andrena OBJECTS

Pokale oder Pyramiden – brauchen wir noch Unit-Tests?

JavaForumStuttgart, 13.07.2023

andrena OBJECTS

Jacques Marco Jung

Lars Alvincz

In Kundenprojekten hört man häufig…

„Unit-Tests erschweren Refactorings“

„Bei Unit-Tests muss ich immer so viel mocken“

„Integrations-tests sind einfacher zu schreiben“

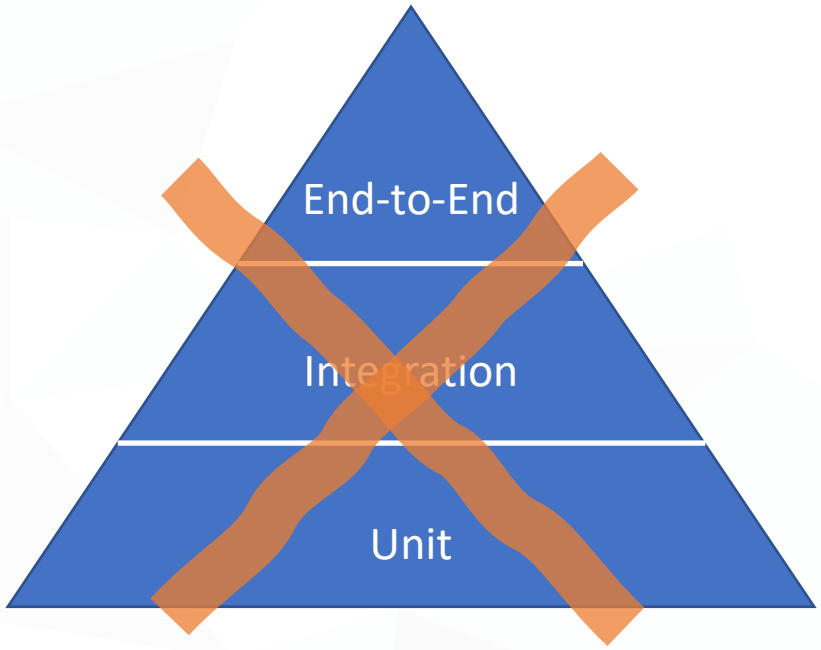
„Integrations-Tests ermöglichen Wartbarkeit“

„Integrations-tests decken eh alles ab“

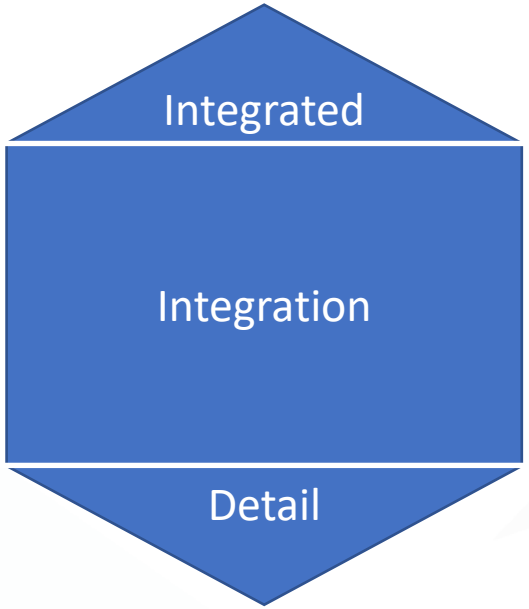
andrena OBJECTS

Modelle und Definitionen

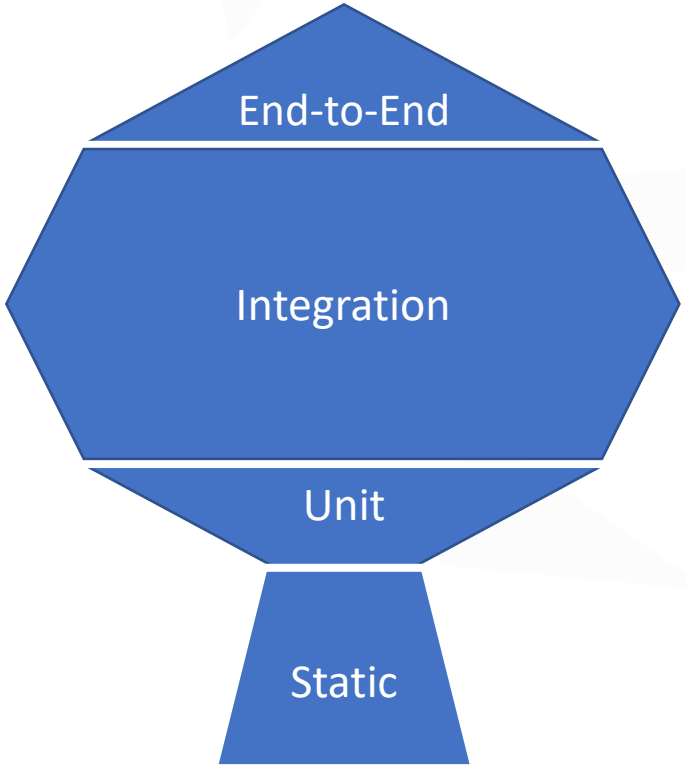
Pyramide



Bienenwabe



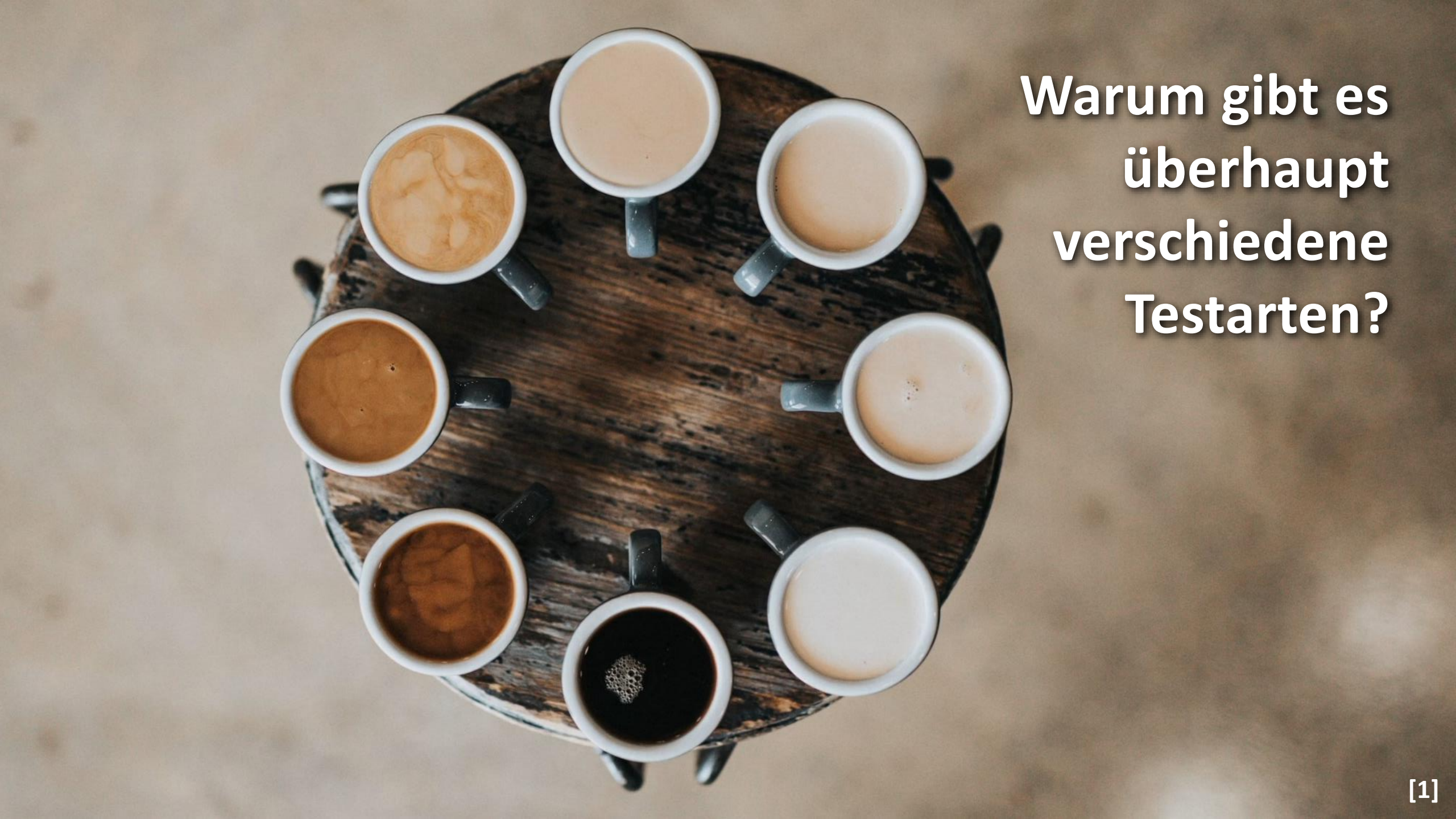
Pokal



?

Brauchen wir noch so viele Unit-Tests?

Quellen:
https://martinfowler.com/bliki/TestPyramid.html
https://engineering.atspotify.com/2018/01/testing-of-microservices/
https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications



Warum gibt es überhaupt verschiedene Testarten?

