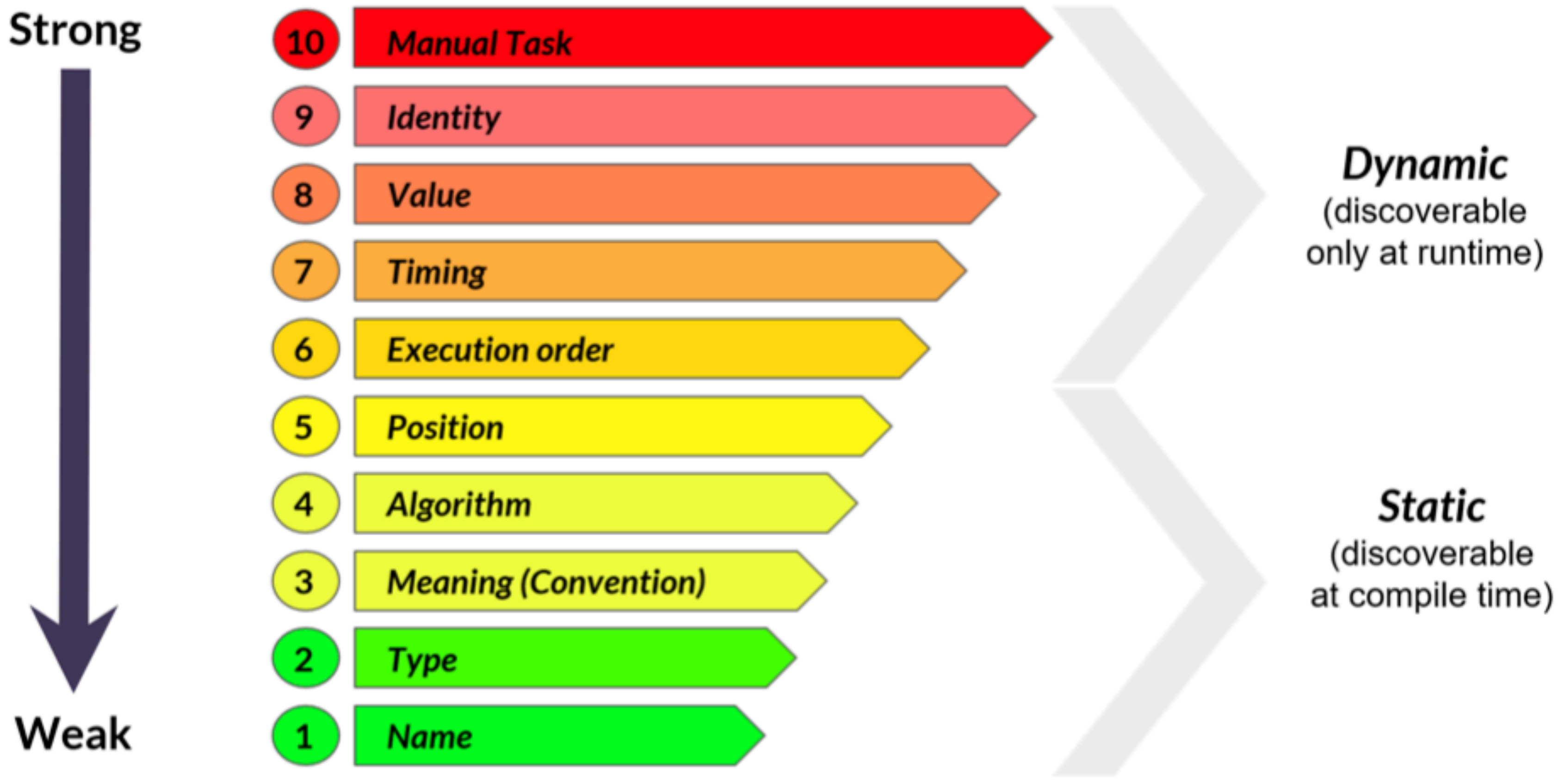
- weaker is better
- how strong is the **type** of this connascence
- stronger means: harder to discover (and harder to refactor)

3 Refactoring Options

Towards better design

1. reduce degree
2. increase locality
3. reduce strength

Types of Connascence



Connascence of Name

- obvious
- caller has to know
  - name of method
  - name of variable for type upon method is called

Connascence of Type

- obvious
- caller has to know
  - type of object to know what methods / messages are available
  - type of parameters and return values (obvious in static languages)
- not so obvious in dynamic languages

