Advanced Software Engineering

Build for ten years and more

DI Dr. techn. Mario Bernhart

Advanced Software Engineering (first 5 lectures)

- Example Project: Vienna International Airport AODB Core System
- Release your stuff 3 times a day
  - Dependency Management
  - Build Management and -Automation
  - Continuous Integration, Continuous Delivery
- Five challenges you solve for every project
  - Error Management
  - Transaction Management
  - Logging
  - Auditing
  - Declarative Authentication and Authorization
- **Build for ten years and more**
  - **Layered Software Design / API Design**
  - **Modularization / Service Design**
  - **Decoupling / Event Driven Design**
  - **Interfacing / Integration**
- From prototype to product (make it work 24/7)
  - Clustering
  - Performance
  - Monitoring
  - Automating Operational Tasks

Key Problems

- How to build for an extended lifycycle?
- Key factor is change:
  - Reuse
  - Extendability
  - Feature Changes
  - Scaleability (Change in load / throughput)
  - Maintainability (Robustness of change)
  - Testing for regressions

- Objective: Design software to minimize the cost of change

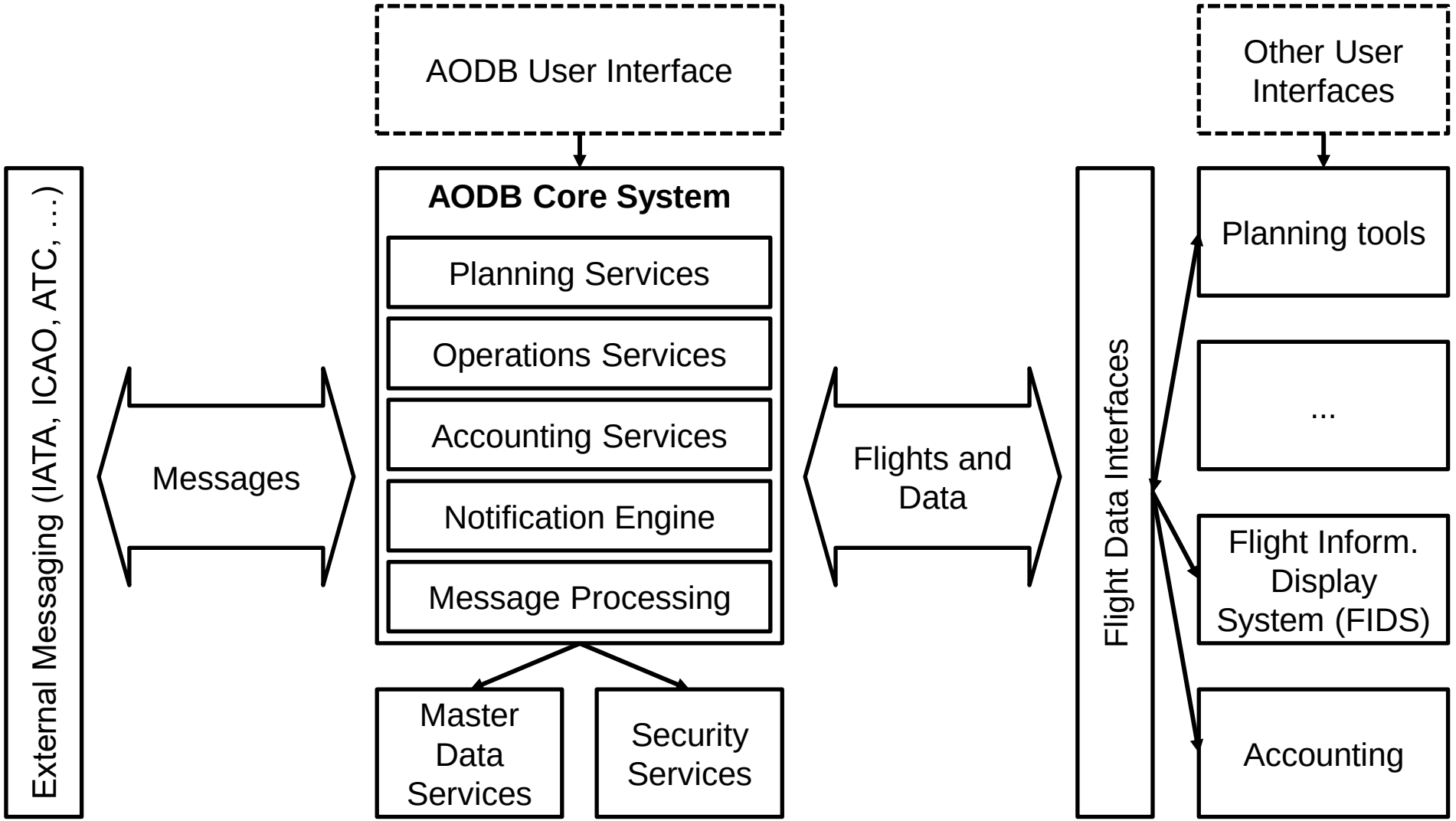
Fundamental approach

- Decomposition of a system into independent parts
- Recomposition of parts into a coherent system
  - Context aware
  - Multiple system instances
  - Static (build-time) vs. Dynamic (run-time)

