**Information Systems Institute**

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2015 (184.260)

**Assignment 3**

Submission Deadline: 11.6.2015, 18:00

## General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the TUWEL forum) are allowed, but the code has to be written alone. If we find that two students submitted the same, or very similar assignments, these students will be graded with 0 points (no questions asked). We will use automated plagiarism checks to compare solutions. Note that we will also include the submissions of previous years in our plagiarism checks.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution compiles and runs without errors. If you are unsure, test compiling your submission on different computers (e.g., one of the ZID lab computers). Preferred target platform for the entire lab (all 3 assignments) is JDK 7[1]. Before you submit, make sure to test your solution with JDK 7.
- The assignment project is set up using Apache Maven[2]. We provide a project template that contains some code interfaces and JUnit tests, which will assist you in developing your code. Please stick exactly to the provided interfaces as we will check your solutions in an automated test environment. The Maven project is split up into multiple modules which correspond to the different sub-parts of the assignments. **Note:** If all unit tests are passing in your solution, it does *not* necessarily mean that you will receive all the points. We will perform additional code checks and run tests that are not provided in the template. The tests included in the template should merely help you get started and assist you in developing your solution. If you want to further increase your test coverage, you may also add new unit tests, but this is optional. Do not modify any of the pre-defined tests - in any case, we will check your code with the original tests from the template.
- The root folder of the template contains the Maven metadata file (`pom.xml`) and four submodule directories (`ass3-aop`, `ass3-event`, `ass3-jms`, `ass3-shared`). You can simply pull these additional submodules and the updated pom file from the provided git-repository. The dependencies should then be automatically set up by Maven.
- For the AspectJ development in Task 3, you can use the Springsource Toolsuite (STS)[3], although we do not require it. However, STS eases the development of aspects and advices significantly, as STS includes tooling for syntax-checking and matching of joinpoint definitions (i.e., STS tells you without running the application what your current joinpoints actually match in your application).
- We use (as can be seen in the persistence-unit configuration of persistence.xml) a H2[4] database named `dst`. You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Again: make sure that all settings are as expected before you submit!
- Please make sure to add reasonable logging output to help us keep track of what your solution does. No debug output is very bad, and too much (e.g., many screen pages) is just as bad. Aim for a good middle ground, which allows us to check your solution quickly.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/`
[2] `http://maven.apache.org/`
[3] `http://spring.io/tools`
[4] `http://www.h2database.com/html/main.html`

# A. Code Part

| | |
|---|---|
| **Command for testing part 1.:** | `mvn install -Pass3-jms` |
| **Command for testing part 2.:** | `mvn install -Pass3-event` |
| **Command for testing part 3.:** | `mvn install -Pass3-aop` |

## 3.1. Messaging (18 Points)

In this task you will create a simple JMS-based messaging application for the MOC platform management system.

In the last assignment we have focused on assigning and streaming lectures for a given number of students. Now it is time to have a look at how the system streams lectures where the number of students is not known in advance or was simply not specified. For this purpose, we will build a messaging system based on JMS and Message-Driven Beans (MDB). The communication should happen via **queues** and **topics**.

After a lecture is assigned by a lecturer, the platform's scheduler takes care of scheduling and streaming the lecture. To that end, the scheduler creates a new lecture-wrapper (which wraps the lecture) and sends it to the virtual-schools. One of the virtual-schools takes the lecture, classifies it and decides whether one of its classrooms can be used for streaming this lecture or not. The scheduler shall be informed if the lecture cannot be streamed. In the other case (i.e., if the lecture can be streamed), the lecture is forwarded to the classrooms. Every classroom in our system belongs to one virtual school and is responsible for a certain type of lectures. According to that, the lecture is streamed in the respective classroom. For example if virtual-school vs1 classifies a lecture as `PRESENTATION`, the lecture is streamed in a classroom that belongs to virtual-school vs1 and is responsible for `PRESENTATION` lectures. After the lecture has been streamed in the classroom, the scheduler is informed.

The steps in the procedure above are performed automatically in the provided test cases. As usual, feel free to extend the template by adding your own test classes (however, avoid changing existing tests).