Connascence of Type

- examples
- signature

```
age (day, month, year)
```

- what is right?

```
age (1, 1, 1970)
age ("1", "1", "1970")
age (1, "January", 1970)
```

Connascence of Convention

- aka Connascence of Meaning
- when two or more components must agree on the meaning of specific values, hence using a convention

Connascence of Convention

- examples
- java compareTo

  -1, 0, +1
- meaning of null

  not found, invalid argument, …
- money representation
- flags

  true, false
- modes

Connascence of Algorithm

- when two or more components must agree on using a particular algorithm
- examples:
  - frontend / backend validation
  - all kinds of symmetries: encrypt/decrypt, encode/decode, compress/decompress

Connascence of Position

- when multiple components must be adjacent or must appear in a particular order
- often when passed within a positional structure like an array or a tuple
- examples
  - constructor / method parameters all strings vs. Builder pattern (or named parameters)
  - List (order of entries with different meaning) -> Map (name of entries)

Connascence of Execution Order

- when the caller of a component must have some unexpressed knowledge about the correct order of the methods to be called
- examples
  - stateful usage: set, set, get

Connascence of Timing

- when the success of two or more calls depends on the timing of when they occur
- examples
    - threads

Connascence of Value

- when two or more components’ values are related or have an intrinsic range of validity in their input not expressed by their primitive types.
- examples
    - same values in production-code and test-code
    - special values known in lots of places
    - implicit validity of values

Connascence of Identity

- when one or more components must reference exactly one particular instance of another entity to work correctly
- examples
  - message queues and listeners for topics (test only 1, prod many - bind queues / listeners to topics, else messages out of order)
  - using ORM frameworks (different instances of entities; state and re-reading from persistence)

Locality: Remote APIs

- Originally more about function level and class level
- How about Microservices? Serverless?
- e.g. REST APIs: same types
  - service discovery, alias and redirection, api gateways
  - which commands are valid? (duplicate logic or HAL)
  - encode / decode + assemble / parse json docs (CoA, duplicate conventions etc)
  - "Ironically the more generic your API the more coupling you create“ (Oliver Drotbohm)

Test Driven Design

- Transformation Priority Premise
  - tells us what test to write next
  - what is the next behavior that requires the least complex code
- Connascence
  - tells us what refactoring to apply next
  - still trade-offs (for example: strength vs locality)