[1]

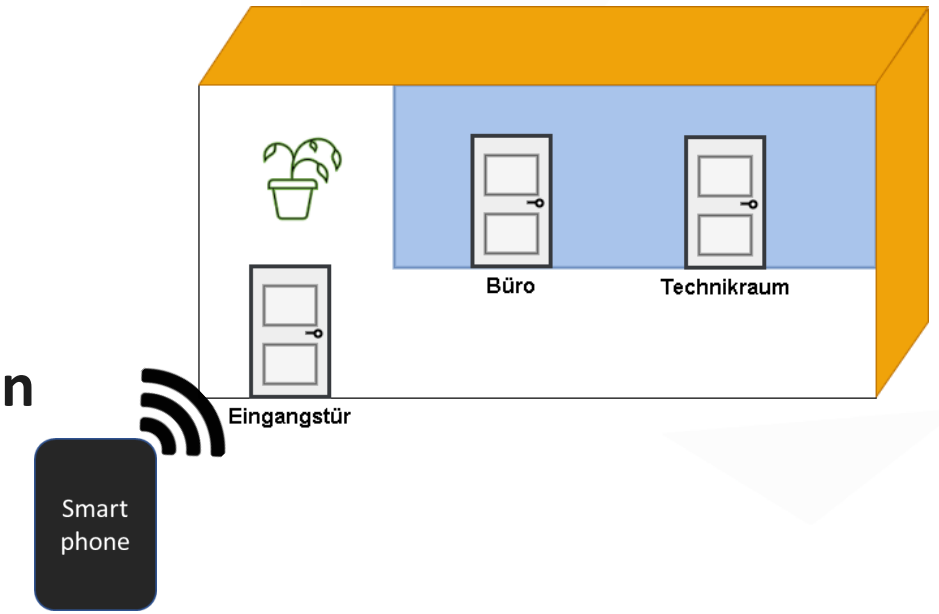
Unser Fallbeispiel:

Zugangs-kontrolle

[2]

andrena OBJECTS

Zugangskontrolle

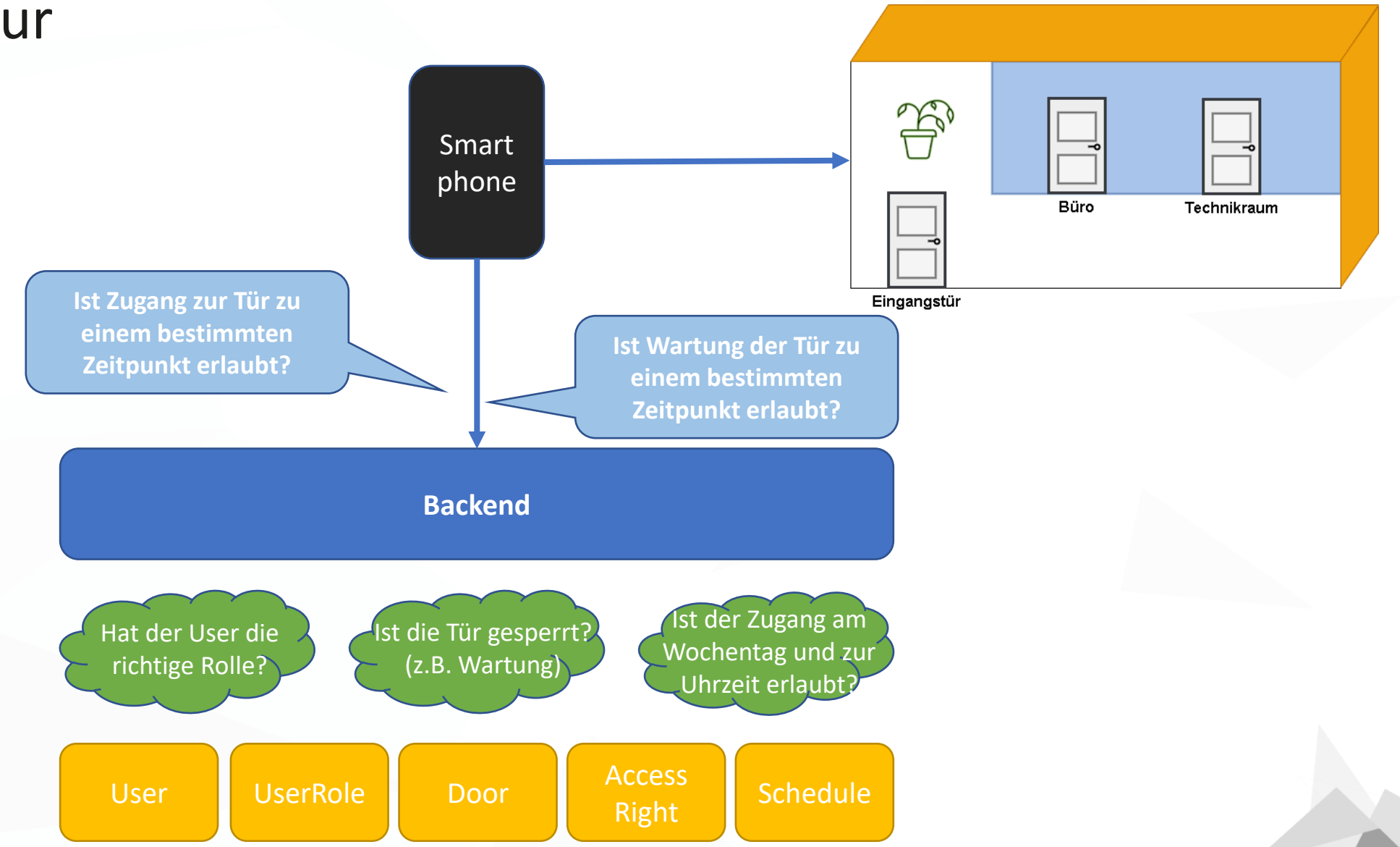


Fachlichkeit:

- **Elektronische Schlösser mit der Handy-App öffnen**
- **Use Cases**: Tür öffnen, Wartungszugriff
- **Nutzerrollen**:
  - Manager: immer öffnen, keinen Wartungszugriff
  - Angestellte: öffnen nach Zeitplan, keinen Wartungszugriff
  - Maintainer: öffnen/Wartung nach Zeitplan

andrena OBJECTS

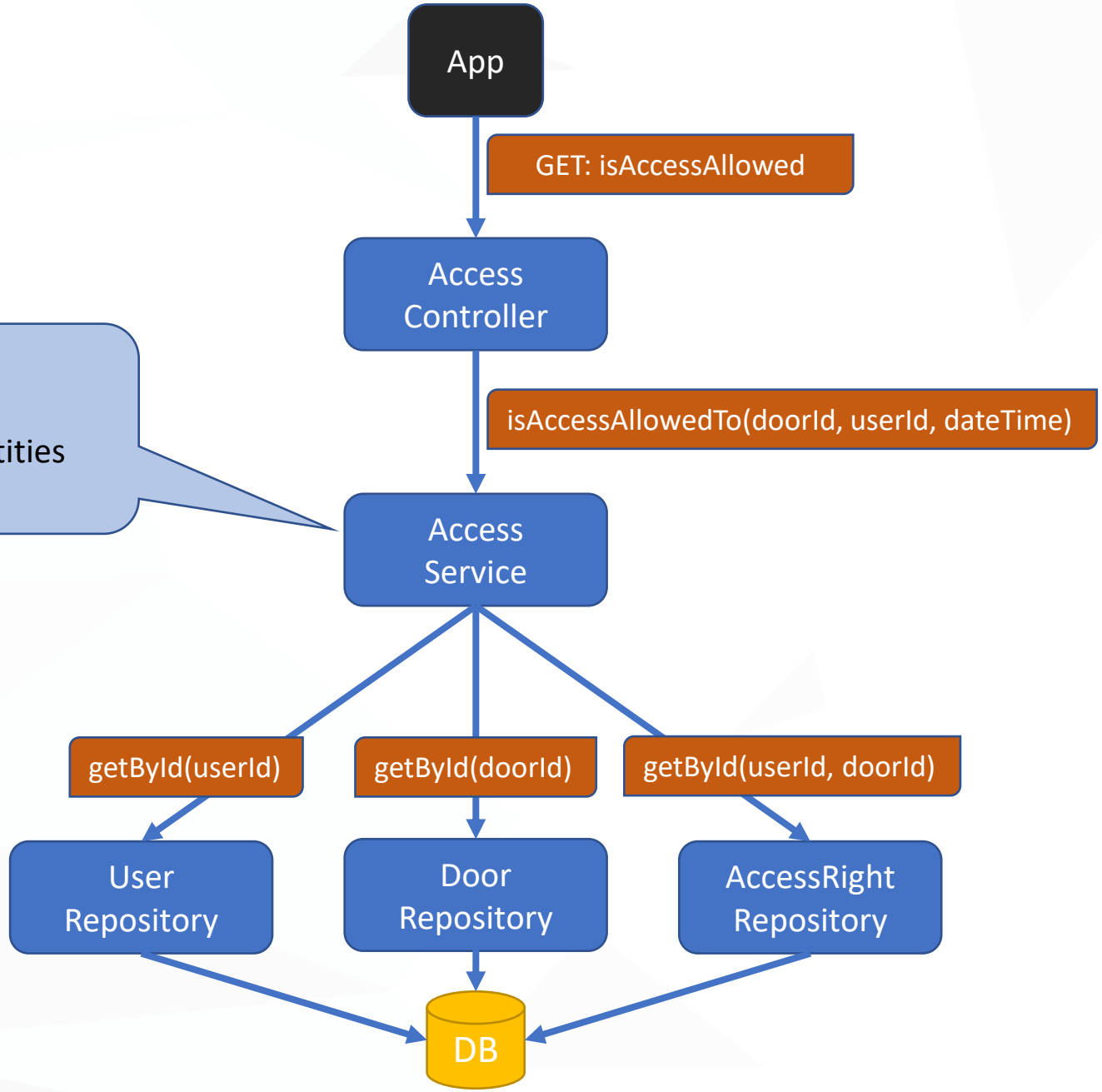
Die Architektur



andrena OBJECTS

Die Umsetzung

Logik im Service:
- Holt alle Entities
- Holt Informationen aus den Entities
- Ermittelt, ob Zugriff erlaubt ist



