Domain *Re-discovery* Patterns for Legacy Code

DOMAIN DRIVEN DESIGN EUROPE
31.05.24

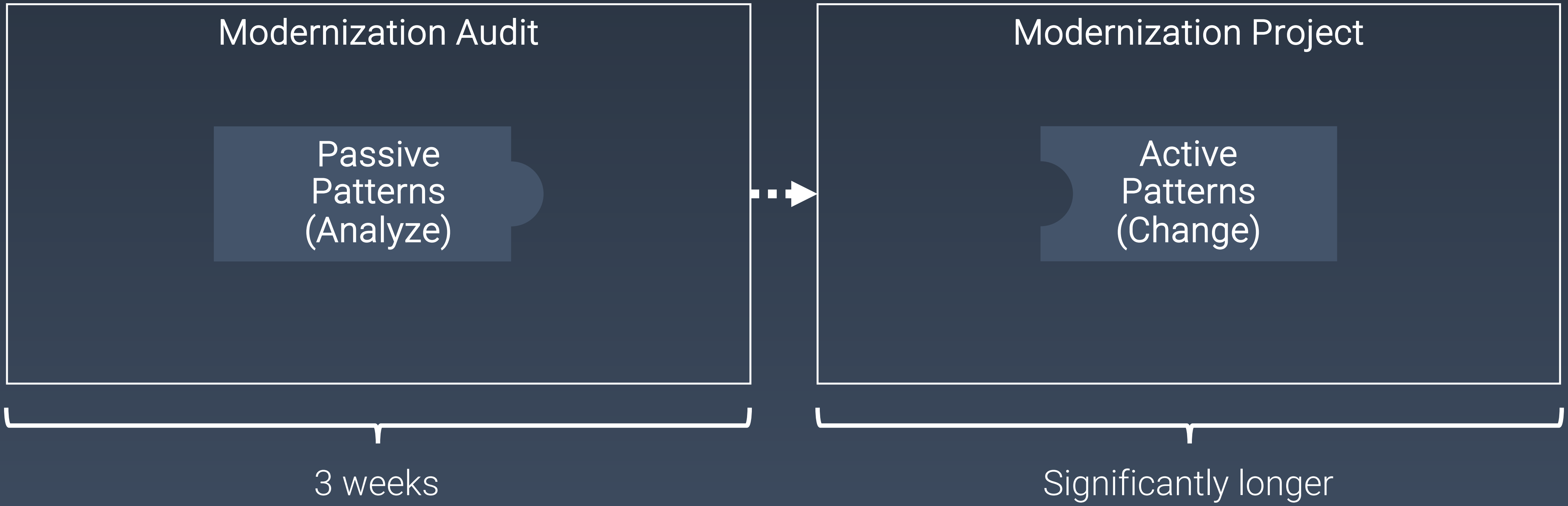
Richard Gross (he/him)

Archaeology TestDSLs Hypermedia

speakerdeck.com/richargh
richargh.de/
@arghrich

MAIBORNWOLFF

A lot of helpful rediscovery patterns



Modernization Audit

Passive Patterns (Analyze)

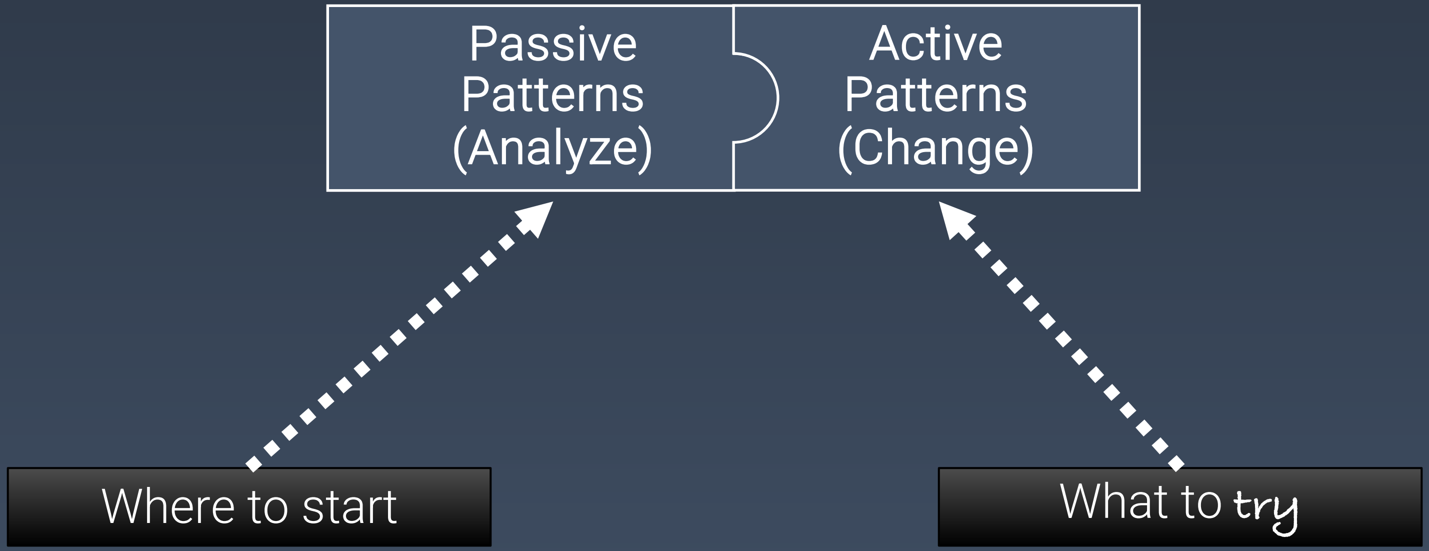
Modernization Project

Active Patterns (Change)

3 weeks

Significantly longer

A lot of helpful rediscovery patterns



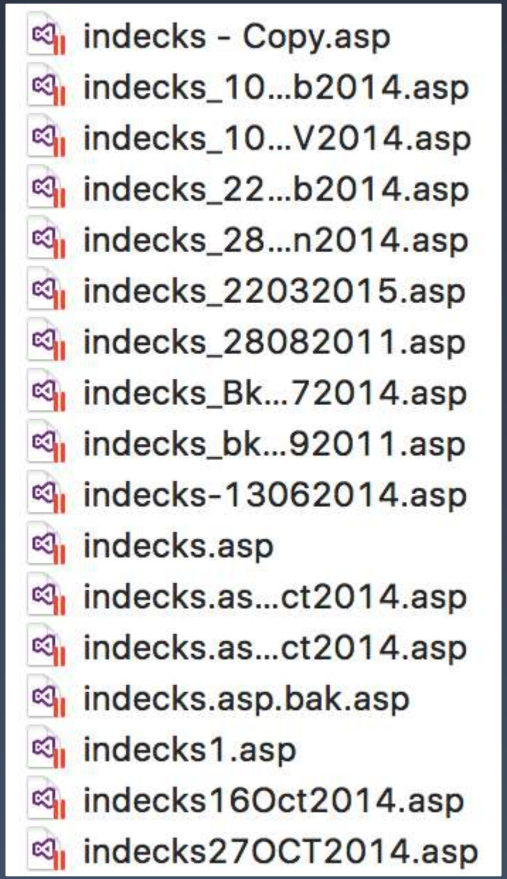
Passive Patterns (Analyze)

Active Patterns (Change)

Where to start

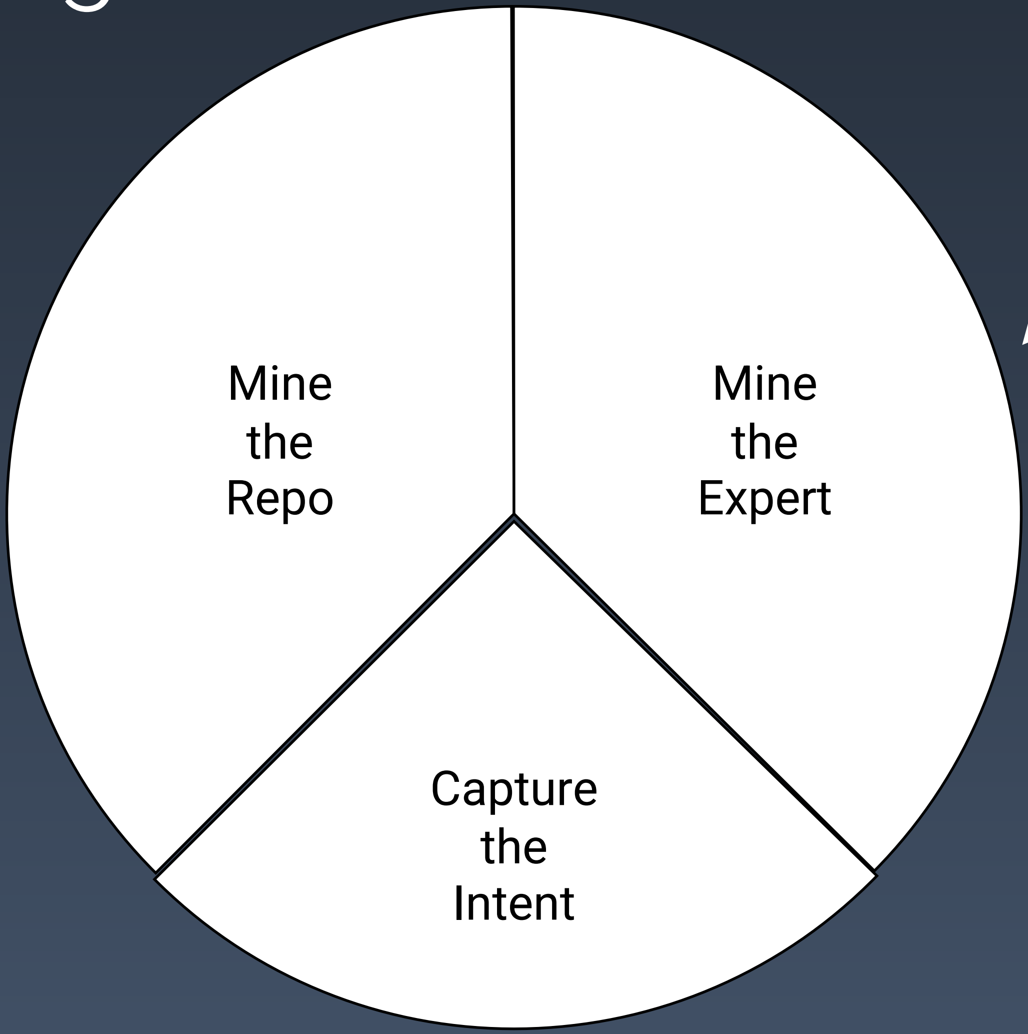
What to *try*

Some Modernizations required *dirtier* patterns than others



```
indecks - Copy.asp
indecks_10...b2014.asp
indecks_10...V2014.asp
indecks_22...b2014.asp
indecks_28...n2014.asp
indecks_22032015.asp
indecks_28082011.asp
indecks_Bk...72014.asp
indecks_bk...92011.asp
indecks-13062014.asp
indecks.asp
indecks.as...ct2014.asp
indecks.as...ct2014.asp
indecks.asp.bak.asp
indecks1.asp
indecks16Oct2014.asp
indecks27OCT2014.asp
```

Pattern categories



Not the focus today

Slides by richargh.de and

The Dirty (but useful) Patterns

Our highest priority is to satisfy the customer by not changing what doesn’t need changing.

The second principle of the legacy software manifesto (if one is ever written).

Passive Pattern: Activity Logging

Context

- Know which code parts are reached often and *potentially* critical
- Know which code parts are not reached at all and are *potentially* obsolete

Approach

- Identify system entry points & deep interna, then log there
  - Alt: Prometheus Counter
- Count in production

Caveat

- Some things are cyclical yearly/monthly (reports)

Active Pattern: Legacy Toggle

Context

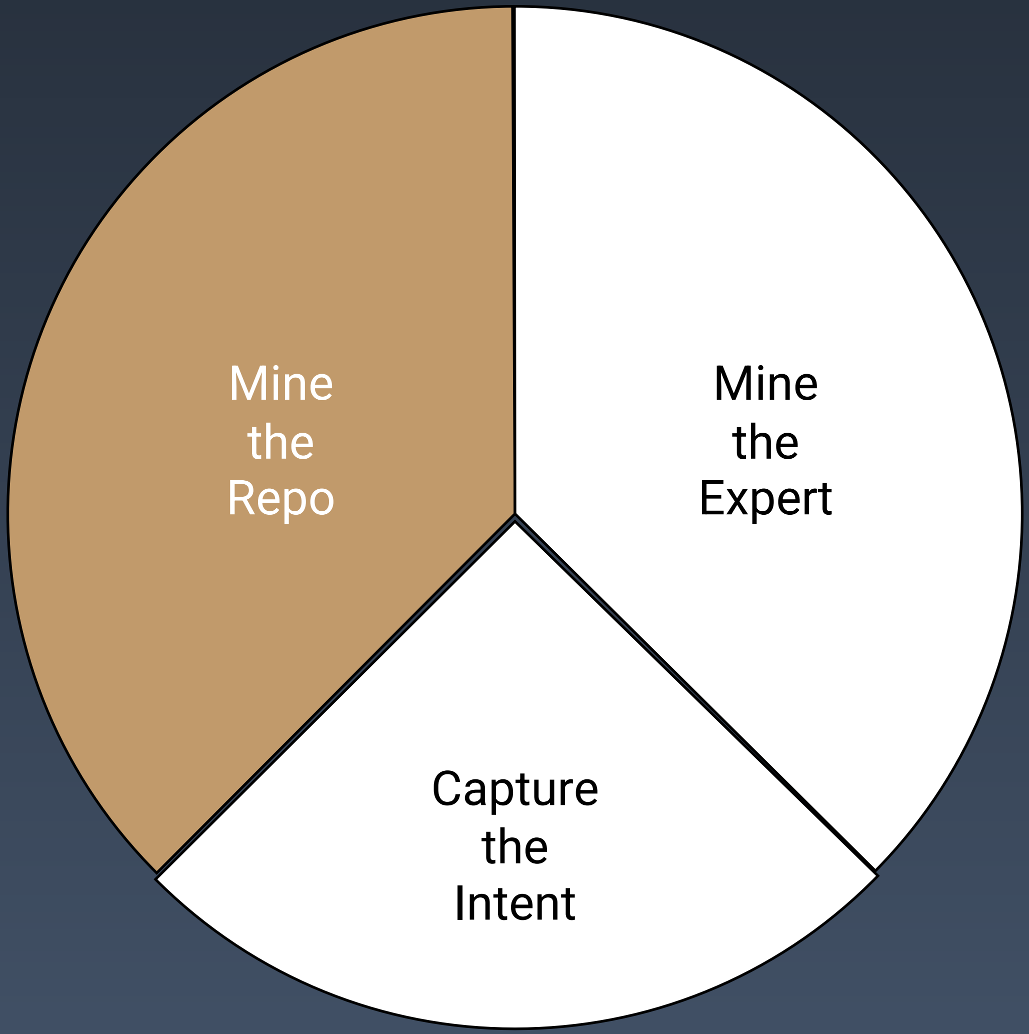
- Know if a feature really is obsolete and deletable

Approach

- Add a UI toggle, count if activated (soft)
- Deactivate in backend via env variable, reactivate env if someone complains (hard)
- Increasing Thread.sleep before answer (evil)
- Return static result, see if someone complains (rockstar)

