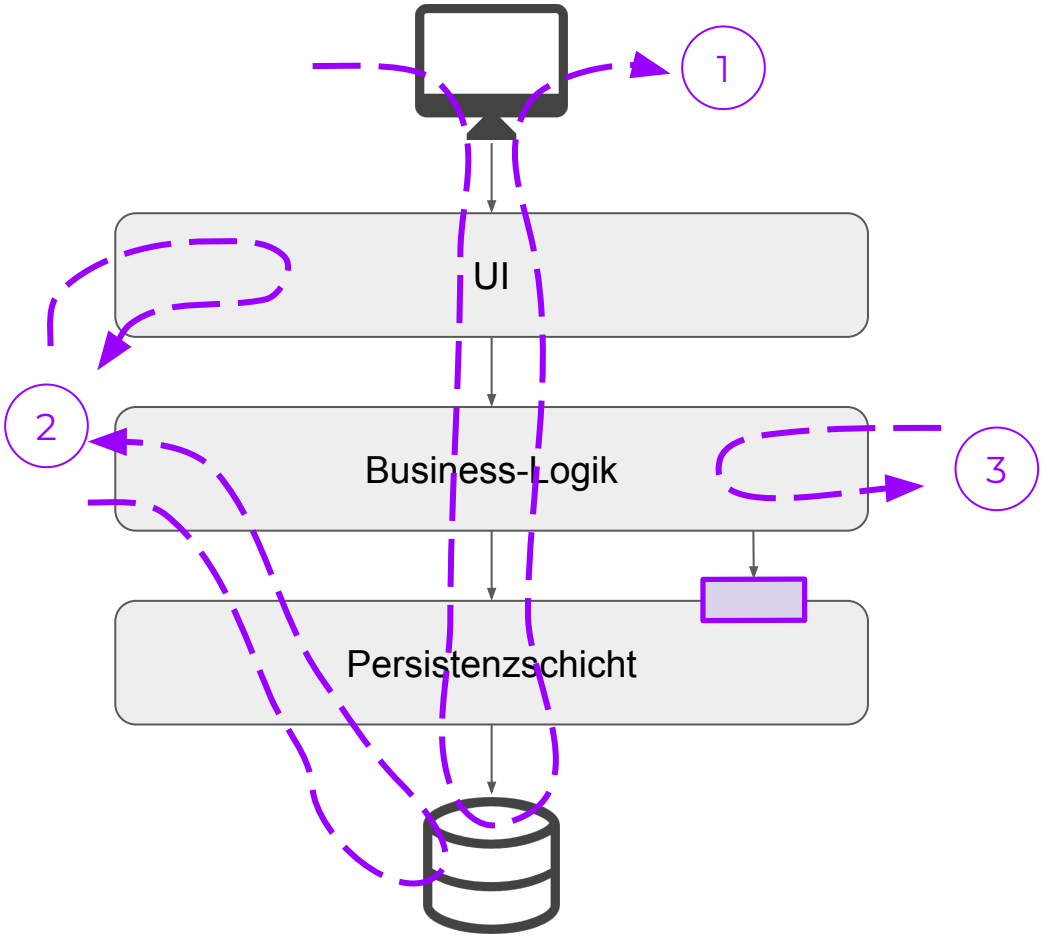
Teststrategien für Hexagonale Architektur

Kennt ihr das?

Agenda

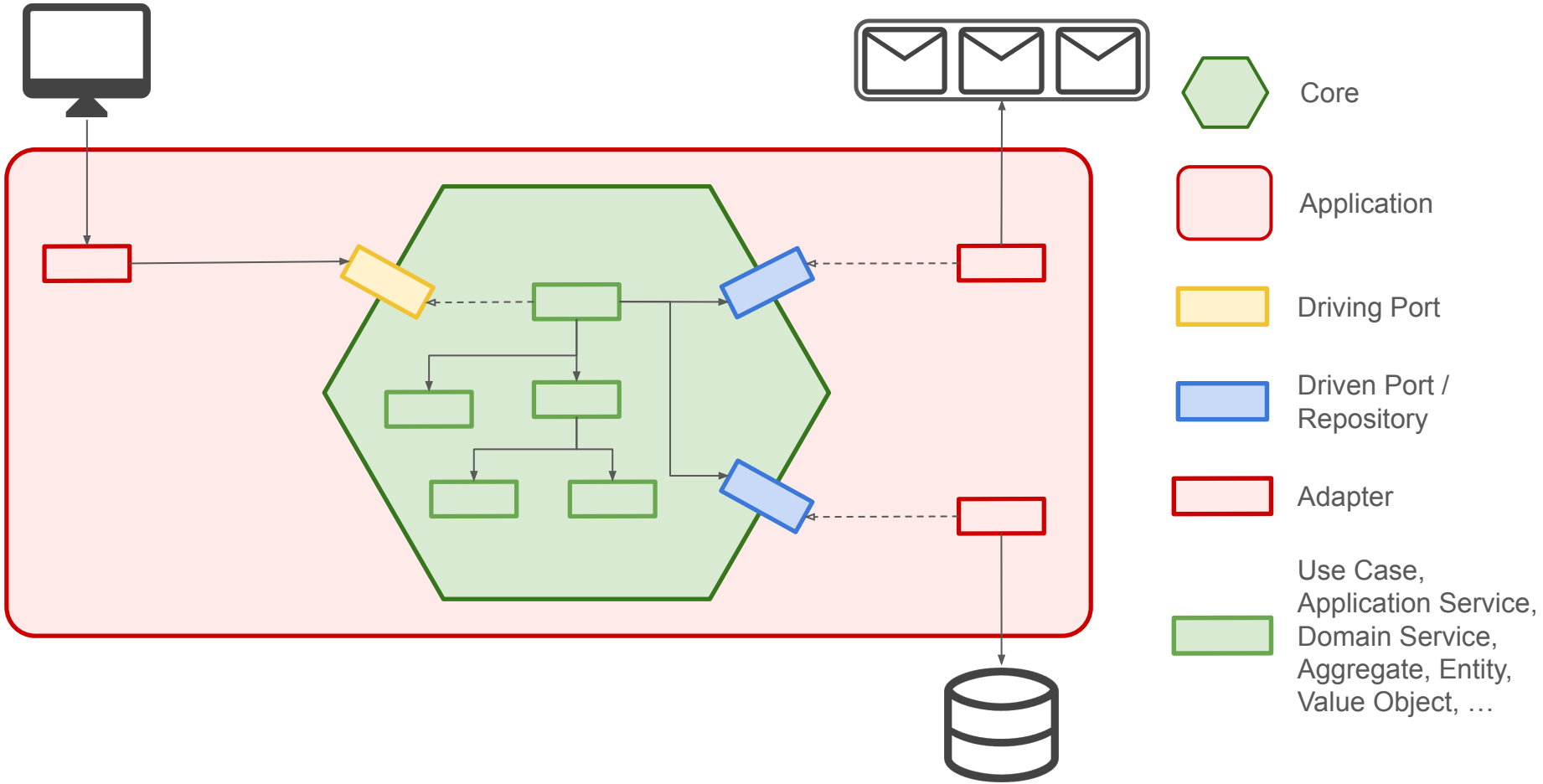
1. Warum Hexagonale Architektur gut testbar ist
2. Qualitätsmerkmale von Tests
3. Testarten in der Hexagonalen Architektur
4. Woher kommen die Testdaten?
5. Test Doubles bewusst einsetzen
6. Architektur-Tests
7. Testabdeckung messen
8. Mehrere Hexagone zusammen Testen



1 E2E-Tests:
sehr langsam, fragil

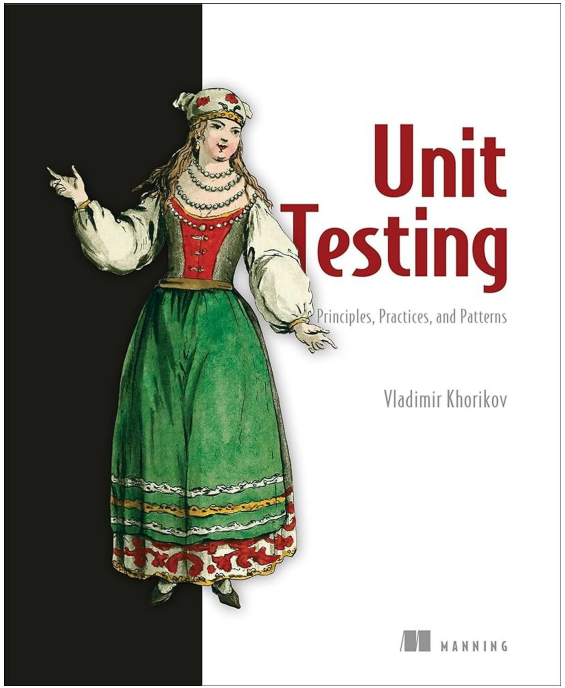
2 UI- & Integration-Tests:
auch langsam

3 Unit-Tests:
schnell, aber brechen
bei Refactorings


Core
Application
Driving Port
Driven Port /
Repository
Adapter
Use Case,
Application Service,
Domain Service,
Aggregate, Entity,
Value Object, …

