

openTCS

Developer's Guide

The openTCS developers

Version 6.1.2

Table of Contents

1. Development with openTCS in general	1
1.1. System requirements	1
1.2. Available artifacts and API compatibility	1
1.3. Third-party dependencies	3
1.4. Modularity and extensibility	3
1.5. Logging	3
1.6. Working with the openTCS source code	4
1.7. openTCS kernel APIs	4
2. The kernel's Java API	5
2.1. Acquiring service objects	6
2.2. Working with transport orders	6
2.2.1. A transport order's life cycle	7
2.2.2. Structure and processing of transport orders	8
2.2.3. How to create a transport order	9
2.2.4. How to create a transport order that sends a vehicle to a point instead of a location ...	10
2.2.5. How to work with order sequences	11
2.2.6. How to withdraw a transport order	12
2.2.7. How to withdraw a transport order via a vehicle reference	13
2.3. Using the event bus	13
3. Generating an integration project	14
4. Customizing and extending the kernel application	15
4.1. Guice modules	15
4.2. Replacing default kernel components	15
4.3. Developing vehicle drivers	16
4.3.1. Classes and interfaces for the kernel	16
4.3.2. Classes and interfaces for the control center application	17
4.3.3. Steps to create a new vehicle driver	18
4.3.4. Registering a vehicle driver with the kernel	18
4.4. Sending messages to communication adapters	19
4.5. Acquiring data from communication adapters	19
4.6. Developing peripheral drivers	20
4.6.1. Classes and interfaces for the kernel	20
4.6.2. Classes and interfaces for the control center application	21
4.6.3. Steps to create a new peripheral driver	22
4.6.4. Registering a peripheral driver with the kernel	22
4.7. Executing code in kernel context	23
5. Customizing and extending the control center application	25
5.1. Guice modules	25

5.2. Registering driver panels with the control center	25
6. Customizing and extending the Model Editor and the Operations Desk applications	27
6.1. Guice modules	27
6.2. How to add import/export functionality for plant model data	27
6.3. How to create a plugin panel for the Operations Desk client	28
6.4. How to create a location/vehicle theme for openTCS	29
7. Application configuration	30
7.1. Supplementing configuration sources using gestalt	30
8. Translating the user interfaces	31
8.1. Extracting default language files	31
8.2. Creating a translation	32
8.3. Integrating a translation	32
8.4. Updating a translation	33

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) 21. To run the resulting binaries, you need a Java Runtime Environment (JRE) 21. 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 the [Maven Central](#) artifact repository, 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 {  
    mavenCentral()  
}  
  