**Your task in the following is to design and implement the described communication based on a message queuing approach**.

To keep things simple, there is only one persistent entity you need to manage in the application this time: the `LectureWrapper` (Figure 1). Besides the mandatory id, this entity contains information about the lecture, a state, the name of the virtual-school, which was responsible for classifying the lecture, as well as the type of the lecture. The state field represents the current state of the lecture in the process described above. The type field is set after the virtual-school in charge has classified the lecture. You do not need to implement these new entities – the interfaces and implementing classes are provided in the template (see module **ass3-shared**).

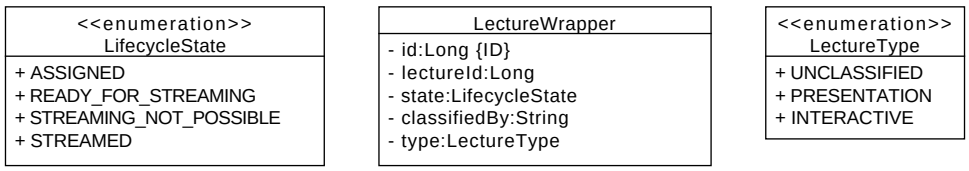


Figure 1: Task Entity

In our scenario a **server** provides the communication infrastructure. Whenever messages are exchanged (between the scheduler and the virtual-schools or the virtual-school and the classroom), the server is responsible for forwarding the messages. **Clients never communicate directly with each other**. This way it is also possible to keep the information about lectures up to date and to store the new state (and any other information that may have changed) to the database.

In total, the clients of our messaging system are (1) the scheduler, (2) possibly multiple virtual-schools and (3) various classrooms, all of which may act as senders and receivers. Implement the `JMSFactory` methods for instantiating these components (interfaces are provided). Put your implementation of the respective interfaces into `impl` subpackages.

**Note:** For this assignment, we use an embedded **OpenEJB**[5] container, which is automatically started with the tests when running "mvn install -Pass3-jms". In OpenEJB you can use the standard injection annotations like `@Resource`, `@PersistenceContext`, etc. The container should be able to automatically find and initialize your annotated resources (e.g., message queues and topics). Additionally, there is a `@org.apache.openejb.api.LocalClient` annotation, which allows to inject resources into self-contained objects that are not managed by the container. Make sure that the self-contained "client" components in our architecture (scheduler, virtual-schools, classrooms) are annotated with `@LocalClient`. This ensures that these components can also be started in separate JVM instances, which do not have the classpath dependency to OpenEJB. You may want to take a look at the `dst.ass3.AbstractJMSTest` class, to see how the container and all components are started/initialized.

### 3.1.a. Scheduler

Our platform contains a single scheduler, which is responsible for assigning new lectures and providing information about existing lectures. Sending and receiving messages takes place concurrently. Use the method `IScheduler.start` to initialize the messaging infrastructure (lookup queues, create connections, etc.). Implement `IScheduler.stop` to close connections and release all resources. In addition, the interface in the template provides the following core methods (also refer to the comments inside the interface file):

- `IScheduler.assign(long lectureId)`: Advises the server to create a new lecture. The server creates an entry in the database and automatically forwards the lecture to the next available virtual-school. The state of the lecture is set to `ASSIGNED` and the type is set to `UNCLASSIFIED`. In return to this command, the scheduler (asynchronously) receives an object message with a `LectureWrapperDTO` object, which contains the *id* of the newly created lecture.
- `IScheduler.info(long lectureWrapperId)`: Advises the server to send information about a lecture identified by the respective id. This data must also contain all the relevant fields.
- `IScheduler.setSchedulerListener(ISchedulerListener listener)`: Sets a listener for the scheduler. The scheduler listener is implemented by the test classes provided in the template; the listener is used to notify about incoming messages from the server (`CREATED, INFO, STREAMED, DENIED`).
- `ISchedulerListener.notify(InfoType type, LectureWrapperDTO lectureWrapper)`: Use this method in your `IScheduler` implementation to notify about status messages received from the server. Note that you do not need to implement `ISchedulerListener` - this interface is implemented by the test classes and serves as the link between the tests and your code.

