Functional Core für einen seiteneffektfreien Anwendungskern

Eine fiktive Geschichte um den eigentlichen Kern

Kai Schmidt
- Selbständig
- Software Architect und Entwickler

Thomas Ruhroth
- msg systems ag - Travel & Logistics
- Software Architect und Entwickler
- Lead IT Consultant

Rollen - Wie im wirklichen Leben

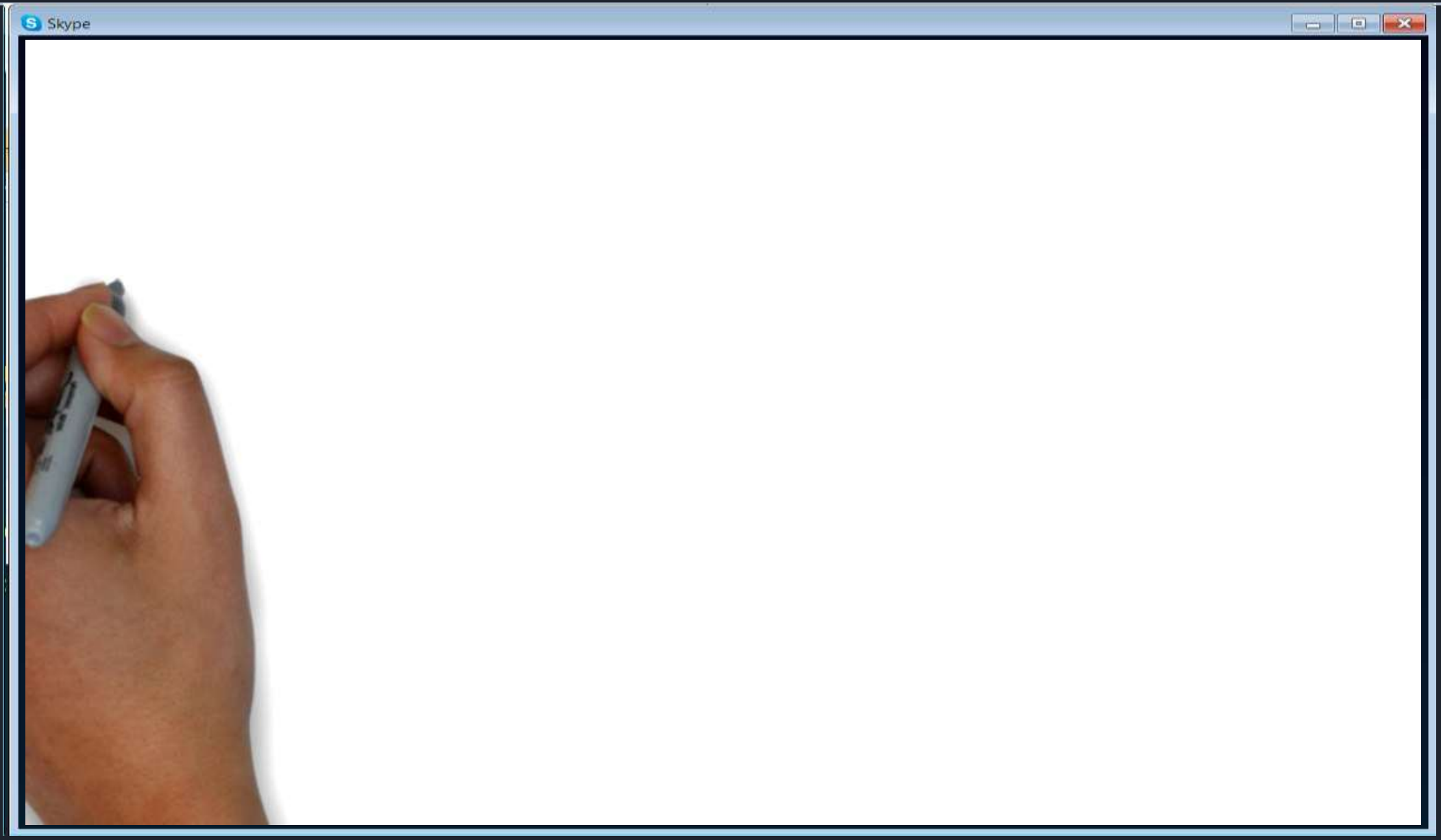
Entwickler

Der Auftrag