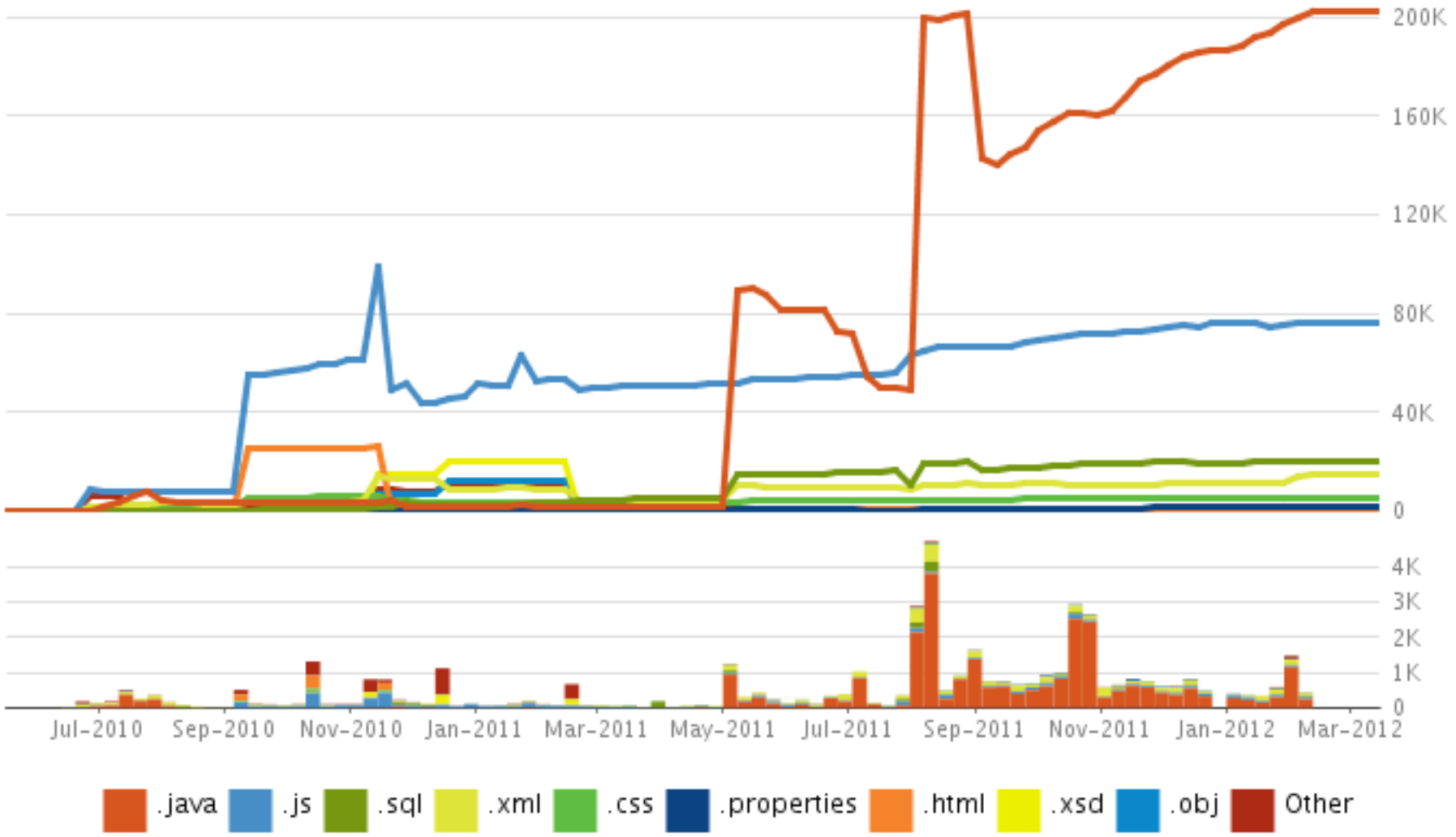
Component vs. Service (after Fowler)

- I use **component** to mean a glob of software that's intended to be used, without change, by an application that is out of the control of the writers of the component. By 'without change' I mean that the using application doesn't change the source code of the components, although they may alter the component's behavior by extending it in ways allowed by the component writers.
- A **service** is similar to a component in that it's used by foreign applications. The main difference is that I expect a component to be used locally (think jar file, assembly, dll, or a source import). A service will be used remotely through some remote interface, either synchronous or asynchronous (eg web service, messaging system, RPC, or socket.)