### 3.1.b. Virtual-Schools

There may be several virtual-schools listening to the server concurrently. Each of the virtual-schools is identified via a unique name. The server does not know which or how many virtual-schools there are at any given moment, but you can assume there will be at least one at any time. However, it is absolutely important that every lecture is handled by exactly one virtual-school **(not more!)**.

Once a virtual-school is entrusted with a new lecture (which is simply printed out to the console), the virtual-school automatically stops listening to the server, processes the lecture, and after finishing processing starts listening again. **A virtual-school is never handling more than one lecture at any time. However, it can handle several lectures throughout its entire existence**. In the following we discuss the key interface methods:

[5] `http://tomee.apache.org`

- `IVirtualSchool.setVirtualSchoolListener(IVirtualSchoolListener listener)`: Sets a listener for this virtual-school. The virtual-school listener is implemented by the test classes provided in the template; use the listener to determine whether a lecture should be accepted or not, and to find out which type (either `PRESENTATION` or `INTERACTIVE`) should be assigned to the lecture.
- `IVirtualSchoolListener.decideLecture(ClassifyLectureWrapperDTO lectureWrapper, String virtualSchoolName)`: Use this method in your `IVirtualSchool` implementation to decide on how a lecture should be streamed. Note that you do not need to implement `IVirtualSchoolListener` - this interface is implemented by the test classes and serves as the link between the tests and your code.

Each time your virtual-school receives a lecture from the scheduler, invoke the listener's `decideLecture` method to find out how to proceed. Then, based on the outcome of `decideLecture`, the virtual-school should automatically add all required information to the `TypeLectureWrapperDTO`. First, update the `classifiedBy` field (the name of the virtual-school). Then, if the listener's decision indicates `ACCEPT`, update the type and set the state to `READY_FOR_STREAMING` in the database. After the successful update, pass the lecture on to the classrooms. Otherwise, if the virtual-school is not able to stream this lecture (i.e., `DENY` decision), do not assign the lecture to the classrooms, but update the state to `STREAMING_NOT_POSSIBLE` in the database (and do not update the type field). After the update, inform the scheduler about the denied lecture. On the scheduler side, call the listener's `notify` method to inform the test framework about the denied lecture.

### 3.1.c. Classrooms

Every classroom belongs to one virtual-school and is responsible for exactly one lecture type. The server defines a special communication endpoint to let all classrooms listen for lectures that 1) are classified by the virtual-school they belong to, and 2) have the type they are responsible for. Therefore, the server application can simply label the request with the virtual-school's name (the lecture's `classifiedBy` field), the type of the lecture and propagate it to this endpoint. You can assume that there is at least one classroom (possibly more) listening for a certain virtual-school and type, and that all responsible classrooms stream the lecture simultaneously and collaboratively.

The classroom must be designed in a way that it only receives messages it is responsible for (virtual-school and type), using the labels the server added to the message (check the possibilities to do this with JMS). Your infrastructure should also be able to deal with classrooms that are currently not listening, otherwise the request might get lost. The core interface methods are as follows:

- `IClassroom.setClassroomListener(IClassroomListener listener)`: Similar to the listeners for virtual-schools, the test framework in the template also injects listeners for classrooms. The classroom listener provides a means to wait until the (simulated) lecture streaming is finished.
- `IClassroomListener.waitTillStreamed(StreamLectureWrapperDTO lectureWrapper, String classroomName, LectureType acceptedType, String virtualSchoolName)`:
  This method blocks until a lecture has successfully finished the streaming in a classroom. Again, this listener interface is implemented by the tests (i.e., you do not have to implement it); simply use the listener in your classroom instance to simulate the waiting time. After the simulated lecture streaming has finished, the classroom can finally update the lecture's state to `STREAMED` and inform the scheduler. You can assume that the classrooms (which host a lecture collaboratively) coordinate themselves to make sure each classroom is free before this command is sent. The server performs the state change immediately, which means that only one of the classrooms has to send the `streamed` command. Hence, if more than one of the involved classrooms send this command, the first received command leads to the state change and the remaining ones have no effect (since the state is already set to `STREAMED`).