Das Problem

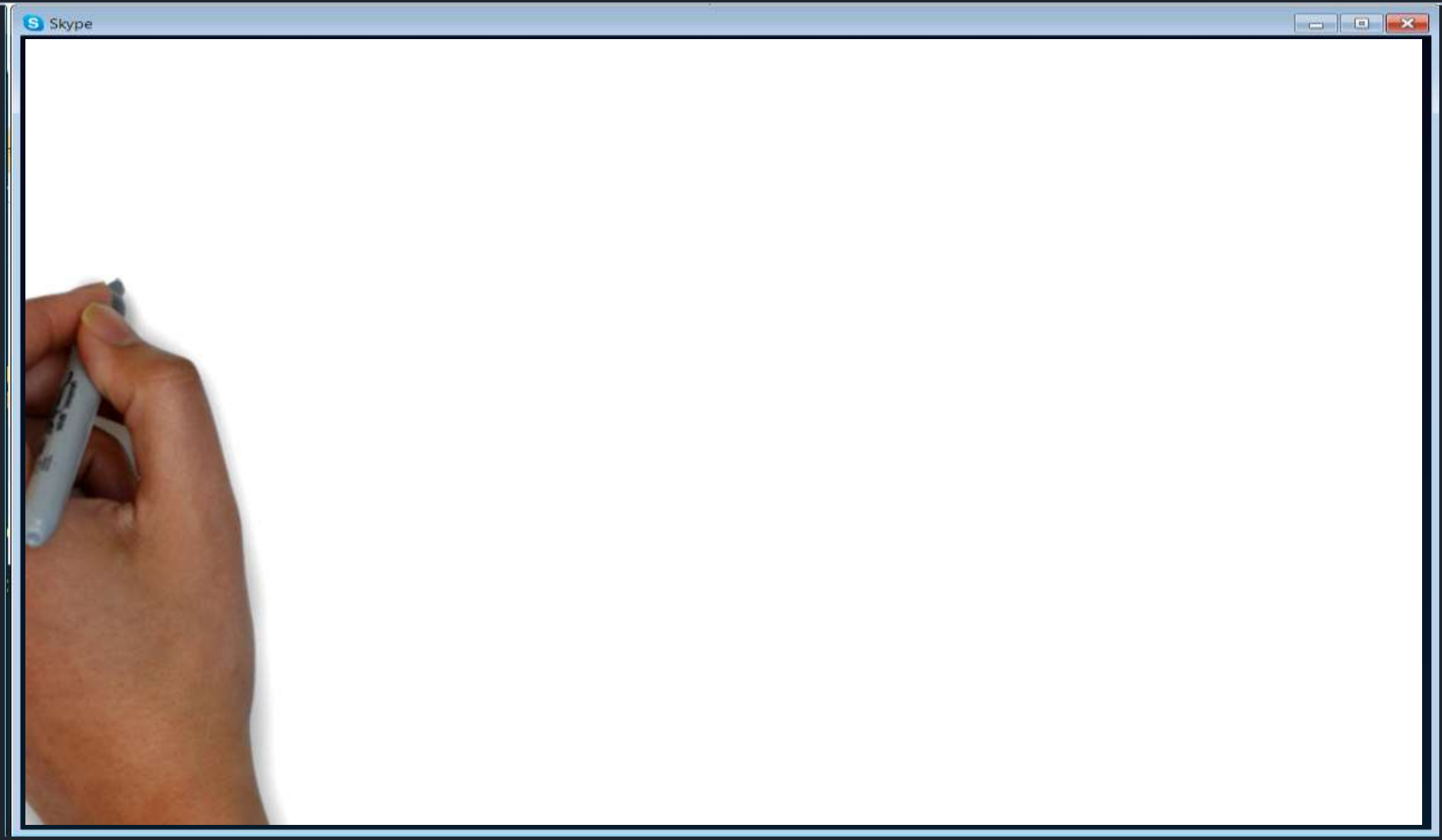
Architekt

Der Mentor
Die Lösung

Der Auftrag



Der Auftrag



An die Arbeit…

- Die Anforderungen:
  - Funktional:
    - Bewertungssystem für verschiedene Veranstaltung-/Konferenzsysteme
  - Nicht-Funktional:
    - Portierbarkeit
    - Saubere Schnittstelle
    - Gute Testbarkeit der Kernfunktionalität
