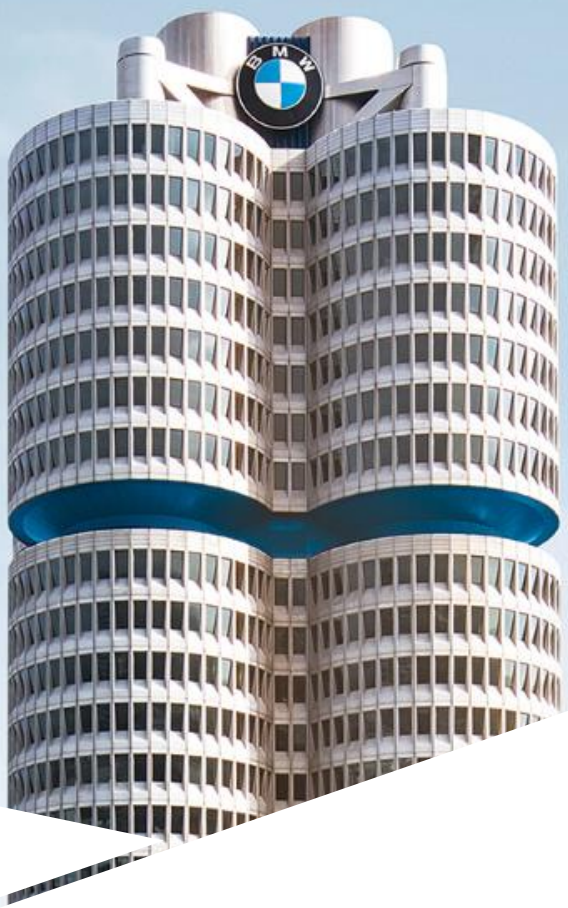
OPENMDM HACKATHON

BEST PRACTICES GRADLE / JEE / REST



Stefan Ebeling, FG-410

April 21st 2016

BMW GROUP THE NEXT 100 YEARS

Rolls-Royce Motor Cars Limited

GRADLE

GRADLE

- Each project (Git Repository) has to contain one overall Gradle build script.
  - It defines all global properties (plugins, maven repositories, dependencies, versions, etc.)
  - Its settings.gradle includes the subprojects.
  - It builds all its subprojects

- Subprojects (components) are built by separate scripts.
  - These scripts define component related stuff only.

- Declare dependencies within the project by `compile project(':org.eclipse.mdm:org.eclipse.mdm.navigator')`

- Prefer standard task before creating new ones.
- Integrate the TypeScript compiler into the build process.

IN GENERAL

LOGGING

Operation monitors these log levels:

- (DEBUG: for developers only, not used in production)
- INFO: Something interesting happened (e.g. Starting a Service).
- WARN: Needs to be observed, but the system is fine (e.g. an expected setting wasn't found, but the system is using a default)
- ERROR: An error which needs to be fixed (mistakes in user input are usually not logged with this level).

FATAL means the server is down. TRACE (finer than DEBUG) is rarely used.

EXCEPTIONS

A component related exception isn't a good candidate. An exception should indicate a technical or even better a business related error.

Integrate these exceptions into the defined structure (usually boundary or entity). Do not create an "exceptions" package.

Do not log an error and rethrow it. The same error should occur only once in a log file.

If the caller can't handle the exception (beside logging it), use unchecked exceptions (extends RuntimeException). When in doubt: Create an unchecked exception.

