ELCA
We make it work.

Stefan Heinzer
JUG Bern
03.09.2025

JAVA USER GROUP CH

Implementing DDD made easy

using Spring and jMolecules

What is Domain-driven design?

Why

- Software fit for purpose / client needs
- Less misunderstandings in team
- Greatly improved maintainability

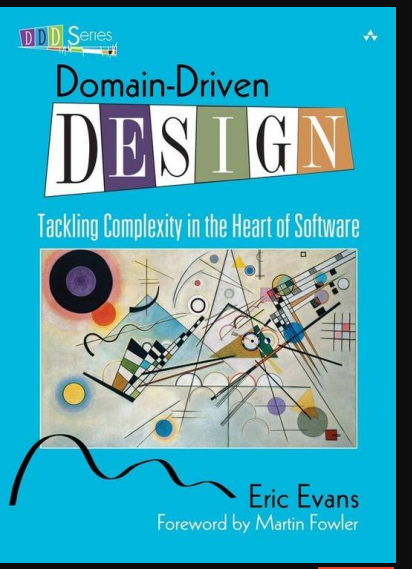
DDD is **letting the code talk business**

How

- Closely collaborate with domain experts
- Use succinct ubiquitous language
- Separate domain from technical logic

"The heart of software is its ability to solve domain-related problems for its users."

Eric Evans
Author of «the blue book»



What you will learn in this talk

1 — Structure the code

2 — Implement the domain

3 — Add persistence and web API

4 — DDD at scale

Our Task

Event Model

- Aggregates
- Commands
- Domain events

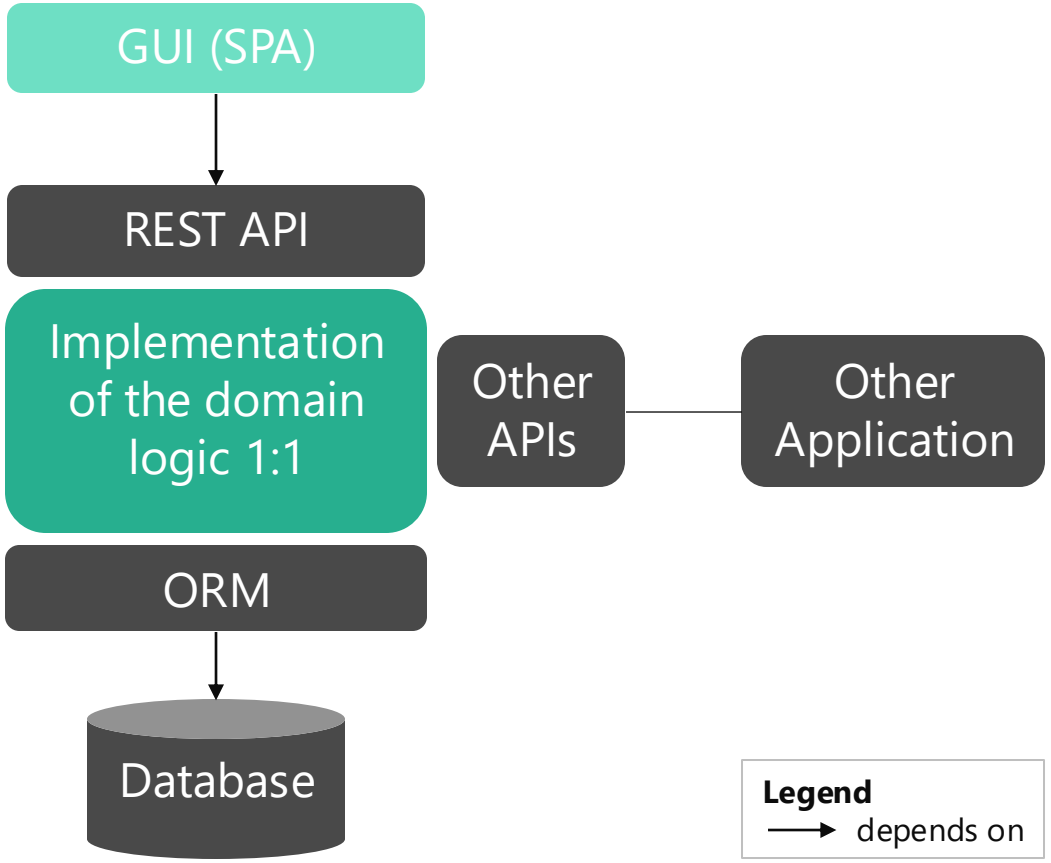


Image from https://h55.ch

Sample Domain: planeZ

Startup «planeZ»

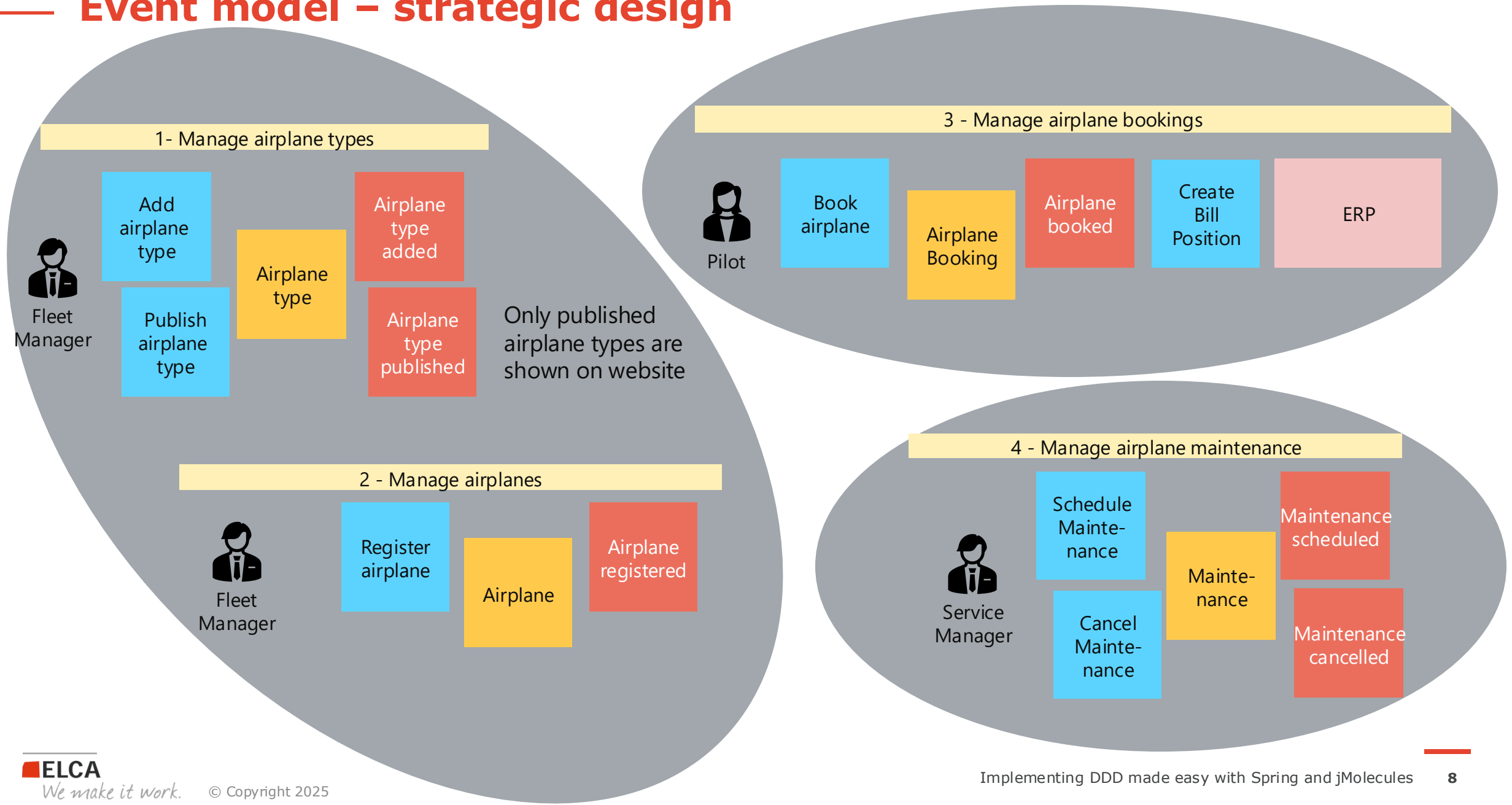
Rent airplanes with net zero emission

- Browse different airplanes available
- Book them on an hourly basis
- Access the airplane through app
- Pay per use