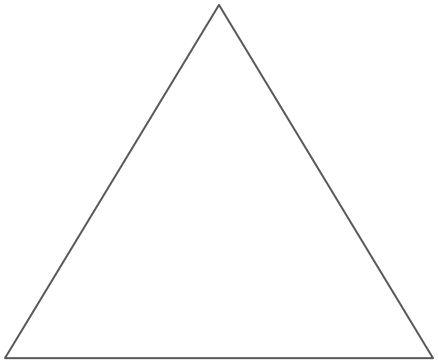
Was sind gute Tests? (4 Pillars nach Khorikov)

- 🛡️ Schutz vor Regression
- 🏗️ Beständigkeit gegen Refactorings
- ⚡ Schnelles Feedback
- 🛠️ Wartbarkeit



🛡️

Schutz vor Regression

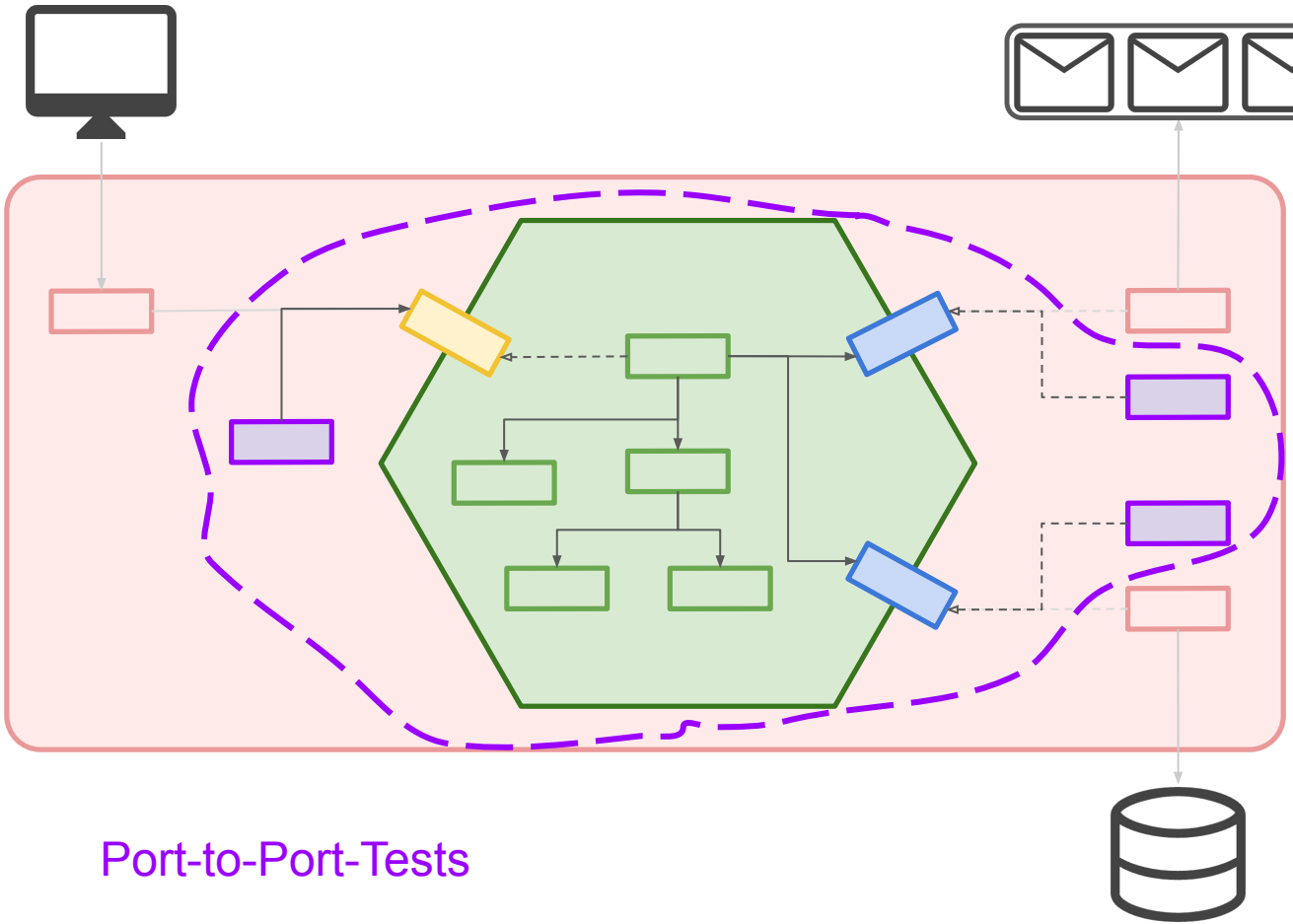


⚡

Schnelles Feedback

🏗️