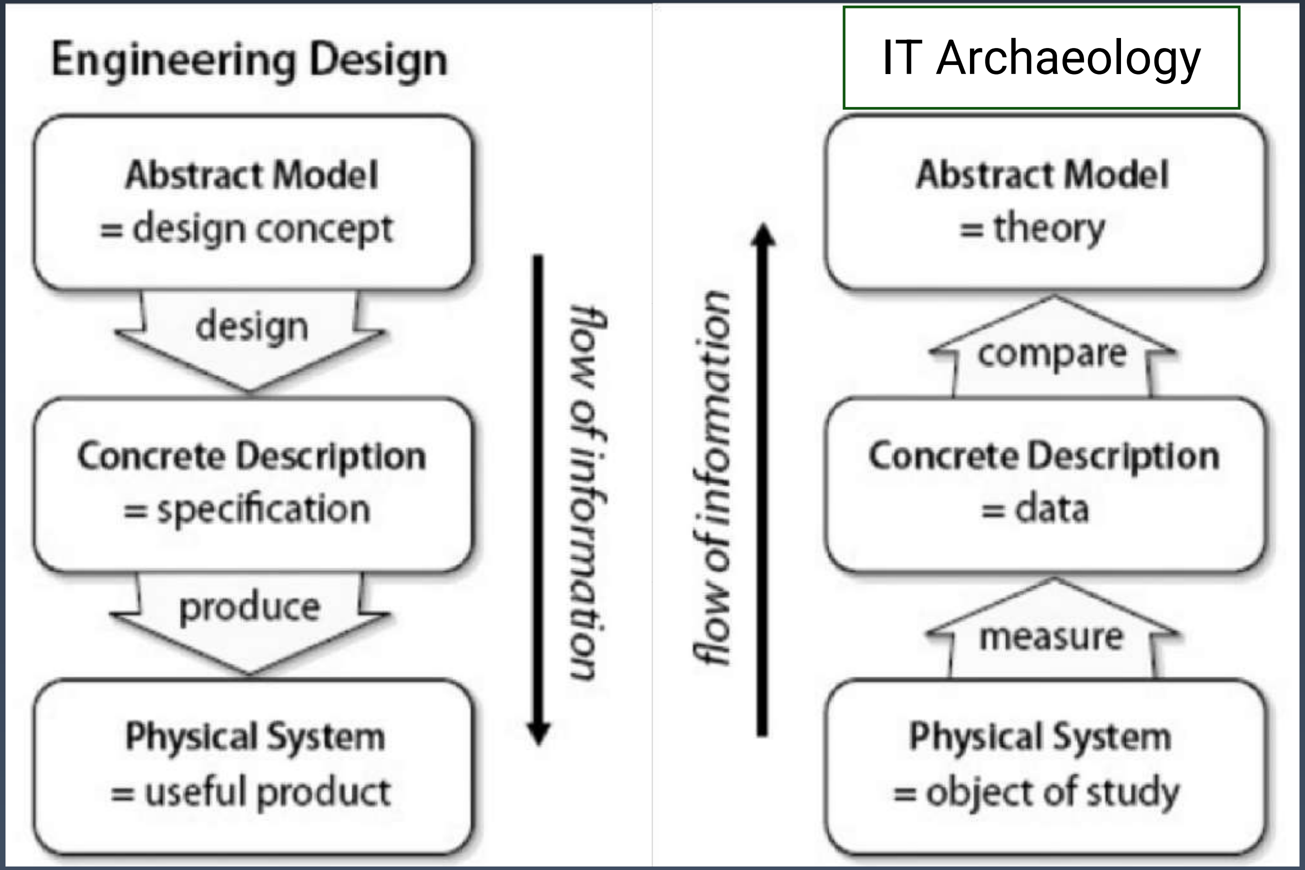
Caveat

- Some things are cyclical (reports)
- People still might not complain



Slides by richargh.de and

We’re working Backwards from Code



Engineering Design

Abstract Model
= design concept

design

Concrete Description
= specification

produce

Physical System
= useful product

flow of information

IT Archaeology

Abstract Model
= theory

compare

Concrete Description
= data

measure

Physical System
= object of study

flow of information

https://fs.blog/2013/07/the-difference-between-science-and-engineering/

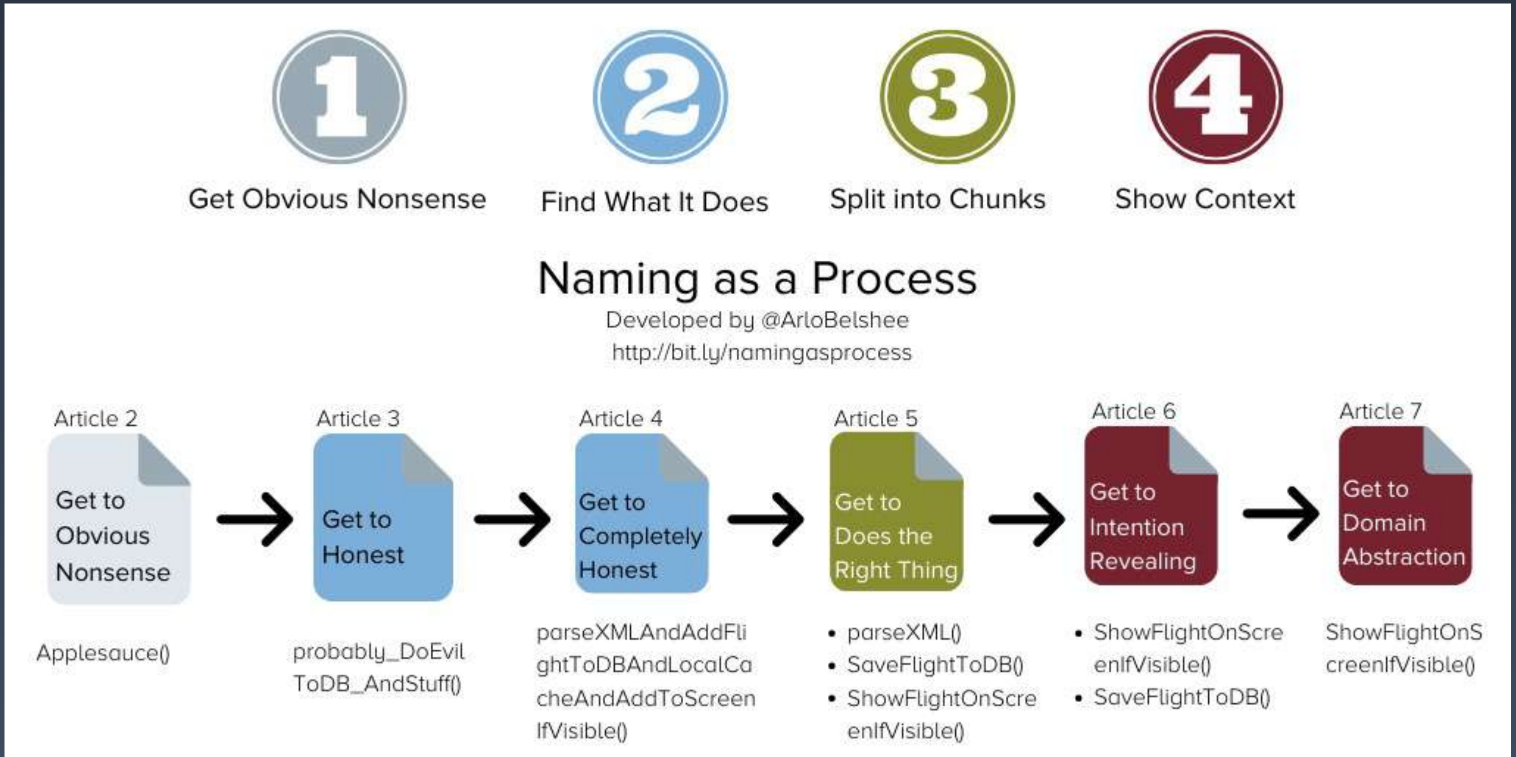
Warning: we'll be talking about *metrics*

1. ⚠ Every measure which becomes a target becomes a bad measure[1]
2. ⚠ Metric hotspots are only conversation starters
3. ⚠ The truth is in the conversation

[1] https://en.wikipedia.org/wiki/Goodhart%27s_law

Warning: we’ll be talking about *names*

1. Naming is hard
2. Metrics tell us where to start refactoring
3. Refactoring helps us find what (new) concept to name
4. Finding a good name is still not immediate but a process



1 Get Obvious Nonsense

2 Find What It Does

3 Split into Chunks

4 Show Context

Naming as a Process

Developed by @ArloBelshee

http://bit.ly/namingasprocess

Article 2 — Get to Obvious Nonsense → Article 3 — Get to Honest → Article 4 — Get to Completely Honest → Article 5 — Get to Does the Right Thing → Article 6 — Get to Intention Revealing → Article 7 — Get to Domain Abstraction

Applesauce()

probably_DoEvilToDB_AndStuff()

parseXMLAndAddFlightToDBAndLocalCacheAndAddToScreenIfVisible()

- parseXML()
- SaveFlightToDB()
- ShowFlightOnScreenIfVisible()

- ShowFlightOnScreenIfVisible()
- SaveFlightToDB()

ShowFlightOnScreenIfVisible()

https://www.digdeeproots.com/articles/on/naming-process/

Slides by richargh.de and

Passive Pattern: Code Tag Cloud[1,2]

Context

- Get a high level overview of possible domain terms

Approach

- Generate a tag cloud from code by extracting either the names or the behaviors (invoked methods)

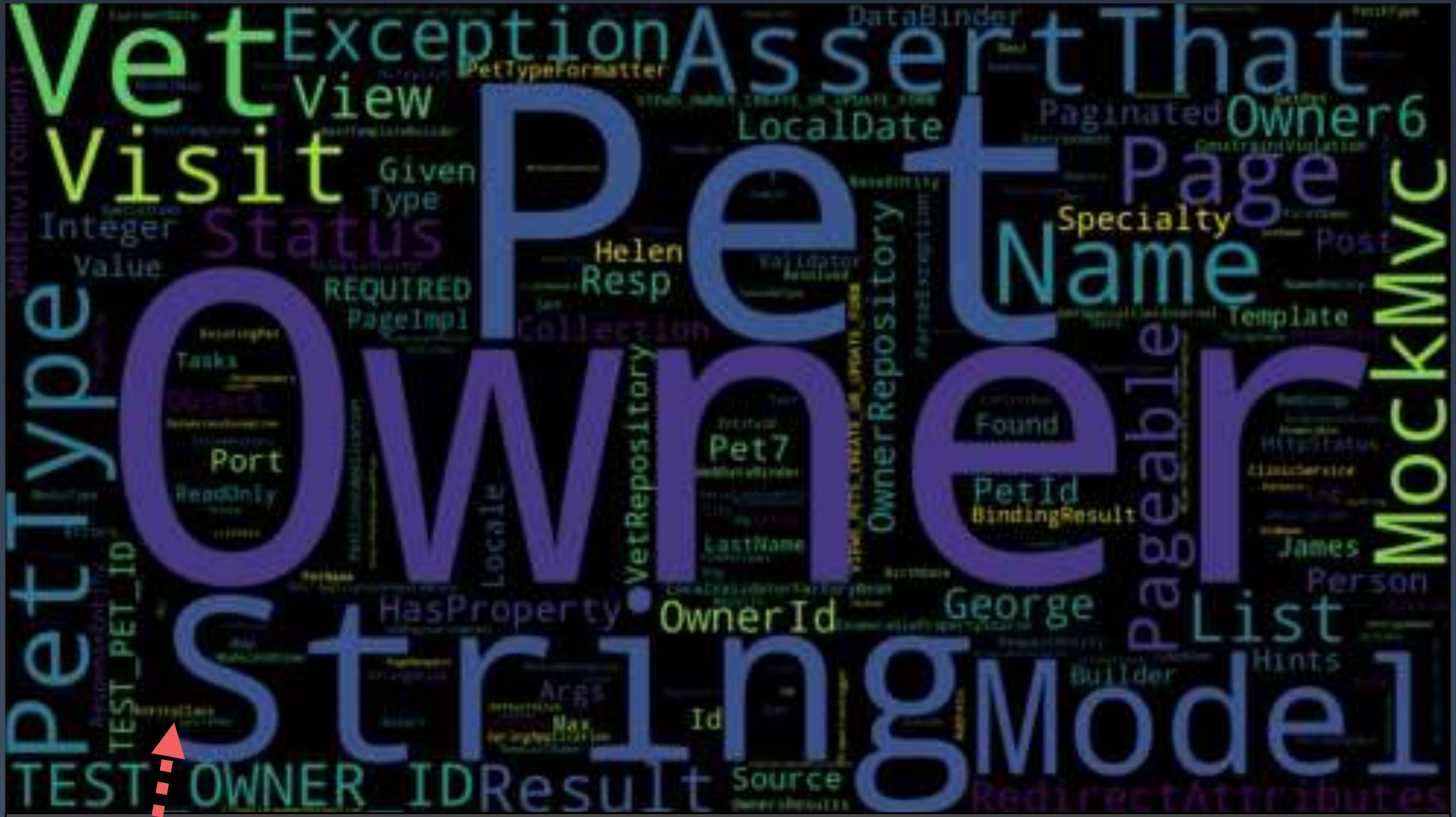
Caveat

- Repetition wins, not necessarily importance

[1] Original Idea, probably by Kevlin Henney https://youtu.be/SUIUZ09mnwM?feature=shared&t=2226
[2] Original tag cloud code: https://www.feststelltaste.de/word-cloud-computing/

Pattern: Code Tag Cloud

Discover the modeled names



```
./main.py ../oss/spring-petclinic/ java name
```

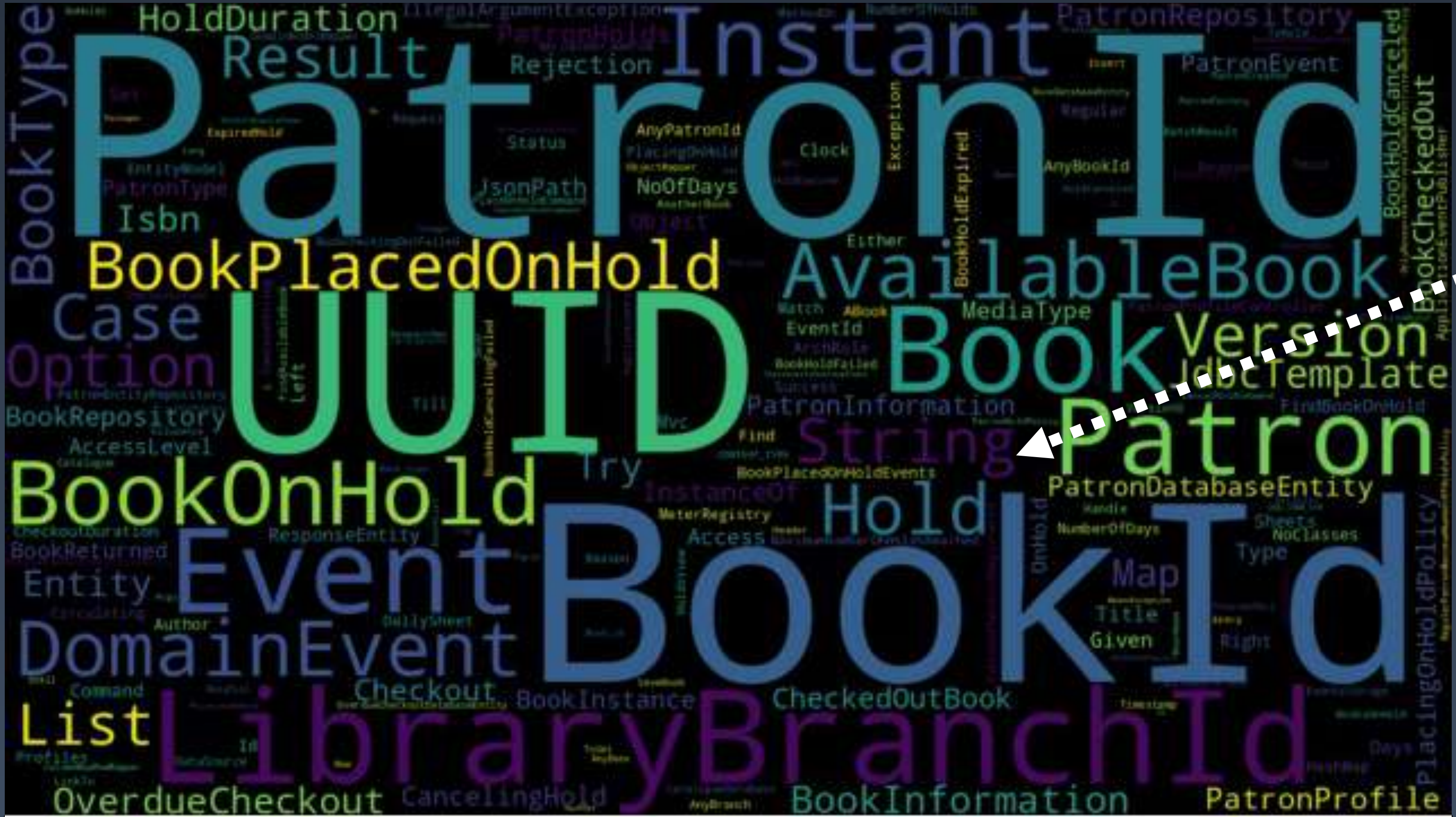
Stringly or Strongly Typed?