    - Leichte Erweiterbarkeit der Nutzung

An die Arbeit…

- Nicht-Funktional:
  - Portierbarkeit
  - Saubere Schnittstelle
  - Gute Testbarkeit der Kernfunktionalität
  - Leichte Erweiterbarkeit der Nutzung

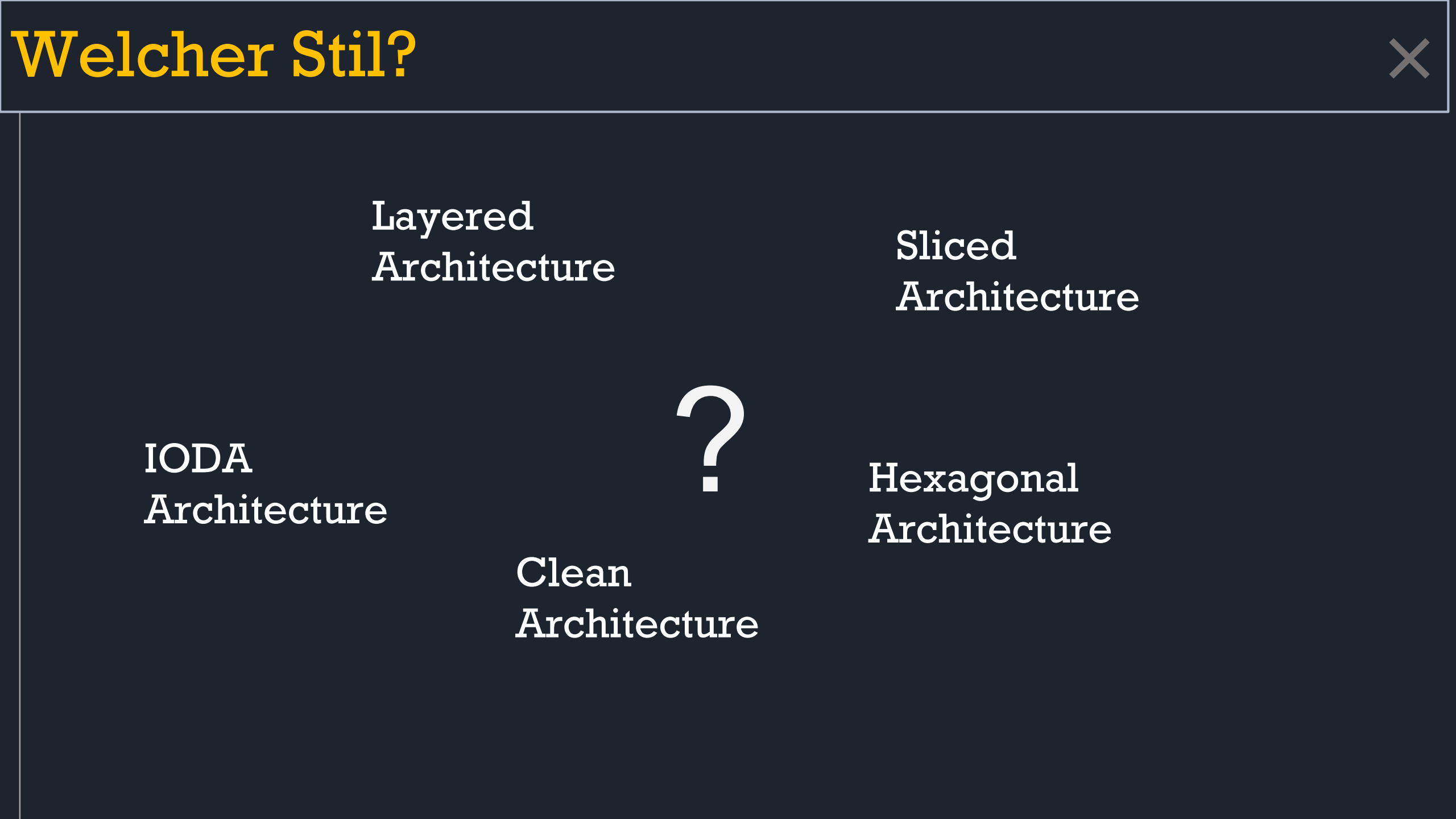
Layered Architecture	Adressiert nicht die architekturellen Eigenschaften die gefordert sind. Annahmen über die tieferliegenden Schichten. …