You should now think about an appropriate message queuing infrastructure and decide about the features the respective queues and topics should provide to their clients. Your solution has to satisfy all the requirements stated above and should be as simple as possible. After configuring this communication infrastructure, implement the three components: scheduler, virtual-school, and classroom.

Note: The persistent entities should never be used for transmission directly. Therefore, we provide DTOs in the template, containing only the relevant information (plain text messages are not sufficient in this assignment, you should use object messages). The server has to update the information about the persistent lectures every time it retrieves relevant messages.

In case of failures (like unknown lecture ids, commands to already streamed lectures, ...) no distributed communication is necessary. However, your application should be able to deal with such sort of requests.

## 3.2. Complex Event Processing (12 Points)

In the following we take a closer look at Complex Event Processing (CEP), a technique that is becoming more and more important in today's business processes and loosely coupled distributed systems. Generally speaking, an "event" is anything (i.e., a phenomenon, happening or similar) that is of interest for the application. CEP collectively refers to techniques for processing of event messages, including event routing, event aggregation, event pattern detection, etc.

The focus of this exercise is to perform event-based queries (which continuously process the new incoming events) and identify event patterns related to the processing of platform lectures in our scenario. For this exercise, we slightly extend the processing flow of the messaging example, and assume that each `LectureWrapper` assigned to the platform can traverse back and forth between states during its lifetime. In particular, we assume that a `LectureWrapper` changes from state `ASSIGNED` to `READY_FOR_STREAMING`, and then either to state `STREAMED` (if everything goes well), or on to state `STREAMING_NOT_POSSIBLE` and back to `READY_FOR_STREAMING` (if an error occurs during streaming). Overall, we are interested in the transition between states `ASSIGNED` and `STREAMED`, and in the transition(s) between states `READY_FOR_STREAMING` and `STREAMING_NOT_POSSIBLE`.

Your implementation will be based on Esper[6], a popular open-source engine for CEP. Esper provides the Event Processing Language (EPL), a powerful SQL-like language, which covers numerous features tailored to efficient processing of queries over event streams. The detailed Esper reference can be found here[7]. A good starting point for your own work is the Esper tutorial[8]. The required dependencies are already included in the template. This time, you do not need to store any entities to the database; simply keep the `LectureWrapper` instances in memory using Esper. For simplicity, the following description speaks of `LectureWrapper` instances, but in fact you should again use the `LectureWrapper` DTOs (i.e., do not feed the entity classes into Esper).

The core classes/interfaces required in your code to properly interact with the test framework are `IEventProcessing` (initialization of queries, publication of events) and `EventingFactory` (object instantiation).

The detailed requirements are as follows:

- Initialize the Esper platform and obtain an instance of `EPServiceProvider` and `EPAdministrator`. Register the `LectureWrapper` class from exercise 1 of this assignment as an event type with Esper (please use the constants in `dst.ass3.event.Constants`). Provide a method to pass `LectureWrapper` objects as events to Esper. You may choose an arbitrary name for the `EPServiceProvider` instance, e.g., "EsperEngineDST".
- Use the dynamic type definition capabilities of Esper (hint: `"create schema ..."`) to define three new event types. The names of the event types are `LectureAssigned`, `LectureStreamed` and `LectureDuration` (please use the constants in `dst.ass3.event.Constants`). Each of the three types should have a `lectureId` property (which corresponds to `lectureId` in the `LectureWrapper` class). `LectureAssigned` and `LectureStreamed` have a `timestamp` property, and `LectureDuration` has a `duration` property. All of the mentioned properties are of type `long`. Note that you should register these event types in Esper *without* creating any corresponding Java classes.