Generated with: https://github.com/Richargh/code-tagcloud-py-sandbox
Generated from: https://github.com/spring-projects/spring-petclinic

Slides by richargh.de and

Pattern: Code Tag Cloud

Your domain can be quite rich



```
./main.py ../oss/ddd-library/ java name
```

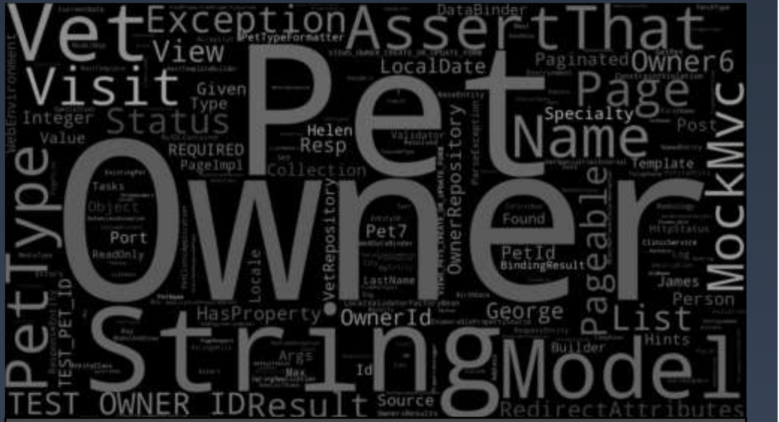
Where is “String”?

Generated with: https://github.com/Richargh/code-tagcloud-py-sandbox
Generated from: https://github.com/ddd-by-examples/library

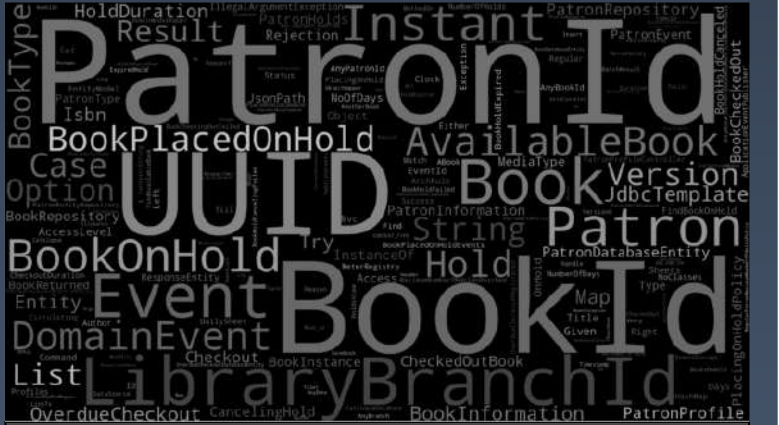
Pattern: Code Tag Cloud

Your service offering dictates name richness

Specific Service

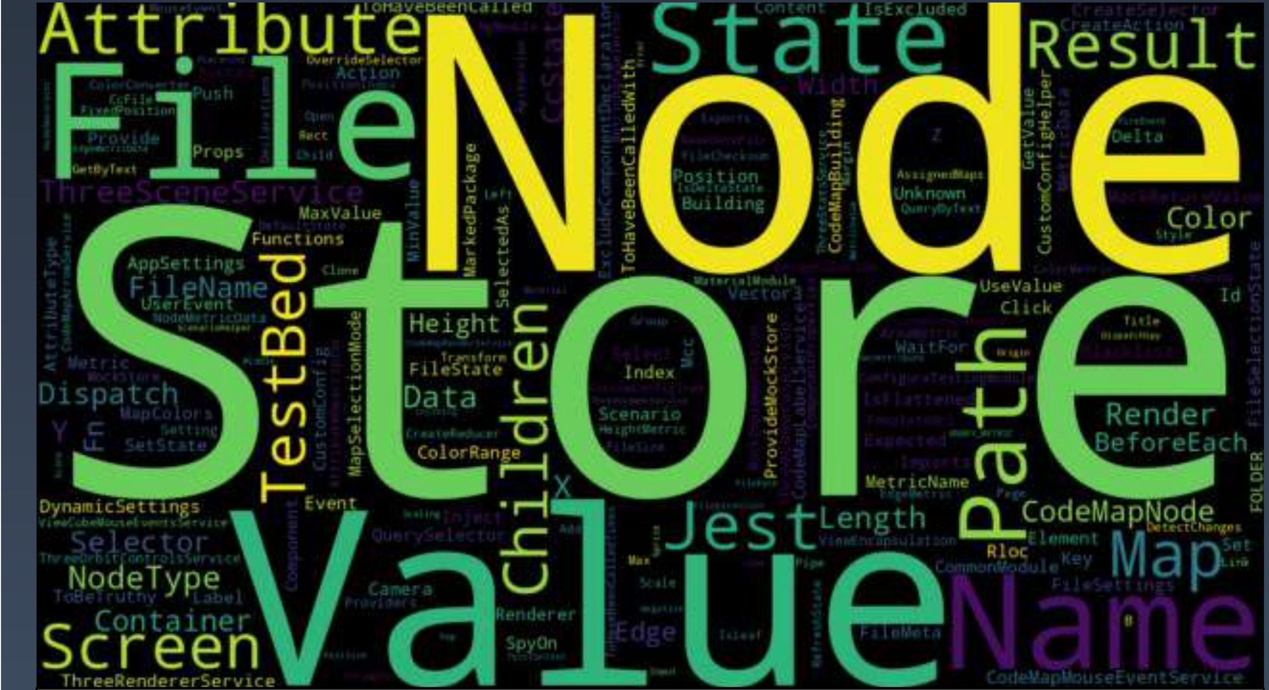


https://github.com/spring-projects/spring-petclinic



https://github.com/ddd-by-examples/library

Generic/customizable Service



https://github.com/MaibornWolff/codecharta/tree/main/visualization

Generated with: https://github.com/Richargh/code-tagcloud-py-sandbox

Slides by richargh.de and

Active Pattern: Strong[1] Code Tag Cloud

Bring the domain forward

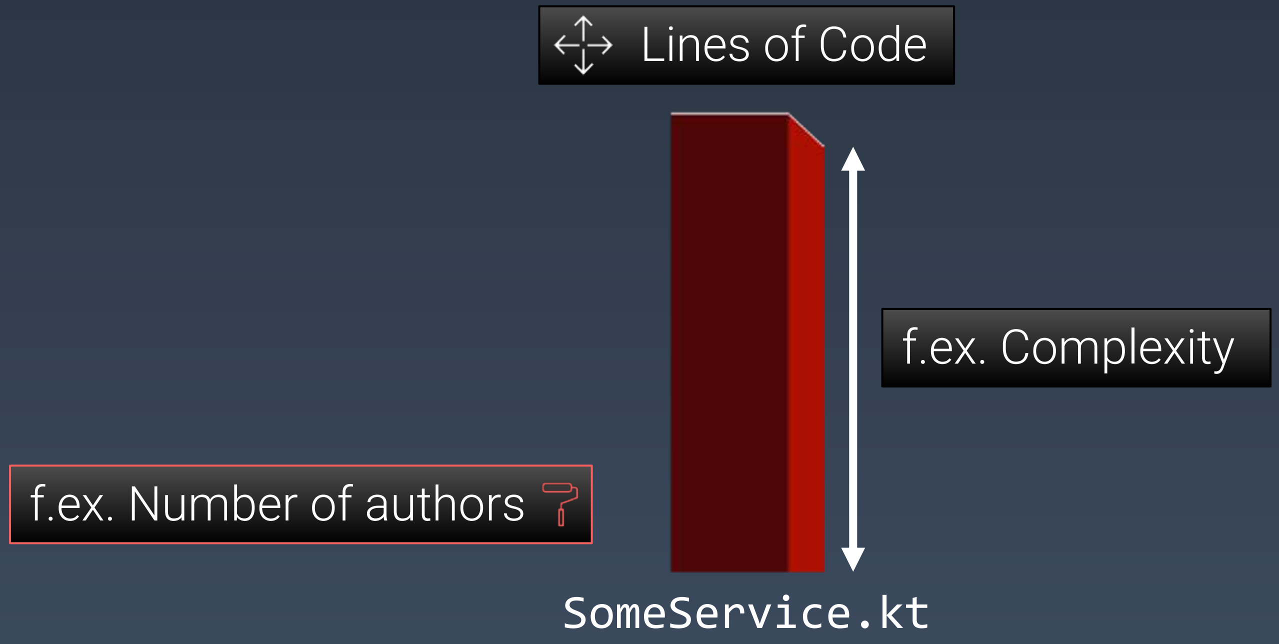
- Model your ids
  `record UserId(String rawValue){}`
  → Remove a bug source, see connections better
- Primitive Value Objects
  - Is it a anything-goes string or are there domain constraints?
  - Does a number have significance, can you give it a name[2] or type?
- Elements that are always passed/returned together
  → is there a domain concept missing?

[1] Opposite of stringly https://www.hanselman.com/blog/stringly-typed-vs-strongly-typed

[2] https://luzkan.github.io/smells/magic-number

Slides by richargh.de and

We‘re now visualizing metrics with CodeCharta[1] buildings



Lines of Code

f.ex. Complexity

f.ex. Number of authors

SomeService.kt

Plug by Zaufishan
Some icons by https://www.reshot.com/
Gource is a cool git visualizer https://gource.io/
CodeScene is a good Charta-alternative: https://codescene.com/

[1] CodeCharta is open-source https://maibornwolff.github.io/codecharta/

Slides by richargh.de and

Passive Pattern: Cluster Invest

Context

- Grasp the modeled structure based on which parts had the most code invest

Approach

1. Generate a tree-map of the code.
2. Highlight logical clusters that contain a lot of lines of code (LoC)

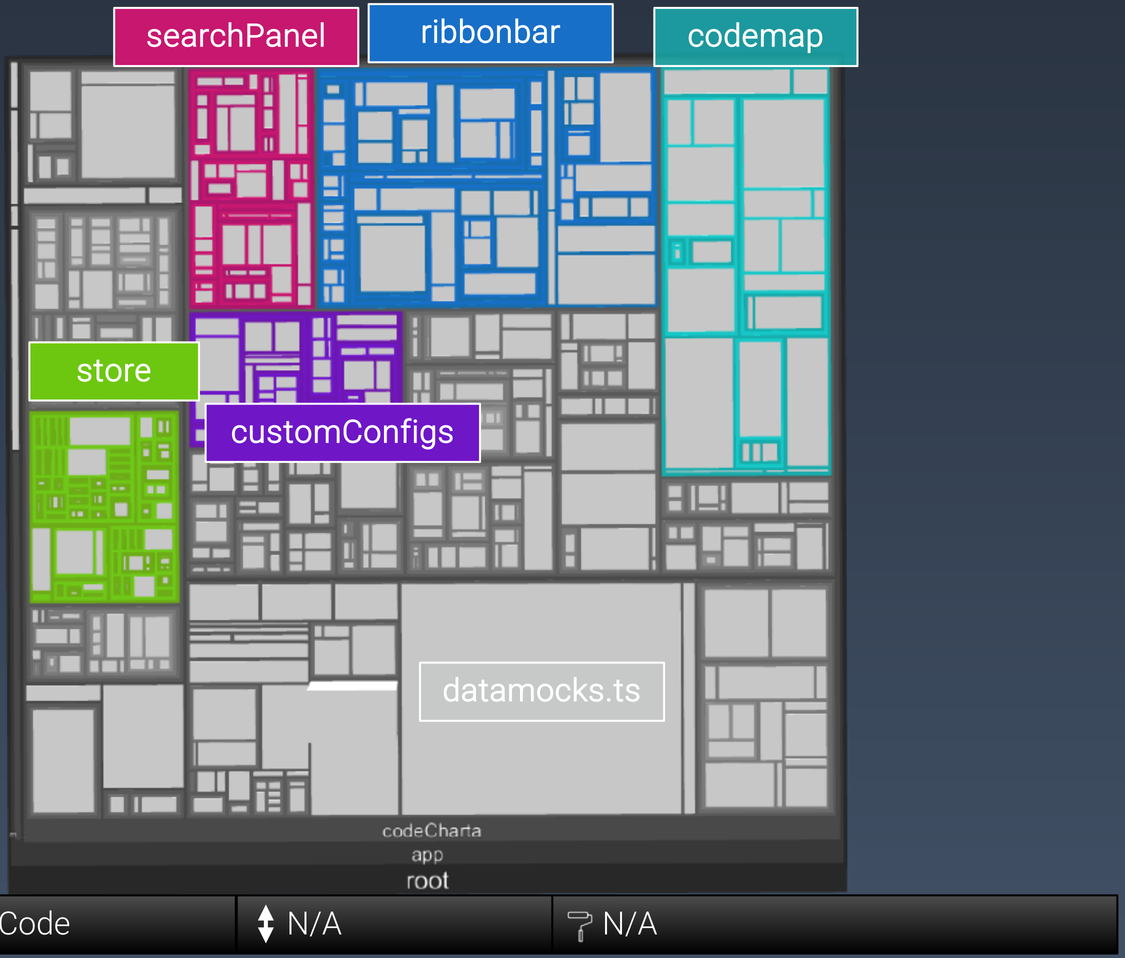
Caveat

- Shows accidental + essential complexity[1] not necessarily importance

[1] https://en.wikipedia.org/wiki/No_Silver_Bullet

Passive Pattern: Cluster Invest

Do the cluster names and sizes match domain expectations?