An die Arbeit…

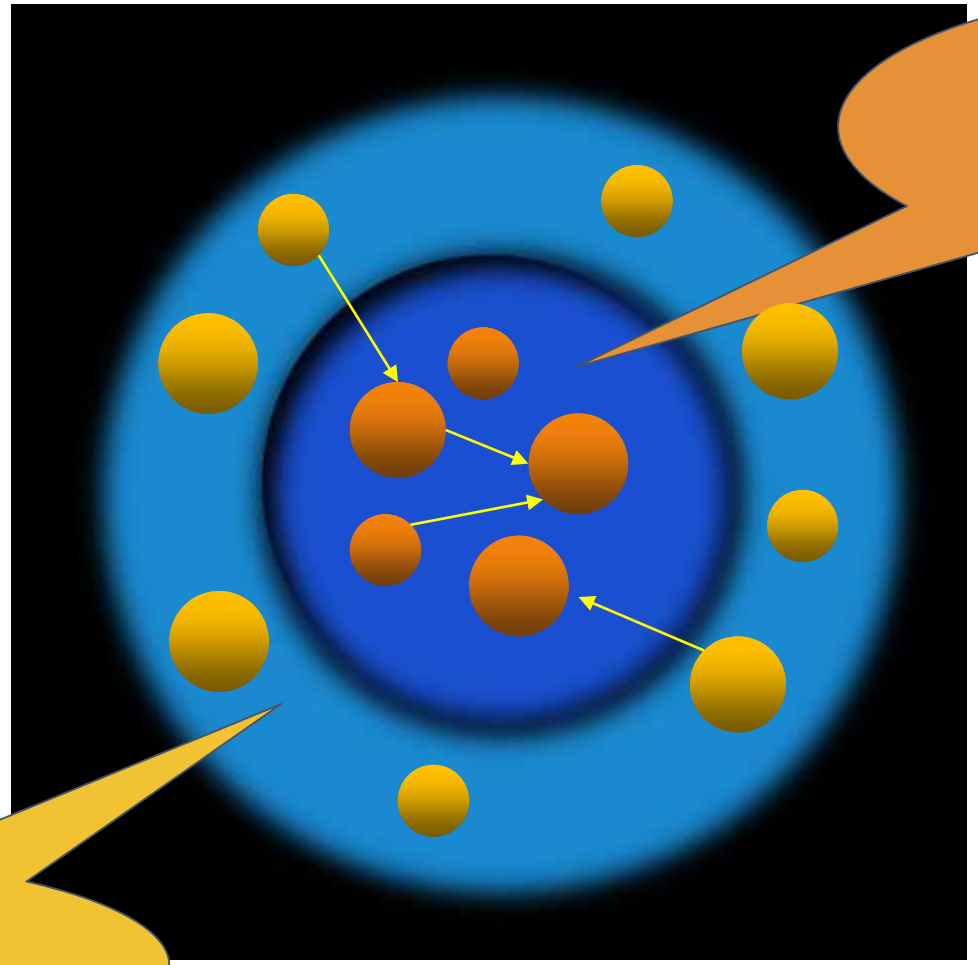
- Nicht-Funktional:
  - Portierbarkeit
  - Saubere Schnittstelle
  - Gute Testbarkeit der Kernfunktionalität
  - Leichte Erweiterbarkeit der Nutzung

Hexagonal Architecture Nur Schnittstelle zu einem/wenigen Systemen.