Beständigkeit gegen Refactorings



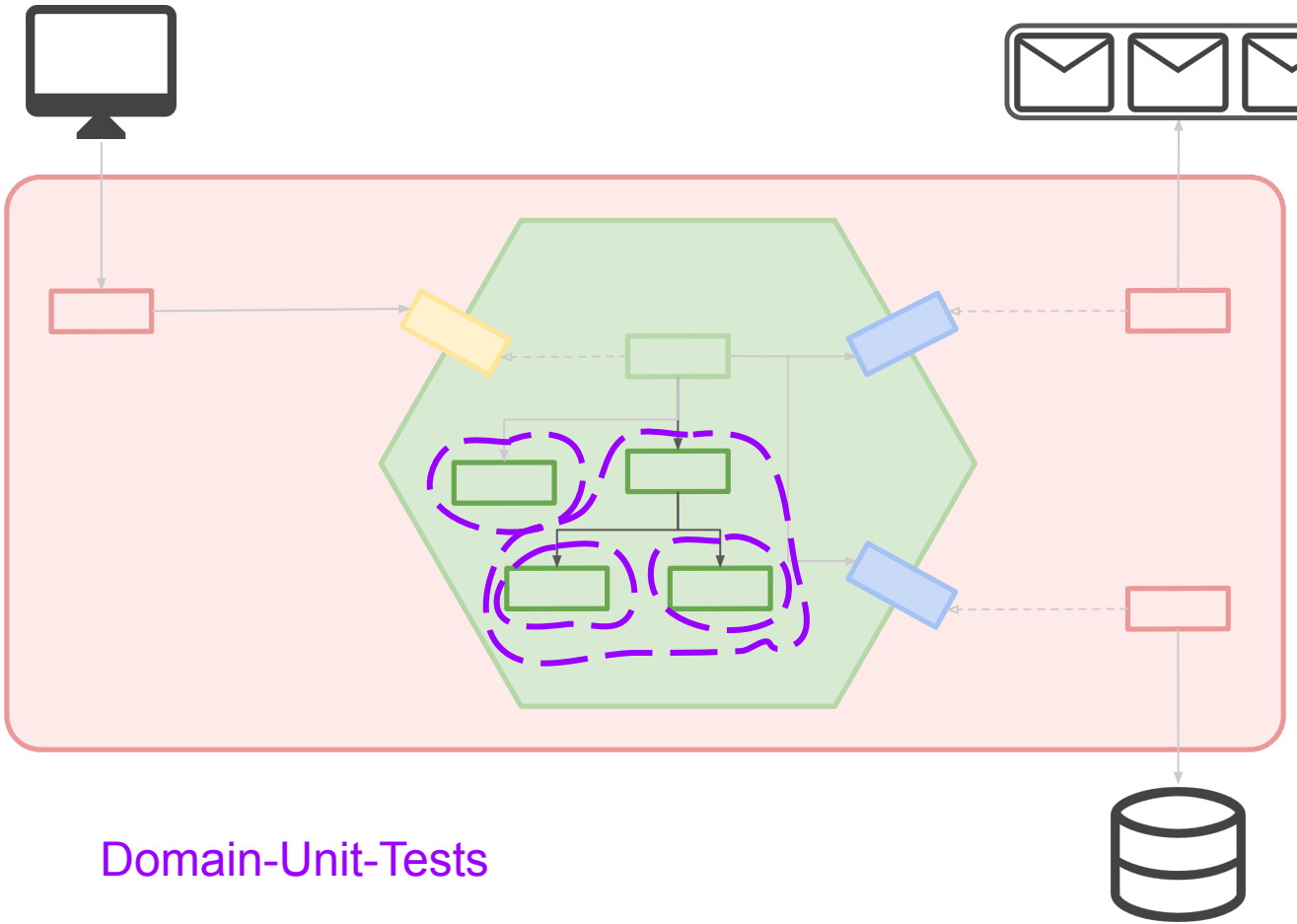
0 ····· 1

0 ····· 1

0 ····· ⚡ 1

0 ····· 1

Port-to-Port-Tests



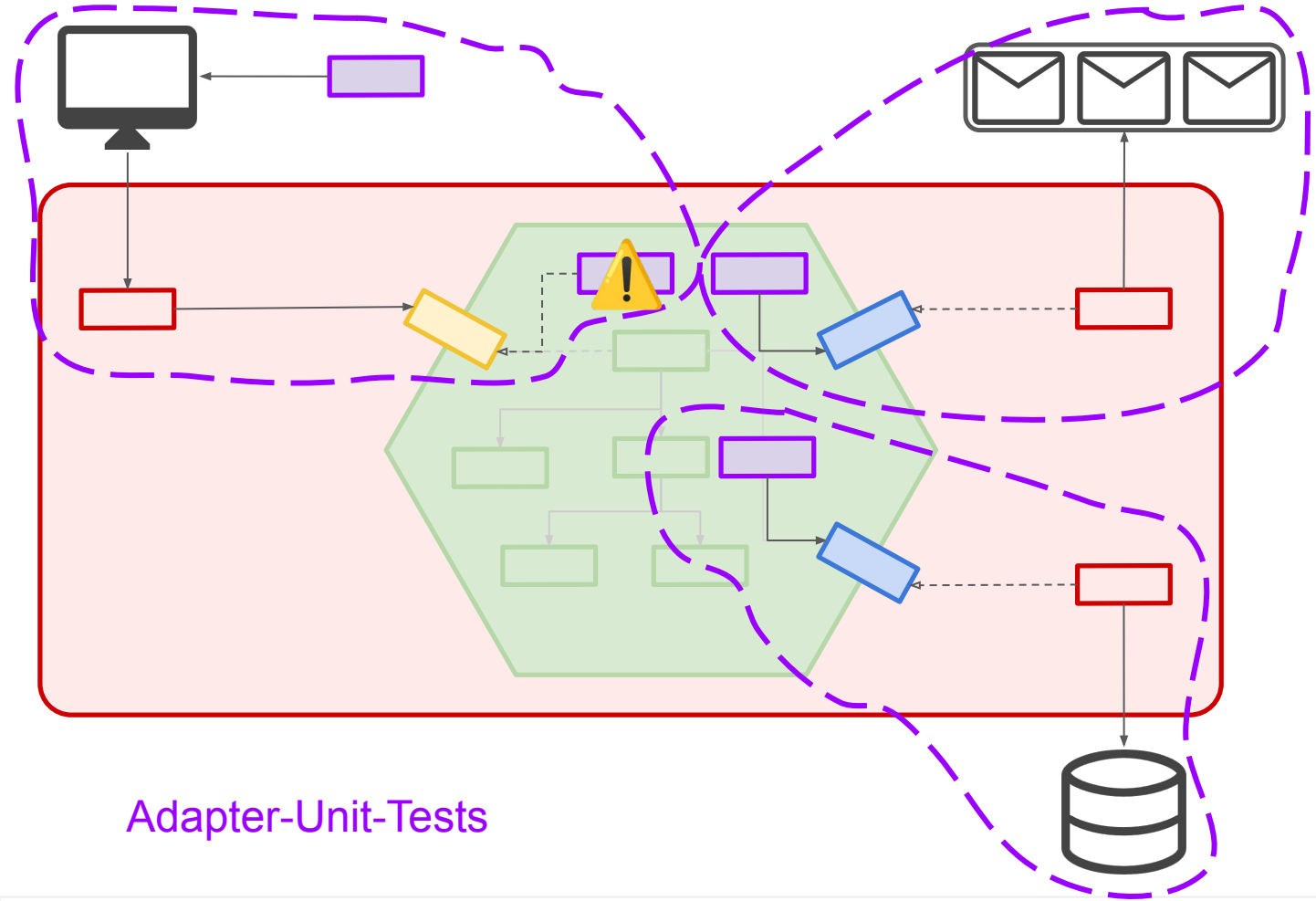
0 · · · · · · 1

0 · · · · · · 1

0 · · · · · · · · · · ⚡ 1

0 · · · · · · · · · ·

Domain-Unit-Tests

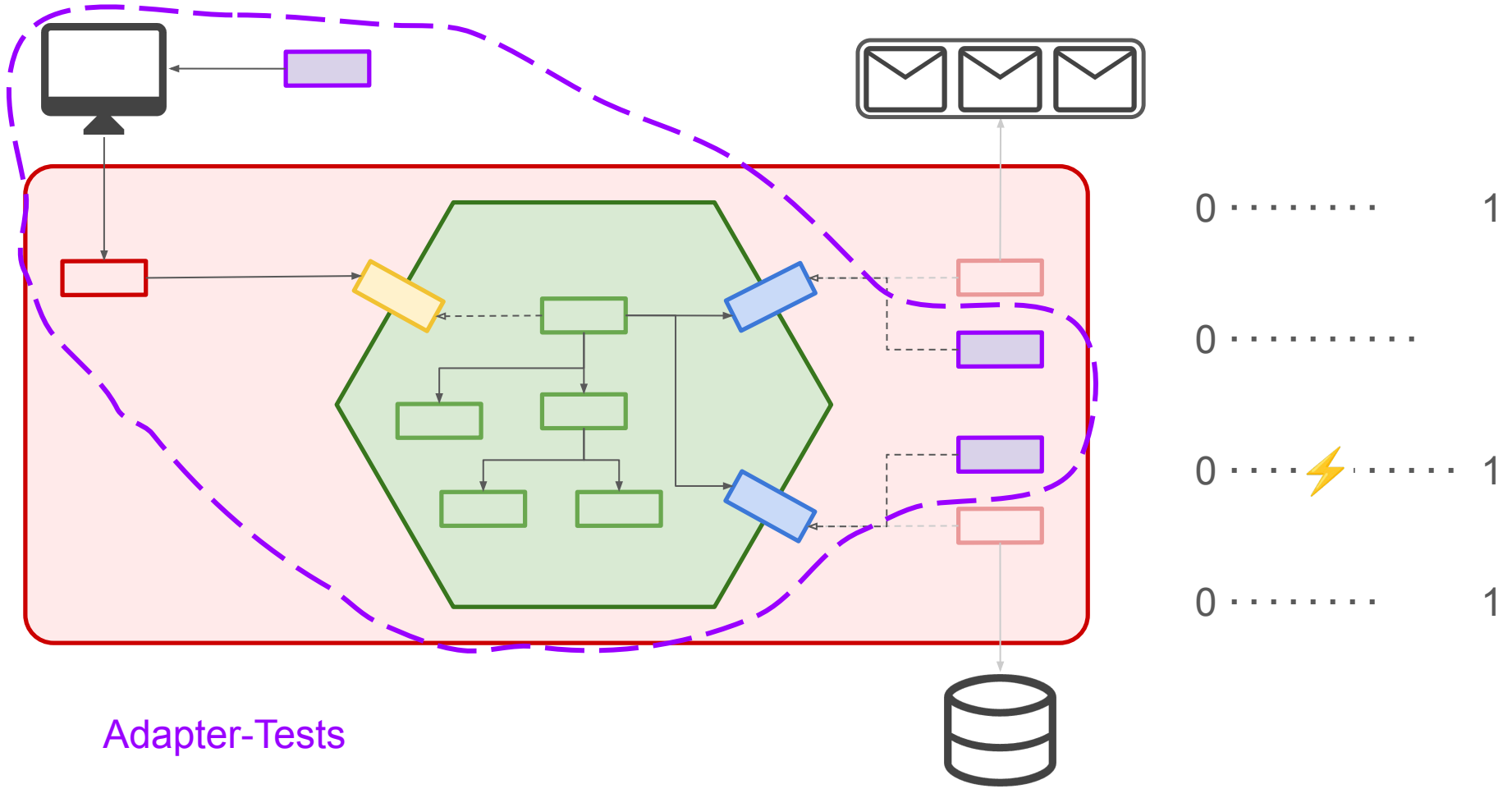


