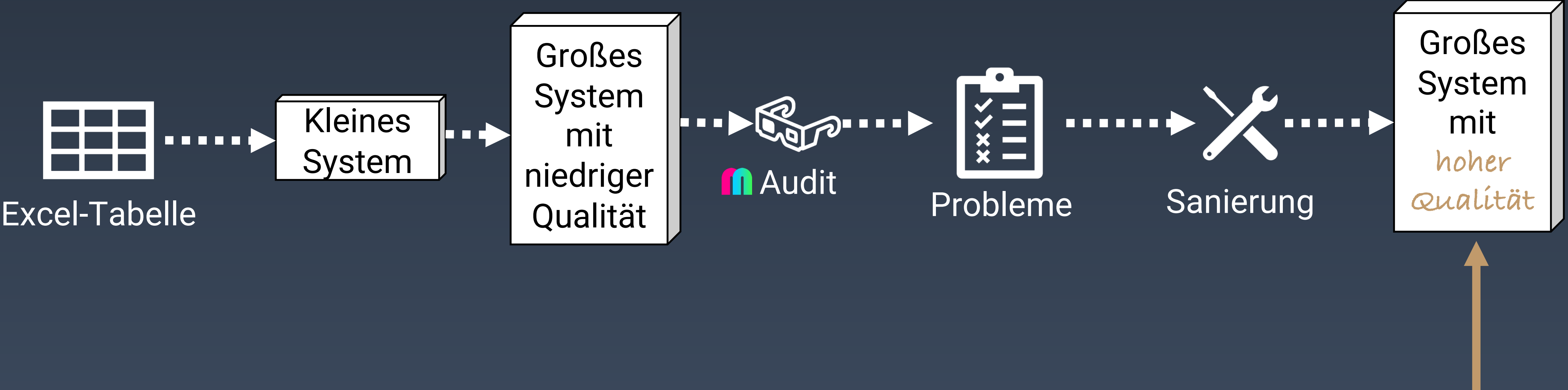
Bienenstock Architektur

29.09.23

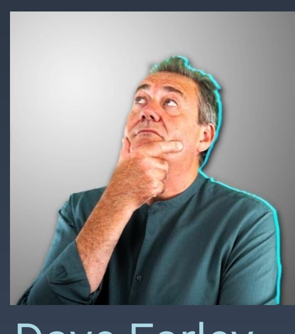
Richard Gross (he/him)

Der Lebenszyklus von Software



Welche *Eigenschaften* hat hoch-qualitative Software?

"The Quality of a System is Defined by Our *Ability to Change* it!



Dave Farley
@davefarley77

Dave Farley, author of the best-selling book "Continuous Delivery"

Slides by richargh.de

Warum?

Most Systems *mechanise a human or societal activity*, are embedded in and modify the real world they model and *must change* when the real world does[1]

[1] "Programs, Life Cycles, and Laws of Software Evolution" - M M Lehman

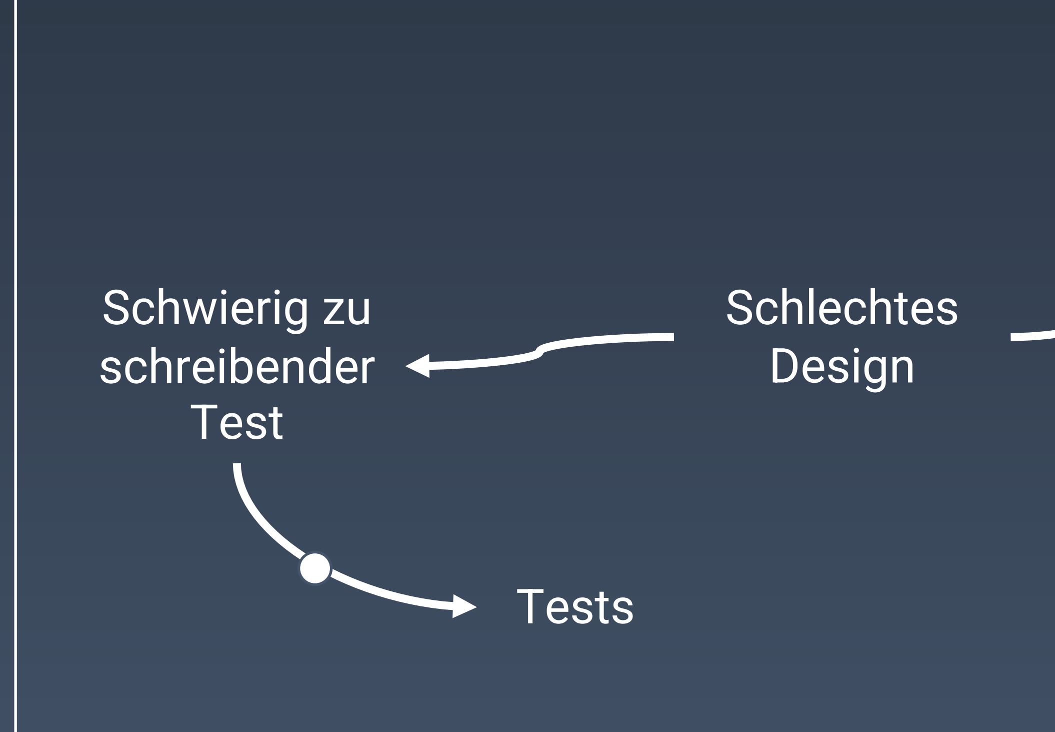
Der paradoxe Legacy Code

- Wir müssen den Code verändern
- Wir haben keine Tests (aka Legacy Code)
- Wir haben Angst den Code zu verändern
- Wir sollten testen
- Aber Tests an der falschen Stelle zementieren unser Design

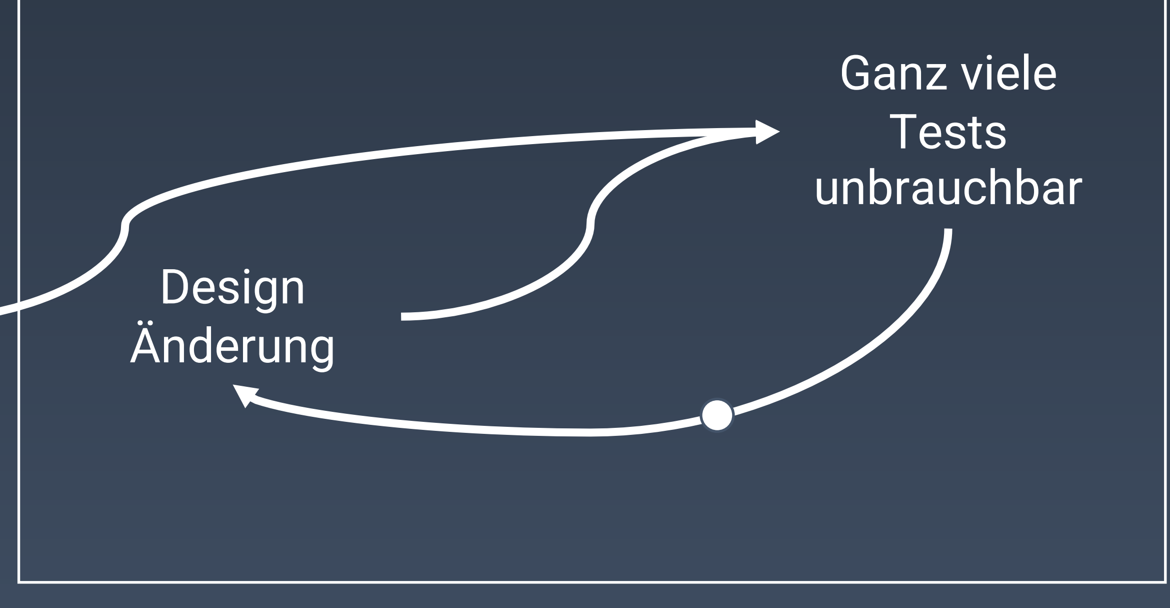
- → Entweder wir können verändern oder wir haben Tests

Synergie zwischen Testability und Design

Tests geben Design Feedback



Tests zementieren Design

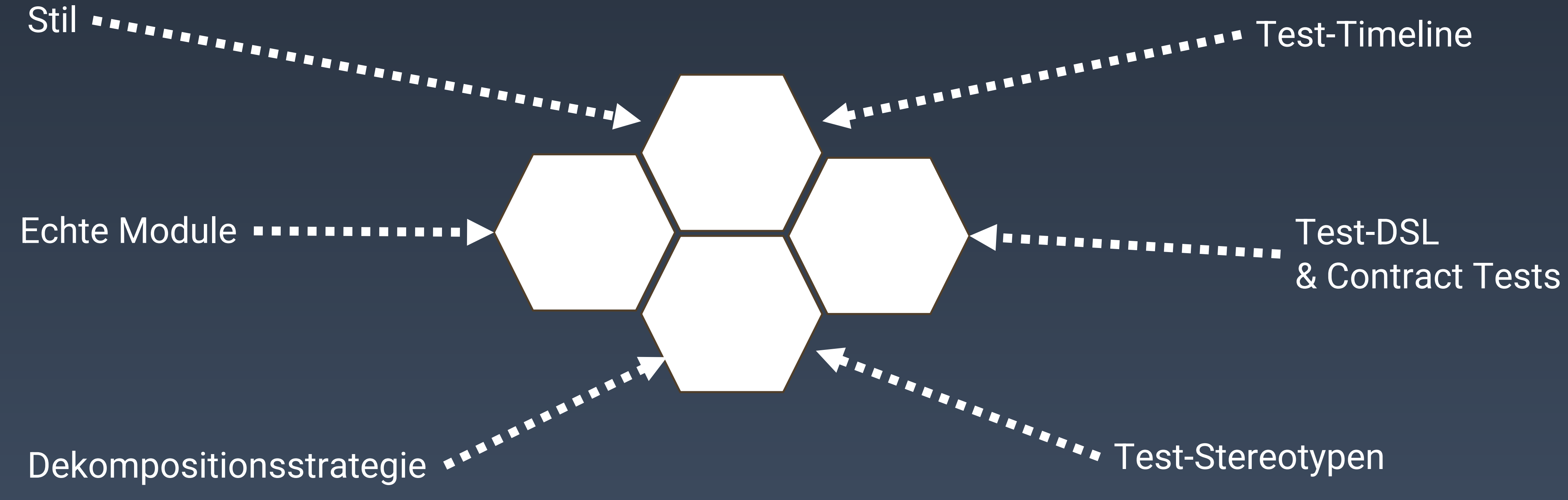


Siehe https://systemsthinking.dev

Slides by richargh.de

Architektur und Tests müssen ganzheitlich betrachtet werden

Die Bienenstock-Architektur umfasst



Die Bienenstock-Architektur umfasst

Stil

Echte Module

Dekompositionsstrategie

Test-Timeline

Test-DSL
& Contract Tests

Test-Stereotypen

Ein typisches System verarbeitet Daten*

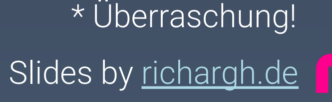
Daten rein

System

Daten raus

Datenfluss im System

* Überraschung!