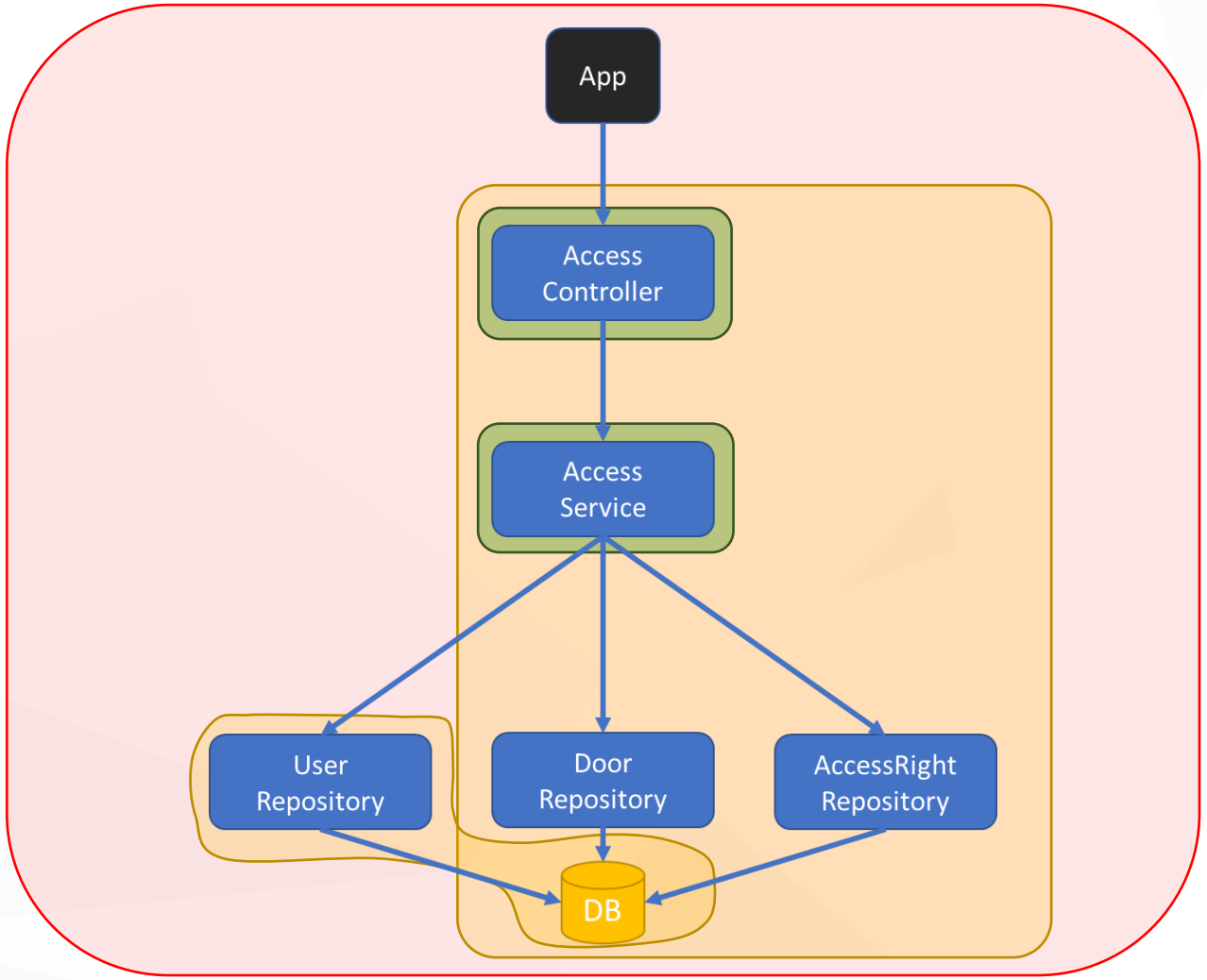
andrena OBJECTS

Wie können wir testen?

Gesamtanwendung als Black Box

Service mit Externen Systemem testen

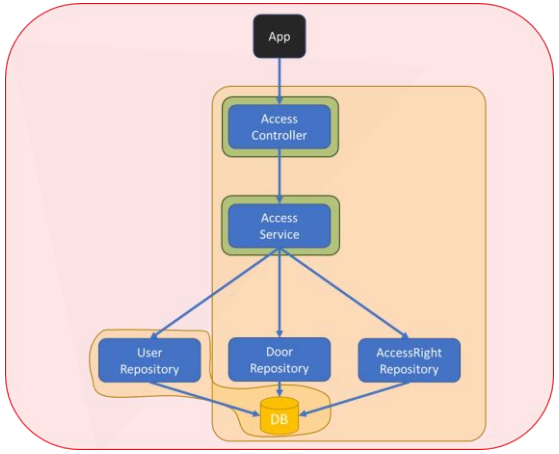
Einzelne/Mehrere Klassen isoliert testen



andrena OBJECTS

Eine Definition

Ziel: schnelles & häufiges Feedback
- ➢ Tests sollten schnell laufen
- ➢ Hohe Abdeckung durch Unit Tests



	Betrachteter Umfang	Testdaten	Laufzeit	Fehlersuche
End-to-End-Tests	Gesamte Anwendung und alle externe Abhängigkeiten	UI-Interaktion, JSON, DB-Einträge, …	langsam	schlecht
Integrationstest	Teile der Anwendung und externe Abhängigkeiten	Methodenparameter, DB-Einträge, …	langsam	eher schlecht
Unit-Test	eine/mehrere Klassen, keine externen Abhängigkeiten, ggf. Mocks	Methodenparameter	schnell	sehr gut

andrena OBJECTS

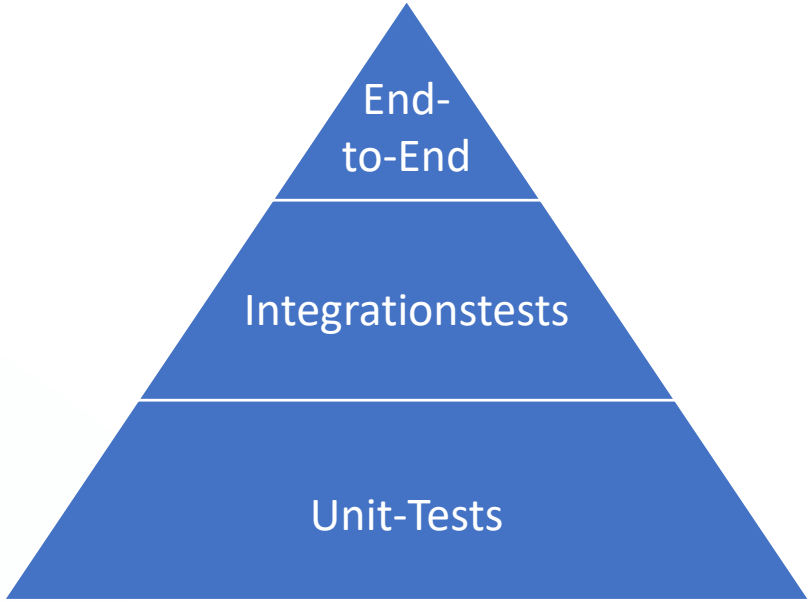
Effizientes Testen mit der Testpyramide

Fokus

End-to-End-Tests Gesamtsystem funktioniert (exemplarisch)

Integrationstest Integration funktioniert (Datenfluss)

Unit-Test Logik funktioniert, alle Fälle betrachten



betrachteter Ausschnitt

➢Aufwand, Dauer

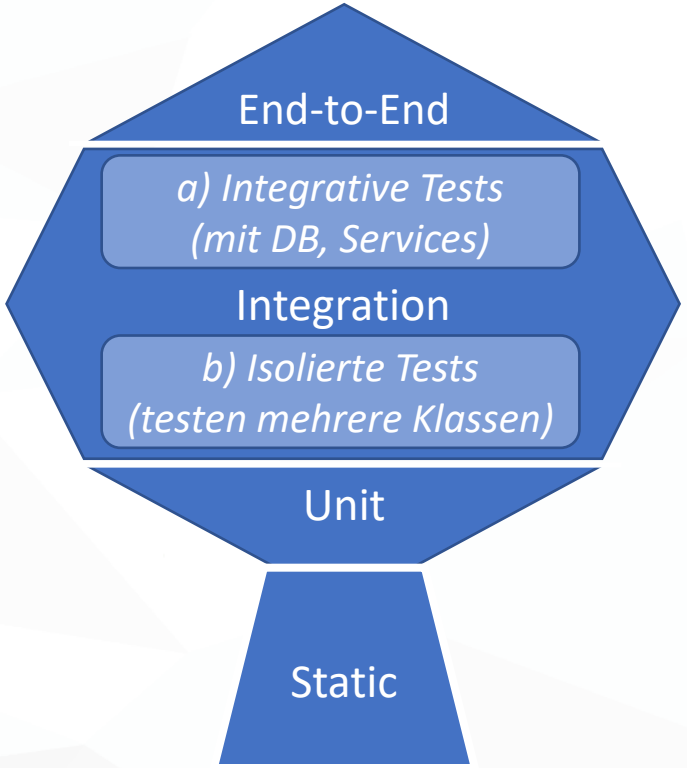
Wie sieht es mit dem Pokal aus?

[3]

andrena OBJECTS

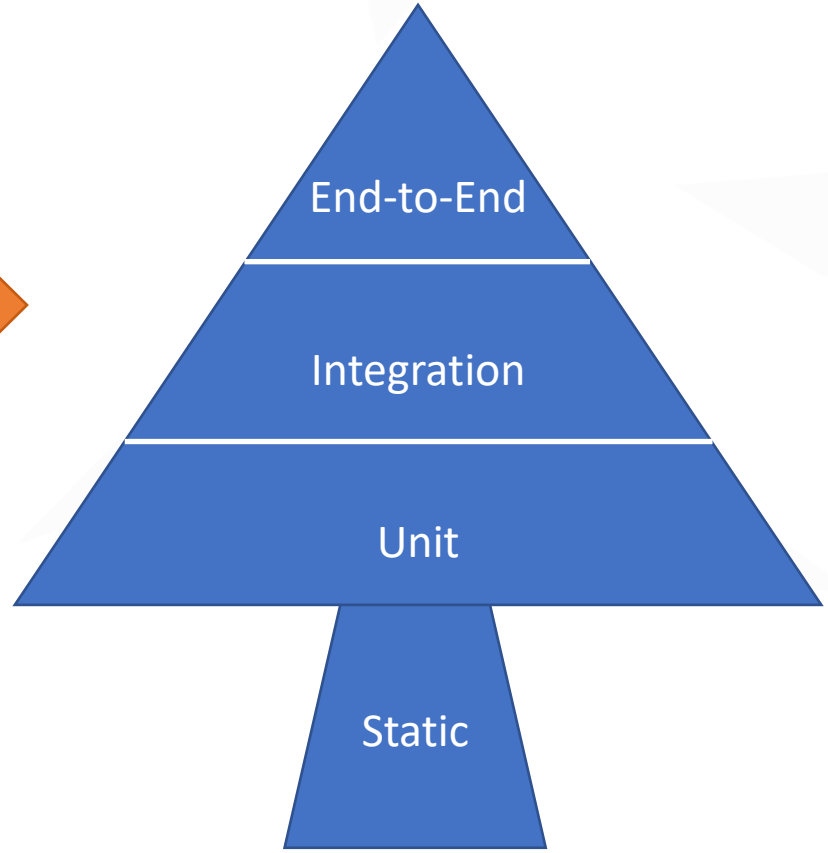
Test-Pokal

Pokal



Verschieben der isolierten Tests (b) zur Testkategorie „Unit“

Pokal goes Pyramide?