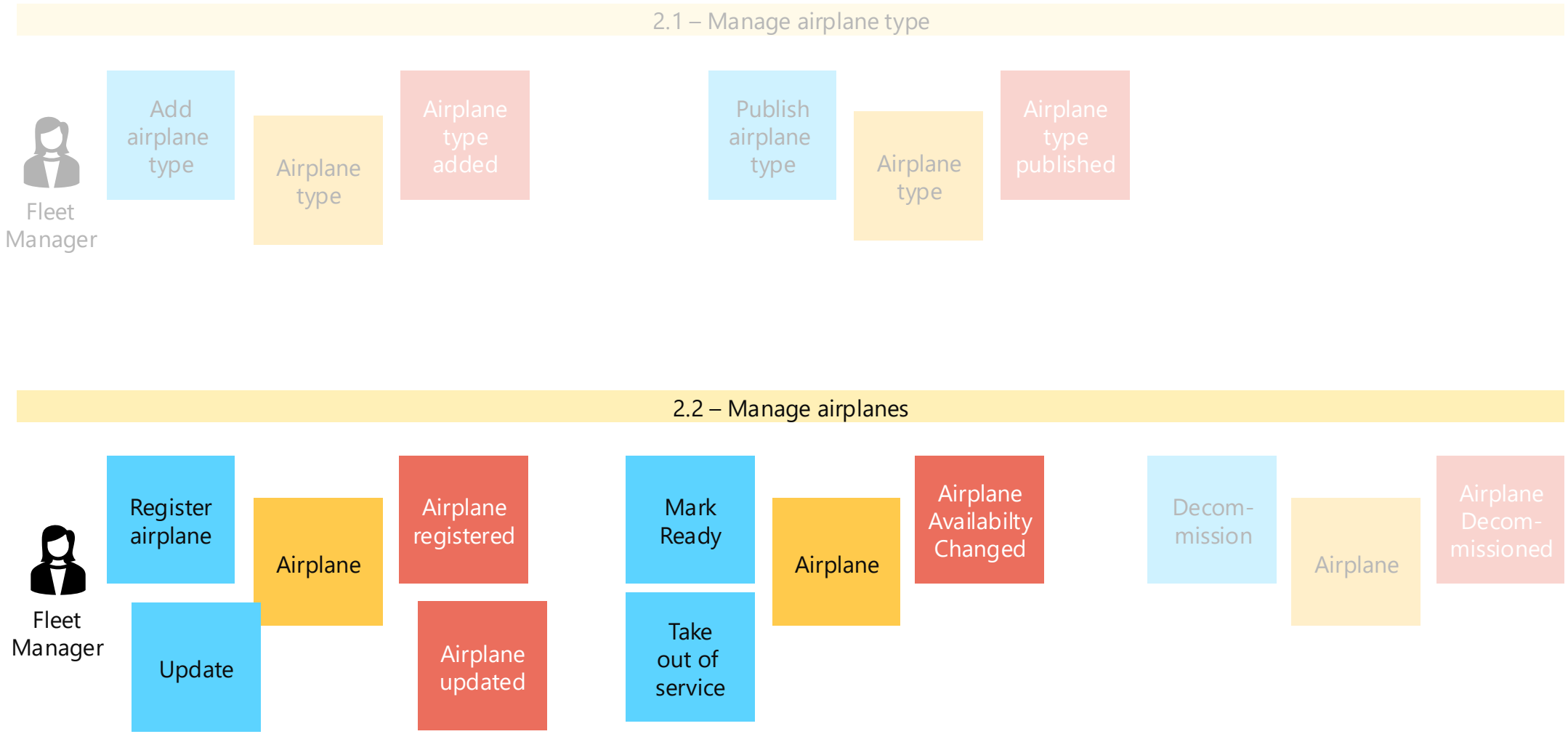
System to manage airplanes, their bookings as well as service cycles, where planes are not available.

Standard ERP for back-office tasks such as billing and book-keeping.

Event model – strategic design



Event model – airplane context



1 — **Structure the code**

2 — Implement the domain

3 — Add persistence and web API

4 — Insights from real projects

Find a suitable package structure

Classical: technical layers

```
myapp/
    entities/
    repositories/
    services/
    controllers/
```

Does not express any domain concept!

Domain-driven: package by feature!

```
myapp/
    airplane/
    booking/
    maintenance/
```

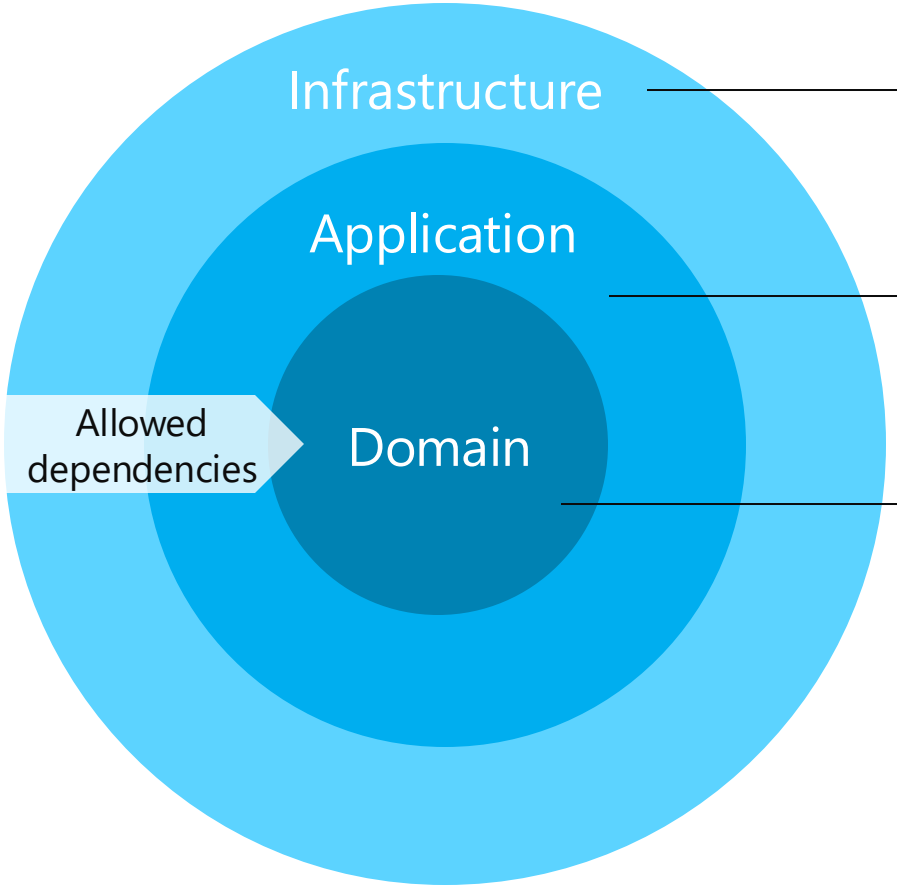
Where to place technical stuff?

```
myapp/
    airplanes/
        repository/
        service/
        controller/
    bookings/
    maintenance/
```