AODB example archticture



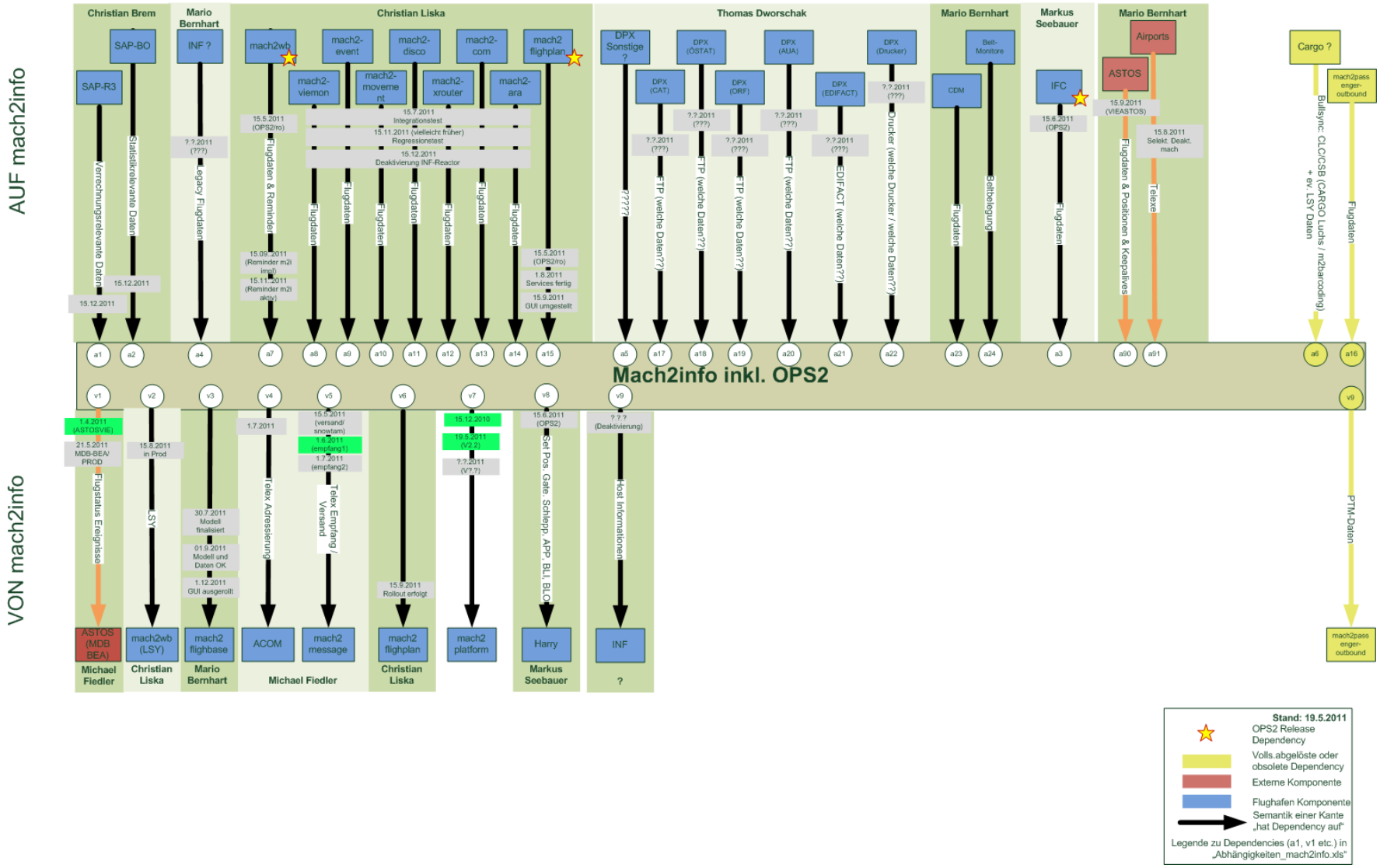
AODB LOC development



AODB technical features

- Separation between user interface and business services
  - via REST (Integration!)
  - Support for future UI-technologies
- Auto-refresh UIs (via JS-polling)
- Customizable workflows for all business processes
- Customizable rules and layouts for the notification system
- Customizable templates for message sending
- 100% open-source based

„Anatomy“ of a core system (dependency graph)



Interfacing / Integration

- Key design decision for integration is
  - Service (pull),
  - message (push)
  - or data coupling

Interfacing / Integration

- Design decisions for a service interface
  - Runtime (eg. webservice) vs. Build-Time (eg. Java library)
  - General vs. Specific interfaces
  - Synchronous (request-response) vs. Asynchronous (call-back)
  - ID vs. Natural Key object identifiers
  - Primitive type parameters vs. DTOs
  - Delta vs. Full updates of data/informaton set
  - Transformation (legacy interfaces/views)
  - Versioning of interfaces
  - Validations, Return values, Error codes
  - Reuse of components/ressources e.g. through a R/O Project

Interfacing / Integration

- Design decitions for a message interface
  - Synchronous vs. Asynchronuos
  - Event data models (payload)
  - Internal vs. external events
  - Primitive vs. complex (compound) event types

- Typical event payload
  - Reference to a primary object
  - Actual value(s)
  - Previous value(s)
  - Associated action (e.g. BLOCK_OFF)
  - Time of event creation
  - Source of event creation

Interfacing / Integration

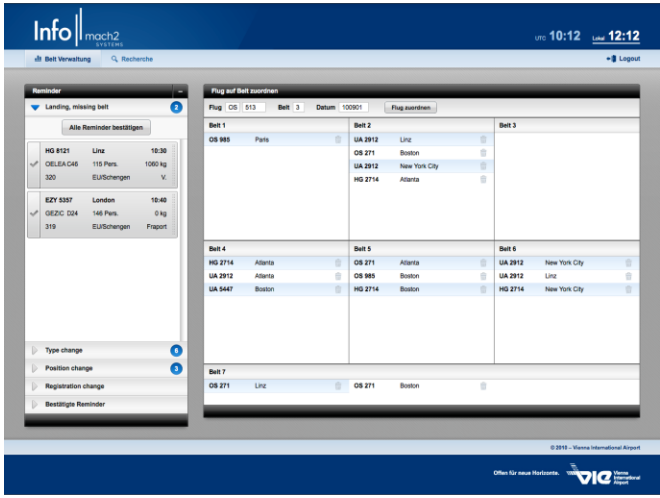
- Design decisions for data coupling
  - Separation of schemata
  - Read access through views
  - Write access through procedures

- The easiest type of integration to achieve and the hardest to get rid of

REST Example for /baggage/belt

URL	HTTP Method	Payload	Parameter	Beschreibung
/	GET	-		Beltzuordnungen für alle Belts
/{beltName}	GET	-		Beltzuordnung für Belt <beltName>
/{beltName}	POST	Flight		Herstellen der Zuordnung zwischen Belt `beltName` und dem übergebenen Flug, retourniert Beltzuordnung für Belt <beltName>
/{beltName}/{flightDesignator}	DELETE	-	day	Auflösen der Zuordnung zwischen Belt `beltName` und dem mit `designator` identifizierbaren Flug retourniert Beltzuordnung für Belt <beltName>

