openTCS

Developer's Guide

The openTCS developers

openTCS 4.20.0

Table of Contents

Chapter 1. Development with openTCS in general

1.1. System requirements

The openTCS source code is written in Java. To compile it, you need a Java Development Kit (JDK) 1.8. To run the resulting binaries, you need a Java Runtime Environment (JRE) 1.8. All other required libraries are included in the openTCS distribution or will be downloaded automatically when building it from source code.

1.2. Available artifacts and API compatibility

The openTCS project publishes artifacts for releases via JCenter, so you can easily integrate them with build systems such as Gradle or Maven. In Gradle build scripts, for example, use something like the following to integrate an openTCS library:

```
repositories {
  jcenter()
}

dependencies {
  compile group: 'org.opentcs', name: '${ARTIFACT}', version: '4.20.0'
}
```

Set the version number of the openTCS release you actually want to work with, and select the appropriate `${ARTIFACT}` name from the following table:

Table 1. Artifacts published by the openTCS project

Artifact name	API compatibility between minor releases	Content
`opentcs-api-base`	Yes	The base API for clients and extensions. This is what most developers probably want to use.
`opentcs-api-injection`	Yes	API interfaces and classes used for dependency injection within the kernel and plant overview applications. This is required in integration projects customizing these applications, e.g. adding components like vehicle driver implementations.
`opentcs-common`	No	A collection of utility classes used by openTCS components.
`opentcs-impl-configuration-cfg4j`	No	An implementation of the base API's configuration interfaces based on cfg4j.

Artifact name	API compatibility between minor releases	Content
opentcs-kernel-extension-http-services	No	A kernel extension providing the web API implementation.
opentcs-kernel-extension-rmi-services	No	A kernel extension providing the RMI interface implementation.
opentcs-kernel-extension-statistics	No	A kernel extension providing the statistics collection implementation.
opentcs-kernel-extension-tcp-host-interface	No	A kernel extension providing the (deprecated) TCP/IP host interface implementation.
opentcs-plantoverview-panel-loadgenerator	No	The load generator panel implementation for the plant overview.
opentcs-plantoverview-panel-resourceallocation	No	The resource allocation panel implemenation for the plant overview.
opentcs-plantoverview-panel-statistics	No	The statistics panel implementation for the plant overview.
opentcs-plantoverview-themes-default	No	The default themes implementation for the plant overview.
opentcs-commadapter-loopback	No	A very basic vehicle driver simulating a virtual vehicle.
opentcs-strategies-default	No	The default implementations of strategies that are used by the kernel application.
opentcs-kernel	No	The kernel application.
opentcs-kernelcontrolcenter	No	The kernel control center application.
opentcs-plantoverview	No	The plant overview application.

Note that only the basic API libraries provide a documented API that the openTCS developers try to keep compatible between minor releases. (For these libraries, the rules of semantic versioning are applied.) All other artifacts' contents can and will change regardless of any compatibility concerns, so if you use these and switch to a different version of openTCS, you may have to adjust and recompile your code.

1.3. Third-party dependencies

The kernel and the client applications depend on the following external frameworks and libraries:

- SLF4J (https://www.slf4j.org/): A simple logging facade to keep the actual logging implementation replaceable.
- Google Guice (https://github.com/google/guice): A light-weight dependency injection framework.

- Cfg4j (http://www.cfg4j.org/): A configuration library supporting binding interfaces.
- Google Guava (https://github.com/google/guava): A collection of small helper classes and methods.

The kernel application also depends on the following libraries:

- JGraphT (http://jgrapht.org/): A library for working with graphs and using algorithms on them.
- Spark (http://sparkjava.com/): A framework for creating web applications.
- Jackson (https://github.com/FasterXML/jackson): Provides JSON bindings for Java objects.
- JDOM (http://www.jdom.org/): A library for reading and writing XML data.

The plant overview application has the following additional dependencies:

- JHotDraw (http://www.jhotdraw.org/): A framework for drawing graph structures (like driving course models).
- Docking Frames (http://www.docking-frames.org/): A framework for docking and undocking of GUI panels

For automatic tests, the following dependencies are used:

- JUnit (https://junit.org/): A simple unit-testing framework.
- Mockito (http://mockito.org/): A framework for creating mock objects.
- Hamcrest (http://hamcrest.org/): A framework for assertion matchers that can be used in tests.

The artifacts for these dependencies are downloaded automatically when building the applications.

1.4. Modularity and extensibility

The openTCS project heavily relies on Guice for dependency injection and wiring of components as well as for providing plugin-like extension mechanisms. In the injection API, relevant classes can be found in the package `org.opentcs.customizations`. For examples, see Customizing and extending the kernel application, Customizing and extending the plant overview application and Customizing and extending the control center application.

1.5. Logging

The code in the official openTCS distribution uses SLF4J for logging. Thus, the actual logging implementation is easily interchangeable by replacing the SLF4J binding in the respective application's classpath. The kernel and plant overview client applications come with SLF4J's bindings for `java.util.logging` by default. For more information on how to change the actual logging implementation, e.g. to use log4j, please see the SLF4J documentation.

1.6. Working with the openTCS source code

The openTCS project itself uses Gradle as its build management tool. To build openTCS from source code, just run `gradlew build` from the source distribution's main directory. For details on how to work with Gradle, please see its documentation.

These are the main Gradle tasks of the root project you need to know to get started:

- `build`: Compiles the source code of all subprojects.
- `release`: Builds and packages all system components to a distribution in `build/`.
- `clean`: Cleans up everything produced by the other tasks.

To work with the source code in your IDE, see the IDE's documentation for Gradle integration. There is no general recommendation for any specific IDE. Note, however, that the openTCS source code contains GUI components that have been created with the NetBeans GUI builder. If you want to edit these, you may want to use the NetBeans IDE.

In case you use NetBeans, install the Gradle Support plugin from the NetBeans plugin portal. You should then be able to open the source distribution's root directory as a Gradle project and compile and run the components from within NetBeans.

1.7. openTCS kernel APIs

openTCS provides the following APIs to interact with the kernel:

- The kernel's Java API for both extending the kernel application as well as interfacing with it via RMI. See The kernel's Java API for details.
- A web API for interfacing with the kernel via HTTP calls. See the separate interface documentation that is part of the openTCS distribution for details.
- A legacy TCP/IP-based interface for creating transport orders and fetching status information. See TCP/IP-based interfaces to other systems for details.
  Note that this interface is deprecated in favour of the web API and will be removed with the release of openTCS 5.0.

Chapter 2. The kernel’s Java API

The interfaces and classes required to use the kernel API are part of the `opentcs-api-base` JAR file, so you should add that to your classpath/declare a dependency on it. (See Available artifacts and API compatibility.) The basic data structures for plant model components and transport orders you will encounter often are:

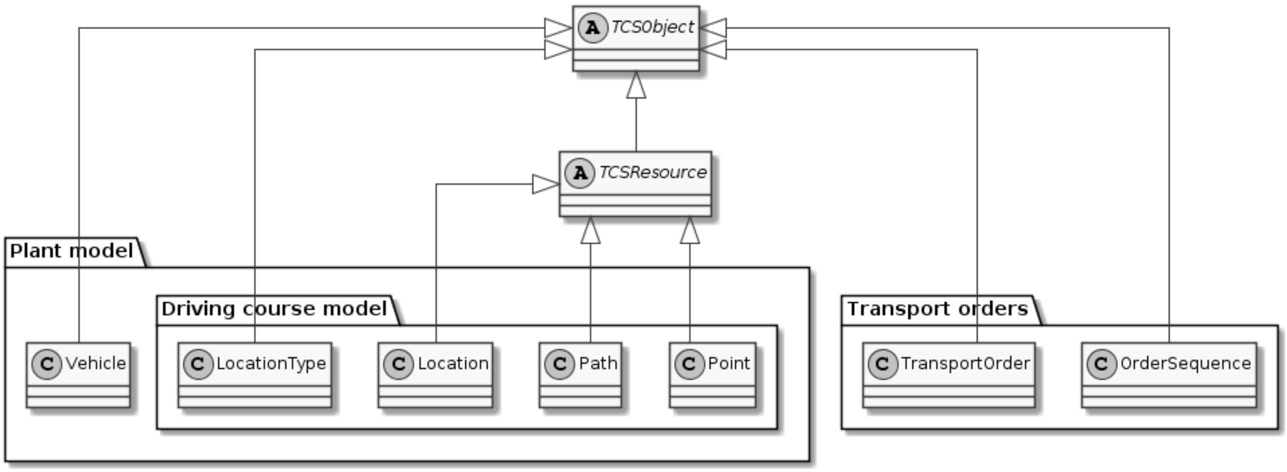


Figure 1. Basic data structures

The service interfaces that are most often interacted with to fetch and manipulate such objects are:

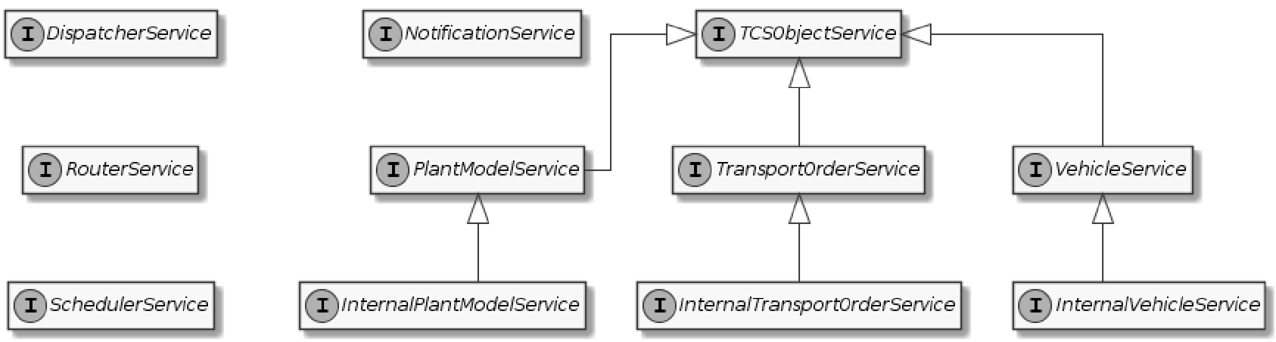


Figure 2. Service interfaces

2.1. Acquiring service objects

To use the services in code running inside the kernel JVM, e.g. a vehicle driver, simply request an instance of e.g. `PlantModelService` to be provided via dependency injection. You may also work with an instance of `InternalPlantModelService` here, which provides additional methods available only to kernel application components.

To access the services from another JVM, e.g. in a client that is supposed to create transport orders or to receive status updates for transport orders or vehicles, you need to connect to them via Remote Method Invocation (RMI). The easiest way to do this is by creating an instance of the `KernelServicePortalBuilder` class and letting it build a `KernelServicePortal` instance for you. (For now, there isn’t much support for user management, so it is recommended to ignore the methods that require user credentials.) After creating the `KernelServicePortal` instance, you can use it to get service instances and fetch kernel events from it. See also the class documentation for `KernelServicePortalBuilder` in the base API's JavaDoc documentation.

```
KernelServicePortal servicePortal = new KernelServicePortalBuilder().build();

// Connect and log in with a kernel somewhere.
servicePortal.login("someHost", 1099);

// Get a reference to the plant model service...
PlantModelService plantModelService = servicePortal.getPlantModelService();
// ...and find out the name of the currently loaded model.
String modelName = plantModelService.getLoadedModelName();

// Poll events, waiting up to a second if none are currently there.
// This should be done periodically, and probably in a separate thread.
List<Object> events = servicePortal.fetchEvents(1000);
```

2.2. Working with transport orders

A transport order, represented by an instance of the class `TransportOrder`, describes a process to be executed by a vehicle. Usually, this process is an actual transport of goods from one location to another. A `TransportOrder` may, however, also just describe a vehicle's movement to a destination position and an optional vehicle operation to be performed.

All of the following are examples for "transport orders" in openTCS, even if nothing is actually being transported:

- A classic order for transporting goods from somewhere to somewhere else:
  a. Move to location "A" and perform operation "Load cargo" there.
  b. Move to location "B" and perform operation "Unload cargo" there.
- Manipulation of transported or stationary goods:
  a. Move to location "A" and perform operation "Drill" there.
  b. Move to location "B" and perform operation "Hammer" there.
- An order to move the vehicle to a parking position:
  a. Move to point "Park 01" (without performing any specific operation).
- An order to recharge the vehicle's battery:
  a. Move to location "Recharge station" and perform operation "Charge battery" there.

2.2.1. A transport order's life cycle

1. When a transport order is created, its initial state is `RAW`.
2. A user/client sets parameters for the transport order that are supposed to influence the

transport process. These parameters may be e.g. the transport order's deadline, the vehicle that is supposed to process the transport order or a set of generic, usually project-specific properties.

3. The transport order is activated, i.e. parameter setup is finished. Its state is set to `ACTIVE`.
4. The kernel's router checks whether routing between the transport order's destinations is possible at all. If yes, its state is changed to `DISPATCHABLE`. If routing is not possible, the transport order is marked as `UNROUTABLE` and not processed any further.
5. The kernel's dispatcher checks whether all requirements for executing the transport order are fulfilled and a vehicle is available for processing it. As long as there are any requirements not yet fulfilled or no vehicle can execute it, the transport order is left waiting.
6. The kernel's dispatcher assigns the transport order to a vehicle for processing. Its state is changed to `BEING_PROCESSED`.
   - If a transport order that is being processed is withdrawn (by a client/user), its state first changes to `WITHDRAWN` while the vehicle executes any orders that had already been sent to it. Then the transport order's state changes to `FAILED`. It is not processed any further.
   - If processing of the transport order fails for any reason, it is marked as `FAILED` and not processed any further.
   - If the vehicle successfully processes the transport order as a whole, it is marked as `FINISHED`.
7. Eventually — after a longer while or when too many transport orders in a final state have accumulated in the kernel's order pool — the kernel removes the transport order.

The following state machine visualizes this life cycle:

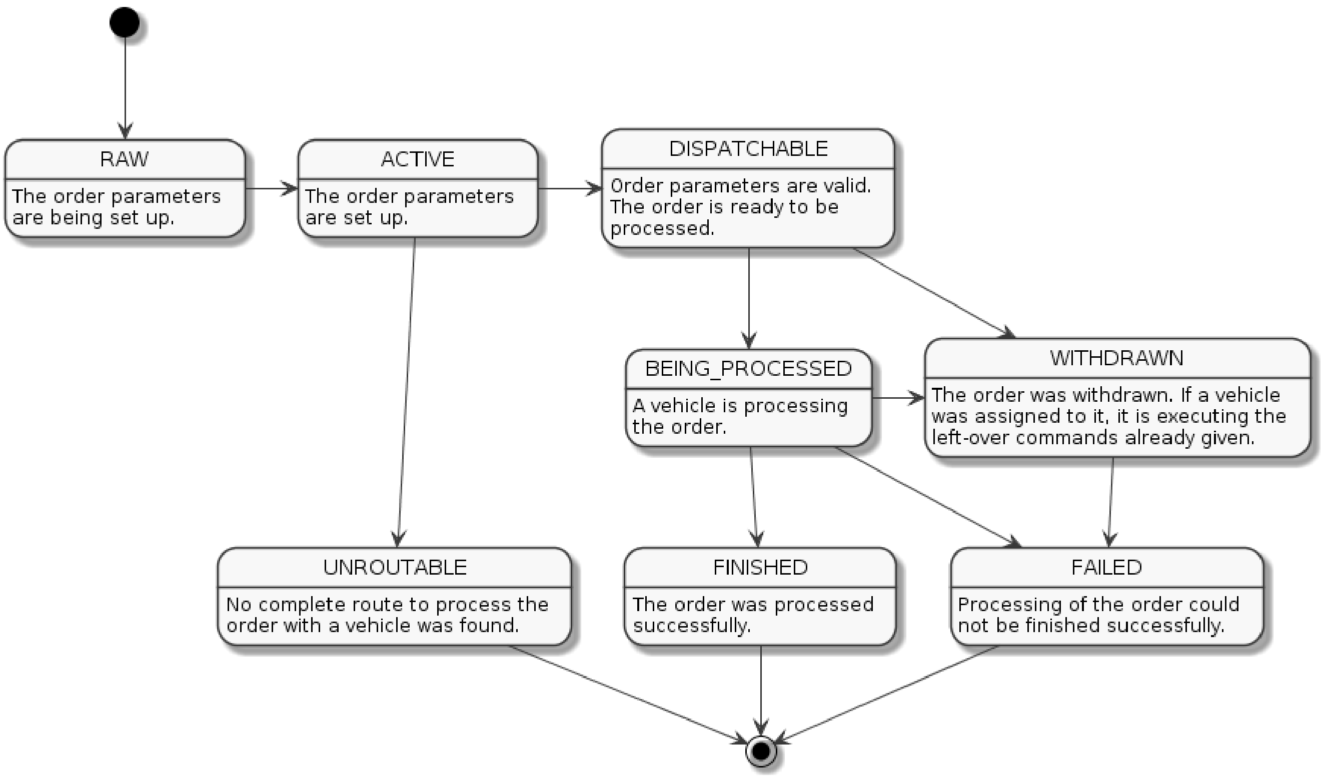


Figure 3. Transport order states

2.2.2. Structure and processing of transport orders

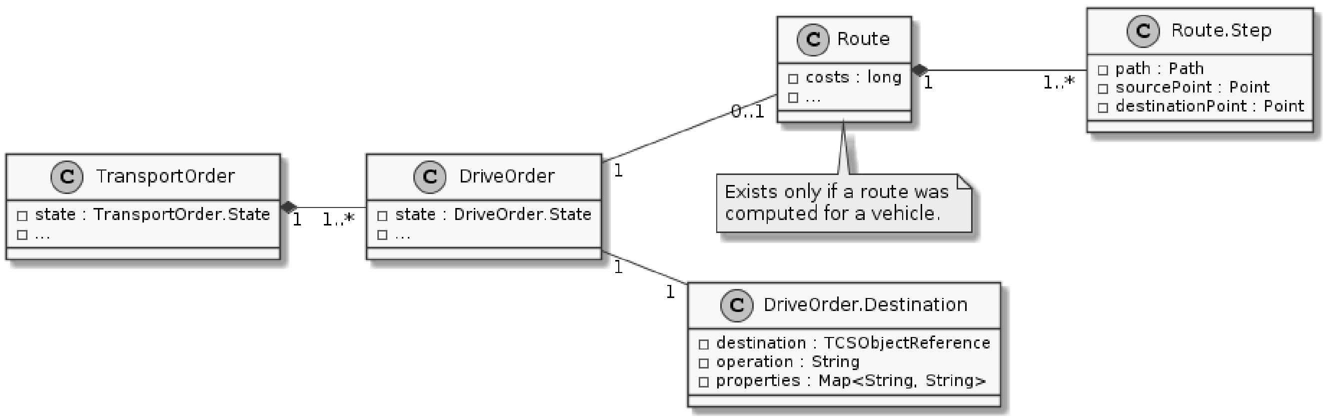
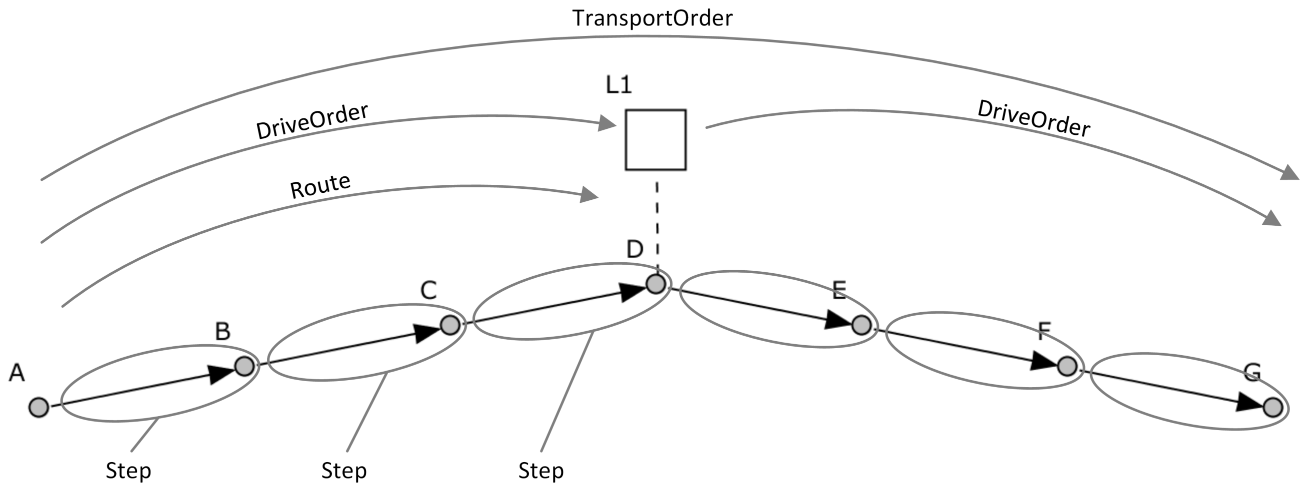


Figure 4. Transport order classes

A transport order is created by calling `TransportOrderService.createTransportOrder()`. As its parameter, it expects an instance of `TransportOrderCreationTO` containing the sequence of destinations to visit and the operations a vehicle is supposed to perform there. The kernel wraps each `Destination` in a newly-created `DriveOrder` instance. These `DriveOrder`s are themselves wrapped by the kernel in a single, newly-created `TransportOrder` instance in their given order.

Once a `TransportOrder` is being assigned to a vehicle by the `Dispatcher`, a `Route` is computed for each of its `DriveOrder`s. These `Route`s are then stored in the corresponding `DriveOrder`s.



As soon as a vehicle (driver) is able to process a `DriveOrder`, the single `Step`s of its `Route` are mapped to `MovementCommand`s. These `MovementCommand`s contain all information the vehicle driver needs to reach the final destination and to perform the desired operation there.

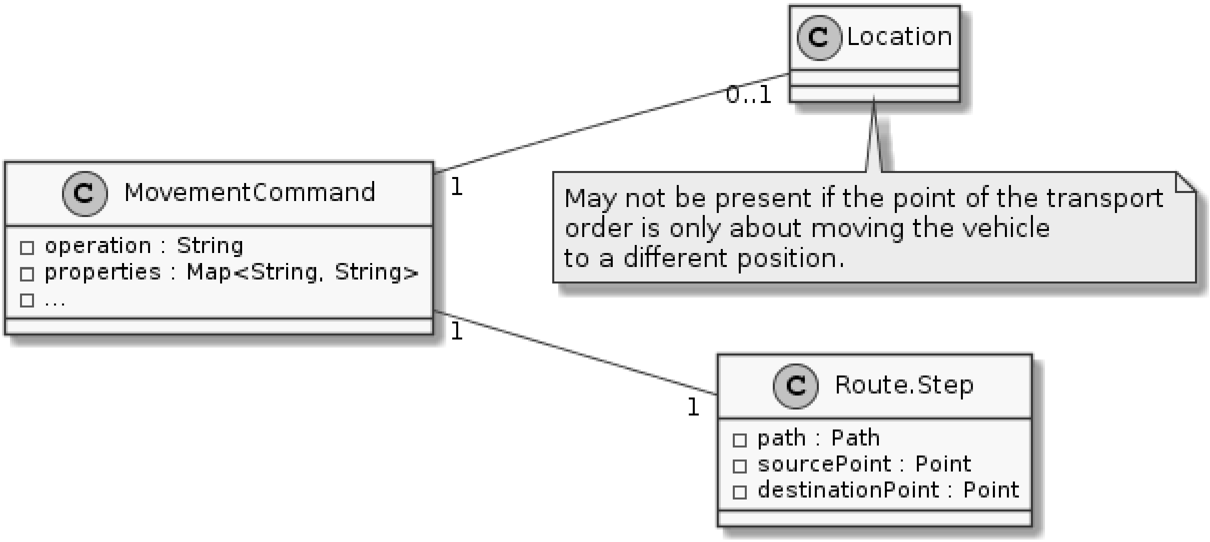


Figure 5. MovementCommand-related classes

The `MovementCommand`s for the partial routes to be travelled are sent to the vehicle driver bit by bit. The kernel only sends as many `MovementCommands`s in advance as is required for the vehicle driver to function properly. It does this to maintain fine-grained control over the paths/resources used by all vehicles. A vehicle driver may set the maximum number of `MovementCommand`s it gets in advance by adjusting its command queue capacity.

As soon as a `DriveOrder` is finished, the `Route` of the next `DriveOrder` is mapped to `MovementCommand`s. Once the last `DriveOrder` of a `TransportOrder` is finished, the whole `TransportOrder` is finished, as well.