Are you kidding?

```
myapp/
    airplanes/
        domain classes (top level / flat)
    bookings/
    maintenance/
    _application
    _infrastructure
```

Simplified Onion Architecture



- Persistence
- APIs, Adapters
- Security

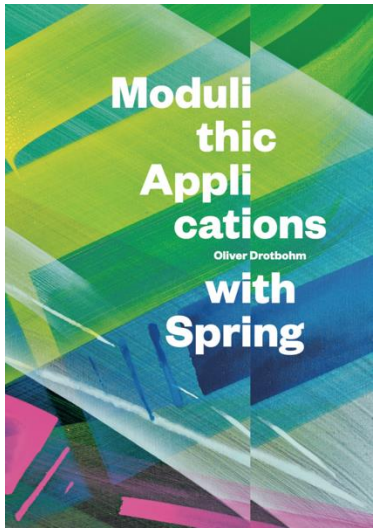
- Orchestration
- Transactions

- Domain model
- Business rules
- Ports

Spring Modulith

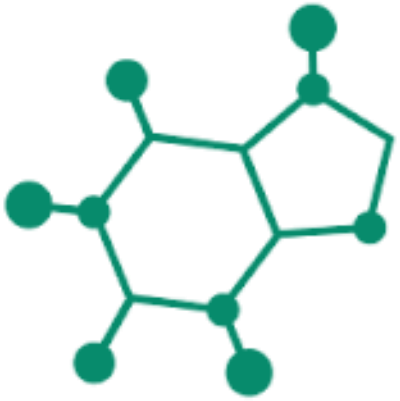
- Package-based module architecture
- Convention over configuration
- Bootstrapping of verticals / modules to keep test execution fast
- Detect cycles in modules
- Inter-module eventing patterns ready to scale out to micro-services
- Event-persistence to let unpublished events survive system restarts

Oliver Drotbohm
ex. Spring Data



Book on Leanpub
66% complete

jMolecules – “Architecturally evident code”



Interfaces or Annotations

- Onion, Hexagonal, Layered architecture
- Elements of tactical DDD (Repository, Aggregate, Value Object, etc.)
- Optional technology integration: Spring, JPA, Jackson, etc.
- Compile-time and build-time **validation of architectural rules**

Oliver Drotbohm

1 — Structure the code

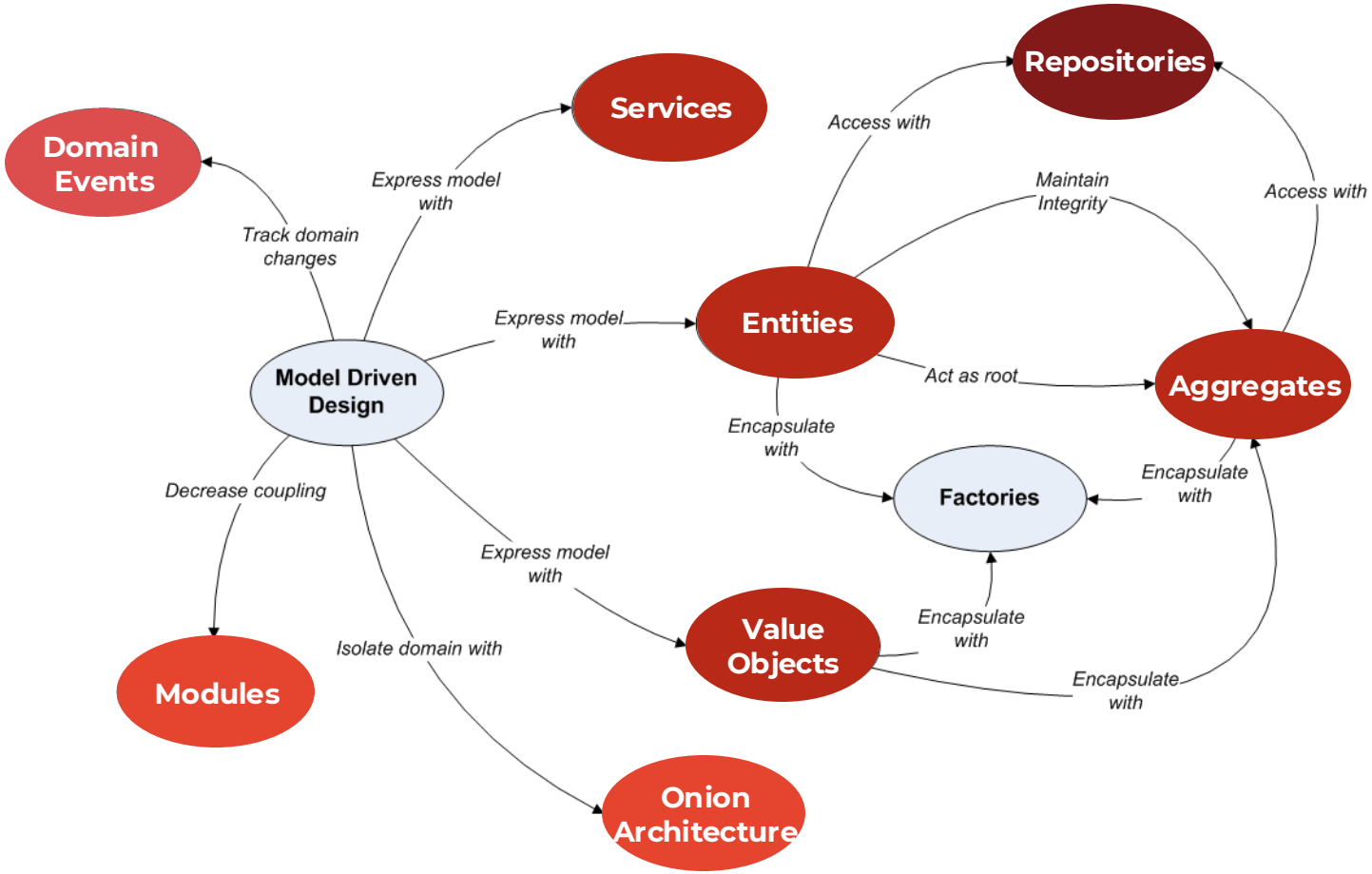
2 — **Implement the domain**

3 — Add persistence and web API

4 — Insights from real projects

Elements of tactical DDD

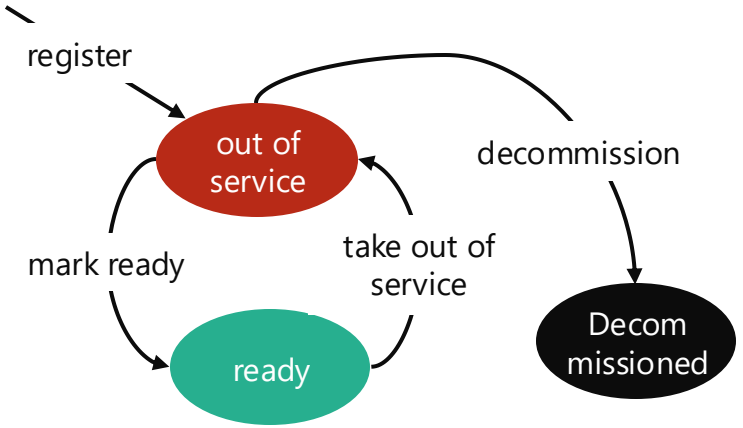
- **Data and logic:** Entities, value objects, aggregates, domain service
- **State change:** Domain events
- **Persistence:** Repositories
- **Structure:** Modules, layered architecture