0 ····· ·· 1

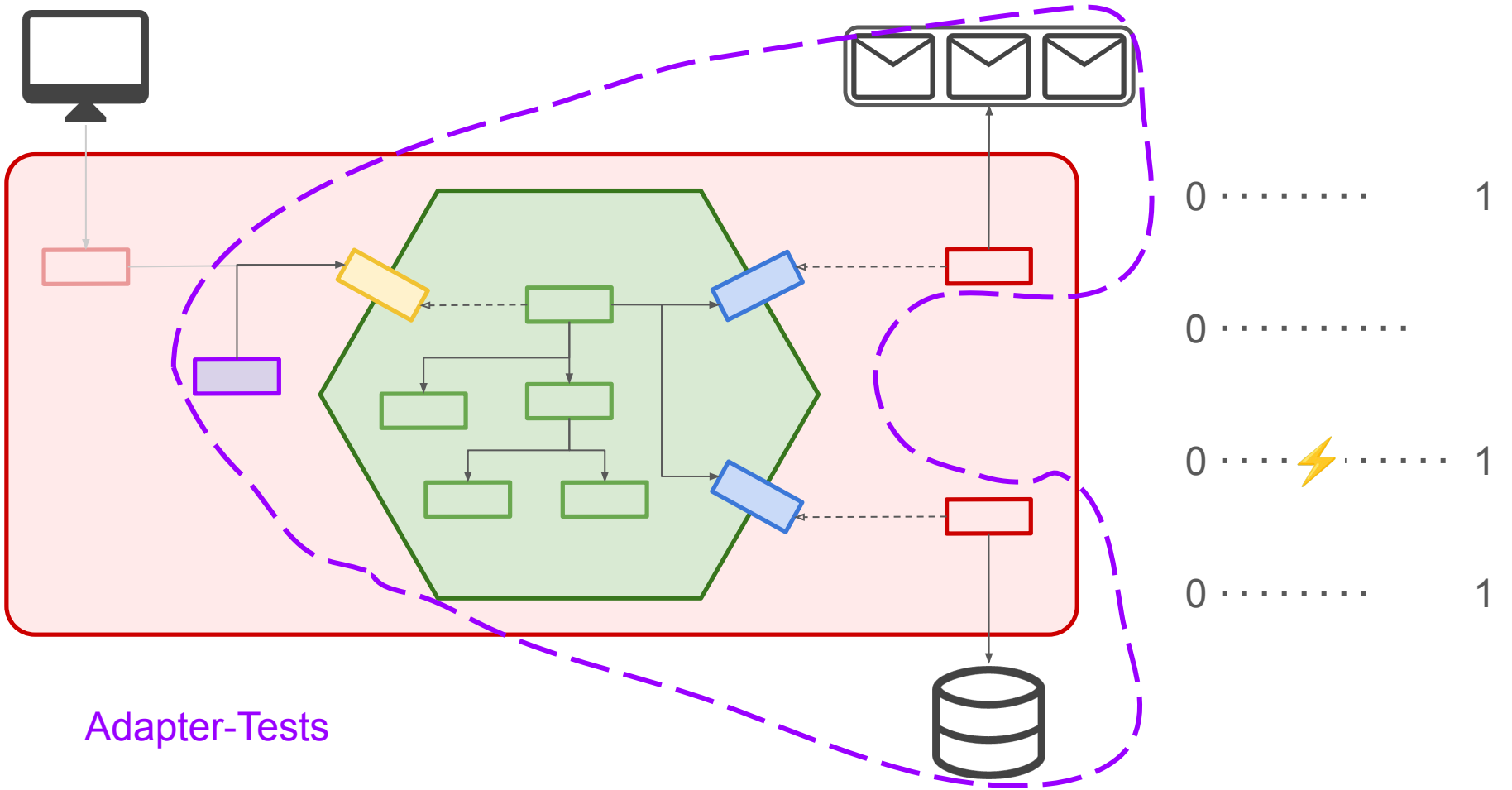
0 ··········

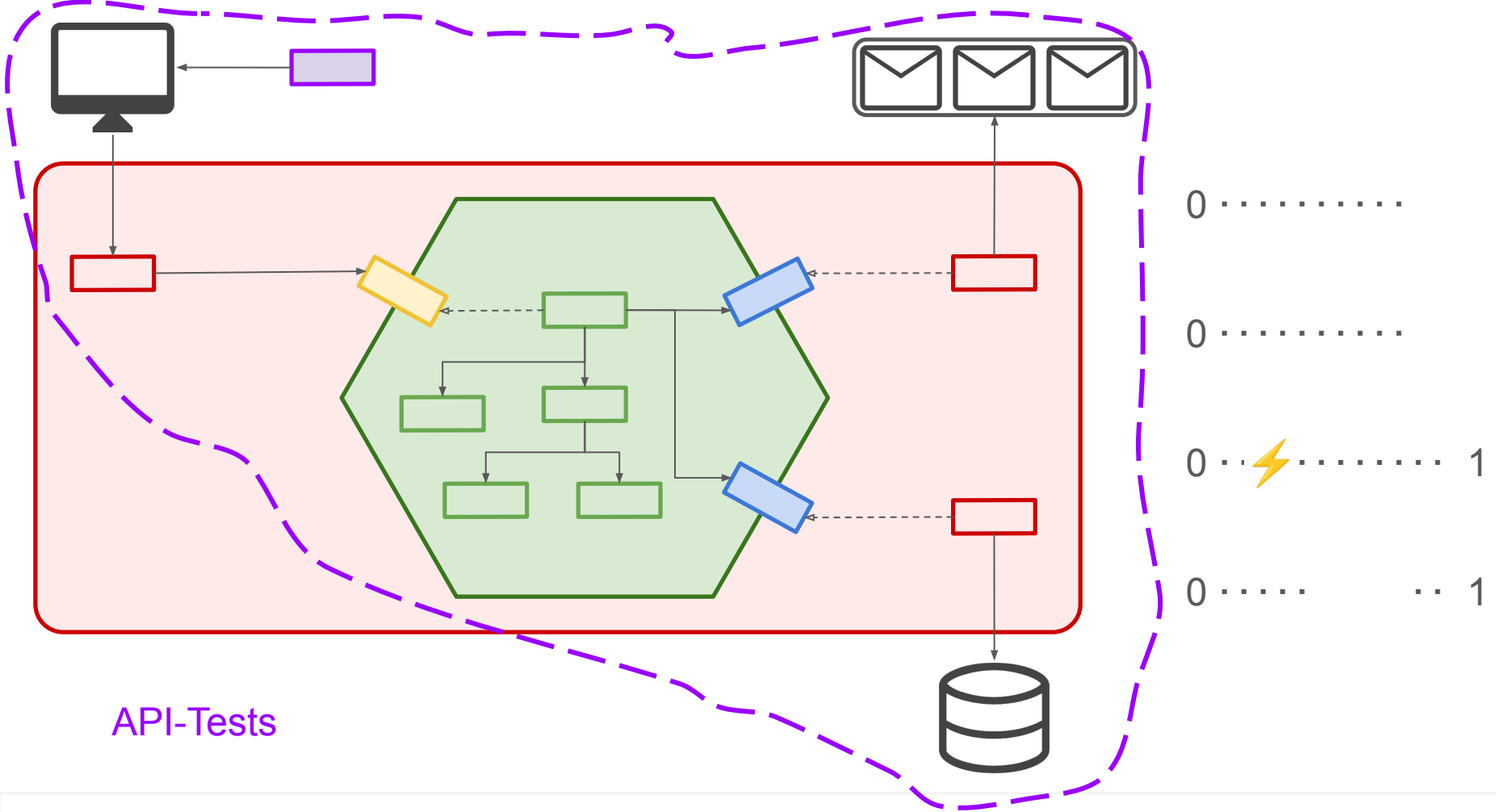
0 ····⚡····· 1

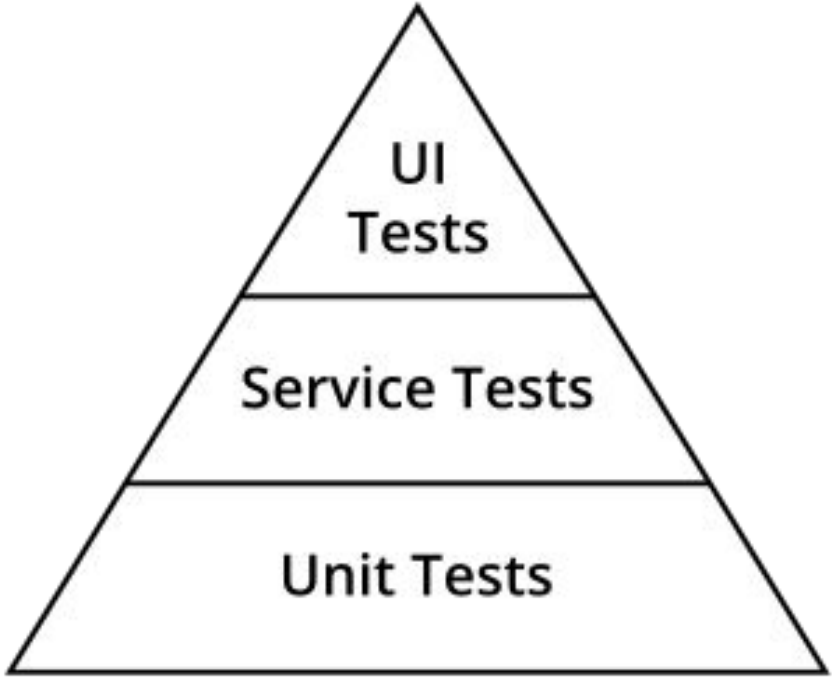
0 · ····· 1

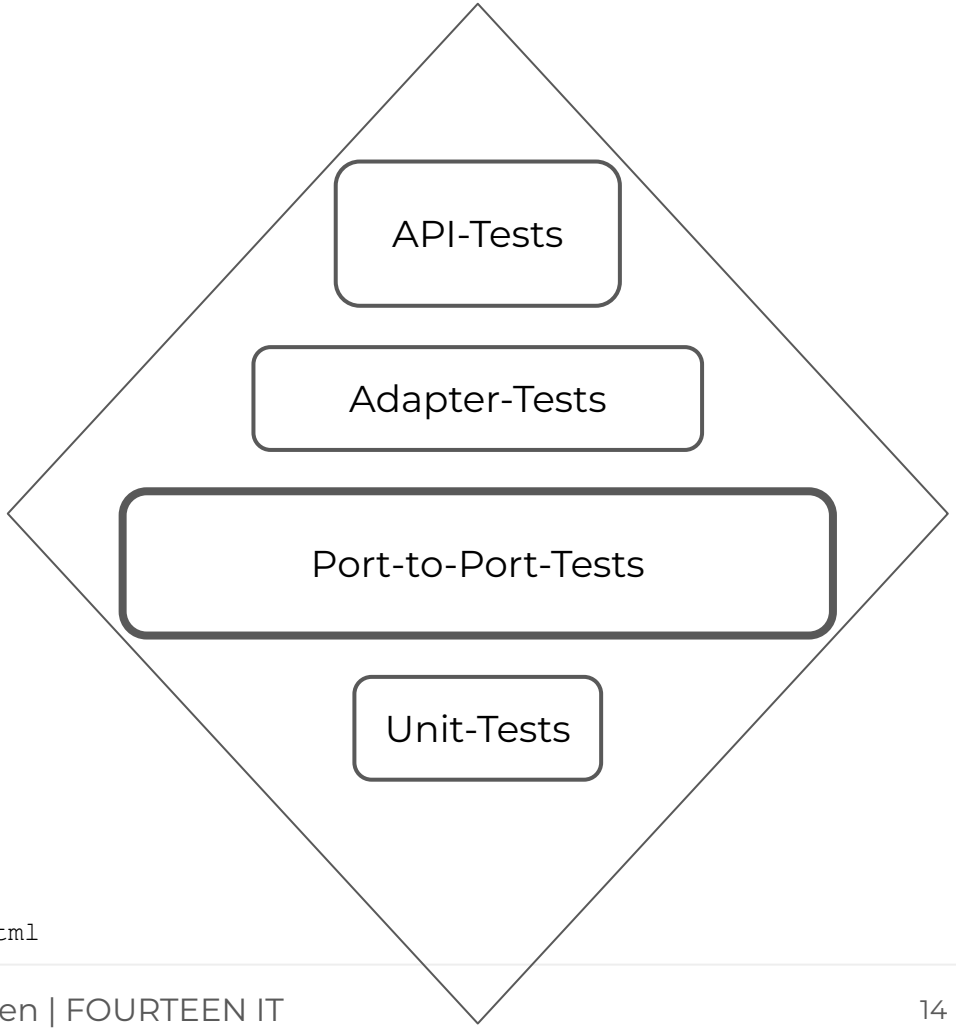


Adapter-Tests

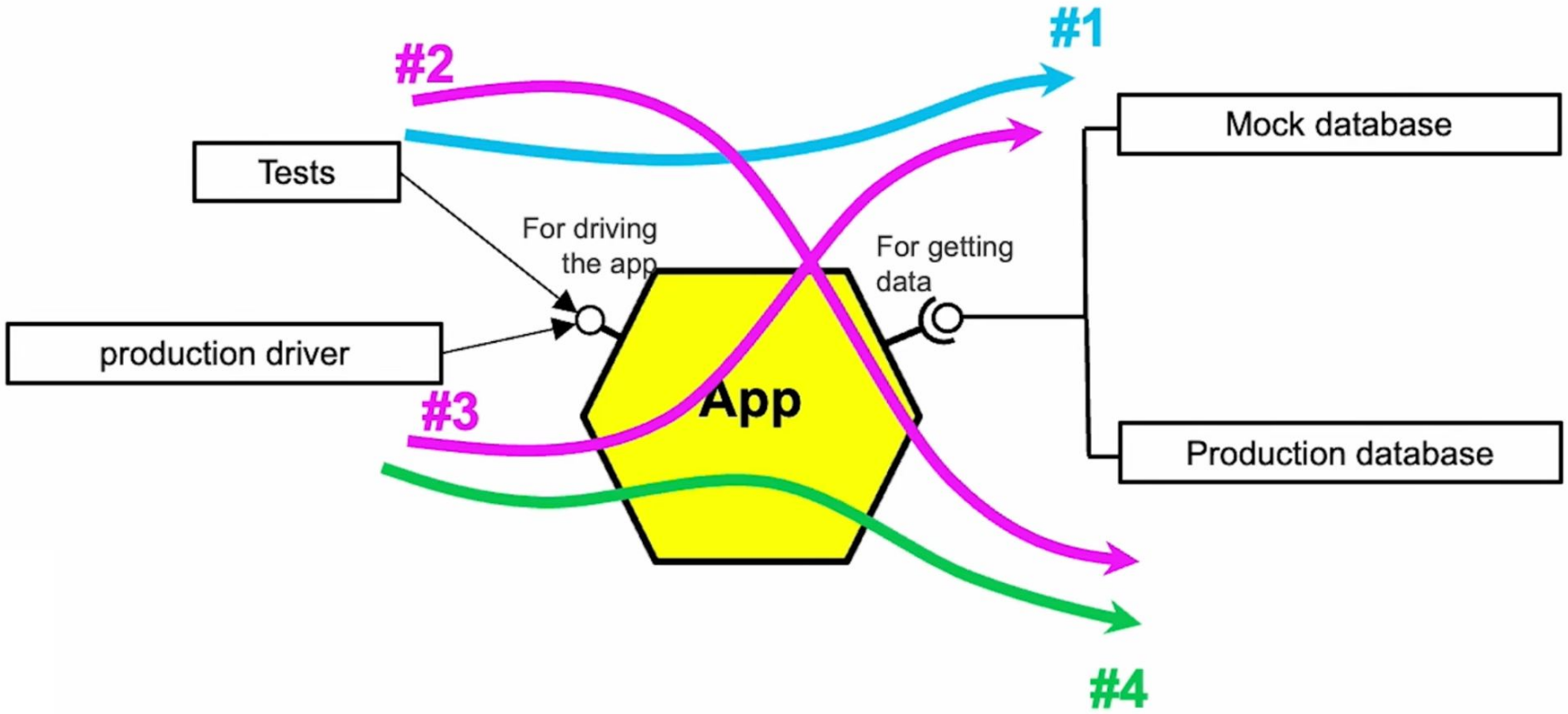

0 1
0
0 1