2.2.3. How to create a new transport order

```java
    // The transport order service instance we're working with
    TransportOrderService transportOrderService = getATransportOrderServiceReference(
);

    // The dispatcher service instance we're working with
    DispatcherService dispatcherService = getADispatcherServiceReference();

    // A list of destinations the transport order the vehicle is supposed
    // to travel to:
    List<DestinationCreationTO> destinations = new LinkedList<>();
    // Create a new destination description and add it to the list.
    // Every destination is described by the name of the destination
    // location in the plant model and an operation the vehicle is supposed
    // to perform there:
    destinations.add(new DestinationCreationTO("Some location name",
                                               "Some operation"));
    // Add as many destinations to the list like this as necessary. Then
    // create a transport order description with a name for the new transport
    // order and the list of destinations.
    // Note that the given name needs to be unique.
    TransportOrderCreationTO orderTO
        = new TransportOrderCreationTO("MyTransportOrder",
                                       destinations);
    // Optionally, express that the actual/full name of the order should be
    // generated by the kernel.
    orderTO = orderTO.withIncompleteName(true);
    // Optionally, assign a specific vehicle to the transport order:
    orderTO = orderTO.withIntendedVehicleName("Some vehicle name");
    // Optionally, set a deadline for the transport order:
    orderTO = orderTO.withDeadline(Instant.now().plus(1, ChronoUnit.HOURS));

    // Create a new transport order for the given description:
    TransportOrder newOrder = transportOrderService.createTransportOrder(orderTO);

    // Trigger the dispatch process for the created transport order.
    dispatcherService.dispatch();
```

2.2.4. How to create a transport order that sends a vehicle to a point instead of a location

```java
    // The transport order service instance we're working with
    TransportOrderService transportOrderService = getATransportOrderServiceReference(
);

    // The dispatcher service instance we're working with
    DispatcherService dispatcherService = getADispatcherServiceReference();

    // Create a list containing a single destination to a point.
    // Use Destination.OP_MOVE as the operation to be executed:
    List<DestinationCreationTO> destinations = new LinkedList<>();
    destinations.add(new DestinationCreationTO("Some point name",
                                               Destination.OP_MOVE));
    // Create a transport order description with the destination and a
    // unique name and assign it to a specific vehicle:
    TransportOrderCreationTO orderTO
        = new TransportOrderCreationTO("MyTransportOrder",
                                       destinations)
            .withIntendedVehicleName("Some vehicle name")
            .withIncompleteName(true);

    // Create a transport order using the description:
    TransportOrder dummyOrder = transportOrderService.createTransportOrder(orderTO);

    // Trigger the dispatch process for the created transport order.
    dispatcherService.dispatch();
```

2.2.5. Using order sequences

An order sequence can be used to force a single vehicle to process multiple transport orders in a given order. Some rules for using order sequences are described in the API documentation for `OrderSequence`, but here is what you would do in general:

```java
    // The transport order service instance we're working with
    TransportOrderService transportOrderService = getATransportOrderServiceReference(
);

    // The dispatcher service instance we're working with
    DispatcherService dispatcherService = getADispatcherServiceReference();

    // Create an order sequence description with a unique name:
    OrderSequenceCreationTO sequenceTO
        = new OrderSequenceCreationTO("MyOrderSequence");
    // Optionally, express that the actual/full name of the sequence should be
    // generated by the kernel.
    sequenceTO = sequenceTO.withIncompleteName(true);
    // Optionally, set the sequence's failure-fatal flag:
    sequenceTO = sequenceTO.withFailureFatal(true);

    // Create the order sequence:
    OrderSequence orderSequence = transportOrderService.createOrderSequence(
sequenceTO);

    // Set up the transport order as usual,
    // but add the wrapping sequence's name:
    List<DestinationCreationTO> destinations = new ArrayList<>();
    destinations.add(new DestinationCreationTO("Some location name",
                                               "Some operation"));
    TransportOrderCreationTO orderTO
        = new TransportOrderCreationTO("MyOrder-" + UUID.randomUUID(),
                                       destinations)
            .withWrappingSequence(orderSequence.getName());

    // Create the transport order:
    TransportOrder order = transportOrderService.createTransportOrder(orderTO);

    // Create and add more orders as necessary.
    // Eventually, set the order sequence's complete flag to indicate that more
    // transport orders will not be added to it.
    transportOrderService.markOrderSequenceComplete(orderSequence.getReference());

    // Trigger the dispatch process for the created order sequence.
    dispatcherService.dispatch();
```

As long as the sequence has not been marked as complete and finished completely, the vehicle selected for its first order will be tied to this sequence. It will not process any orders not belonging to the same sequence until the whole sequence has been finished.

Once the *complete* flag of the sequence has been set and all transport orders belonging to it have been processed, its *finished* flag will be set by the kernel.

2.2.6. How to withdraw a transport order that is currently being processed

```
// The dispatcher service instance we're working with
DispatcherService dispatcherService = getDispatcherServiceFromSomewhere();

// Get the transport order to be withdrawn.
TransportOrder curOrder = getTransportOrderToWithdraw();
// Withdraw the order.
// The second argument indicates if the vehicle should finish the movements
// it is already assigned to (false) or abort immediately (true).
dispatcherService.withdrawByTransportOrder(curOrder.getReference(), true);
```

2.2.7. How to withdraw a transport order via a reference on the vehicle processing it

```
// The object service instance we're working with
TCSObjectService objectService = getTCSObjectServiceFromSomewhere();

// Get the vehicle from which the transport order shall be withdrawn
Vehicle curVehicle = objectService.fetchObject(Vehicle.class,
                                              getSampleVehicle());

// The dispatcher service instance we're working with
DispatcherService dispatcherService = getDispatcherServiceFromSomewhere();

// Withdraw the order.
// The second argument indicates if the vehicle should finish the movements
// it is already assigned to (false) or abort immediately (true).
dispatcherService.withdrawByVehicle(curVehicle.getReference(), true);
```

2.3. Using the event bus

Each of the main openTCS applications — Kernel, Kernel Control Center and Plant Overview — provides an event bus that can be used to receive or emit event objects application-wide. To acquire the respective application's event bus instance, request it to be provided via dependency injection. Any of the following three variants of constructor parameters are equivalent:

```
public MyClass(@ApplicationEventBus EventHandler eventHandler) {
  ...
}
```

```
public MyClass(@ApplicationEventBus EventSource eventSource) {
  ...
}
```

```
public MyClass(@ApplicationEventBus EventBus eventBus) {
  ...
}
```

Having acquired the `EventHandler`, `EventSource` or `EventBus` instance that way, you can use it to emit event objects to it and/or subscribe to receive event objects.

Note that, within the Kernel application, event objects should be emitted via the kernel executor to avoid concurrency issues — see Executing code in kernel context.

Chapter 3. TCP/IP-based interfaces to other systems

> This interface is **deprecated and scheduled for removal with openTCS 5.0** in favour of the kernel's web API. The web API specification is included in the openTCS distribution's documentation.

In addition to the Java-specific kernel interface, openTCS offers the following interfaces for communication with other systems:

- A bidirectional interface via a TCP/IP connection for the creation of transport orders
- An unidirectional interface via a TCP/IP connection for receiving status messages, e.g. about transport orders being processed

The TCP/IP interfaces are described in the following sections.

3.1. Creating orders via TCP/IP

For creating transport orders, the openTCS kernel accepts connections to a TCP port (default: port 55555). The communication between openTCS and the host works as follows:

1. The host establishes a new TCP/IP connection to openTCS.
2. The host sends a single XML telegram (described in detail in XML telegrams for creating orders and XML telegrams referencing order batches) which either describes the transport orders to be created or identifies batch files that are available with the kernel and that contain the transport order descriptions.
3. The host closes its output stream of the TCP/IP connection or sends two consecutive line breaks (i.e. "`\r\n\r\n`"), letting the kernel know that no further data will follow.
4. openTCS interprets the telegram sent by the host, creates the corresponding transport orders and activates them.
5. openTCS sends an XML telegram (described in detail in Receipts for created orders) to confirm processing of the telegram.
6. openTCS closes the TCP/IP connection.

The following points should be respected:

- Multiple sets of transport orders are not intended to be transferred via the same TCP connection. After processing a set and sending the response, openTCS closes the connection. To transfer further sets new TCP/IP connections need to be established by the peer system.
- openTCS only waits a limited amount of time (default: ten seconds) for incoming data. If there is no incoming data from the peer system over a longer period of time, the connection will be closed by openTCS without any transport orders being created.
- The maximum length of a single XML telegram is limited to 100 kilobytes by default. If more data is transferred, the connection will be closed without any transport orders being created.

3.1.1. XML telegrams for creating orders

Every XML telegram sent to openTCS via the interface described above can describe multiple transport orders to be created. Every order element must contain the following data:

- A string identifying the order element. This string is required for unambiguous matching of receipts (see Receipts for created orders) and orders.
- A sequence of destinations and destination operations defining the actual order.