[6] `http://www.espertech.com/esper/`
[7] `http://www.espertech.com/esper/release-5.2.0/esper-reference/html/index.html`
[8] `http://www.espertech.com/esper/tutorial.php`

- Create and execute 3 EPL queries, which generate events for the three types `LectureAssigned`, `LectureStreamed` and `LectureDuration`. Evidently, `LectureAssigned` is triggered when a `LectureWrapper` event with status `ASSIGNED` occurs, and `LectureStreamed` is triggered in case of a `LectureWrapper` event with status `STREAMED`. The `LectureDuration` events combine the information of the two previous lecture-wrapper types and contain the streaming duration. Be sure to correlate the `lectureId` property of the `LectureAssigned` and `LectureStreamed` types in your query. You can assume that the `LectureDuration` query does not need to consider more than 10.000 historical events (i.e., there are never more than 10.000 events between any two correlated `LectureWrapper` events).
- Implement the following three EPL queries and provide a listener, which outputs (to stdout) new events that result from executing the queries:
  - Receive notifications about all events of type `LectureDuration`.
  - Upon arrival of each `LectureDuration` event, emit a `AvgLectureDuration` event (string defined in `dst.ass3.event.Constants`) that computes the average duration over all lecture-wrappers that finished within the last 15 seconds.
  - Use pattern matching facilities of EPL to detect lecture-wrappers which have 3 times attempted and failed to run (i.e., switched 3 times between the state `READY_FOR_STREAMING` and the state `STREAMING_NOT_POSSIBLE`).
- The test classes provided in the template simulate various non-trivial sequences of input events of type `LectureWrapper`. The events are passed to your implementation of `IEventProcessing`, which is instantiated by the test classes using the factory `EventingFactory`. You can feed the received events directly to the Esper querier, i.e., for this task you do not need to use the messaging infrastructure from earlier to transmit and modify the status of the `LectureWrapper`(s). Be sure to thoroughly test your solution - event-based and asynchronous processing is inherently prone to timing and synchronization faults.

Note that the example queries above cover only a very small subset of Esper's capabilities. Use the Esper reference to get familiar with some of the additional core concepts (e.g., *contexts*). During the interview sessions you should be prepared to report on your experiences with CEP and the strengths/weaknesses of Esper. Also, think about the core differences of the Esper processing model as opposed to querying a standard database like MySQL.

## 3.3. Dynamic Plugins Using AspectJ (14 Points)

Another important aspect of modern application servers is the dynamic loading and deployment of plugins or applications. This feature is also a nice demonstration for using reflection and class loading at runtime, so we will again implement a (simplified) custom solution of our own. Put the code for this task into the **ass3-aop** subproject.

- **3.3.a. Plugin executor (4 Points)**

  Implement the `IPluginExecutor` interface provided in the template. To allow the test framework instantiate instances of this interface, implement the factory method in `PluginExecutorFactory`. `IPluginExecutor` is the main component responsible for executing plugins. It has to monitor (i.e., repeatedly list the contents of) several directories to detect whether new **.jar** files were copied to these directories or existing .jar files were modified. (Note: In addition to monitoring for changes, also check the directories once when initializing your application. Note 2: Monitoring of subdirectories is *not* required. Note 3: In case you choose to employ the Java `WatchService`, be aware that at file creation, the service (depending on the OS) might fire two events (`ENTRY_MODIFY` and `ENTRY_CREATE`) for the same file - please make sure that you only start a plugin once in this case.). The executor then scans the file and looks for classes that implement the `IPluginExecutable` interface. If some plugin executable is found, the executor spawns a new thread and calls its execute method. For this, you should be using a thread pool. Take care of class loading: there must not be any problem with the concurrent execution of different plugins containing classes with equal names.

Also make sure to free all acquired resources after the execution of a plugin has been completed. The second method in the interface (`IPluginExecutor.interrupted()`) will be important later on, you can leave this method body empty for now. Note: if you detect that a plugin .jar file has changed and the plugin is currently still executing (the code in the previous version of the .jar file), you do not need to terminate the existing plugin instance (i.e., you can simply start a new version of the plugin with the updated code, and let the old one terminate normally).

- **3.3.b. Logging Plugin Executions (6 Points)**

  Now we want to implement some (decoupled) logging facilities for our plugin executor framework. To this end, we will make use of AspectJ and Aspect-Oriented Programming (AOP). The required AspectJ dependencies are already part of the Maven template. The tests in the template have been configured to use *run-time weaving* (i.e., objects are instantiated and then the aspects are dynamically weaved into the underlying class definitions of these objects). One alternative would be *load-time weaving* (i.e., using the Java agent mechanism to weave aspects into the classes at class loading time), which we do not use for this assignment.