Welcher Stil?



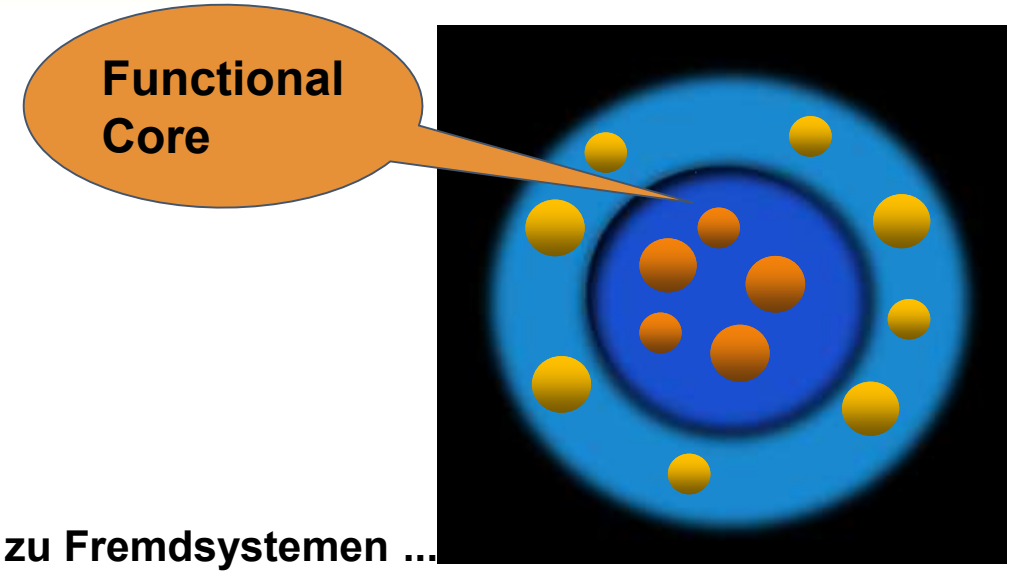
Functional Core / Imperative Shell



Functional Core

Imperative Shell

Functional Core



Functions sind *Pure*

- Keine Seiteneffekte
  - → Kein Schreiben in die Datenbank
  - → Kein Versenden von E-Mails, keine Kommunikation zu Fremdsystemen ...
- Gleiche Eingaben führt zu gleiche Ergebnissen
  - → Keine Verarbeitung von veränderlichen Werten (insb. von “außen”)
  - → Eingaben und Ausgaben sind unveränderlich

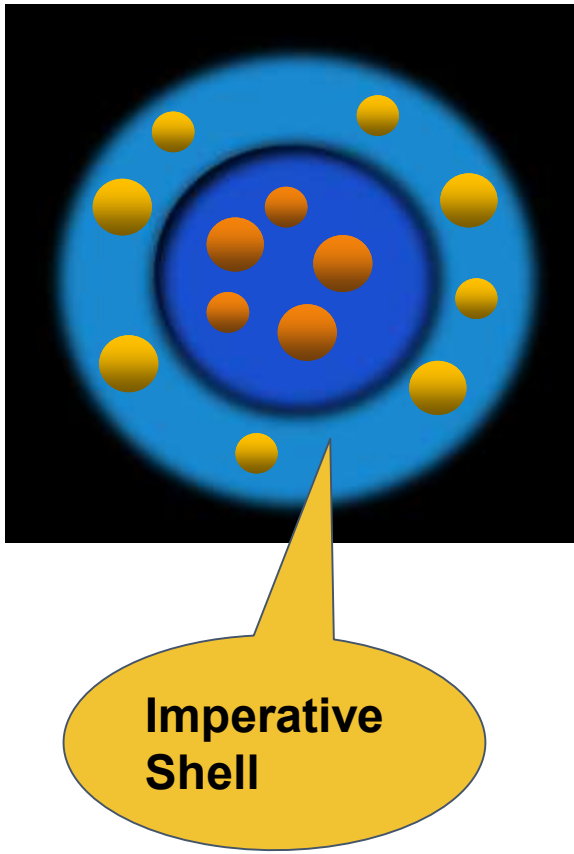
```
public List<String> addNonNull(List<String> l, String elem)
{
    var mutableList m = new ArrayList<String>(l)
    if (elem != null) {
            m.add(elem);
    }
    return List.of(m);
}
```