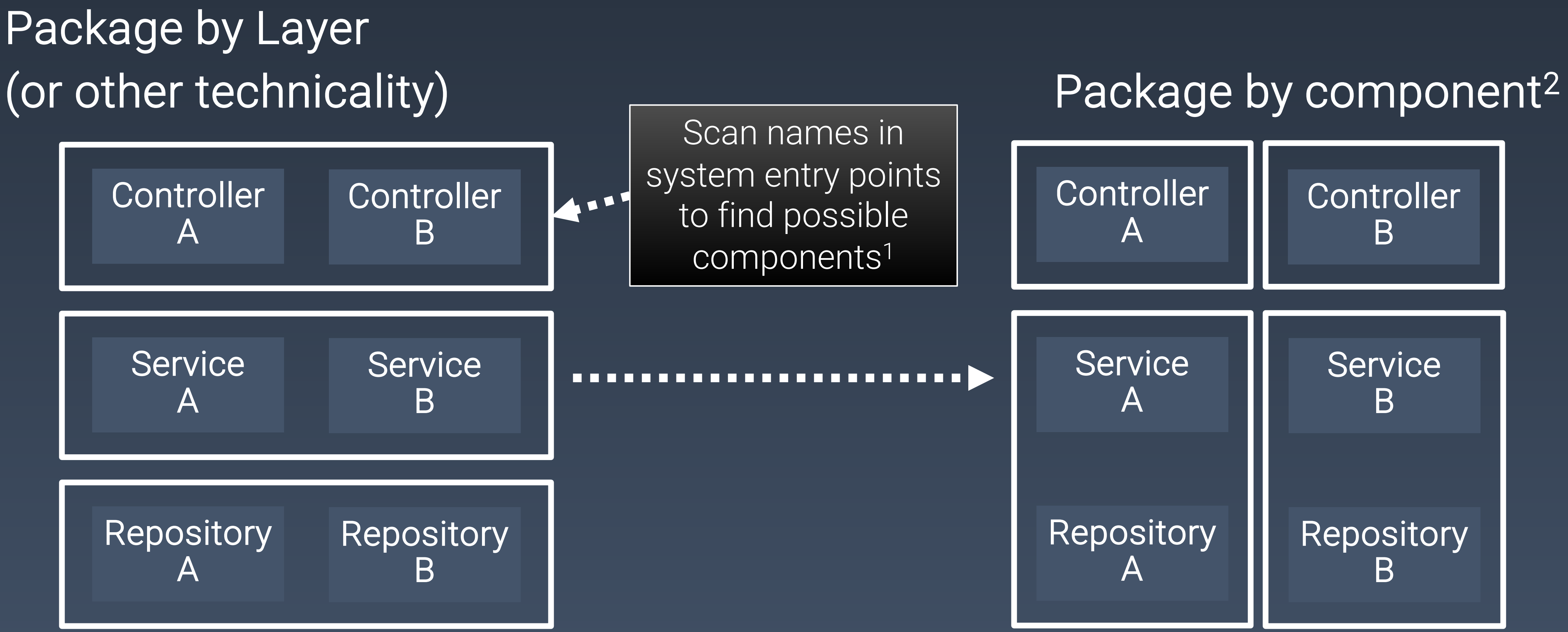


CodeCharta Code visualized by CodeCharta https://maibornwolff.github.io/codecharta/

Slides by richargh.de and

Active Pattern: Component Focus

When you see no meaningful domain clusters



[1] This is a lot easier the more you have discovered from other patterns

[2] https://dzone.com/articles/package-component-and

Slides by richargh.de and

Pattern: Complexity[1] Invest

Context

- Cyclomatic complexity[1] counts places where the control flow branches (if, for, catch, …).
- A lot of complexity is an indicator that domain decisions are being made.

Approach

1. In the code-map mark the places with a lot of complexity

Caveat

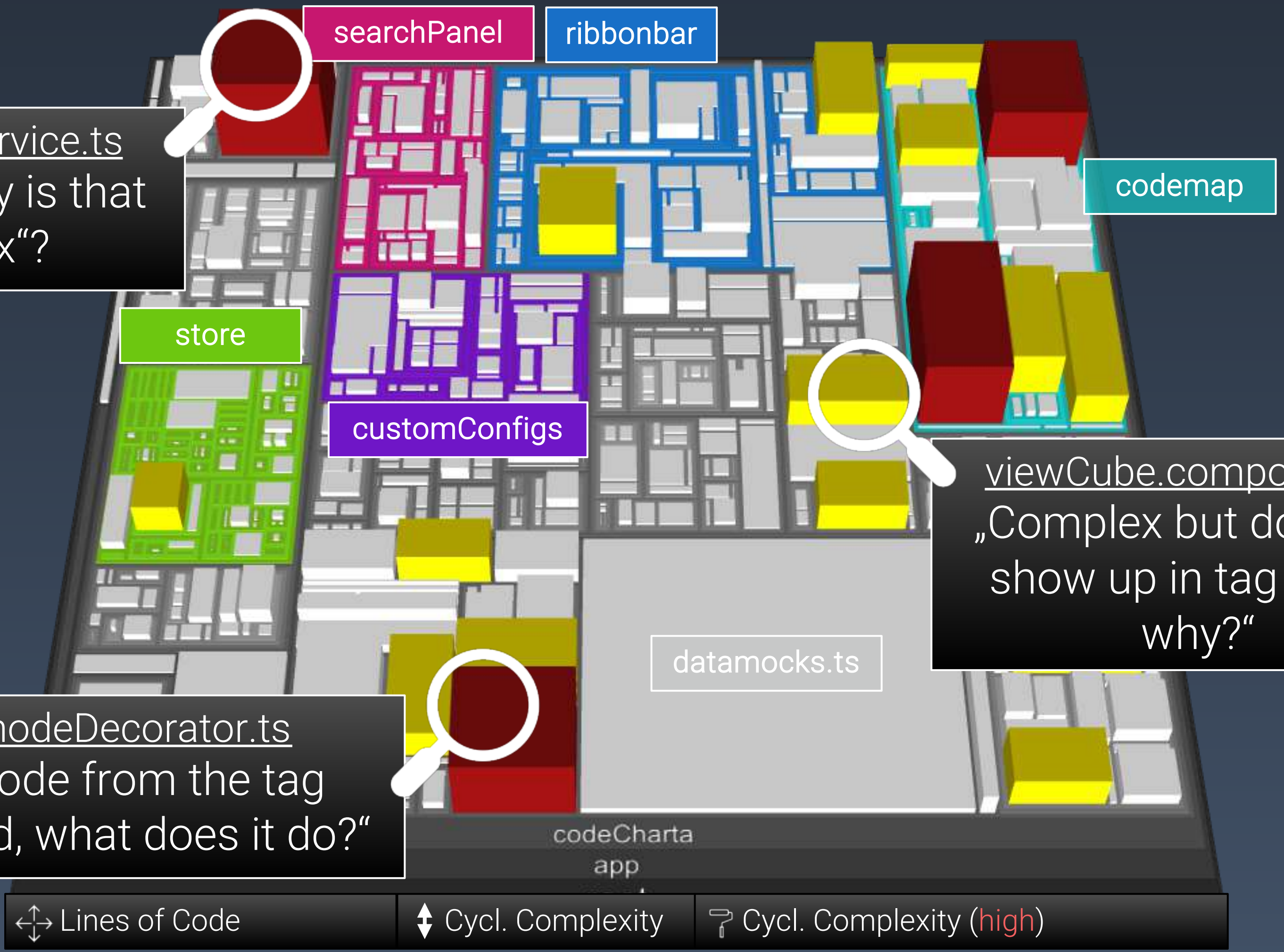
- Cyclomatic complexity penalizes switch cases heavily and ignores indendation[2,3]

[1] McCabe‘s cyclomatic complexity (MCC) counts branches in control flow (if, for, while, catch)

[2] Alternative: Cognitive Complexity https://www.sonarsource.com/resources/cognitive-complexity/

[3] Alternative: Indendation based „Bumby Road“ smell https://codescene.com/engineering-blog/bumpy-road-code-complexity-in-context/

Pattern: Complexity Invest



loadInitialFile.service.ts
„Interesting, why is that so complex“?

viewCube.component.ts
„Complex but does not show up in tag cloud, why?“

nodeDecorator.ts
„Node from the tag cloud, what does it do?“

CodeCharta Code visualized by CodeCharta https://maibornwolff.github.io/codecharta/

Slides by richargh.de and

Active Pattern: Complexity Limit

- Remove indentation with guard clauses
- `switch(anEnum) { case “A”: doThingA() }`
  → polymorphic dispatch `anABCobj.doThing();`
- Replace flag argument[1] with specific methods
- Separate presentation from domain from data[2]
- Finally group things that only interact with each other and extract as new type
- You now have new domain concepts to name

[1] Flag arguments https://martinfowler.com/bliki/FlagArgument.html

[2] Presentation Domain Data Layering https://martinfowler.com/bliki/PresentationDomainDataLayering.html

Slides by richargh.de and

Passive Pattern: Knowledge Silos

Context

- Code elements that are only changed by few authors are likely only understood by these authors.
- If the elements are complex and only have one author, we have a business risk as well.

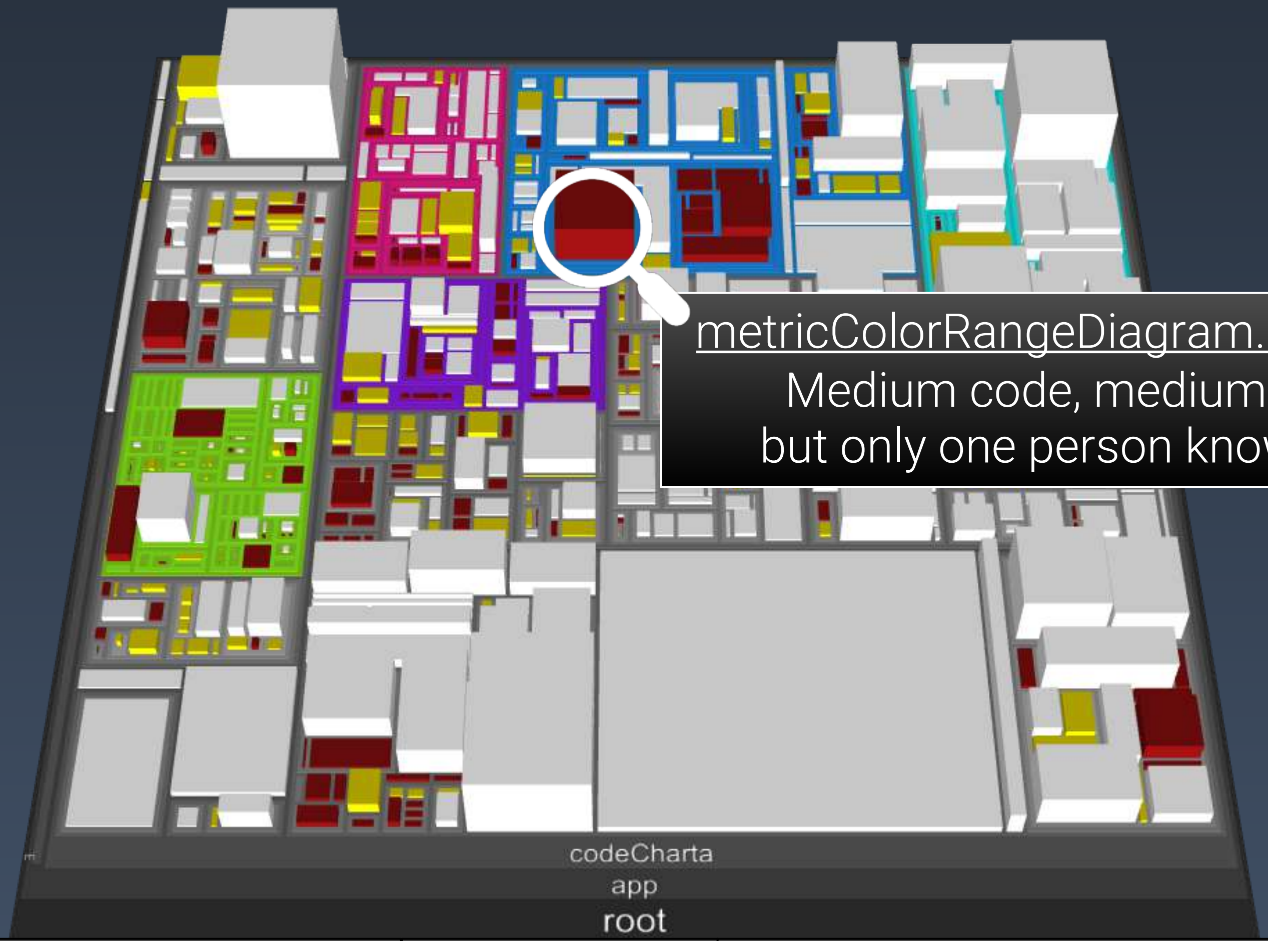
Approach

1. In the code-map, mark complex elements that have only 1 or 2 authors.
2. Hightlight elements where the author is about to leave or has left.

See also https://codescene.com/knowledge-distribution

Slides by richargh.de and

Passive Pattern: Knowledge Silos



metricColorRangeDiagram.component.ts

Medium code, medium complex,
but only one person knows about it

Lines of Code | Cog. Complexity | Number of authors (low)

CodeCharta Code visualized by CodeCharta https://maibornwolff.github.io/codecharta/

Slides by richargh.de and

Active Pattern: Knowledge Sharing

Context

- Mitigate the business risk of knowledge silos

Caveat

- Having everyone know everything is time-consuming and wasteful due to forgetfulness

Approaches

- Simple code
- Specification by (test) example
- “Owner” delegates changes and reviews
- Pair/Mob programming
- Dev Talk Walkthrough

See also https://codescene.com/knowledge-distribution

Slides by richargh.de and

Passive Pattern: Coordination Bottlenecks

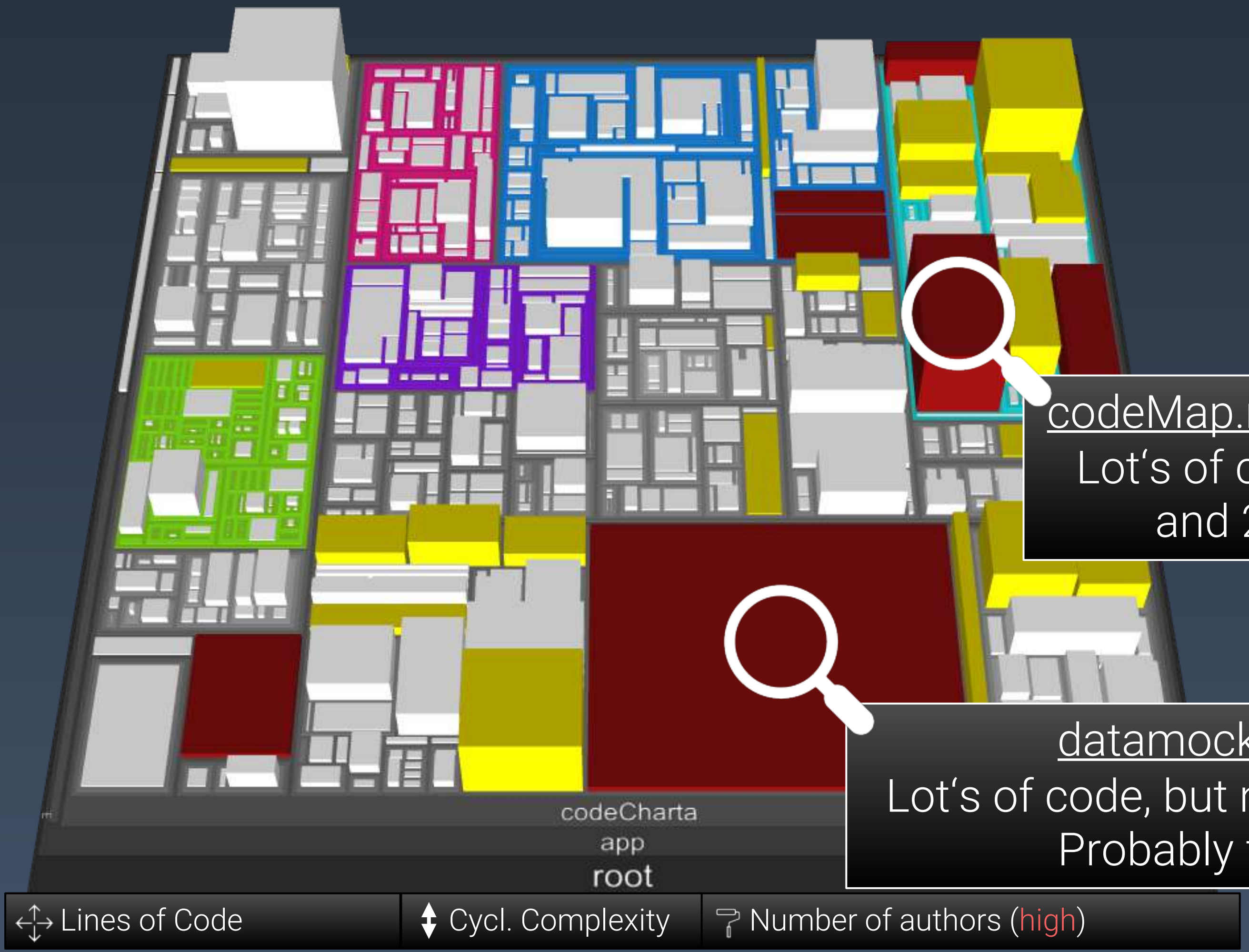
Context

- Code elements that everyone changes usually require extensive coordination to avoid conflicts.