Furthermore, each order element may contain the following data:

- A deadline/point of time at which the order should be finished.
- The name of a vehicle in the system that the order should be assigned to. If this information is omitted, any vehicle in the system may process the order.
- A set of names of existing transport orders that have to be finished before the new order may be assigned to a vehicle.

The following example shows how an XML telegram for the creation of two transport orders could look like.

Example 1: XML telegram for the creation of two transport orders

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsOrderSet>
    <order xsi:type="transport" deadline="2020-12-18T12:01:13.959+01:00"
intendedVehicle="Vehicle-01" id="TransportOrder-01" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <destination locationName="Storage 01" operation="Load cargo"/>
        <destination locationName="Storage 02" operation="Unload cargo"/>
        <property key="waitBefore" value="Unload"/>
    </order>
    <order xsi:type="transport" id="TransportOrder-02" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <destination locationName="Working station 01" operation="Drill">
            <property key="drillSize" value="3"/>
        </destination>
        <destination locationName="Working station 02" operation="Drill">
            <property key="drillSize" value="8"/>
        </destination>
        <destination locationName="Working station 03" operation="Cut"/>
    </order>
</tcsOrderSet>
```

3.1.2. XML telegrams referencing order batches

Alternatively, an XML telegram may also reference order batches which are kept in files on the openTCS system. The (parameters of the) transport orders to be created will then be read from the referenced batch files. A batch file may contain/create an arbitrary number of transport orders and needs to be placed in the kernel application's subdirectory `scripts`. In the openTCS distribution, this directory already contains a couple of templates for batch files (`template.tcs` and `test.tcs`).

The following example shows how an XML telegram referencing a batch file could look like.

Example 2: XML telegram referencing a batch file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsOrderSet>
    <order xsi:type="transportScript" fileName="test.tcs" id="test.tcs" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"/>
</tcsOrderSet>
```

3.1.3. Receipts for created orders

In response to an XML telegram for the creation of transport orders, an XML telegram will be sent back to the peer, reporting the operation's outcome. In the response telegram, every order element of the original telegram will be referenced by a response element with the same ID. Furthermore, every response element contains:

- A flag reflecting the success of creating the respective order
- The name that openTCS internally assigned to the created order. (This name is relevant for interpreting the messages on the status channel - see Status messages via TCP/IP.)

The following example shows how a response to the telegram in Example 1 could look like.

Example 3: XML telegram with receipts for created orders

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsResponseSet>
    <response xsi:type="transportResponse" executionSuccessful="true" orderName=
"TOrder-0001" id="TransportOrder-01" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
    <response xsi:type="transportResponse" executionSuccessful="true" orderName=
"TOrder-0002" id="TransportOrder-02" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
</tcsResponseSet>
```

3.1.4. Receipts for order batches

For referenced order batches, receipts will be sent back to the peer, too. The response contains an element for every batch file referenced by the peer. If the batch file was successfully read and processed, a response for every single order definition it contains will be included.

The following example shows a possible response to the batch file reference in Example 2. In this case, the batch file contains two transport order definitions which have been processed successfully.

Example 4: XML telegram with receipts for orders in batch file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsResponseSet>
    <response xsi:type="scriptResponse" parsingSuccessful="true" id="test.tcs"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <transport executionSuccessful="true" orderName="TOrder-0003" id="test.tcs"/>
        <transport executionSuccessful="true" orderName="TOrder-0004" id="test.tcs"/>
    </response>
</tcsResponseSet>
```

3.2. Status messages via TCP/IP

To receive status messages for transport orders in the system, connections to another TCP port (default: port 44444) may be established. Whenever the state of a transport order changes, an XML telegram will be sent to each connnected client, describing the new state of the order. Each of these telegrams is followed by a string that does not appear in the telegrams themselves (by default, a single pipe symbol: "|"), marking the end of the respective telegram. Status messages will be sent until the peer closes the TCP connection.

The following points should be respected:

- From the peer's point of view, connections to this status channel are purely passive, i.e. openTCS does not expect any messages from the peer and will not process any data received via this connection.
- A peer needs to filter the received telegrams for relevant data itself. The openTCS kernel does not provide any filtering of status messages for clients.
- Due to concurrent processes within openTCS, it is possible that the creation and activation of a transport order and its assignment to a vehicle is reported via the status channel before the peer that created the order receives the corresponding receipt.

The following example shows a status message as it would be sent via the status channel after the first of the two transport orders defined in Example 1 has been created and activated.

Example 5: Status message for the generated order

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsStatusMessageSet timeStamp="2020-12-18T12:01:14.225+01:00">
    <statusMessage xsi:type="orderStatusMessage" orderName="TOrder-0001" orderState=
"ACTIVE" processingVehicleName="Vehicle-0001" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <destination locationName="Storage 01" operation="Load cargo" state="PRISTINE
"/>
        <destination locationName="Storage 02" operation="Unload cargo" state=
"PRISTINE"/>
        <property key="waitBefore" value="Unload"/>
    </statusMessage>
</tcsStatusMessageSet>
```

The following example shows a status message as it would be sent via the status channel for a vehicle state update.

Example 6: Status message for vehicle update

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tcsStatusMessageSet timeStamp="2020-12-18T12:01:15.215+01:00">
    <statusMessage xsi:type="vehicleStatusMessage" position="Point-000"
processingState="PROCESSING_ORDER" state="EXECUTING" transportOrderName=
"TransportOrder-001" vehicleName="Vehicle-000" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
        <precisePosition x="100" y="110" z="120"/>
    </statusMessage>
</tcsStatusMessageSet>
```

3.3. XML Schema definitions for telegrams and scripts

XML schemas describing the expected structure of XML order telegrams and order batch files as well as the structure of receipt telegrams as sent by openTCS are part of the openTCS distribution and can be found in the directory containing the developer documentation.

Chapter 4. Generating an integration project

openTCS integration projects for customer- or plant-specific distributions often have a very similar structure. The openTCS distribution provides a way to easily generate such integration projects. This way, a developer can get started with customizing and extending openTCS components quickly.

To generate a template/skeleton for a new integration project, do the following:

1. Download and unzip the integration project example from the openTCS homepage.
2. Execute the following command from the example project's root directory: `gradlew cloneProject`

The integration project will be generated in the `build/` directory. (Make sure you copy it somewhere else before running the example project's `clean` task the next time.)

The project and the included classes will have generic names. You can adjust their names by setting a couple of properties when running the above command. The following properties are looked at:

- *integrationName*: Used for the names of the project itself and the subprojects within it.
- *classPrefix*: Used for some classes within the subprojects.

For instance, your command line could look like this:

```
gradlew -PintegrationName=MyGreatProject -PclassPrefix=Great cloneProject
```

This would include *MyGreatProject* in the integration project name, and *Great* in some class names.

> Inserting your own source code into a copy of the baseline openTCS project instead of creating a proper integration project as described above is not recommended. This is because, when integrating openTCS by copying its source code, you lose the ability to easily upgrade your code to more recent openTCS versions (for bugfixes or new features).

Chapter 5. Customizing and extending the kernel application

5.1. Guice modules

The openTCS kernel application uses Guice to configure its components. To modify the wiring of components within the application and to add your own components, you can register custom Guice modules. Modules are found and registered automatically via `java.util.ServiceLoader`.

Basically, the following steps are required for customizing the application:

1. Build a JAR file for your custom injection module with the following content:
   a. A subclass of `org.opentcs.customizations.kernel.KernelInjectionModule`, which can be found in the base library, must be contained. Configure your custom components or adjust the application's default wiring in this module. `KernelInjectionModule` provides a few supporting methods you can use.
   b. A plain text file named `META-INF/services/org.opentcs.customizations.kernel.KernelInjectionModule` must also be contained. This file should contain a single line of text with the fully qualified class name of your module.
2. Ensure that the JAR file(s) containing your Guice modules and the implementation of your custom component(s) are part of the class path when you start the kernel application.

For more information on how the automatic registration works, see the documentation of `java.util.ServiceLoader` in the Java class library. For more information on how Guice works, see the Guice documentation.

5.2. Replacing default kernel components

The kernel application comes with default implementations for the dispatching, routing and scheduling components. These default implementations allow the kernel to fulfil all of its responsibilities, but specific use cases might make it necessary to replace them with custom ones. In such cases, they can be replaced with a custom Guice configuration.