dependencies {  
    compile group: 'org.opentcs', name: '${ARTIFACT}', version: '6.1.2'  
}
```

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 client 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-gestalt	No	An implementation of the base API's configuration interfaces based on gestalt.

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-plantoverview-base	No	The base data structures and components used by the Model Editor and the Operations Desk that don't require third-party libraries.
opentcs-plantoverview-common	No	A collection of classes and components commonly used by the Model Editor and the Operations Desk.
opentcs-plantoverview-panel-loadgenerator	No	The load generator panel implementation for the Operations Desk.
opentcs-plantoverview-panel-resourceallocation	No	The resource allocation panel implementation for the Operations Desk.
opentcs-plantoverview-panel-statistics	No	The statistics panel implementation for the Operations Desk.
opentcs-plantoverview-themes-default	No	The default themes implementation for the Operations Desk.
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-modeeditor	No	The Model Editor application.
opentcs-operationsdesk	No	The Operations Desk 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 explicitly use these as dependencies 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.
- Gestalt (<https://github.com/gestalt-config/gestalt>): 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 (<https://jgraph.org/>): A library for working with graphs and using algorithms on them.
- Spark (<https://sparkjava.com/>): A framework for creating web applications.
- Jackson (<https://github.com/FasterXML/jackson>): Provides JSON bindings for Java objects.

The Model Editor and Operations Desk applications have the following additional dependencies:

- JHotDraw (<https://github.com/wrandelshofer/jhotdraw>): A framework for drawing graph structures (like driving course models).
- Docking Frames (<https://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 (<https://site.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 Model Editor and the Operations Desk applications](#) 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 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.