Status code	Fehlerbeschreibung
201 Created	Flug wurde dem Belt zugewiesen
300 Multiple Choices	Flug konnte nicht eindeutig identifiziert werden
404 Not Found	Flug oder Belt nicht gefunden
409 Conflict	Flug ist bereits einem anderen Belt zugewiesen



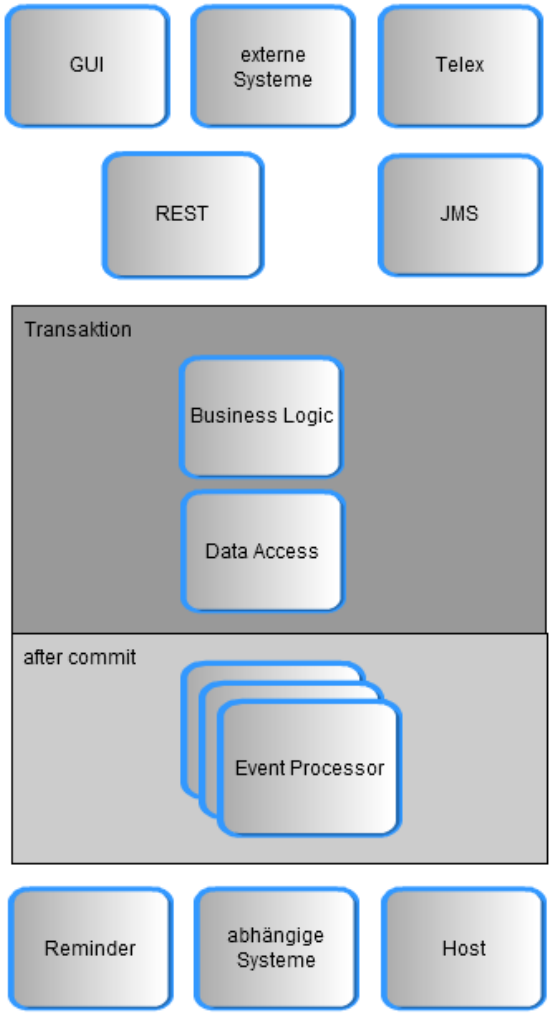
Layered Software Design / API Design

- Why to cut software into layers / modules
  - Separation of Concerns
  - Abstraction
  - Testability
  - Error Handling
  - Transaction management (What is the exact transaction scope?)
  - Reuse
    - Frameworks
    - Custom (e.g. DAOs in other projects)

- How to cut software into layers / modules
  - Separate UI from Logic
  - Separate Model from Logic
  - Separate Data Access from Logic (via a common interface)
  - Separate Connectors from Logic (via a common interface)

Single Service, Multiple-Consumers

- Callstack:
  - Clients
    - GUI
    - external system
    - Telex
  - REST Layer, JMS Consumer
  - Business Service
  - Data Access Layer
  - Tx-Boundry (commit)
  - Postprocessing
    - Notifications
    - Connected systems (data push)
    - Legacy system sync



Example software stack

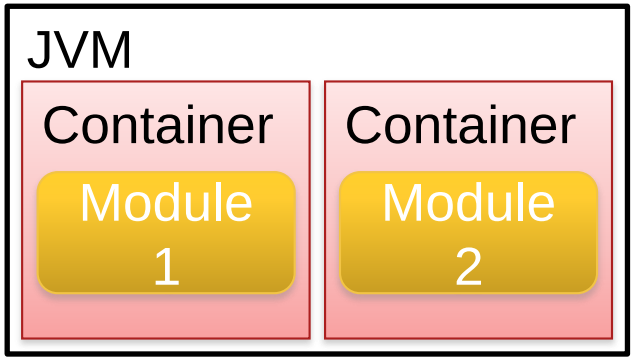
- Glue: Spring 3.0 (core, tx, security, integration, jms, orm, oxm)
- REST (JSR-311): Apache CXF 2.4
- JSON: Jackson 1.6
- JPA 2.0 (JSR-317): Hibernate 3.6
- Bean Mapping: Dozer 5.3
- Clustering: Hazelcast 1.9
- Database: Oracle 10.x
- JMS: Oracle AQ
- UI: HTML + JavaScript (jQuery) + CSS

Key Questions for choosing the right level of modularization

- Fine grained vs. Business Services
- Requirements on transactional capabilities
- Requirements on high availability & distribution
- Release and deployment scenarios
- Lifecycle (e.g. legacy connectors)