Approach

1. In the code-map, mark complex elements where most of the team have made recent changes.

Passive Pattern: Coordination Bottlenecks



codeMap.mouseEvent.service.ts
Lot‘s of code, many decisions and 20 authors. Why?

datamocks.ts
Lot‘s of code, but no decisions. Probably fine.

CodeCharta Code visualized by CodeCharta https://maibornwolff.github.io/codecharta/

Slides by richargh.de and

Passive Pattern: Multi-Level Dependency Graph

Context

- Imports between elements mean coupling
- Code Coupling is (roughly) domain coupling
- Any circle (tangle) creates knots in our brain

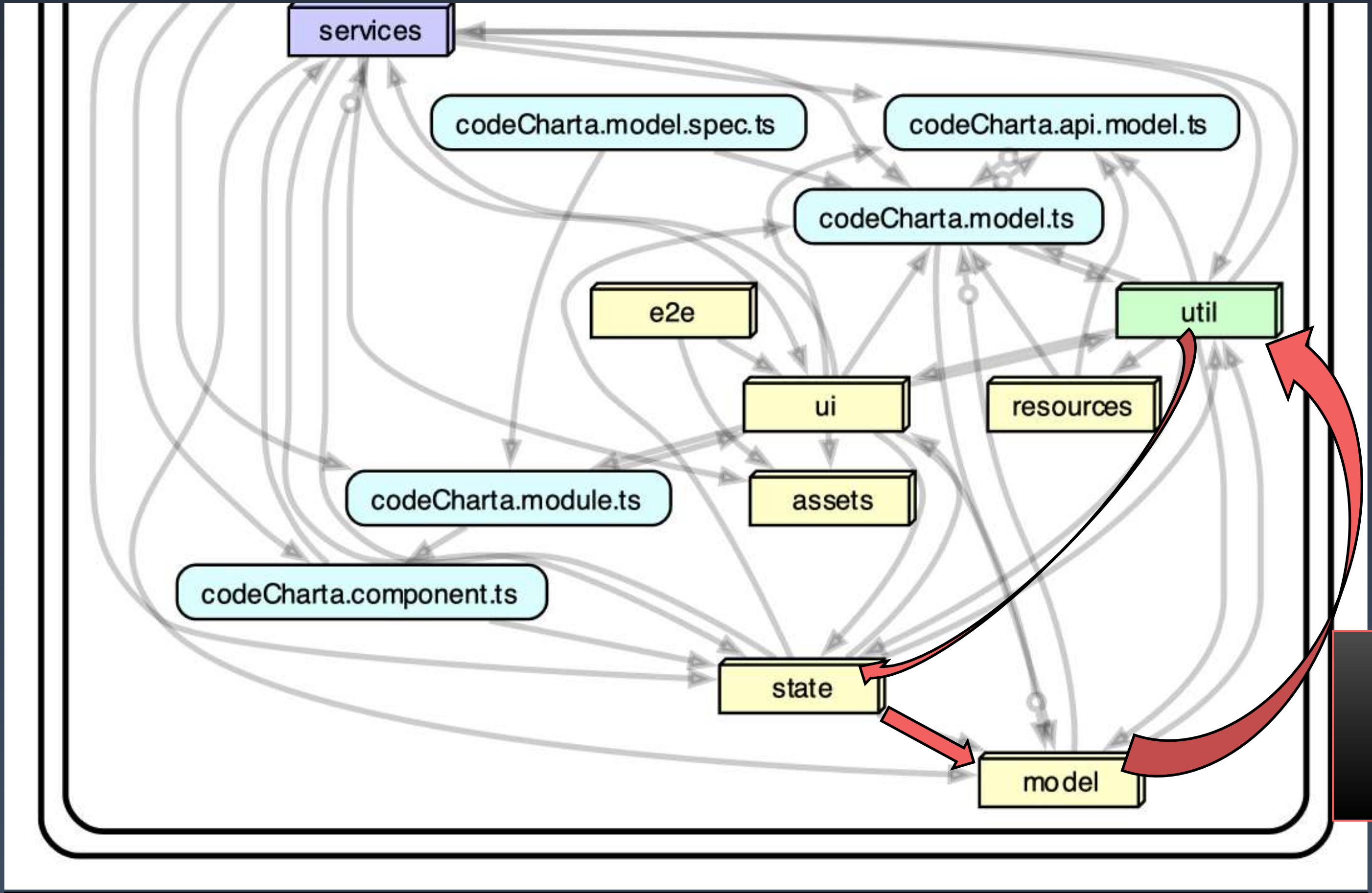
Approach

1. Graph the import statements between elements. Stable elements (with no dependencies) at the bottom.
2. Mark arrows that go "up" in red, they create tangles[1].
3. View graph, first on high-level, then focus on subsets.

[1] https://structure101.com/help/java/studio/Content/xs/tangle.html

Slides by richargh.de and

Passive Pattern: Multi-Level Dependency Graph



```
npx depcruise app/codeCharta/ --output-type dot | dot -T svg
```

Tangle
model → util
→ state → model

CodeCharta Code visualized by Dependency Cruiser https://github.com/sverweij/dependency-cruiser
Alternative: Structure101 and tangles: https://structure101.com/help/cpa/studio6/Content/restructure101/tangles.html

Slides by richargh.de and

Passive Pattern: Temporal Coupling[1]

Context

- If a change in A often requires a change in B, the elements are temporally coupled[1].
- The reason for this often *invisible* coupling is a *high Connascence*.

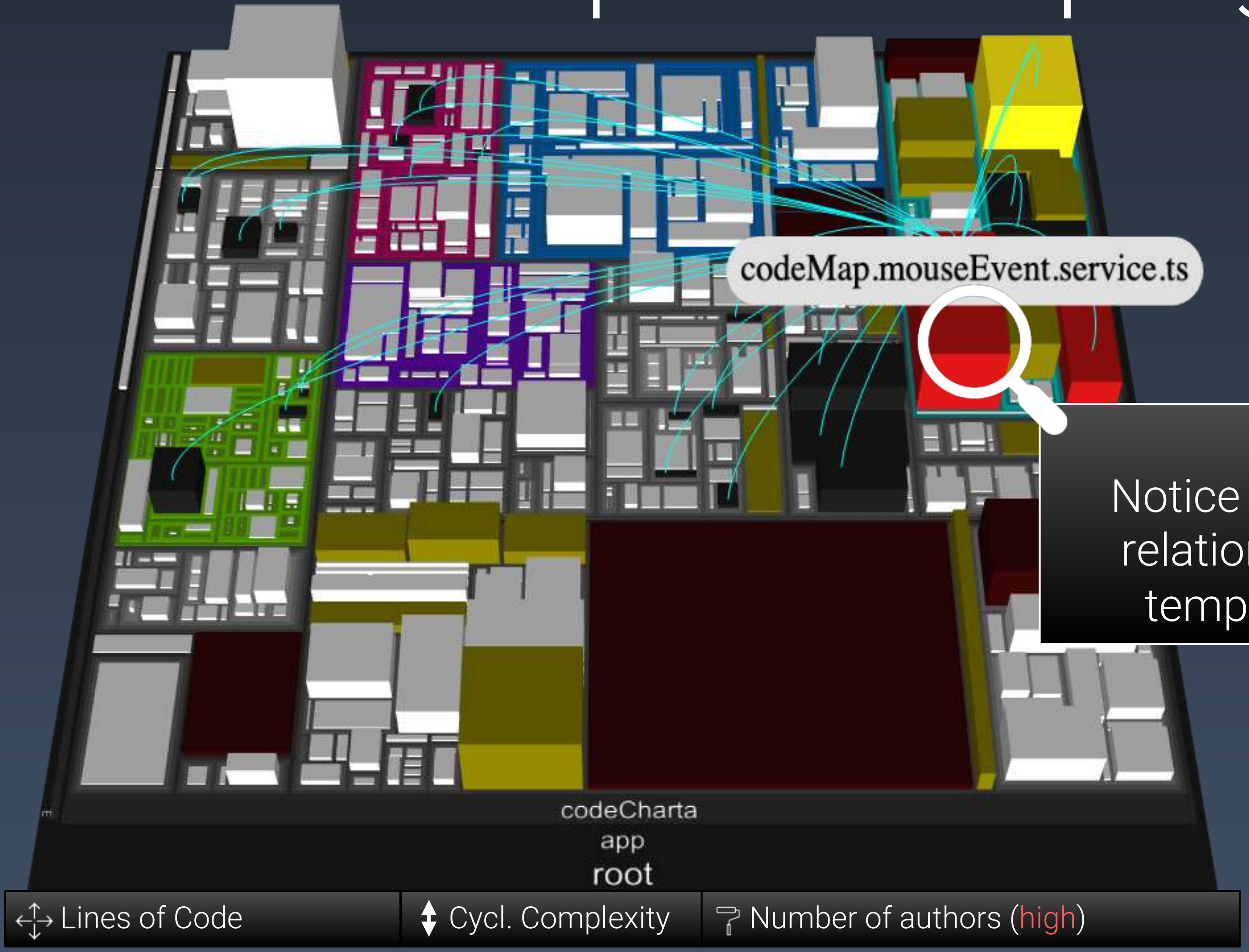
Approach

1. Count how often A was commited together with B. If high draw A → B.
2. Count how often B was commited without A. If high don't draw B → A.
3. In the code-map, mark these elements.

[1] from the book https://pragprog.com/titles/atcrime/your-code-as-a-crime-scene

Slides by richargh.de and

Passive Pattern: Temporal Coupling[1]



codeMap

Notice that no compile-time relation exists between the temporally coupled files.

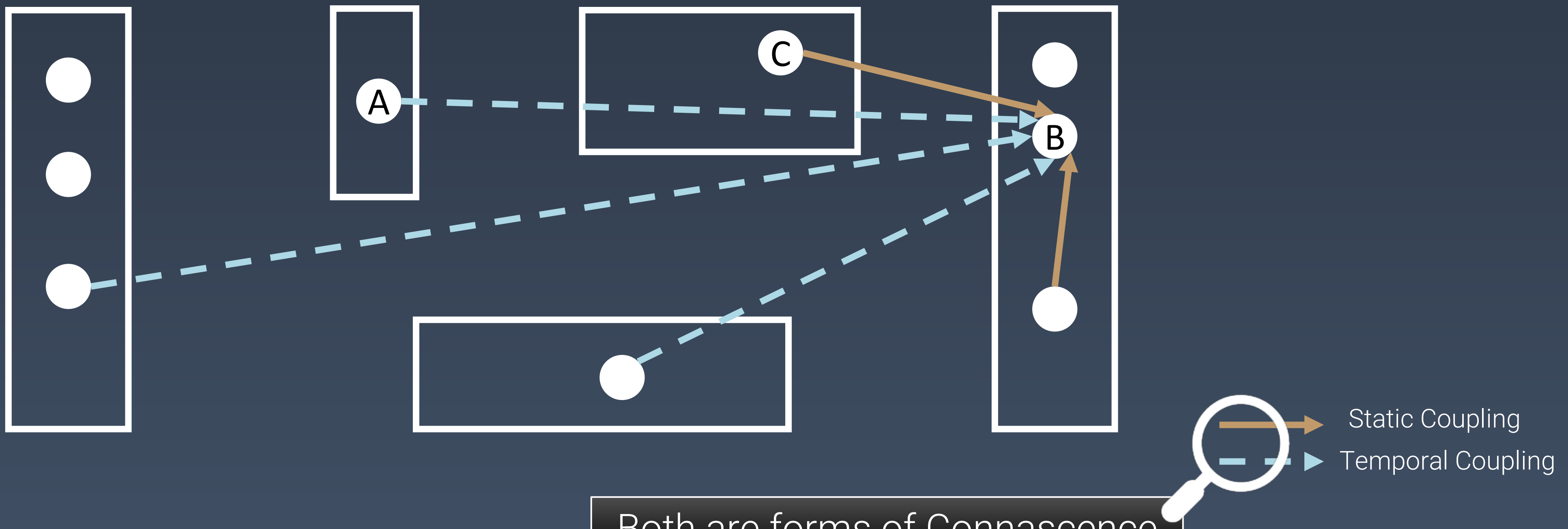
Ingoing Temporal Coupling

[1] from the book https://pragprog.com/titles/atcrime/your-code-as-a-crime-scene

CodeCharta Code visualized by CodeCharta https://maibornwolff.github.io/codecharta/

Slides by richargh.de and

How does temporal coupling happen?

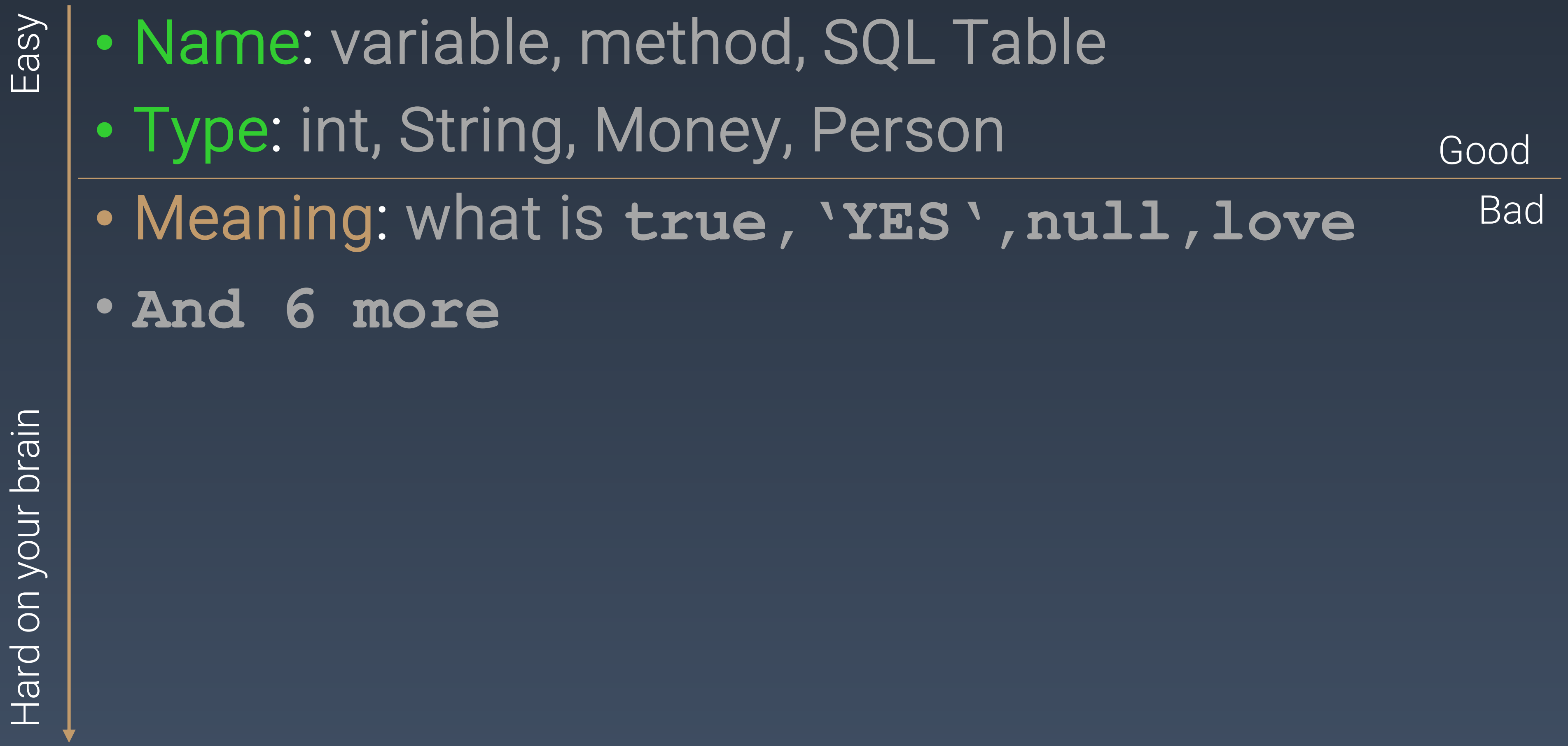


Both are forms of Connascence

> 2 elements A,B are connascent if there is at least 1 possible change to A requires a change to B in order to maintain overall correctness.