See: http://stackoverflow.com/questions/27578/when-to-choose-checked-and-unchecked-exceptions

JEE

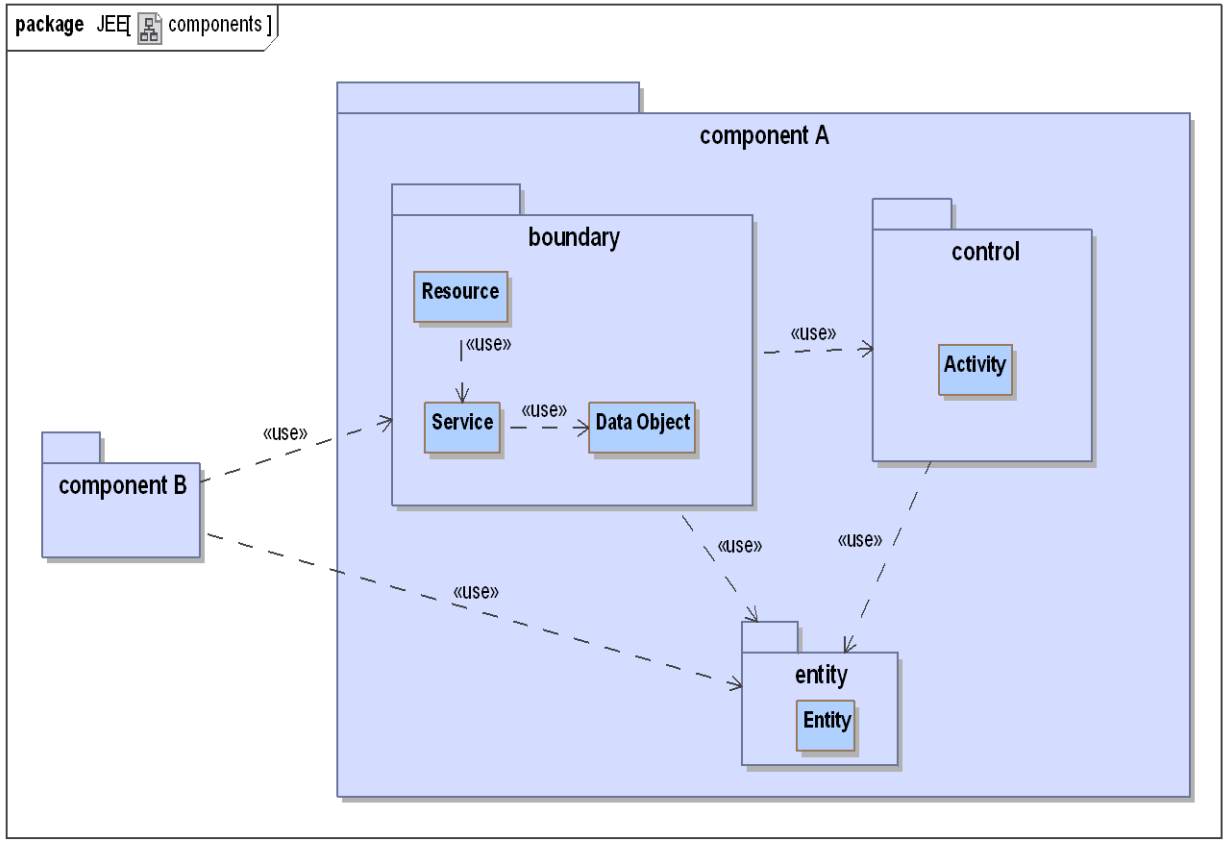
RECAP: ENTITY CONTROL BOUNDARY (ECB)

Entity elements: An entity is a long-lived, passive element that is responsible for some meaningful **chunk of information**. This is not to say that entities are "data," while other design elements are "function". Entities **perform behavior** organized around some cohesive amount of data.

Control elements: A control element manages the **flow of interaction** of the scenario. A control element could manage the end-to-end behavior of a scenario or it could manage the interactions between a subset of the elements. Behavior and business rules relating to the information relevant to the scenario should be assigned to the entities; the control elements are responsible only for the flow of the scenario.

Boundary elements: A boundary element lies on the **periphery of a system** or subsystem, but within it. For any scenario being considered either across the whole system or within some subsystem, some boundary elements will be "front end" elements that accept input from outside of the area under design, and other elements will be "back end," managing communication to supporting elements outside of the system or subsystem.

UML



- A Resource just takes care for the REST related stuff.
- The Service is responsible for the real work.
- If a Service gets to large, just delegate tasks to stateless Activities. These may be reused.
- Define DataObjects only when needed. It is allowed to expose the entities.
- Entities may define logic (e.g. using their own fields).