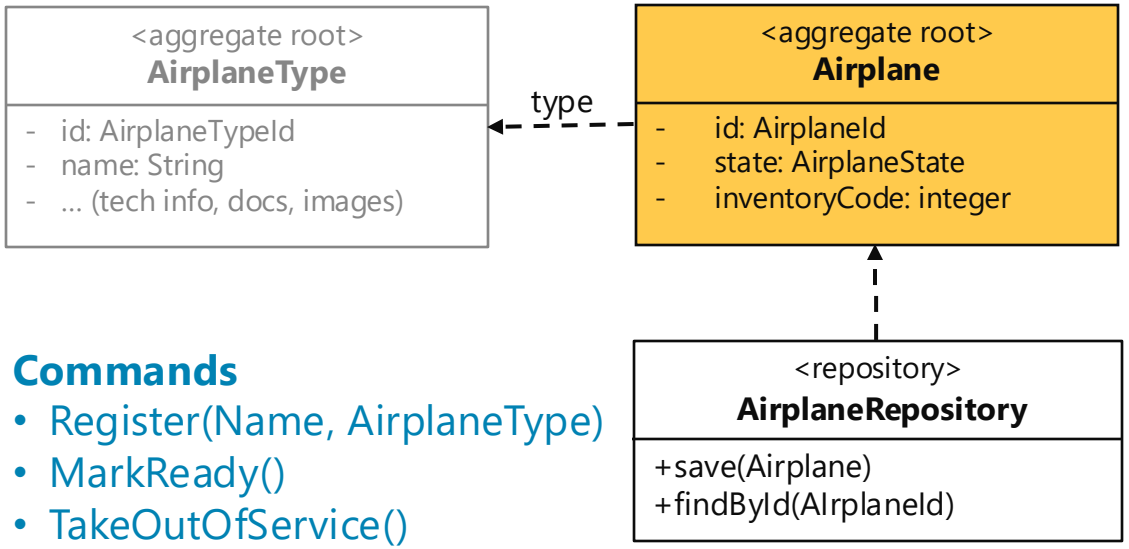


Airplane model 0.1

Business Rules

- After registration, the airplane is out of service until marked ready
- An airplane can be taken out of service at any time as long as it has no been decommissioned
- The inventory code can only be changed if the airplane is out of service





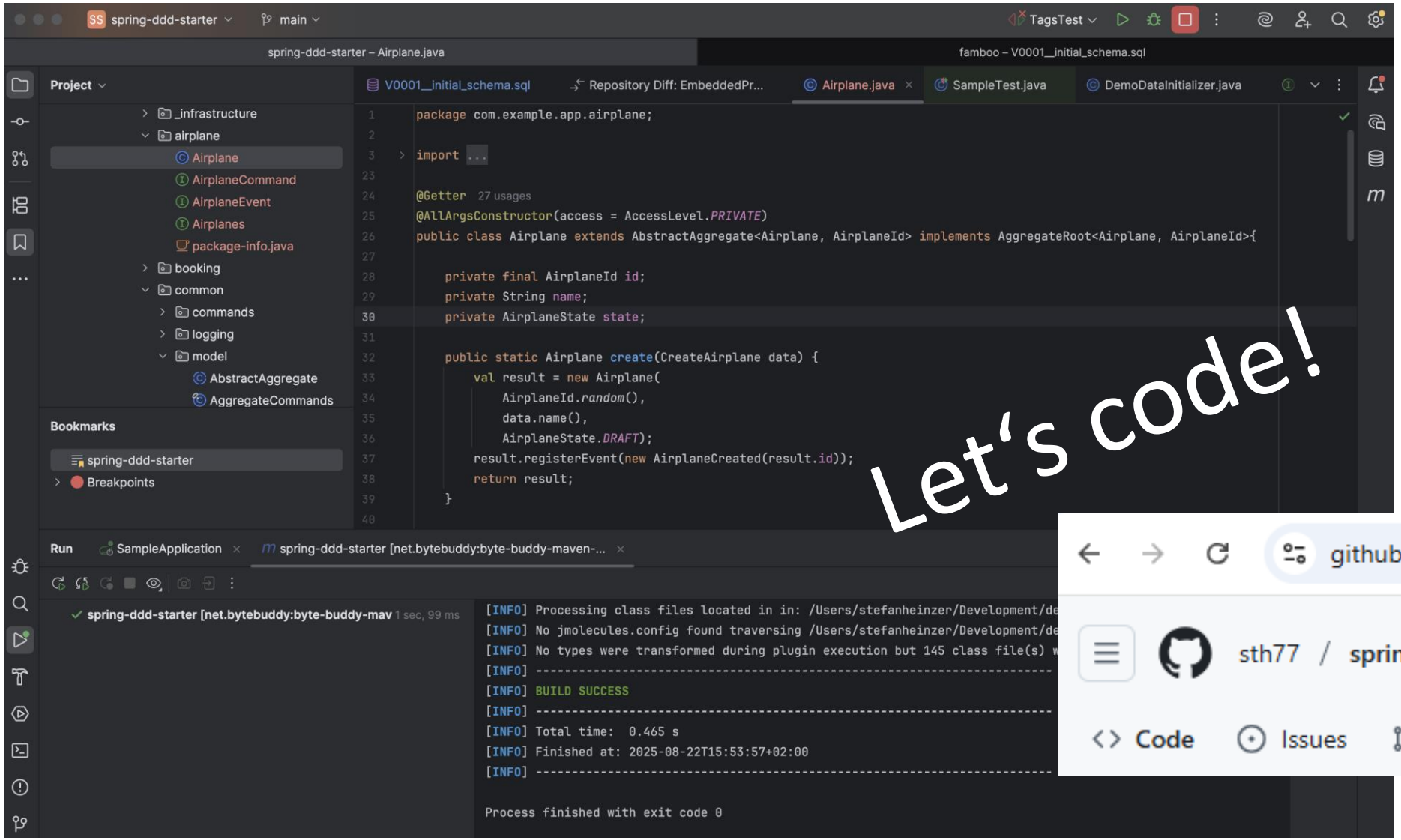
Commands

- Register(Name, AirplaneType)
- MarkReady()
- TakeOutOfService()

Domain Events

- AirplaneRegistered(AirplaneId)
- AirplaneAvailabilityChanged(AirplaneId, AirplaneState)

Expressing the domain model in Java



1 — Structure the code

2 — Implement the domain

3 — **Add persistence and web API**

4 — Insights from real projects

How to persist aggregates?

Option 1: Map the aggregates directly
Add persistence annotations to aggregates to make them persistable by an ORM

Option 2: Use separate persistence model
Map aggregates to and from separate persistence model, which lives in the infrastructure layer

Option 3: Event-sourced
Do not store the fields of the aggregate, but the history of events that lead to that state.

Option 1b: jMolecules Byte Buddy
Generate persistence annotations at build time on byte code only

Maven plugin in pom.xml

```
<plugin>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy-maven-plugin</artifactId>
    <version>${bytebuddy.version}</version>
    <executions>
        <execution>
            <goals>
                <goal>transform-extended</goal>
            </goals>
        </execution>
    </executions>
```

Invisible JPA for free

```java
@Getter
@AllArgsConstructor(access = AccessLevel.PRIVATE)
public class Airplane implements AggregateRoot<Airplane, AirplaneId> {

    private final AirplaneId id;
    private String name;
    private Association<AirplaneTy
    private AirplaneState state;
```

Purely domain

compile, byte-buddy:transform-extended

decompiled byte code

```java
@Entity(
    name = ""
)
public class Airplane implements AggregateRoot<Airplane, AirplaneId>, Persistable<AirplaneId> {
    @EmbeddedId
    private final AirplaneId id;
    private String name;
    @Convert(
        converter = AirplaneTypeAssociationConverter.jMolecules.wrh2L5gA.class,
        disableConversion = false,
        attributeName = ""
    )
    private Association<AirplaneType, AirplaneType.AirplaneTypeId> type;
    private AirplaneState state;
    @Transient
    private boolean __jMolecules__isNew;
```

✓ JPA persistable

Implementing the REST API

Hooking into Spring Data REST

Approach:
1) Use read operations of Spring Data REST (getOne, getAll, search/findBy, projections)
2) Force aggregate updates to go through commands