Imperative Shell

Lagert alle Seiteneffekte „nach außen“

Datenbank / UI / Interface-Aufrufe...

- **Orchestriert Seiteneffekt-behaftete Teile und Seiteneffekt-freie Funktionalität**
- **„Spricht“ über Werte mit dem Core → Nicht über Interfaces**




```
public List<String> addElement(String elem) {
        var newList = core.addNonNull(appState.list, elem);
        appState.list = newList;
        someFancyORM.persist(ListEntity.from(newList));
}
```

An die Arbeit… Talk

```
public final class Talk
                implements Comparable<Talk> {
  private final String topic;

  private Talk(String topic) {
          this.topic = topic;
  }

  public String getTopic() {
     return topic;
  }

  public String toString() {
          return topic;
  }

  public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || getClass() != o.getClass()) return false;
       Talk talk = (Talk) o;
       return topic.equals(talk.topic);
  }

  public int hashCode() {
      return Objects.hash(topic);
  }

  public int compareTo(Talk other) {
      return this.topic.compareTo(other.topic);
  }

}
```

Agenda

```
public final class Agenda {

   private final List<Talk> talks;
   private final String lastOperationMessage;

   private Agenda(List<Talks> talks, String lastOperationMessage) {
      this.talks = talks;
      this.lastOperationMessage = lastOperationMessage;
   }


   public static Agenda initializeAgenda() {
      return new Agenda(List.of(), "Keine Vorträge vorhanden");
   }


  public Agenda addNewTalk(Agenda agenda, String topic) {
      return (talkExists(topic))
         ? new Agenda(agenda.talks, String.format("Vortrag %s existiert bereits", topic));
         : new Agenda(addToCurrentList(Talk.createNewTalk(topic)), String.format("Vortrag %s erstellt",  topic));
   }
```

```
   public List<Talk> getTalksSortedByTopic() {
      Collections.sort(talks);
      return talks;
   }

   private List<Talk> addToCurrentList(Talk newTalk) { … }
   private boolean talkExists(String topic) { … }
   List<Talk > getTalks() { … }

   public String getLastOperationMessage() …
   public Agenda addRatingToTalk(String topic, Rating rating)
…
   public Agenda toggleStatus(String topic) …
}
```

Moment mal – Da stimmt doch was nicht…

```
private final List<Talk> talks;

public List<Talk> getTalksSortedByTopic() {
     Collections.sort(talks);
     return talks;
}
```

Achtung: Eine versteckte Änderung des State

```
public List<Talk> getTalksSortedByTopic() {
        return
       talks.stream().sorted()
       .collect(Collectors.toUnmodifiableList());
    }
```

FauxO

Funktionaler Programmierstil:

```
public final class Agenda {
	public Agenda addNewTalk(Agenda agenda, String topic) {...}
}
```

```
agenda = agenda.addNewTalk(agenda, topic)
```

Mit FauxO:

```
public final class Agenda{
	public Agenda addNewTalk(String newTopic) {...}
}
```

```
agenda = agenda.addNewTalk(topic);
```

Nutzung in der Imperative Shell

```
@Controller
public class TalkController {
    @Autowired
    private ApplicationState applicationState

    @GetMapping("/")
    public String showAgenda(@ModelAttribute(Model model)) {
        List<Talk> talks = applicationState.getAgenda().getTalksSortedByName();
        model.addAttribute("talks", TalkRepresentation.listFrom(talks));
        return "talk-list";
    }


    @PostMapping("/talk")
    public RedirectView createTalk(@RequestParam(value = "Vortragsthema") String topic, RedirectAttributes attributes) {
        createNewTalk(topic);
        attributes.addAttribute("topicCreateMessage", applicationState.getAgenda().getLastOperationMessage());
        return new RedirectView("/");
    }
```