Die Datenverarbeitung geschieht in mehreren Schichten [1]

Schicht für die wir *bezahlt* werden

Daten rein

Presentation

Domain

Data

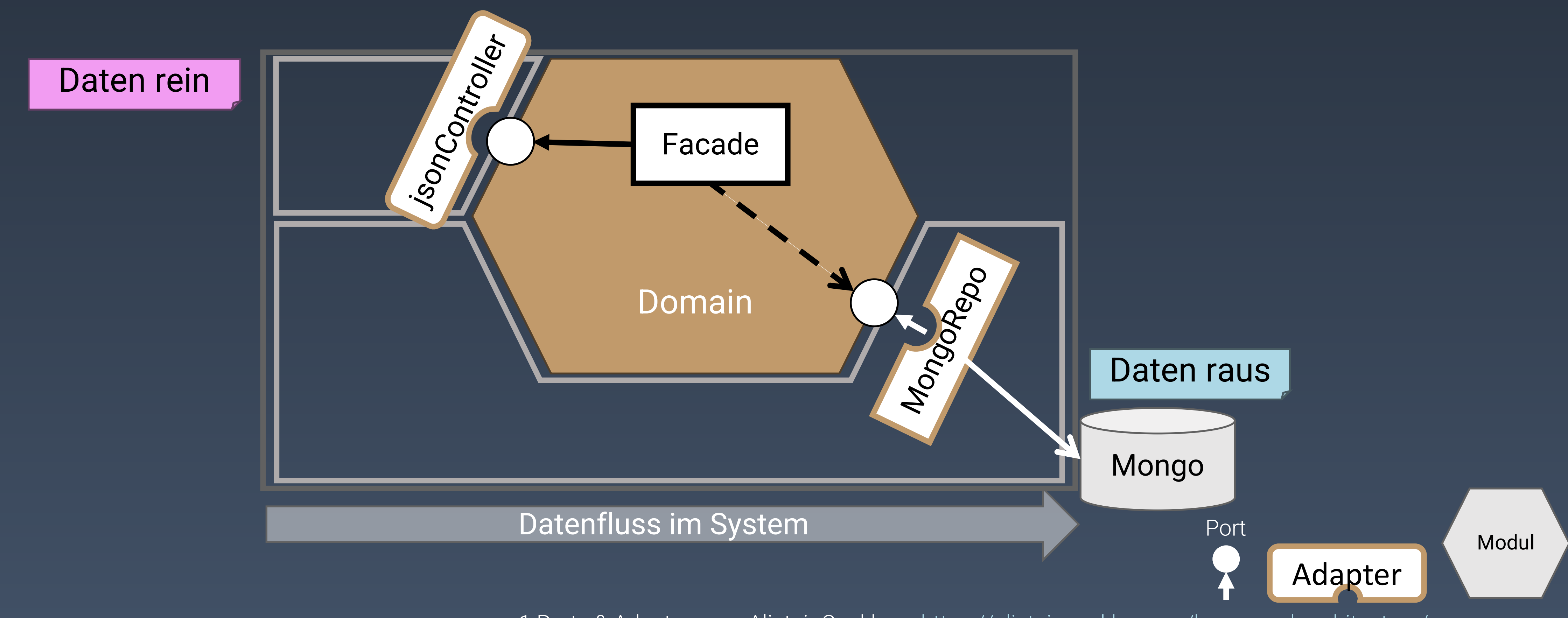
Daten raus

Notwendige Übel

Datenfluss im System

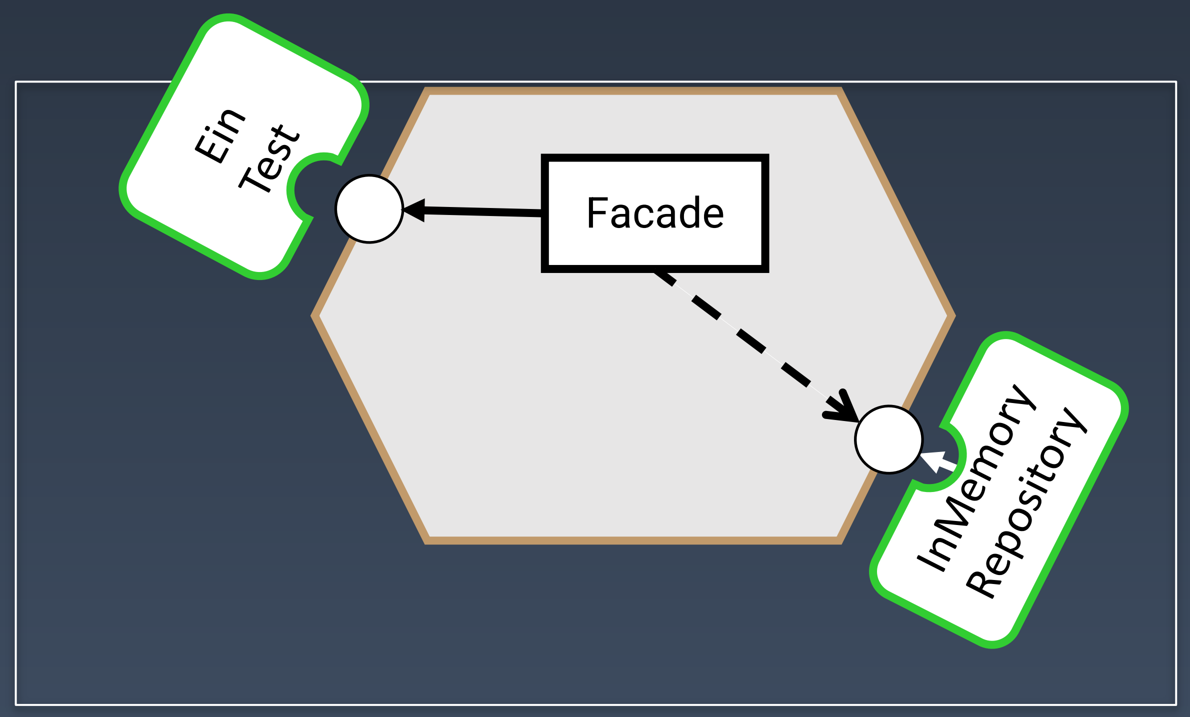
1 https://www.martinfowler.com/bliki/PresentationDomainDataLayering.html

Wir machen Domain Testbar mit Adaptern und Ports[1]



1 Ports & Adapters von Alistair Cockburn https://alistair.cockburn.us/hexagonal-architecture/

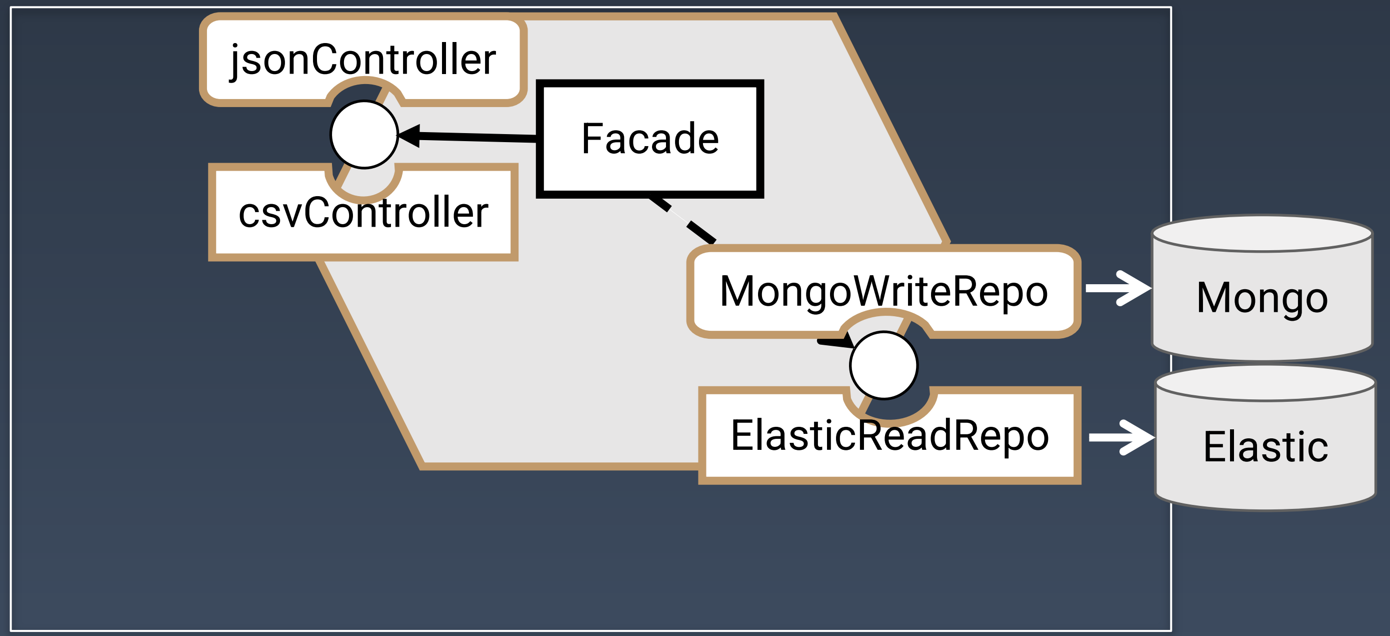
Dann testen wir von außen und nutzen Test Doubles



1 Ports & Adapters von Alistair Cockburn https://alistair.cockburn.us/hexagonal-architecture/

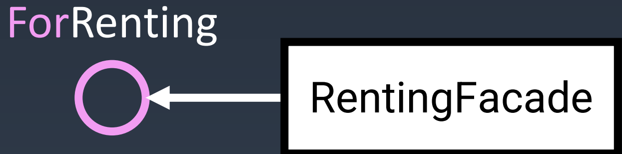
Slides by richargh.de

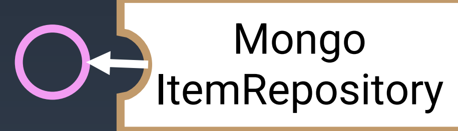
Dieser Stil lässt uns dann auf Änderungen reagieren[1]



1 Ports & Adapters von Alistair Cockburn https://alistair.cockburn.us/hexagonal-architecture/

Beispiel eines kleinen Moduls





<module>/renting.facade.ts

```
1.  export interface ForRenting { // Port
2.    rent(userId: UserId, itemId: ItemId);
3.  }
4.
5.
6.  class RentingFacade
                implements ForRenting {
7.
8.    #items: ForWritingItems; // Port
9.
10.   rent(userId: UserId, itemId: ItemId){
11.     // do stuff
12.   }
13. }
```

<module>/internal/items.ts

```
1.  export interface ForWritingItems { // Port
2.    add(item: Item);
3.  }
4.
5.
6.  class MongoItemRepository // Adapter
7.              extends MongoBaseRepository
                implements ForWritingItems {
8.    // do stuff
9.  }
```

Port
Name beginnt immer mit For*

Die Bienenstock-Architektur umfasst

Stil

Test-Timeline

Echte Module

Test-DSL
& Contract Tests

Dekompositionsstrategie

Test-Stereotypen

> 95% of the words [about software architecture] are spent extolling the benefits of "modularity" and that little, if anything, is said about *how to* achieve it.

Glenford J. Myers

Was macht ein Modul aus?