WebConfiguration.java

```
@Override  no usages  Heinzer Stefan
public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {
    // for aggregates, force modifying operations to go through the aggregate operation controller instead of using
    // the Spring Data REST CRUD API.
    config.getExposureConfiguration().withCollectionExposure(( ResourceMetadata metadata, ConfigurableHttpMethods httpMethods)
            -> AggregateRoot.class.isAssignableFrom(metadata.getDomainType())
            ? httpMethods.disable(HttpMethod.POST, HttpMethod.PATCH, HttpMethod.PUT)
            : httpMethods);
    config.getExposureConfiguration().withItemExposure(( ResourceMetadata metadata, ConfigurableHttpMethods httpMethods)
            -> AggregateRoot.class.isAssignableFrom(metadata.getDomainType())
            ? httpMethods.disable(HttpMethod.POST, HttpMethod.PATCH, HttpMethod.PUT)
            : httpMethods);
    config.setBasePath(BASE_PATH);
}
```

```java
@Transactional
@RequiredArgsConstructor
@RepositoryRestController
@ExposesResourceFor(Airplane.class)
@SecurityRequirement(name = "basicAuth")
public class AirplaneCommandController {

    private final Airplanes airplanes;

    @Secured("ROLE_USER")
    @PostMapping("/airplanes")
    public ResponseEntity<EntityModel<Airplane>> create(@RequestBody RegisterAirplane data) {
        val result = airplanes.save(Airplane.register(data));
        return ResponseEntity.ok(EntityModel.of(result));
    }

    @Secured("ROLE_USER")
    @PostMapping(path = "/airplanes/{airplaneId}/update")
    public ResponseEntity<EntityModel<Airplane>> update(@PathVariable AirplaneId airplaneId,
                                                        @RequestBody UpdateAirplane data) {
        return doWithAirplane(airplaneId, Airplane it -> it.update(data));
    }

    @Secured("ROLE_ADMIN")
    @PostMapping(path = "/airplanes/{airplaneId}/takeOutOfService")
    public ResponseEntity<EntityModel<Airplane>> publish(@PathVariable AirplaneId airplaneId) {
        return doWithAirplane(airplaneId, Airplane::takeOutOfService);
    }

    private ResponseEntity<EntityModel<Airplane>> doWithAirplane(AirplaneId airplaneId, Consumer<Airplane> action) {  2 usages
        return airplanes.doWith(airplaneId, action)  Optional<Airplane>
                .map(EntityModel::of)  Optional<EntityModel<...>>
                .map(ResponseEntity::ok)  Optional<ResponseEntity<...>>
                .orElse(ResponseEntity.notFound().build());
    }
}
```

5 Run all operations in transaction

1 Hook into Spring Data REST

2 Secure operations

3 Take command as input

4 Execute command on domain model

Key features of this architecture

- No DTOs needed in Web layer
  serializing directly aggregates, deserializing directly commands
- No separate model in persistence layer
  saving directly byte-buddy enhanced aggregates
- Only required packages
  separating domain modules and architectural layers
- Every element has its well-defined place

Not recommended for public APIs!

→ Lean & crisp

1 — Structure the code

2 — Implement the domain

3 — Add persistence and web API

4 — **Insights from real projects**

Does it work at scale?

Personnel Deployment Planning

- Demand / Operations
- Required number of employees per skill
- Allocation to supplying organizational units

Allocation to Org Units

Demand

- Vacation allotment
- Vacation requests

Absence

- Offered courses
- Trainiers
- Subscriptions

Training

Monthly Shift Plans

- Staffing of Operations with suitable employees
- Enforce labor laws and regulations