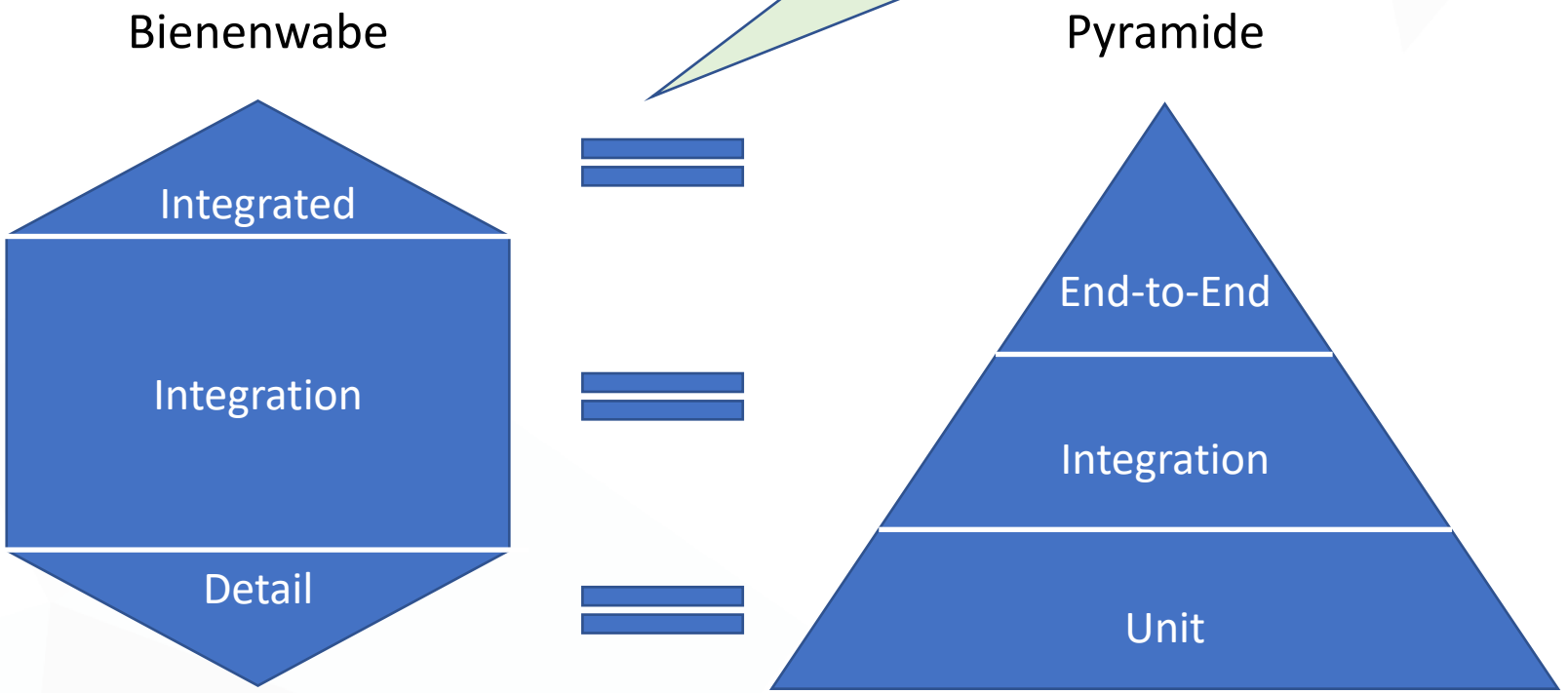
Worin unterscheidet sich die Bienenwabe?

[4]

andrena OBJECTS

Modelle und Definitionen

Testkategorien von Bienenwabe und Testpyramide entsprechen einander. Jedoch empfiehlt die Bienenwabe, den Fokus auf Integrationstests zu legen.

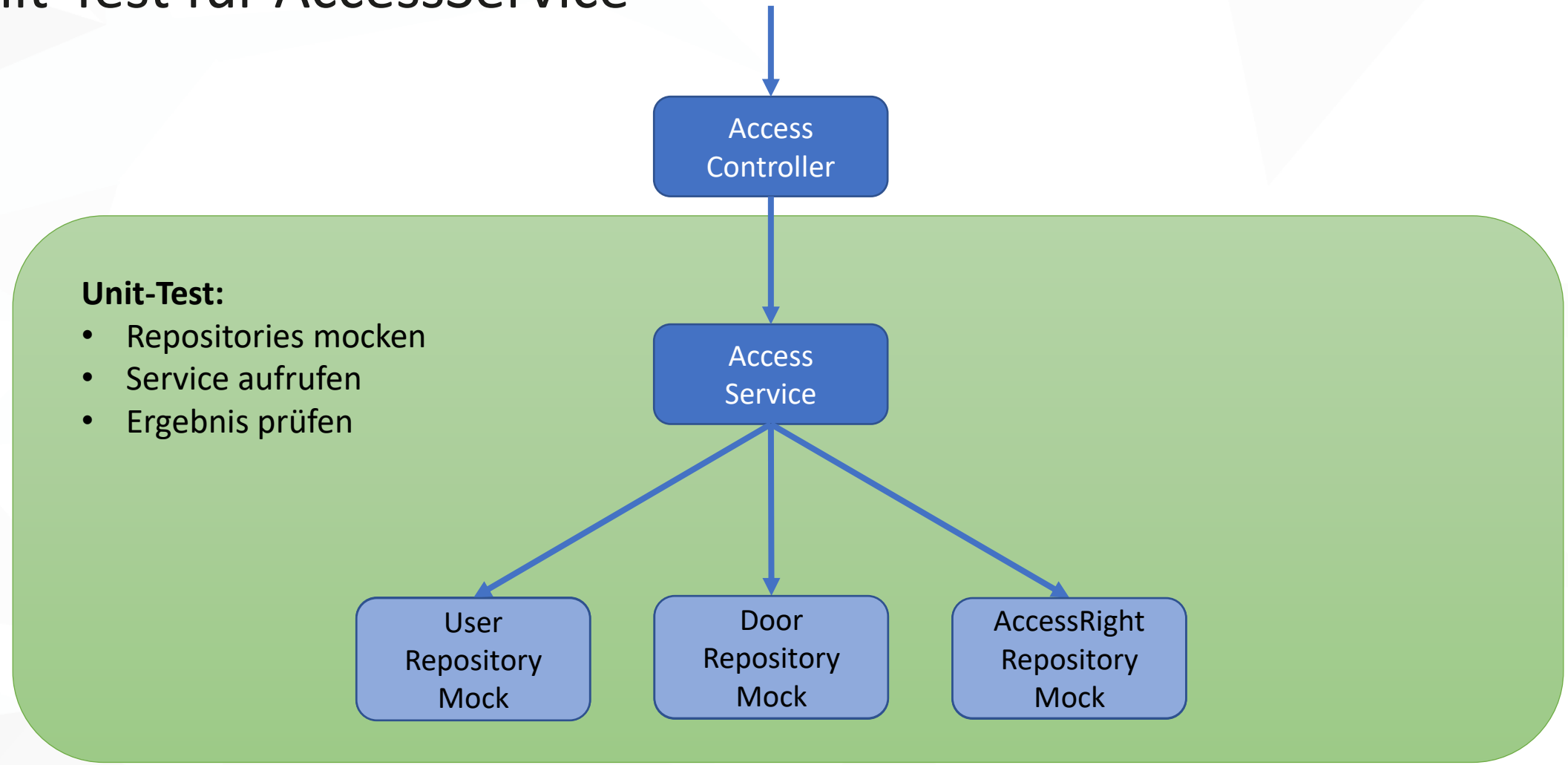



1811
Was bedeutet das für unser Beispiel?
Wie sehen Unit- und Integrations-tests beim Access-Service aus?

[2]

andrena OBJECTS

Unit-Test für AccessService



Unit-Test:

- Repositories mocken
- Service aufrufen
- Ergebnis prüfen

Codebeispiel Unit-Test (3 Mocks)


```kotlin
@ExtendWith(MockKExtension::class)
internal class AccessServiceTest {
    @MockK
    private lateinit var doorRepo: DoorRepository
    @MockK
    private lateinit var userRepo: UserRepository
    @MockK
    private lateinit var accessRightRepo: AccessRightRepository

    @InjectMockKs
    private lateinit var service: AccessService

    @BeforeEach
    internal fun setUp() {
        every { doorRepo.findItById(doorId) } returns door
        every { userRepo.findItById(userId) } returns User( userId: 1,UserRole.REGULAR)
        every { accessRightRepo.findByUserIdAndDoorId(userId, doorId) } returns null
    }

    @Test
    internal fun isAccessAllowedTo_doorDoesNotExists_ReturnsFalse() {
        every { doorRepo.findItById(any()) } returns null
        val isAccessAllowed :Boolean  = service.isAccessAllowedTo(doorId, userId, LocalDateTime.now())
        assertThat(isAccessAllowed).isFalse
    }
```

Was testen wir?

- Laden der Daten (Tür/User nicht gefunden)
- Umgang mit gesperrten Türen
- Verschiedene Rollen (Angestellte, Maintainer, …)
- Zeitpläne für Zugriff (Wochentage, Uhrzeiten)

➢ ca. 30 Testfälle

Codebeispiel Unit-Test (4 Mocks)


```
22  @ExtendWith(MockKExtension::class)
23  internal class AccessServiceTest {
24      @MockK
25      private lateinit var doorRepo: DoorRepository
26      @MockK
27      private lateinit var userRepo: UserRepository
28      @MockK
29      private lateinit var accessRightRepo: AccessRightRepository
30
31      @InjectMockKs
32      private lateinit var service: AccessService
33
34      @BeforeEach
35      internal fun setUp() {
36          every { doorRepo.findItById(doorId) } returns door
37          every { userRepo.findItById(userId) } returns User( userId: 1,UserRole.REGULAR)
38          every { accessRightRepo.findByUserIdAndDoorId(userId, doorId) } returns null
39      }
40
41      @Test
42      internal fun isAccessAllowedTo_doorDoesNotExists_ReturnsFalse() {
43          every { doorRepo.findItById(any()) } returns null
44          val isAccessAllowed : Boolean  = service.isAccessAllowedTo(doorId, userId, LocalDateTime.now())
45          assertThat(isAccessAllowed).isFalse
46      }
```

Einschätzung:

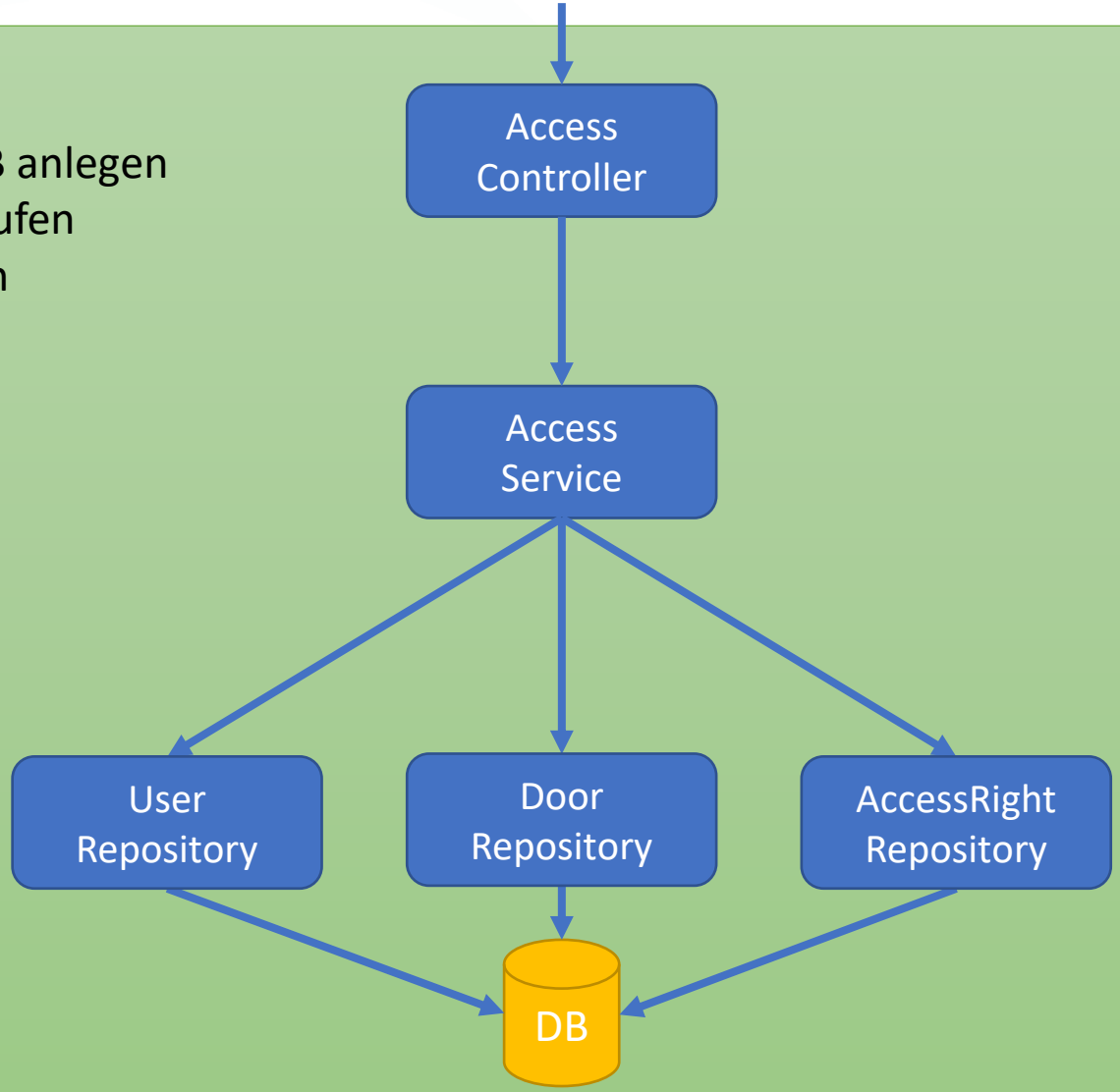
- Setup ist aufwändig
- Viele Mocks, viele Testdaten
- Viele verschiedene Testfälle

➢ **Wäre ein Integrationstest nicht einfacher?**

Integrationstest für AccessController

Integrationstest:

- Testdaten in DB anlegen
- Controller aufrufen
- Ergebnis prüfen



Codebeispiel Integrationstest

```kotlin
@SpringBootTest
@Transactional
internal class AccessControllerIntegrationTest {
    @Autowired
    private lateinit var entityManager: EntityManager
    @Autowired
    private lateinit var controller: AccessController

    @BeforeEach
    internal fun setUp() {
        entityManager.persist(door)
    }

    @Test
    internal fun isAccessAllowed_NoData_ReturnsFalse() {
        assertThat(controller.isAccessAllowed( doorId: 888,  userId: 999, LocalDateTime.now())).isFalse
    }

    @Test
    internal fun isAccessAllowed_RegularUserWithoutRights_ReturnsFalse() {
        entityManager.persist(accessRightNever)
        entityManager.persist(regularUser)
        assertThat(controller.isAccessAllowed(doorId, userId, LocalDateTime.now())).isFalse
    }
```

Was testen wir?

- Laden der Daten (Tür/User nicht gefunden)
- Umgang mit gesperrten Türen
- Verschiedene Rollen (Angestellte, Maintainer, …)
- Zeitpläne für Zugriff (Wochentage, Uhrzeiten)
- Mapping DTO

➢ **> 30 Testfälle**

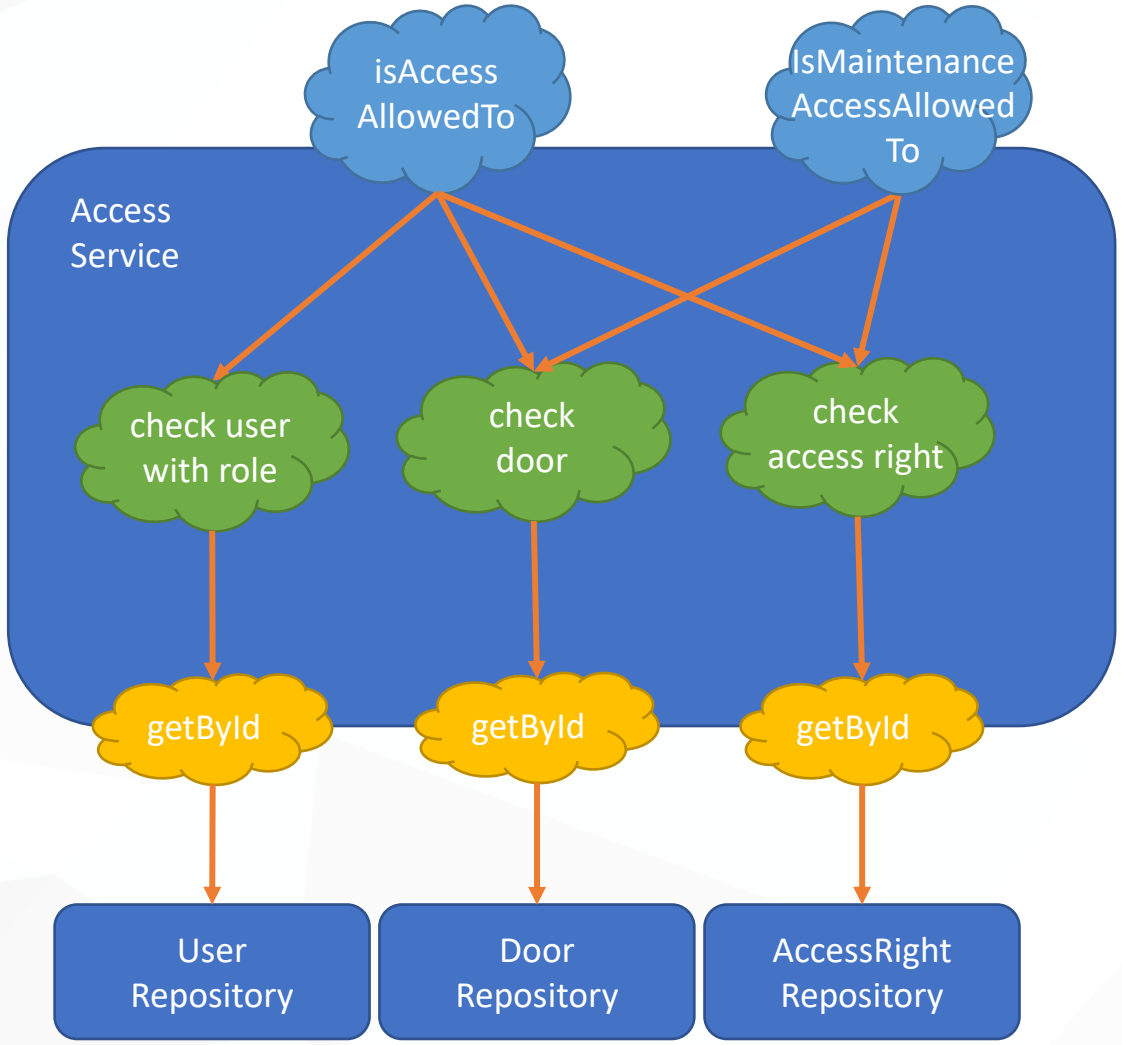
Codebeispiel Integrationstest

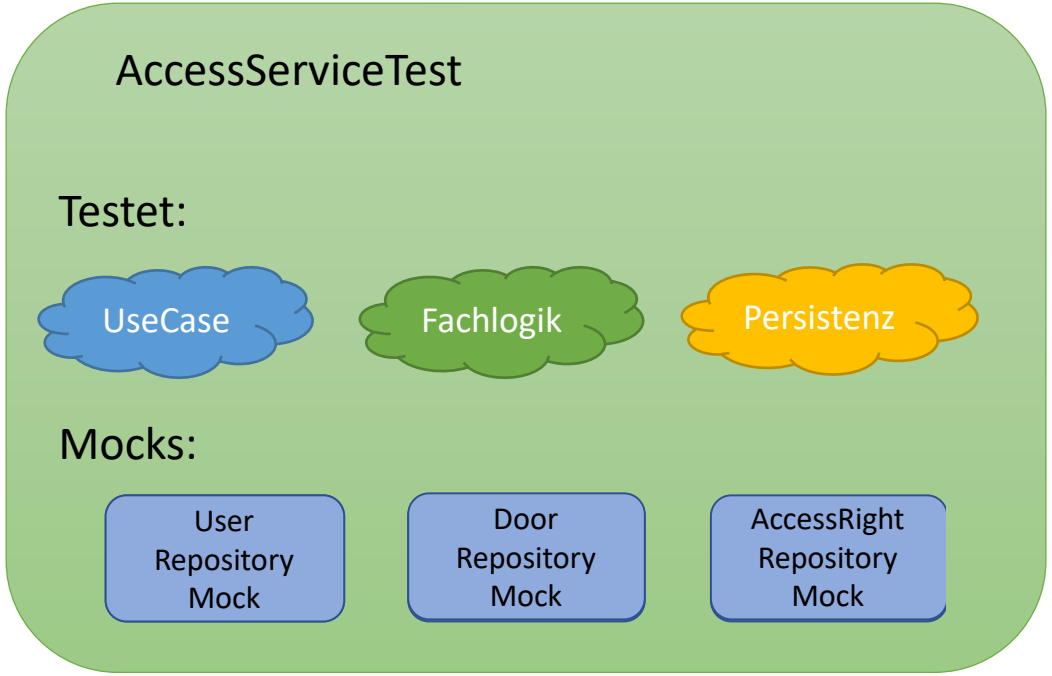

```
31  @SpringBootTest
32  @Transactional
33  internal class AccessControllerIntegrationTest {
34      @Autowired
35      private lateinit var entityManager: EntityManager
36      @Autowired
37      private lateinit var controller: AccessController
38
39      @BeforeEach
40      internal fun setUp() {
41          entityManager.persist(door)
42      }
43
44      @Test
45      internal fun isAccessAllowed_NoData_ReturnsFalse() {
46          assertThat(controller.isAccessAllowed( doorId: 888,  userId: 999, LocalDateTime.now())).isFalse
47      }
48
49      @Test
50      internal fun isAccessAllowed_RegularUserWithoutRights_ReturnsFalse() {
51          entityManager.persist(accessRightNever)
52          entityManager.persist(regularUser)
53          assertThat(controller.isAccessAllowed(doorId, userId, LocalDateTime.now())).isFalse
54      }
```

Das Setup ist einfacher.
Aber wollen wir so
alle Konstellationen testen?
(Laufzeit, Testdaten)


Warum haben wir eigentlich
so viele Testfälle?