Expected Benefits of Modular Programming

The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication: (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

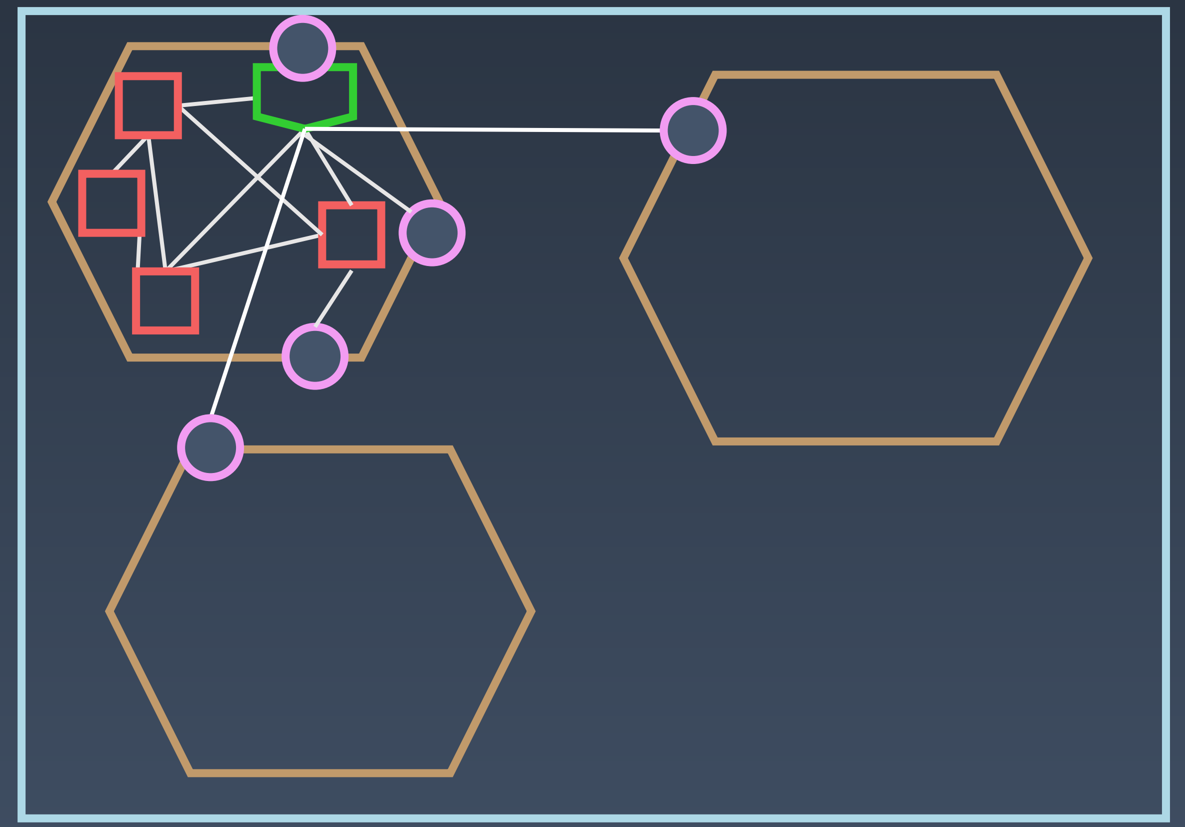
On the Criteria To Be Used in Decomposing Systems into Modules – DL Parnas

Maintainability is inversely proportional to the number of *exposed* classes, dependencies, microservices[1]

[1] Frei nach Simon Brown

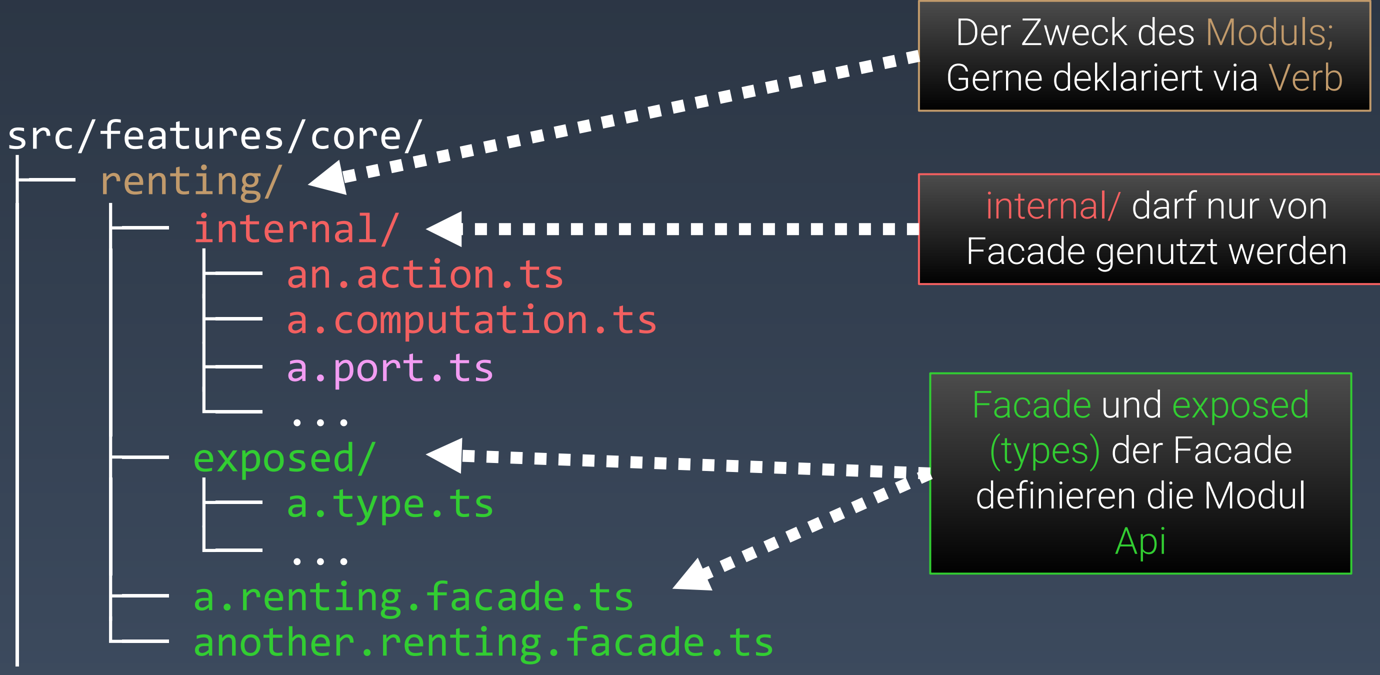
Wir brauchen *echte* Module

- Systeme bestehen aus Modulen
- Module
  - Sind Deep (viel Funktionalität hinter simpler Api)[1]
  - Verstecken Informationen, ihre Interna
  - Greifen nur auf die Api anderer Module zu, nie auf die Interna
  - Schützen sich vor der Außenwelt mit Ports
  - Bestehen aus immer kleineren Submodulen, bis runter zu Leafs 🍃
  - Führen Actions [2] aus (Nutzen die Außenwelt)
  - Oder führen Computations [2] aus (Pur, ohne Auswirkung auf Außenwelt)
  - Können als einzige ihre Daten *schreiben* 🪶



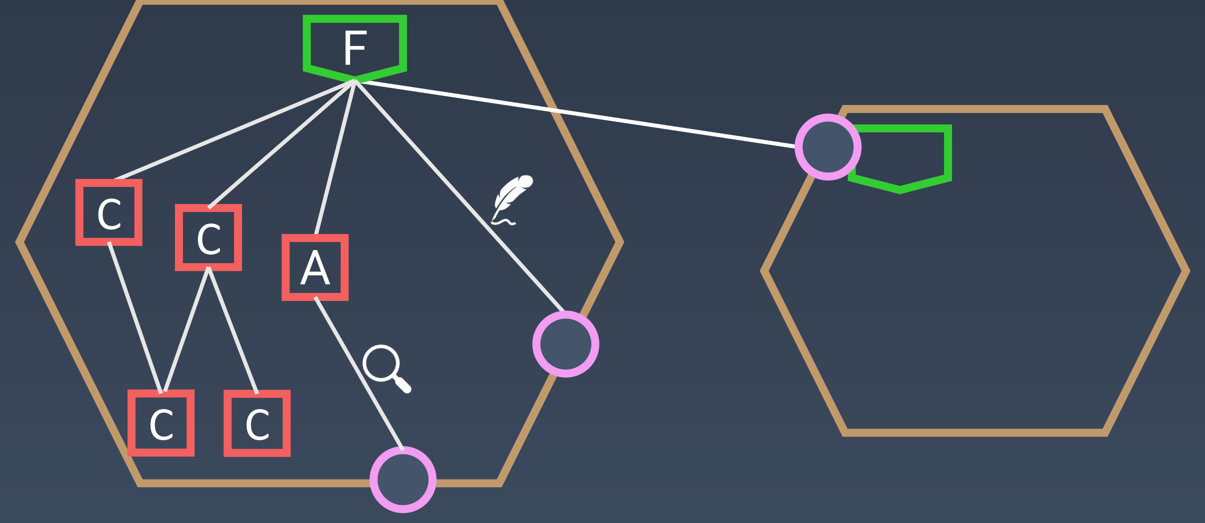
1 A Philosophy of Software Design
2 Grokking Simplicity

Per Konvention zeigen wir wofür ein Top-Level Modul ist, was die Api und was internal ist



Die Facade bietet Flexibilität die interne Modulstruktur zu verändern

- Facades sind die einzigen Einstiegspunkte von Modulen
  - Stellen die simple Api nach außen bereit und exposen dafür types
  - Composen dafür Actions [1] und Computations [1]
  - Delegieren nur und haben *keine Logik* (if,map,filter, …)
  - Können als einzige *schreibende* Operationen auf Ports etc. ausführen



1 Grokking Simplicity

Die Bienenstock-Architektur umfasst

Stil

Test-Timeline

Echte Module

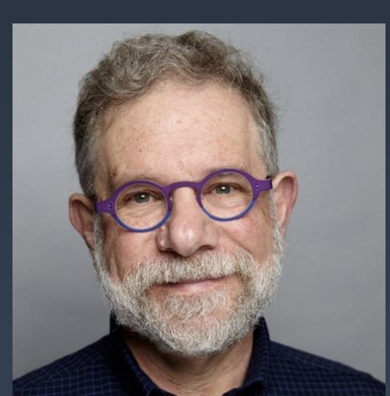
Test-DSL
& Contract-Tests

Dekompositionsstrategie

Test-Stereotypen

Welche Aufteilung bietet uns die max. Flexibilität auf Änderungen zu reagieren?

the key to incremental architecture is to build on a framework that can *accommodate change*, … that framework is the domain itself. By *modeling the domain*, you can more *easily handle changes* to the domain.[1]