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.

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 the following:

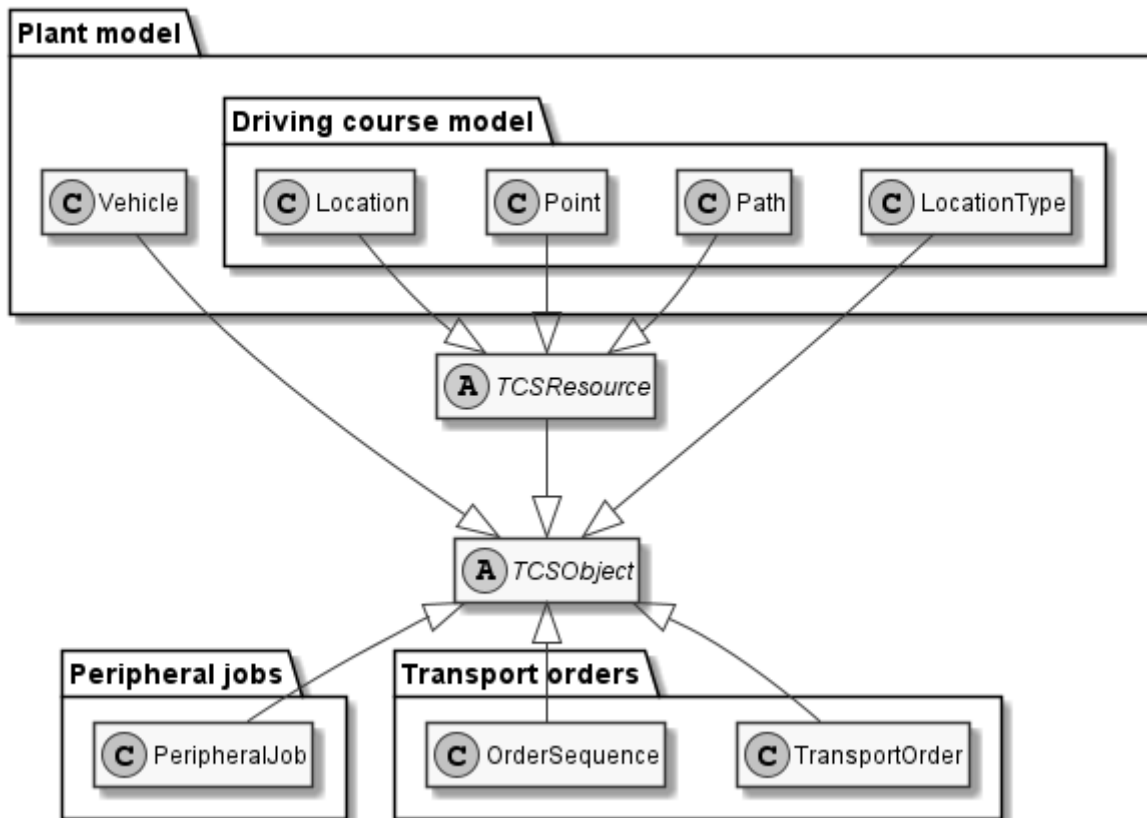


Figure 1. Basic data structures

The service interfaces that are most often interacted with to fetch and manipulate such data structures are these:

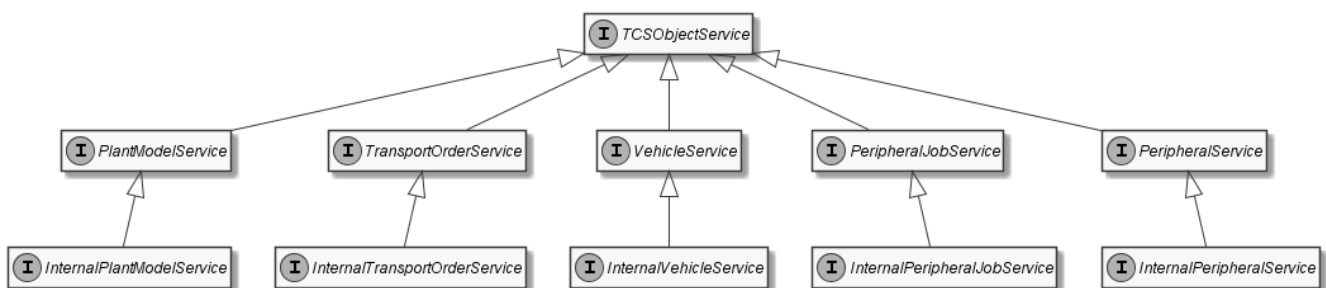


Figure 2. TCSObject-related service interfaces

A few more interfaces are available to interact with various parts of the kernel, as shown in the following diagram:



Figure 3. Additional service interfaces



*Peripheral** are classes/interfaces related to experimental integration of peripheral devices. These features are not documented in detail, yet, and developers using any of them are on their own, for now.

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:

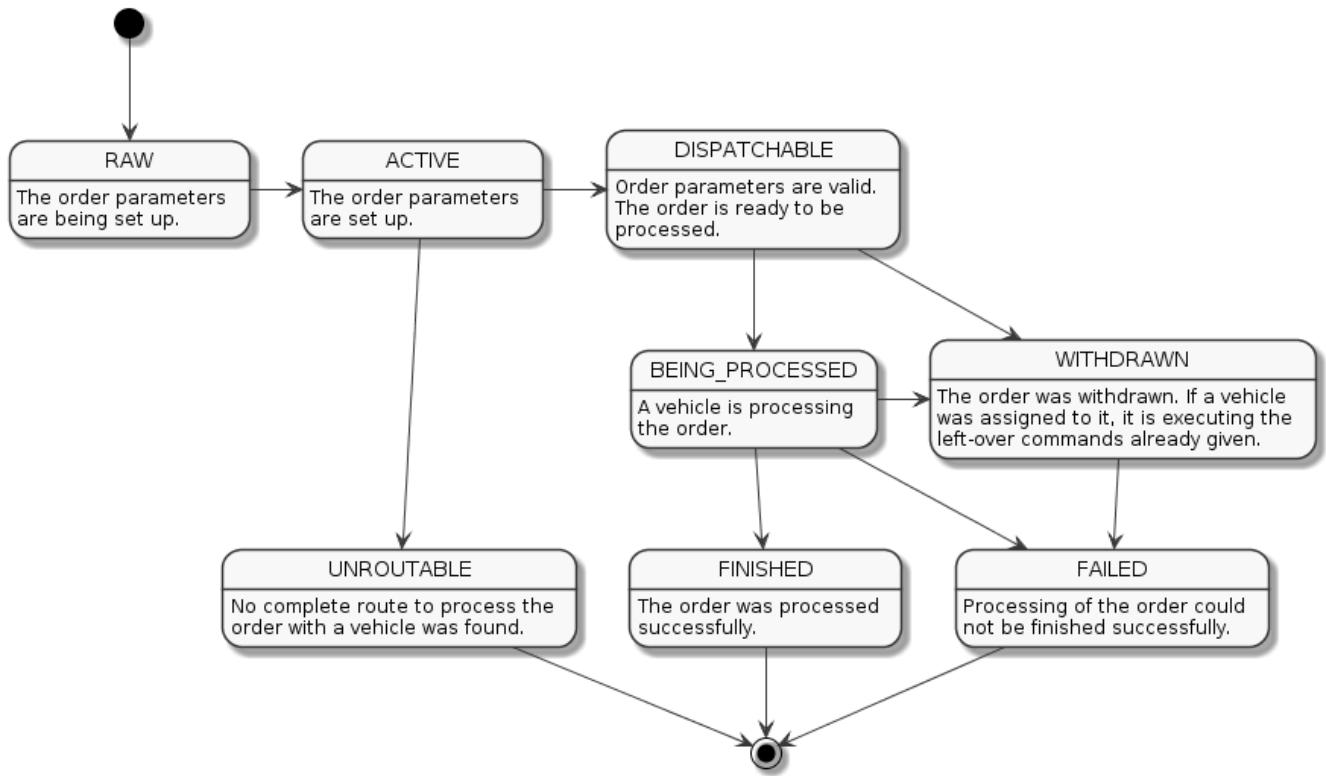


Figure 4. Transport order states

2.2.2. Structure and processing of transport orders

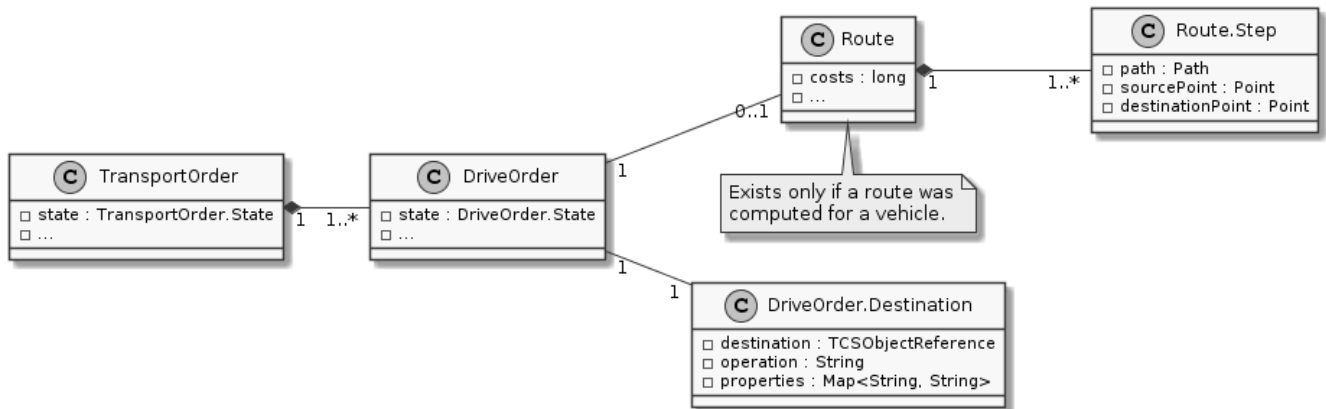
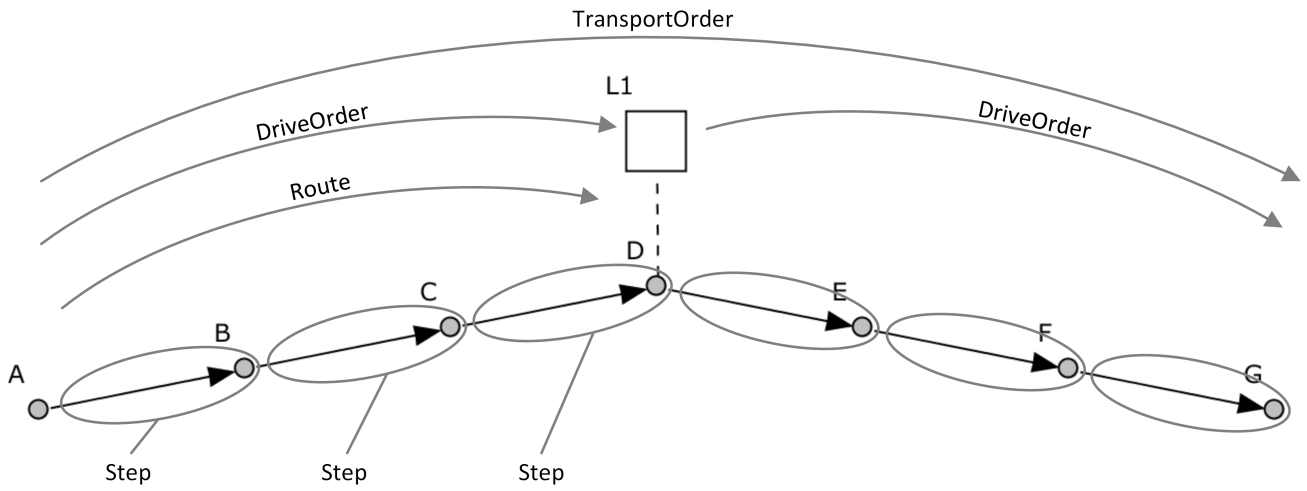


Figure 5. Transport order classes

A transport order is created by calling `TransportOrderService.createTransportOrder()`. As its parameter, it expects an instance of `TransportOrderCreationT0` 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 `DriveOrders` 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 `DriveOrders`. These `Routes` are then stored in the corresponding `DriveOrders`.



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

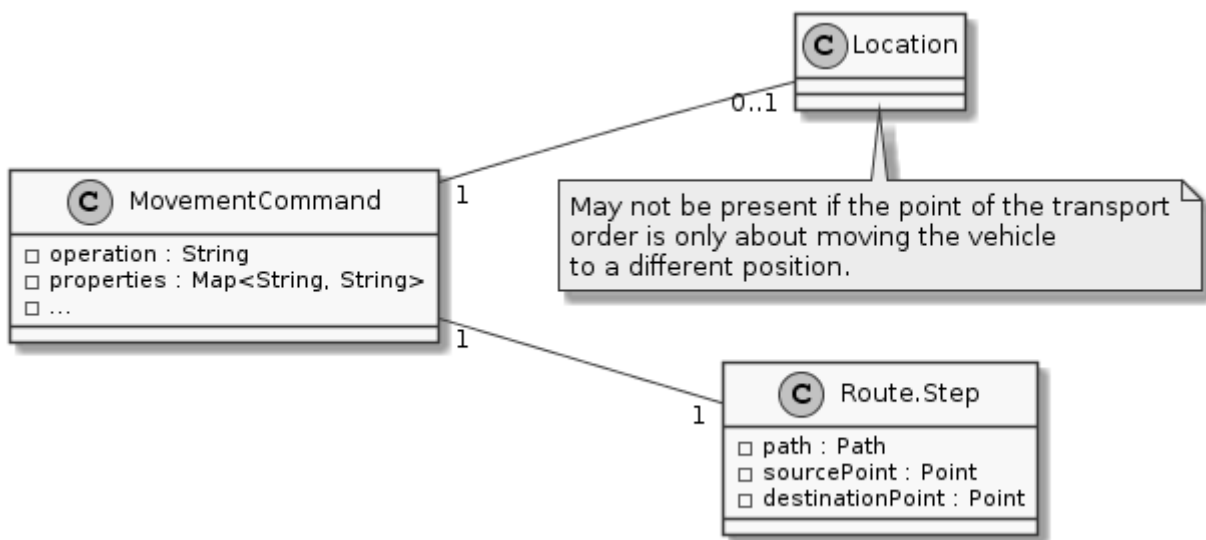


Figure 6. *MovementCommand-related classes*

The **MovementCommands** for the partial routes to be travelled are sent to the vehicle driver bit by bit. The kernel only sends as many **MovementCommandss** 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 **MovementCommands** 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 **MovementCommands**. Once the last **DriveOrder** of a **TransportOrder** is finished, the whole **TransportOrder** is finished, as well.

2.2.3. How to create a transport order

Create a list of destinations the vehicle is supposed to travel to. 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:

```
List<DestinationCreationT0> destinations
    = List.of(
        new DestinationCreationT0("Some location", "Some operation")
    );
```

Put as many destinations into the list as necessary. Then create a transport order description with a name for the new transport order and the list of destinations.

```
TransportOrderCreationT0 orderT0
    = new TransportOrderCreationT0("MyTransportOrder", destinations);
```

Optionally, express that the full name of the order should be generated by the kernel. (If you do not do this, you need to ensure that the name of the transport order given above is unique.)

```
orderT0 = orderT0.withIncompleteName(true);
```

Optionally, set more parameters for the transport order, e.g. set a deadline for the order or assign a specific vehicle to it:

```
orderT0 = orderT0
    .withIntendedVehicleName("Some vehicle")
    .withDeadline(Instant.now().plus(1, ChronoUnit.HOURS));
```

Get a **TransportOrderService** (see [Acquiring service objects](#)) and ask it to create a transport order using the given description:

```
TransportOrderService transportOrderService = getATransportOrderService();
transportOrderService.createTransportOrder(orderT0);
```

Optionally, get a **DispatcherService** and trigger the kernel's dispatcher explicitly to have it check for a vehicle that can process the transport order. (You only need to do this if you need the dispatcher to be triggered immediately after creating the transport order. If you do not do this, the dispatcher will still be triggered periodically.)

```
DispatcherService dispatcherService = getADispatcherService();
dispatcherService.dispatch();
```

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

Create a list containing a single destination to a point, using **Destination.OP_MOVE** as the operation to be executed:

```
List<DestinationCreationTO> destinations
    = List.of(
        new DestinationCreationTO("Some point", Destination.OP_MOVE)
    );
```

Create a transport order description with a name for the new transport order and the (single-element) list of destinations:

```
TransportOrderCreationTO orderTO
    = new TransportOrderCreationTO("MyTransportOrder", destinations)
        .withIntendedVehicleName("Some vehicle")
        .withIncompleteName(true);
```

Get a `TransportOrderService` (see [Acquiring service objects](#)) and ask it to create a transport order using the given description:

```
TransportOrderService transportOrderService = getATransportOrderService();
transportOrderService.createTransportOrder(orderTO);
```

Optionally, get a `DispatcherService` and trigger the kernel's dispatcher explicitly to have it check for a vehicle that can process the transport order. (You only need to do this if you need the dispatcher to be triggered immediately after creating the transport order. If you do not do this, the dispatcher will still be triggered periodically.)