  Before starting to develop, you should familiarize yourself with the concepts of aspects, advices, joinpoints and pointcuts (you can also tackle Theory Question 5 as you go along). Then, refer to the AspectJ Development Kit Developer's Notebook[9] for support on how these concepts are implemented in AspectJ. Use the annotation-based development style[10] to define your aspects.

  Your first task is now to write a simple logging aspect for plugins. Essentially, the aspect should write a single line of logging output before a plugin starts to execute, and after a plugin is finished. The bare class definition of `LoggingAspect` is already included in the template - add the required methods and annotations to this class. The log message can be very brief, but needs to contain the actual class name of the plugin:

```
[java] Plugin dst3.dynload.sample.PluginExecutable started to execute
[java] Plugin dst3.dynload.sample.PluginExecutable is finished
```

  In some cases, users of the plugin framework might want to disable logging for some plugins. The template defines a method annotation `Invisible`. Whenever an `IPluginExecutable.execute()` is annotated as invisible, its execution should not be logged. Make sure that this condition is already considered in the pointcut definition of your logging advice (i.e., you should **not** match just any plugin method and filter out invisible plugins in your Java code).

  Additionally, your logging aspect should re-use the logger of the plugin, if the plugin has defined one. That is, if the plugin has a member field of a subclass of `java.util.logging.Logger`, your log statements should be written to that logger. If no such logger is defined, use `System.out`. Configure the logging system to print all log messages of at least level INFO or higher.

- **3.3.c. Plugin Performance Management (4 Points)**

  Plugin frameworks like the one we are implementing often need some way to influence the execution of the managed plugins. Hence, we now implement some means to interrupt plugins whose execution takes too long.

  The template already defines a method annotation (`Timeout`), which has one Long parameter. Users can use this annotation on `IPluginExecutable.execute()` methods to define the "normal" maximum execution time (in milliseconds) of their plugins. Then, annotate and implement the aspect class `ManagementAspect` (see template) to enforce this defined maximum execution time. Have the aspect hook into each invocation of `IPluginExecutable.execute()` to keep track of the currently running plugins and their start time. From the start time you can then derive the current execution duration, e.g., by polling in regular intervals or using a timer task. If a plugin is detected

[9] `http://www.eclipse.org/aspectj/doc/released/adk15notebook/`
[10] `http://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj.html`

that takes longer than its maximum defined time, call this plugin's `interrupted()` method. You do not need to take any further action (i.e., we can assume that the developer of the plugin actually terminates the plugin if this callback is invoked). However, keep in mind that the `interrupted()` method can itself take some time to execute, so do not block while the client is cleaning up. If no timeout is defined for a plugin, you can assume that the plugin can run for as long as it needs to.

The package `dst.ass3.aop.sample` in the template contains a couple of simple plugin examples for testing. You may optionally add your own plugin classes to test your solution more thoroughly.

# B. Theory Part

The following questions will be discussed during the practice lesson. At the beginning of the each lesson we hand out a list where you can specify which questions you have prepared and are willing to present. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a correct and well-founded answer, you will lose **all** points for the theory part of this assignment.

## 3.4. Class loading (1 point)

Explain the concept of class loading in Java. What different types of class loaders do exist and how do they relate to each other? How is a class identified in this process? What are the reasons for developers to write their own class loaders?

## 3.5. AOP Fundamentals (2 points)

Explain the concept of Aspect Oriented Programming (AOP). Think of typical usage scenarios. What are aspects, concerns, pointcuts and joinpoints, and how do these concepts relate to each other? Why is it so important to write minimally matching pointcut definitions?

## 3.6. Weaving Times in AspectJ (1 point)

What happens during weaving in AOP? At what times can weaving happen in AspectJ? Think about advantages and disadvantages of different weaving times.

## 3.7. Esper Processing Model (2 point)

Study the details of the Esper processing model (available in the online reference of Esper). Describe the core API elements, and illustrate the main EPL query types based on an exemplary event timeline.