Allen Holub

@allenholub

[1] https://twitter.com/allenholub/status/1099074412530196482

Wir fragen die Domain Experten und teilen fachlich

Daten rein

System

KernFunktion 1

KernFunktion 2

Kern-Entität 1

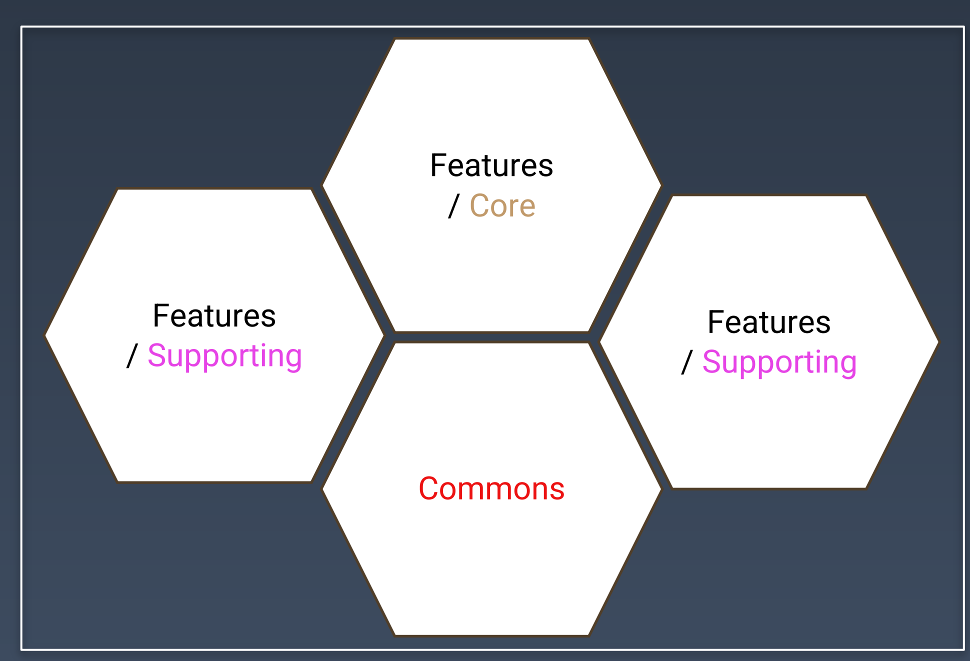
Kern-Entität 2

Daten raus

Datenfluss im System

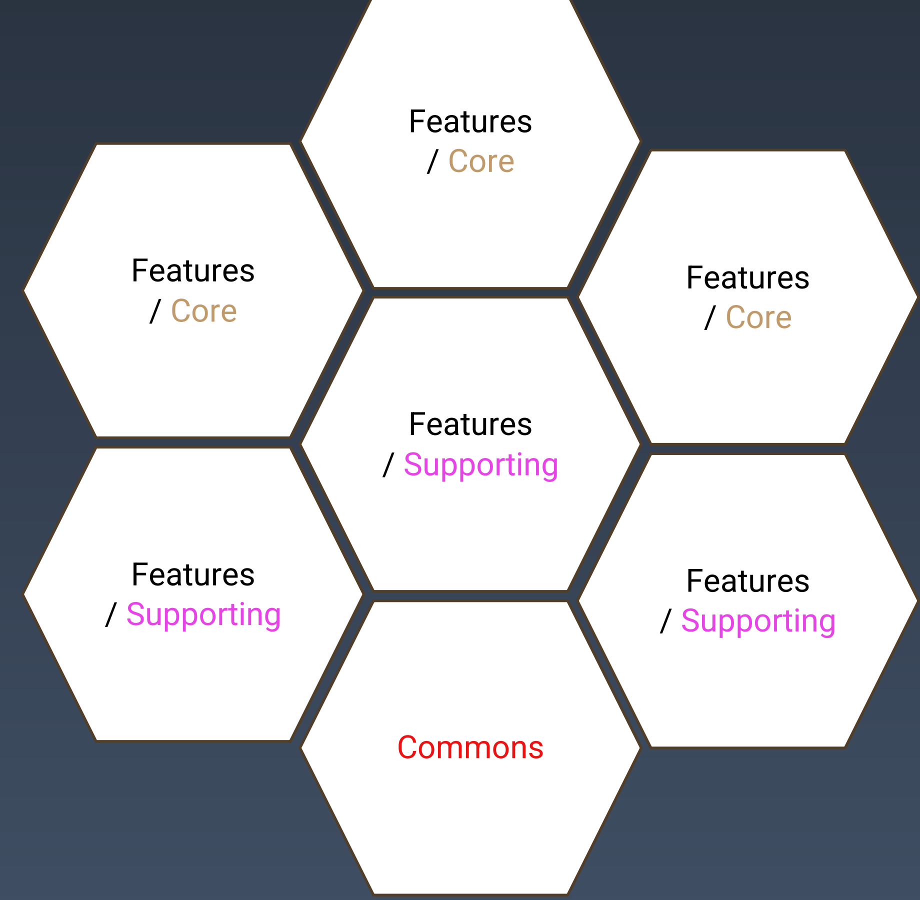
Domain Experten liefern die Grundlage für die Dekomposition

- src/features/core/<modul>
  - Die Kernmodule sind der Grund warum das System existiert
- src/features/supporting/<modul>
  - Unterstützende Daten für Kern
  - Kommen von anderen Systemen
- src/commons/<modul>
  - Technischer Code ohne *Business* Logik
  - Unsere eigene commons-library
  - z.B. repository, permissions, base-types

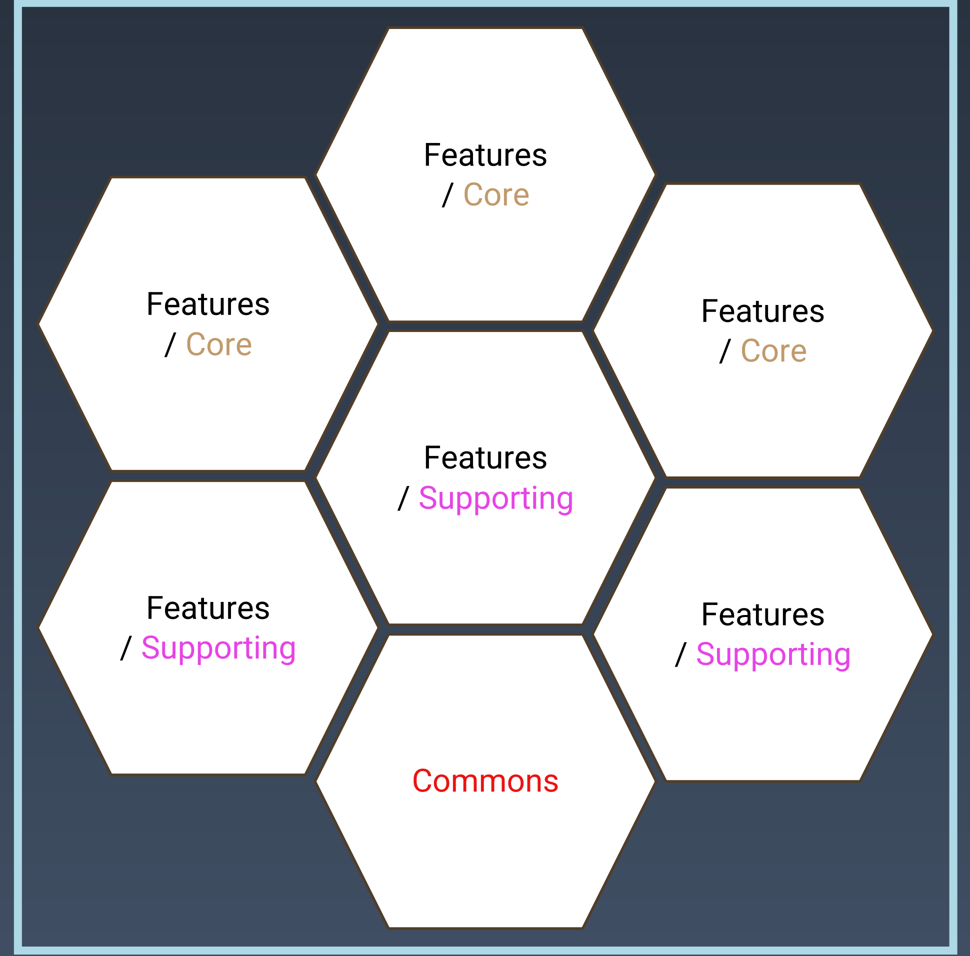


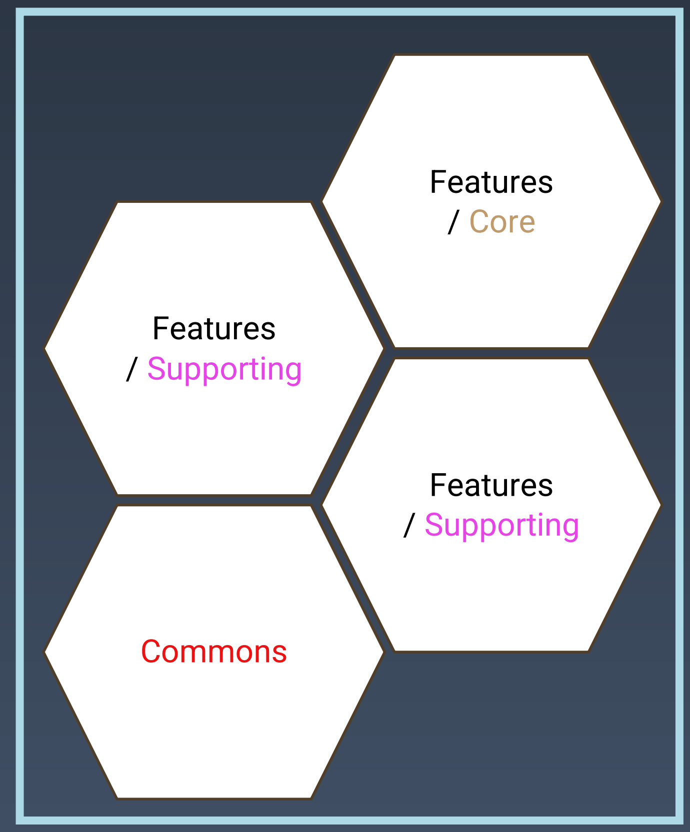
1 Package by Feature
2 DDD Context Classification at Module Level

Unser Bienenstock wächst



Bis wir ihn dann in Systeme splitten[1]