0 1
Adapter-Tests


0
0
0
1
0
1
API-Tests





https://martinfowler.com/articles/practical-test-pyramid.html



The Hexagonal - Ports & Adapters Architecture | Alistair Cockburn | SAG 202
https://www.youtube.com/watch?v=ChUlRa0xsWo

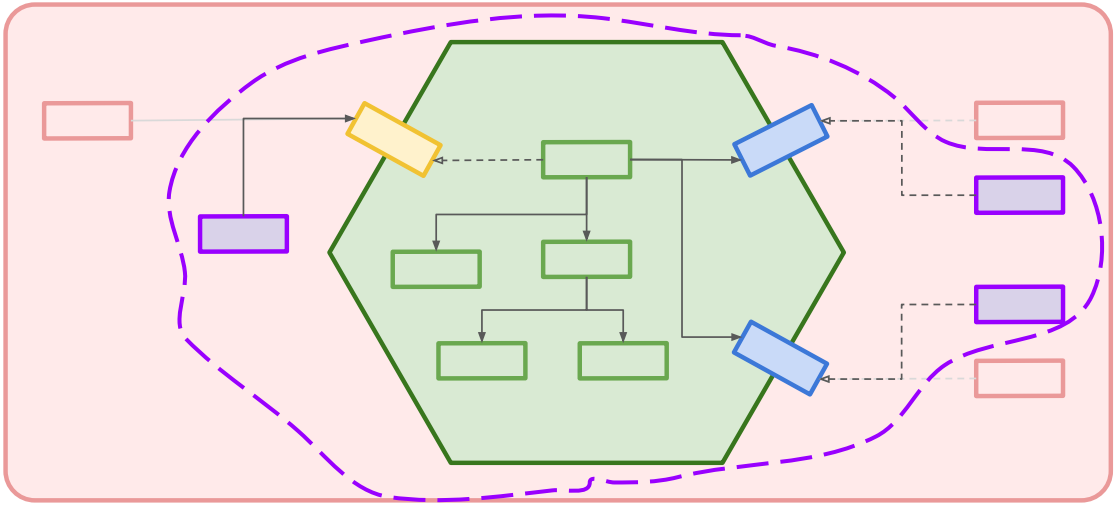
Woher kommen die Testdaten?