Personnel

- Employees, Skills
- Booking of employees

ERP

Bounded Contexts

Training

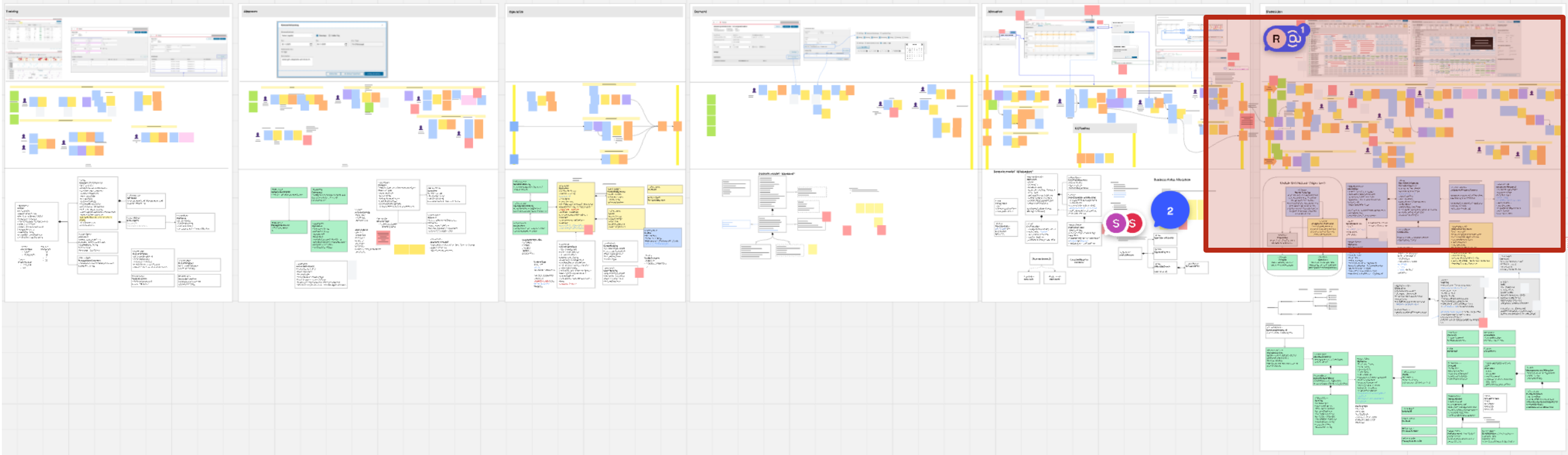
Absence

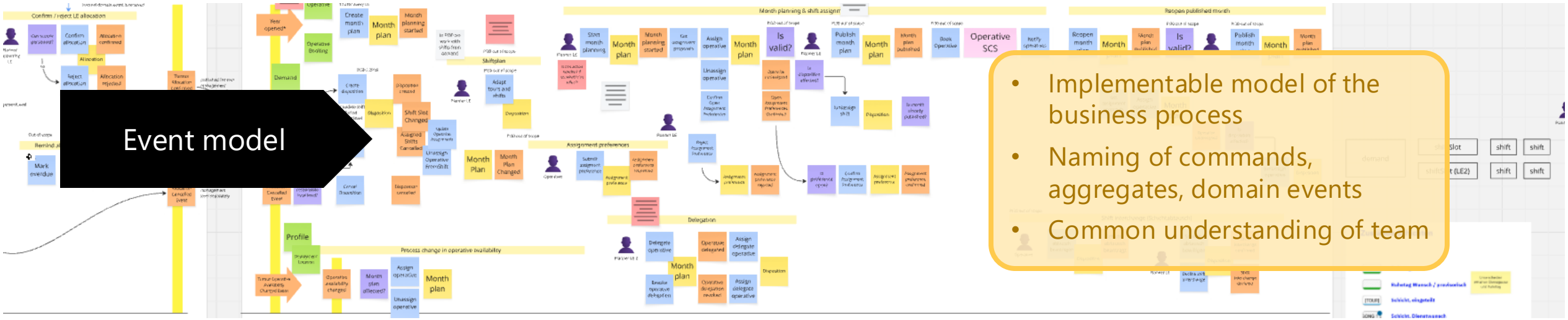
Employee

Demand

Allocation

Disposition





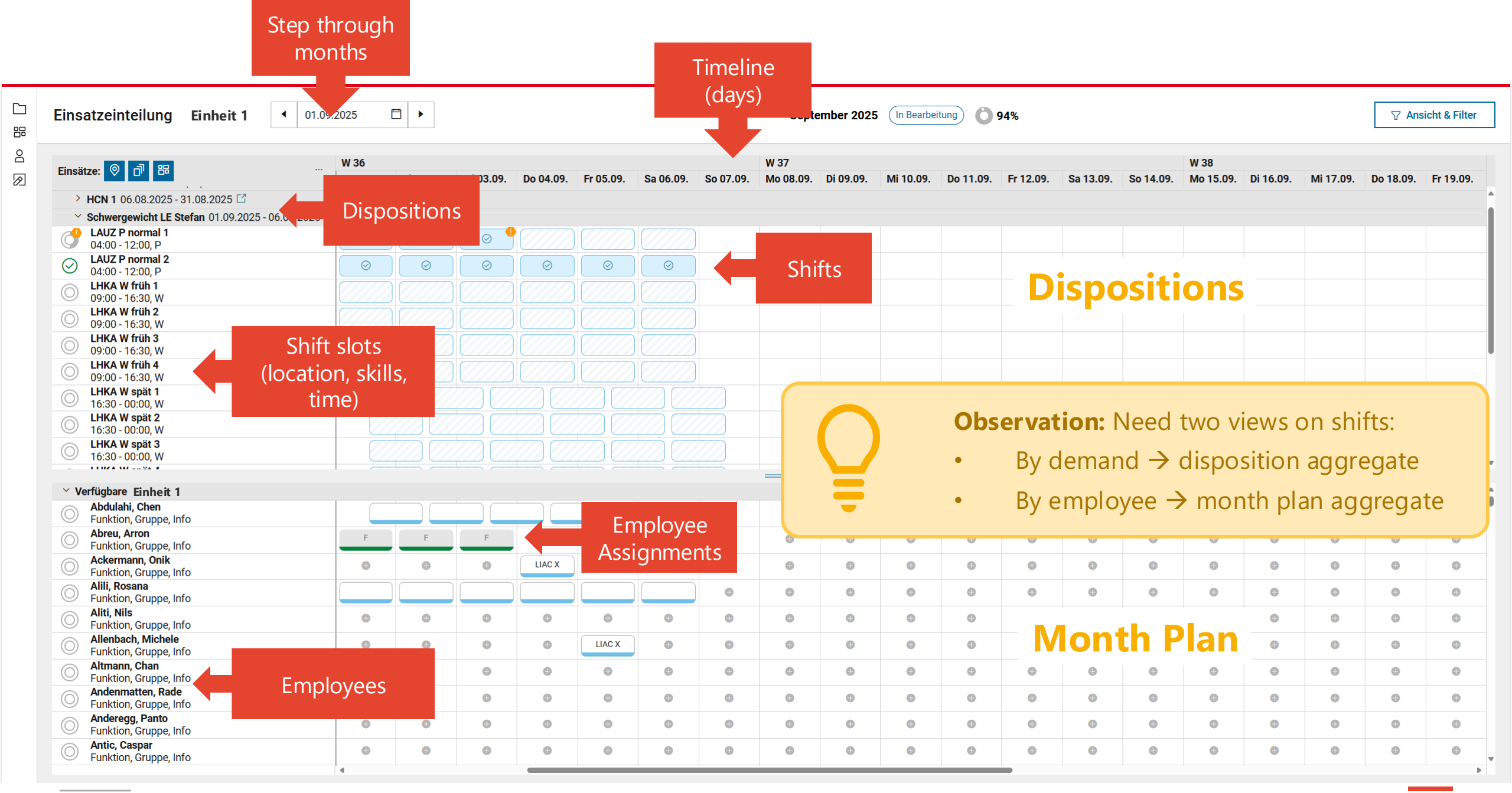
- Implementable model of the business process
- Naming of commands, aggregates, domain events
- Common understanding of team

Domain Event (facts)

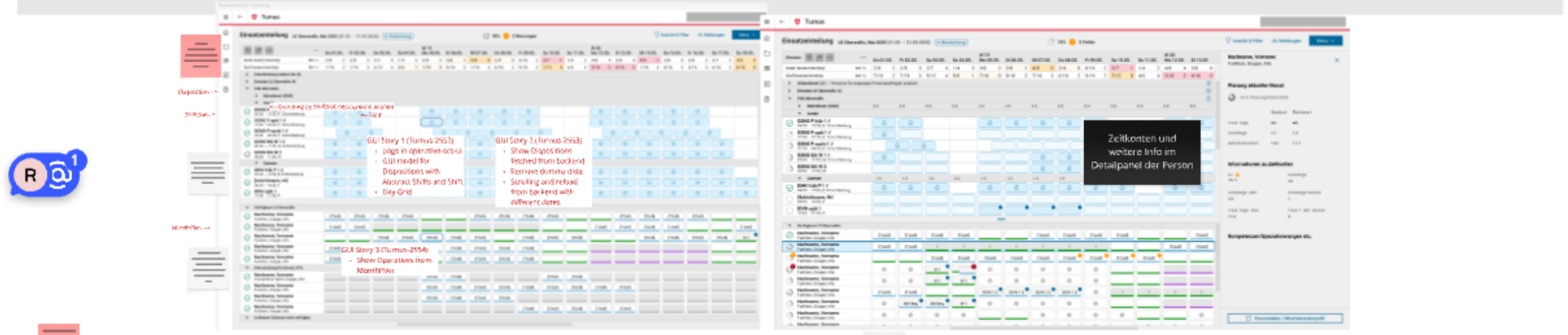
Command (intentions)

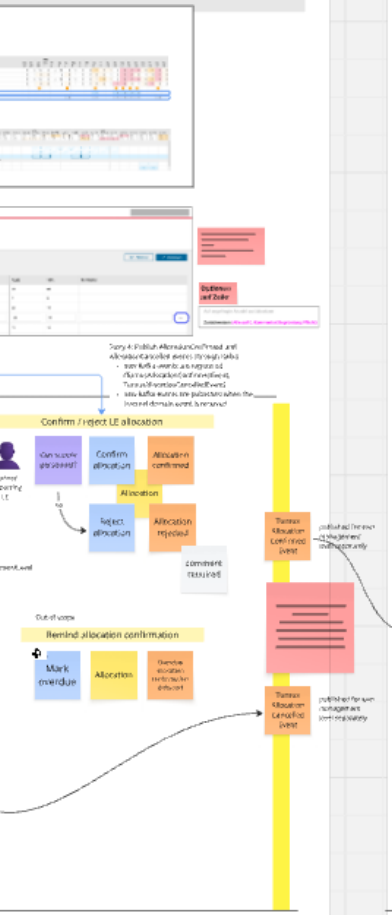
Aggregate (business object)