For each of these components, `KernelInjectionModule` provides a convenience method for (re)binding the implementation. To replace e.g. the default `Dispatcher` implementation, simply register a Guice module in which you call `bindDispatcher()`. The module's implementation could look like this:

```
@Override
protected void configure() {
  configureSomeDispatcherDependencies();
  bindDispatcher(CustomDispatcher.class);
}
```

> Note that all component implementations are bound as singletons. This is important for the following reason: Components may be injected and used at multiple places within the kernel application; at the same time, every component may also have to keep an internal state to do its work. If they were not bound as singletons, a new instance would be created for every injection, each of them with their own, separate internal state. Build custom components with that in mind, and implement their `initialize()` and `terminate()` methods appropriately!

5.3. Developing vehicle drivers

openTCS supports integration of custom vehicle drivers that implement vehicle-specific communication protocols and thus mediate between the kernel and the vehicle. Due to its function, a vehicle driver is also called a communication adapter. The following sections describe which requirements must be met by a driver and which steps are necessary to create and use it.

5.3.1. Classes and interfaces for the kernel

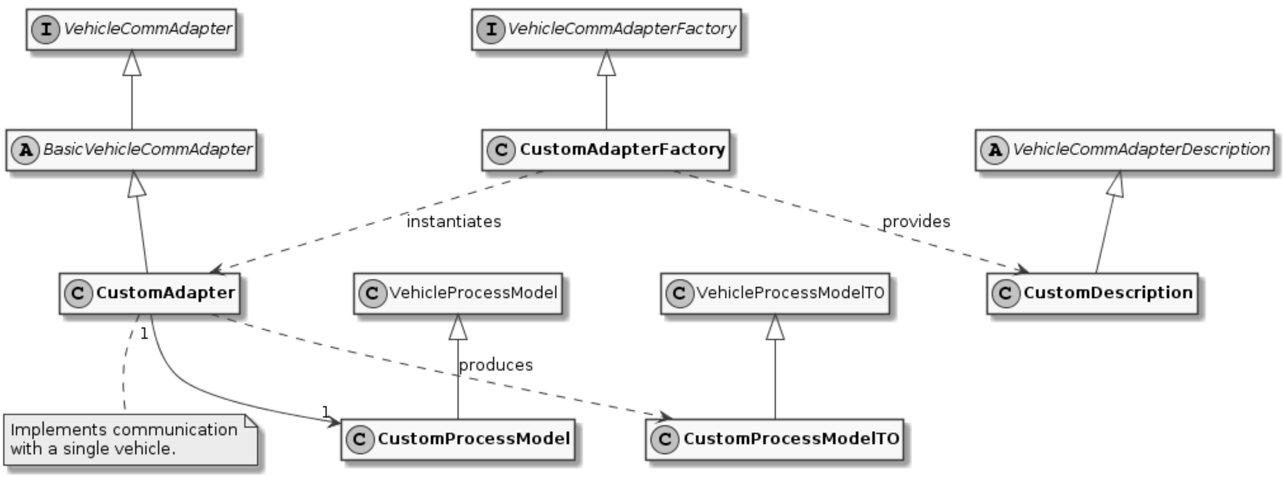


Figure 6. Classes of a comm adapter implementation (kernel side)

When developing a vehicle driver, the most important classes and interfaces in the base library are the following:

- `VehicleCommAdapter` declares methods that every comm adapter must implement. These methods are called by components within the kernel, for instance to tell a vehicle that it is supposed to move to the next position in the driving course. Classes implementing this interface are expected to perform the actual communication with a vehicle, e.g. via TCP, UDP or some field bus.
- `BasicVehicleCommAdapter` is the recommended base class for implementing a `VehicleCommAdapter`. It primarily provides some basic command queueing.
- `VehicleCommAdapterFactory` describes a factory for `VehicleCommAdapter` instances. The kernel instantiates and uses one such factory per vehicle driver to create instances of the respective `VehicleCommAdapter` implementation on demand.
- A single `VehicleProcessModel` instance should be provided by every `VehicleCommAdapter` instance in which it keeps the relevant state of both the vehicle and the comm adapter. This model

instance is supposed to be updated to notify the kernel about relevant changes. The comm adapter implementation should e.g. update the vehicle's current position in the model when it receives that information to allow the kernel and GUI frontends to use it. Likewise, other components may set values that influence the comm adapter's behaviour in the model, e.g. a time interval for periodic messages the comm adapter sends to the vehicle. `VehicleProcessModel` may be used as it is, as it contains members for all the information the openTCS kernel itself needs. However, developers may use driver-specific subclasses of `VehicleProcessModel` to have the comm adapter and other components exchange more than the default set of attributes.

5.3.2. Classes and interfaces for the control center

> The Kernel Control Center GUI provided by the kernel application itself is **deprecated and scheduled for removal with openTCS 5.0**. A separate Kernel Control Center application is available and should be used instead. For the remote version of the Kernel Control Center the following interfaces are the most important.

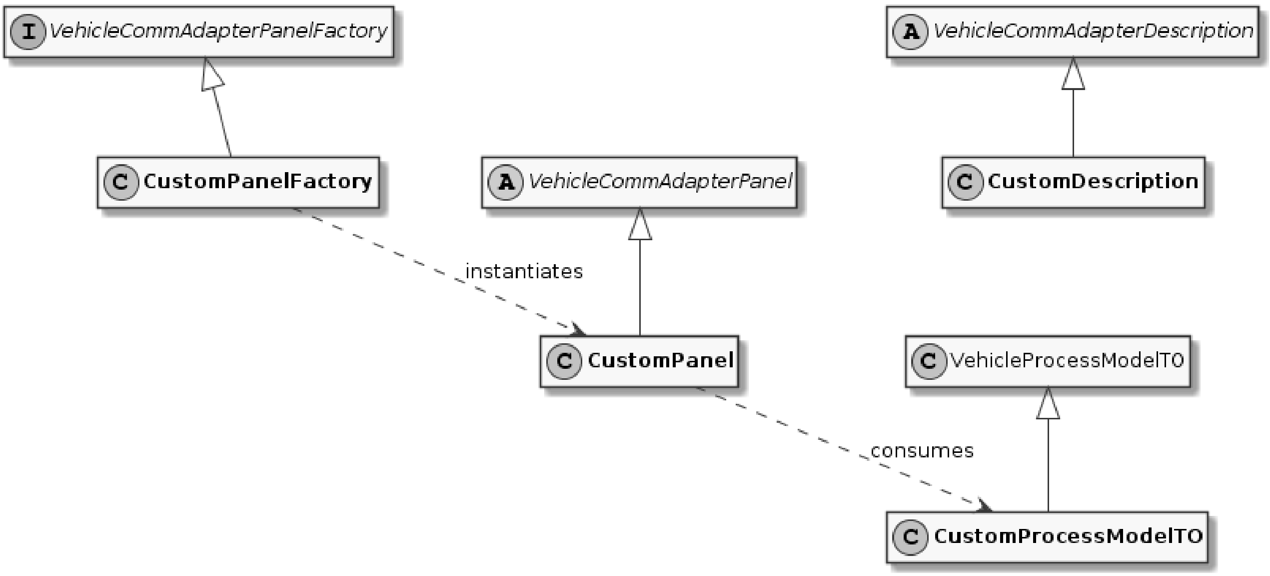


Figure 7. Classes of a comm adapter implementation (control center side)

- `VehicleCommAdapterPanel` instances may be created by a `VehicleCommAdapterPanelFactory` e.g. to display information about the associated vehicle or send low-level messages to it.
- `VehicleProcessModelTO` instances should be provided by every `VehicleCommAdapter` instance according to the current state of its `VehicleProcessModel`. Instances of this model are supposed to be used in a comm adapter's `VehicleCommAdapterPanel` instances for updating their contents only. Note that `VehicleProcessModelTO` is basically a serializable representation of a comm adapter's `VehicleProcessModel`. Developers should keep that in mind when creating driver-specific subclasses of `VehicleProcessModelTO`.
- Instances of `VehicleCommAdapterDescription` provide a string describing/identifying the comm adapter implementation. This string is shown e.g. when the user may select one of a set of driver implementations and should thus be unique. It is also used for attaching a comm adapter implementation via `VehicleService.attachCommAdapter()`.

- AdapterCommand instances can be sent from a panel to a VehicleCommAdapter instance via VehicleService.sendCommAdapterCommand(). They are supposed to be executed by the comm adapter and can be used to execute arbitrary methods, e.g. methods of the VehicleCommAdapter itself, or update contents of the comm adapter's VehicleProcessModel. Note that AdapterCommand instances can only be sent to and processed by the kernel application if they are serializable and present in the kernel application's classpath.