Woher kommen die Testdaten für Port-to-Port-Tests?

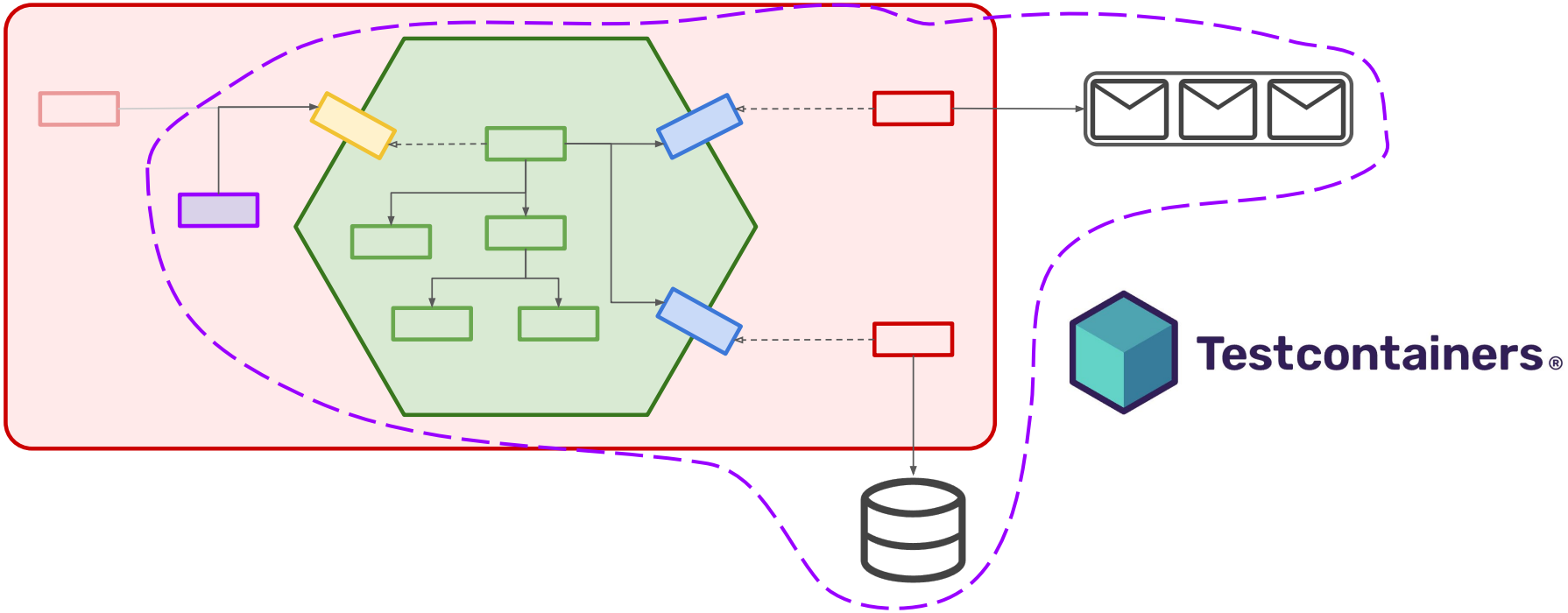
```
aCustomer()
  .withName("Jens Müller")
  .withDateOfBirth("1987-11-27")
  .withStatus(BLOCKED)
  .build();

aCustomer()
  .withStatus(BLOCKED)
  .build();

jens();
```



Woher kommen die Testdaten für Adapter-Tests?



https://testcontainers.com/

Woher kommen die Testdaten für Lasttests?

Synthetische Testdaten aus Produktionsdaten
durch Tools wie Synthetic Data Vault oder TONIC
oder selbstgebaute Pipelines

Anonymisierung allein reicht oft nicht aus für Datenschutz

https://docs.sdv.dev/sdv
https://www.tonic.ai/

Faustregel zu Testdaten

Testdaten sollen die fachliche Absicht des Tests sichtbar machen. Wenn man im Testcode mehr Zeit damit verbringt, das Setup zu verstehen als die eigentliche Assertion, ist das Daten-Setup falsch.

Test Doubles bewusst einsetzen

Was sind Test Doubles?

Dummy Wird übergeben, aber nicht verwendet

Stub Liefert vordefinierte Antworten auf Anfragen

Spy Zeichnet auf, wie er aufgerufen wurde

Mock Schlägt sofort fehl, wenn Erwartungen nicht erfüllt werden

Fake Eine funktionierende, für Tests vereinfachte Implementierung

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

Spy vs. Fake

```
MailServer spy =
    mock(MailServer.class);

//...

verify(spy).send(new Mail(
  "test@test.de", "Lorem ipsum…"));

verify(spy).send(new Mail(
  "test@test.de", "Hello again…"));
```

```
class MailServerFake
    implements MailServer {
  List<Mail> mailsSent =
    new ArrayList<>();

  void send(Mail mail) {
    mailsSent.add(mail);
  }

  void sendAll(Mail… mails) {
    mailsSent.addAll(mails);
  }

  List<Mail> mailsSent() {
    return mailsSent;
  }
}
```

Faustregeln zu Test Doubles

1. Für Driven Ports im Port-to-Port-Test:
   Bevorzuge Fakes & teile sie zwischen Tests
2. Für externe Systeme mit Seiteneffekten: am besten Fakes
3. Mocks mit strikten Erwartungen und Spies sparsam einsetzen,
   sie bremsen Refactorings
4. Stubs für Query-Operationen, wenn Fakes Overkill wären
5. Niemals den Domänenkern doublen

Architektur-Tests

Architektur-Tests

Hexagonale Architektur lebt von Abhängigkeitsregeln.

Diese Regeln durchzusetzen ist eine eigene Test-Disziplin.

ArchUnit

pytest-archon

ArchUnitNET

https://www.archunit.org/
https://github.com/sverweij/dependency-cruiser
https://pypi.org/project/pytest-archon/
https://archunitnet.readthedocs.io/en/stable/

Beispiele in ArchUnit

```
noClasses()
  .that().resideInAPackage("..domain..")
  .should()
  .dependOnClassesThat().resideInAPackage("..adapter..");
```

Zur Inspiration: github.com/whiskeysierra/archunit-hexagonal

Testabdeckung messen

Gute & Schlechte Metriken

Code Coverage …

… mist, ob eine Zeile durchlaufen wurde

… mist nicht, ob ihr Verhalten geprüft wurde

Gute & Schlechte Metriken

Mutation Testing …

- … misst, welche Code-Änderungen durch Tests gefunden werden
- … betrachtet nur den Schutz vor Regression
- … ist rechenintensiv
- … lohnt sich für komplizierte Teilsysteme (≥ 99 %)

mutmut

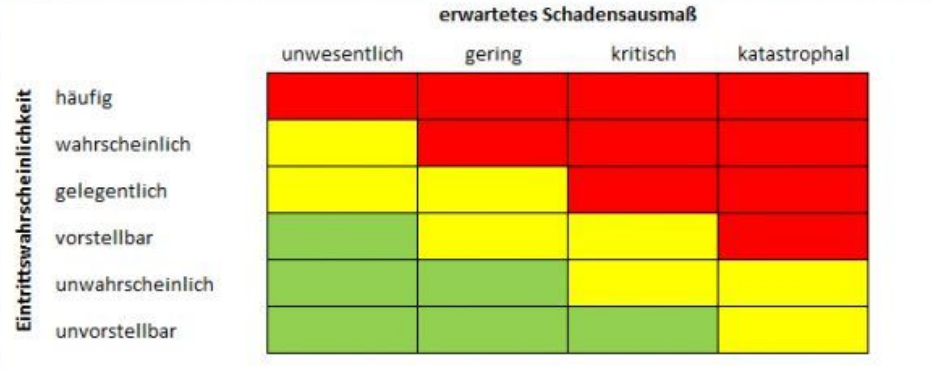
https://pitest.org/
https://stryker-mutator.io/
https://mutmut.readthedocs.io/

Gute & Schlechte Metriken

Welche Fehler wären für uns katastrophal?