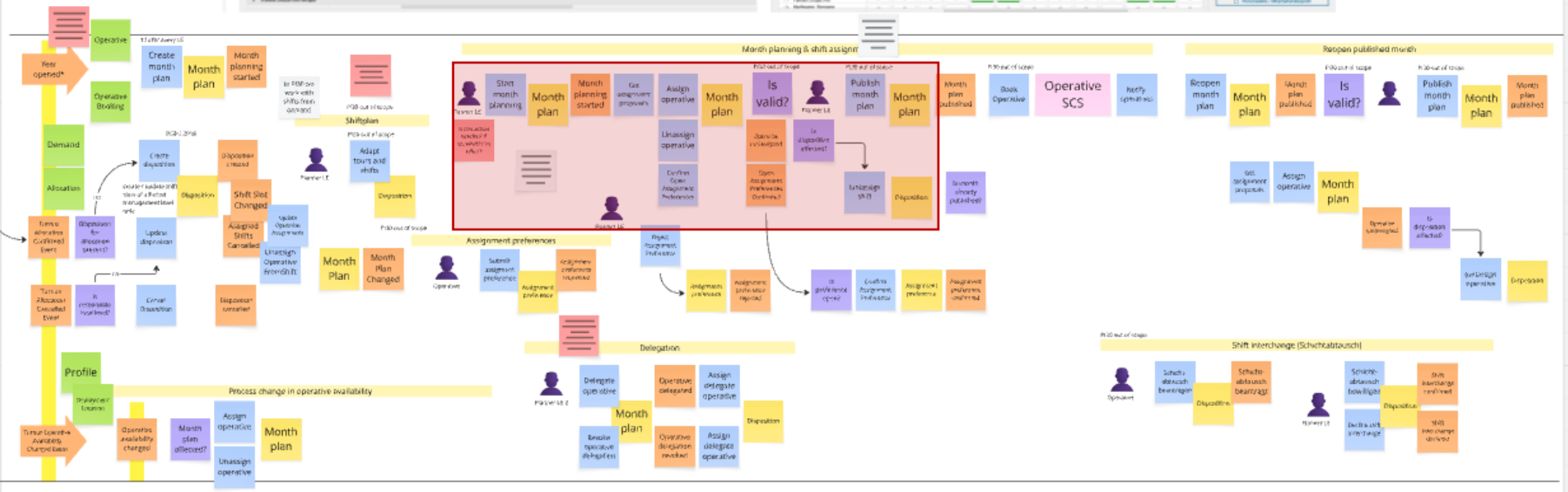
Read model (input data)