Viele Zuständigkeiten in einem Service erhöhen den Test-Aufwand



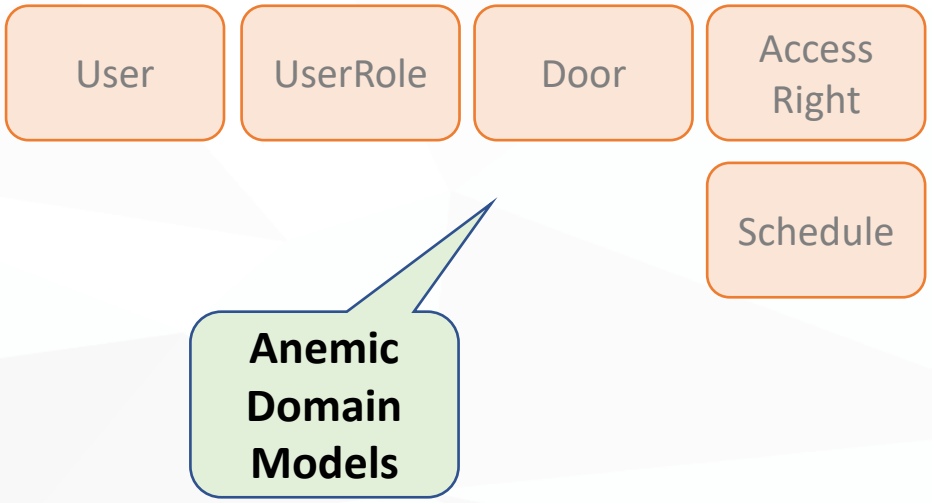


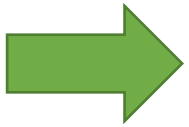

Lösung:
Abstrak-
tionen
finden
und
Zustän-
digkeiten
verteilen

[6]

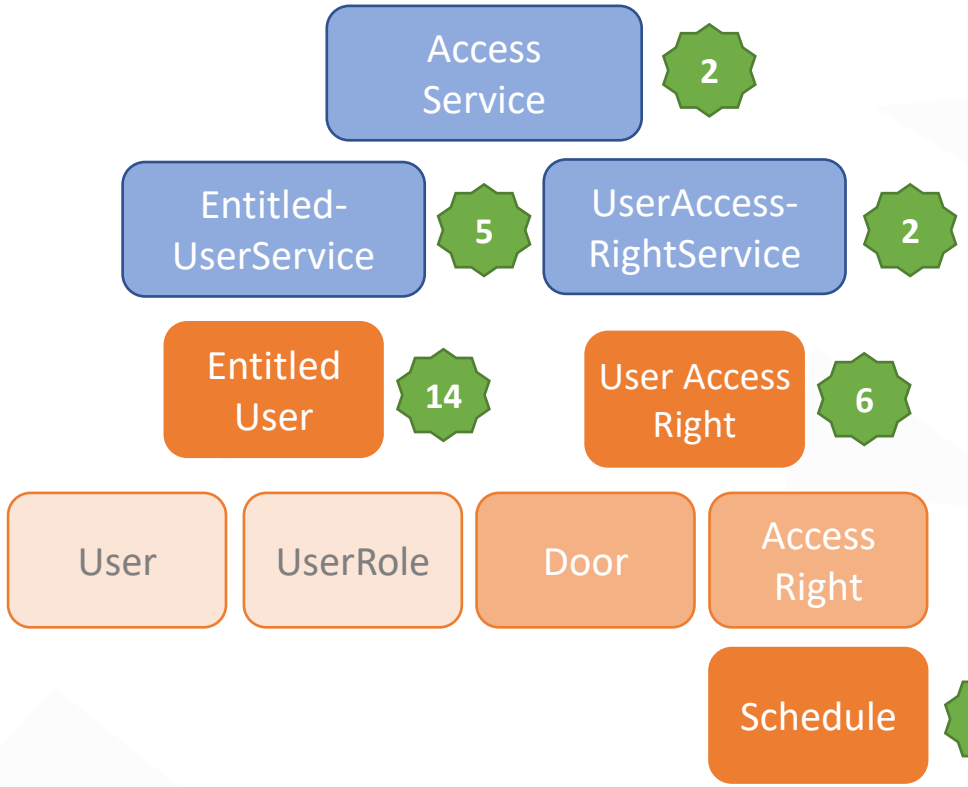
Abstraktionen finden und Zuständigkeiten verteilen

Ursprünglicher Schnitt



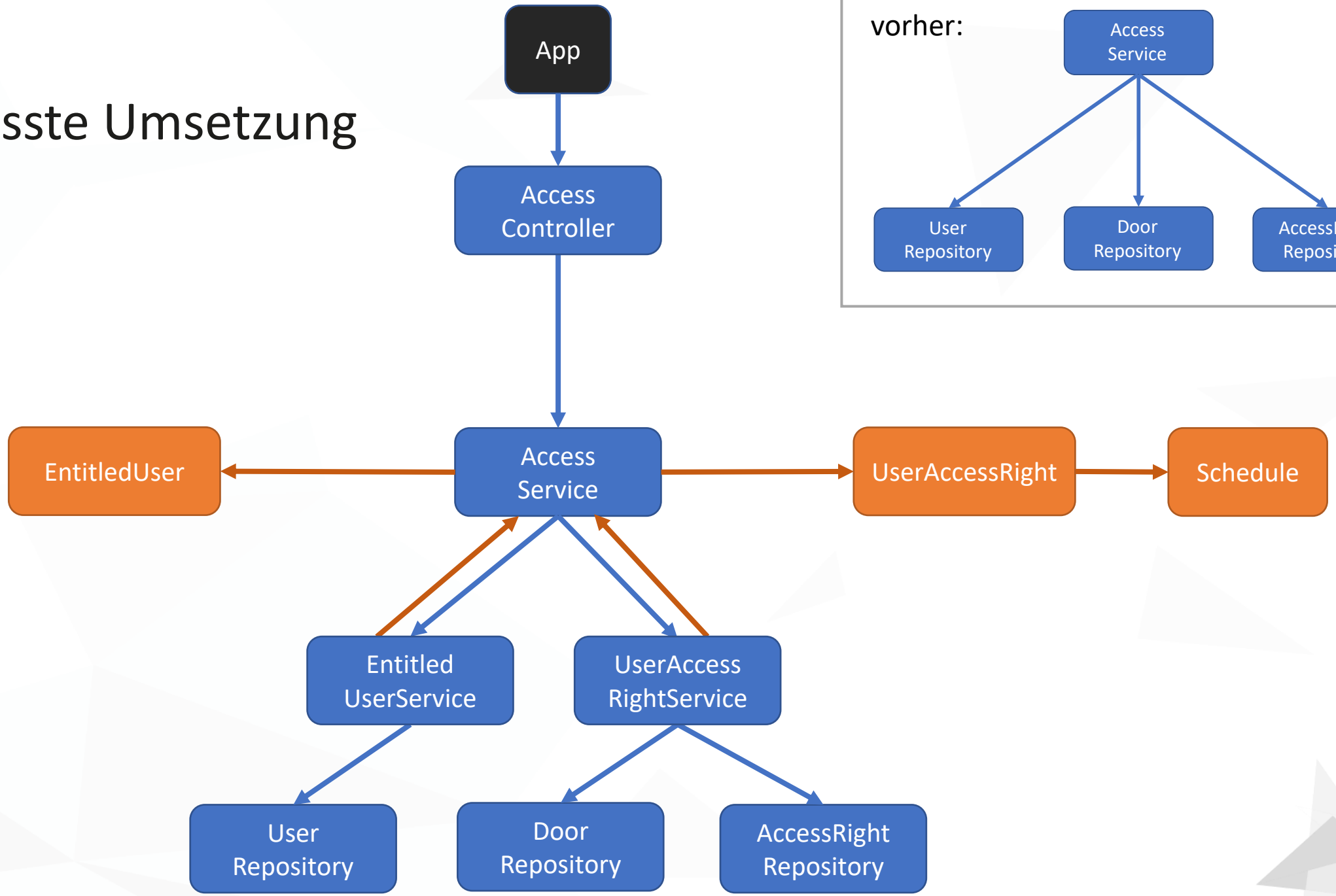


Neuer Schnitt



andrena OBJECTS

Angepasste Umsetzung

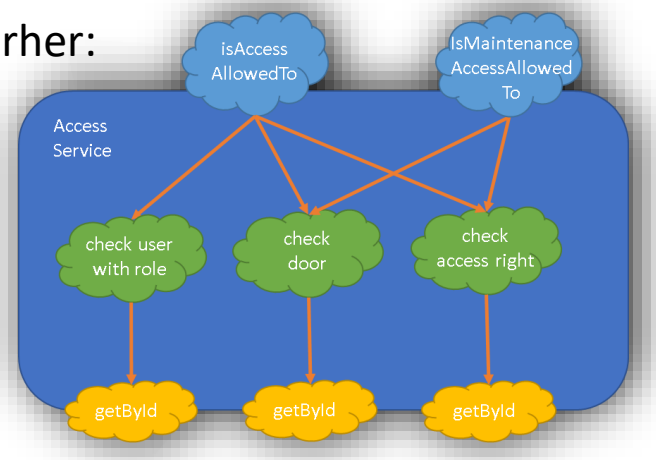


vorher:

andrena OBJECTS

Logik fachlich aufgeteilt und gekapselt

vorher:



Logik in Domain-Objekten:
- Hält alle notwendigen Informationen
- Ermittelt, ob der Zugriff gerade erlaubt ist