Wie wahrscheinlich ist es, dass diese Fehler auftreten?

⇒ Sind diese Pfade gut getestet?

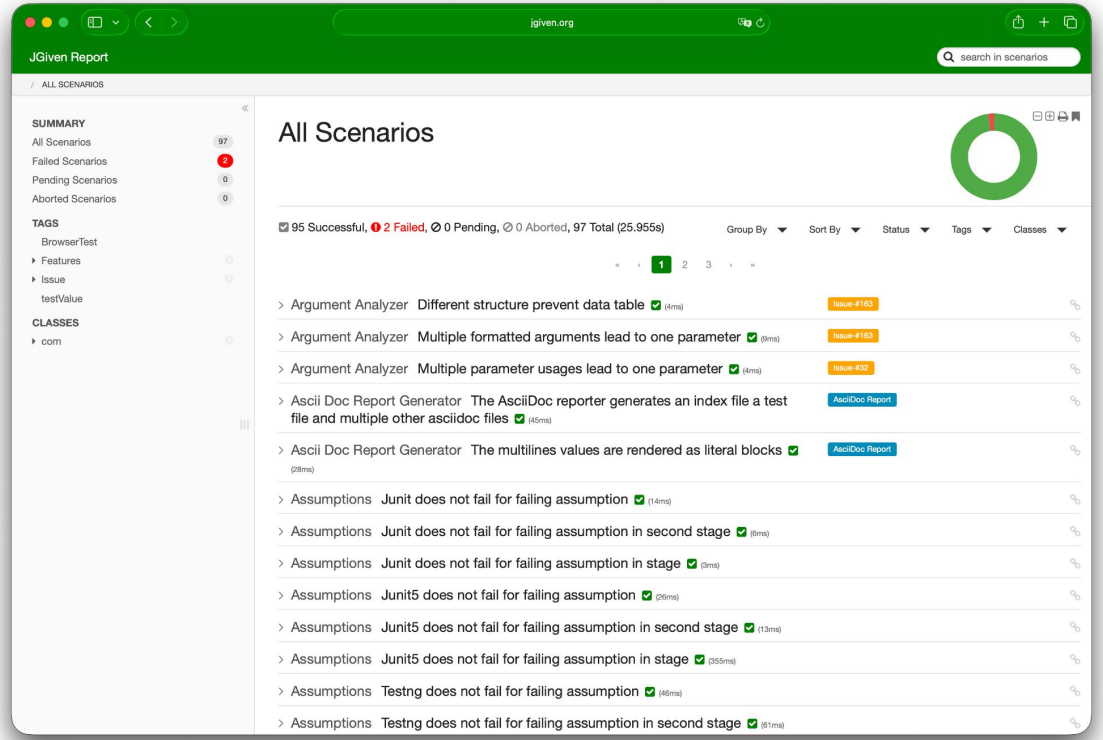


https://der-qmb.info/54-risikoinventar-risikomatrix/

Gute & Schlechte Metriken

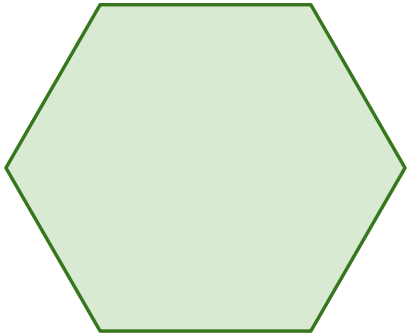
Feature-Abdeckung statt Code-Abdeckung:
Sind alle fachlichen Features durch Tests abgedeckt?

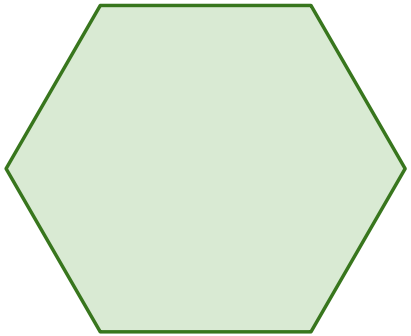
Lösungsansatz:
Behavior-driven Development
z.B. mit JGiven