1 Situativ, Monolith aus echten Modulen sind sehr angenehm. Das Aufsplitten will gut begründet sein.

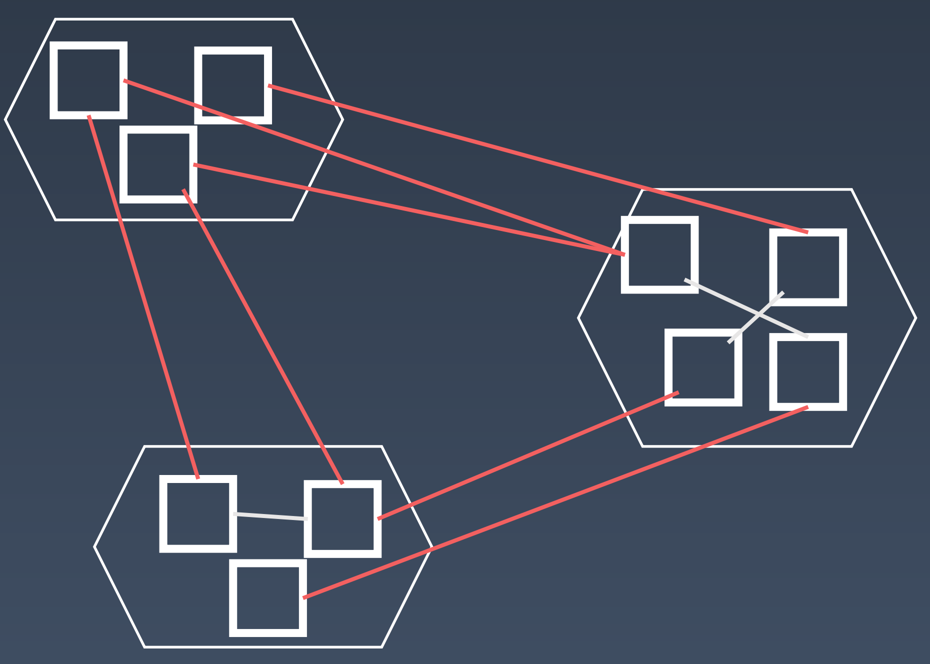
Group together what fires together

1 Feature, 1 Commit, 1 Modul

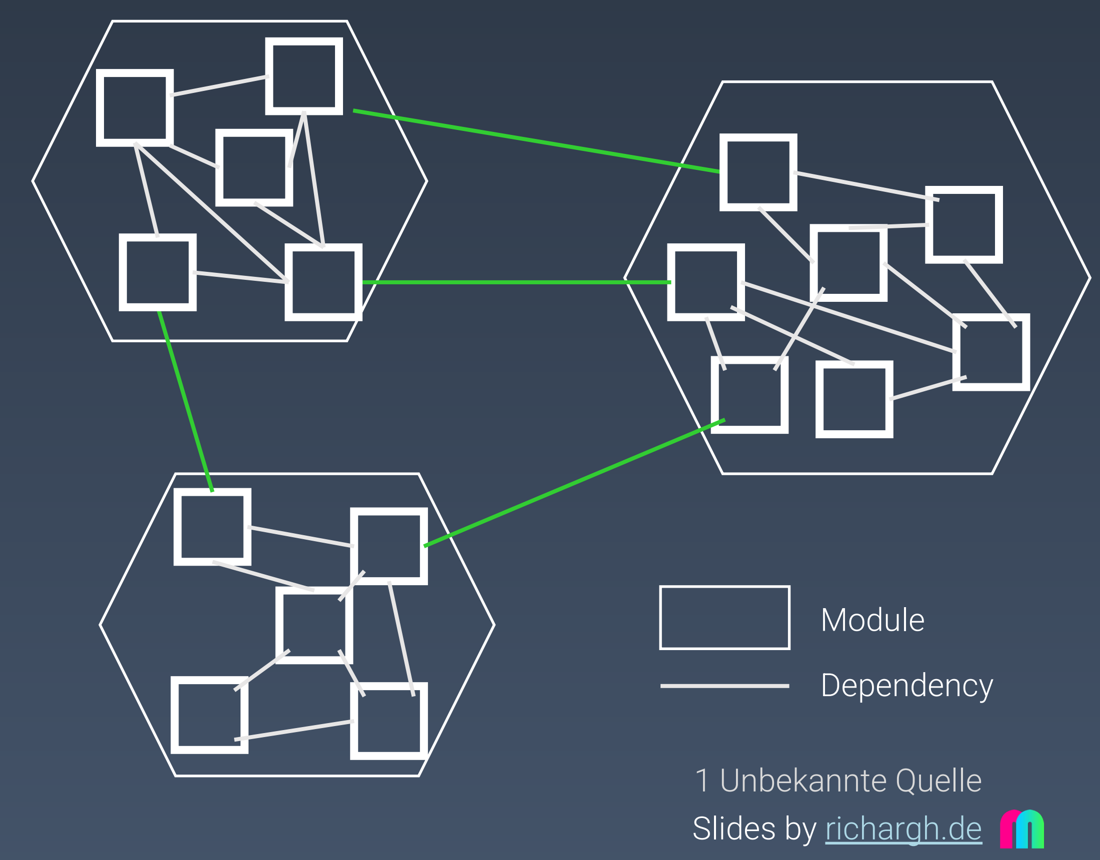
Frei nach der Hebbschen Lernregel

Wir gewinnen je weniger wir preisgeben

High Coupling
Low Cohesion[1]



Low Coupling
High Cohesion[1]



1 Unbekannte Quelle

Slides by richargh.de

Die Bienenstock-Architektur umfasst

Stil

Test-Timeline

Echte Module

Test-DSL
& Contract-Tests

Dekompositionsstrategie

Test-Stereotypen

Was macht Tests teuer?

Gutes/Schlechtes Design
Kompliziertes Test-Harness
Zementiertes Design
Code-Änderungen
Lesbarkeit, gerade bei roten Tests
Ausführungs-zeit
Schreibzeit
Wartungszeit
Feedbackzeit
Testkosten

Wie teuer ist das Feedback?

Ausführungszeit
Pro Test[1]

3,9 s/Test

200 ms/Test

50 ms/Test

0

Small[2,4]

Medium[2]

Large[3]

1 Echte Werte aus einem ts-node + Jest Projekt - 2 Tests parallelisiert - 3 Paralellisiert, da geht aber noch was
4 könnte mit anderem Setup wahrscheinlich auf 10ms gedrückt werden

Test Sizes[1], nicht Bike-Shedding

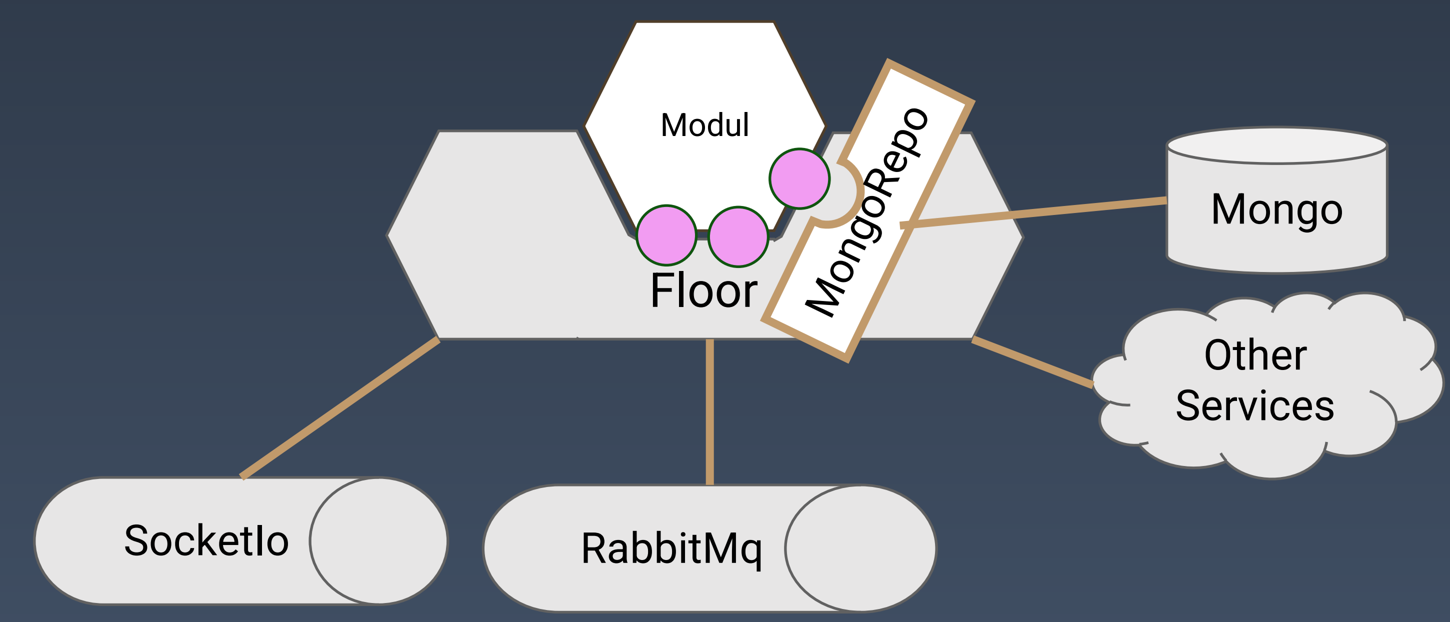
Außenwelt (Network access, Database, File system access, Use external system)

Feature	Small	Medium	Large
Network access	No	Localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external system	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

1 Google Test Sizes https://testing.googleblog.com/2010/12/test-sizes.html

Slides by richargh.de

Die Außenwelt, auf der ein ganzes System steht, nennen wir *Floor*



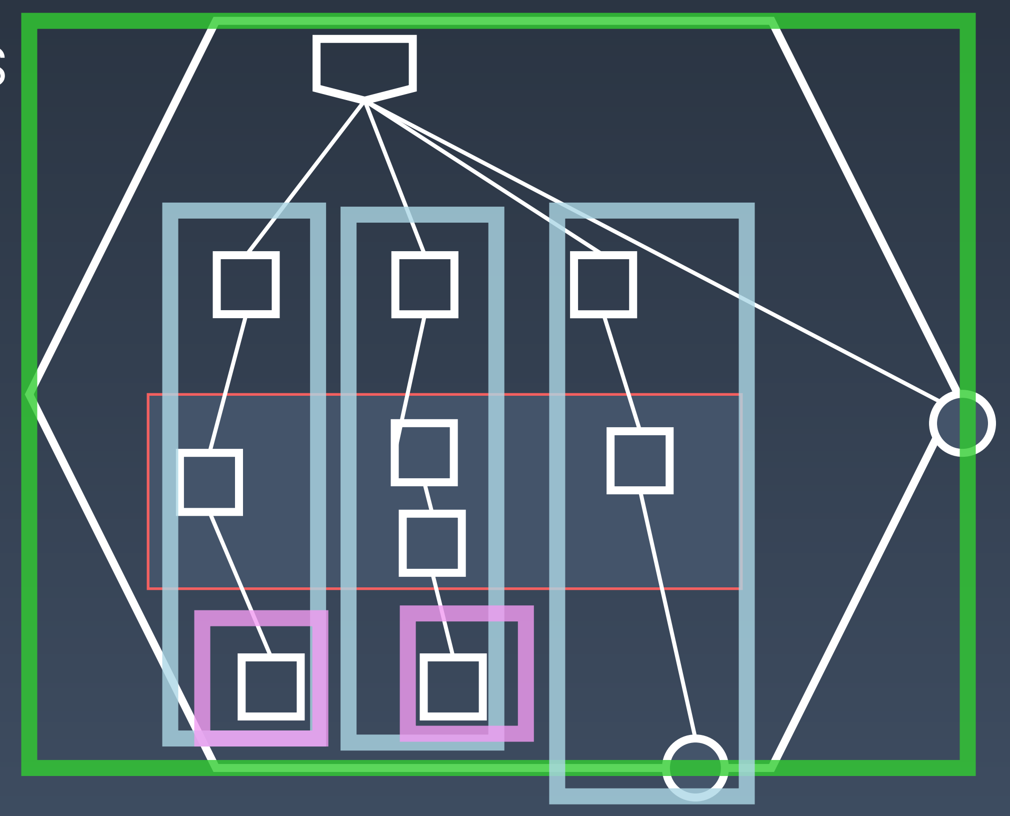
> Write tests where you want your *module boundaries* to be. Writing them any other place just shifts the boundary — and your modularization.