Access Controller

isAccess AllowedTo

IsMaintenance AccessAllowed To

check user with role

EntitledUser

Access Service

check access right

UserAccessRight

check access

Schedule

check door

Entitled UserService

load

UserAccess RightService

load

load

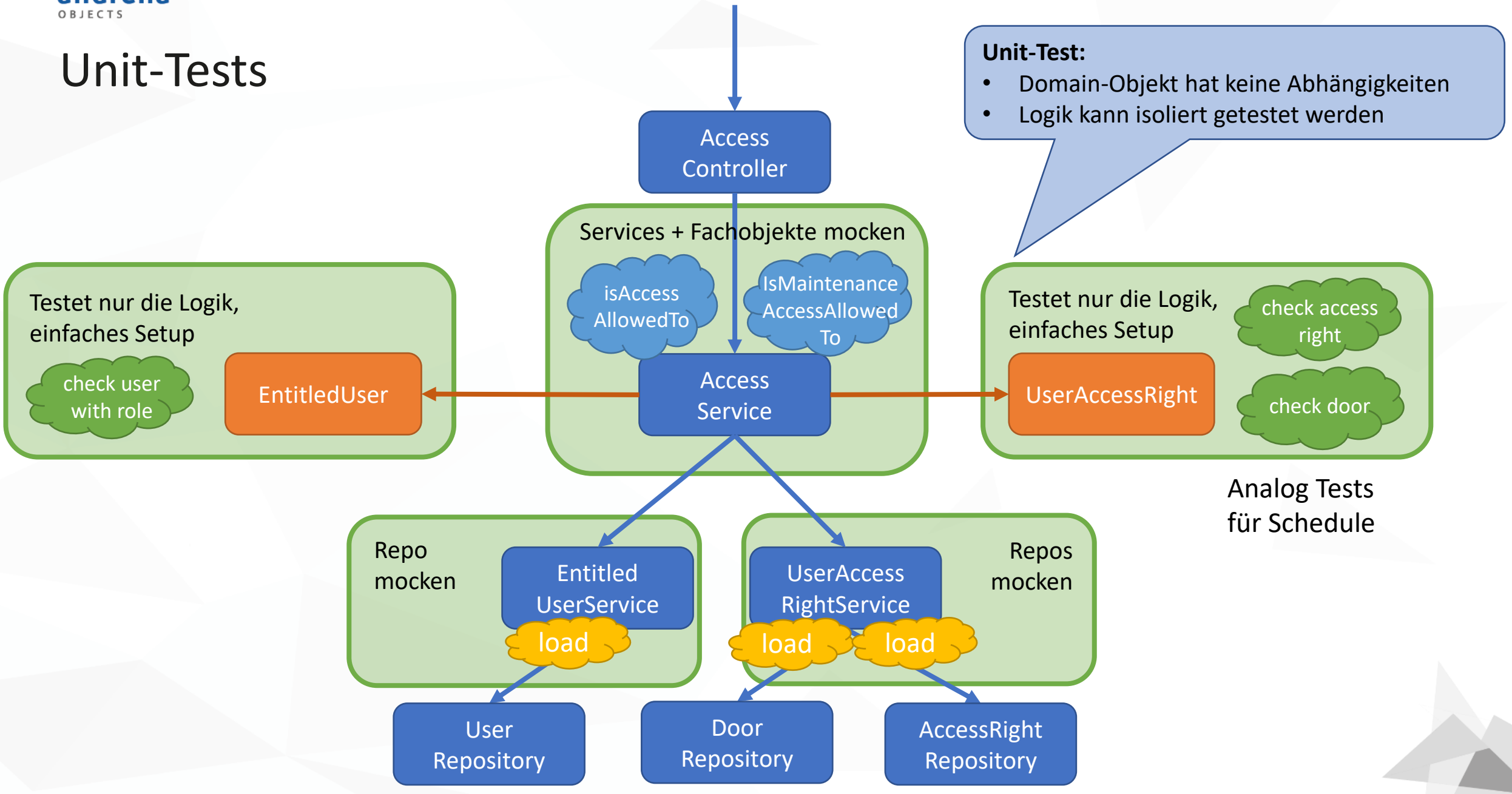
User Repository

Door Repository

AccessRight Repository

andrena OBJECTS

Unit-Tests



Zur Erinnerung: der alte Unit-Test (3 Mocks)


```
22  @ExtendWith(MockKExtension::class)
23  internal class AccessServiceTest {
24      @MockK
25      private lateinit var doorRepo: DoorRepository
26      @MockK
27      private lateinit var userRepo: UserRepository
28      @MockK
29      private lateinit var accessRightRepo: AccessRightRepository
30
31      @InjectMockKs
32      private lateinit var service: AccessService
33
34      @BeforeEach
35      internal fun setUp() {
36          every { doorRepo.findItById(doorId) } returns door
37          every { userRepo.findItById(userId) } returns User( userId: 1,UserRole.REGULAR)
38          every { accessRightRepo.findByUserIdAndDoorId(userId, doorId) } returns null
39      }
40
41      @Test
42      internal fun isAccessAllowedTo_doorDoesNotExists_ReturnsFalse() {
43          every { doorRepo.findItById(any()) } returns null
44          val isAccessAllowed : Boolean  = service.isAccessAllowedTo(doorId, userId, LocalDateTime.now())
45          assertThat(isAccessAllowed).isFalse
46      }
```

30 Testfälle

andrena OBJECTS

Unit-Test für Schedule (0 Mocks)


```
15  internal class ScheduleTest {
16      @ParameterizedTest
17      @MethodSource("dayOfWeeks")
18      fun map(allowedDays: List<DayOfWeek>, actualDay: DayOfWeek, expected: Boolean) {
19          val schedule = Schedule( id: -1, LocalTime.MIN, LocalTime.MAX, allowedDays)
20          assertThat(schedule.isAccessAllowed(LocalDateTime.now().with(actualDay))).isEqualTo(expected)
21      }
22      @ParameterizedTest
23      @MethodSource("time")
24      fun map(allowedStart: LocalTime, allowedEnd: LocalTime, actualTime: LocalTime, expected: Boolean) {
25          val schedule = Schedule( id: -1, allowedStart, allowedEnd, DayOfWeek.values().toList())
26          assertThat(schedule.isAccessAllowed(LocalDateTime.now().with(actualTime))).isEqualTo(expected)
27      }
28      companion object {
29          @JvmStatic
30          fun time() : List<Arguments!> = listOf(
31              Arguments.of(AT_9_AM, AT_5_PM, AT_9_AM.minusSeconds( secondsToSubtract: 1), false),
32              Arguments.of(AT_9_AM, AT_5_PM, AT_9_AM, true),
33              Arguments.of(AT_9_AM, AT_5_PM, AT_5_PM, true),
34              Arguments.of(AT_9_AM, AT_5_PM, AT_5_PM.plusSeconds( secondstoAdd: 1), false),
35          )
36
37          @JvmStatic
38          fun dayOfWeeks() : List<Arguments!> = listOf(
39              Arguments.of(emptyList<DayOfWeek>(), DayOfWeek.MONDAY, false),
40              Arguments.of(DayOfWeek.values().toList(), DayOfWeek.MONDAY, true),
```

andrena OBJECTS

Unit-Test für EntitledUser (0 Mocks)


```
internal class EntitledUserTest {
    private lateinit var accessRight: UserAccessRight

    @Nested
    inner class ManagerUserTest {
        @Test
        internal fun hasAccess_passedAccessRightIsIgnored_ReturnsAlwaysTrue() {
            accessRight = UserAccessRights.neverAccess()
            assertThat(ManagerUser().hasAccess(accessRight, now)).isTrue
        }
        @Test
        internal fun hasMaintenanceAccess_passedAccessRightIsIgnored_ReturnsAlwaysFalse() {
            accessRight = UserAccessRights.alwaysAccess()
            assertThat(ManagerUser().hasMaintenanceAccess(accessRight, now)).isFalse
        }
    }

    @Nested
    inner class RegularUserTest {
        @Test
        internal fun hasAccess_hasCurrentlyNoAccess_ReturnsFalse() {
            accessRight = UserAccessRights.onlyAccessAt(now)
            assertThat(RegularUser().hasAccess(accessRight, now.plusSeconds( seconds: 1))).isFalse
        }
```

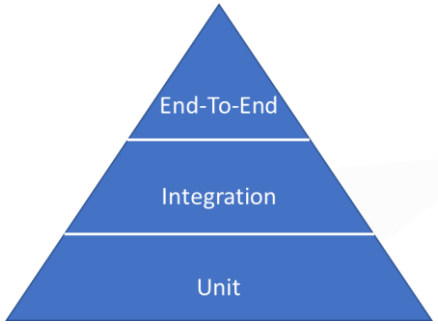
Unit-Test für UserAccessRightService (2 Mocks)


```
19  @ExtendWith(MockKExtension::class)
20  internal class UserAccessRightServiceTest {
21      @MockK
22      private lateinit var doorRepo: DoorRepository
23      @MockK
24      private lateinit var accessRightRepo: AccessRightRepository
25      private val accessRight = AccessRights.alwaysAccess()
26      @InjectMockKs
27      private lateinit var service: UserAccessRightService
28
29      @BeforeEach
30      internal fun setUp() {
31          every { accessRightRepo.findByUserIdAndDoorId(userId, doorId) } returns accessRight
32      }
33      @Test
34      internal fun getUserAccessRight_doorDoesNotExists_ReturnsNoRight() {
35          every { doorRepo.findItById(doorId) } returns null
36          val right : UserAccessRight = service.getUserAccessRight(doorId, userId)
37          assertThat(right).isSameAs(NO_RIGHT)
38      }
39      @Test
40      internal fun getUserAccessRight_ReturnsUserAccessRight() {
41          every { doorRepo.findItById(doorId) } returns door
42          val right : UserAccessRight = service.getUserAccessRight(doorId, userId)
43          assertThat(right).isEqualTo(UserAccessRight(accessRight, door))
44      }
45  }
```

Beobachtungen und Bewertung

Abstraktion fachlicher Konzepte (Schedule, UserAccessRight) erleichtert Unit-Test deutlich

- Die Logik lässt sich besser über Unit-Tests abtesten,
  wir brauchen weniger Integrationstests
- Testpyramide lässt sich leicht umsetzen