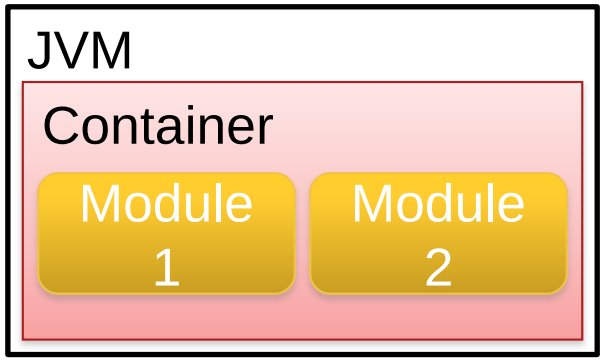
Forms of modularization

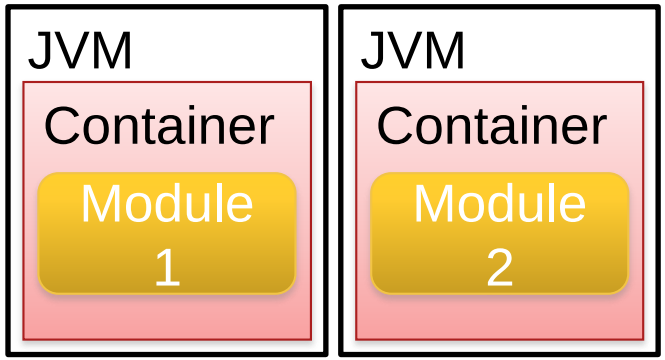
Build Time:
- Multiple JARs possible
- Single Bundle
- Update requires complete redeploy
- Easy operation
- Easy & fast inter-module communication



Runtime (multi VM):
- Multiple JARs required
- Multiple Bundles
- Update requires partial redeploy
- Highly complex operation
- Complex and possibly slow inter-module communication

Runtime (single VM):
- Multiple JARs required
- Multiple Bundles
- Update requires partial redeploy
- Medium complex operation
- More complex but fast inter-module communication

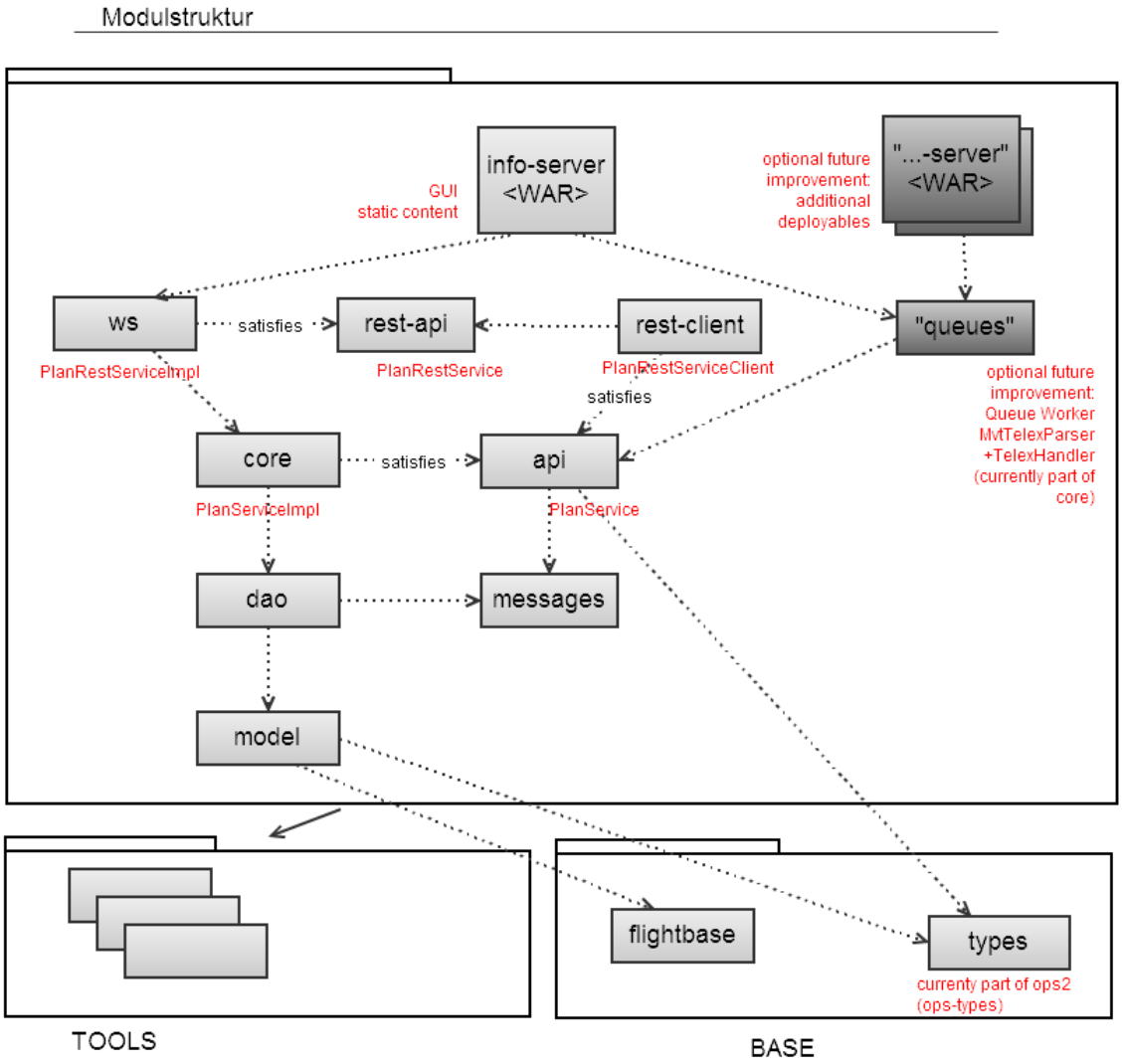




Java technologies for modularization

- OSGI (runtime)
  - Initially created for the embedded systems domain
  - Recent release of Enterprise OSGI (R4.2) specification
  - Additional control over how classpath is constructed (fine grained export of packages/services)
  - (Still) targeted for single-VM operation (e.g. Eclipse Platform)
- Maven (build time)
  - Support for simultaneous assmebly of multiple modules (possibly with multiple levels of hierarchy)
  - Management of direct and transitive dependencies
- Project Jigsaw (language level modularization)