```
DispatcherService dispatcherService = getADispatcherService();
dispatcherService.dispatch();
```

2.2.5. How to work with 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. First, create an order sequence description, providing a name:

```
OrderSequenceCreationTO sequenceTO
    = new OrderSequenceCreationTO("MyOrderSequence");
```

Optionally, express that the full name of the sequence should be generated by the kernel. (If you do not do this, you need to ensure that the name of the order sequence given above is unique.)

```
sequenceTO = sequenceTO.withIncompleteName(true);
```

Optionally, set the sequence's failure-fatal flag:

```
sequenceT0 = sequenceT0.withFailureFatal(true);
```

Get a `TransportOrderService` (see [Acquiring service objects](#)) and ask it to create an order sequence using the given description:

```
TransportOrderService transportOrderService = getATransportOrderService();
OrderSequence orderSequence
    = transportOrderService.createOrderSequence(sequenceT0);
```

Create a description for the transport order as usual, but set the wrapping sequence's name via `withWrappingSequence()` to associate the transport order with the order sequence. Then, create the transport order using the `TransportOrderService`.

```
TransportOrderCreationT0 orderT0
    = new TransportOrderCreationT0(
        "MyOrder",
        List.of(
            new DestinationCreationT0("Some location", "Some operation")
        )
    )
    .withIncompleteName(true)
    .withWrappingSequence(orderSequence.getName());

transportOrderService.createTransportOrder(orderT0);
```

Create and add more orders to the order sequence as necessary. Eventually, set the order sequence's *complete* flag to indicate that no further transport orders will be added to it:

```
transportOrderService.markOrderSequenceComplete(
    orderSequence.getReference()
);
```

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

To withdraw a transport order, get a `DispatcherService` (see [Acquiring service objects](#)) and ask it to withdraw the order, providing a reference to it:

```
DispatcherService dispatcherService = getADispatcherService();
dispatcherService.withdrawByTransportOrder(someOrder.getReference(), true);
```

The second argument indicates whether the vehicle should finish the movements it is already assigned to (**false**) or abort immediately (**true**).

2.2.7. How to withdraw a transport order via a vehicle reference

To withdraw the transport order that a specific vehicle is currently processing, get a **DispatcherService** (see [Acquiring service objects](#)) and ask it to withdraw the order, providing a reference to the vehicle:

```
DispatcherService dispatcherService = getADispatcherService();
dispatcherService.withdrawByVehicle(curVehicle.getReference(), true);
```

The second argument indicates whether the vehicle should finish the movements it is already assigned to (**false**) or abort immediately (**true**).

2.3. Using the event bus

Each of the main openTCS applications—Kernel, Kernel Control Center, Model Editor and Operations Desk—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. 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 4. Customizing and extending the kernel application

4.1. 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.

4.2. 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!

4.3. 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.

4.3.1. Classes and interfaces for the kernel

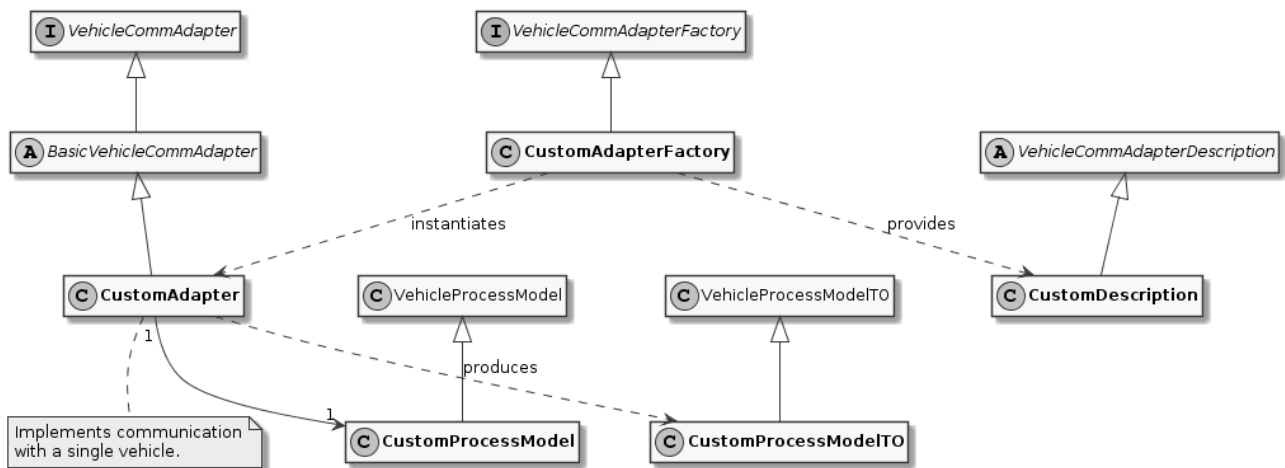


Figure 7. Classes of a comm adapter implementation (kernel side)

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.

4.3.2. Classes and interfaces for the control center application

For the kernel control center application, the following interfaces and classes are the most important ones:

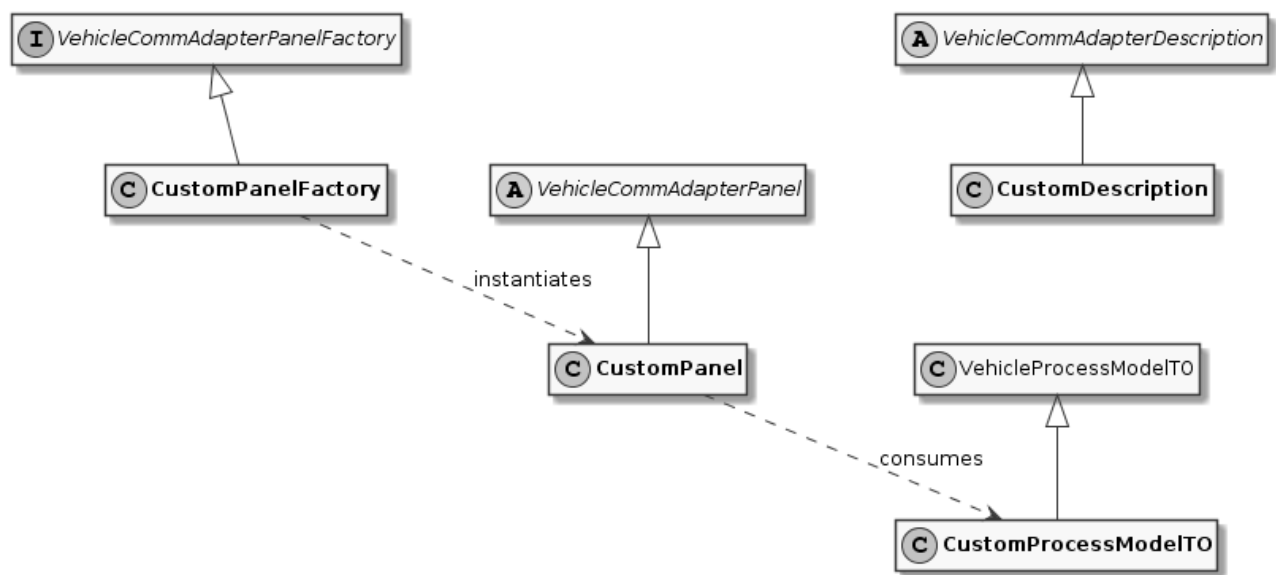


Figure 8. Classes of a comm adapter implementation (kernel control center side)

- `VehicleCommAdapterPanel` instances may be created by a `VehicleCommAdapterFactory` 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.

4.3.3. 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.
 - 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 `VehicleCommAdapterPanelFactory`s 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.)

4.3.4. 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 necessary 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.

4.4. 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.

4.5. 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 `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:

```
Vehicle vehicle = getSomeVehicle();
String value = vehicle.getProperty("someKey");
processPropertyValue(value);
```

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:

```

TransportOrder transportOrder = getSomeTransportOrder();
String value = transportOrder.getProperty("someKey");
processPropertyValue(value);

```

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

4.6. Developing peripheral drivers

In addition to vehicle drivers, openTCS supports the integration of custom peripheral drivers that implement peripheral-specific communication protocols and thus mediate between the kernel and the peripheral device. In openTCS, a peripheral device is a device a vehicle may interact with along its route, e.g. an elevator or a fire door. Due to its function, a peripheral driver is also called a peripheral communication adapter. The following sections describe which requirements must be met by a driver and which steps are necessary to create and use it.

4.6.1. Classes and interfaces for the kernel

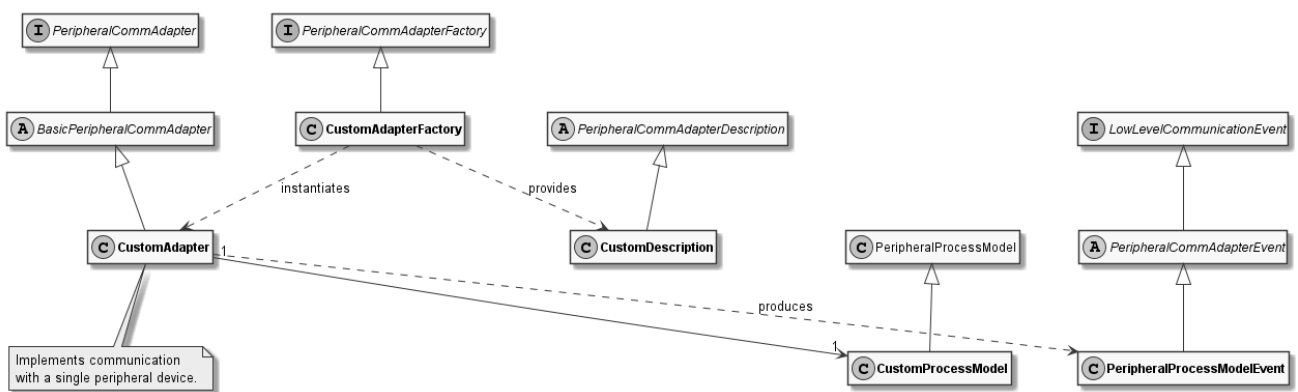


Figure 9. Classes of a peripheral comm adapter implementation (kernel side)

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

- `PeripheralCommAdapter` declares methods that every comm adapter must implement. These methods are called by components within the kernel, for instance to tell a peripheral device that it is supposed to perform an operation, e.g. for an elevator to move to a specific floor. Classes implementing this interface are expected to perform the actual communication with a peripheral device, e.g. via TCP, UDP or some field bus.
- `BasicPeripheralCommAdapter` is the recommended base class for implementing a `PeripheralCommAdapter`. It primarily provides some basic event dispatching with regards to the `PeripheralProcessModel`.
- `PeripheralCommAdapterFactory` describes a factory for `PeripheralCommAdapter` instances. The kernel instantiates and uses one such factory per peripheral driver to create instances of the respective `PeripheralCommAdapter` implementation on demand.
- A single `PeripheralProcessModel` instance should be provided by every `PeripheralCommAdapter`

instance in which it keeps the relevant state of both the peripheral device 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 peripheral device's current state in the model when it receives that information to allow the kernel and GUI frontends to use it. `PeripheralProcessModel` 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 `PeripheralProcessModel` to have the comm adapter and other components exchange more than the default set of attributes.

4.6.2. Classes and interfaces for the control center application

For the kernel control center application, the following classes and interfaces are the most important:

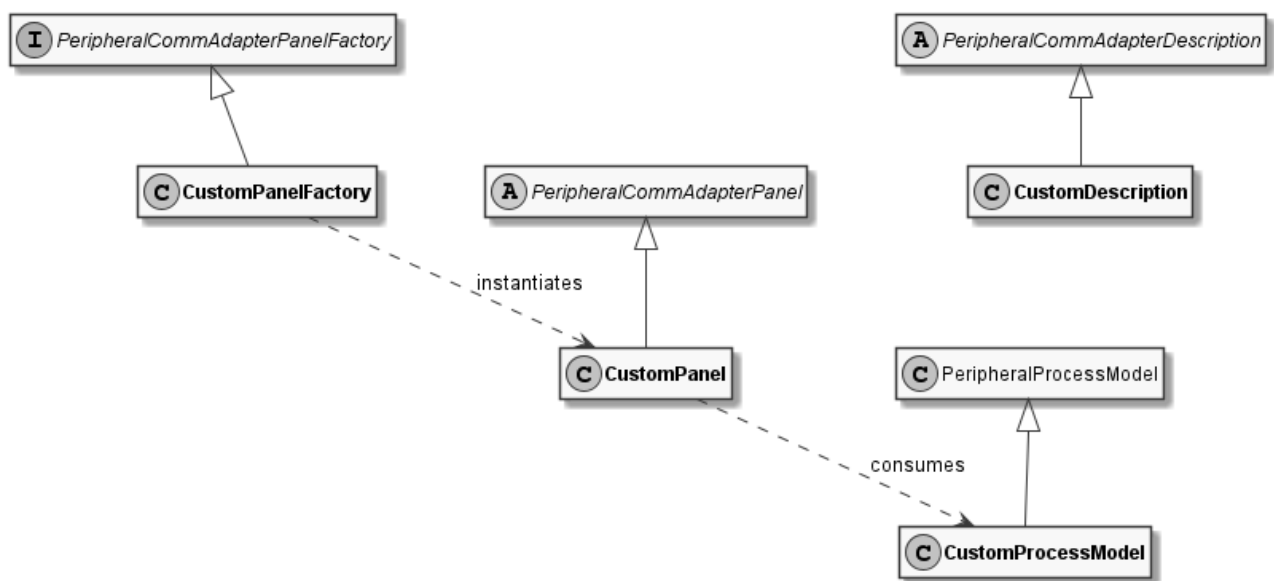


Figure 10. Classes of a peripheral comm adapter implementation (kernel control center side)

- `PeripheralCommAdapterPanel` instances may be created by a `PeripheralCommAdapterFactory` e.g. to display information about the associated peripheral device or send low-level messages to it.
- `PeripheralProcessModel` instances are used in a comm adapter's `PeripheralCommAdapterPanel` instances for updating their contents. In contrast to a `VehicleCommAdapter`, there is no class such as `PeripheralProcessModelTO`, since `PeripheralProcessModel` already implements the `Serializable` interface. Developers should keep that in mind when creating driver-specific subclasses of `PeripheralProcessModel`.
- Instances of `PeripheralCommAdapterDescription` 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 `PeripheralService.attachCommAdapter()`.
- `PeripheralAdapterCommand` instances can be sent from a panel to a `PeripheralCommAdapter` instance via `PeripheralService.sendCommAdapterCommand()`. They are supposed to be executed by the comm adapter and can be used to execute arbitrary methods, e.g. methods of the `PeripheralCommAdapter` itself, or update contents of the comm adapter's `PeripheralProcessModel`.

Note that `PeripheralAdapterCommand` instances can only be sent to and processed by the kernel application if they are serializable and present in the kernel application's classpath.

4.6.3. Steps to create a new peripheral driver

1. Create an implementation of `PeripheralCommAdapter`:
 - a. Subclass `BasicPeripheralCommAdapter` unless you have a reason not to.
 - b. Implement the abstract methods of `BasicPeripheralCommAdapter` in the derived class to realize communication with the peripheral device.
 - c. In situations in which the state of the peripheral device 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 and publish a corresponding `PeripheralProcessModelEvent` via the kernel's `ApplicationEventBus`. Most importantly, call `PeripheralJobCallback.peripheralJobFinished()` on the callback instance provided with `PeripheralCommAdapter.process()` when the controlled peripheral device's reported state indicates that it has finished a job.
2. Create an implementation of `PeripheralCommAdapterFactory` that provides instances of your `PeripheralCommAdapter` for given `Location` objects.
3. Optional: Create any number of implementations of `PeripheralCommAdapterPanel` 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 `PeripheralCommAdapterPanelFactory`s implementation.

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

4.6.4. Registering a peripheral driver with the kernel

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