[Michael Feathers](https://twitter.com/mfeathers)
[@mfeathers](https://twitter.com/mfeathers)

https://twitter.com/mfeathers/status/1585720577477615616

Tests auf der richtigen Ebene: Flexibilität in der internen Modulstruktur

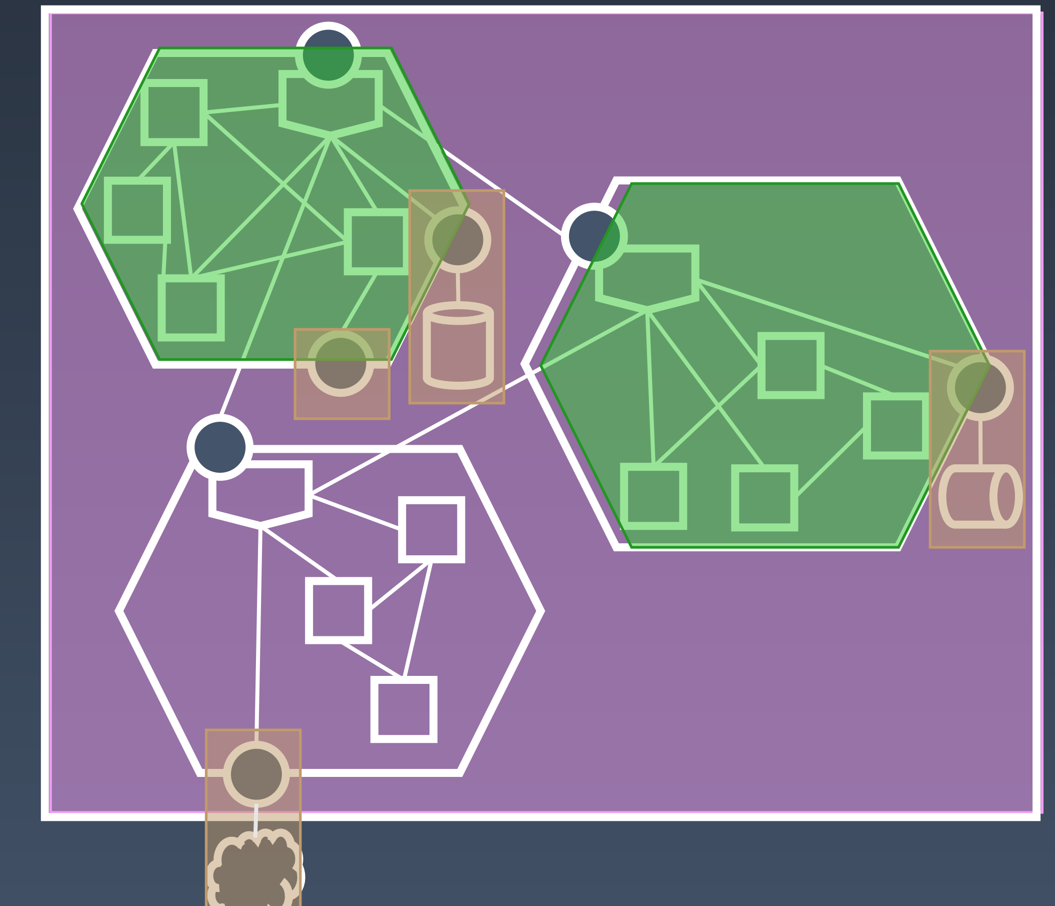
- Small (Scope) Tests laufen in-process
- Ports implementiert von Test Doubles
- Wir testen viel outside-in
- Es gibt
  - Facade Tests
  - Top-Module Action & Calculation Tests
  - Leaf Tests
  - Keine Tests im “Murky Middle”, da Struktur-zementierend



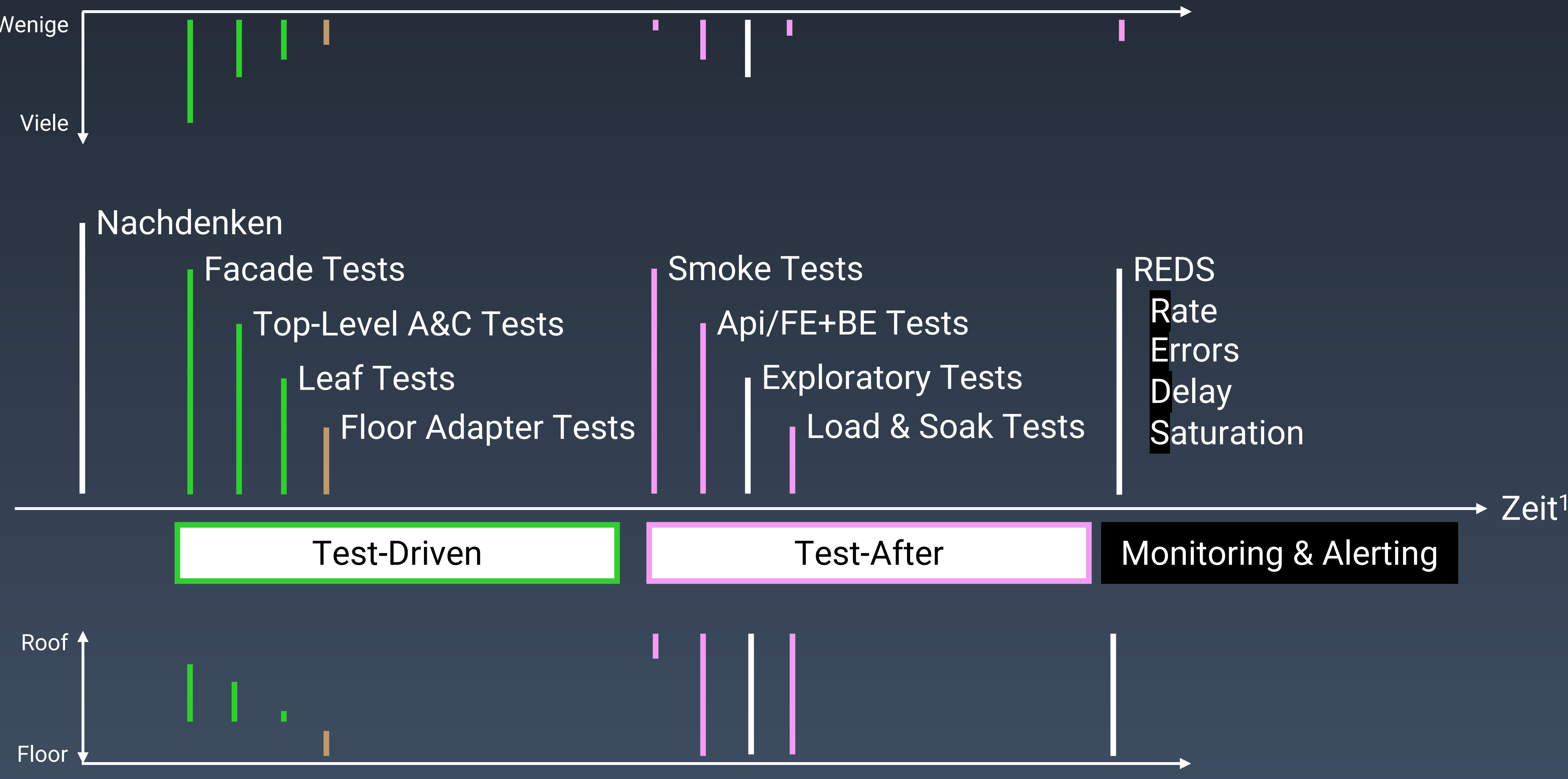
1 Grokking Simplicity

Das Ziel der Tests ist Abhängig vom Scope[1]

- Small Tests pro Modul
  - (Business) Logik funktioniert
  - Kleiner Scope, spezifische Assertions
- Medium (Contract) Tests für Adapter
  - Floor Adapter funktioniert
- Large Tests für Api, FE+BE
  - Für den Anwender funktioniert alles
  - Performant machen mit Test Data Aliasing oder Mandantenfähigkeit
  - Grober Scope, grobe Assertions



1 Google Test Sizes https://testing.googleblog.com/2010/12/test-sizes.html



1 Kleine Differenzen zu, aber im Kern => Urs Enzler – Die Agile Testpyramide https://www.youtube.com/watch?v=-35GpOJnjmM

Die Bienenstock-Architektur umfasst

Stil

Test-Timeline

Echte Module

Test-DSL
& Contract Tests

Dekompositionsstrategie

Test-Stereotypen

Woran erkennt man schlechte Tests?

Test-Smells

- *Langsamer* Test
  - Länger Warten → Weniger ausgeführte Tests → Weniger getestete Fachlichkeit → Weniger Refactoring
- *Langer* Test
  - Viel zu lesen, viele versteckte Fehler
- *Irrelevante* Details
  - Muss es ein bestimmtes Buch sein oder ist jedes möglich?
  - Muss dieses Feld den Wert haben, damit der Test erfolg hat?
- *Struktur-zementierende* Tests
  - Domain Entitäten von Hand erstellt, neue Felder nicht hinzufügbar
  - In vielen Tests wird redundant die gleiche Methode gemockt

1 http://xunitpatterns.com/Test%20Smells.html

2 Art of Unit Tests – 3rd Edition

Langer Test voller irrelevanter Details

```
<module>/a.small.test.ts
1.
2.
3.
4.
5. test('should be able to rent book', () => {
6.   // GIVEN
7.   const book = new Item(id: "1", "Refactoring", "Martin Fowler");
8.   const permission = new Permission(id: "1", "CAN_RENT_BOOK");
9.   const role = new Role(id: "1", "Renter", [permission.id]);
10.  const user = new User(id: "1", "Alex Mack", role.id);
11.
12.  const items = new InMemoryItemsDouble(item);
13.  const permissions = new InMemoryPermissionsDouble(permission);
14.  const roles = new InMemoryRoleDouble(role);
15.  const users = new InMemoryUsersDouble(user);
16.
17.  const testee = new RentingFacade(new ClockDouble(), items, permissions, roles, users, …);
18.  // WHEN
19.  const result = testee.rentBook(book, user);
20.  // THEN
21.  expect(result.isRented).toBeTrue();
22. }
```

Welche der Parameter sind tatsächlich relevant?

Weitere unnötige Details

Duplizierte initialisierung zementiert Struktur

Versteckt hier unten ist was wir eigentlich testen

Mit einer TestDsl zu verständlichen und wartbaren Tests

<module>/a.small.test.ts

```
1. // create the low-level test-DSL
2. // small test, all floor ports are now stubs or fakes, they never connect to the real world
3. const { a, floor } = smallTest().withoutState().buildDsl();
4.
5. test('should be able to rent book', () => {
6.   // GIVEN
7.   const book = a.book(); // I need a book, don't care which
8.   const { user } = a.userBundle(it => it.hasPermission("CAN_RENT_BOOK"); // a user, don't care who
9.
10.   await a.saveTo(floor); // store book and user entities in floor
11.
12.   const testee = rentingFacadeWith(floor); // configure renting facade to use floor
13.   // WHEN
14.   const result = testee.rentBook(book, user);
15.   // THEN
16.   expect(result.isRented).toBeTrue();
17.}
```

Mit einer TestDsl zu verständlichen und wartbaren Tests

<module>/a.medium.test.ts

```
1. // create the low-level test-DSL
2. // most floor ports now use real world
3. const { a, floor } = mediumTest().withoutState().buildDsl();
4.
5. test(‘should be able to rent book’, () => {
6.   // GIVEN
7.   const book = a.book(); // I need a book, don’t care which
8.   const { user } = a.userBundle(it => it.hasPermission(“CAN_RENT_BOOK”); // a user, don’t care who
9.
10.   await a.saveTo(floor); // store book and user entities in floor
11.
12.   const testee = rentingFacadeWith(floor); // configure renting facade to use floor
13.   // WHEN
14.   const result = testee.rentBook(book, user);
15.   // THEN
16.   expect(result.isRented).toBeTrue();
17.}
```

Mit der DSL vermeiden wir *type-zementierende*, weil duplizierte, *Initialisierung*

Test 1

Test 2

Test …

Test n

new Item("1", "Abc", …)

new Item("2", "Bcd", …)

new Item("3", "Cde", …)

new Item("n", "Xyz", …)

class Item { … }

Die duplizierte Initialisierung *zementiert* das Design vom Type

Small Tests nutzen, trivial zu schreibende, InMemory Fakes

ForWritingItems

InMemory ItemsDouble

<module>/internal/items.ts

```
1. export interface ForWritingItems { // Port
2.   add(item: Item);
3. }
```

fixtures/<module>/in-memory-items.ts

```
1. export class InMemoryItemsDouble // test double
2.        implements ForWritingItems {
3.
4.   #allItems: Item[] = [];
5.
6.   add(item: Item){
7.     this.allItems.push(item);
8.   }
9. }
```

GOH
Good Old Handcraft

Doch verhalten sich die Fakes wie der Produktionscode?

Halten sich beide Implementierungen an den selben Contract?