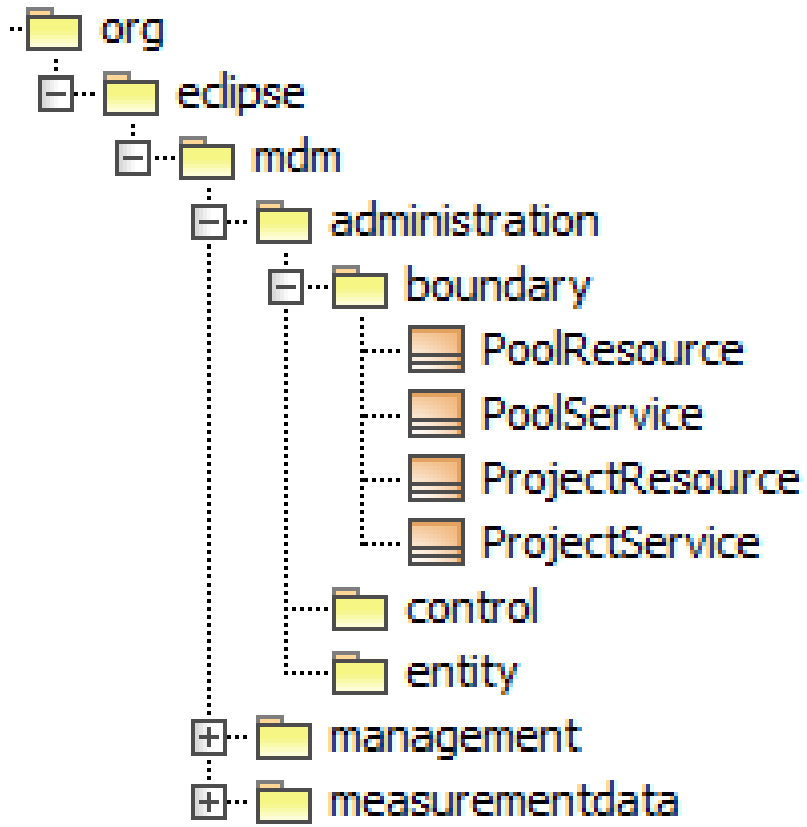
BUSINESS COMPONENTS

Administration org.eclipse.mdm.administration

Management Data org.eclipse.mdm.managementdata

Context Data org.eclipse.mdm.contextdata

etc.



REST

REST: RESOURCES

- A Resource just takes care for the REST related stuff.
- A Resource does not contain any logic. It:
  - Unmarshalls the request,
  - Delegates to its Service and finally
  - Marshalls the response.
- Usually each method just contains a single line of code.

REST: JSON

It's not necessary to build a JSON result manually. Instead of writing

```
return new Gson().toJson(new EntryResponse(Environment.class, list));
```

just annotate the class with:

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class EntryResponse
```

This allows a proper return signature of the method as well:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<EntryResponse> getEnvironments()
```

REST: EXCEPTIONS

Exceptions need to be reported to the client and therefore mapped to HTTP status codes:

- 4xx client error and
- 5xx server error.

The `javax.ws.rs.ext.ExceptionMapper` supports this task (finally log the error here as well). Sample:

```
@Provider
public class AppExceptionMapper implements ExceptionMapper<MyException> {
  public Response toResponse(AppException ex) {
    return Response.status(400).entity(new ErrorMessage(ex))
      .type(MediaType.APPLICATION_JSON).build();
    }
}
```

See http://www.codingpedia.org/ama/error-handling-in-rest-api-with-jersey/

MORE CONVENTIONS (TO BE DISCUSSED)

We should …

- not use the annotations Local and LocalBean as they aren't necessary.
- define interfaces only when necessary (e.g. a second implementation is instantly written).
- follow the package structure defined by ecb.
- create Resources (e.g. Test, TestSteps) and Services containing the business logic (incl. simple actions "delete" and "i18n").