Connascence by Meilir Page-Jones, “Fundamentals of Object-Oriented Design in Uml”

Connascence of

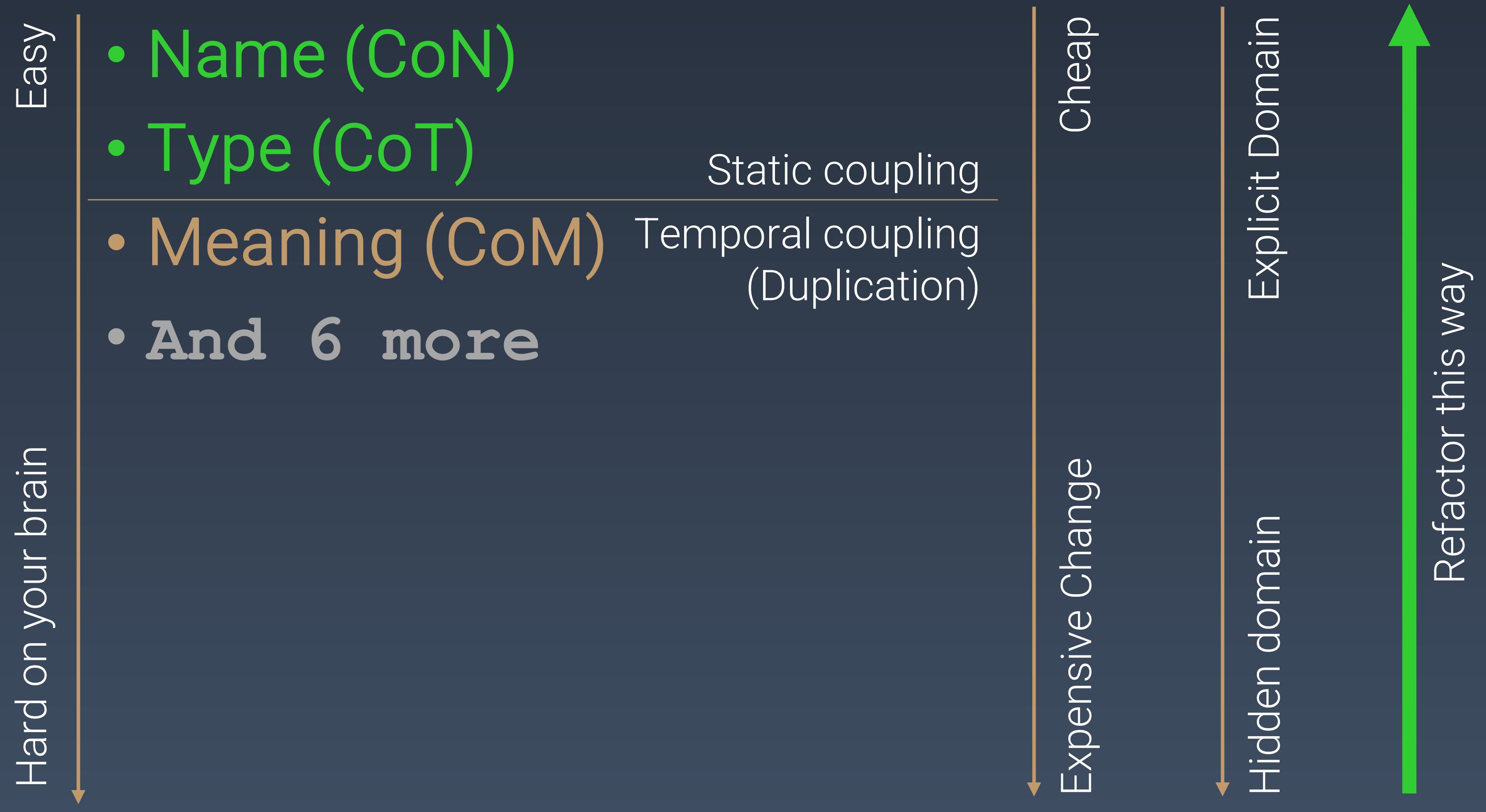


- Name: variable, method, SQL Table
- Type: int, String, Money, Person
- Meaning: what is `true,‘YES‘,null,love`
- `And 6 more`

Connascence: https://www.maibornwolff.de/en/know-how/connascence-rules-good-software-design/

Slides by richargh.de and

Connascence describes the type of coupling



Easy

Hard on your brain

- Name (CoN)
- Type (CoT)

Static coupling

- Meaning (CoM)

Temporal coupling
(Duplication)

- **And 6 more**

Cheap

Expensive Change

Explicit Domain

Hidden domain

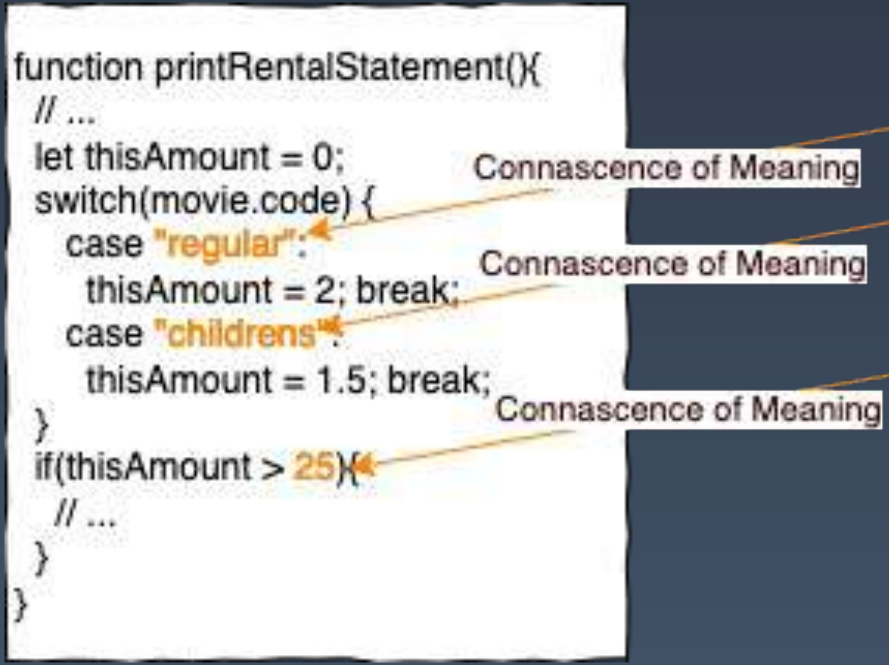
Refactor this way

Connascence: https://www.maibornwolff.de/en/know-how/connascence-rules-good-software-design/

Slides by richargh.de and

Connascence guides refactoring

CoM



```
function printRentalStatement(){
  // ...
  let thisAmount = 0;
  switch(movie.code) {
    case "regular":
      thisAmount = 2; break;
    case "childrens":
      thisAmount = 1.5; break;
  }
  if(thisAmount > 25){
    // ...
  }
}
```

CoN Alternative

```
// A)
enum MovieType { }
// B)
sealed interface Movie permits RegularMovie { }
// C)
interface Movie {
  int amount(){ … }
}
// D)
// appropriate solution is a team effort
```

```
// A)
static int OLD_PEOPLE_PENALTY = 25;
// B)
// appropriate solution is a team effort ☺
```

Connascence: https://www.maibornwolff.de/en/know-how/connascence-rules-good-software-design/

Slides by richargh.de and

Active Pattern: Temporal decoupling

- Find elements with lots of temporal coupling
- Identify type of Connascence that leads to the coupling

1. Try to move strongly connascent elements closer to each other
2. Try to refactor to a connascence of lower strength
3. If all else fails, “lock” the connascent elements → move them to a place that won’t receive changes

(2) Will usually uncover new domain concepts

Connascence: https://www.maibornwolff.de/en/know-how/connascence-rules-good-software-design/

Passive Pattern: Entity Ownership

Context

- Which part of the code “owns” an entity is highly related to which WRITEs to the table.
- Only one code part should “own” an entity so it can protect its contract (invariants, pre-/post conditions)

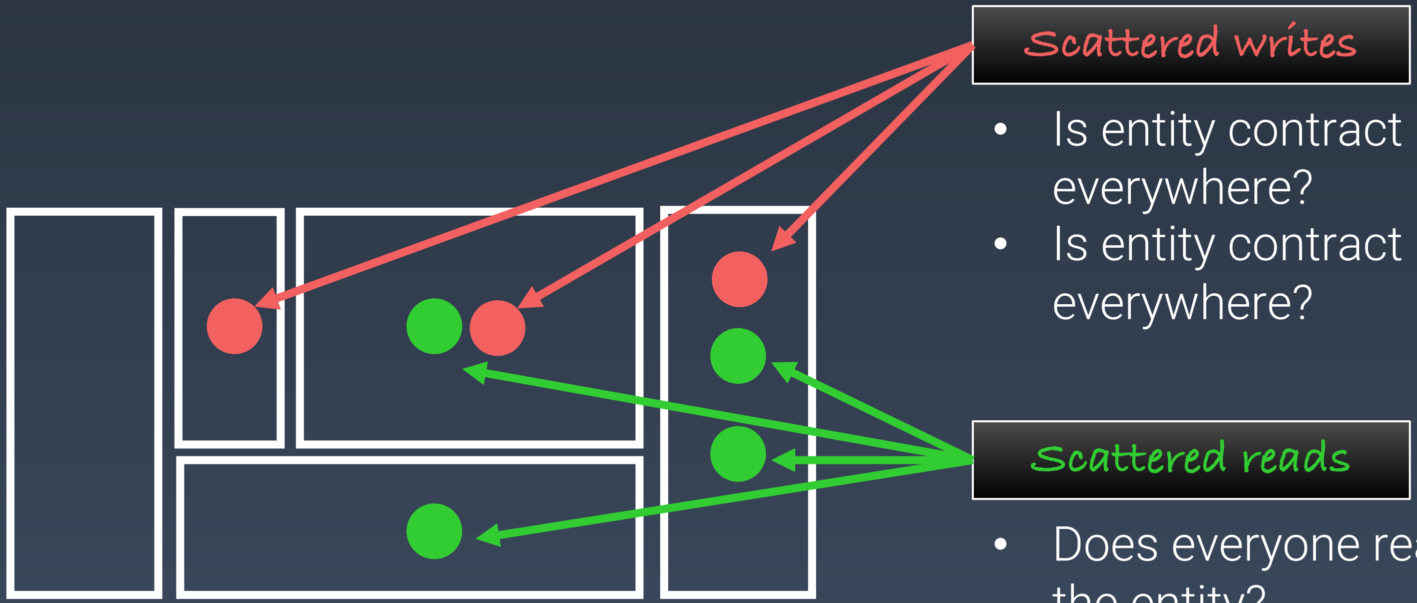
Approach

1. grep Reads: SELECT, JOIN
2. grep Writes: INSERT, UPDATE, DELETE
3. Table or plot for each entity which components reads an entity and which writes

[1] https://structure101.com/help/java/studio/Content/xs/tangle.html

Slides by richargh.de and

Passive Pattern: Entity Ownership



Scattered writes

- Is entity contract protected everywhere?
- Is entity contract in sync everywhere?

Scattered reads

- Does everyone really need the entity?
- Does everyone need the same fields?

Read ●

Write ●

Slides by richargh.de and

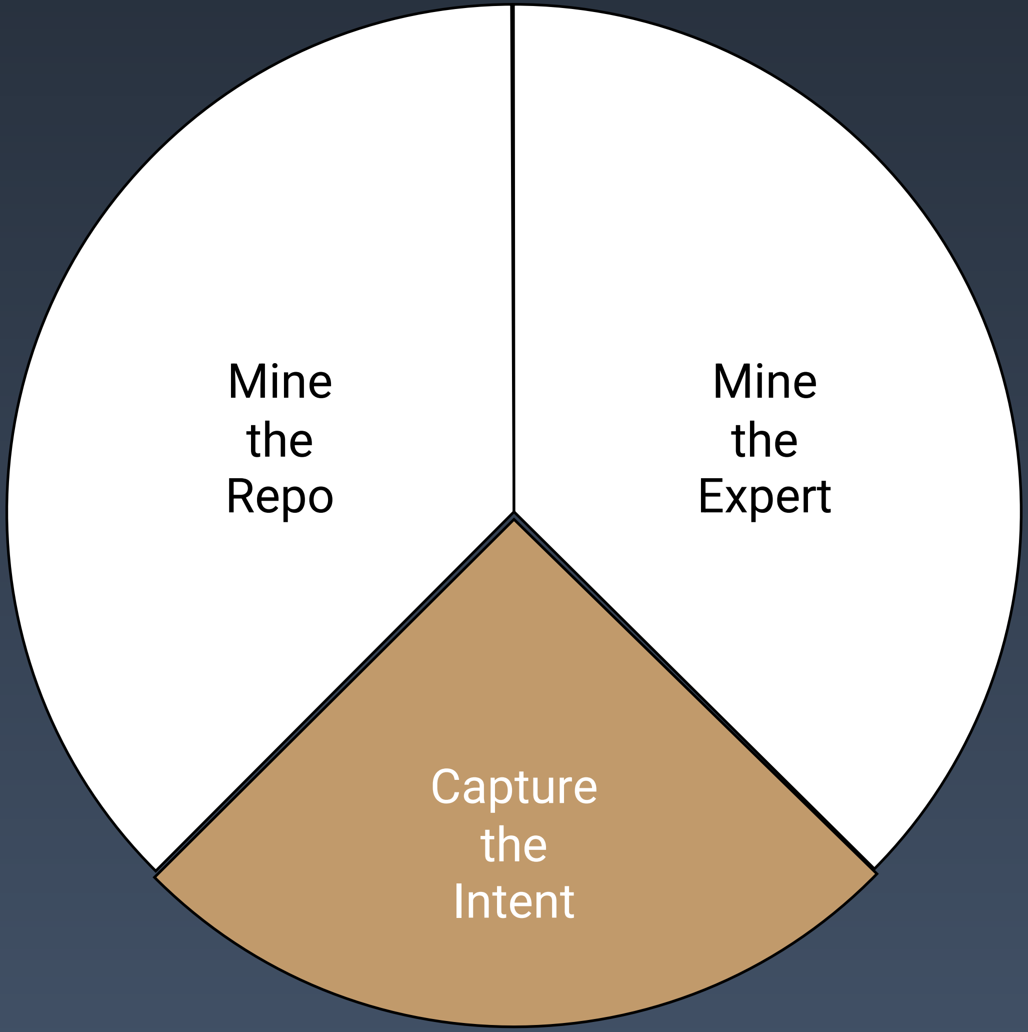
Active Pattern: Entity Ownership Bounds

Only one write location

- Don’t write the entity if you don’t own it
- If you have to write, delegate to owner
- The owner knows what a valid entity is

