Space Based Computing WS15/16
Drones factory

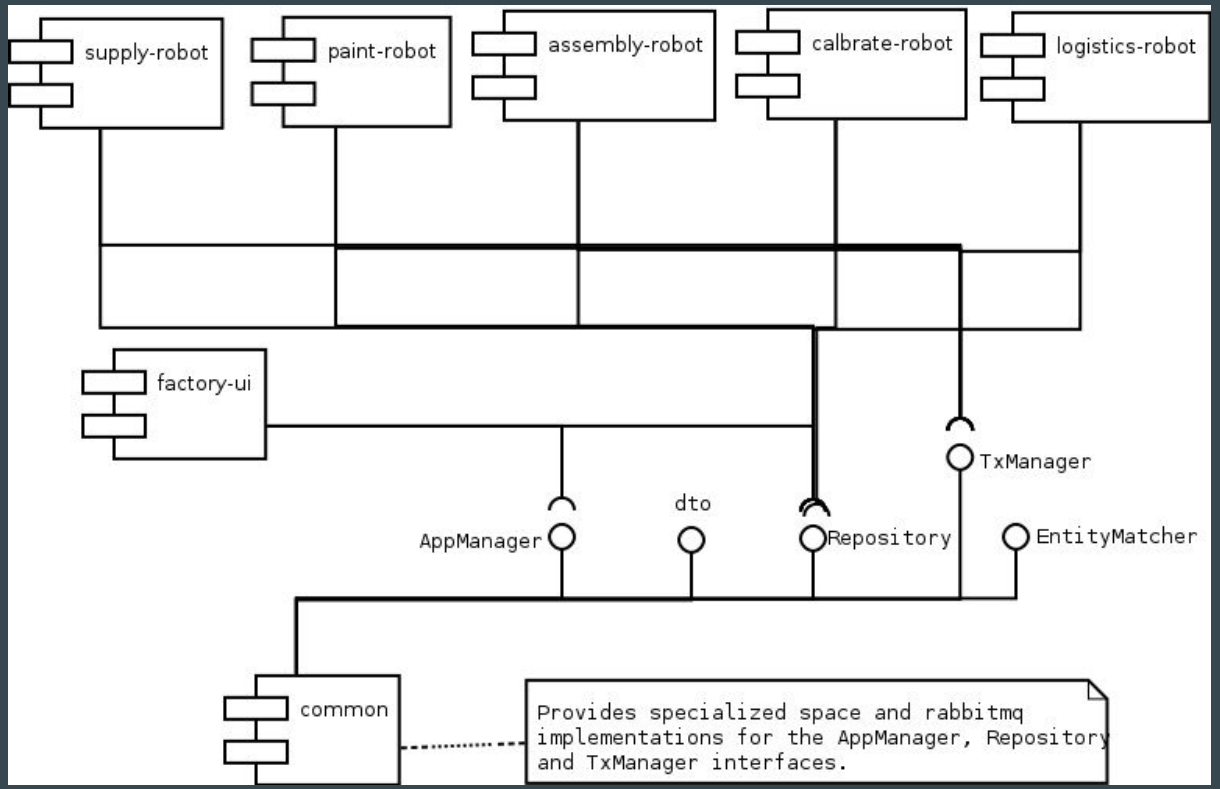
Mihai Alexandru Lepadat (1127960)
Valentin - Mihai Neacsu (1127157)

Used technologies

- space based implementation with **MozartSpaces**
  - more powerful and more flexible than JavaSpaces
  - unlike JavaSpaces it allows integration with other components not implemented in java
  - fairly easy/intuitive to use
  - does not require sophisticated infrastructure setup/configuration
- alternative solution with **RabbitMQ**
  - we chose a message queues solution in favor of others (socket, RPC, ...), because it already offered most of the features we needed, such as transactions
  - again, we wanted a solution capable of integrating with non-java processes, therefore we chose rabbitmq over jms
  - it implements a well know open standard (AMQP)
  - fairly straightforward infrastructure setup: using the rabbitmq docker image from Docker Hub

Components overview

- Our goals were:
  - reuse as much code as possible between the two implementations
  - be flexible



Abstraction layer for coordination strategies

- **Repository:**
  - storeEntity(Serializable)
  - take(EntityMatcher)
  - readAll(EntityMatcher)
  - count(EntityMatcher)
  - onEntityStored(EntityMatcher, Consumer)
  - onEntityTaken(EntityMatcher, Consumer)
- **EntityMatcher:**
  - matches based on entity class and rules for fields (equals or not equals)
  - translates to Query with MatchMakers and Property rules for the space based solution
  - provides a matches(Serializable) method to check if an entity matches the given criteria -> used for the rabbitmq implementation

Abstraction layer for coordination strategies

- **TxManager:**
  - beginTransaction() -> starts a new transaction and binds it to the current thread. Can be queried using the currentTransaction() method
  - commit()
  - rollback()
  - currentTransaction()
- **AppManager:**
  - prepareInfrastructure() -> starts embedded space server for space based solution or initializes the lock server for the rabbitmq strategy
  - shutdown() -> closes resources for a clean termination

Abstraction layer for coordination strategies

- **Advantages:**
  - offers flexibility and extensibility
  - hides/encapsulates communication details out of the business logic (code reuse)
- **Disadvantages:**
  - while implementing the business logic, we always had to think what constructs would fit for both communication strategies, which sometimes led to small compromises
  - most of the problems/challenges appeared for the rabbitmq solution:
    - we had to use some workarounds to be able to perform a read or count operation, getting the message and then not acknowledge it or with transactional rollbacks with message recovery
    - a separate locking mechanism was needed between the actors to be able to support operations such as read or count -> usage of another technology/framework
    - the FIFO order might not be preserved anymore due to those tricks

Effort estimations

- **~90 hours** for implementing both solutions
- **~245 LOC** for the space based solution specific components
- **~420 LOC** for the alternative solution specific components
- **~1500 LOC** for the common business logic parts

Implementation challenges

- after the first assignment we had to perform quite some important refactoring steps, in order to be able to maintain our first architectural approach:
  - we changed from TypeCoordinator to QueryCoordinator in the space based implementation, which was pretty straightforward -> this led to the EntityMatcher abstraction
  - this abstraction took some time to master for the alternative, RabbitMQ based solution -> it took some trials, until we came up with the separate synchronization mechanism, in order to be able to deal operations such as read/count, because each get operation consumed messages from queue, both when using transactional behaviour
- however, after dealing with those changes, it was fairly easy to implement the new changes, i.e. to extend the business logic

Thanks for your attention!