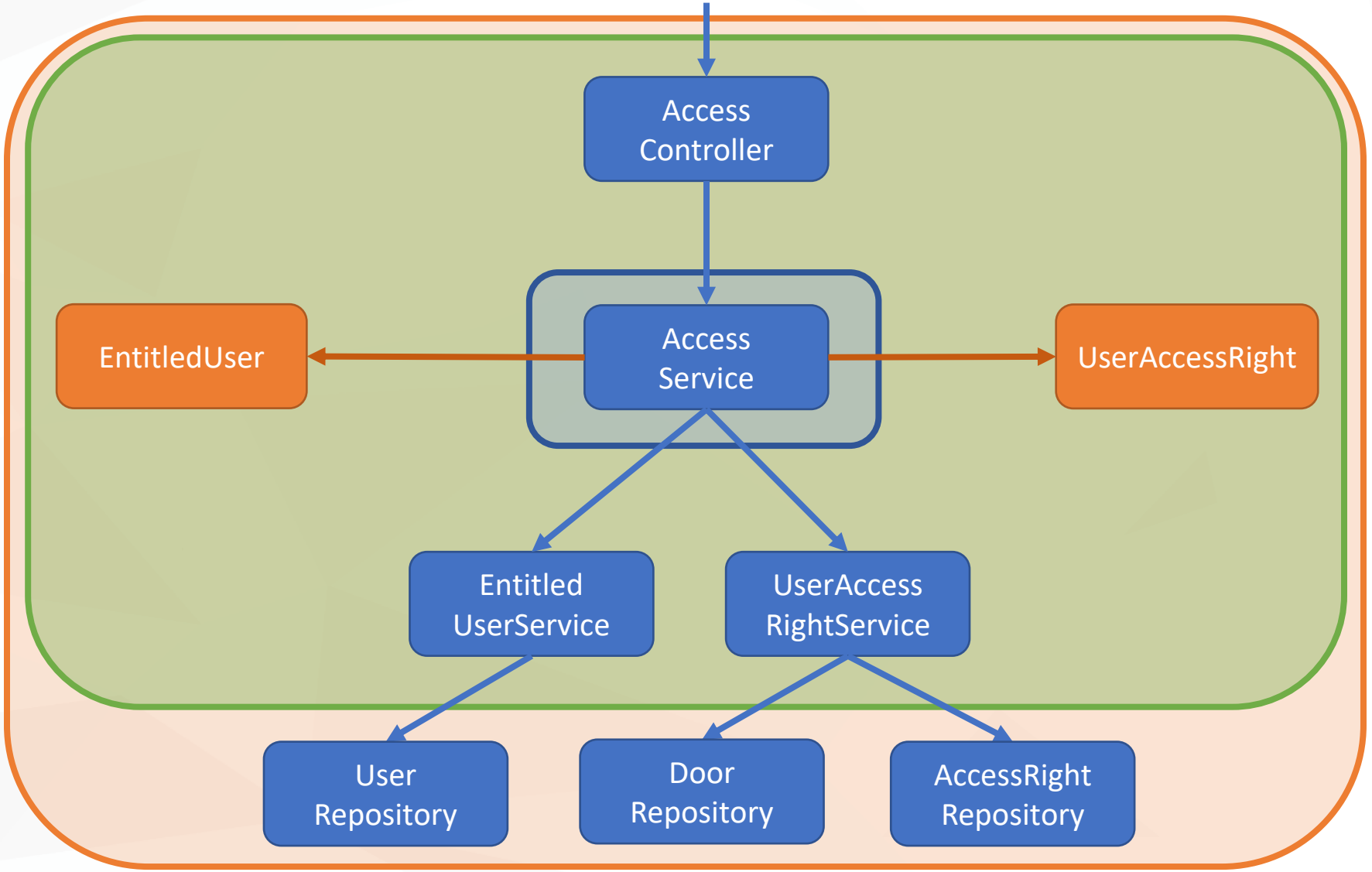
Abstraktion und Bündelung der Logik erhöhen Robustheit bei Refactorings

- Weniger Gründe, warum sich ein Test ändern muss
- Produktivcode ist modularer und dadurch besser verständlich und wartbar

**Ist es aufwändig, Unit-Tests zu schreiben bzw. zu warten?
Dann ist die Lösung nicht unbedingt einen Integrationstest zu schreiben.
Vielleicht kann man stattdessen den Produktiv-Code verbessern!**

[2]

Teststrategien für AccessService



https://martinfowler.com/bliki/UnitTest.html

Solitary Unit-Test:
- Services/Fachobjekte mocken

Sociable Unit-Test:
- Repositories mocken
- Controller aufrufen
- Ergebnis prüfen

Integrationstest:
- Daten in die DB eintragen
- Controller aufrufen
- Ergebnis prüfen

Solitary Unit-Test für AccessService mit Mocks (2+1 Mocks, 55 Zeilen)


```
26  @ExtendWith(MockKExtension::class)
27  internal class AccessServiceMockTest {
28      @MockK
29      private lateinit var userAccessRightService: UserAccessRightService
30      @MockK
31      private lateinit var entitledUserService: EntitledUserService
32      private val userAccessRight = UserAccessRights.alwaysAccess()
33      private var entitledUser = mockk<EntitledUser>()
34      @InjectMockKs
35      private lateinit var service: AccessService
36      @BeforeEach
37      internal fun setUp() {
38          every { userAccessRightService.getUserAccessRight(DOOR_ID, USER_ID) } returns userAccessRight
39          every { entitledUserService.getEntitledUser(USER_ID) } returns entitledUser
40      }
41      @AfterEach
42      internal fun tearDown() {
43          checkUnnecessaryStub(entitledUserService, userAccessRightService, entitledUser)
44      }
45      @Test
46      internal fun isAccessAllowedTo() {
47          every { entitledUser.hasAccess(userAccessRight, now) } returns true
48          assertThat(service.isAccessAllowedTo(DOOR_ID, USER_ID, now)).isTrue
49      }
50      @Test
51      internal fun isMaintenanceAccessAllowedTo() {...}
```

Sociable Unit-Test für AccessController (3 Mocks, 80 Zeilen)

andrena OBJECTS

```
28  @ExtendWith(MockKExtension::class)
29  internal class AccessControllerUnitTest {
30      @MockK
31      private lateinit var doorRepo: DoorRepository
32      @MockK
33      private lateinit var accessRightRepo: AccessRightRepository
34      @MockK
35      private lateinit var userRepository: UserRepository
36      private lateinit var userAccessRightService: UserAccessRightService
37      private lateinit var entitledUserService: EntitledUserService
38      private lateinit var service: AccessService
39      private lateinit var controller: AccessController
```

```
@BeforeEach
internal fun setUp() {
    userAccessRightService = UserAccessRightService(doorRepo, accessRightRepo)
    entitledUserService = EntitledUserService(userRepository)
    service = AccessService(userAccessRightService, entitledUserService)
    controller = AccessController(service)
    every { doorRepo.findItById(doorId) } returns door
```

```
50      @Test
51      internal fun isAccessAllowedTo_RegularUserWithAccess_ReturnsTrue() {
52          every { accessRightRepo.findByUserIdAndDoorId(userId, doorId) } returns AccessRights.alwaysAccess()
53          every { userRepository.findItById(userId) } returns User( id: -1, UserRole.REGULAR)
54
55          assertThat(controller.isAccessAllowed(doorId, userId, now)).isTrue
56      }
57
58      @Test
59      internal fun isAccessAllowedTo_RegularUserWithoutAccess_ReturnsFalse() {...}
```

Integrationtest für AccessController (75 Zeilen)


```
@SpringBootTest
@Transactional
internal class AccessControllerIntegrationTest {
    @Autowired
    private lateinit var entityManager: EntityManager
    @Autowired
    private lateinit var controller: AccessController
    @BeforeEach
    internal fun setUp() {
        entityManager.persist(door)
    }
    @Test
    internal fun isAccessAllowed_NoData_ReturnsFalse() {
        assertThat(controller.isAccessAllowed( doorId: 888,  userId: 999, LocalDateTime.now())).isFalse
    }
    @Test
    internal fun isAccessAllowed_RegularUserWithoutRights_ReturnsFalse() {...}
    @Test
    internal fun isAccessAllowed_RegularUserWithRights_ReturnsTrue() {
        entityManager.persist(accessRightAlways)
        entityManager.persist(regularUser)
        assertThat(controller.isAccessAllowed(doorId, userId, LocalDateTime.now())).isTrue
    }
```

```
private val accessRightAlways = AccessRight(
    id: 111, userId, doorId, listOf(Schedule( id: 222, LocalTime.MIN, LocalTime.MAX,
        DayOfWeek.values().toList()))
)
private val accessRightNever = AccessRight(
    id: 111, userId, doorId, listOf(Schedule( id: 222, LocalTime.MIN, LocalTime.MAX,
        emptyList()))
)
private val regularUser = User(userId, UserRole.REGULAR)
private val maintainerUser = User(userId, UserRole.MAINTAINER)
```

Beobachtung und Bewertung

	Solitary Unit-Test für den AccessService	Sociable Unit-Test für den AccessController	Integrationtest für den AccessController
Beobachtung	+ überprüft exakten Aufruf (doppelte Buchführung) + kurzes Setup – dupliziert die Umsetzung, fragil bei Refactorings	+ isoliert, läuft sehr schnell + gut für komplexe fachliche Szenarien – prüft Logik nur indirekt – umfangreicheres Setup – fragil bei Refactorings der Repositories	+ stellt integriertes Zusammenspiel sicher – prüft Logik nur indirekt – Datensetup nötig
Bewertung	kann sinnvoll sein, allerdings recht mechanisch	bietet hier wenig Mehrwert, falls Integrationstest vorhanden	sinnvoll um Zusammenspiel zu überprüfen, nicht für Korrektheit der Logik

Zusammengefasst (für AccessService)**:**

- Integrationstest ohnehin nötig um Zusammenspiel zu testen
- Ggf. könnte Unit-Test mit Mocks sinnvoll sein (bei komplexer Logik)

andrena OBJECTS

Fazit

Schnelles Feedback durch Tests ist **immens wichtig**

für wartbaren und korrekten Produktivcode

- ➢ Große Abdeckung durch Unit-Tests ermöglicht Refactorings und hält Code flexibel
- ➢ Testpyramide als gute Grundlage

Aufwand für Unit-Test zu hoch? (Setup/Wartung)

- ➢ Zu viele Zuständigkeiten? Produktivcode verbessern?
- ➢ Fachlichkeiten bündeln, Konzepte abstrahieren
- ➢ Code ist besser verständlich, wartbar & einfacher testbar

Service hat kaum Fachlogik,
ist eher Datendrehscheibe?

- ➢ Ggf. höherer Anteil an Integrationstests sinnvoll (Bienenwabe, Testpokal)

Testvorgehen sind kein Selbstzweck –
das Ziel ist schnelles Feedback durch schnelle Tests!

andrena OBJECTS

Bildnachweise

- [0] https://unsplash.com/de/fotos/gcDwzUGuUoI
- [1] https://unsplash.com/de/fotos/pMW4jzELQCw
- [2] https://unsplash.com/de/fotos/F_hQqE6uN30
- [3] https://unsplash.com/de/fotos/_XTY6lD8jgM
- [4] https://unsplash.com/de/fotos/hAYy2mFLjS8
- [5] https://unsplash.com/de/fotos/3UbsiRcrFV4
- [6] https://unsplash.com/de/fotos/fqkrXYMosT4