Build-time modularization in mach2info

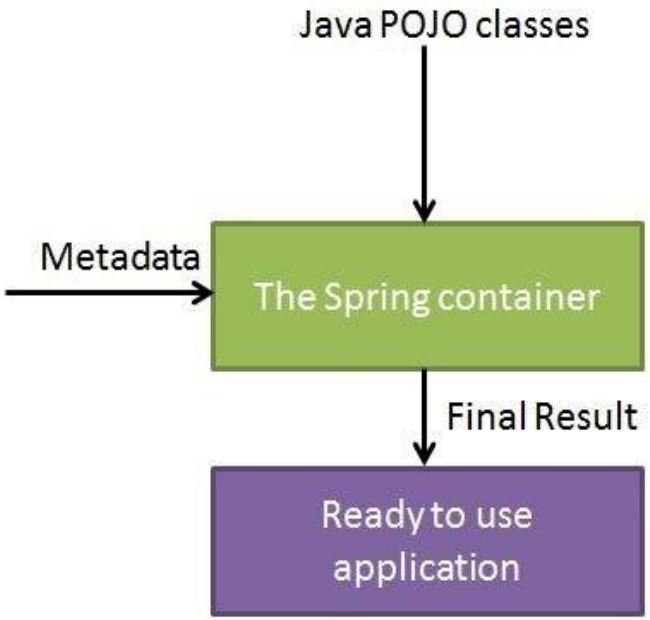


Dependency Injection (DI)

- Dependency Injection (DI) is a pattern, where the gluing of objects is separated from the implementation.
- All implementation is against the API (!)
- There is a central definition and container that creates and binds objects togeher.
- DI supports code reuse and independently testing classes.
- DI support different bindings for different environments.
- DI supports laizy creation of objects and (e.g. useful for limited memory environments)
- A DI framework (often based on AOP) provides the runtime services for DI. e.g. the Spring Framework

About the Spring Framework

- A popular application development framework for enterprise Java
- Spring Framework (Architecture) is modular and allows you to pick and choose modules that are applicable to your application.
- POJO's (plain old java object) are called 'beans' and those objects instantiated, managed, created by Spring IoC container.
- The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.
- DI helps in gluing loosely coupled classes together and at the same time keeping them independent
- Spring supports the utilization of existing frameworks e.g. for logging, ORM etc.



Spring Framework cornerstones

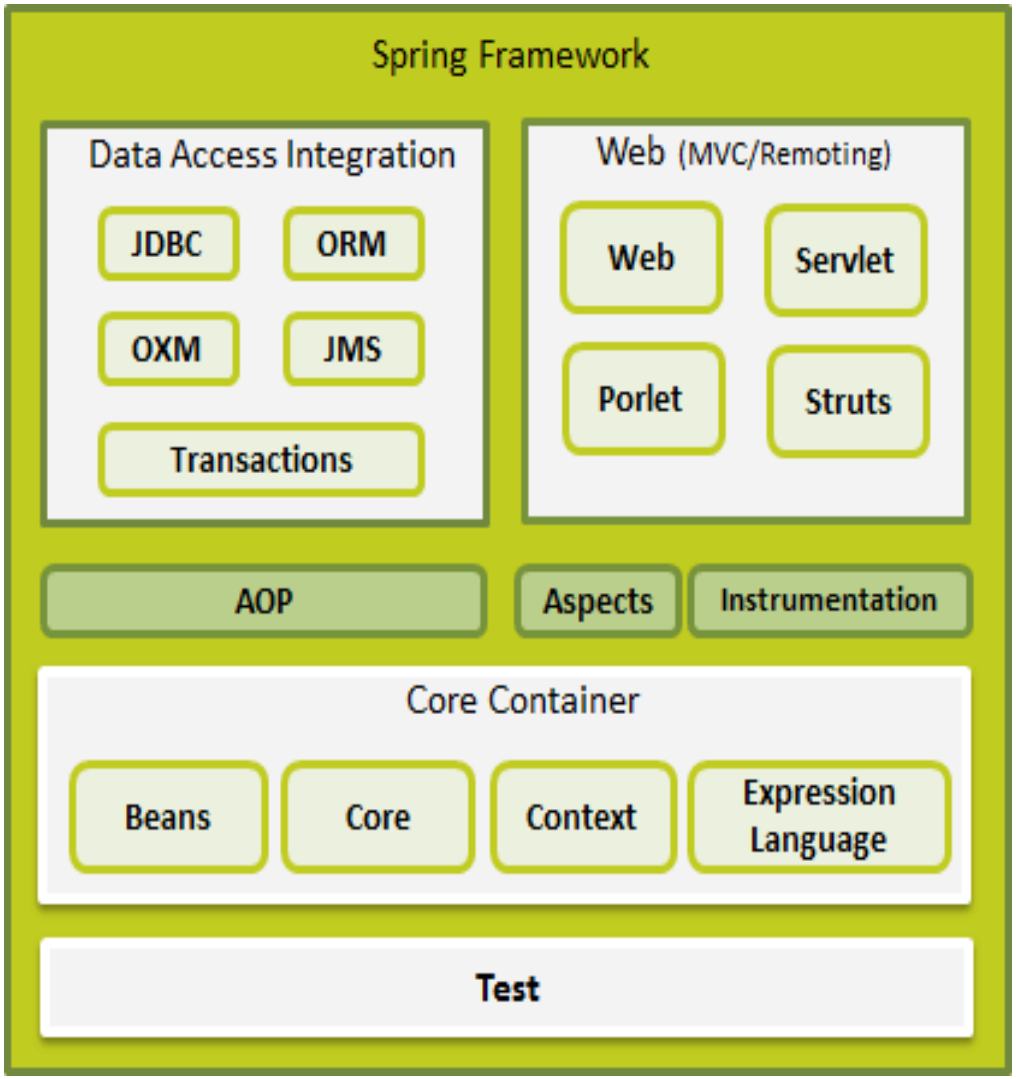
- Spring Framework is a well-designed web model-view-controller (MVC) framework
- provides a coherent transaction management interface that be applicable to a local transactions() local transactions or global transactions (JTA)
- provides a suitable API for translating technology-specific exceptions (for instance, thrown by JDBC, Hibernate, or JDO,) into consistent, unchecked exceptions.
- The Inversion of Control (IoC) containers are lightweight, especially when compared to EJB containers.