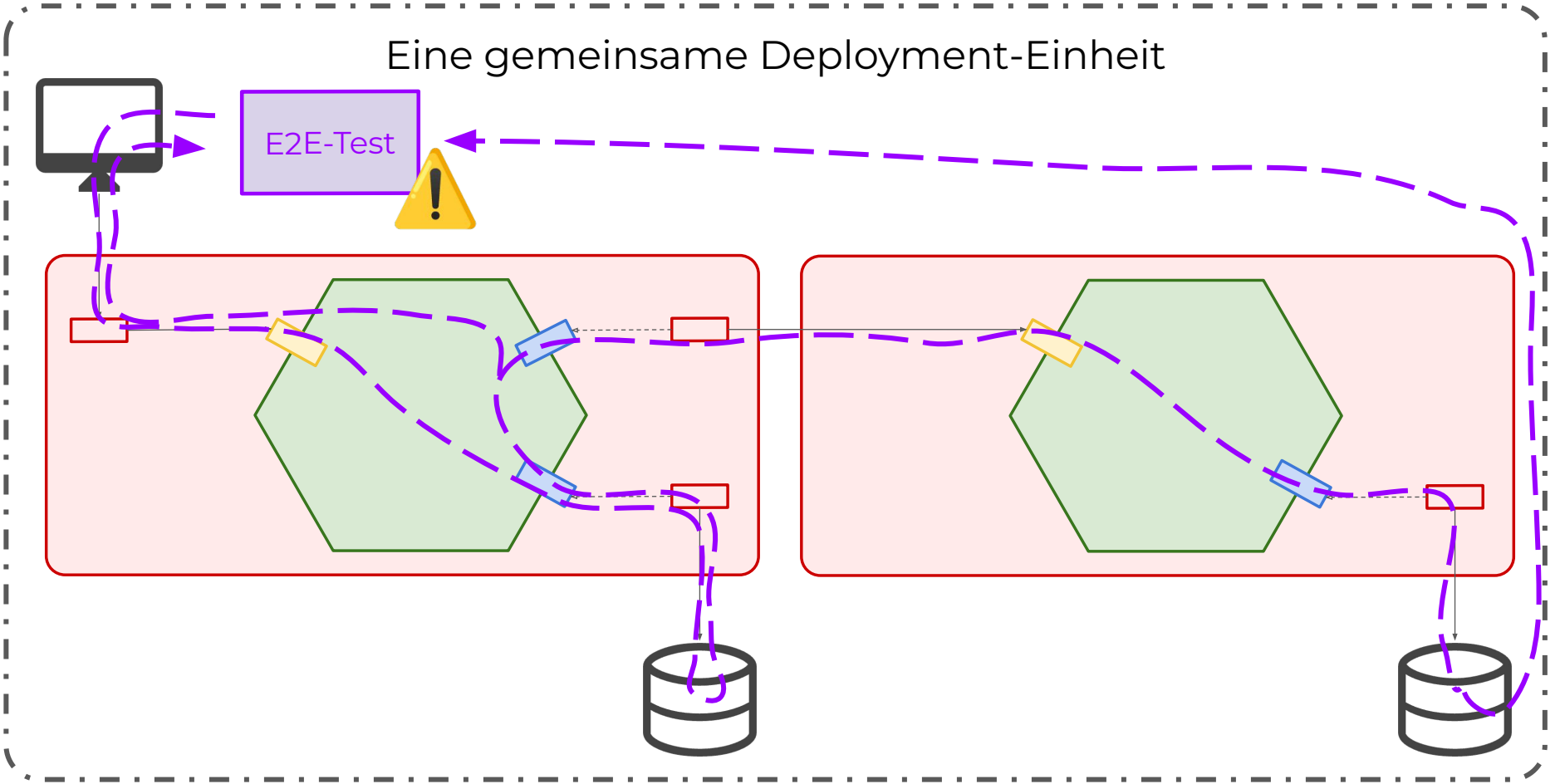
`https://jgiven.org/userguide/`

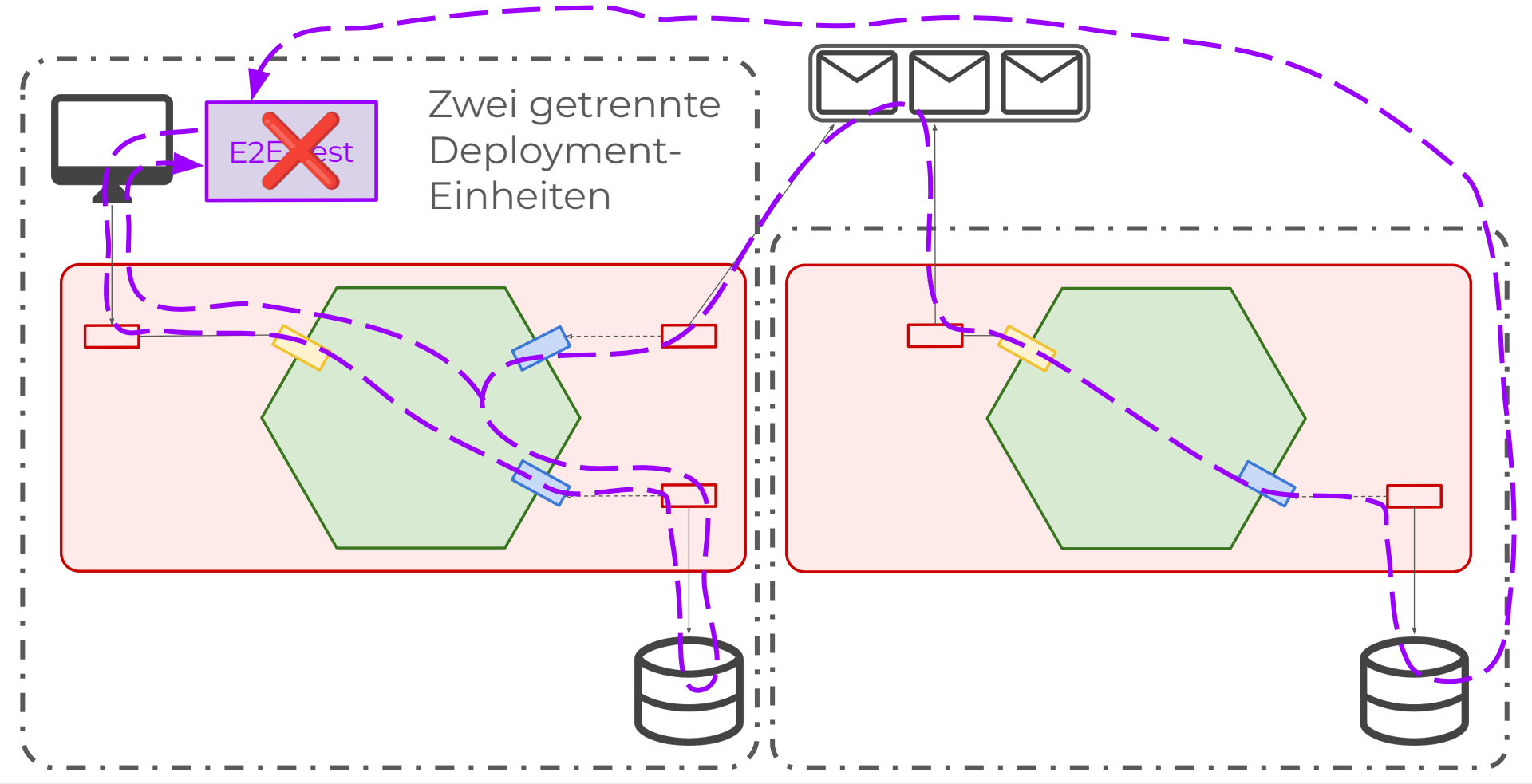
Testen über mehrere Hexagone hinweg

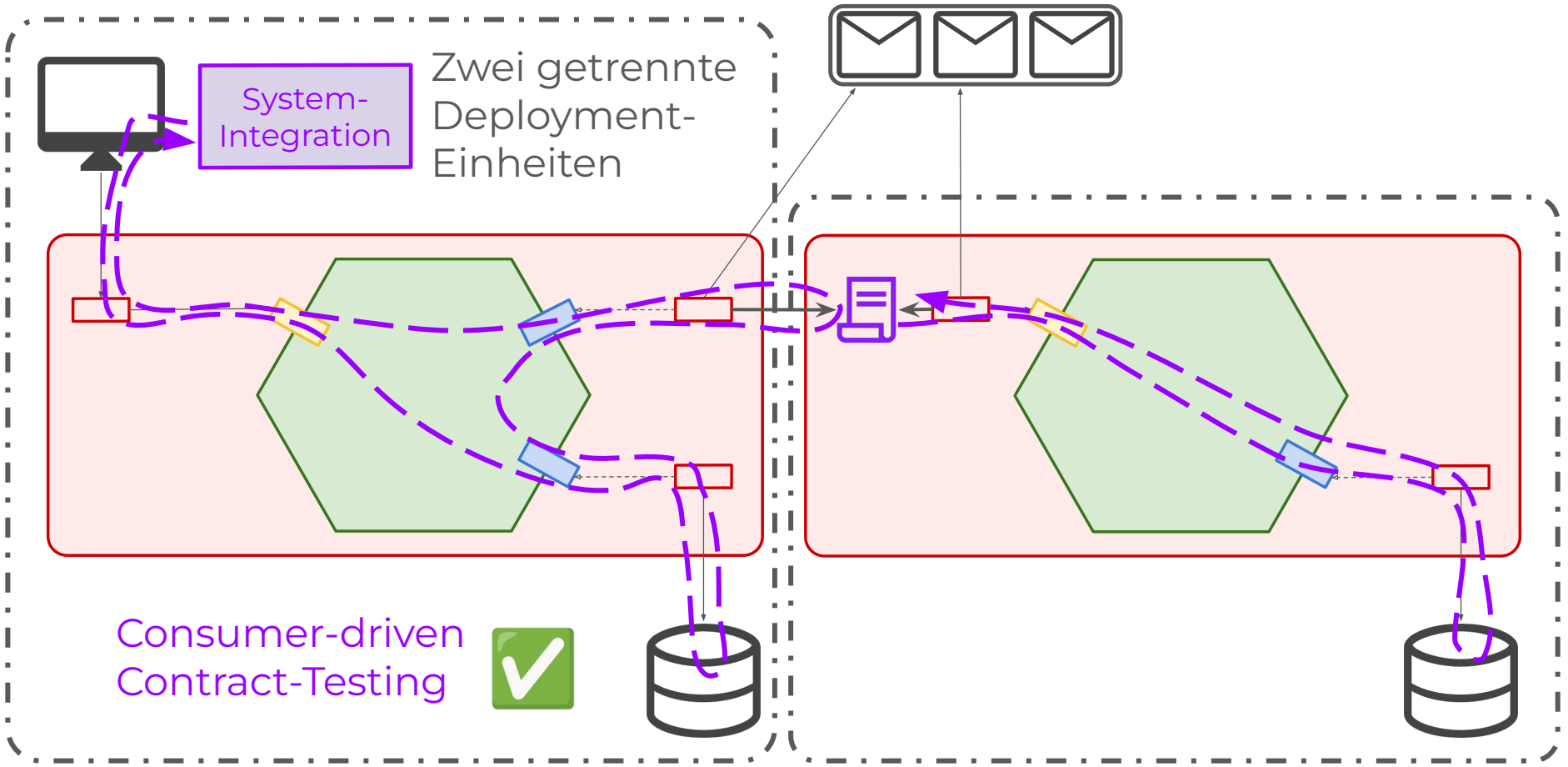




Eine gemeinsame Deployment-Einheit




E2E Test
Zwei getrennte Deployment-Einheiten


System-
Integration
Zwei getrennte
Deployment-
Einheiten
Consumer-driven
Contract-Testing

Irgendwas mit KI....

Takeaways

Takeaways

Hexagonale Architektur und gute Testbarkeit gehen Hand in Hand – die Architektur „leitet" einen zur richtigen Teststrategie.

Khorikovs 4 Pillars sind der ehrlichste Maßstab für Testqualität – Trade-offs sind unvermeidbar, aber bewusste Entscheidungen sind möglich.

Cockburns Reihenfolge: Vom Driving Port nach innen, dann nach außen.

Testdaten sind Architektur – Builder, Testcontainers und (wenn nötig) synthetische Daten aus Produktion.

Fakes statt Mocks, wo immer es geht.

Architektur-Tests automatisieren die Einhaltung der Hexagon-Regeln.

Mutation Score statt Line Coverage. Risiko- und Feature-Abdeckung berücksichtigen.

Contract Tests > E2E-Tests für Service-Integration.

Ich biete Trainings & Workshops zu Testing, Architektur und Craftmanship an:

fourteen-it.de

oose.

LinkedIn

www.fourteen-it.de

info@fourteen-it.de

andreas-juergensen