Testen des Functional Cores

```
 @Test
public void If_talk_exists_Then_feedback_is_added() {
        //Arrange
        Agenda agenda = Agenda.initializeAgenda();
        agenda = agenda
                        .addNewTalk(FIRST_TALK).addNewTalk(SECOND_TALK)
                        .toggleStatus(FIRST_TALK).toggleStatus(SECOND_TALK);

        //Act
        agenda = agenda
                                .addRatingToTalk(FIRST_TALK, Rating.TOP)
                                .addRatingToTalk(SECOND_TALK, Rating.OKAY).addRatingToTalk(SECOND_TALK, Rating.OKAY);

        //Assert
        assertThat(agenda.getTalks().size()).isEqualTo(2);
        assertThat(agenda.getTalks().stream().filter(t -> t.getTopic().equals(FIRST_TALK)).map(Talk::getTop)).containsExactly(1);
        assertThat(agenda.getTalks().stream().filter(t -> t.getTopic().equals(SECOND_TALK)).map(Talk:::getOkay)).containsExactly(2);
}
```

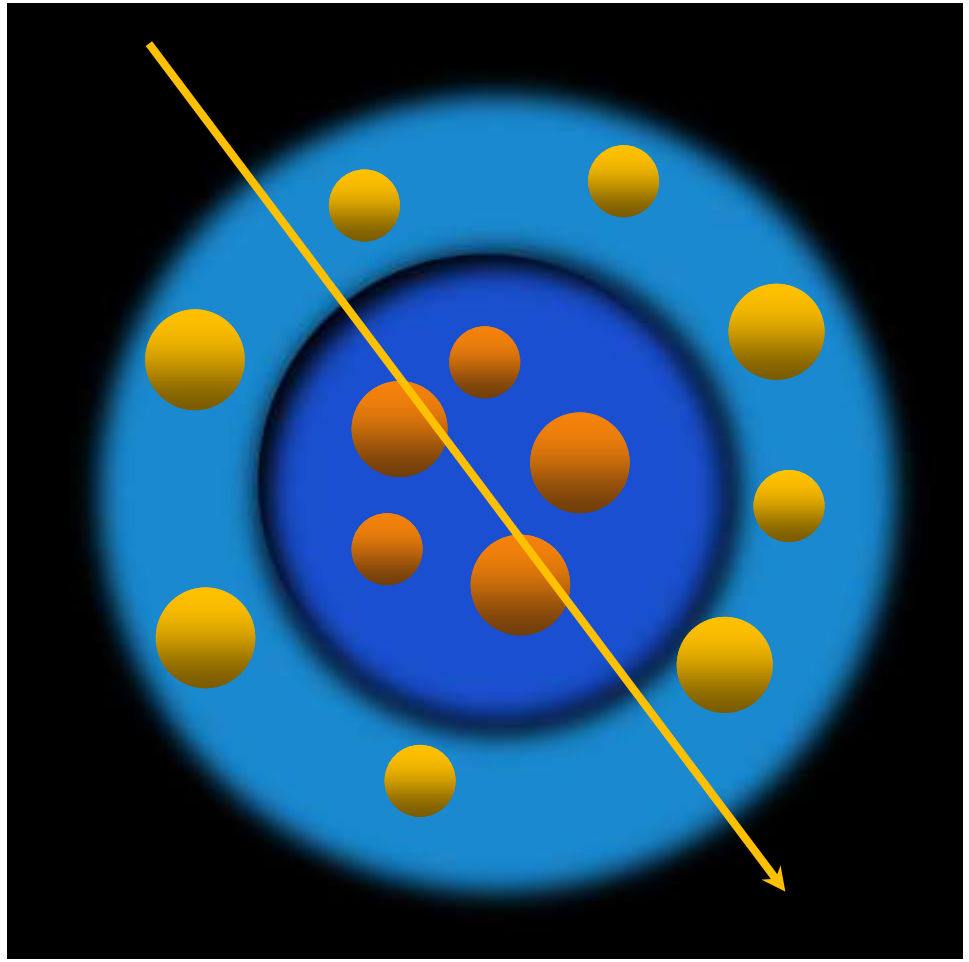
- Functional Core läßt sich gut durch Junit-Test testen
  - Keine Infrastruktur nötig
  - Keine Mocks nötig
  - Schnell laufende Tests