5.3.3. Steps to create a new vehicle driver

1. Create an implementation of VehicleCommAdapter:
   a. Subclass BasicVehicleCommAdapter unless you have a reason not to. You don't have to, but if you don't, you also need to implement command queue management yourself.
   b. Implement the abstract methods of BasicVehicleCommAdapter in the derived class to realize communication with the vehicle and to provide driver-specific visualization panels, if any.
   c. In situations in which the state of the vehicle changes in a way that is relevant for the kernel or the comm adapter's custom panels, the comm adapter should call the respective methods on the model. Most importantly, call setVehiclePosition() and commandExecuted() on the comm adapter's model when the controlled vehicle's reported state indicates that it has moved to a different position or that it has finished an order.
2. Create an implementation of VehicleCommAdapterFactory that provides instances of your VehicleCommAdapter for given Vehicle objects.
3. Optional: Create any number of implementations of VehicleCommAdapterPanel that the kernel control center application should display for the comm adapter. Create and return instances of these panels in the getPanelsFor() method of your VehicleCommAdapterPanelFactorys implementation.

See the API documentation for more details. For an example, refer to the implementation of the loopback comm adapter for virtual vehicles in the openTCS source distribution. (Note, however, that this implementation does not implement communication with any physical vehicle.)

5.3.4. Registering a vehicle driver with the kernel

1. Create a Guice module for your vehicle driver by creating a subclass of KernelInjectionModule. Implement the configure() method and register a binding to your VehicleCommAdapterFactory. For example, the loopback driver that is part of the openTCS distribution registers its own factory class with the following line in its configure() method:

```
    vehicleCommAdaptersBinder().addBinding().to(
LoopbackCommunicationAdapterFactory.class);
```

2. In the JAR file containing your driver, ensure that there exists a folder named META-INF/services/ with a file named org.opentcs.customizations.kernel.KernelInjectionModule. This file should consist of a single line of text holding simply the name of the Guice module class, e.g.:

```
org.opentcs.virtualvehicle.LoopbackCommAdapterModule
```

Background: openTCS uses `java.util.ServiceLoader` to automatically find Guice modules on startup, which depends on this file (with this name) being present. See the JDK's API documentation for more information about how this mechanism works.

3. Place the JAR file of your driver including all neccessary resources in the subdirectory `lib/openTCS-extensions/` of the openTCS kernel application's installation directory *before* the kernel is started. (The openTCS start scripts include all JAR files in that directory in the application's classpath.)

Drivers meeting these requirements are found automatically when you start the kernel.

5.4. Sending messages to communication adapters

Sometimes it is required to have some influence on the behaviour of a communication adapter (and thus the vehicle it is associated with) directly from a kernel client - to send a special telegram to the vehicle, for instance. For these cases, `VehicleService.sendCommAdapterMessage(TCSObjectReference<Vehicle>, Object)` provides a one-way communication channel for a client to send a message object to a communication adapter currently associated with a vehicle. A comm adapter implementing `processMessage()` may interpret message objects sent to it and react in an appropriate way. Note that the client sending the message may not know which communication adapter implementation is currently associated with the vehicle, so the adapter may or may not be able to understand the message.

5.5. Acquiring data from communication adapters

For getting information from a communication adapter to a kernel client, there are the following ways:

Communication adapters may publish events via their `VehicleProcessModel` instance to emit information encapsulated in an event for any listeners registered with the kernel. Apparently, listeners must already be registered before such an event is emitted by the communication adapter, or they will miss it. To register a client as a listener, use `EventSource.subscribe()`. You can get the EventSource instance used by the kernel through dependency injection by using the qualifier annotation `org.opentcs.customizations.ApplicationEventBus`.

Alternatively, communication adapters may use their `VehicleProcessModel` to set properties in the corresponding `Vehicle` object. Kernel clients may then retrieve the information from it:

```
    // The object service instance we're working with
    TCSObjectService objectService = getTCSObjectServiceFromSomewhere();

    // Get the vehicle from which information shall be retrieved
    Vehicle vehicle = objectService.fetchObject(Vehicle.class, getTheVehicleName());

    // Get the actual property you're looking for
    String property = vehicle.getProperty("someKey");
```

Communication adapters may also use their `VehicleProcessModel` to set properties in the corresponding `TransportOrder` object a vehicle is currently processing. This basically works the same way as with the `Vehicle` object:

```
    // The Kernel instance we're working with
    TCSObjectService objectService = getTCSObjectServiceFromSomewhere();

    // Get the tansport order from which information shall be retrieved
    TransportOrder transportOrder = objectService.fetchObject(TransportOrder.class,

getTheTransportOrderName());

    // Get the actual property you're looking for
    String property = transportOrder.getProperty("someKey");
```

Unlike information published via events, data stored as properties in `Vehicle` or `TransportOrder` objects can be retrieved at any time.

5.6. Executing code in kernel context

Within the kernel, concurrent modifications of the data model — e.g. contents of the plant model or transport order properties — need to be synchronized carefully. Similar to e.g. the Swing framework's Event Dispatcher Thread, a single thread is used for executing one-shot or periodics tasks performing data modifications. To help with this, an instance of `java.util.concurrent.ScheduledExecutorService` is provided. Custom code running within the kernel application, including vehicle drivers and implementations of additional funcionality, should also perform changes of the data model via this executor only to avoid concurrency issues.

To make use of the kernel's executor, use the `@KernelExecutor` qualifier annotation and inject a `ScheduledExecutorService`:

```
@Inject
public MyClass(@KernelExecutor ScheduledExecutorService kernelExecutor) {
  ...
}
```

You can also inject it as a `java.util.concurrent.ExecutorService`:

```java
@Inject
public MyClass(@KernelExecutor ExecutorService kernelExecutor) {
  ...
}
```

Injecting a `java.util.concurrent.Executor` is also possible:

```java
@Inject
public MyClass(@KernelExecutor Executor kernelExecutor) {
  ...
}
```

Then, you can use it e.g. to lock a path in the plant model in kernel context:

```java
kernelExecutor.submit(() -> routerService.updatePathLock(ref, true));
```

Due to the single-threaded nature of the kernel executor, tasks submitted to it are executed sequentially, one after another. This implies that submitting long-running tasks should be avoided, as they would block the execution of subsequent tasks.

When event objects, e.g. instances of `TCSObjectEvent`, are distributed within the kernel, this always happens in kernel context, i.e. from a task that is run by the kernel executor. Event handlers should behave accordingly and finish quickly/not block execution for too long. If processing an event requires time-consuming actions to be taken, these should be executed on a different thread.

> As its name indicates, the kernel executor is only available within the kernel application. It is not available for code running in other applications like the Plant Overview, and it is not required there (for avoiding concurrency issues in the kernel).

Chapter 6. Customizing and extending the control center application

6.1. Guice modules

The openTCS kernel control center application uses Guice to configure its components. To modify the wiring of components within the application and to add your own components, you can register custom Guice modules. Modules are found and registered automatically via `java.util.ServiceLoader`.

Basically, the following steps are required for customizing the application:

1. Build a JAR file for your custom injection module with the following content:
   a. A subclass of `org.opentcs.customizations.controlcenter.ControlCenterInjectionModule` must be contained. Configure your custom components or adjust the application's default wiring in this module. `ControlCenterInjectionModule` provides a few supporting methods you can use.
   b. A plain text file named `META-INF/services/org.opentcs.customizations.controlcenter.ControlCenterInjectionModule` must also be contained. This file should contain a single line of text with the fully qualified class name of your module.
2. Ensure that the JAR file(s) containing your Guice modules and the implementation of your custom component(s) are part of the class path when you start the control center application.

For more information on how the automatic registration works, see the documentation of `java.util.ServiceLoader` in the Java class library. For more information on how Guice works, see the Guice documentation.

6.2. Registering driver panels with the control center

1. Create a Guice module for your vehicle driver by creating a subclass of `ControlCenterInjectionModule`. Implement the `configure()` method and register a binding to your `VehicleCommAdapterPanelFactory`. The following example demonstrates how this module's `configure()` method looks like for the loopback driver that is part of the openTCS distribution:

```
  @Override
  protected void configure() {
    commAdapterPanelFactoryBinder().addBinding().to(
LoopbackCommAdapterPanelFactory.class);
  }
```

2. In the JAR file containing your driver, ensure that there exists a folder named `META-INF/services/` with a file named `org.opentcs.customizations.controlcenter.ControlCenterInjectionModule`. This file should

consist of a single line of text holding simply the name of the Guice module class, e.g.:

```
org.opentcs.controlcenter.LoopbackCommAdapterPanelsModule
```

> Background: openTCS uses `java.util.ServiceLoader` to automatically find Guice modules on startup, which depends on this file (with this name) being present. See the JDK's API documentation for more information about how this mechanism works.