```
peripheralCommAdaptersBinder().addBinding().to  
(LoopbackPeripheralCommAdapterFactory.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.commadapter.peripheral.loopback.LoopbackPeripheralKernelModule
```



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 necessary 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.

4.7. 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 periodic 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 functionality, 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`:

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

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

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

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

```
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 Operations Desk, and it is not required there (for avoiding concurrency issues in the kernel).

Chapter 5. Customizing and extending the control center application

5.1. 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.

5.2. 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 `VehicleCommAdapterFactory`. 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() {
    commAdapterFactoryBinder().addBinding().to(
        LoopbackCommAdapterFactory.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 necessary 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 6. Customizing and extending the Model Editor and the Operations Desk applications



The process of customizing and extending the Model Editor and the Operations Desk is basically identical for both applications. For the sake of simplicity, this section describes the process using the Operations Desk application as an example. Where necessary, differences between the two applications are explicitly mentioned.

6.1. Guice modules

Analogous to the kernel application, the Operations Desk 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 Operations Desk 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. How to add import/export functionality for plant model data

It is not uncommon for plant model data to be shared with other systems. For instance, it is possible that the navigation computer of a vehicle already has detailed information about the plant environment such as important positions and connections between them. Entering this data manually via the Model Editor application would be a tiresome and error-prone work. To help relieve users from such work, the Model Editor supports integrating components for importing plant model data from external resources — e.g. files in a foreign file format — or exporting plant

model data to such resources.

To integrate a model import component, do the following:

1. Create a class that implements the interface `PlantModelImporter` and implement the methods declared by it:
 - a. In `importPlantModel()`, implement what is necessary to produce an instance of `PlantModelCreationTO` that describes the plant model. The actual steps necessary here may vary depending on the source and the type of model data to be imported. In most cases, however, the process will probably look like the following:
 - i. Show a dialog for the user to select the file to be imported.
 - ii. Parse the content of the selected file according to its specific file format.
 - iii. Convert the parsed model data to instances of `PointCreationTO`, `PathCreationTO` etc., and fill a newly created `PlantModelCreationTO` instance with them.
 - iv. Return the `PlantModelCreationTO` instance.
 - b. `getDescription()` should return a short description for your importer, for instance "Import from XYZ course data file". An entry with this text will be shown in the Model Editor's **[File › Import plant model]** submenu, and clicking on this entry will result in your `importPlantModel()` implementation to be called.
2. Create a Guice module for registering your `PlantModelImporter` with openTCS by subclassing `PlantOverviewInjectionModule`. Implement the `configure()` method and add a binding to your `PlantModelImporter` using `plantModelImporterBinder()`.
3. Build and package the `PlantModelImporter` and Guice module into a JAR file.
4. 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. (See [How to create a plugin panel for the Operations Desk client](#) for an example.)
5. Place the JAR file in the Model Editor application's class path (subdirectory `lib/openTCS-extensions/` of the application's installation directory) and start the application.

To integrate a model *export* component, you follow the same steps, but implement the interface `PlantModelExporter` instead of `PlantModelImporter`.

6.3. How to create a plugin panel for the Operations Desk client

The Operations Desk 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`.



The `PluggablePanelFactory.providesPanel(state)` method is used to determine which `Kernel.State` a factory provides panels for. For plugin panels that are intended to be used with the Model Editor application only, this method should return `true` for the kernel state `Kernel.State.MODELLING`. For plugin panels that are intended to be used with the Operations Desk application only, this method should return `true` for the kernel state `Kernel.State.OPERATING`. Otherwise the plugin panels won't be shown in the respective application.

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 Operations Desk application's class path (subdirectory `lib/openTCS-extensions/` of the application's installation directory) and start the application.

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

Locations and vehicles are visualized in the Operations Desk client using configurable themes. To customize the appearance of locations and vehicles, new theme implementations can be created and integrated into the Operations Desk 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 Operations Desk 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 Operations Desk application's configuration file.

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

Chapter 7. Application configuration

As described in the openTCS User's Guide, the openTCS Kernel, Kernel Control Center, Model Editor and Operations Desk applications read their configurations from properties files. This functionality is provided by the [gestalt](#) library.

7.1. Supplementing configuration sources using gestalt

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 [org.opentcs.configuration.gestalt.SupplementaryConfigSource](#). This interface is part of the [opentcs-impl-configuration-gestalt](#) artifact, which must be on your project's classpath.
 - b. A plain text file named [META-INF/services/org.opentcs.configuration.gestalt.SupplementaryConfigSource](#). 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 8. 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.

8.1. 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 Model Editor or Operations Desk 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.

8.2. 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  
        ...
```

8.3. 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.

8.4. 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.