Synchron halten mit Contract Tests[1,2,3]

<module>/internal/items.ts

```
1. export interface ForWritingItems { // Port
2.   add(item: Item);
3. }
4.
5. export interface ForReadingItems { // Port
6.   getById(id: ItemId): Item;
7. }
```

tests/<module>/items.contract.ts

```
1.  const implementations = [
2.    [() => new InMemoryItemsDouble()], // Fake
3.    [() => new MongoItems()] // Real
4.  ];
5.
6.  describe.each(implementations, (configure) => {
7.    test('a saved item can be retrieved', () => {
8.      // GIVEN
9.      const testee = configure();
10.     const item = a.item();
11.     // WHEN
12.     testee.add(item);
13.     // THEN
14.     const result = testee.getById(item.id);
15.     expect(result).toEqual(item);
16.   });
17. });
```

1 J.B. Rainsbergers Original Post https://blog.thecodewhisperer.com/permalink/getting-started-with-contract-tests
2 Meine Kotlin Variante http://richargh.de/posts/Contract-Tests-in-Kotlin
3 Alternativer Name, Role Tests https://codesai.com/posts/2022/08/role-tests-jest

Die Bienenstock-Architektur umfasst

Stil

Test-Timeline

Echte Module

Test-DSL
& Contract Tests

Dekompositionsstrategie

Test-Stereotypen

Bei Fire&Forget Ports setzen wir auf Doubles

ForMobilePush

MobilePush NoopDouble

commons/mobile-push/mobile-push-facade.ts

```
1.  export interface ForMobilePush { // Port
2.    push(event): void;
3.  }
```

fixtures/commons/mobile-push/noop-mobile-push.double.ts

```
1.  export class NoopMobilePushDouble
2.          implements ForMobilePush {
3.
4.    push(event): void {
5.      // noop ☺
6.    }
7.  }
```

Bei Ports dessen Return der Test steuern muss, setzen wir auf EchoDoubles

ForAuthentication

Echo AuthenticationDouble

capi/auth/authentication.facade.ts

```
1. export interface ForAuthentication { // Port
2.   authenticate(user: User): Token;
3. }
```

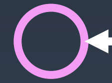
fixtures/capi/auth/echo-authentication.dobule.ts

```
1. export class EchoAuthenticationDouble
2.        implements ForAuthentication {
3.
4.   constructor(
5.     #authenticateEcho: Token = someToken()){ }
6.
7.   authenticate(user: User){
8.     return #authenticateEcho;
9.   }
10. }
```

Keine ifs, keine Logik.

Bei Ports die wichtige Parameter bekommen nutzen wir *spying*Doubles

ForAuthentication



Authentication EchoDouble

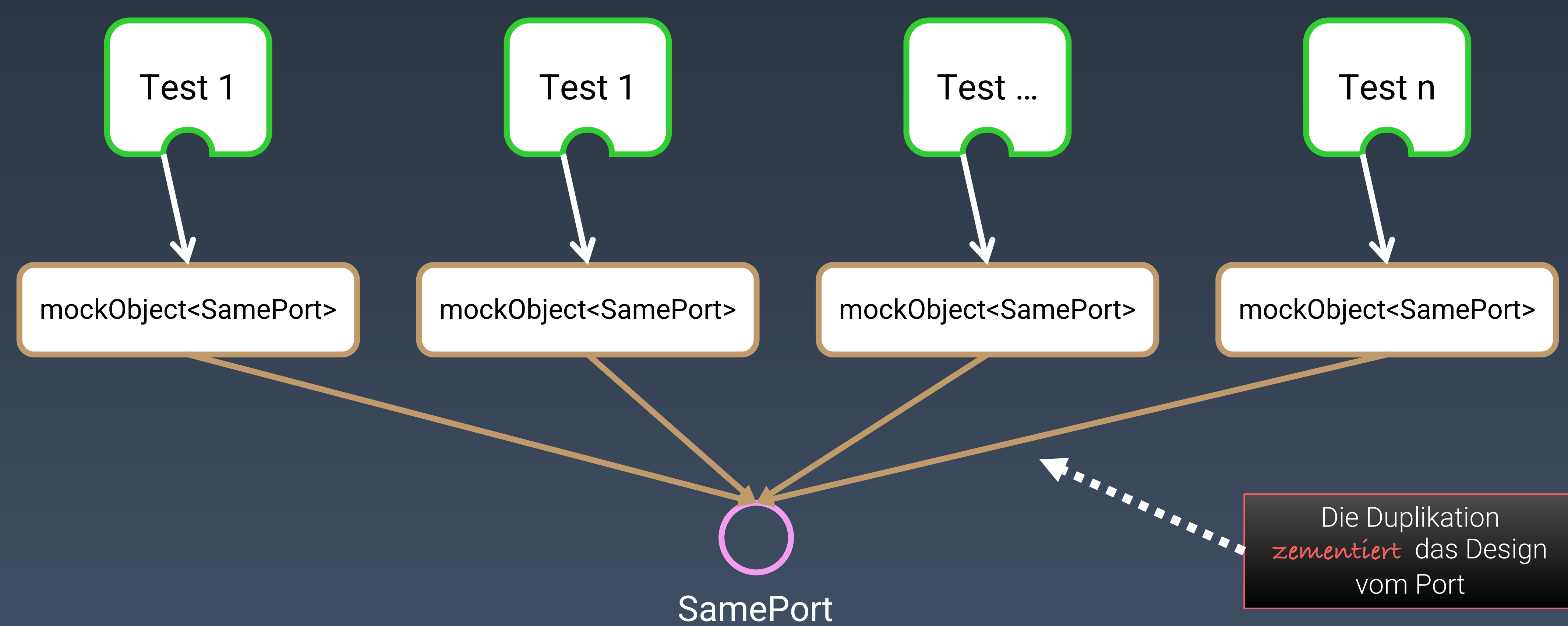
commons/events/events.facade.ts

```
1. export interface ForWritingEvents { // Port
2.   emit(event: Event): void;
3. }
```

fixtures/commons/event/spying-events.double.ts

```
1. export class SpyingEventsDouble
2.          implements ForWritingEvents {
3.
4.   #events: Event[] = [];
5.
6.   emit(event: Event): void;
7.     this.#events.push(event);
8.   }
9.
10.  eventsOfType(type: EventType): Event[] {
11.    // …
12.  }
13. }
```

Generell vermeiden strukturzementierendes, weil dupliziertes, Mocking



Übermäßiges Mocking hat Probleme, situatives ist praktisch

- Startup Zeit des Mocking-Frameworks
- Struktur-Zementierendes Mock-Setup
  - x Tests die die selbe Methode "mocken"
  - Behindern dann Strukturveränderungen dieser Methode
- In 95% der Fälle brauchen wir keinen Mock[1]
  - Bei Repos brauchen wir einen (InMemory) Fake
  - Bei Events einen Spy
  - Meistens einen (Echo) Stub
- Nur bei 3rd Party Services brauchen wir manchmal einen *Mock*[2]

1 https://martinfowler.com/articles/mocksArentStubs.html
2 Art of Unit Testing

Unterschiedliche Test-Stereotypes, alle fördern eine softe Struktur

Port

Prod

Test Double[1]

Spy[1]

Stub[1]

Fake[1]

Mock[1]

Test-Stereotypes
(Good Old Handcraft)

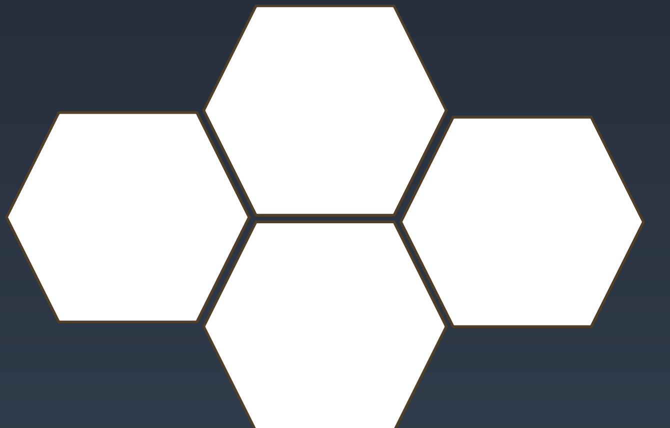
(Spying)Noop

(Spying)Echo

InMemory

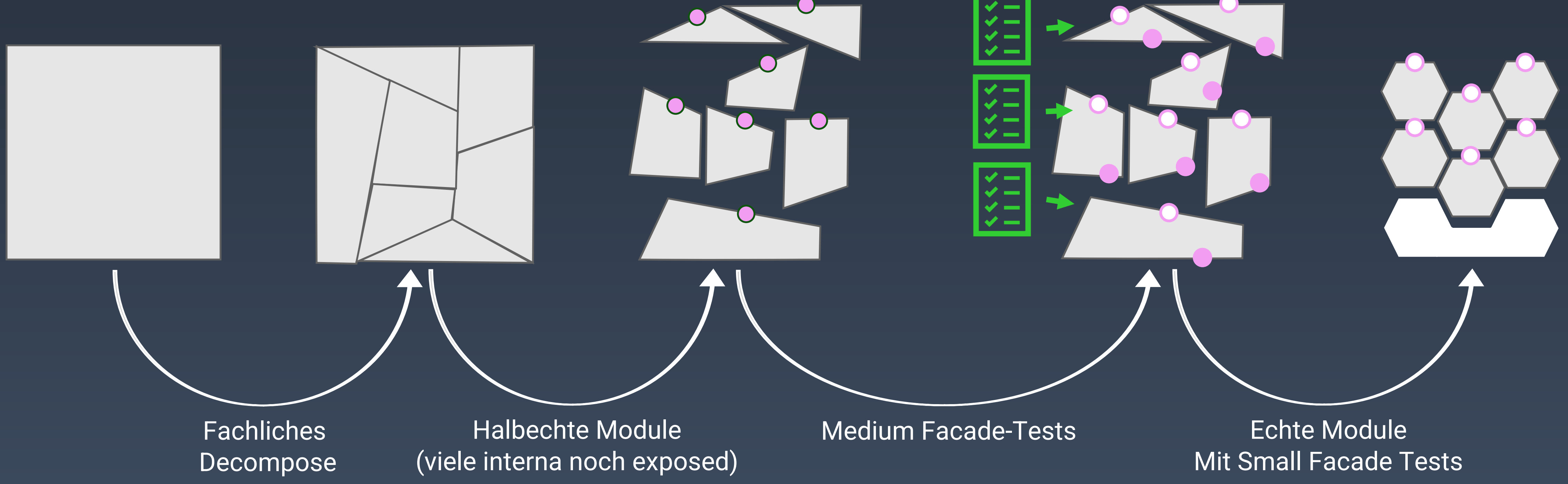
Port-Änderungen nur an *einer* Stelle nötig

1 http://xunitpatterns.com/Test%20Double%20Patterns.html



Die Bienenstock-Architektur bei Legacy Code

Recompose Legacy



Bienenstock Architektur…

Fordert

- Echte Module
- Entkopplung
- Findable SW Architecture
- Test-Dsl, Doubles und Contracts

Bietet

- Tests die
  - Struktur-flexibel
  - Schnell
  - Verlässlich
  - Geringer Wartungsaufwand
  - Wachsen mit Legacy Code mit

Dankeschön

Richard Gross (he/him)

Hypermedia-Designer

Archaeologist

Auditor

speakerdeck.com/richargh

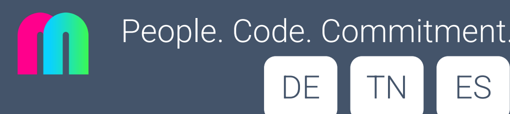
richargh.de/

@arghrich

Works for maibornwolff.de/
People. Code. Commitment.