Testen der Imperative Shell

```
@Test
    public void Then_ratings_are_displayed() throws Exception {
```

- Aber was ist mit der Imperative Shell?
  - Alles ist voll mit Abhängigkeiten
  - Viel zu Mocken
  - ....

Testbarkeit



Functional Core Tests

- Sind immer Unit-Test
- Sind schnell
- Prüfen funktional strukturierten Code
- Benötigen keinen Application Context
- Benötigen keine Mocks

Imperative Shell Tests

- Sind (meist) Integration-Tests
- Prüfen für eine Benutzerinteraktion die „Integration der Seiteneffekte“
- Prüfen lineare Logik und sind daher einfach strukturiert
- Benötigen keine Mocks

Testen Imperative Shell

```
private static final String CONTENT_TYPE = APPLICATION_FORM_URLENCODED_VALUE

@Test
public void Then_ratings_are_displayed() throws Exception {
    //Arrange
    mockMvc.perform(post("/talk").contentType(CONTENT_TYPE).content(“Thema=Talk").accept(CONTENT_TYPE));
    mockMvc.perform(post("/talk/toggleStatus").contentType(CONTENT_TYPE).content(“Thema=Talk").accept(CONTENT_TYPE));

    //Act
    mockMvc.perform(post("/talk/addFeedback/TOP")
                        .contentType(CONTENT_TYPE).content(“Thema=Talk").accept(CONTENT_TYPE));
     mockMvc.perform(post("/talk/addFeedback/TOP")
                        .contentType(CONTENT_TYPE).content(“Thema=Talk").accept(CONTENT_TYPE));

   //Assert
   String result = mockMvc.perform(get("/")).andExpect(status().isOk()).andReturn().getResponse().getContentAsString();
   assertThat(result).contains(title=\"Name:\" value=\"Talk\");
   assertThat(result).contains("title=\"RateTop:\" value=\"2\");
}
```

Lessons learned

- Trennung von Shell und Core in der Praxis nicht immer ganz einfach
  - Definition von Graubereichen
  - Orchestrierung des Codes
- Definition von relevanten Seiteneffekten notwendig.
  Beispiel Logging:
  - Seiteneffekt: Ja
  - Hierdurch hervorgerufene Fehler in der Anwendung: Eher Nein
- Zwar sehr häufig, aber nicht immer möglich Verzweigungscode aus der Imperative Shell fern zu halten
- Manchmal dennoch Entscheidung für Mocks in Tests, um komplizierte Konstellationen einfacher nachstellen zu können

Dar

Imperative Shell

Lagert alle Seiteneffekte „nach außen“

- Orchestriert Seiteneffekt-behaftete Teile und Seiteneffekt-freie Funktionalität

- „Spricht“ über Werte mit dem Core
  → Nicht über Interfaces

Kann den Anwendungszustand halten und hierfür veränderliche Zeiger verwenden

Functional Core

Functions sind Pure
- Keine Seiteneffekte
- Gleiche Eingaben
  → Gleiche Ergebnisse

Objekte sind unveränderlich
- Keine Zustandsänderung bestehender Objekte möglich

„FauxO“ möglich:
Daten und Funktionalität einer Verantwortlichkeit können gekoppelt sein

Weiterführende Links

GitHub-Repository

https://github.com/electronickai/functional-core-demo

Separation of immutable and mutable logic (gist Link-List)

https://gist.github.com/kbilsted/abdc017858cad68c3e7926b03646554e

Testing without Mocks:
https://www.jamesshore.com/Blog/Testing-Without-Mocks.html