Contact

Thorsten Brunzendorf codecentric

@thbrunzendorf

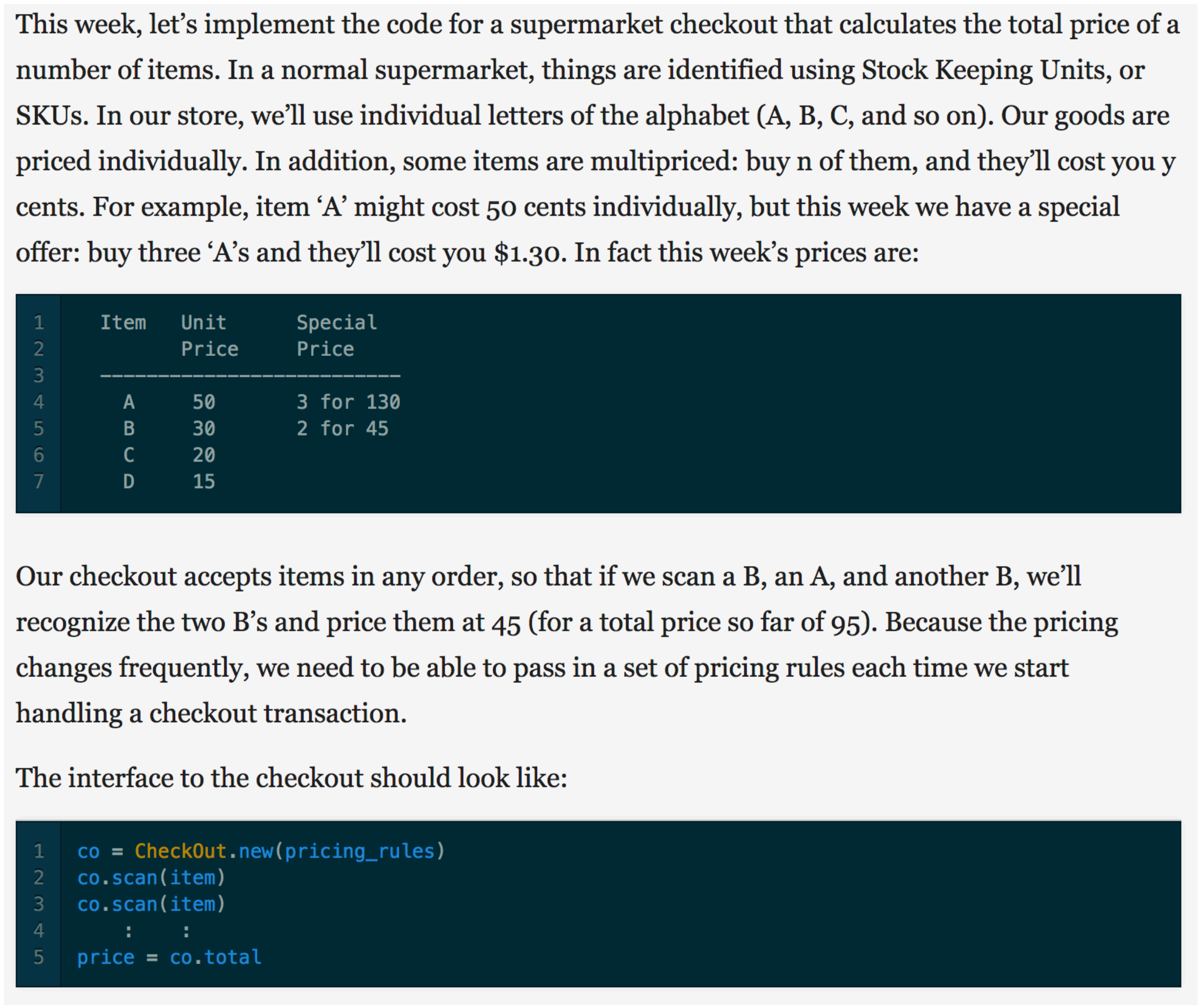
thorsten.brunzendorf@codecentric.de

Thank you!

Hands-on coding

- Kata applying Connascence
- “Back to the Checkout“ (Dave Thomas @PragDave)
  - http://codekata.com/kata/kata09-back-to-the-checkout/
- Exercised by Kevin Rutherford
  - https://silkandspinach.net/2015/01/22/connascence-of-value/
  - https://www.youtube.com/watch?v=fSr8LDcb0Y0

Back to the Checkout



This week, let's implement the code for a supermarket checkout that calculates the total price of a number of items. In a normal supermarket, things are identified using Stock Keeping Units, or SKUs. In our store, we'll use individual letters of the alphabet (A, B, C, and so on). Our goods are priced individually. In addition, some items are multipriced: buy n of them, and they'll cost you y cents. For example, item 'A' might cost 50 cents individually, but this week we have a special offer: buy three 'A's and they'll cost you $1.30. In fact this week's prices are:

```
Item   Unit      Special
       Price     Price
--------------------------
  A     50       3 for 130
  B     30       2 for 45
  C     20
  D     15
```

Our checkout accepts items in any order, so that if we scan a B, an A, and another B, we'll recognize the two B's and price them at 45 (for a total price so far of 95). Because the pricing changes frequently, we need to be able to pass in a set of pricing rules each time we start handling a checkout transaction.

The interface to the checkout should look like:

```
co = CheckOut.new(pricing_rules)
co.scan(item)
co.scan(item)
    :    :
price = co.total
```

Short Retrospective

- What did you observe?
- How has it felt?
- What would you like to try next time?

References

https://refactoring.com/
https://martinfowler.com/articles/workflowsOfRefactoring/
https://twitter.com/kentbeck/status/250733358307500032
https://blog.jbrains.ca/permalink/the-four-elements-of-simple-design
https://silkandspinach.net/2012/09/03/the-problem-with-code-smells/
http://kevinmahoney.co.uk/articles/gut-driven-development/
https://twitter.com/sandromancuso/status/588503877235781632
https://codurance.com/2015/05/12/does-tdd-lead-to-good-design/
https://twitter.com/mbohlende/status/431446680874258433
https://leanpub.com/agiletechnicalpracticesdistilled
https://speakerdeck.com/olivergierke/rest-beyond-the-obvious-api-design-for-ever-evolving-systems
https://twitter.com/olivergierke/status/997354136064204800
https://www.codesai.com/2017/01/about-connascence
https://silkandspinach.net/tag/connascence/
http://xpsurgery.com/resources/connascence/
https://silkandspinach.net/2015/05/20/red-green-what-now/
http://connascence.io/
http://confreaks.tv/videos/aac2009-the-grand-unified-theory
http://codekata.com/kata/kata09-back-to-the-checkout/