3. Place the JAR file of your driver including all neccessary resources in the subdirectory `lib/openTCS-extensions/` of the control center application's installation directory *before* the application is started. (The openTCS start scripts include all JAR files in that directory in the application's classpath.)

Panels meeting these requirements are found automatically when you start the kernel control center application.

Chapter 7. Customizing and extending the plant overview application

7.1. Guice modules

Analogous to the kernel application, the plant overview application uses Guice to configure its components. To modify the wiring of components within the application and to add your own components, you can register custom Guice modules. Modules are found and registered automatically via `java.util.ServiceLoader`.

Basically, the following steps are required for customizing the application:

1. Build a JAR file for your custom injection module with the following content:
   a. A subclass of `PlantOverviewInjectionModule`, which can be found in the base library, must be contained. Configure your custom components or adjust the application's default wiring in this module. `PlantOverviewInjectionModule` provides a few supporting methods you can use.
   b. A plain text file named `META-INF/services/org.opentcs.customizations.plantoverview.PlantOverviewInjectionModule` must also be contained. This file should contain a single line of text with the fully qualified class name of your module.
2. Ensure that the JAR file(s) containing your Guice modules and the implementation of your custom component(s) are part of the class path when you start the plant overview application.

For more information on how the automatic registration works, see the documentation of `java.util.ServiceLoader` in the Java class library. For more information on how Guice works, see the Guice documentation.

7.2. How to create a plugin panel for the plant overview client

The plant overview client offers to integrate custom panels providing project-specific functionality.

1. Implement a subclass of `PluggablePanel`.
2. Implement a `PluggablePanelFactory` that produces instances of your `PluggablePanel`.
3. Create a Guice module for your `PluggablePanelFactory` by subclassing `PlantOverviewInjectionModule`. Implement the `configure()` method and add a binding to your `PluggablePanelFactory` using `pluggablePanelFactoryBinder()`. For example, the load generator panel that is part of the openTCS distribution is registered with the following line in its module's `configure()` method:

```
pluggablePanelFactoryBinder().addBinding().to(ContinuousLoadPanelFactory.class
);
```

4. Build and package the `PluggablePanel`, `PluggablePanelFactory` and Guice module into a JAR file.
5. In the JAR file, register the Guice module class as a service of type `PlantOverviewInjectionModule`. To do that, ensure that the JAR file contains a folder named `META-INF/services/` with a file named `org.opentcs.customizations.plantoverview.PlantOverviewInjectionModule`. This file should consist of a single line of text holding simply the name of the guice module class, e.g.:

```
org.opentcs.guing.plugins.panels.loadgenerator.LoadGeneratorPanelModule
```

6. Place the JAR file in the Plant Overview application's class path (subdirectory `lib/openTCS-extensions/` of the application's installation directory) and start the application.

7.3. How to create a location/vehicle theme for openTCS

Locations and vehicles are visualized in the plant overview client using configurable themes. To customize the appearance of locations and vehicles, new theme implementations can be created and integrated into the plant overview client.

1. Create a new class which implements `LocationTheme` or `VehicleTheme`.
2. Place the JAR file of your theme, containing all required resources, in the subdirectory `lib/openTCS-extensions/` of the openTCS plant overview application's installation directory *before* the application is started. (The openTCS start scripts include all JAR files in that directory in the application's classpath.)
3. Set the `locationThemeClass` or `vehicleThemeClass` in the Plant Overview application's configuration file.

Vehicles or locations in plant models are then rendered using your custom theme.

Chapter 8. Supplementing configuration sources

As described in the openTCS User’s Guide, the openTCS Kernel, Kernel Control Center and Plant Overview applications read their configurations from properties files. This functionality is provided by the cfg4j library.

It is possible to register additional configuration sources, e.g. for reading configuration data from network resources or files in different formats. The mechanism provided by `java.util.ServiceLoader` is used for this. The following steps are required for registering a configuration source:

1. Build a JAR file with the following content:
   a. An implementation of `SupplementaryConfigurationSource`. This interface is part of the `opentcs-impl-configuration-cfg4j` artifact, which must be on your project’s classpath.
   b. A plain text file named `META-INF/services/org.opentcs.configuration.cfg4j.SupplementaryConfigurationSource`. This file should contain a single line of text with the fully qualified class name of your implementation.
2. Ensure that the JAR file is part of the classpath when you start the respective application.

It is possible to register multiple supplementary configuration sources this way.

The configuration entries provided by any registered supplementary configuration source may override configuration entries provided by the properties files that are read by default. Note that the order in which these additional configuration sources are processed is unspecified.

For more information on how the automatic registration works, see the documentation of `java.util.ServiceLoader` in the Java class library.

Chapter 9. Translating the user interfaces

Each openTCS application with a user interface is prepared for internationalization based on Java's `ResourceBundle` mechanism. As a result, the applications can be configured to display texts in different languages, provided there is a translation in the form of resource bundle property files. (How this configuration works is described in the User's Guide.) The openTCS distribution itself comes with language files for the default language (English) and German. Additional translations can be integrated primarily by adding JAR files containing property files to the class path.

The following sections explain how to create and integrate a new translation.

> Parts of the texts in the distribution may change between openTCS releases. While this might not happen often, it still means that, when you update to a new version of openTCS, you may want to check whether your translations are still correct. If there were textual changes in the openTCS distribution, you may need to update your language files.

9.1. Extracting default language files

To create a new translation pack for an application, you first need to know what texts to translate. The best way to do this is to look at the existing language files in the openTCS distribution. These are contained in the applications' JAR files (`opentcs-*.jar`), and are by convention kept in a common directory `/i18n/org/opentcs` inside these JAR files.

To start your translation work, extract all of the application's language files into a single directory first. Since JAR files are really only ZIP files, this can be done using any ZIP utility you like. As an example, to use `unzip` in a shell on a Linux system, issue the following command from the application's `lib/` directory:

```
unzip "opentcs-*.jar" "i18n/org/opentcs/*.properties"
```

Alternatively, to use 7-Zip in a shell on a Windows system, issue the following command from the application's `lib/` directory:

```
7z x -r "opentcs-*.jar" "i18n\org\opentcs\*.properties"
```

You will find the extracted language files in the `i18n/` directory, then. For the Plant Overview application, an excerpt of that directory's contents would look similar to this:

```
i18n/
  org/
    opentcs/
      plantoverview/
        mainMenu.properties
        mainMenu_de.properties
        toolbar.properties
        toolbar_de.properties
        ...
```

Files whose names end with `_de.properties` are German translations. You will not need these and can delete them.

9.2. Creating a translation

Copy the whole `i18n/` directory with the English language files to a new, separate directory, e.g. `translation/`. Working with a copy ensures that you still have the English version at hand to look up the original texts when translating.

Then rename all property files in the new directory so their names contain the appropriate language tag for your translation. If you are e.g. translating to Norwegian, rename `mainMenu.properties` to `mainMenu_no.properties` and the other files accordingly. It is important that the base name of the file remains the same and only the language tag is added to it.

The next step is doing the actual translation work — open each property file in a text editor and translate the properties' values in it.

After translating all the files, create a JAR file containing the `i18n/` directory with your language files. You can do this for instance by simply creating a ZIP file and changing its name to end with `.jar`.

The result could be a file named e.g. `language-pack-norwegian.jar`, whose contents should look similar to this:

```
i18n/
  org/
    opentcs/
      plantoverview/
        mainMenu_no.properties
        toolbar_no.properties
        ...
```

9.3. Integrating a translation

Finally, you merely need to add the JAR file you created to the translated application's class path. After configuring the application to the respective language and restarting it, you should see your translations in the user interface.

9.4. Updating a translation

As development of openTCS proceeds, parts of the applications' language files may change. This means that your translations may also need to be updated when you move from one version of openTCS to a more recent one.

To find out what changes were made and may need to be applied to your translations, you could do the following:

1. Extract the language files for the old version of the application, e.g. into a directory `translation_old/`.
2. Extract the language files for the new version of the application, e.g. into a directory `translation_new/`.
3. Create a diff between the two language file versions. For example, on a Linux system you could run `diff -urN translation_old/ translation_new/ > language_changes.diff` to write a diff to the file `language_changes.diff`.
4. Read the diff to see which new language files and/or entries were added, removed or changed.

Based on the information from the diff, you can apply appropriate changes to your own language files. Then you merely need to create new JAR files for your translations and add them to the applications' class paths.