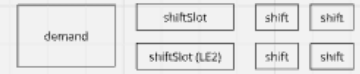


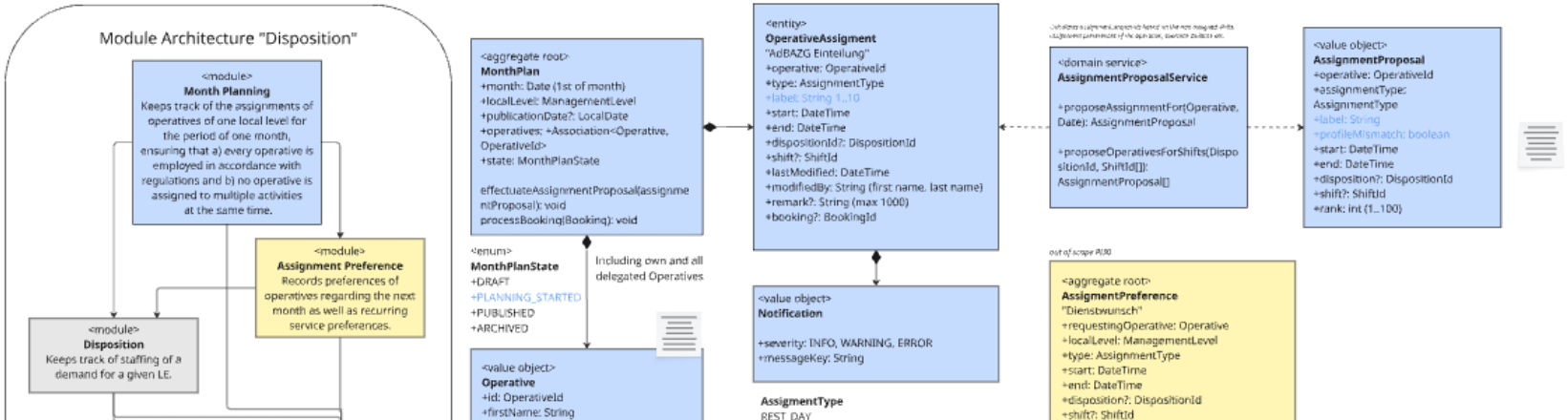
Disposition

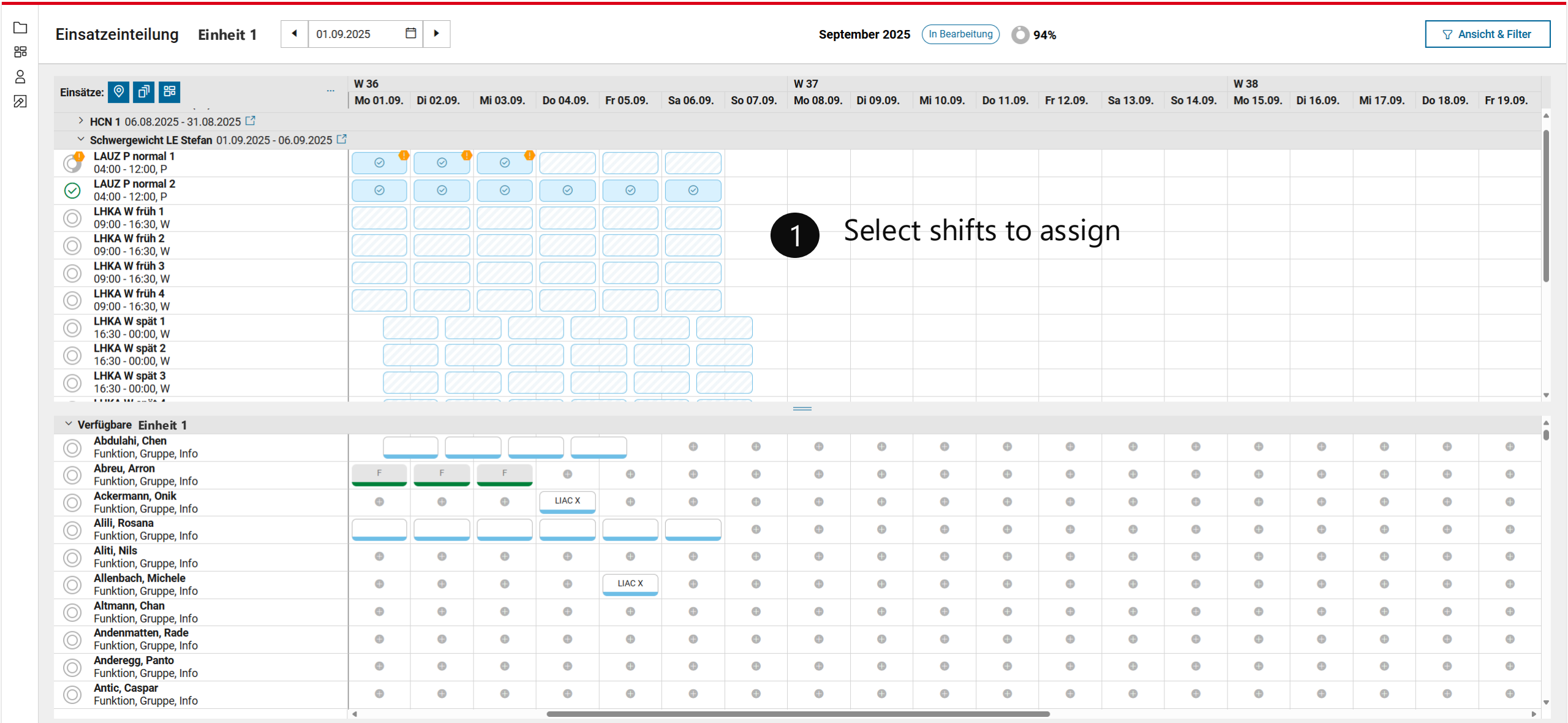


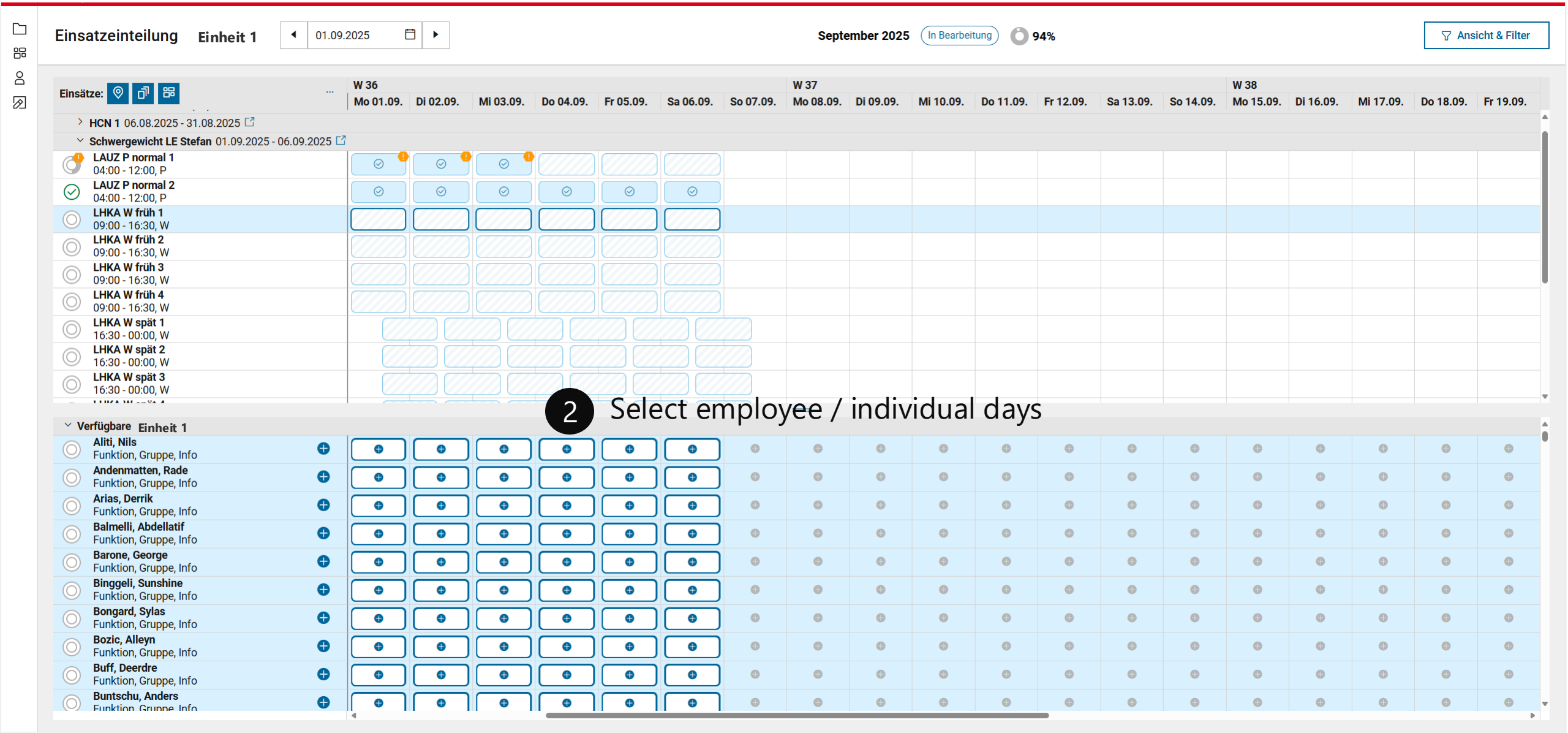










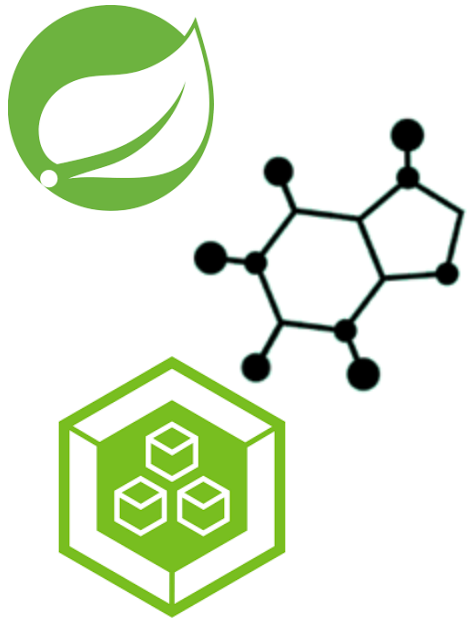


Key learnings

- ✓ DDD works perfectly at scale
- ✓ Modularization by bounded contexts and aggregates keeps complexity manageable
- ✓ Continuous work on the event model keeps the team aligned (Customer, UX, BA, Dev, Test)
- ✓ Close collaboration between user centric and domain driven design highly beneficial

Conclusion

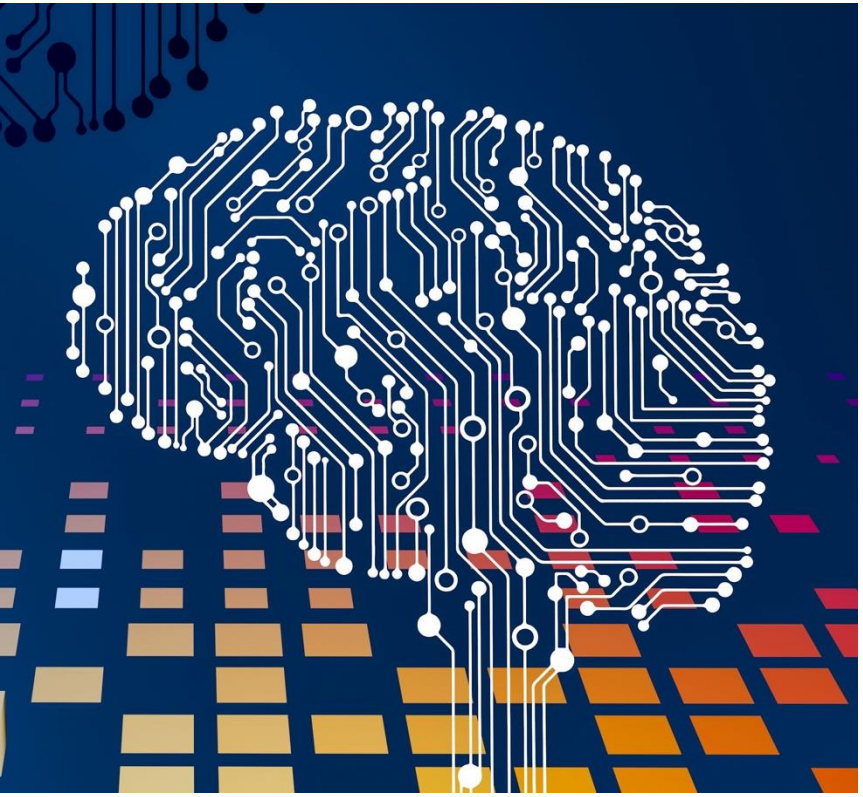
Wrap up



- DDD greatly helps **tackling complexity** through separation of concerns (contexts, modules, layers)
- jMolecules helps **expressing DDD concepts** in code and adds strong support for technology integration (e.g. JPA)
- Spring modulith comes in handy to **separate modules** and offers transactional event publication

With these tools, implementing DDD has become easy and lightweight

Outlook: DDD and AI



- Complexity of the world increases rapidly
- AI generated code which follows principles of DDD will still be **verifiable by humans**
- Our work will shift from coding to design and validation, with a strong focus on safety and security

"Software development is a learning process. Working code is a side effect."

Eric Evans

Author of «the blue book»

Questions?

Contact

Stefan Heinzer
Architecture BL
stefan.heinzer@elca.ch

Thank you!

ELCA Informatique SA
Lausanne 021 613 21 11 | Genève 022 307 15 11

ELCA Informatik AG
Zürich 044 456 32 11 | Bern 031 556 63 11 | Basel 044 456 32 11

www.elca.ch