  → Being lightweight is beneficial for developing and deploying applications on computers with limited resources (RAM&CPU).
- Testing is simple because environment-dependent code is moved into this framework.

Aspect Oriented Programming (AOP)

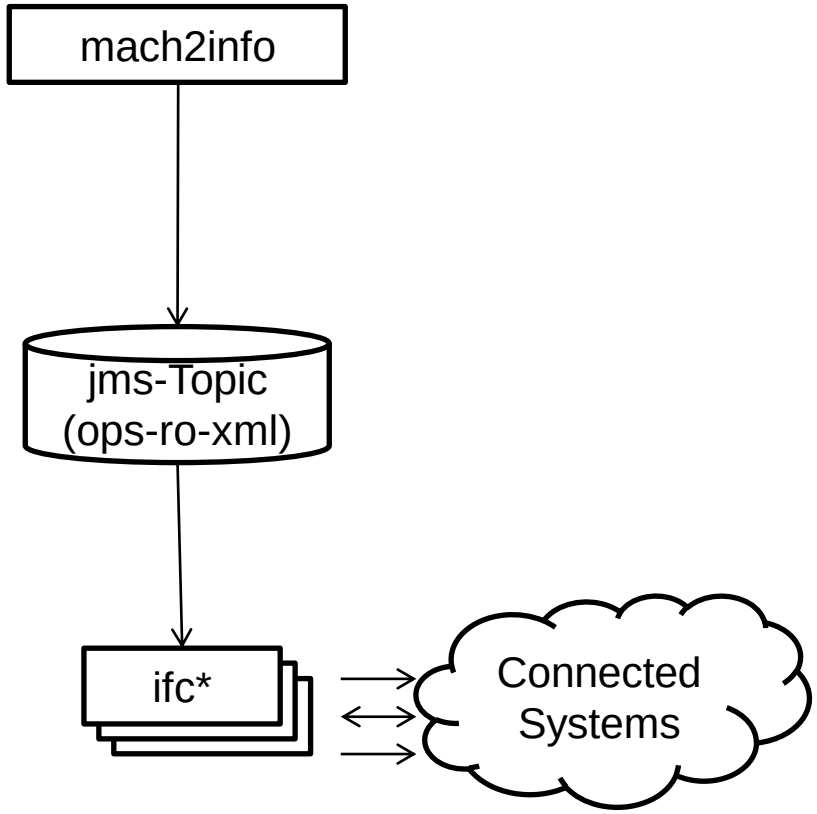
- The key unit of modularity is the aspect in AOP (class in OOP)
- Cross-cutting concerns are the functions that span multiple points of an application.
- Cross-cutting concerns are conceptually separate from the application's business logic.
- AOP helps you decouple cross-cutting concerns from the objects that they affect Examples (logging, declarative transactions, security, and caching)
- Aspects are „woven in“ at compile time or runtime