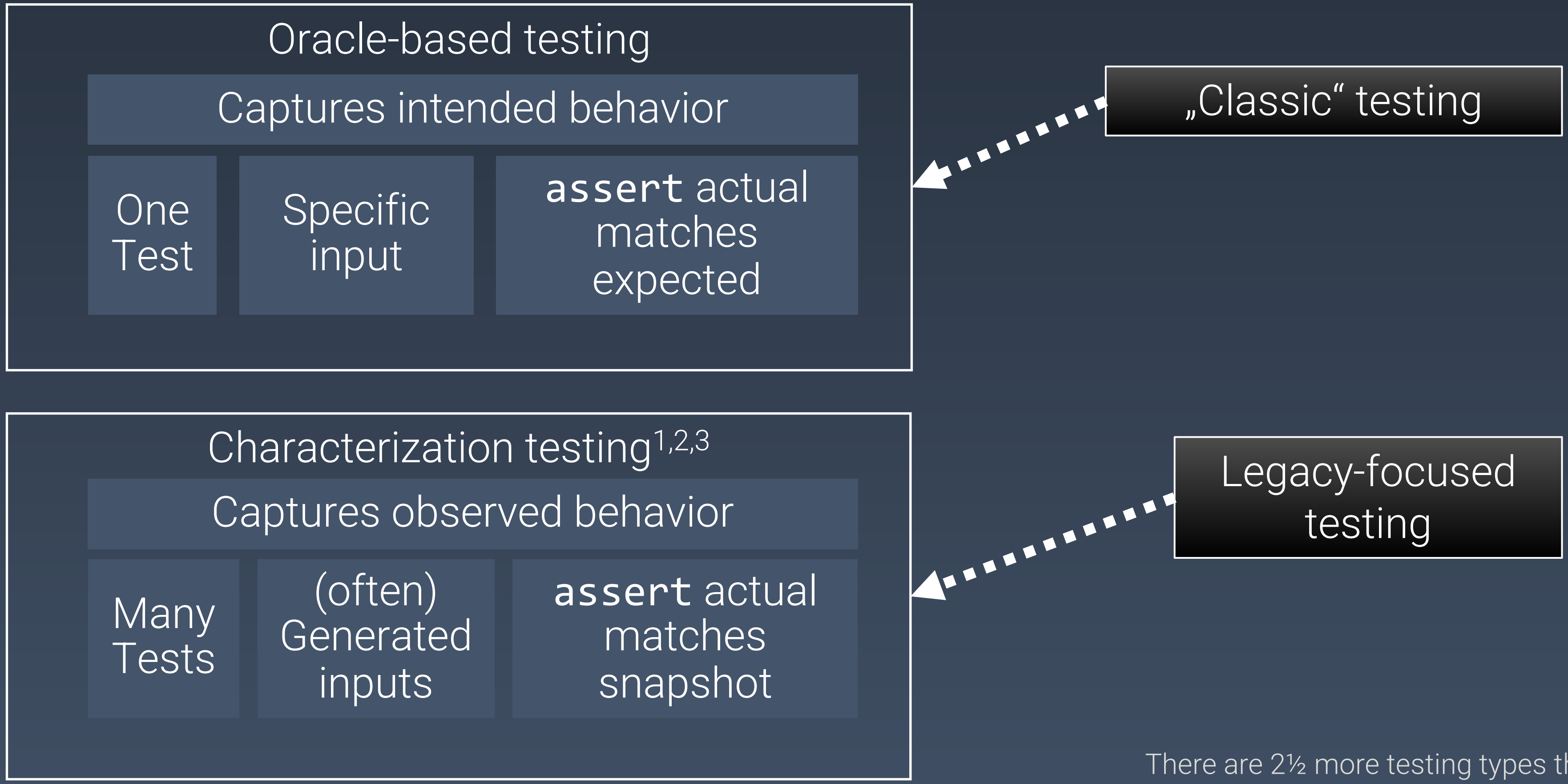
Read

- If scattered reads have little field overlap, consider splitting
- Get feedback on domain names of splits
- See if split has a different owner
- Keep it in sync via events



Slides by richargh.de and

Characterization tests capture a snapshot of the system



Oracle-based testing

Captures intended behavior

One Test

Specific input

`assert` actual matches expected

„Classic“ testing

Characterization testing[1,2,3]

Captures observed behavior

Many Tests

(often) Generated inputs

`assert` actual matches snapshot

Legacy-focused testing

There are 2½ more testing types that we won‘t get into here.

[1] see also „Golden Master“ https://en.wikipedia.org/wiki/Characterization_test

[2] see approval test framework https://approvaltests.com

[3] see verify framework https://github.com/VerifyTests/Verify/

Slides by richargh.de and

Active Pattern: *Inverse* Object Mother

Context

- Learn the minimal domain and document it as code.

Approach

1. Start application with empty database
2. Click through a UseCase
3. Analyse exceptions and errors
   - *„App needs at least an object A with this field“*
4. Expand Domain Modell with your finding
5. Create required state in the DB *with your model*
6. Document finding as characterization test
7. Repeat

Active Pattern: *Inverse* Object Mother

```
// Required state, temporarily in main
// we’ll move this to test soon
void main() {
  oneCharacterization();
}

// characterizations have no concept of why
void displaysListOfBooksOnStart(){
  // needs a user
  createUser();
  // needs at least one author
  var author = createAuthor();
  // needs at least one book
  var book = createBook(author);
}
```

Approach

1. Start application with empty database
2. Click through a UseCase
3. Analyse exceptions and errors
   - *„App needs at least an object A with this field“*
4. Expand Domain Modell with your finding
5. Create required state in the DB *with your model*
6. Document finding as characterization test
7. Repeat

Active Pattern: Outside-in Tests via Dsl

Context

- Keep tests structure-insensitive when you don’t know what your future structure will look like
- Be able to convert integration tests to unit tests after remodelling

Approach

- Use an abstraction for the test setup. Don‘t let tests directly …
  - create entities
  - put entities into db
- Stub out external systems
- Write tests outside-in

Active Pattern: Outside-in Tests via Dsl

Start with an integration test

<module>/renting.integration.test.ts

```
// create the low-level test-DSL
// small test, all secondary ports are now stubs or fakes, they never connect to the real world
const { a, secondaryPorts } = integrationTest().withoutState().buildDsl();

test(‘should be able to rent book’, () => {
  // GIVEN
  const book = a.book(); // I need a book, don’t care which
  const { user } = a.userBundle(it => it.hasPermission(“CAN_RENT_BOOK”); // a user, don’t care who

  await a.saveTo(secondaryPorts); // store book and user entities in repositories

  const testee = configureRentingComponent(floor); // configure dependencies of component
  // WHEN
  const result = testee.rentBook(book, user);
  // THEN
  expect(result.isRented).toBeTrue();
}
```

Active Pattern: Outside-in Tests via Dsl

Go unit with min. change, once all db logic is in domain

<module>/renting.unit.test.ts

```
// create the low-level test-DSL
// small test, all secondary ports are now stubs or fakes, they never connect to the real world
const { a, secondaryPorts } = unitTest().withoutState().buildDsl();

test(‘should be able to rent book’, () => {
  // GIVEN
  const book = a.book(); // I need a book, don’t care which
  const { user } = a.userBundle(it => it.hasPermission(“CAN_RENT_BOOK”); // a user, don’t care who

  await a.saveTo(secondaryPorts); // store book and user entities in repositories

  const testee = configureRentingComponent(floor); // configure dependencies of component
  // WHEN
  const result = testee.rentBook(book, user);
  // THEN
  expect(result.isRented).toBeTrue();
}
```

Always look for opportunities to document domain-understanding with oracle tests

You want to start with characterization tests as a safety-net

Active Pattern: Annotation-Whiskers

Context

- Types are great for modeling but they are quite invasive, especially when you are not yet sure of the end result
- Instead we'll use annotations to model assumptions and check them statically

Approach

0. Select an annotation-checker[1]
1. Use existing annotation or write your own
2. Run static analysis
3. Fix errors or make new assumption (go back to 2)
4. If useful, model as type

When you just want the null-checking, the options depend on your language http://richargh.de/talks/#fixing-the-billion-dollar-mistake

[1] the checker framework is good for this in java https://checkerframework.org/

Active Pattern: Annotation-Whiskers

As annotation[1]

```java
@m int meters = 5 * UnitsTools.m;
@s int seconds = 2 * UnitsTools.s;

// allowed
@mPERs int speed = meters / seconds;

// produces a compile-error
@m int foo = meters + seconds;
```

Vs as type (when you are sure)

```java
Meters meters = Meters.of(5);
Seconds seconds = Seconds.of(2);

// allowed
Speed speed = meters.per(seconds);

// produces a design-time error
var foo = meters.plus(seconds);
```

[1] uses the checker framework https://checkerframework.org/

Active Pattern: North-Star Architecture

Context

- At some point we want to write new code but still have so much confusing legacy
- We want to make sure we're all working toward the same goal
- We don't want to wait with feature development until everything is shiny

Approach

0. Define the north-star: the architecture you desire
1. Codify north-star[1,2,3]
2. Freeze existing violations
3. Continuously run static analysis
4. Fix new violations immediately

[1] Using ArchUnit for Java https://www.archunit.org/
[2] TSArch for TS/JS https://github.com/ts-arch/ts-arch
[3] or Dependency Cruiser for TS/JS https://github.com/sverweij/dependency-cruiser

Active Pattern: North-Star Architecture

```
@ArchTest
static final ArchRule no_classes_should_depend_on_service =
                        freeze(        // accept existing violations
                          noClasses()
                            .should()
                            .dependOnClassesThat()
                            .resideInAPackage("..service.."));

@ArchTest
static final ArchRule services_should_not_access_controllers =
                        noClasses() // only green without violations
                          .that().resideInAPackage("..service..")
                          .should().accessClassesThat().resideInAPackage("..controller..");
```

Example uses ArchUnit https://www.archunit.org/

Slides by richargh.de and

Communication Pattern: Quality Views[1]

Context

- Your domain re-discovery is going to take years
- It’s best to communicate where new features are easy and where they are hard

Approach

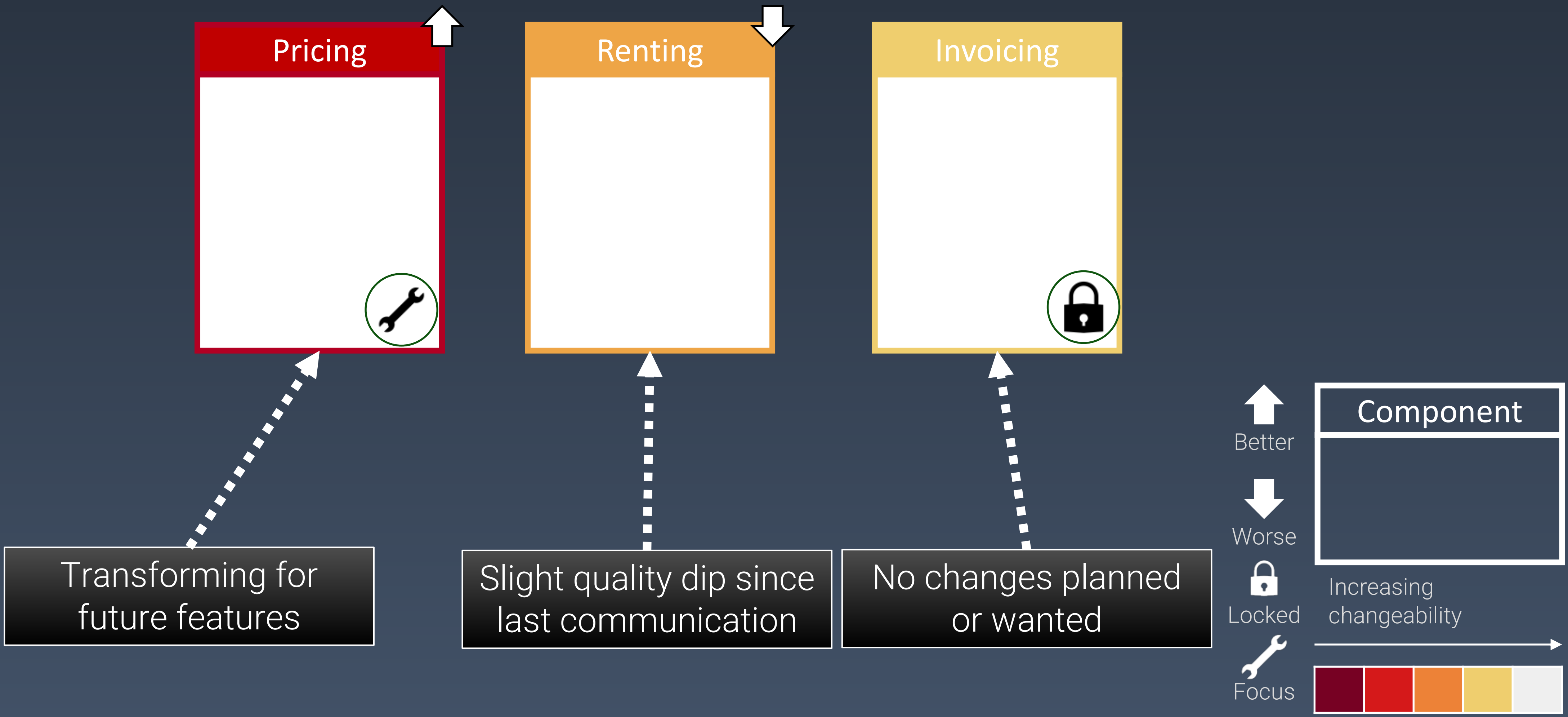
1. Draw your code structure
2. Colorize based on ease-of-change
   1. Explicit web-Api?
   2. Enforced component bounds?
   3. Presentation-Data-Domain Layering
   4. Categorization tests?
   5. Significant oracle tests?
   6. If you can‘t quantify, fist-of-five-it^{TM}
3. Mark what you work on right now

[1] Based-on https://blog.colinbreck.com/using-quality-views-to-communicate-software-quality-and-evolution/