DE TN ES

Backup



Alternative Stile

- Hexagonale Architektur nach eigenen Wünschen
- A-Frame Architecture und Nullable Infrastructure
- Test-Timeline [3,4]

3 Die agile Testpyramide https://www.youtube.com/watch?v=-35GpOJnjmM
4 Architektur, aber bitte agil! https://www.youtube.com/watch?v=12q6LDM8DIY

How hard is the code to understand?

Connascence als Refactoringhilfe

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*

Meilir Page-Jones

Jim Weirich's "Grand Unified Theory of Software Development" ☺

Connascence of

Weak

- Name: variable, method, SQL Table
- Type: int, String, Money, Person
- Meaning: what is `true`, `'YES'`, `null`, `love`
- Position: order of value
- Algorithm: encoding, SPA vs Server

Static

Dynamic

- Execution (order): one before other
- Timing: doFoo() in 500ms | doBar() in 400ms
- Value: constraints on value, invariants
- Identity: reference same entity

Strong

Meilir Page-Jones | Jim Weirich's "Grand Unified Theory of Software Development" ☺

Connascence of

Strong

- Identity
- Value
- Timing
- Execution order

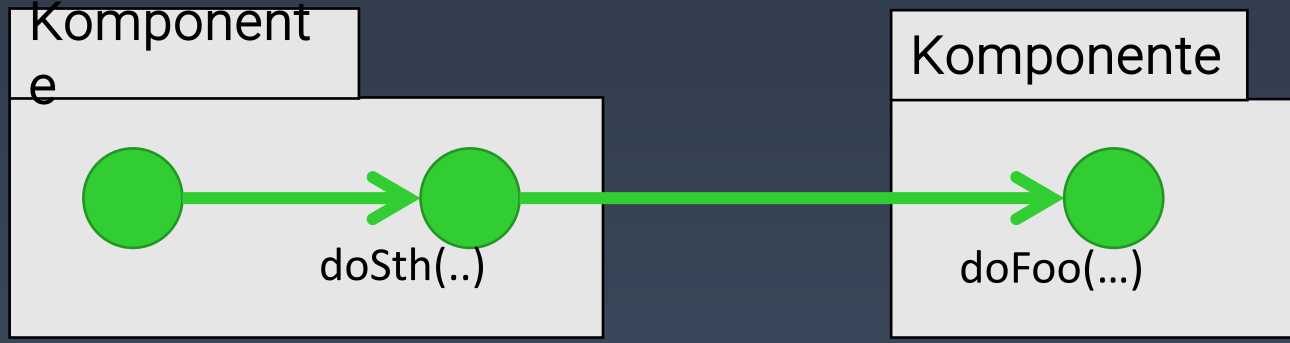
- Algorithm
- Position
- Meaning
- Type
- Name

Weak

Refactor this way

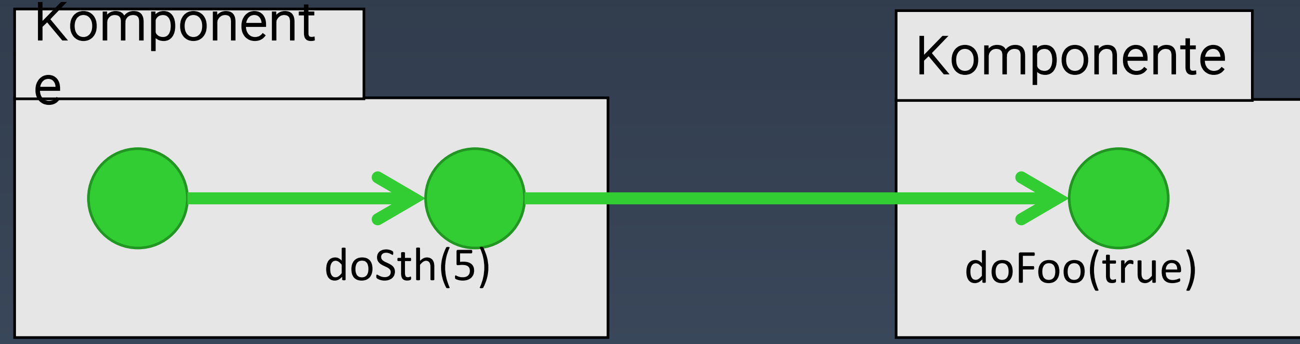
Meilir Page-Jones | Jim Weirich's "Grand Unified Theory of Software Development" ☺

Connascence of Name



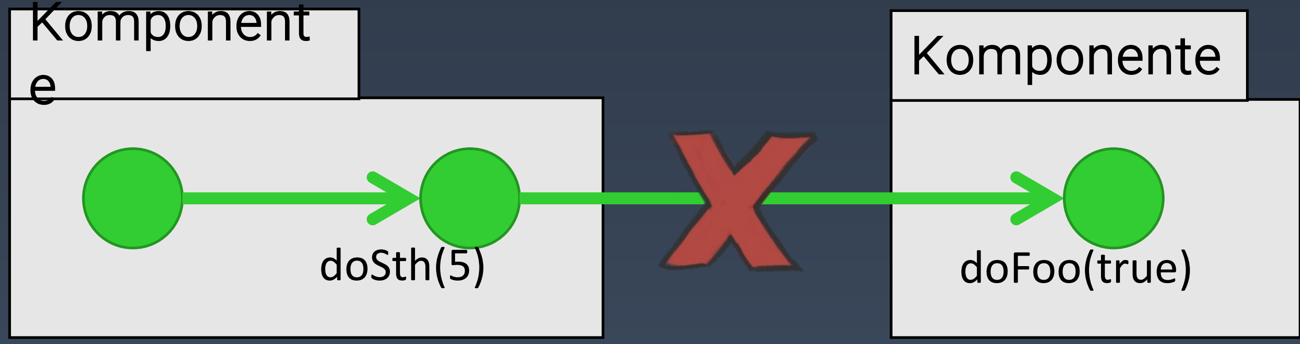
Jim Weirich's "Grand Unified Theory of Software Development" ☺

Connascence of Meaning



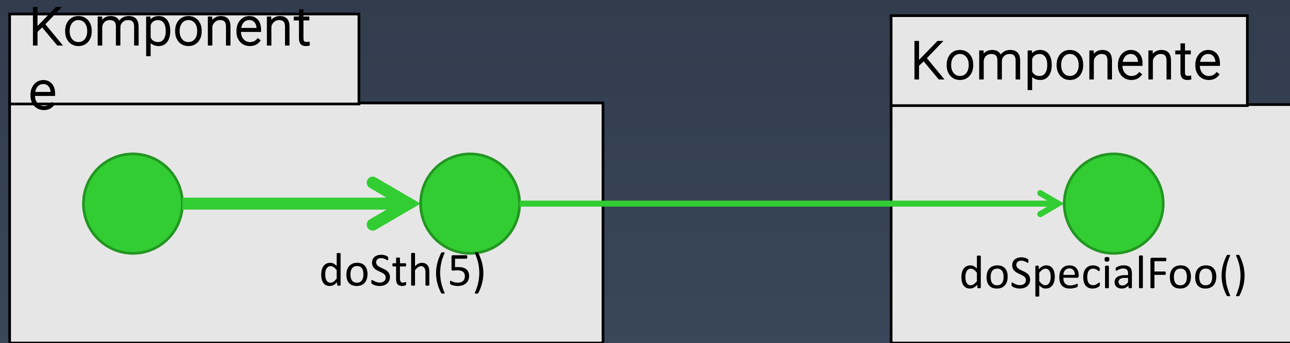
Jim Weirich's "Grand Unified Theory of Software Development" ☺

Locality is important



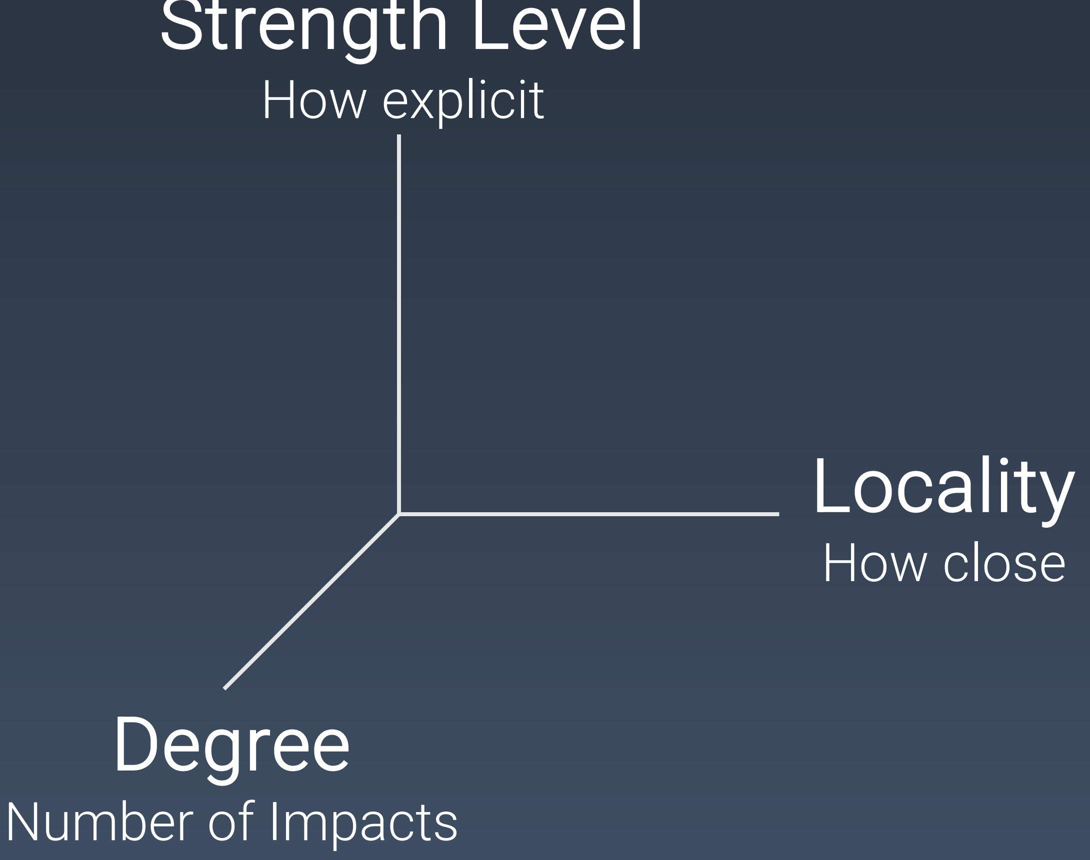
Jim Weirich‘s “Grand Unified Theory of Software Development” ☺

Local can be stronger



Jim Weirich's "Grand Unified Theory of Software Development" ☺

3-axes of Connascence



Meilir Page-Jones | Jim Weirich's "Grand Unified Theory of Software Development" ☺

A Set of Unit Testing Rules

A test is not a unit test if:

- It talks to a database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

1 A Set of Unit Testing Rules https://www.artima.com/weblogs/viewpost.jsp?thread=126923

> High-quality code is modular, loosely-coupled, has high-cohesion, good separation of concerns, and exhibits information-hiding.

Dave Farley

@davefarley77

https://pages.xebia.com/whitepaper-test-driven-development-is-not-about-unit-tests

Encapsulation

- Ein objekt guarantiert, dass es immer in einem erlaubten Zustand ist
  - Vorbedingungen beschreiben die Veranwortlichkeit des Aufrufenden
  - Nachbedingungen beschreiben die Verantwortlichkeit des Objkets

Slides by richargh.de