Spring Framework Architecture



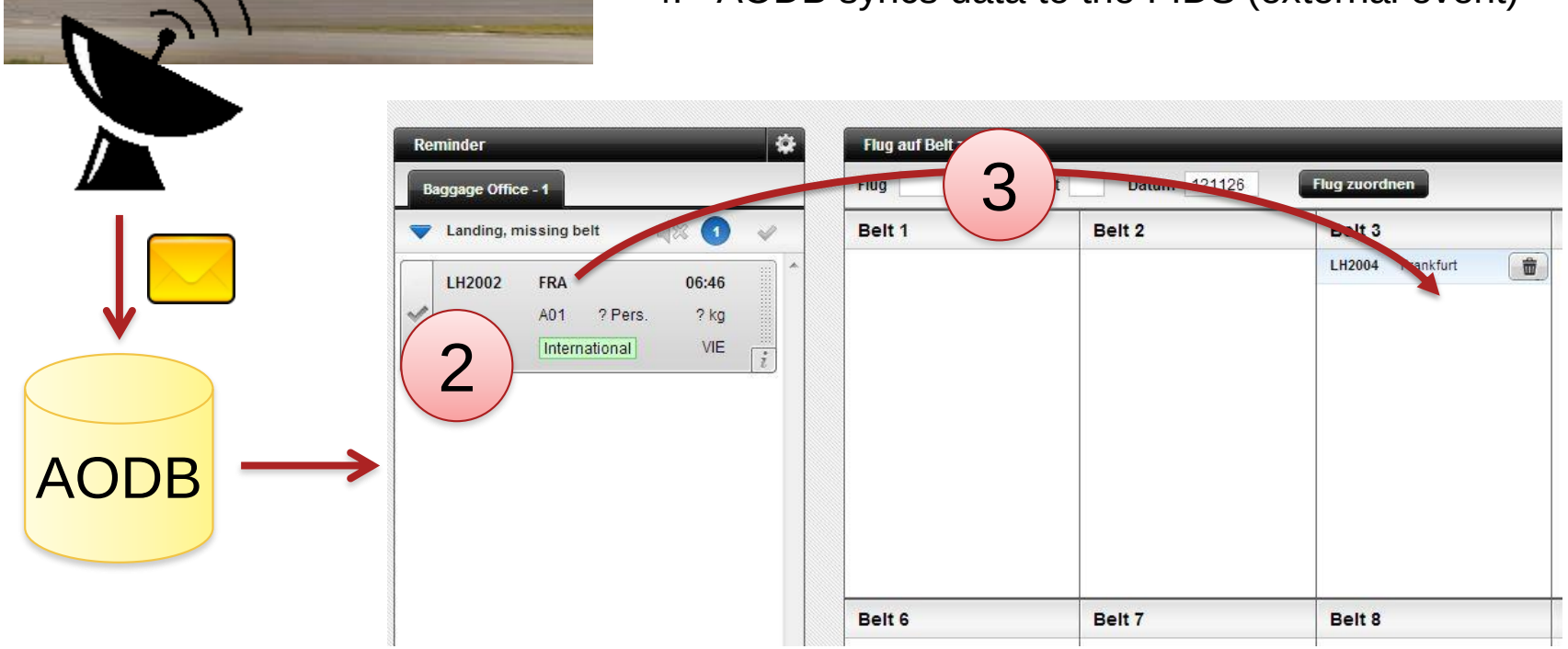
Example for an event based architecture

- Loose coupling
- Activator: "after Transaction Commit"
- Foundation for asynchronal processing
  - Connected systems
  - Messaging within the system
  - Messaging to other systems
- Implementation with Spring Integration
  - Enables persistent Queuing (asynchronal processing)
  - Existing Producer-Consumer pattern



Eventing usage example

1. Touch down registered by ground radar and transmitted to AODB (system message)
2. If flight has no belt assigned, “Landing & No Belt” notification is created (based in internal event)
3. User can drag notification from panel onto correct belt for assignment and confirmation of message
4. AODB syncs data to the FIDS (external event)



Eventing vs. Batch

- On the legacy system:
  - Time-triggered batch-processing every 5 minutes
- Advantages
  - Fail-Safe through retry
  - Quicker transaction (user wait)
- Disadvantages
  - Hard to trace
  - Time-delay (transaction X starts action Y after 5min.)
  - Testability
  - Parallel operations of eventing (new system) vs. batching (legacy system)

External configuration and functionality in AODB

- Reminder
  - Benutzerbenachrichtigung auf Basis von System-Events (potentiell zeitverzögert und wiederholt)
  - Zur Laufzeit änderbares Verhalten via Spring Expression Language (EL)
  - Bsp: #{flight.isCanceled && blockOffTime.isAfterNow}
- Application Properties
  - Mehrstufige Auflösung:
    1. System Property
    2. Spring Context
    3. Datenbank
    4. Laufzeit
  - Konfiguration & Verwaltung über JMX (cache invalidation, …)

Example: AODB user notifications (“Reminders”)

- filter
  - wenn positiv evaluiert wird der Reminder zugestellt
  - z.B. *!#flight.isCancelled()*
- activating events
  - Events, die die Evaluierung des Filters triggern
  - z.B. *LANDING_SET, DIVERSION_DELETE*
- payload
  - Key-Value-Pairs für die Darstellung des Reminders im GUI bzw. an Schnittstellen
  - z.B. registration → *#flight.rotation?.aircraft?.registration*
- deferral
  - Verspätete Auslösung des Reminders. Kann auch wieder deaktiviert werden
- deactivating events
  - Events, die die Zustellung des Reminders verhindern
- recurrence
  - Zeitinterval für wiederholte Zustellung des Reminders

Example: landingMissingBelt

- filter: *!#flight.isCargo() and #flight.belt == null*
- activating event: *LANDING_SET*
- payload
  - aircraftType #flight.actualAircraftType?.codeIata
  - flight #flight.flightNumber
  - Scheduled #flight.scheduledDateTime?.toString()
  - schengen #flight.schengenStatus
  - registration #flight.rotation?.aircraft?.registration
  - pax #flight.paxBooked
  - position #flight.position
  - departureAirport #flight.departureAirportCode
  - load #flight.totalLoad
  - handling #flight.handlingAgent?.toString()

Example: missingDeparture

- filter: *#flight.actualDateTimeOfBlockOff != null and #flight.actualDateTimeOfTakeOff == null*
- activating events: *BLOCKOFF_SET, BLOCKOFF_CHANGE, TAKE_OFF_DELETE*
- payload
  - flight #flight.flightNumber
  - bo_time #flight.actualDateTimeOfBlockOff.toString()
- deferral: *#flight.actualDateTimeOfBlockOff.plusMinutes(#flight.deicing .isActive() ? 90 : (#isInTimeRange (#flight.scheduledDateTime, 3, 31, 11, 1) ? 20 : 40))*
- deactivating events: *TAKE_OFF_SET, BLOCKOFF_DELETE, BLOCKOFF_CHANGE*
- recurrence: *300*

References

- http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/aydin.pdf
- http://martinfowler.com/articles/injection.html
- http://www.apress.com/9781590596432
- http://c2.com/cgi/wiki?EventDrivenProgramming