Slides by richargh.de and

Communication Pattern: Quality Views

Highest-Level



Pricing

Renting

Invoicing

Transforming for future features

Slight quality dip since last communication

No changes planned or wanted

Better

Worse

Locked

Focus

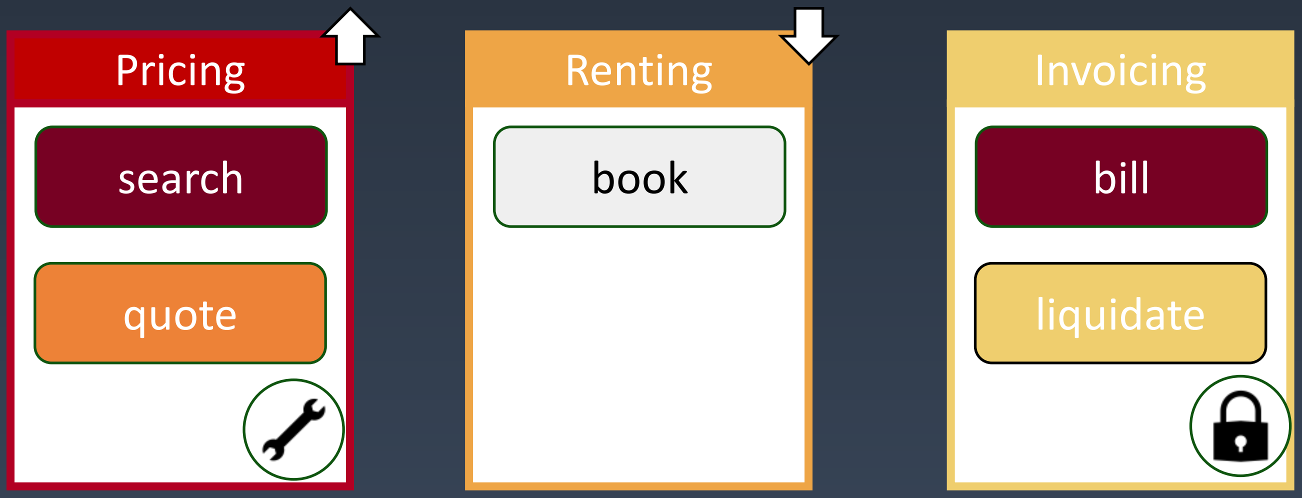
Component

Increasing changeability

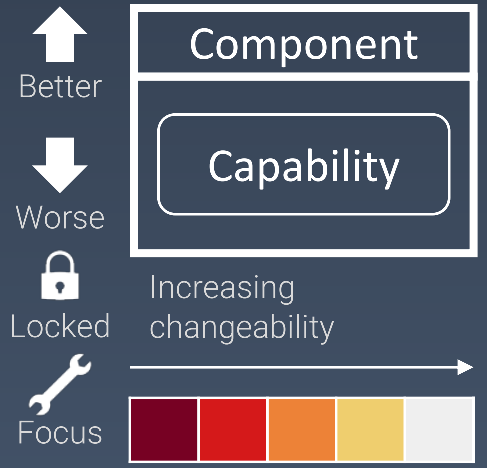
Slides by richargh.de and

Communication Pattern: Quality Views

Zoom in



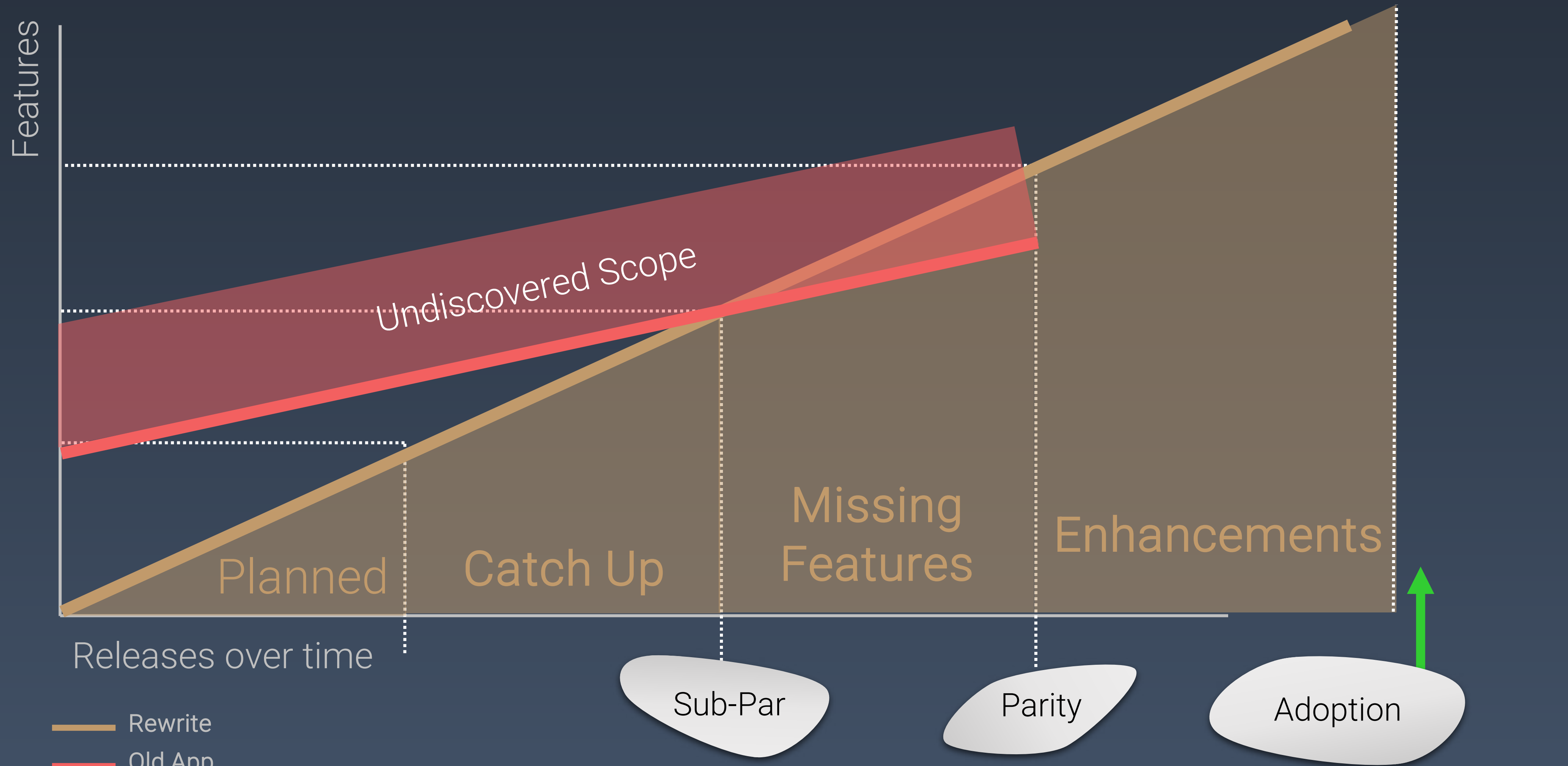
Feature-planning often requires more details



Slides by richargh.de and

Why all the effort to re-discover? We could just start a-new!

The True Cost of Feature-based Rewrites



Original Article by Doug Bradbury [The True Cost of Rewrites]

Slides by [richargh.de] and

Thanks

CodeCharta at https://maibornwolff.github.io/codecharta/

Code Tag Cloud at https://github.com/Richargh/code-tagcloud-py-sandbox
TestDsl Code at https://github.com/Richargh/testdsl

Ask me about these topics ☺

Richard Gross (he/him)

Archaeology + Audits

TestDSLs

Hypermedia

Cartography

Slides, Code, Videos

richargh.de/talks

@arghrich

Contact

Works for maibornwolff.de/

People. Code. Commitment.

DE TN ES

Backup

Slides by richargh.de and

The cost of the rewrite depends on your approach

Feature-based rewrite

- Goal = Feature + Feature + Feature
- Incrementally build feature after feature
- Release when all are done

Outcome-based rewrite

- Goal = achieve an outcome
- Write the minimal thing to achieve outcome
- Iterate

Generate and view a CodeCharta map

```
npm install -g codecharta-analysis
git clone git@github.com:MaibornWolff/codecharta.git

ccsh sonarimport https://sonarcloud.io -o petclinic.code.cc.json
ccsh gitlogparser repo-scan --repo-path=spring-petclinic/ -o petclinic.git.cc.json

ccsh merge petclinic.git.cc.json.gz petclinic.code.cc.json.gz -o petclinic.cc.json
```

→ Open petclinic.cc.json.gz in https://maibornwolff.github.io/codecharta/visualization/app/index.html

The official docs: https://maibornwolff.github.io/codecharta

Slides by richargh.de and

How do we refactor what we don‘t understand?

„Nopefactoring“
The No-thinking refactoring“

- Lift-up conditional
- Split to Classes

Advanced Testing & Refactoring Techniques

Emily Bache

@emilybache

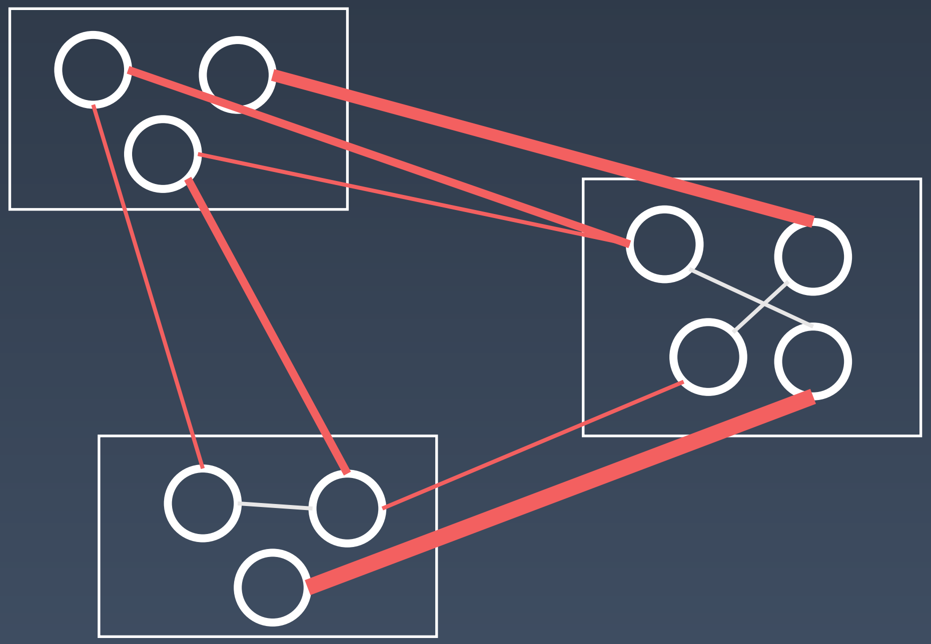
Cutting Code Quickly

Llewellyn Falco

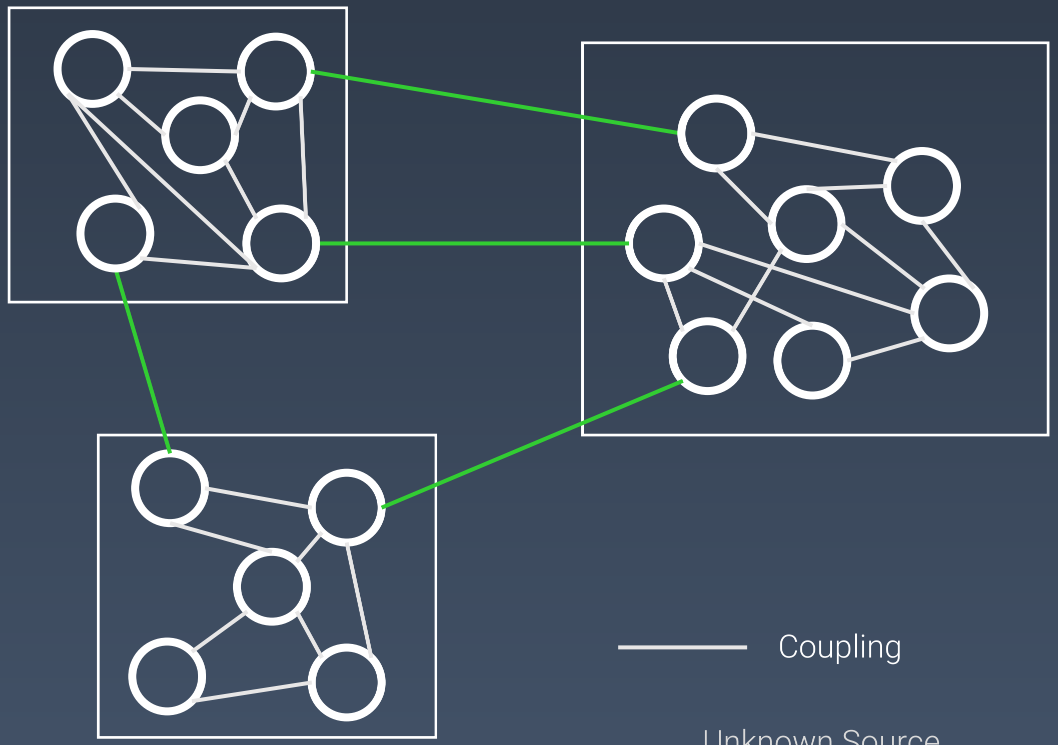
@LlewellynFalco

A brief coupling primer

High Coupling
Low Cohesion



Low Coupling
High Cohesion



Unknown Source

Slides by richargh.de and

Connascence Guides Refactoring

Refactor this way

Easy

- Name: variable, method, SQL Table
- Type: int, String, Money, Person

Good

Bad

- Meaning: what is `true,‘YES‘,null,love`
- Position: order of value
- Algorithm: encoding, SPA vs Server
- Execution (order): one before other
- Timing: doFoo() in 500ms | doBar() in 400ms

Really Bad

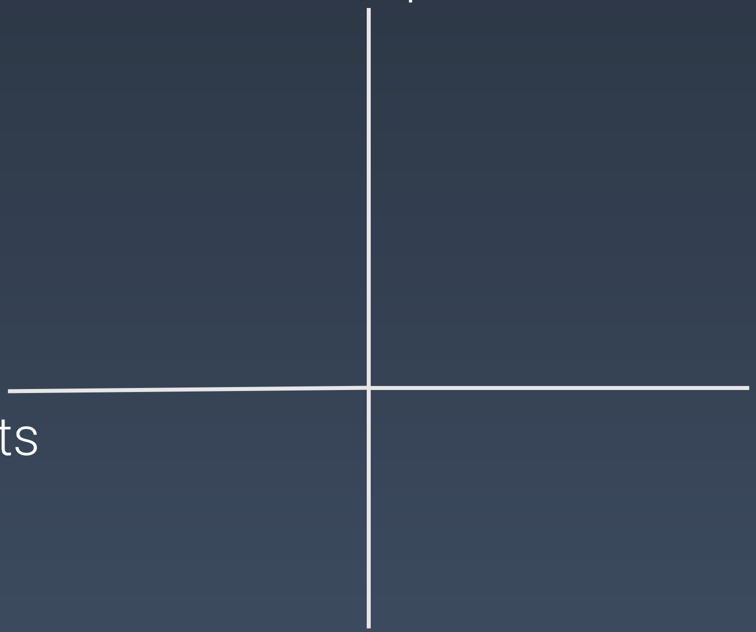
- Value: constraints on value, invariants
- Identity: reference same entity

Hard on your brain

Connascence: 🇩🇪 https://www.maibornwolff.de/know-how/connascence-regeln-fuer-gutes-software-design/

Slides by richargh.de and

4-axes of Connascence



Strength Level
How explicit

Degree
Number of Impacts

Locality
How close

Volatiliy
How much change

Slides by richargh.de and

The 4½ types of testing

Oracle-based testing

Captures intended behavior

One Test	Specific input	assert actual matches expected

Property-based testing[1]

Captures intended properties

One Test	Random inputs	assert one property of all outputs

Characterization testing[2,3]

Captures observed behavior

Many Tests	(often) Generated inputs	assert actual matches snapshot

Metamorphic testing[4]

Captures intended metamorphic relations

One Test	One source, Derived inputs	assert outputs keep relation to source

The remaining ½ is the mutation which you can use to test your tests. Mutate code, run test, see if enough tests break.

[1] see jqwik https://jqwik.net/

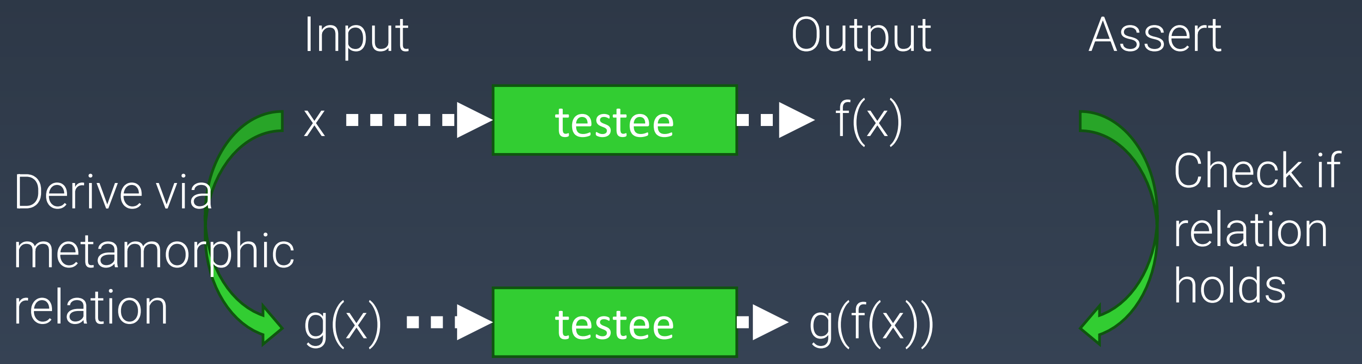
[2] see also „Golden Master“ https://en.wikipedia.org/wiki/Characterization_test

[3] Alternative name, „Approval Tests“ including test framework https://approvaltests.com

[4] see https://www.hillelwayne.com/post/metamorphic-testing/

Slides by richargh.de and

Metamorphic testing



Slides by richargh.de and